Représentation des nombres

Alain MÉRIGOT

Université Paris Saclay

Plan

1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. Nombres entiers signés
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
   - Passage d'un codage en virgule fixe à un codage flottant
6. Codage des caractères

Bits et mots

Élement d'information : **bit** (*binary digit*)
Deux valeurs : `vrai` et `faux`, ou 0 et 1.

Bits regroupés en *mots* de n bits, généralement numérotés de $n-1$ à 0

A mot formé de n bits a_{n-1} a_{n-2} $\ldots$ a_1 a_0

bit de poids fort
most significant bit
MSB

bit de poids faible
least significant bit
LSB

Mot de n bits : n éléments pouvant prendre 2 valeurs $\Longrightarrow$ 2^n configurations différentes.

Problème du codage : définir une application de préférence bijective associant des mots à des élements d'autres ensembles : entiers naturels $\mathbb{N}$, entiers relatifs $\mathbb{Z}$ ou ou nombres réels $\mathbb{R}$.

Plan

1. L'élément d'information : le bit
2. **Nombres entiers non signés**
3. Codage hexadécimal
4. Nombres entiers signés
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
   - Passage d'un codage en virgule fixe à un codage flottant
6. Codage des caractères

Codage des entiers naturels

Soit un mot A codé sur n bits : $A = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$
On peut associer au mot A, sa valeur en base 2 $A' \in \mathbb{N}$ telle que que chaque bit a_i de A corresponde à la puissance 2^i dans la décomposition en base 2 de A'

Un mot $A = (a_{n-1}, a_{n-2}, \ldots, a_0)$ est le **code binaire naturel** (CBN) sur n bits du nombre A' si

$$\begin{aligned} A' &= 2^{n-1} \times a_{n-1} + 2^{n-2} \times a_{n-2} + \cdots + 2 \times a_1 + a_0 \\ &= \sum_{i=0}^{n-1} 2^i \times a_i \end{aligned} \tag{1}$$

Nous identifierons en général A et A'.

$$A \text{ sur } n \text{ bits} \implies \underbrace{0}_{\text{code (000...0)}} \leq A \leq \underbrace{2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = 2^n - 1}_{\text{code (111...1)}}$$

$$n = 8 : 0 \leq A \leq 2^8 - 1 = 255$$

$$n = 16 : 0 \leq A \leq 2^{16} - 1 = 65\,535$$

$$n = 32 : 0 \leq A \leq 2^{32} - 1 = 4\,294\,967\,295$$

Codage des entiers naturels (cont.)

Calcul de la valeur numérique entière associée à un mot de n bits
$\Longrightarrow$ utilisation directe de la formule (1).

Calcul du CBN d'un nombre entier naturel : différentes méthodes

Exemple : reste des divisions successives de A par 2

```
    ▷ Soit A le nombre à coder en binaire
1   pour i ← 0 à n − 1 faire
2       bit[i] ← A % 2;          ▷ Le reste de la division par 2 du nombre
3       A ← A ÷ 2;
4   fin pour
    ▷ Le tableau bit[n − 1]...bit[0] contient le code binaire naturel de A
```

Exemple : code de 57 sur 6 bits

$57 \div 2 = 28$ reste **1** (bit 0)	$7 \div 2 = 3$ reste **1** (bit 3)
$28 \div 2 = 14$ reste **0** (bit 1)	$3 \div 2 = 1$ reste **1** (bit 4)
$14 \div 2 = 7$ reste **0** (bit 2)	$1 \div 2 = 0$ reste **1** (bit 5)

D'où le code 111001 (car 57=32+16+8+0×4+0×2+1)

Plan

1. L'élément d'information : le bit
2. Nombres entiers non signés
3. **Codage hexadécimal**
4. Nombres entiers signés
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
   - Passage d'un codage en virgule fixe à un codage flottant
6. Codage des caractères

Code hexadécimal

binaire : code souvent long et générateur d'erreurs

$\Longrightarrow$ codage hexadécimal (base 16)

Codage d'un ensemble de 4 bits[1] par un digit d'un alphabet formé par les chiffres décimaux 0 . . . 9 et les lettres A…F (ou a…f)

Déc.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin.	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hexa.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

En C, utilisation du préfixe `0x`

Exemple : $254 = 1111\ 1110 = 15 \times 16^1 + 14 \times 16^0 \equiv$ `0XFE` (ou `0xfe`)

[1] souvent appelé *nibble* en anglais

Plan

1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. **Nombres entiers signés**
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
   - Passage d'un codage en virgule fixe à un codage flottant
6. Codage des caractères

Plan

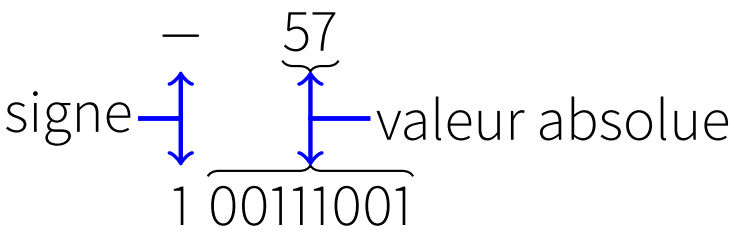
1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. **Nombres entiers signés**
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
6. Codage des caractères

Nombres entiers signés

signe-valeur absolue

Pas de représentation naturelle pour les nombres signés ($\in \mathbb{Z}$).

Codage « signe-valeur absolue » (SVA)



Un bit (généralement le MSB) indique le signe (0 pour +[2] et 1 pour −), les autres la valeur absolue

Si A codé en SVA sur n bits, $|A|$ est codé sur $n-1$ bits et donc $|A| \leq 2^{n-1}-1$
D'ou $-(2^{n-1}-1) \leq A \leq 2^{n-1}-1$

Inconvénients :

- l'ordre des codes ne correspond pas toujours à celui des nombres :
  - si $A \geq 0$, $\text{code}(A) < \text{code}(-A)$ (inévitable si + codé par 0)
  - si $A, B \geq 0, A > B \implies \text{code}(A) > \text{code}(B)$,
    si $A, B < 0, A > B$ $(|A| < |B|) \implies \text{code}(A) < \text{code}(B)$
- deux représentations de zéro : $+0(0\,00\ldots0)$ et $-0(1\,00\ldots0)$

[2] pour que le code sur n bits de A (non signé) soit identique à celui de $+A$ (signé)

Plan

1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. **Nombres entiers signés**
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
6. Codage des caractères

On appelle **complément à deux sur n bits** de A, le nombre $C_2(A) = 2^n - A$.

On appelle **code en complément à deux sur n bits** (CC_2) d'un nombre A :

si $0 \leq A \leq 2^{n-1} - 1$ **:** code binaire naturel sur n bits de A

si $-2^{n-1} \leq A < 0$ **:** code binaire naturel sur n bits de $C_2(|A|)$

Remarques :

1. Si $A < 0$, $C_2(|A|) = 2^n - |A| = 2^n + A$.
   Le code complément à 2 revient à ajouter 2^n aux nombres négatifs pour les rendre positifs et les coder en binaire naturel.
   Ceci permet de simplifier *considérablement* les opérations arithmétiques.
2. Si $A \geq 0$, comme $A < 2^{n-1}$, le MSB a_{n-1} sera à zéro
   Si $A < 0$, Comme $|A| \leq 2^{n-1}$, alors $C_2(A) = 2^n - |A| \geq 2^{n-1}$ et le MSB a_{n-1} du CBN de $C_2(A)$ est à 1.
   Le MSB a_{n-1} du CC_2 d'un nombre A est donc à 0 si $A \geq 0$ et à 1 si $A < 0$ et joue le rôle d'un **bit de signe**.

Quelques propriétés du complément à 2

- $C_2(C_2(A)) = 2^n - (2^n - A) = A$
- $A + C_2(A) = 2^n$
- $C_2(A) = \overline{A} + 1$
  En effet $A + \overline{A} = \sum_{i=0}^{n-1} a_i \times 2^i + \sum_{i=0}^{n-1} \overline{a_i} \times 2^i$
  $= \sum_{i=0}^{n-1} (a_i + \overline{a_i}) \times 2^i = \sum_{i=0}^{n-1} 2^i = 2^n - 1$
  $\Longrightarrow \overline{A} + 1 = 2^n - A = C_2(A)$
- une seule représentation de 0 :
  $-0 = C_2(0) = 2^n - 0 = 0$ (sur n bits)

A est codable en CC_2 sur n bits $\equiv -2^{n-1} \leq A \leq 2^{n-1} - 1$
Sur 8 bits : $-128 \leq A \leq 127$
Sur 16 bits : $-32768 \leq A \leq 32767$
Sur 32 bits : $-2147483648 \leq A \leq 2147483647$

Pour retrouver la valeur d'un mot $A = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$ codé en complément à deux :

$$\begin{array}{lllll} A & = & \mathrm{CBN}(A) & & \text{si } A \geq 0 \quad (a_{n-1} = 0) \\ & = & 0 \times 2^{n-1} & + \sum_{i=0}^{n-2} a_i \times 2^i & \end{array}$$

$$\begin{array}{llllll} A & = & -2^n & + \mathrm{CBN}(A) & & \text{si } A < 0 \quad (a_{n-1} = 1) \\ & = & -2^n & + 1 \times 2^{n-1} & + \sum_{i=0}^{n-2} a_i \times 2^i & \\ & = & & 1 \times (-2^{n-1}) & + \sum_{i=0}^{n-2} a_i \times 2^i & \end{array}$$

La valeur numérique d'un nombre $A = (a_{n-1}, \ldots, a_0)$ codé en CC_2 sur n bits est :

$$A = -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} 2^i \times a_i \qquad (2)$$

Tout se passe comme si le bit de poids fort correspondait à un poids -2^{n-1}.

Plan

1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. **Nombres entiers signés**
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
6. Codage des caractères

Nombres entiers signés (cont.)

codes par excès

Le code complément à deux respecte l'ordre relatif des nombres de même signe,
mais les nombres positifs ont tous un code inférieur aux nombres négatifs.

Dans certains il est important de respecter cet ordre : **codes par excès**

On ajoute un décalage fixe k (excès) à *tous* les nombres à coder
et on code en binaire naturel $A + k$

Sur n bits, il faut $0 \leq A + k \leq 2^n - 1$
Un code par excès k peut coder $-k \leq A \leq 2^n - k - 1$

L'ordre relatif des nombres est parfaitement respecté, mais

- le code d'un nombre positif est différent de son code en binaire naturel
- les opérations arithmétiques ($+$, $-$, etc) sont plus complexes, car il faut retrancher l'excès avant d'effectuer le calcul.

Nombres entiers signés (cont.)

récapitulatif

Codage	Binaire naturel	Signe-Valeur abs.	Complém. à 2	Excès 8
0000	0	+0	0	−8
0001	1	+1	+1	−7
0010	2	+2	+2	−6
0011	3	+3	+3	−5
0100	4	+4	+4	−4
0101	5	+5	+5	−3
0110	6	+6	+6	−2
0111	7	+7	+7	−1
1000	8	−0	−8	0
1001	9	−1	−7	+1
1010	10	−2	−6	+2
1011	11	−3	−5	+3
1100	12	−4	−4	+4
1101	13	−5	−3	+5
1110	14	−6	−2	+6
1111	15	−7	−1	+7

croissant

croissant

décroissant

croissant

croissant

Nombres entiers signés (cont.)

récapitulatif

taille	nom	signé/non signé	en C	`stdint.h`
8 bits	octet (*byte*)	signé	`char`	`int8_t`
		non signé	`unsigned char`	`uint8_t`
16 bits	demi mot (*half*)	signé	`short`	`int16_t`
		non signé	`unsigned short`	`uint16_t`
32 bits	mot (*word*)	signé	`int` (ou `long`)	`int32_t`
		non signé	`unsigned` (`int`)	`uint32_t`
64 bits	double mot	signé	`long long`	`int64_t`
		non signé	`unsigned long long`	`uint64_t`

NB : le standard C n'impose pas de taille pour les types entiers.
Il impose juste `sizeof(char)`$\leq$`sizeof(short)`$\leq$`sizeof(int)`$\leq$`sizeof(long)`$\leq$`sizeof(long long)`.
Sur certains compilateurs `sizeof(int)`=16 ou `sizeof(long)`=64, etc.

En C99, introduction de `<stdint.h>` qui permet d'avoir des types de taille définie.

Plan

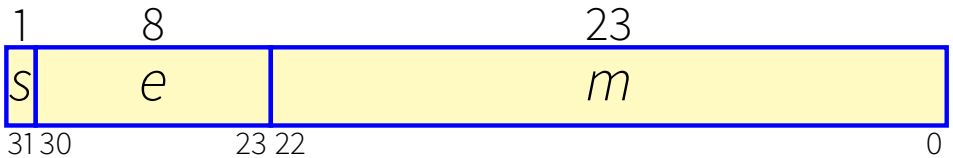
1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. Nombres entiers signés
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. **Nombres flottants**
   - Passage d'un codage en virgule fixe à un codage flottant
6. Codage des caractères

Nombres flottants

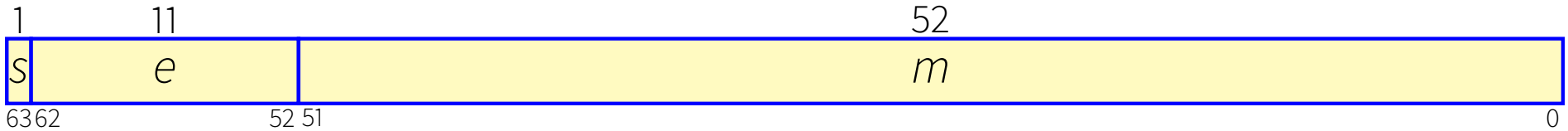
Pour de nombreuses applications, nécessité d'avoir 1/ des nombres fractionnaires et 2/ de dynamique élevée : *nombres à virgule flottante* (ou *nombres flottants*).
Nombre flottant $N \equiv$ triplet (s, m, e) *signe*, *mantisse*, *exposant* tel que
$N = (-1)^s \times m \times b^e$
Généralement $b = 2$

Standardisation **IEEE-754** : spécifie les modes de codage de s, m et e pour divers formats.

simple précision (32 bits) (`float` en C) (s : 1 bits, e : 8 bits, m : 23 bits)



double précision (64 bits) (`double` en C) (s : 1 bits, e : 11 bits, m : 52 bits)



Il existe également précision étendue (80 bits), quadruple précision (128 bits), octuple précision (256 bits), demi précision (16 bits), etc.

Pour assurer représentation unique, mantisse normalisée : $m \in [1, 2[$
Représente un nombre $1.000\ldots0 \leq M \leq 1.111\ldots1$

Les bits a_i correspondent à des puissances négatives de 2.
Comme le bits de poids fort est toujours à 1, il n'est pas codé : « bit caché »

$$\begin{aligned} \text{Si } M &= 1.a_{n-1}a_{n-2}\ldots a_1a_0 \\ M &= 1 + a_{n-1} \times 2^{-1} + a_{n-2} \times 2^{-2} + \cdots + a_1 \times 2^{1-n} + a_0 \times 2^{-n} \\ &= 1 + \sum_{i=n-1}^{0} a_i \times 2^{i-n} \end{aligned}$$

Exemple : $1.101 = 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 1.625$

Le poids de l'unité en dernière position (ULP *unit in the last place*) donne la **précision** du code.
En simple précision, $2^{-23} \approx 1.2 \times 10^{-7}$ (6 à 7 décimales précises)

Les exposants sont codés *par excès* pour permettre une comparaison simple.

En simple précision ($e = 8$ bits) : excès de 127
En double précision ($e = 11$ bits) : excès de 1023

Pour un code IEEE 754 dont l'exposant est sur l bits, l'excès sera se $2^{l-1} - 1$

En simple précision, ajouter 127 pour coder un exposant
retrancher 127 pour obtenir l'exposant depuis son code

Le code des exposants peut varier de 1 (2^{-126}) à 254 (2^{127}).

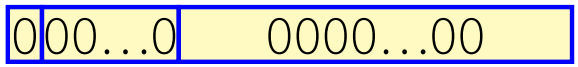
Les exposants déterminent la **dynamique** des nombres.

Nombres flottants (cont.)

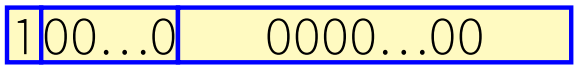
Les exposants extrèmes (0 et 255) codent des cas particuliers.

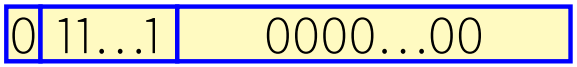
Pour un nombre flottant simple précision N codé par le triplet (s, e, m)

	$e = 0$	$1 \leq e \leq 254$	$e = 255$
$m = 0$	$N = (-1)^s \times 0.0$	$N = (-1)^s \times 1.0 \times 2^{e-127}$	$N = (-1)^s \times \infty$
$m \neq 0$	$N = (-1)^s \times 0.m \times 2^{-126}$	$N = (-1)^s \times 1.m \times 2^{e-127}$	**NaN** (Not A Number)

zéro est représenté par 

(comme les codes entiers)

Il existe un code pour -0.0 : 

Codage de $\pm\infty$ (`inf` en C) 

($e = 255, m = 0$)

$\pm$zéro correspond à des nombres trop petits pour être codables en IEEE 754, $\pm\infty$ à des nombres trop grands.

Les codes $e = 255$ et $m \neq 0$ sont dit *not a number* (**nan** en C)
$\rightarrow$ détection des erreurs (débordements numériques, $0 \times \infty$, $\div 0$)
ou codage d'informations de manière non spécifiée par la norme.
Toute opération dont un argument est à **nan** génère un **nan** en sortie.

Nombres flottants (cont.)

Les codes d'exposant nul sont dits **dénormalisés** (*subnormals*)

Le *bit caché* est à 0 (au lieu de 1)
Permet un accroissement de la *dynamique* (vers les exposants < 0) en réduisant la *précision*

Exemple : $m = \underbrace{0.0000010111001\ldots010}$

dynamique accrue de 2^{-6} ↑ ↑ mais précision de 18 bits (contre 23+1)

Plus grands et plus petits nombres codables en simple et en double précision IEEE 754.

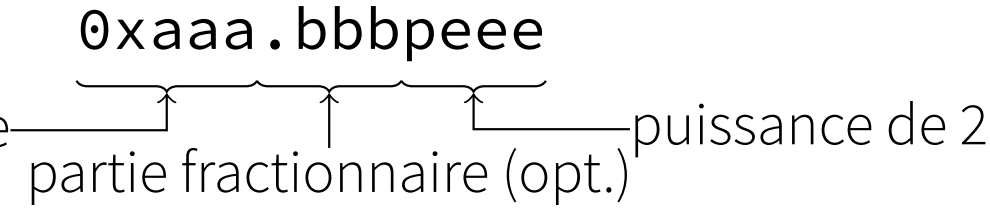
	Plus petit nombre normalisé	Plus petit nombre dénormalisé	Plus grand nombre
Simple précision	1.175494e-38	1.401298e-45	3.40282e+38
Double précision	2.225074e-308	4.940656e-324	1.79769e+308

Nombres flottants (cont.)

Depuis le standard C99, possibilité d'entrer et d'afficher en C les nombres flottants dans un format hexadécimal.

- prise en compte du codage d'un nombre
- évite les erreurs d'arrondi dans la conversion décimale ⇔ flottant

Pour coder le nombre aaa.bbb 2^{eee} (en hexadécimal) :



Spécifieur `%a` de `printf()` pour afficher les flottants hexadécimaux.

```
float  fmax=0x1.fffffep127; // 1.111...11 2^127
double dmax=0x1.fffffffffffffp1023; // 1.111...11 2^1023
printf ("Plus grand float : %g (code:%a)\n",fmax,fmax);
printf ("Plus grand double : %g (code:%a)\n",dmax,dmax);
// affiche :
// Plus grand float : 3.40282e+38 (code:0x1.fffffep+127)
// Plus grand double : 1.79769e+308 (code:0x1.fffffffffffffp+1023)
```

Plan

1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. Nombres entiers signés
5. **Nombres flottants**
   - Passage d'un codage en virgule fixe à un codage flottant
6. Codage des caractères

Le codage en *virgule fixe* consiste à coder un nombre fractionaire avec des puissances positives et négatives de 2.
Historiquement utilisé (et encore utile dans certaines réalisation matérielles).
Intermédiaire utile pour convertir un nombre décimal en flottant.

Conversion décimal $\rightarrow$ flottant

1. coder la valeur absolue du nombre en binaire naturel avec une virgule fixe décomposition en puissances positives (pour la partie entière) et négatives (pour la partie fractionnaire) de 2.
2. décaler ce code de k vers la droite ou la gauche ($k < 0$) pour avoir un "1" à gauche la virgule
   Pour garder la valeur numérique, multiplier par 2^k
3. réaliser le codage suivant la norme IEEE 754

Exemple : $N = -1.275 \times 10^1$ à coder en flottant IEEE simple précision.
Passage en virgule fixe en l'écrivant $N = -12.75$.

1. codage en virgule fixe en base 2 de $|N|$:

$$\begin{aligned} |-12.75| &= 12.75 \\ &= 8 + 4 + 0.5 + 0.25 \\ &= 2^3 + 2^2 + 2^{-1} + 2^{-2} \\ &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &\quad +1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + \cdots \\ &= 1100.110\ldots \end{aligned}$$

2. normalisation du nombre entre 1.0 et 2.0
   décalage de 3 positions vers la droite pour avoir un nombre normalisé :
   $12.75 = 1.10011 \times 2^{+3}$
3. codage en flottant IEEE :
   $m = (1.)100110\ldots$
   $e + 127 = 130 = 10000010$
   $s = 1$
   d'où le code :

1	10000010	10011000000000000000000

Nombres flottants (cont.)

Dans le cas général, conversion d'un nombre fractionnaire décimal → binaire :

▷ *Soit N le nombre à coder en binaire. On suppose* $N < 1$

1 **pour** $i \leftarrow n-1$ **à** 0 **faire**
2 $N \leftarrow N \times 2$;
3 **si** $N \geq 1$ **alors**
4 bit[i] $\leftarrow 1$;
5 $N \leftarrow N - 1$;
6 **sinon**
7 bit[i] $\leftarrow 0$;
8 **fin si**
9 **fin pour**

▷ *Le code de N est 0.bit[n − 1]...bit[0]*

Exemple : codage en binaire fractionnaire $0.1_d = 0 \times 10^0 + 1 \times 10^{-1}$.

i	N	$2 \times N$	$N \geq 1$?	$N-1$	bit[i]
$n-1$	0.1_d	0.2_d	non		0
$n-2$	0.2_d	0.4_d	non		0
$n-3$	0.4_d	0.8_d	non		0
$n-4$	0.8_d	1.6_d	oui	0.6_d	1
$n-5$	0.6_d	1.2_d	oui	0.2_d	1
$n-6$	0.2_d	0.4_d	non		0
etc.					

L'étape $n-6$ étant identique à $n-2$, itération infinie.

$0.1_d = 0.0\underbrace{0011}\underbrace{0011}\underbrace{0011}\ldots$

Pas de représentation binaire exacte de 0.1_d.

Plan

1. L'élément d'information : le bit
2. Nombres entiers non signés
3. Codage hexadécimal
4. Nombres entiers signés
   - Codage Signe–Valeur absolue
   - Code complément à 2
   - Codes par excès
5. Nombres flottants
   - Passage d'un codage en virgule fixe à un codage flottant
6. **Codage des caractères**

Codage des caractères

Codage normalisé des caractères : ASCII 7 bits (1963) (ANSI X3.4 1986)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	‖	}	~	DEL

Les codes `0x00` à `0x1F` sont des caractères spéciaux.
Pour la plupart inutiles actuellement sauf NUL (`'\0'`) BEL (`'\g'`), BS (*backspace*, `'\b'`), HT (tabulation, `'\t'`), LF (*line feed*, `'\n'`) et ESC (*escape*, `'\e'`).

Étendu par l’ISO (ISO 8859) pour caractères accentués, symboles mathématiques, etc.

Actuellement, norme *unicode* pour la plupart des caractères internationaux.
variante **utf-8** (*Universal Character Set Transformation Format 8 bits*) utilise

- un octet pour les 128 premiers caractères (ASCII)
- 2 à 4 octets pour extension aux caractères internationaux.

Addition entière

Alain MÉRIGOT

Université Paris sud

Plan

1. **Additionneurs 1 bit**
2. **Additionneur n bits à propagation de retenue**
   - Addition non signée
3. **Addition/soustraction de nombre signés**
   - Additionneur/soustracteur
   - Validité du résultat
4. **Additionneur à sauvegarde de retenue (*carry save adder*)**
5. **Additionneur à retenue anticipée**

Additionneurs 1 bit

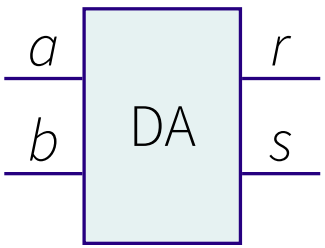
demi-additionneur

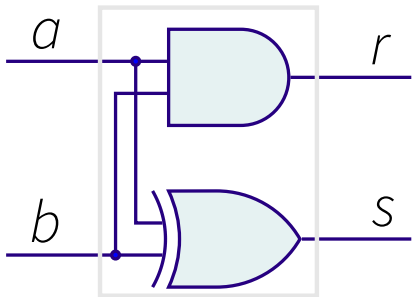
Addition de deux bits : **demi-additionneur**
entrées : opérandes a et b,
sorties : somme des deux opérandes sur 2 bits s et r
Le bit de poids fort r du résultat est appelé **retenue** (*carry*)
Le bit de poids faible s est appelé **somme**

a	b	$a+b$	r	s
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

$$a + b = 2^1 \times r + 2^0 \times s$$

$$s = a \oplus b$$
$$r = a \,\&\, b$$





Additionneurs 1 bit (cont.)

additionneur complet

Addition de mots de n bits : nécessité de prendre en compte la retenue :
additionneur complet
3 entrées : opérandes a b et c, 2 sorties : s et r

a	b	c	$a+b+c$	r	s
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

$$a+b+c = 2^1 \times r + 2^0 \times s$$

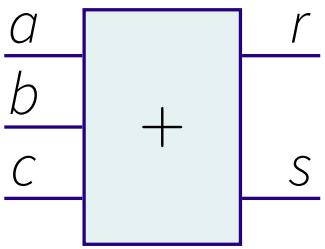
$$s = a \oplus b \oplus c$$
$$r = a \,\&\, b + a \,\&\, c + b \,\&\, c$$

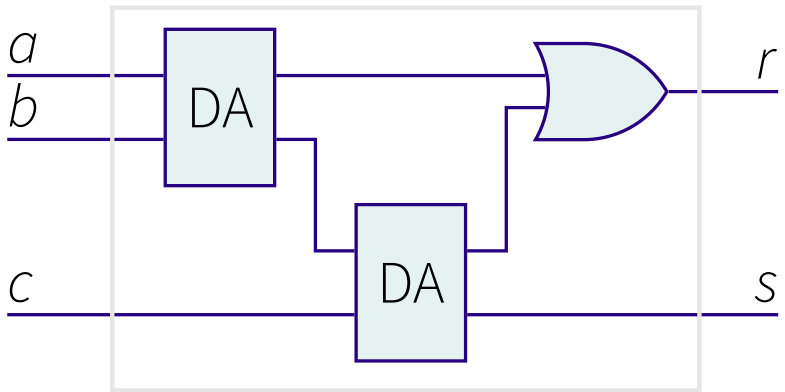
Additionneurs 1 bit (cont.)

additionneur complet

$$s = \underbrace{a \oplus b}_{s_{a+b}} \oplus c$$

$$\begin{aligned} r &= a \,\&\, b + a \,\&\, c + b \,\&\, c \\ &= a \,\&\, b + a \,\&\, (\overline{b} + b) \,\&\, c + (\overline{a} + a) \,\&\, b \,\&\, c \\ &= a \,\&\, b + a \,\&\, \overline{b} \,\&\, c + \cancel{a \,\&\, b \,\&\, c} + \overline{a} \,\&\, b \,\&\, c + \cancel{a \,\&\, b \,\&\, c} \\ &= \underbrace{a \,\&\, b}_{r_{a+b}} + (\underbrace{a \oplus b}_{s_{a+b}}) \,\&\, c \end{aligned}$$





Plan

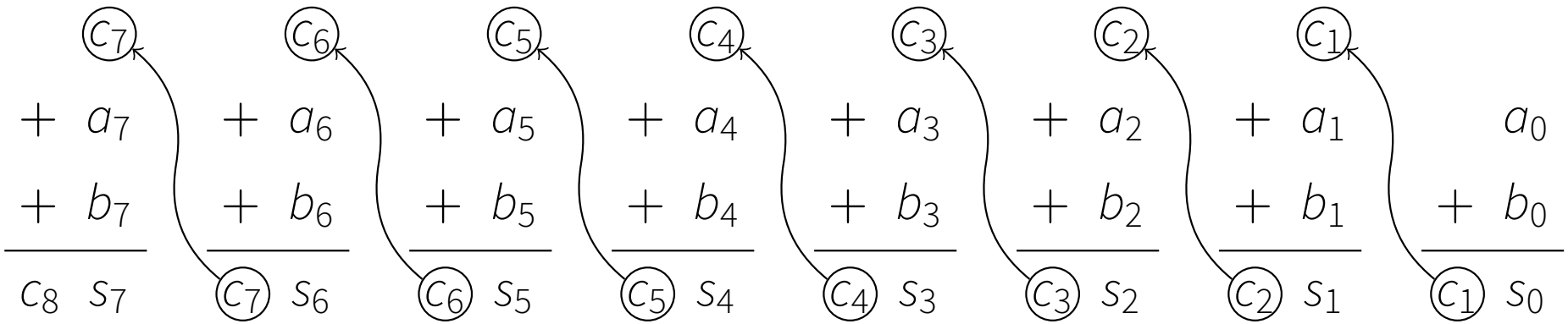
1. Additionneurs 1 bit
2. **Additionneur n bits à propagation de retenue**
   - Addition non signée
3. Addition/soustraction de nombre signés
   - Additionneur/soustracteur
   - Validité du résultat
4. Additionneur à sauvegarde de retenue (*carry save adder*)
5. Additionneur à retenue anticipée

Plan

1. Additionneurs 1 bit
2. **Additionneur n bits à propagation de retenue**
   - Addition non signée
3. Addition/soustraction de nombre signés
4. Additionneur à sauvegarde de retenue (*carry save adder*)
5. Additionneur à retenue anticipée

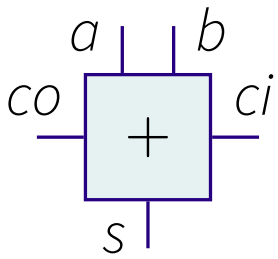
Additionneur à propagation de retenue

Addition de 2 nombres entiers non signés $A = (a_{n-1}, a_{n-2}, \ldots, a_0)$ et $B = (b_{n-1}, b_{n-2}, \ldots, b_0)$.
Addition de a_0 et $b_0 \rightarrow$ somme de poids 2^0 (s_0) et retenue de poids 2^1 (c_1).
c_1 sera pris en compte avec les bits de même poids a_1 et b_1.

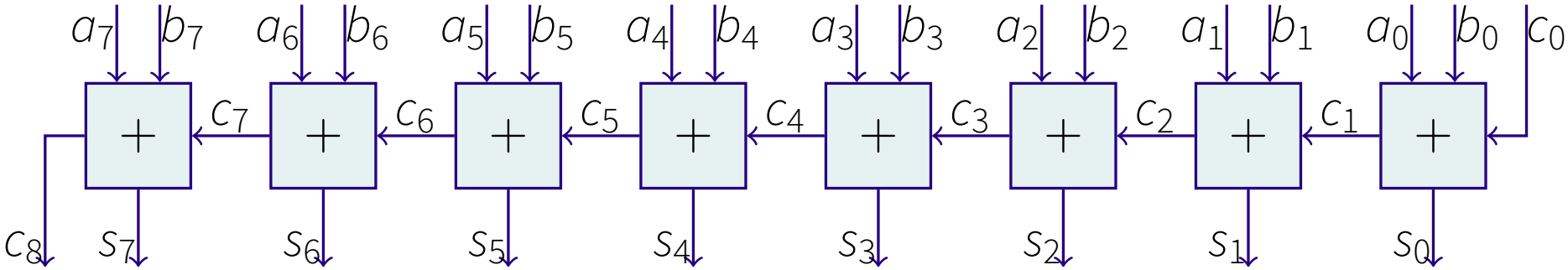


Addition à propagation de retenue (*carry propagate adder*) :
la retenue de c_{i+1} de l'étage i est réinjectée à l'entrée c de l'addition de l'étage $i + 1$.

Additionneur à propagation de retenue (cont.)



ci et *co* sont respectivement la **retenue entrante** et la **retenue sortante** de l'additionneur complet.



La *retenue entrante* de l'additionneur n bits c_0 est généralement à 0, mais peut permettre de calculer $A + B + 1$ ou de cascader des additionneurs.
La *retenue sortante* c_n peut être interprétée de deux façons :

- Si A et B sont sur n bits, $0 \leq A, B \leq 2^n - 1$ et $0 \leq S = A + B \leq 2^{n+1} - 2$.
  S nécessite donc en général $n + 1$ bits de codage.
  c_n est alors le $n + 1^{\text{ième}}$ bit du résultat (poids 2^n).
- Si le résultat S *doit* être codé sur n bits, il sera valide ssi $S < 2^n$, donc si $c_n = 0$.
  c_n est alors un indicateur de **débordement de capacité**.

Additionneur à propagation de retenue (cont.)

temps de calcul

Prise en compte des **temps de traversée** des additionneurs.

Nous supposons que pour les additionneurs complets, la somme s_i et la retenue c_{i+1} seront stables τ après la stabilisation des entrées a_i, b_i et c_i

Si à l'instant $t = 0$, les entrées ai et b_i et la retenue c_0 sont stables $\forall i$

- à l'instant $t = \tau$, la somme s_0 et la retenue c_1 seront stables.
- comme les entrées de l'additionneur 1 sont toutes stables à l'instant τ, les sorties c_2 et s_1 seront stables en $t = 2 \times \tau$
- ...
- à l'instant $t = n \times \tau$, les sorties c_n et s_{n-1} seront stables.

Dans le pire cas, le temps de calcul de l'additionneur à propagation de retenue de n bits est de $t_n = n \times \tau$.
Évolution linéaire du temps.
Très problématique pour des valeurs de n au delà de quelques unités.
Nécessité de schémas d'additioneurs rapides

Plan

1. Additionneurs 1 bit
2. Additionneur n bits à propagation de retenue
   - Addition non signée
3. **Addition/soustraction de nombre signés**
   - Additionneur/soustracteur
   - Validité du résultat
4. Additionneur à sauvegarde de retenue (*carry save adder*)
5. Additionneur à retenue anticipée

Plan

1. Additionneurs 1 bit
2. Additionneur n bits à propagation de retenue
3. **Addition/soustraction de nombre signés**
   - Additionneur/soustracteur
   - Validité du résultat
4. Additionneur à sauvegarde de retenue (*carry save adder*)
5. Additionneur à retenue anticipée

Addition de nombres signés

additionneur-soustracteur

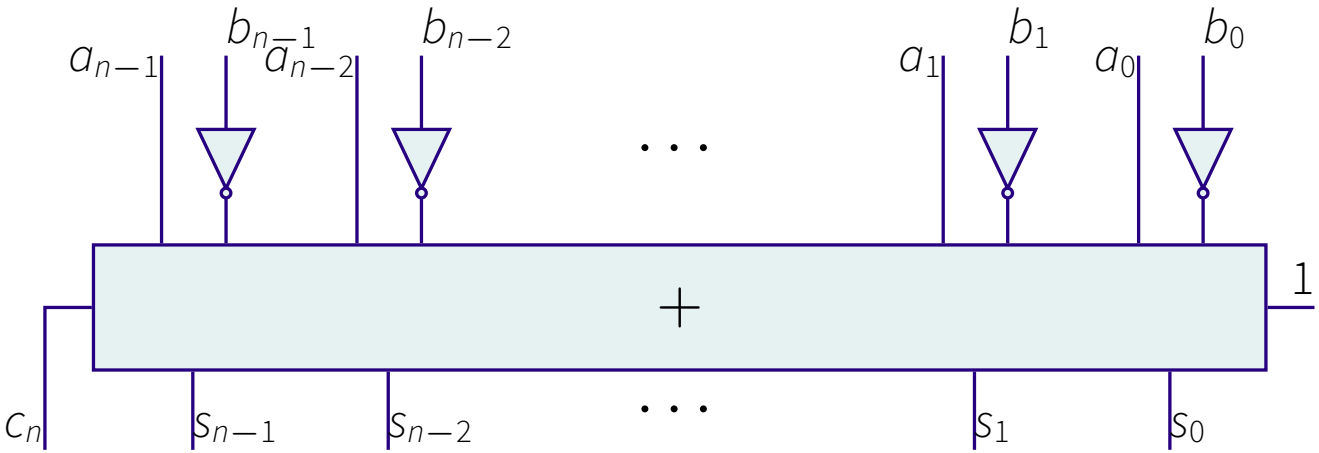
Comme $C_2(B) = 2^n - |B|$, à 2^n près, il est équivalent de calculer $A - B$ ou $A + C_2(B)$

Soustracteur = calcul du complément à deux du 2e opérande + addition avec le 1er opérande.

Comme $C_2(B) = \overline{B} + 1, A - B = A + \overline{B} + 1$.

Pour transformer un additionneur en soustracteur, il faut :

- inverser tous les bits de l'opérande B
- calculer $A + B$ en mettant la retenue entrante c_0 à 1



Construction d'un soustracteur n bits à partir d'un additionneur

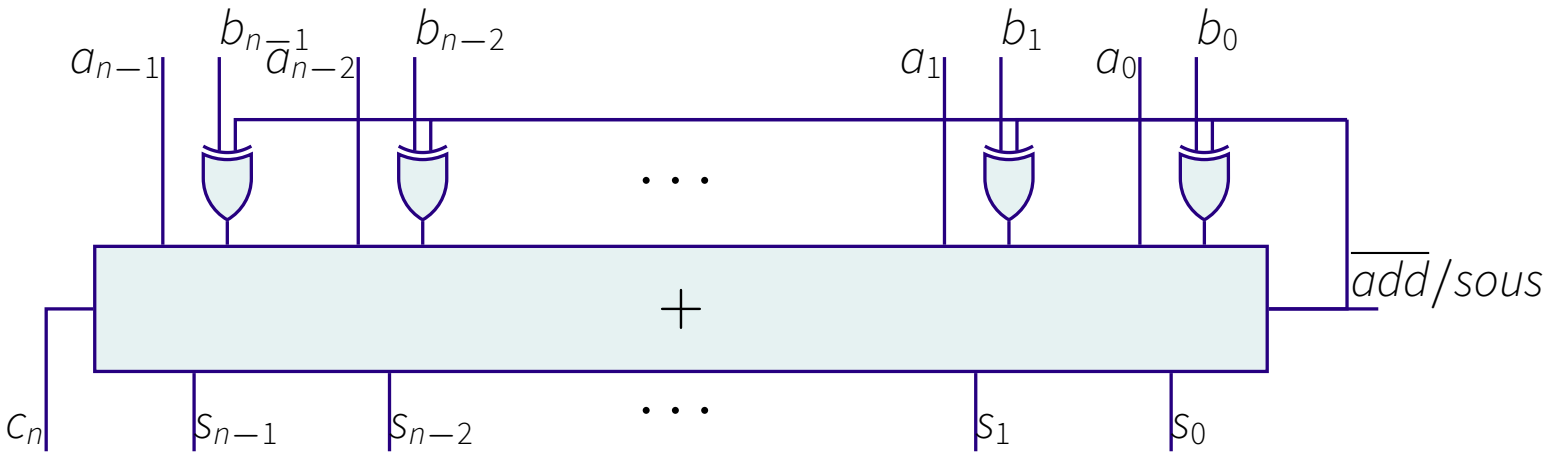
Addition de nombres signés (cont.)

additionneur-soustracteur

Il est possible de faire simplement un **additionneur/soustracteur**

Différence entre addition et soustraction :

- complémentation ou non du deuxième opérande
- mise à zéro ou à un de la retenue entrante c_0.



Si le signal $\overline{add}/sous = 0$ alors $S = A + (00\ldots0 \oplus B) + 0 = A + B$

Si le signal $\overline{add}/sous = 1$ alors $S = A + (11\ldots1 \oplus B) + 1 = A + \overline{B} + 1 = A - B$

Plan

1. Additionneurs 1 bit
2. Additionneur n bits à propagation de retenue
3. **Addition/soustraction de nombre signés**
   - Additionneur/soustracteur
   - Validité du résultat
4. Additionneur à sauvegarde de retenue (*carry save adder*)
5. Additionneur à retenue anticipée

Addition de nombres signés (cont.)

validité du résultat

La retenue sortante n'est **pas** un indicateur de validité du résultat pour l'addition de nombres signés.

3 cas à considérer :

A et B positifs ou nuls

Nous aurons $A + B \geq 0$ (toujours)

$$\begin{array}{rcccccccc} & & c_n = 0 & c_{n-1} & c_{n-2} & & c_1 & \\ A & & & 0 & a_{n-2} & \dots & a_1 & a_0 \\ +B & & + & 0 & b_{n-2} & \dots & b_1 & b_0 \\ \hline S & & & c_{n-1} & s_{n-2} & \dots & s_1 & s_0 \end{array}$$

Le résultat est valide si $0 \leq A + B < 2^{n-1}$.
Il faut donc $s_{n-1} = 0$.
Pour cela, il faut $c_{n-1} = 0 (= c_n)$.

A et B négatifs

$A + B$ est toujours < 0.

$$
\begin{array}{rccccccc}
 & & c_n = 1 & c_{n-1} & c_{n-2} & & c_1 & \\
A & & & 1 & a_{n-2} & \dots & a_1 & a_0 \\
+B & & + & 1 & b_{n-2} & \dots & b_1 & b_0 \\
\hline
S & & & c_{n-1} & s_{n-2} & \dots & s_1 & s_0
\end{array}
$$

Résultat valide si $-2^{n-1} \leq A + B < 0$
le bit de poids fort s_{n-1} doit être à 1.
Il faut donc $c_{n-1} = 1 (= c_n)$ pour que le résultat soit valide.

Remarque : la retenue sortante est *toujours* à 1, car $A = 2^n - |A|$, $B = 2^n - |B|, A + B = 2^{n+1} - (|A| + |B|) > 2^n$ si $|A|, |B| < 2^{n-1}$.

A et B de signes contraires

Nous supposerons $A \geq 0$ et $B < 0$.

Le résultat $A + B$ est de signe quelconque.

$$\begin{aligned} 0 \leq & \quad A \quad & < 2^{n-1} \\ -2^{n-1} \leq & \quad B \quad & < 0 \\ -2^{n-1} \leq & \; A + B & < 2^{n-1} \end{aligned}$$

Le résultat est donc *toujours* valide.

$$\begin{array}{rrcccccc} & c_n = c_{n-1} & & c_{n-1} & c_{n-2} & & c_1 & \\ A & & & 0 & a_{n-2} & \dots & a_1 & a_0 \\ +B & & + & 1 & b_{n-2} & \dots & b_1 & b_0 \\ \hline S & & & \overline{c_{n-1}} & s_{n-2} & \dots & s_1 & s_0 \end{array}$$

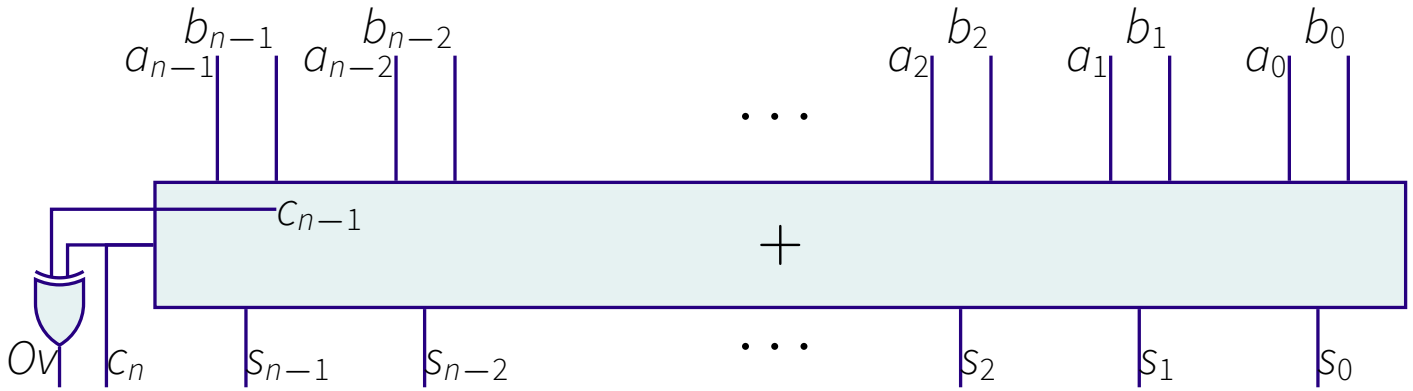
Dans ce cas, nous avons toujours $c_{n-1} = c_n$ et un résultat toujours valide.

Récapitulatif

A et $B \geq 0$	S valide ssi $c_{n-1} = 0$	$c_n = 0$ toujours
A et $B < 0$	S valide ssi $c_{n-1} = 1$	$c_n = 1$ toujours
A et B de signes contraires	S toujours valide	$c_n = c_{n-1}$ toujours

Dans tous les cas, le résultats sera valide si et seulement si $c_n = c_{n-1}$

Le **ou** exclusif des deux dernières retenues est un indicateur de débordement de capacité (*overflow*)



Implications logicielles

Indicateurs de débordement de capacité en signé et non-signé existent dans les processeurs et sont exploitables en assembleur.
Mais *les langages de haut niveau ne font* ***aucune*** *vérification de la validité d'un résultat*.

```
unsigned char x=17*17; // 289 > 255 nécessiterait plus de 8 bits
                       // Interprété comme 33 (289-256)
char y=15*15;          // 225 codable sur 8 bits, mais représente
                       // alors un nombre négatif sur un 'char'.
                       // Interprété comme -256+225=-31
```

Problème similaire pour les calculs intermédiaires.
Pour limiter les problèmes de débordement de capacité, le langage C spécifie que les calculs intermédiaires entiers sont effectués en `long` (en `double` pour des données flottantes).
Toujours laisser le compilateur décomposer un calcul.

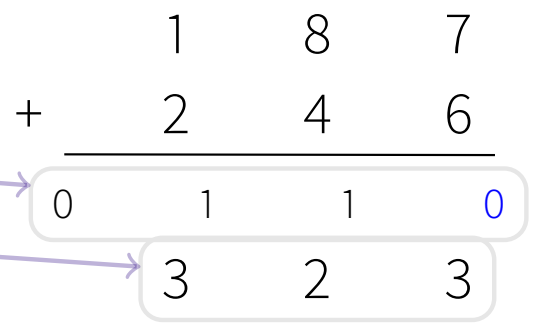
Plan

1. Additionneurs 1 bit
2. Additionneur n bits à propagation de retenue
   - Addition non signée
3. Addition/soustraction de nombre signés
   - Additionneur/soustracteur
   - Validité du résultat
4. **Additionneur à sauvegarde de retenue (*carry save adder*)**
5. Additionneur à retenue anticipée

Additionneurs à sauvegarde de retenue

Une manière de réduire le temps de calcul d'un APR est de ne *pas* propager la retenue, mais de la mémoriser :
additionneur à sauvegarde de retenue (ASR) (*carry save adder*).

Le « résultat » du calcul est alors formé de deux parties : la retenue R et la somme S.



« Résultat sous forme RS »
Il faut les additionner pour avoir le « vrai » résultat.

Une addition à sauvegarde de retenue assure que le bit de poids faible de R c_0 est à zéro.
$n-1$ additions supplémentaires pour assurer que $(c_{n-1}c_{n-2}\ldots c_0)$ sont tous nuls.

N'est pas intéressant en général :
- n étapes supplémentaires pour le calcul de $R+S$
- chaque étape nécessite de mémoriser de S et C et le temps de calcul d'une étape est $\tau_{\text{add}}+\tau_{\text{reg}}$

sauf pour faire des **accumulations**.

Additionneurs à sauvegarde de retenue (cont.)

accumulation : suite d'opérations (généralement additions), telle que le résultat de l'opération k soit un des opérandes de l'opération $k+1$.

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

Accumulation des éléments du vecteur `A[]` dans `S`.

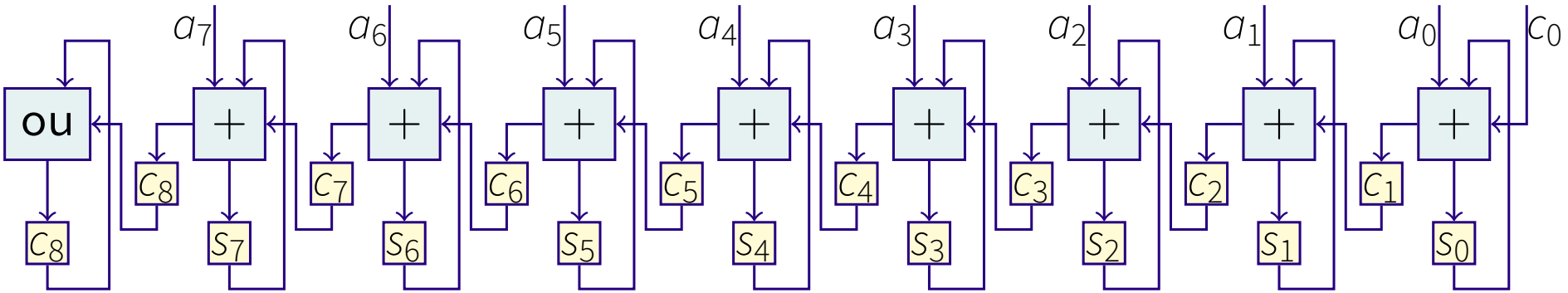
Seul compte le résultat final, et avoir les résultats intermédiaires sous forme RS n'est pas un problème si cela permet d'améliorer les performances.

Il sera nécessaire à la fin :

- soit d'accumuler $n-1$ fois fois zéro pour assurer que la retenue est identiquement nulle
- soit d'utiliser un additionneur normal pour calculer la somme de S et R

Les accumulations se rencontrent dans de nombreux problèmes (par exemple en traitement de signal) et sont à la base d'opérations comme la **multiplication**.

Additionneurs à sauvegarde de retenue (cont.)



Noter le **ou** pour préserver la retenue c_8 comme indicateur de débordement

Le temps d'une étape est $\tau + t_r$

- τ maximum entre temps (entrées $\rightarrow s$) et (entrées $\rightarrow co$) de l'additionneur
- t_r temps de traversée des registres ($\sqcup \rightarrow$ sortie)

Pour accumuler N nombres sur n bits :

	période horloge	nombre étapes	temps total
APR	$n \times \tau + t_r$	N	$N \times (n \times \tau + t_r)$
ASR	$\tau + t_r$	$N + (n-1)$	$(N + n - 1) \times (\tau + t_r)$

Si $\tau = t_r$ et $N \gg n \gg 1$, $t_{\text{APR}} \approx N \times n \times \tau$ et $t_{\text{ASR}} \approx N \times 2 \times \tau$

Plan

1. Additionneurs 1 bit
2. Additionneur n bits à propagation de retenue
   - Addition non signée
3. Addition/soustraction de nombre signés
   - Additionneur/soustracteur
   - Validité du résultat
4. Additionneur à sauvegarde de retenue (*carry save adder*)
5. **Additionneur à retenue anticipée**

Additionneurs à génération de retenue anticipée

Principe des **additionneurs à génération de retenue anticipée** (ARA) (*carry lookahed adder*)

- on calcule préalablement *toutes* les retenues entrantes des différents bits
- on calcule ensuite l'addition en temps unitaire

Pour le $i$ème bit, calcul de la retenue sortante $c_i + 1$ en fonction de a_i, b_i et la retenue entrante c_i

$$\begin{aligned} c_{i+1} &= a_i b_i + a_i c_i + b_i c_i \\ &= g_i + p_i c_i \end{aligned}$$

en posant

$$\begin{aligned} g_i &= a_i b_i && \text{fonction de } \textit{génération} \text{ de retenue} \\ p_i &= a_i + b_i && \text{fonction de } \textit{propagation} \text{ de retenue} \end{aligned}$$

En itérant ce calcul, on peut calculer tous les c_i en fonction de c_0

Si l'on arrive à calculer rapidement les c_i, en fonction des a_k, b_k, $\forall k < i$ et de c_0, on accélère l'addition.

Additionneurs à génération de retenue anticipée (cont.)

Calcul des retenues successives

$$
\begin{aligned}
c1 &= g_0 + p_0 c_0 \\
c2 &= g_1 + p_1 c_1 \\
&= g_1 + p_1 (g_0 + p_0 c_0) \\
&= \underbrace{g_1 + p_1 g_0}_{G_{10}} + \underbrace{p_1 p_0}_{P_{10}}\, c_0 \\
&= G_{10} + P_{10} c_0
\end{aligned}
$$

avec

$$
\begin{array}{lcll}
G_{10} & = & g_1 + p_1 g_0 & \text{fonction de } \textit{génération} \text{ sur les bits 1–0} \\
P_{10} & = & p_1 p_0 & \text{fonction de } \textit{propagation} \text{ sur les bits 1–0}
\end{array}
$$

A l'ordre $i+1$

$$
\begin{aligned}
c_{i+1} &= g_i + p_i c_i \\
&= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \\
&= \cdots \\
&= \underbrace{g_i + p_i g_{i-1} + \cdots + p_i p_{i-1} \ldots p_1 g_0}_{G_{i0}} + \underbrace{p_i p_{i-1} \ldots p_0}_{P_{i0}}\, c_0 \\
&= G_{i0} + P_{i0}\, c_0
\end{aligned}
$$

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Pour simplifier la notation, on considère :

- le doublet génération-propagation $(g_i, p_i) = (a_i b_i, a_i + b_i)$ associé à chaque bit
- l'opérateur $\blacksquare$ qui calcule c_{i+1} en fonction de (g_i, p_i) et de c_i
  $c_{i+1} = (g_i, p_i) \blacksquare c_i = g_i + p_i c_i$
- l'opérateur $\circ$ qui permet de construire des fonctions de génération-propagation sur plusieurs bits (G_{ij}, P_{ij}) en combinant des doublets (g_i, p_i)

$$\begin{aligned}(G_{i,i-1}, P_{i,i-1}) &= (g_i + p_i g_{i-1}, p_i p_{i-1}) \\ &= (g_i, p_i) \circ (g_{i-1}, p_{i-1})\end{aligned}$$

Le calcul de la retenue à l'ordre $i+1$ en fonction de c_0 s'écrit alors simplement:

$$\begin{aligned}c_{i+1} &= (G_{i,0}, P_{i,0}) \blacksquare c_0 \\ &= \big((g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \cdots \circ (g_0, p_0) \big) \blacksquare c_0\end{aligned} \tag{1}$$

Le problème se ramène à combiner des applications de l'opérateur $\circ$.

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Propriétés de l'opérateur $\circ$

- il n'est ***pas*** commutatif
  $(g', p') \circ (g, p) = (g' + p'g, p'p) \neq (g, p) \circ (g', p') = (g + pg', pp')$
  On ne peut pas permuter des opérandes
- il est **idempotent** : $(g, p) \circ (g, p) = (g + pg, pp) = (g, p)$
- il est **associatif** : $((g'', p'') \circ (g', p')) \circ (g, p) = (g'', p'') \circ ((g', p') \circ (g, p))$

$$((g'', p'') \circ (g', p')) \circ (g, p) = (g'' + p''g', p''p') \circ (g, p) = (g'' + p''g' + p''p'g, p''p'p)$$

$$(g'', p'') \circ ((g', p') \circ (g, p)) = (g'', p'') \circ (g' + p'g, p'p) = (g'' + p''g' + p''p'g, p''p'p)$$

$$= (g'', p'') \circ (g', p') \circ (g, p)$$

Un opérateur associatif donne un résultat identique, quel que soit le parenthésage.
Mais des parenthésages différents peuvent donner des temps de calcul différents.

Nombreux opérateurs associatifs en arithmétique, traitement de signal, calcul numérique, etc.

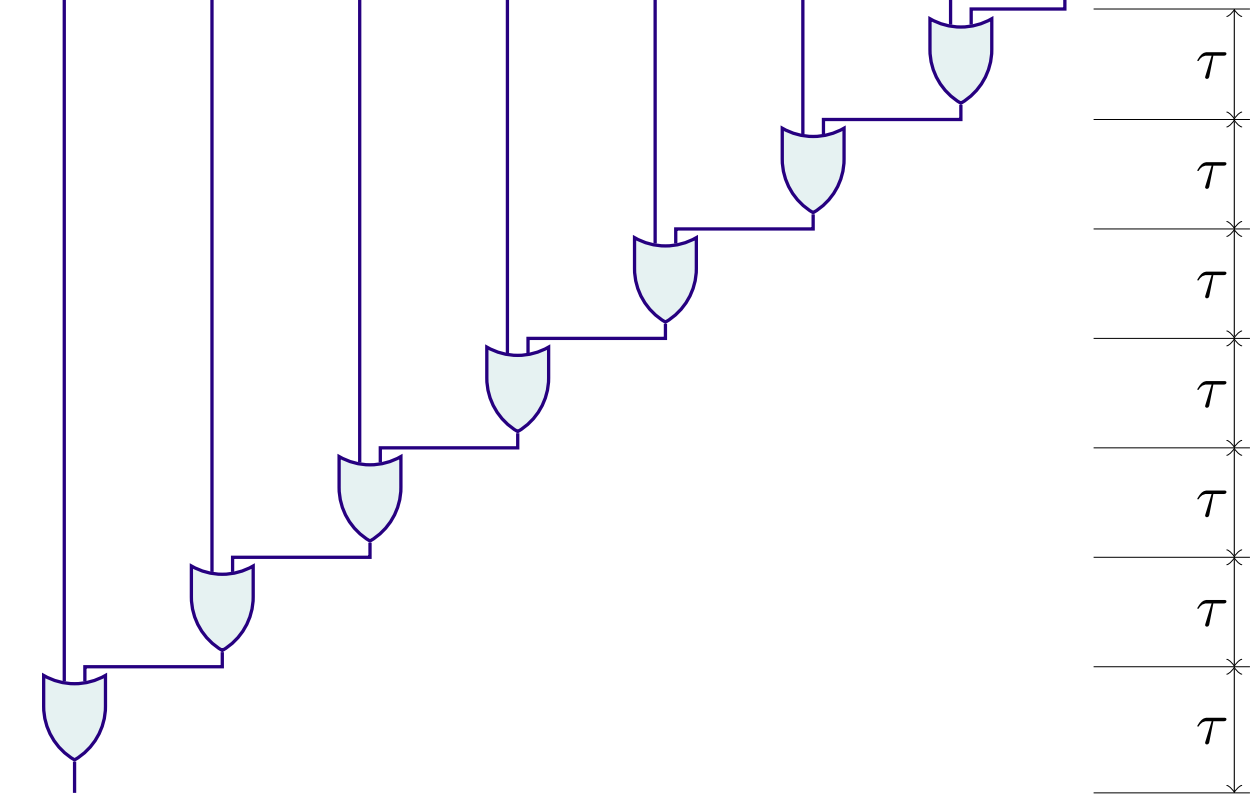
Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Exemple : **ou** (associatif) à 8 entrées à partir de portes **ou** à deux entrées

$$x_7+x_6+x_5+x_4+x_3+x_2+x_1+x_0 = (x_7+(x_6+(x_5+(x_4+(x_3+(x_2+(x_1+x_0)))))))$$

$$(x_7+(x_6+(x_5+(x_4+(x_3+(x_2+(x_1+x_0)))))))$$



Temps total : 7τ

Dans le cas général, pour n bits, temps : $(n-1)\times\tau$

Le temps varie linéairement avec le nombre de bits.

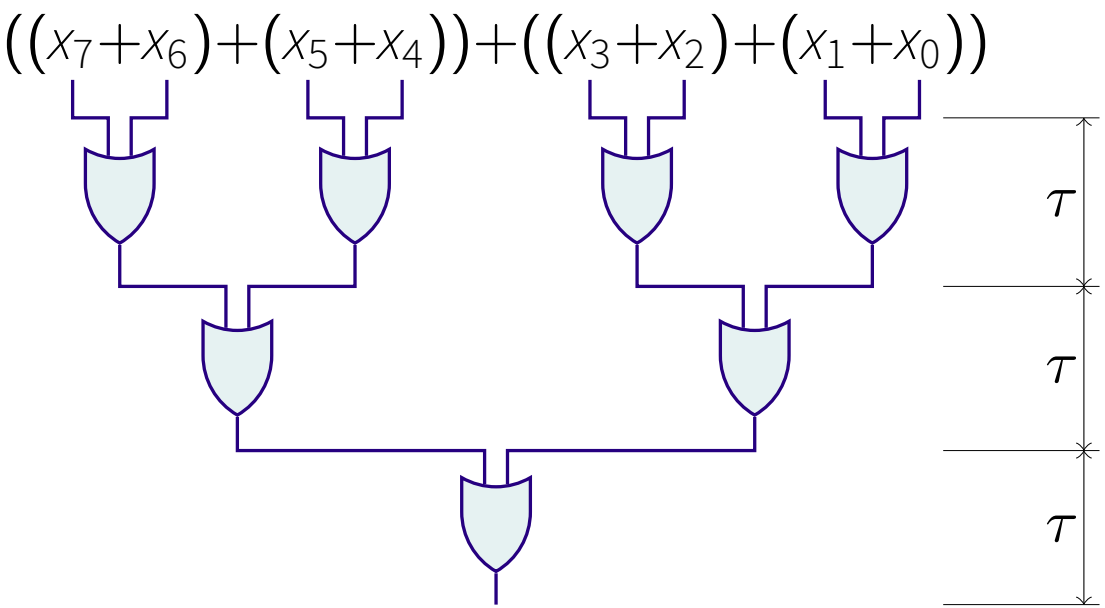
« **ou** à propagation »

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Utilisation d'un parenthésage différent

$$x_7+x_6+x_5+x_4+x_3+x_2+x_1+x_0 = ((x_6+x_7)+(x_4+x_5))+((x_2+x_3)+(x_0+x_1))$$



Trois couches à traverser.

Le temps de calcul n'est plus que 3τ

Doubler la taille ne rajoute qu'une couche

Dans le cas général de 2^k bits, temps de calcul
$t = k \times \tau$
ou pour n bits
$t = \log_2 n \times \tau$

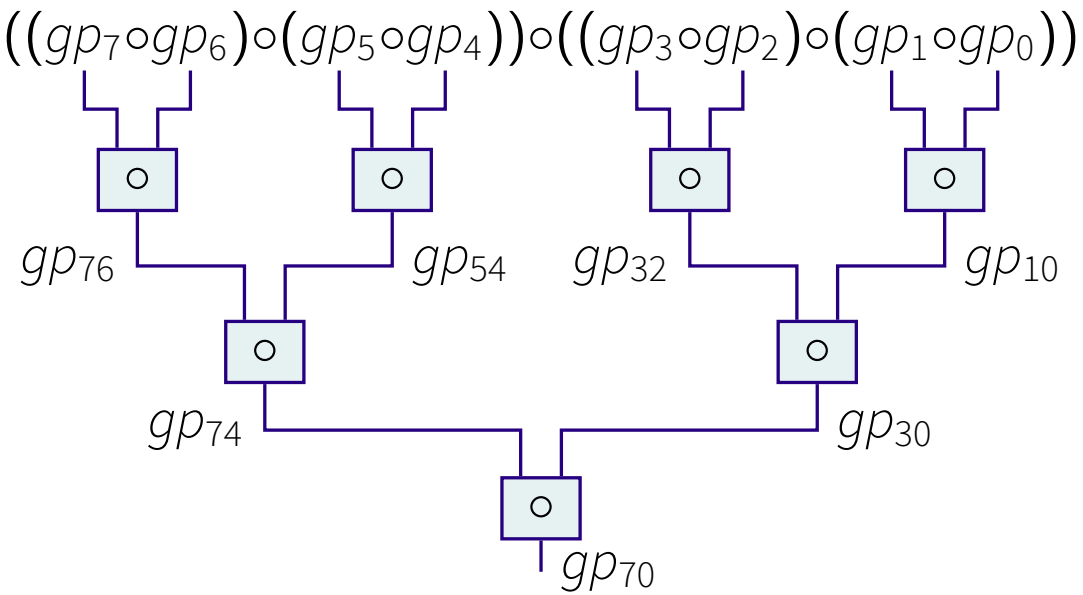
On passe d'un temps de calcul linéaire à un temps logarithmique.

Réduction du temps grâce au calcul simultané de plusieurs **ou**.
Exploitation du **parallélisme**

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Même principe pour l'opérateur $\circ$



Pour n bits,
à l'étage $l \in \{1, \ldots, \log n\}$,
on calcule (G_{ij}, P_{ij}) tel que

- $j = k \times 2^l : j < n$
- $i = j + 2^l - 1$

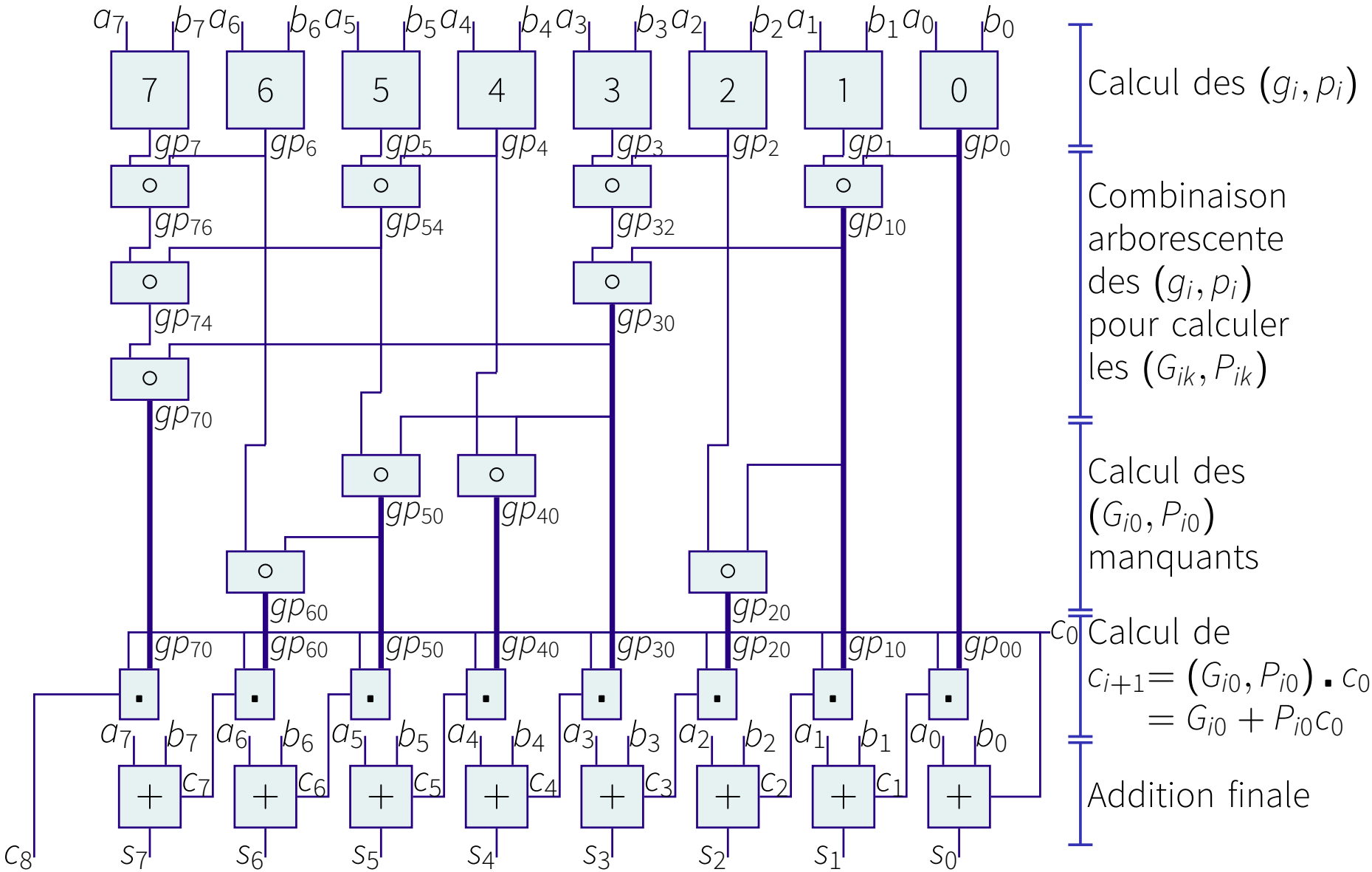
On obtient les (G_{i0}, P_{i0}) pour $i = 2^l - 1$

Problème à résoudre un peu différent : trouver *toutes* les retenues entrantes.
Pour cela on a besoin de *tous* les $(G_{i0}, P_{i0}), \forall i \in \{1, \ldots, 2^n - 1\}$

Revient à *distribuer* l'application de l'opérateur $\circ$ sur les données.
« calcul préfixe »

Peut se faire en combinant les différents (G_{ij}, P_{ij}) partiels calculés.

Additionneurs à génération de retenue anticipée (cont.)



Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Le chemin critique est pour le calcul de (G_{60}, P_{60}) (calcul de c_7).
Nécessite de traverser 4 couches de $\circ$

Dans le cas général, on traverse $2 \times (\log n - 1)$ couches.

	APR	ARA
	n	$2 \times (\log n - 1)$
8	8	4
16	16	6
32	32	8
64	64	10

Temps logarithmique

Doubler le nombre de bits ne rajoute que 2 couches à traverser.

Intéressant pour $n \geq 16$

Schéma présenté : Brent-Kung.
Il existe d'autres schémas d'ARA un peu plus réguliers et de temps de calcul identique. Ils utilisent le caractère idempotent de $\circ$ pour faire des calculs redondants et nécessitent donc plus de portes.

Il existe d'autre structures d'additionneurs rapides, par exemple en $\sqrt{n}$.

Multiplication entière

Alain MÉRIGOT

Université Paris Saclay

Plan

1. Multiplication non signée par addition décalage (ou série-parallèle)
2. Cas des nombres signés — algorithme de Booth
3. Multiplieur en tableau
4. Mise en œuvre en pipeline

Multiplication non signée

Soient $A = (a_{n-1}, \ldots, a_1, a_0)$ et $B = (b_{n-1}, \ldots, b_1, b_0)$ deux entiers non signés de n bits.

$$\begin{aligned} A \times B &= A \times \sum_{j=0}^{n-1} b_j \times 2^j \\ &= \sum_{j=0}^{n-1} A \times b_j \times 2^j \end{aligned}$$

Comme

$$A \times b_j = \begin{cases} 0 & \text{si } b_j = 0 \\ A & \text{si } b_j = 1 \end{cases}$$

et

$$A \times 2^j = (a_{n-1}, \ldots, a_1, a_0, \overbrace{0, \ldots, 0, 0}^{\times j})$$

la multiplication se ramène à une suite d'additions et de décalages vers la gauche.

Le résultat sera sur $2n$ bits car $0 \leq A < 2^n$ et $0 \leq B < 2^n \implies 0 \leq A \times B < 2^{2n}$

Multiplication non signée par addition décalage

Déduction d'un algorithme de l'expression $A \times B = \sum_{j=0}^{n-1} A \times b_j \times 2^j$

```
1   résultat ← 0
2   pour j ← 0 à n − 1
3       résultat ← résultat + b_j × A × 2^j
```

Réécriture de l'algorithme sous une forme plus adaptée à une mise en oeuvre matérielle

```
1   résultat ← 0
2   pour j ← 0 à n − 1
3       si b_j = 1
4           résultat ← résultat + A          ▷ Addition
5       A ← A × 2                            ▷ Décalage
```

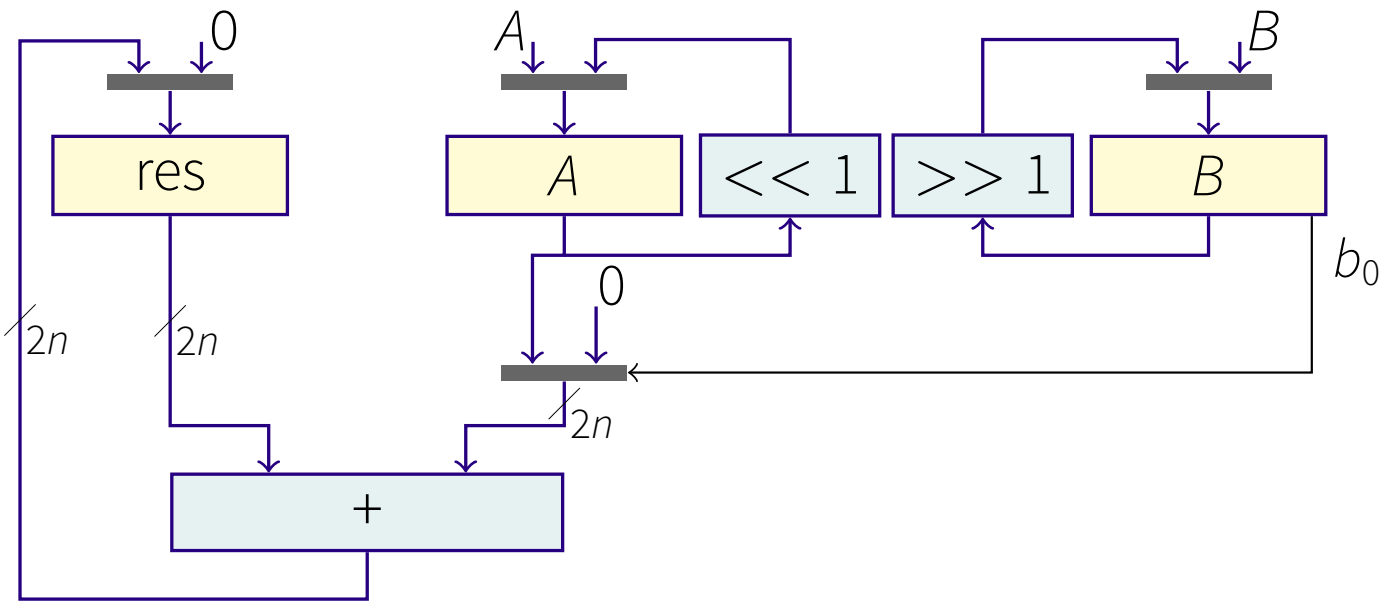
Algorithme de multiplication non signée par addition–décalage (ou série–parallèle)

Cet algorithme peut aisément se mettre en œuvre à l'aide d'un automate sur un chemin de données adapté.

Multiplication non signée par addition décalage (cont.)

Le calcul $A +$ res doit se faire sur $2n$ bits.
L'extraction du $i$ème bit b_i de B est un peu complexe.
$\Longrightarrow$ Utilisation de b_0 et d'un décalage à droite de B.

```
1   résultat ← 0
2   pour j ← 0 à n − 1
3       résultat ← résultat + si b_0 = 1 alors A sinon 0 fin si
4       A ← A × 2 ; B ← B ÷ 2
```



Un additionneur $2n$ bits

2 registres $2n$ bits, un registre n bits

Multiplication non signée par addition décalage (cont.)

De nombreuses optimisations sont possibles.

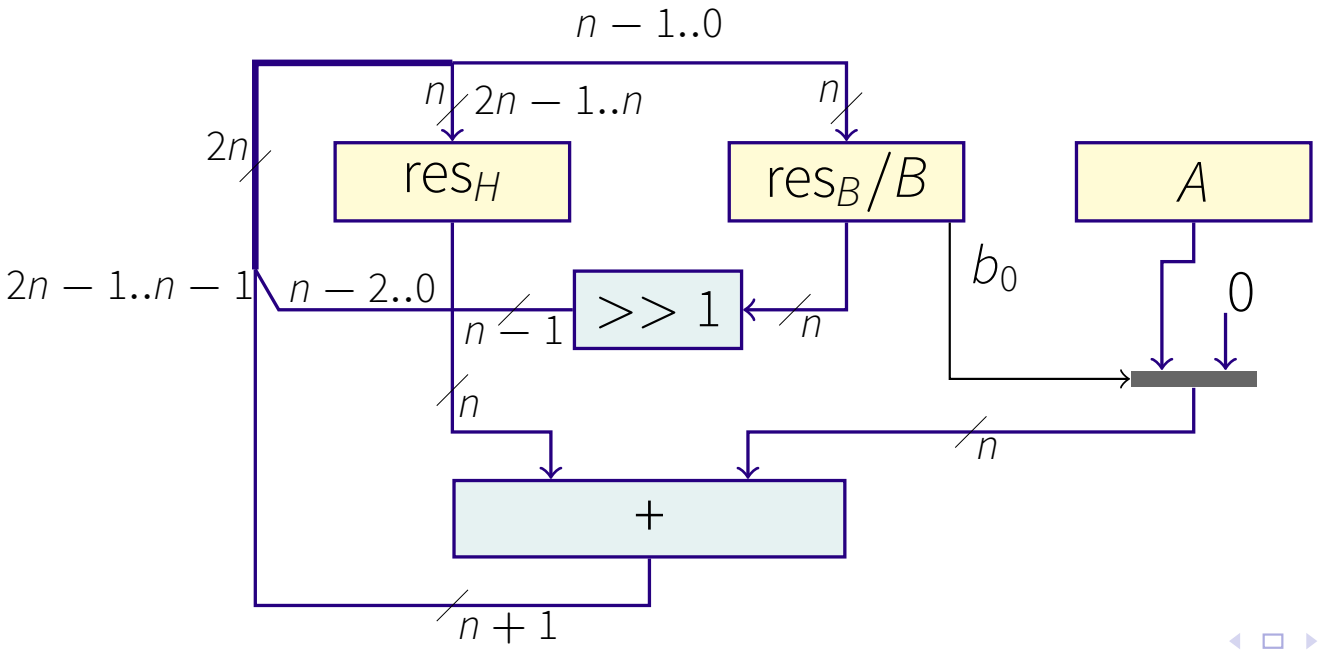
- A sur $2n$ bits et à l'étape i, les i LSB et les $n - i$ MSB de A sont à 0.
- pour $i = 0$, res $= 0$ et à l'étape i, ses $n - 1 - i$ bits de poids fort sont nulls.

$\Longrightarrow$ On peut effectuer les calculs sur uniquement n bits (et non $2n$).

De plus, à l'étape i, les i bits de poids faibles de res ne seront plus modifiés et les i bits de poids forts de B seront à zéro.

$\Longrightarrow$ utilisation d'un registre unique pour B et les poids faibles de res.

Décalage à gauche de A remplacé par décalage à droite de res/B.



un additionneur n bits

3 registres n bits.

Contrôlé par un automate

Nécessite n cycles de calcul

Plan

1. Multiplication non signée par addition décalage (ou série-parallèle)
2. **Cas des nombres signés — algorithme de Booth**
3. Multiplieur en tableau
4. Mise en œuvre en pipeline

Cas des nombres signés

Soit $B = (b_{n-1}, \ldots, b_1, b_0)$ un nombre signé codé en complément à deux.

$$B = -2^{n-1} \times b_{n-1} + \sum_{i=n-2}^{0} \times b_i 2^i$$

L'irrégularité entre le bit $n-1$ et les autres complexifie la mise en oeuvre.

En remarquant que $2^i = 2^{i+1} - 2^i$, B peut se réécrire

$$\begin{array}{rcl}
B & = & -2^{n-1} \times b_{n-1} + 2^{n-2} \times b_{n-2} + 2^{n-3} \times b_{n-3} + \cdots + 2b_1 + b_0 \\
& = & -2^{n-1} \times b_{n-1} + 2^{n-1} \times b_{n-2} \\
& & \quad - 2^{n-2} \times b_{n-2} + 2^{n-2} \times b_{n-3} \\
& & \quad - 2^{n-3} \times b_{n-3} + \cdots \\
& & \quad - 2b_1 + 2b_0 \\
& & \quad - b_0 \\
& = & \sum_{i=n-1}^{0} (-b_i + b_{i-1}) \times 2^i
\end{array}$$

en posant $b_{-1} = 0$.

Algorithme (ou réécriture) de **Booth**

Cas des nombres signés (cont.)

En posant $b'_i = -b_i + b_{i-1}$

$$A \times B = \sum_{i=n-1}^{0} b'_i \times 2^i \times A$$

Suivant les cas, b'_i vaut 0, -1 ou $+1$ et il faut ajouter ou soustraire $A \times 2^i$

b_i	b_{i-1}	$b'_i = b_i - b_{i-1}$		
1	1	0	$\rightarrow$	$+0$
1	0	-1	$\rightarrow$	$-2^i \times A$
0	1	1	$\rightarrow$	$+2^i \times A$
0	0	0	$\rightarrow$	$+0$

Cas des nombres signés (cont.)

D'où l'algorithme de multiplication de nombres signés de Booth

```
1  résultat ← 0
2  pour j ← 0 à n − 1
3      selon b_j b_{j−1}
4          cas 10 :        résultat ← résultat - A
5          cas 01 :        résultat ← résultat + A
6          cas 00 ou 11 :  résultat ← résultat
7      A ← A × 2                                  ▷ Décalage
```

Adaptation du schéma du multiplieur non signé.

On utilise fréquemment une décomposition en base 4 pour réduire le nombre d'étapes (*algorithme de Booth modifié*).
A chaque étape, il faut alors ajouter au résultat, soit 0, soit $\pm A$, soit $\pm 2 \times A$

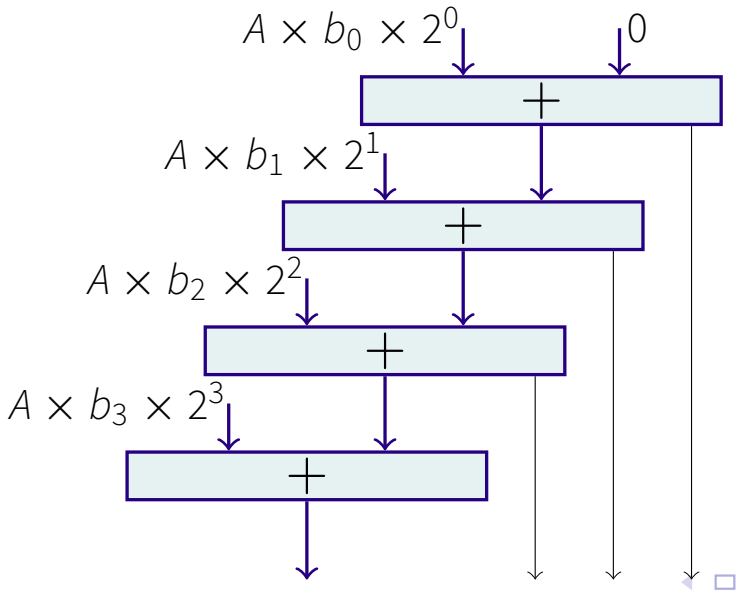
Plan

1. Multiplication non signée par addition décalage (ou série-parallèle)
2. Cas des nombres signés — algorithme de Booth
3. **Multiplieur en tableau**
4. Mise en œuvre en pipeline

Multiplieur en tableau

Pour accélérer, déroulage dans l'espace de la suite d'additions au moyen de plusieurs additionneurs.

$$
\begin{array}{cccccccc}
 & & & & a_{n-1} & a_{n-2} & \cdots & a_1 \quad a_0 \\
 & & & \times & b_{n-1} & b_{n-2} & \cdots & b_1 \quad b_0 \\
\hline
 & & & & b_0a_{n-1} & b_0a_{n-2} & \cdots & b_0a_1 \quad b_0a_0 \\
+ & & & b_1a_{n-1} & b_1a_{n-2} & \cdots & b_1a_1 & b_1a_0 \\
+ & & b_2a_{n-1} & b_2a_{n-2} & \cdots & b_2a_1 & b_2a_0 & \\
\vdots & & & \vdots & \vdots & & & \\
+ & b_{n-1}a_{n-1} & b_{n-1}a_{n-2} & \cdots & b_{n-1}a_1 & b_{n-1}a_0 & &
\end{array}
$$

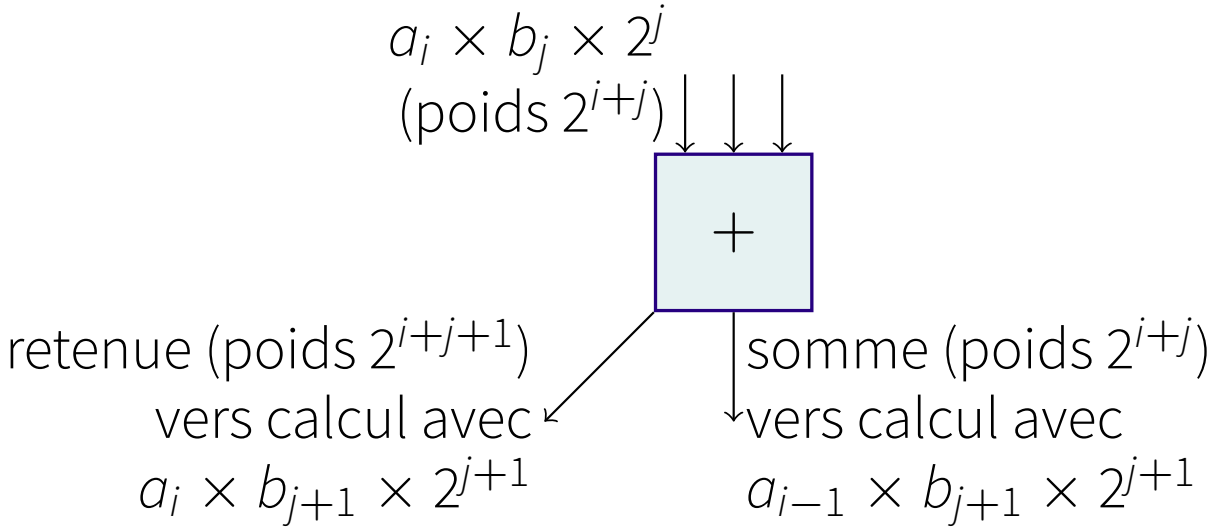


Multiplieur en tableau (cont.)

Problème : n additionneurs à traverser

- avec des additionneurs à propagation, chaque addition prend un temps n et la multiplication est en n^2
- avec des additionneurs rapides (type à génération anticipée), le temps reste de l'ordre de $n \times \log n$

Mais une multiplication est une *accumulation*, et on peut utiliser des **additionneurs à sauvegarde de retenue**.
Chaque additionneur génère somme et retenue et la retenue est prise en compte à l'étage suivant.

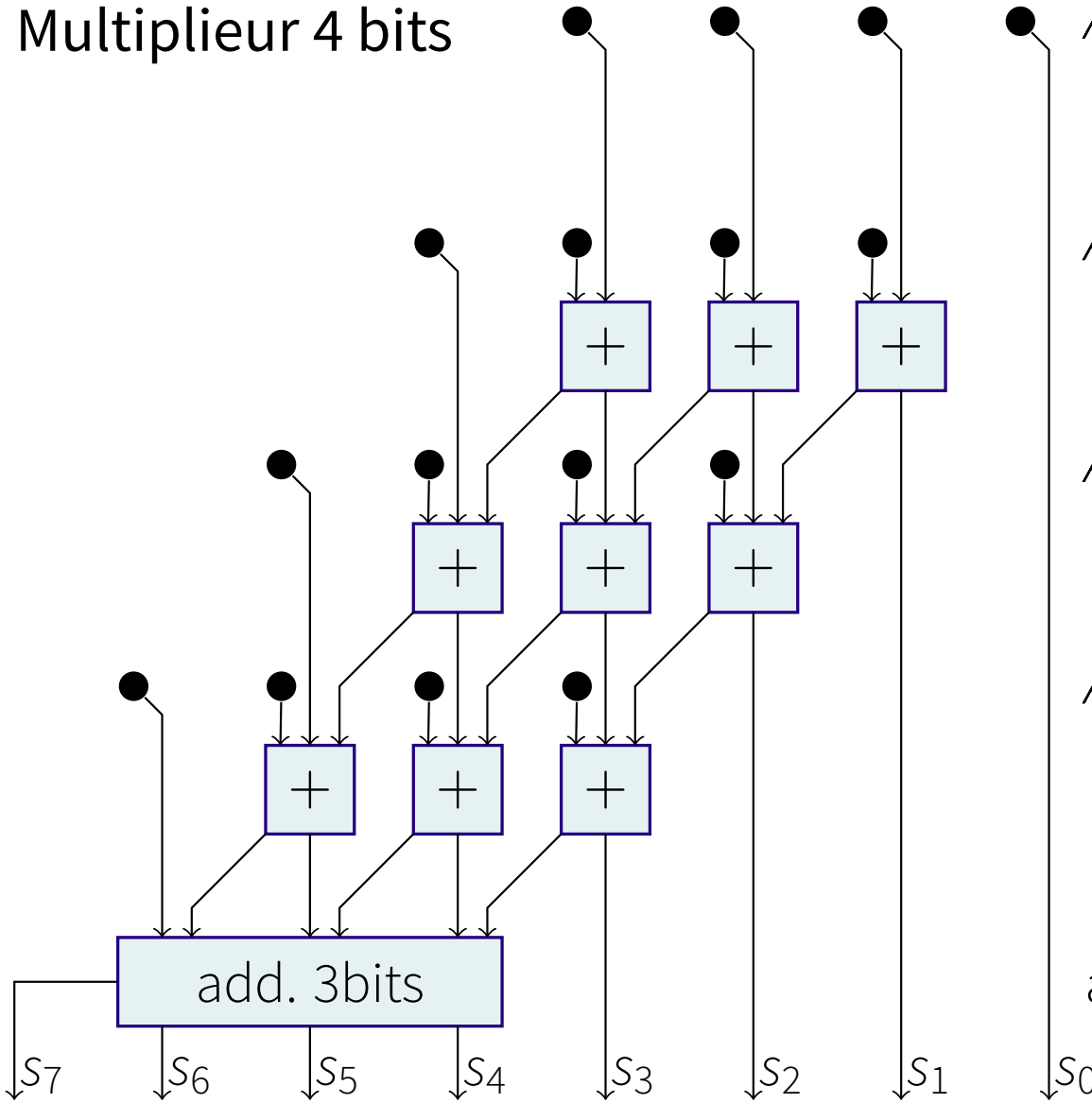


Chaque couche est traversée en temps 1.

Temps total $\approx n$
+ addition finale des sommes et retenue

Multiplieur en tableau (cont.)

Multiplieur 4 bits



$A \times b_0 \times 2^0$

$A \times b_1 \times 2^1$

$A \times b_2 \times 2^2$

$A \times b_3 \times 2^3$

addition finale

Pour une multiplication $n \times n$, il faut :
$(n-1) \times (n-1)$ add. 1 bit
$+ n$ add. 1 bit (add. finale à propagation).

Le temps est $(n-1) \times \tau + (n-1) \times \tau$

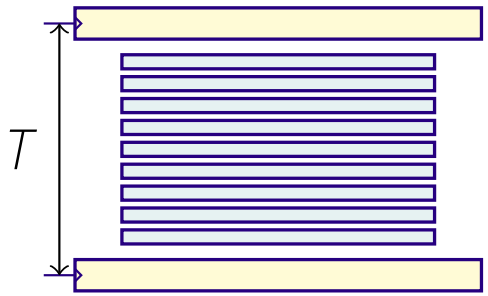
Plan

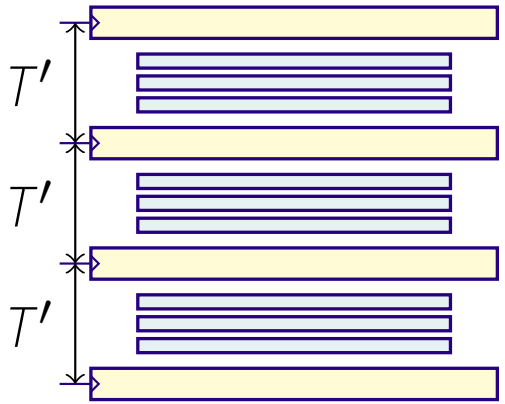
1. Multiplication non signée par addition décalage (ou série-parallèle)
2. Cas des nombres signés — algorithme de Booth
3. Multiplieur en tableau
4. **Mise en œuvre en pipeline**

Mise en oeuvre pipeline

Le temps de calcul $2 \times (n-1)$ est supérieur au chemin critique d'un additionneur rapide.
La présence d'un multiplieur va imposer une période d'horloge T plus *longue*.

Pour éviter de dégrader T et donc l'ensemble des performances
$\Longrightarrow$ découpage du multiplieur en plusieurs *étages*, séparé par des registres.





Si p étages, la période d'horloge sera approximativement divisée par p;

Technique du « **pipeline** »

Mise en oeuvre pipeline (cont.)

L'introduction de registres permet d'effectuer plusieurs multiplications en même temps.
Soit à calculer
`for(int i=0;i<N;i++) R[i]=A[i]*B[i];`

A $t = 0$, début du calcul de `R[0]=A[0]*B[0]`
A $t = 1$, suite du calcul de `R[0]` et début du calcul de `R[1]=A[1]*B[1]`
A $t = 2$, suite des calculs de `R[0]` et `R[1]` et début du calcul de `R[2]`
etc.

t_0	t_1	t_2	t_3	t_4	t_5	t_6
$R^0 = A^0 \times b_0^0$	$R^0 += A^0 \times b_1^0 \times 2^1$	$R^0 += A^0 \times b_2^0 \times 2^2$	$R^0 += A^0 \times b_3^0 \times 2^3$			
	$R^1 = A^1 \times b_0^1$	$R^1 += A^1 \times b_1^1 \times 2^1$	$R^1 += A^1 \times b_2^1 \times 2^2$	$R^1 += A^1 \times b_3^1 \times 2^3$		
		$R^2 = A^2 \times b_0^2$	$R^2 += A^2 \times b_1^2 \times 2^1$	$R^2 += A^2 \times b_2^2 \times 2^2$	$R^2 += A^2 \times b_3^2 \times 2^3$	
			$R^3 = A^3 \times b_0^3$	$R^3 += A^3 \times b_1^3 \times 2^1$	$R^3 += A^3 \times b_2^3 \times 2^2$	$R^3 += A^3 \times b_3^3 \times 2^3$
				$\cdots$	$\cdots$	$\cdots$

Le pipeline permet de :
- réduire le temps de cycle
- lancer un calcul par cycle

$\Longrightarrow$ forte amélioration des performances

Mise en oeuvre pipeline (cont.)

On distingue :

latence temps entre le début d'un calcul et la disponibilité des résultats
inchangée (voire légèrement augmentée) par le pipeline

débit nombre d'opérations possibles par unité de temps
lié au temps de cycle et fortement amélioré par le pipeline

Supposons un multiplieur avec n couches d'additionneurs.
τ_a et τ_r temps de traversée des additionneurs et des registres
mise en pipeline avec k étages de p couches d'additions ($n = k \times p$)

	temps cycle	latence	débit
sans pipeline	$\tau_r + n \times \tau_a$	$\tau_r + n \times \tau_a$	$\frac{1}{\tau_r + n \times \tau_a}$
avec pipeline	$\tau_r + p \times \tau_a$	$k \times (\tau_r + p \times \tau_a)$ $= k \times \tau_r + n \times \tau_a$	$\frac{1}{\tau_r + p \times \tau_a}$

Si $n = 64, p = k = 8, \tau_a = \tau_r$,

- débit $1/65 \rightarrow 1/9$ ($\times 7$)
- latence $65 \rightarrow 72$ ($+10\%$)

Mise en oeuvre pipeline (cont.)

Le pipeline permet

- d'améliorer fortement le débit
- au prix d'une légère augmentation de la latence
- et d'un surcoût matériel limité

Omniprésent dans les architectures actuelles

- opérateurs de calculs (multiplieurs, diviseurs, calcul flottant)
- processeurs (CPU, GPU) (exécution pipeline des instructions)
- opérateurs de calcul vectoriel, traitement de signal, IA, etc

En général le débit est ce qui détermine les performances
mais l'augmentation de latence peut être pénalisante dans certains cas (*récurrences*)

```
for(int i=0;i<N;i++) R *= A[i];
```

Il faut attendre la fin d'un calcul pour démarrer le calcul suivant.
Pas de possibilité d'exécution simple en pipeline.

Mise en oeuvre pipeline (cont.)

Version pipeline de l'additionneur (avec 1 couche d'additionneur par étage)

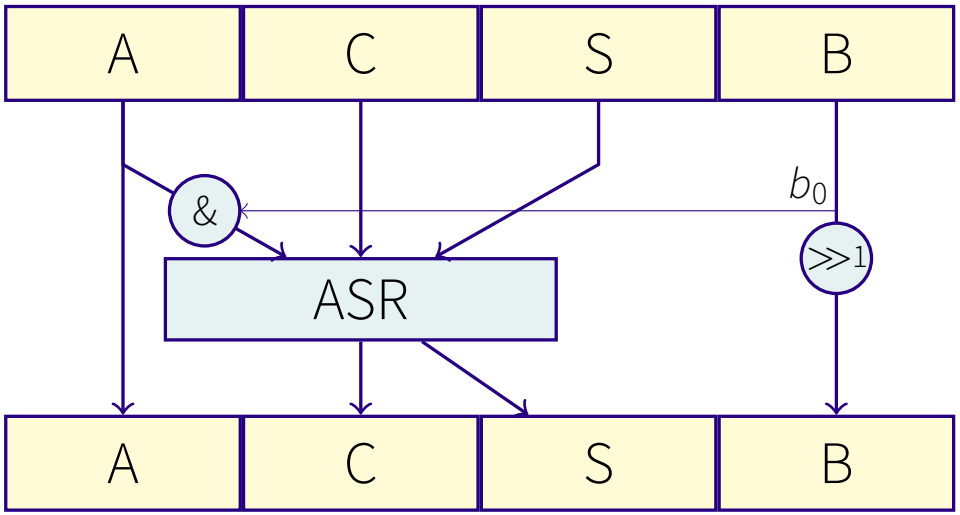
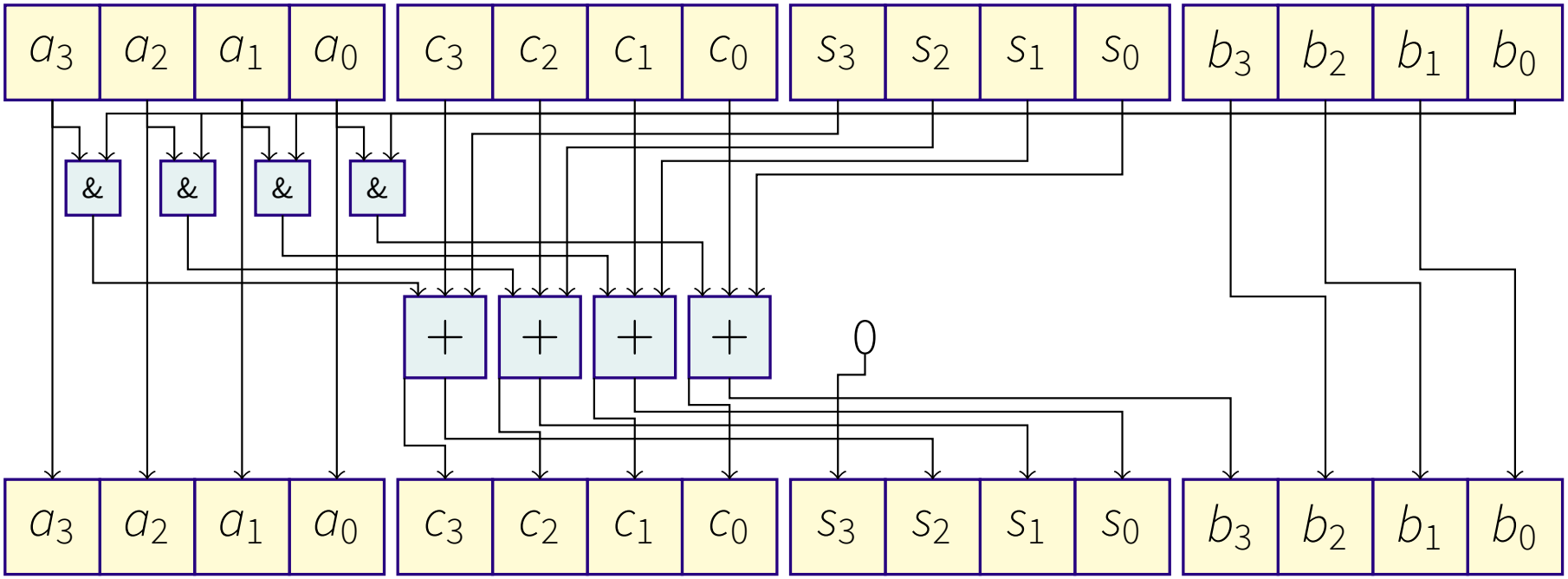


Schéma de principe d'un étage d'additionneur.

Le schéma détaillé assure que *tous* les étages sont identiques. (diverses optimisations sont possibles)



Mise en oeuvre pipeline (cont.)

Il existe des schémas de multiplieurs plus rapides (en un temps $\log n$) (Wallace-Dadda, Luk-Vuillemin).
Mais leur structure est plus irrégulière, et ils sont plus délicats à pipeliner.
Rarement utilisés.

En pratique, sur les processeurs actuels, multiplieurs ($64 \times 64 \rightarrow 64$ ou $32 \times 32 \rightarrow 64$) en tableau, pipelinés sur 3 à 5 étages.

Les premiers étages réalisent les additions à sauvegarde de retenue.
Le dernier étage permet d'effectuer l'addition finale avec un additionneur rapide.

Utilisation fréquente d'un algorithme de Booth en base 4 (algorithme de Booth modifié).

Division entière

Alain MÉRIGOT

Université Paris sud

Plan

1. **Division par soustraction-décalage**
   - Division avec restauration
   - Division sans restauration
   - Mise en oeuvre matérielle de la division sans restauration
2. **Division itérative**

Division par soustraction-décalage

Soit Dd le dividende, Dv le diviseur.
Nous supposerons $0 \leq Dv, Dd < 1$ et $Dd < Dv$
Les nombres seront codés en virgule fixe par des puissances négatives de 2

Si $Dv = (d_{n-1}, \ldots, d_1, d_0)$

$$\begin{aligned} Dv &= 0.d_{n-1}d_{n-2}\cdots d_1 d_0 \\ &= d_{n-1} \times 2^{-1} + d_{n-2} \times 2^{-2} + \cdots + d_1 \times 2^{-n-1} + d_0 \times 2^{-n} \\ &= \sum_{i=n-1}^{0} d_i \times 2^{i-n} \\ &= 2^{-n} \times \sum_{i=n-1}^{0} d_i \times 2^{i} \end{aligned}$$

$Dd \div Dv$ produit deux nombres : le quotient Q et le reste R avec :

$$Dd = Q \times Dv + R$$
$$\text{et } 0 \leq R < Dv$$

Si $0 \leq Dv, Dd < 1$ et $Dd < Dv$, nous aurons aussi $0 \leq Q, R < 1$.

Si Dv et Dd sont codés sur n bits, Q et R sont aussi codés avec la même précision.

Algorithme de division.

Détermination des restes successifs : comparaison avec le diviseur et détermination des bits successifs du quotient

Reste à la la $i$ème itération $R^{(i)}$

$$\begin{aligned} R^{(0)} &= Dd \\ R^{(i+1)} &= 2 \times R^{(i)} - q_{n-i+1} \times Dv \end{aligned}$$

$$\begin{aligned} R^{(1)} &= 2 \times R^{(0)} - q_{n-1} \times Dv \\ R^{(2)} &= 2 \times R^{(1)} - q_{n-2} \times Dv \\ &= 2^2 \times R^{(0)} - (2 \times q_{n-1} + q_{n-2}) \times Dv \\ R^{(n)} &= 2^n \times R^{(0)} - (2^{n-1} \times q_{n-1} + 2^{n-2} q_{n-2} + \cdots + q_0) \times Dv \\ 2^{-n} R^{(n)} &= Dd - \underbrace{(2^{-1} \times q_{n-1} + 2^{-2} q_{n-2} + \cdots + 2^{-n} q_0)}_{Q} \times Dv \end{aligned}$$

$$Dd = Q \times Dv + 2^{-n} R^{(n)}$$

ce qui correspond à la définition de la division.

Plan

1. **Division par soustraction-décalage**
   - Division avec restauration
   - Division sans restauration
   - Mise en oeuvre matérielle de la division sans restauration
2. **Division itérative**

Division avec restauration

Mise en oeuvre directe de la formule précédente.
On cherche le plus grand quotient qui maintienne un reste positif.

Algorithme 1 division avec restauration

1 $R^{(0)} \leftarrow D_d$
2 **pour** $i \leftarrow 0$ **à** $n-1$
3 **si** $2R^{(i)} - D \geq 0$
4 $R^{(i+1)} \leftarrow 2R^{(i)} - D$
5 $q_{n-1-i} \leftarrow 1$
6 **sinon**
7 $R^{(i+1)} \leftarrow 2R^{(i)}$
8 $q_{n-1-i} \leftarrow 0$
9 **fin si**
10 **fin pour**
11 $R_{final} = R \times 2^{-n}$

Division avec restauration (cont.)

Mise en oeuvre matérielle :

- suppression des indices
- un registre contient le reste courant

Algorithme 2 division avec restauration

1 $R \leftarrow D_d$
2 **pour** $i \leftarrow 0$ **à** $n-1$
3 $R \leftarrow 2R - D$ ▷ *calcul du nouveau reste en supposant* $q = 1$
4 **si** $R \geq 0$
5 $q_{n-1-i} \leftarrow 1$
6 **sinon**
7 $R \leftarrow R + D$ ▷ *Restauration si reste négatif*
8 $q_{n-1-i} \leftarrow 0$
9 **fin si**
10 **fin pour**
11 $R_{final} = R \times 2^{-n}$

Problèmes : irrégularité suivant les bits et possibilité de deux additions par itérations

Division avec restauration (cont.)

Division de $Dd = 0.011 = 0.375$ par $Dv = 0.101 = 0.625$ (ou $\frac{3/8}{5/8} = \frac{3}{5}$).

Initialement : $R = Dd = 0.011$

$Dv = 0.101$ $-Dv = 1.011$

Etape 1 :

Calcul de $R \leftarrow 2 \times R - Dv$

$$\begin{array}{lcll} & & {\scriptstyle 11\ 10} & \\ 2 \times R & & 0.110 & \\ -Dv & + & 1.011 & \\ R \leftarrow 2 \times R - Dv & = & 0.001 & \geq 0 \end{array}$$

$\Rightarrow \quad q_{-1} \leftarrow 1 \quad R \leftarrow 0.001$

Etape 2 :

Calcul de $R \leftarrow 2 \times R - Dv$

$$\begin{array}{lcll} & & {\scriptstyle 00\ 10} & \\ 2 \times R & & 0.010 & \\ -Dv & + & 1.011 & \\ R \leftarrow 2 \times R - Dv & = & 1.101 & < 0 \end{array}$$

$\Rightarrow \quad q_{-2} \leftarrow 0 \quad R \leftarrow R + Dv = 1.101 + 0.101 = 0.010$

Etape 3 :

Calcul de $R \leftarrow 2 \times R - Dv$

$$\begin{array}{lcll} & & {\scriptstyle 00\ 00} & \\ 2 \times R & & 0.100 & \\ -Dv & + & 1.011 & \\ R \leftarrow 2 \times R + Dv & = & 1.111 & < 0 \end{array}$$

$\Rightarrow \quad q_{-3} \leftarrow 0 \quad R \leftarrow R + Dv = 1.111 + 0.101 = 0.100$

$Q = 0.100 = 0.5, R = 0.100 \times 2^{-3} = 0.0625$ $(0.375 = 0.625 \times 0.5 + 0.0625)$

Plan

1. **Division par soustraction-décalage**
   - Division avec restauration
   - Division sans restauration
   - Mise en oeuvre matérielle de la division sans restauration
2. **Division itérative**

Division sans restauration

Suppression de l'étape de restauration.
Le reste R peut devenir négatif.
Le signe de R détermine le quotient et l'opération à effectuer
Quotient codé *en digits signés* : $q_i \in \{-1, +1\}$

Algorithme 3 division sans restauration

▷ *On suppose* $D > 0$
1 $R \leftarrow D_d$
2 **pour** $i \leftarrow 0$ **à** $n-1$
3 **si** $R \geq 0$
4 $R \leftarrow 2R - D$
5 $q_{n-1-i} \leftarrow +1$
6 **sinon**
7 $R \leftarrow 2R + D$
8 $q_{n-1-i} \leftarrow -1$
9 **fin pour**
10 **si** $R < 0$
11 $R \leftarrow R + D$ ▷ *Correction d'un reste négatif*
12 $Q \leftarrow Q - 2^{n-1}$ ▷ *Enlever 1 au bit de poids faible de Q*
13 **fin si**
14 $R_{final} = R \times 2^{-n}$

Division sans restauration (cont.)

Si un bit à -1 du quotient (parfois noté $\bar{1}$) est codé par un bit à 0,
on retrouve la valeur du quotient par :
$Q = \sum_{i=n-1}^{0} 2^i \times q_i - \sum_{i=n-1}^{0} 2^i \times \overline{q_i}$

Certaines valeurs du quotient ne sont pas codables avec des ± 1, par exemple si le LSB du quotient doit être à 0.
Dans ce cas, le dernier reste sera < 0

Pour obtenir un résultat avec $R \geq 0$:

- on remplace $q_0 = 1$ par $q_0 = 0$ (et donc on retranche 2^{-n} au quotient)
- on corrige le dernier reste en ajoutant D (car la dernière opération aurait du être $R \leftarrow 2R - q_0 \times D$ avec $q_0 = 0$ et non $q_0 = +1$)

Division sans restauration (cont.)

Division de $Dd = 0.011 = 0.375$ par $Dv = 0.101 = 0.625$.
Initialement : $R = Dd = 0.011$ $Dv = 0.101$ $-Dv = 1.011$

Etape 1:
$R \geq 0 \rightarrow$ Calcul de $R \leftarrow 2 \times R - Dv$

		$^{11\ 10}$	
$2 \times R$		0.110	
$-Dv$	$+$	1.011	
$R \leftarrow 2 \times R - Dv$	$=$	0.001	≥ 0

$\Rightarrow \quad q_{-1} \leftarrow 1 \quad R \leftarrow 0.001$

Etape 2 :
$R \geq 0 \rightarrow$ Calcul de $R \leftarrow 2 \times R - Dv$

		$^{00\ 10}$	
$2 \times R$		0.010	
$-Dv$	$+$	1.011	
$R \leftarrow 2 \times R - Dv$	$=$	1.101	< 0

$\Rightarrow \quad q_{-2} \leftarrow 1 \quad R \leftarrow 1.101$

Etape 3 :
$R < 0 \rightarrow$ Calcul de $R \leftarrow 2 \times R + Dv$

		$^{00\ 00}$	
$2 \times R$		1.010	
$+Dv$	$+$	0.101	
$R \leftarrow 2 \times R - Dv$	$=$	1.111	< 0

$\Rightarrow \quad q_{-3} \leftarrow \bar{1} \quad R \leftarrow 1.001$

...

Division sans restauration (cont.)

...

Soit un résultat final
$Q = 0.11\bar{1} = 0.110 - 0.001 = 0.110 + 1.111 = 0.101 = 0.625$
$R = 1.111 \times 2^{-3} = -0.001 \times 2^{-3} = -0.015625$
$Dd = 0.375 = Dv \times Q + R = 0.625 \times 0.625 - 0.015625$

Mais, comme $R < 0$, une correction du résultat est nécessaire :

1/ on soustrait 2^{-n} à Q
$Q \leftarrow Q - 0.001 = 0.101 - 0.001 = 0.100$
2/ on ajoute D à R
$R \leftarrow R + D = 1.111 + 0.101 = 0.100$

Finalement $D = 0.100 = 0.5_d$ et $R = 0.100 \times 2^{-3} = 0.0625_d$
(identique au cas précédent)

Plan

1. **Division par soustraction-décalage**
   - Division avec restauration
   - Division sans restauration
   - Mise en oeuvre matérielle de la division sans restauration
2. **Division itérative**

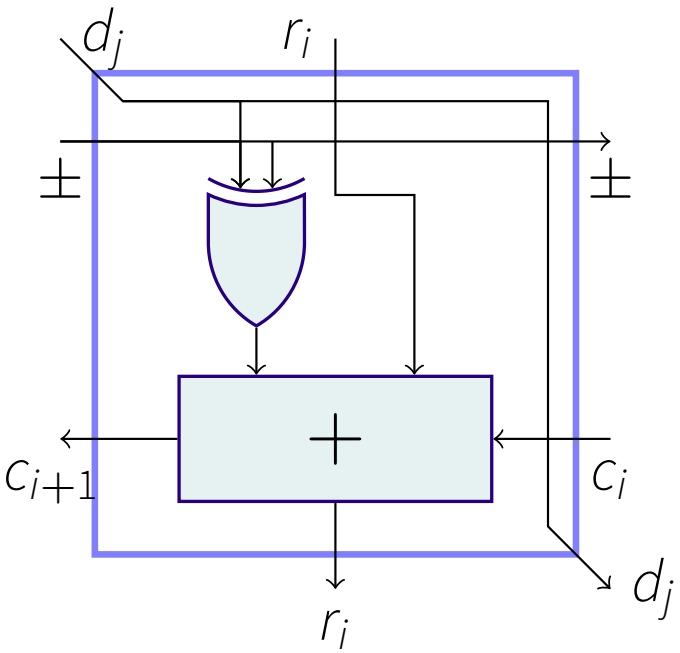
Mise en oeuvre matérielle

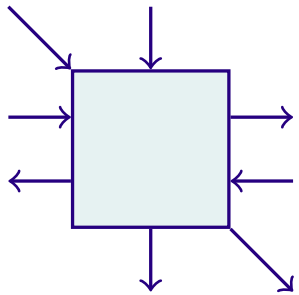
Opération de base de la division sans restauration :
addition/soustraction suivant le signe de R

Contrairement à la multiplication, il n'est *pas* possible d'utiliser des additionneurs à sauvegarde de retenue, car le signe du résultat est requis.

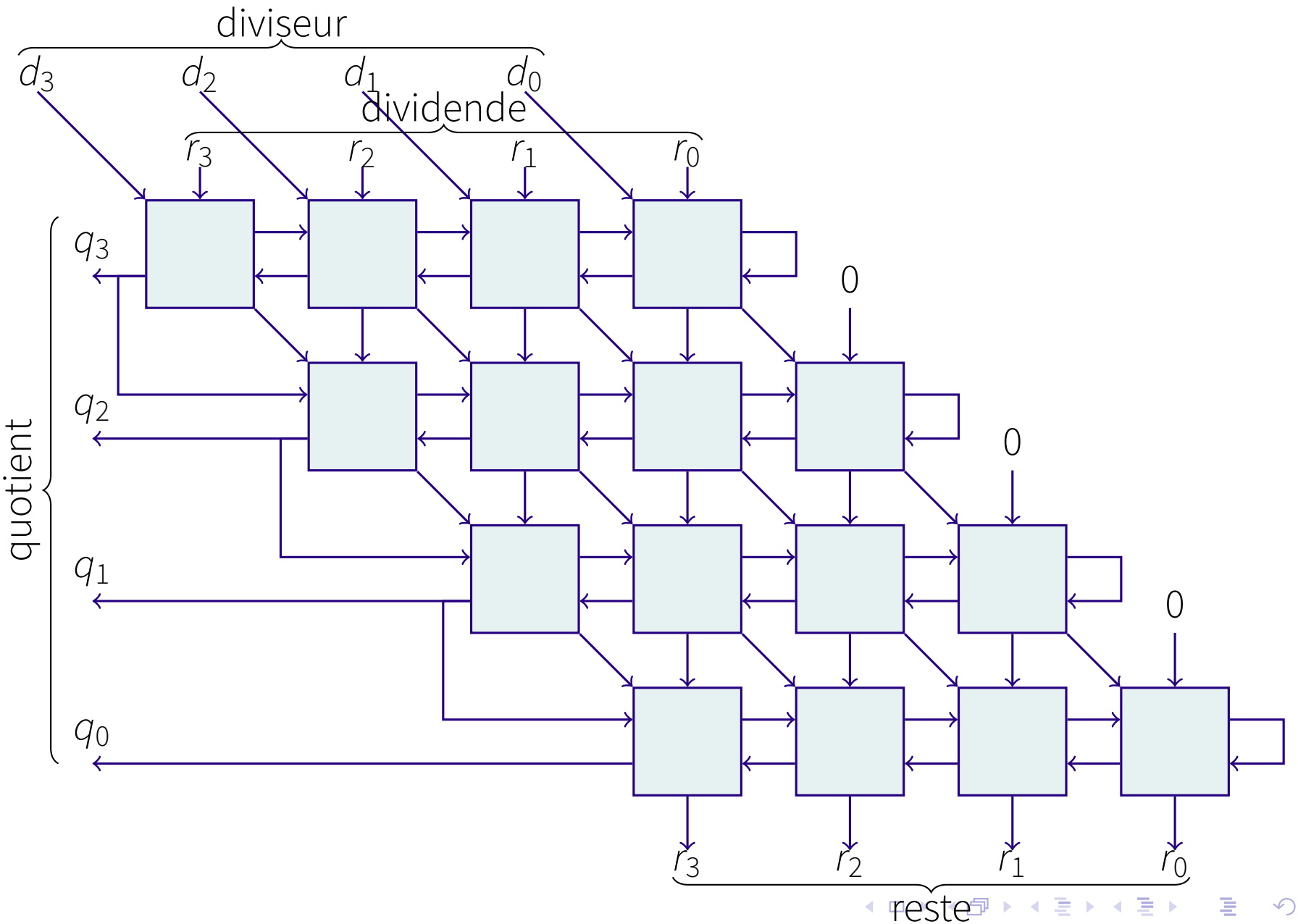
Utilisation d'additionneur/soustracteur à propagation de retenue

Cellule de base :





Mise en oeuvre matérielle (cont.)



Mise en oeuvre matérielle (cont.)

Principal problème de cette approche : l'addition à propagation prend un temps n et le temps total est en n^2.

Les diviseurs des processeurs actuels utilisent une méthode dérivée SRT (Sweeney, Robertson, Tocher).

- le calcul est fait en base 4 ou 8
- les additions/soustractions sont à sauvegarde de retenue
- on cherche à prédire le signe du reste avec les bits de poids fort des opérandes
  si la prédiction est incorrecte, la méthode permet quand même de retrouver le bon résultat aux itérations suivantes

La division est une opération coûteuse : 10–30 cycles sur les processeurs actuels

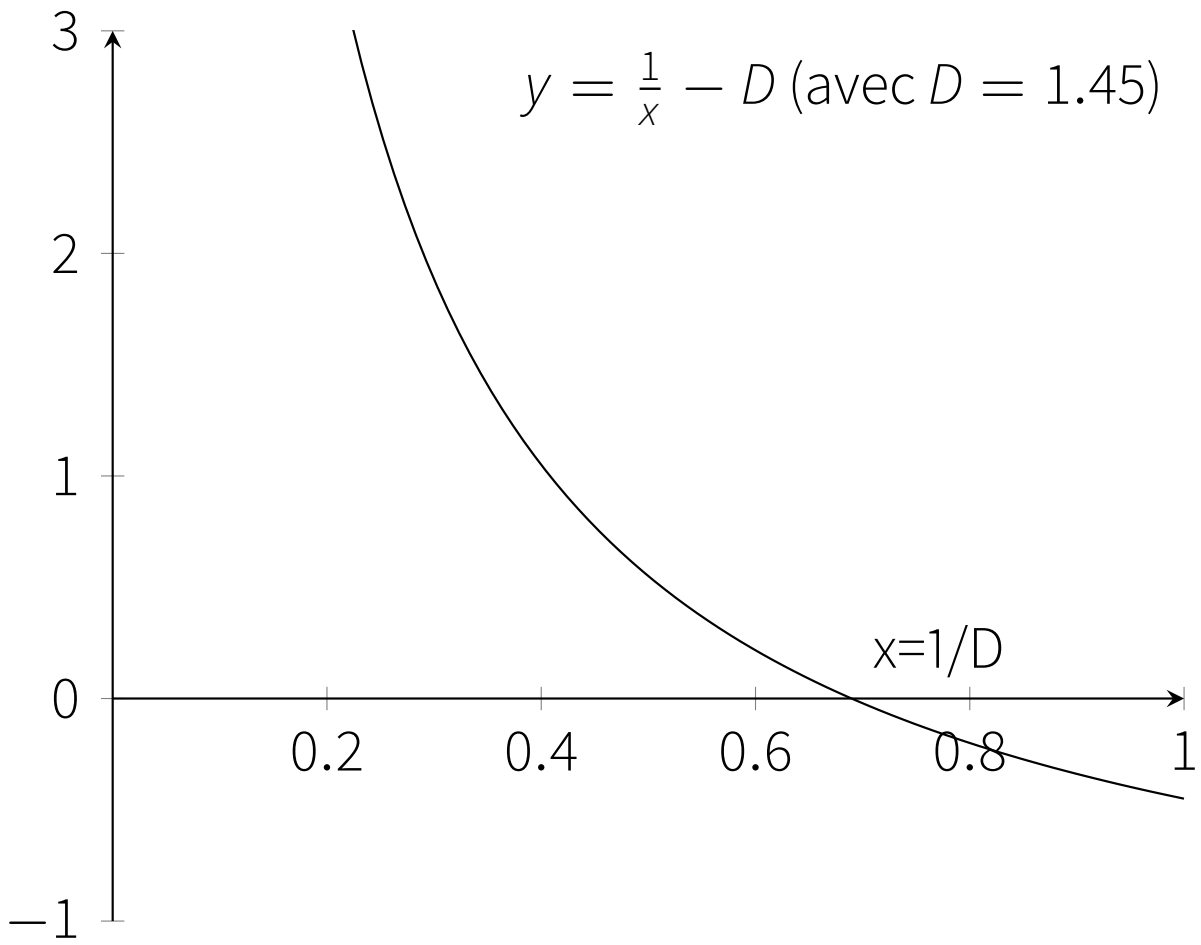
Plan

1. Division par soustraction-décalage
   - Division avec restauration
   - Division sans restauration
   - Mise en oeuvre matérielle de la division sans restauration
2. **Division itérative**

Division itérative

Utilisation de méthodes d'analyse numérique.

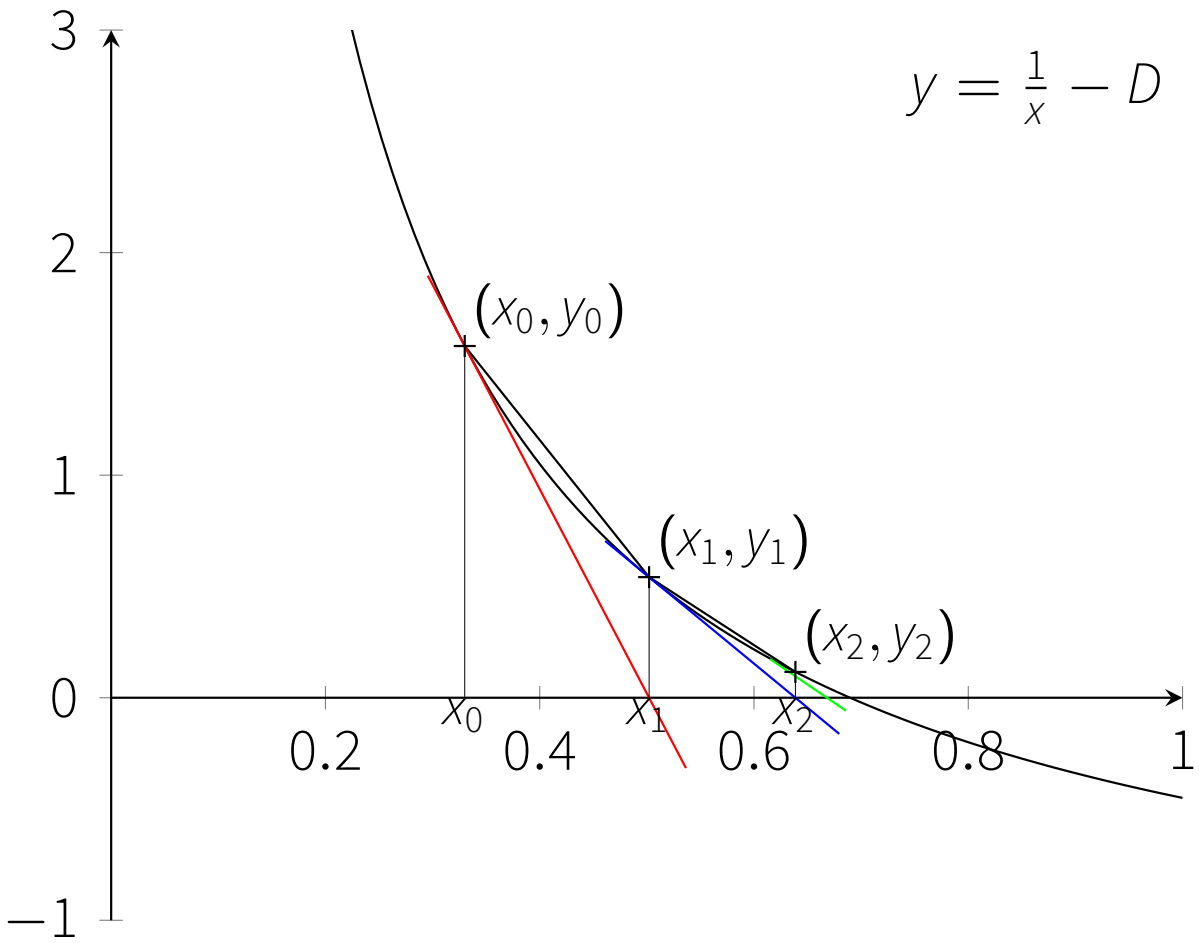
On cherche le point où la fonction $y = \frac{1}{x} - D$ passe par 0.



Pour calculer $N \div D$, il suffit ensuite de multiplier N par $1/D$.

Division itérative (cont.)

Utilisation de méthodes itératives pour trouver le « zéro » de $f = 1/x - D$.



1. on part d'une approximation initiale de la solution x_0
2. autour du point (x_0, y_0), on approxime f par sa tangente
3. on cherche le point x_1 d'intersection de la tangente avec l'axe des x
   x_1 est une meilleure approximation de la solution
4. on itére 2–3 pour trouver les points x_2, x_3, x_4, etc...

Division itérative (cont.)

Equation de la tangente en x_0

$$y = f(x_0) + f'(x_0)(x - x_0)$$

La solution de cette équation en $y = 0$ donne une nouvelle approximation de la solution x_1

$$x = x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

On itére un certain nombre de fois pour trouver la solution de $f(x) = 0$ (méthode de Newton-Raphson).

Pour calculer l'inverse $f(x) = 1/x - D$ qui s'annule en $x = 1/D$.
Nous supposons que $D \in [0.5, 1]$ et donc $1/D \in [1, 2]$.
$f'(x) = -1/x^2$
itération de Newton-Raphson :

$$\begin{aligned} x_{i+1} &= x_i - f(x_1)/f'(x_i) \\ &= x_i - \frac{1/x_i - D}{-1/x_i^2} \\ &= x_i \times (2 - D \times x_i) \end{aligned}$$

Le calcul de x_{i+1} ne nécessite que + et $\times$ $\Longrightarrow$ utilisation possible pour calculer $1/D$

Nous supposons $x_i = \frac{1}{D}(1+\epsilon_i)$.
ϵ_i l'erreur relative algébrique quand on approxime $\frac{1}{D}$ par x_i.
A l'ordre $i+1$,

$$\begin{aligned} x_{i+1} &= x_i(2 - Dx_i) \\ &= \frac{1}{D}(1+\epsilon_i)(2 - D \times \frac{1}{D}((1+\epsilon_i)) \\ &= \frac{1}{D}(1-\epsilon_i^2) \\ \epsilon_{i+1} &= -\epsilon_i^2 \end{aligned}$$

Erreur ϵ_i

- réduite de manière *quadratique*
- toujours négative (approximation par défaut).

Si $|\epsilon_i| = 2^{-k}$ (k bits significatifs), $|\epsilon_{i+1}| = 2^{-2k}$ ($2k$ bits significatifs).
Nombre de bits significatifs doublé à chaque étape,
Nombre maximal d'étapes pour obtenir n bits de précision est $\mathbf{log}_2\, n$.

Division itérative (cont.)

approximation initiale

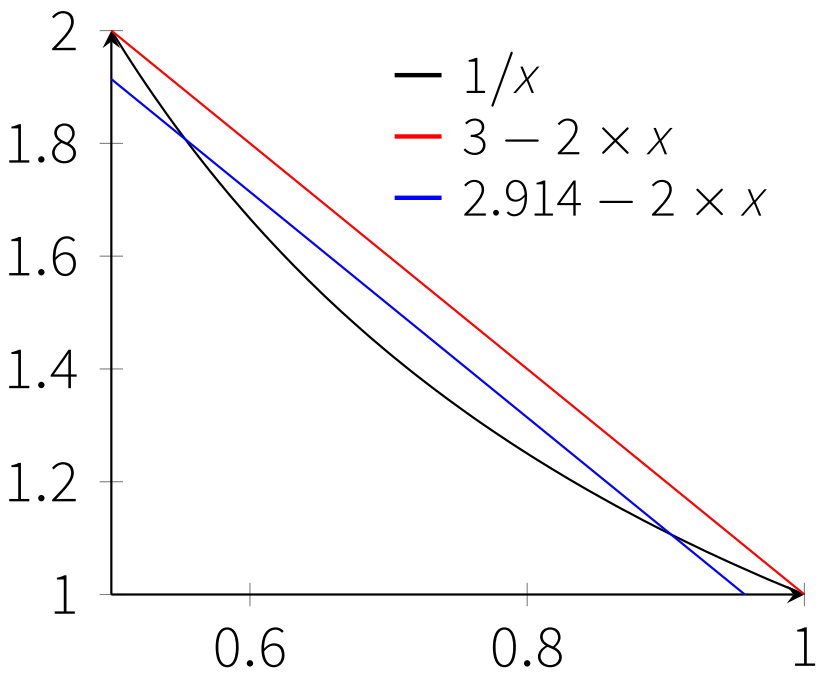
Deux méthodes pour obtenir l'approximation initiale :

valeurs tabulées (*look-up table*)

On approxime la fonction $1/D$ par un tableau avec valeur précalculées

En partant, par exemple, d'une table des inverses sur 4 bits, 32 bits de précision en 3 étapes

Approximation par une droite de la fonction $1/x$ dans l'intervalle considéré ($D \in [0.5, 1]$)



La droite qui passe par $(0.5, 2)$ et $(1, 1)$ a pour équation $y = 3 - 2 \times x$

Une meilleure approximation est $y = k - 2 \times x$ avec $k = 2.914\ldots$ pour minimiser l'erreur d'approximation.

On peut également utiliser des polynomes d'ordre 2.

Conclusion

La division itérative permet d'obtenir de bonnes performances sans diviseur matériel.
Mais il est difficile de respecter les contraintes de la norme IEEE-754 avec cette méthode (à cause des arrondis dans le calcul de $Dd \times (1/Dv)$).
De ce fait, elle n'est plus utilisée dans les processeurs standard, mais c'est une des meilleures manières d'effectuer une division, par exemple sur un FPGA.

Une autre méthode itérative (division de GoldSchmidt) est utilisée dans les processeurs AMD et elle permet de calculer directement Dd/Dv.

Mais la plupart des processeurs actuels (pentium, arm, etc) ont un diviseur basé sur la méthode SRT.

Opérations flottantes

Alain MÉRIGOT

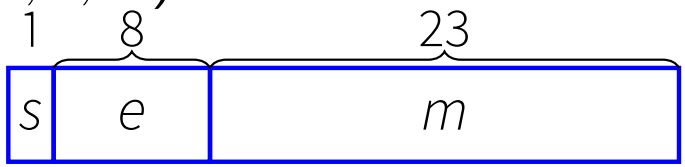
Université Paris sud

Nombres flottants

Rappels :

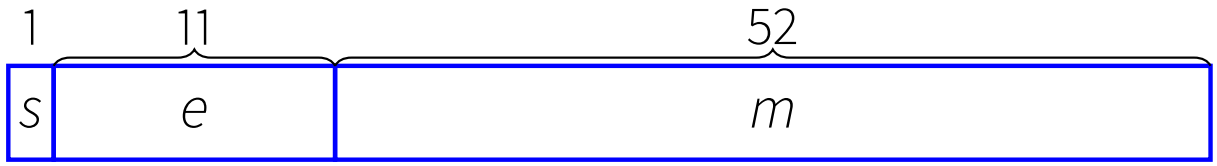
Nombre flottant = triplet (s, e, m)

En simple précision F :



$$F = -1^s \times 2^{e-127} \times 1.m$$

En double précision F :



$$F = -1^s \times 2^{e-1023} \times 1.m$$

Le bit en dernière position dans la mantisse est appelé *ulp* (*unit in the last place*).
Il détermine la *précision* des nombres (2^{-23} en SP, 2^{-52} en double)

Plan

1. Arrondi des nombres flottants
2. Addition flottante
3. Multiplication flottante

Arrondi des nombres flottants

Le résultat d'un calcul doit être conforme à la norme IEEE-754
Les bits excédentaires seront supprimés : *arrondi du résultat*

La norme IEEE prévoit 4 modes d'arrondis :

- vers zéro : on supprime les bits à droite de l'ulp
- vers plus l'infini : on supprime les bits à droite de l'ulp et, si le résultat est positif et que certains bits supprimés étaient à 1, on ajoute 1 à l'ulp.
- vers moins l'infini : on supprime les bits à droite de l'ulp et, si le résultat est négatif et que certains bits supprimés étaient à 1, on ajoute 1 à l'ulp.
- au plus proche.

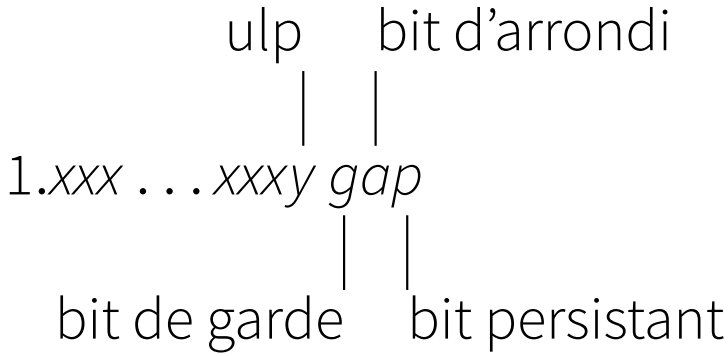
Exemple : arrondi de 1.788 sur deux digits décimaux

	vers 0	vers $+\infty$	vers $-\infty$	au plus proche
1.788	1.78	1.79	1.78	1.79
-1.788	-1.78	-1.78	-1.79	-1.79

Arrondi des nombres flottants (cont.)

arrondi au plus proche

Le cas des arrondis au plus proche implique de garder 3 bits à droite de l'ulp.
Ces bits sont nommés *g* (bit de *garde*), *a* (bit d'*arrondi*) et *p* (bit *persistant*).



Les bits de *garde* et d'*arrondi* nécessitent d'effectuer le calcul sur un nombre de bits suffisant.
Le bit *persistant* est particulier.
$p = 1$, si un des bits à sa droite est à 1.
Permet de discrimer les cas d'arrondi proche de 0.5ulp

Arrondi des nombres flottants (cont.)

Tableau pour déterminer si l'arrondi se fait

- par troncation des bits excédentaires
- ou s'il faut en plus ajouter 1 à l'ulp

bit de garde	bit d'arrondi	bit persistant	valeur numérique	résultat arrondi
0	x	x	$0.gap < 0.5$ ulp	troncation
1	1	x	$0.gap \geq 0.75$ ulp	ajouter 1 à l'ulp
1	0	1	$0.gap = 0.5$ ulp $+ \epsilon$	ajouter 1 à l'ulp
1	0	0	$0.gap = 0.5$ ulp	vers la valeur paire

Pour éviter de faire une erreur d'arrondi systématiquement dans le même sens dans le cas où $gap = 100$ ($O.5$ ulp)
$\rightarrow$ arrondi « vers la valeur paire » (*ties to even*) de telle sorte que le résultat arrondi ait 0 en ulp.

- si ulp$=0$, arrondi vers la valeur inférieure (troncation)
- si ulp$=1$, arrondi vers la valeur immédiatement supérieure en ajoutant $+1$ à l'ulp.

Arrondi des nombres flottants (cont.)

Mode d'arrondi par défaut en C/C++ :

- au plus proche pour les calculs et les conversions `double` → `float`
- vers zéro pour les conversions flottant vers entier (`(int)2.9f` → 2, `(int)-3.6` → −3)

L'arrondi fait que les calculs flottants ne sont **ni associatifs, ni distributifs**

- $(a+b)+c \neq a+(b+c)$, car, $(a+b)+c \equiv$ `round(round(a+b)+c)` ce qui peut être différent de `round(a+round(b+c))` $(a+(b+c))$.
- $a \times (b+c) \neq a \times b + a \times c$ car `round(a*round(b+c))` peut être différent de `round(round(a*b)+round(a*c))`.

De même, certaines relations mathématiques peuvent ne pas être vérifiées ($a \times 1/a \neq 1.0$, $\sqrt{a \times a} \neq a$, etc)

A cause des arrondis, il est *fortement* déconseillé de tester dans un programme l'égalité de deux nombres flottants. Il est préférable de tester si la valeur absolue de leur différence est *suffisamment petite*.

Plan

1. Arrondi des nombres flottants
2. **Addition flottante**
3. Multiplication flottante

Addition flottante

Soit à additionner deux nombres flottants normalisés $A = (s_a, e_a, m_a)$ et $B = (s_b, e_b, m_b)$ et le résultat $A + B = (s, e, m)$
Soit n_b le nombre de bits des mantisses.

Différentes étapes du calcul :

1. Permuter les nombres A et B si $e_a < e_b$ pour que A soit le nombre de plus grand exposant. e_a est (temporairement) l'exposant e du résultat.
2. Faire apparaitre les bits cachés de m_a et m_b ($n_b + 1$ bits)
3. Mettre les mantisses en complément à deux.
   Comme $-2 < m_a, m_b < +2$, $-4 < m_a + m_b < +4$, le résultat en C_2 et les opérandes nécessitent 3 bits à gauche de la virgule. ($n_b + 3$ bits)
4. Décaler *arithmétiquement* m_b de $e_a - e_b$ pas vers la droite.
   Pour faire l'arrondi, on doit garder les 3 bits *gap* à droite de l'ulp pour B et rajouter 3 zéros à A ($n_b + 6$ bits)
   Le bit p de B est obtenu en faisant le **OU** des bits éliminés

Addition flottante (cont.)

5. Ajouter les mantisses

   Repasser en forme signe/valeur absolue.

   Le résultat est la somme S.

6. Renormaliser S dans les cas suivants :

   a) si $2.0 \leq |S| < 4.0$, décaler S d'un cran vers la droite
   Ajouter 1 à e.
   (Peut arriver si A et B sont de même signe)

   b) si $|S| < 1.0$, chercher la position p du premier 1 de S.
   Décaler S de p pas vers la gauche.
   Soustraire p à e.
   (Peut arriver si A et B sont de signe contraire).

   Dans les deux cas, mettre à jour les bits pour les arrondis.

7. Arrondir le résultat pour le repasser à n_b bits, par troncation ou en ajoutant 1 à l'ulp si nécessaire.

8. Dans le cas d'addition d'un 1 à l'ulp, vérifier si le résultat n'atteint pas 2.0 et le renormaliser dans le cas contraire.

Addition flottante (cont.)

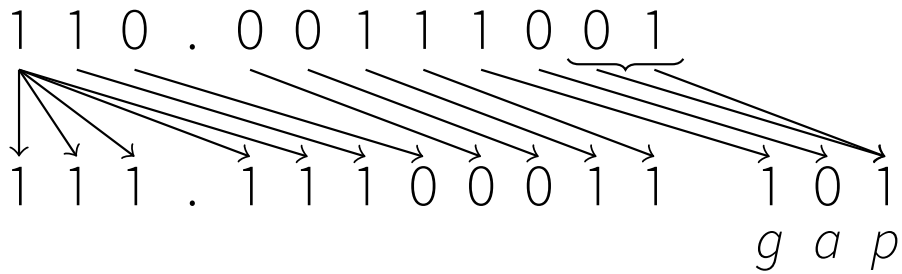
Exemple (mantisse 8 bits) :

$A = 1.10110011 \times 2^1 \quad B = -1.11000111 \times 2^{-3} \quad e_a - e_b = 4$

Passage en C_2 sur $n_b + 3$ bits.

$A = 001.10110011 \times 2^1 \quad B = 110.00111001 \times 2^{-3}$

Passage à $n_b + 6$ bits avec décalage de B de 4 bits vers la droite



Calcul de la somme

$$\begin{array}{rcl} m_a & & 0\,0\,1\,.\,1\,0\,1\,1\,0\,0\,1\,1 \quad 0\,0\,0 \\ m_b & + & 1\,1\,1\,.\,1\,1\,1\,0\,0\,0\,1\,1 \quad 1\,0\,1 \\ \hline m & = & 0\,0\,1\,.\,1\,0\,0\,1\,0\,1\,1\,0 \quad 1\,0\,1 \end{array}$$

Arrondi : $gap = 101 \implies$ ajouter 1 à l'ulp

$001.10010110 + 000.00000001 = 001.10010111$

Résultat final après passage en signe-valeur absolue

$A + B = +1.10010111 \times 2^1$

Plan

1. Arrondi des nombres flottants
2. Addition flottante
3. **Multiplication flottante**

Multiplication flottante

Soit à $A \times B$ avec A et B flottants normalisés $A = (s_a, e_a, m_a)$ et $B = (s_b, e_b, m_b)$ et le résultat $A \times B = (s, e, m)$
Soit n_b le nombre de bits des mantisses.

1. Faire apparaitre les bits cachés des mantisses (pour nombres normalisés)
2. Multiplier les mantisses m_a et m_b $(m = m_a \times m_b)$.
   Additionner les exposants e_a et e_b $(e = e_a + e_b)$.
   Calculer le signe du résultat $(s = s_a \oplus s_b)$.
3. si $2.0 \leq m < 4.0$,
   décaler m d'un cran vers la droite, ajouter 1 à e.

   si un des nombres est dénormalisé, on peut avoir $m < 1.0$
   Il faut alors renormaliser m et e comme pour l'addition.
4. Tous les calculs ont été effectués sur $2n_b$ bits.
   Arrondir le résultat suivant les méthodes générales, et ajouter $+1$ à l'ulp si nécessaire.
5. Détecter les débordements éventuels (résultat à 2.0 après ulp+1).

Généralités sur les architectures des ordinateurs

Alain MÉRIGOT

Université Paris Saclay

Constituants d'un ordinateur

Un ordinateur comprend trois parties principales pour :

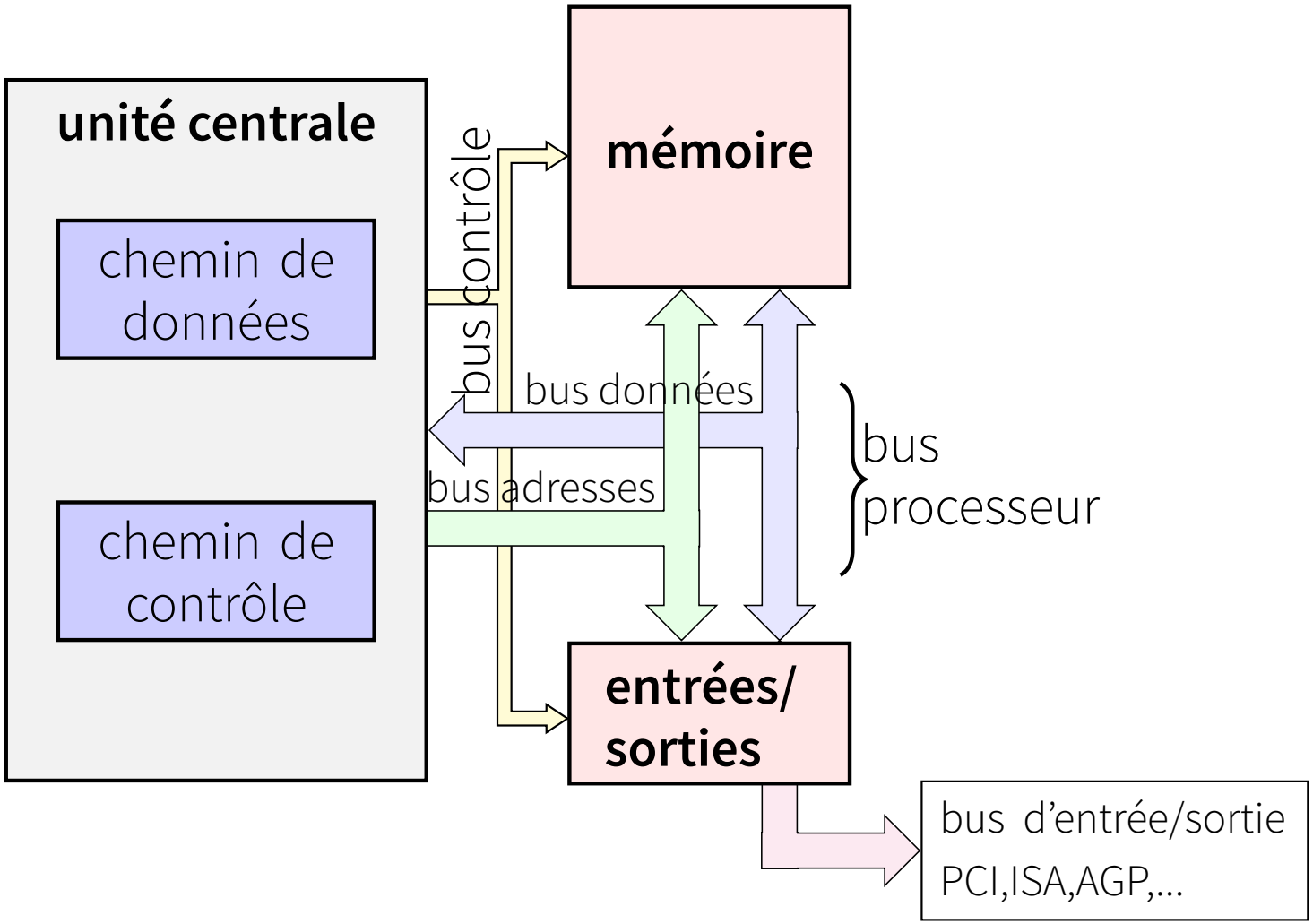
- *traiter* les informations
- *mémoriser*
- et *communiquer* avec l'extérieur.

L'*unité centrale* (UC ou CPU (*Central Processing Unit*)) ou simplement *processeur* transforme les *données* suivant des directives appelées *instructions*.

Ces informations (données et instructions) sont stockées dans l'organe de mémorisation : la *mémoire*

La communication avec l'extérieur (ou les *périphériques*) se fait par le biais des organes d'entrée/sortie.

Constituants d'un ordinateur (cont.)



Constituants d'un ordinateur (cont.)

Ces composants communiquent au moyen du *bus de communication* ou *bus processeur*
Il comprend :

- Le *bus de données* support bidirectionnel sur lequel les informations circulent
- Le *bus d'adresse* qui permet de différencier les informations concernées (emplacement mémoire, organe d'entrée-sortie)
- Le *bus de contrôle* contient des signaux de service pour ces échanges (horloge, sens de transfert mémoire lecture/écriture, indications pour les entrée-sortie, etc.)[1]

[1] On distingue également les **bus d'entrée-sortie** (PCI) qui connectent à des contrôleurs spécialisés, gérant eux même des **bus périphériques** (SATA, IDE, USB, etc).

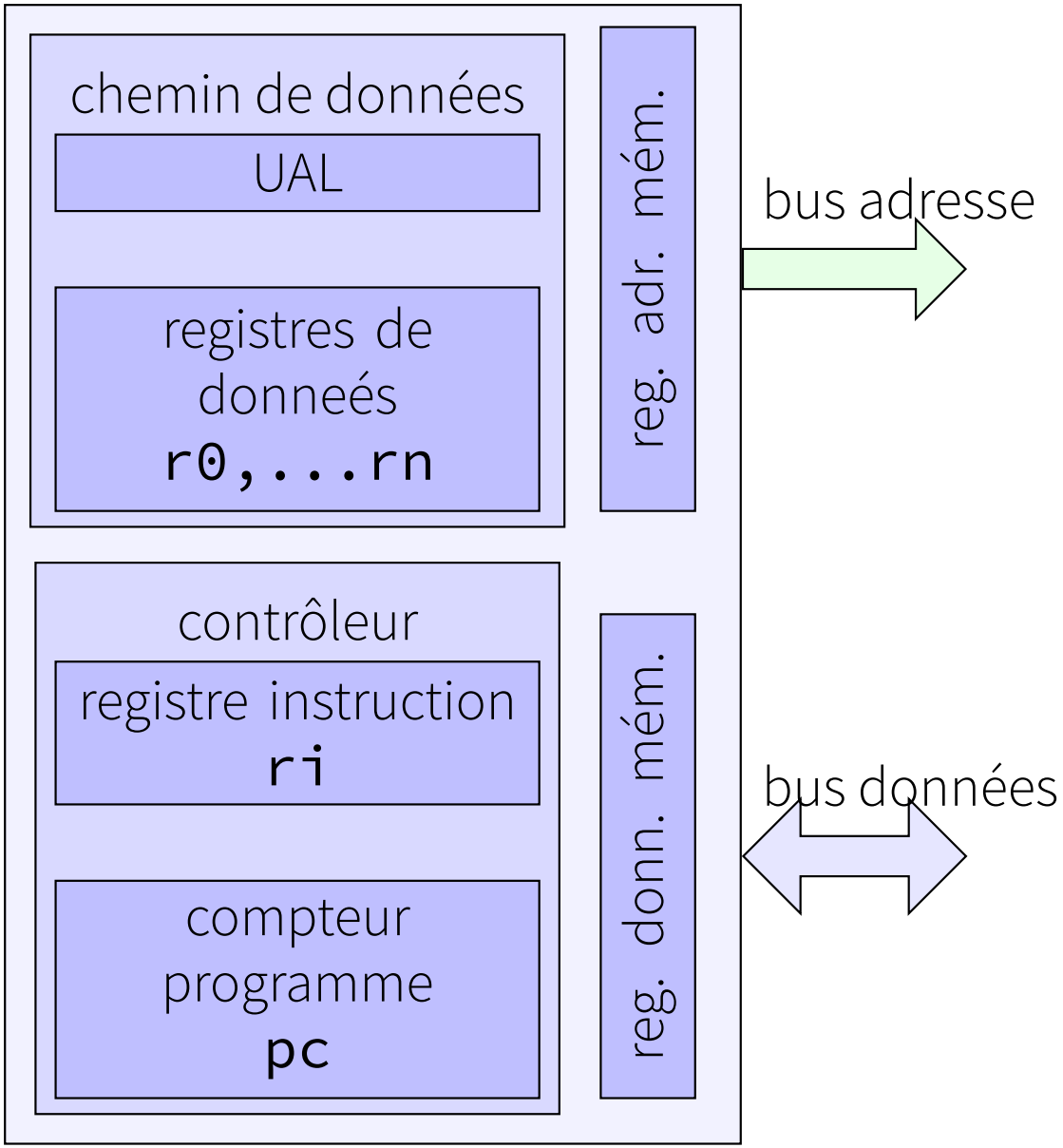
L'unité centrale

Comprend deux parties principales :

- le *séquenceur* (ou *contrôleur*), qui dirige l'exécution des instructions. Il comprend
  - un automate réalisant le séquencement des opérations
  - des registres contenant des informations sur l'instruction en cours d'exécution :
    - registre d'instruction `ri` (contenu de l'instruction)
    - compteur de programme `pc` (adresse en mémoire de l'instruction à exécuter)
- le *chemin de données* qui permet de transformer les données. Il comprend généralement :
  - une *unité arithmétique et logique*
  - des *registres de données* (nommés par exemple `r1`, `r2`, etc)

D'autres registres particuliers ou des registres stockant temporairement l'information (*registres tampons*) peuvent également exister suivant les architectures des processeurs.

L'unité centrale (cont.)



Principaux organes d'une unité centrale

Les instructions d'un processeur

Deux grands types d'instructions :

instructions de modification des données modification de l'état des données stockées dans le processeur et/ou dans la mémoire (calculs, déplacements de données entre la mémoire et le processeur, etc.) *Une fois l'instruction i exécutée, on passe à l'instruction suivante* (celle à l'adresse $i + 1$, ou `pc` +1, `pc` étant le compteur de programme du processeur).

instructions de contrôle de flot Permettent de faire des ruptures du flot d'exécution du programme :

- tests (*if ... then ... else ...*, *switch*)
- boucles (*for*, *while*)
- appels/retour de fonctions

On modifie le compteur de programme `pc`.

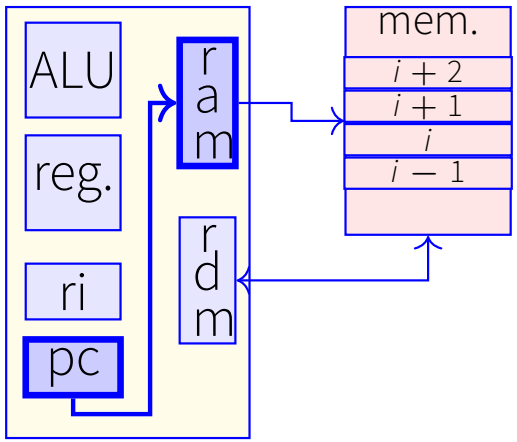
Exécution d'une instruction

Plusieurs actions à réaliser.

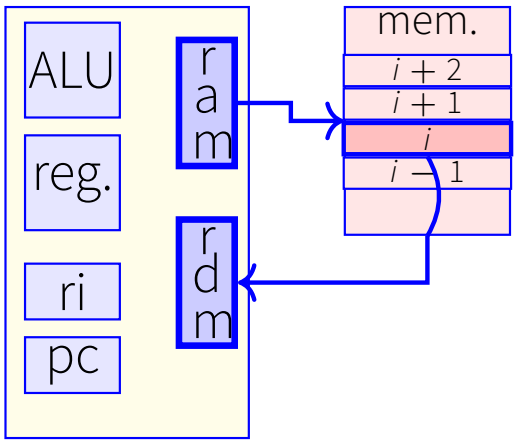
1. *Acquisition de l'instruction*. On va chercher le contenu de la case mémoire dont l'adresse est `pc` et on le copie dans le registre d'instruction `ri`.
2. *Exécution de l'instruction*. On *décode* le contenu du registre `ri`, et on réalise les mouvements de données correspondants à l'instruction.
3. *Préparation de l'instruction suivante*. On met à jour le contenu du compteur de programme `pc`
   - pour une instruction de *modification des données*, on remplace le contenu de `pc` par `pc+1`
   - pour une instruction de *contrôle de flot*, on calcule le nouveau `pc` en fonction de l'instruction.

Exécution d'une instruction (cont.)

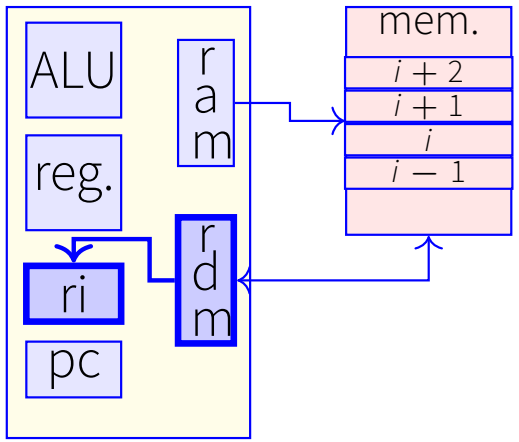
Phase d'acquisition



Transfert de l'adresse de l'instruction à exécuter (`cp`) vers le registre d'adresse mémoire (`ram`).

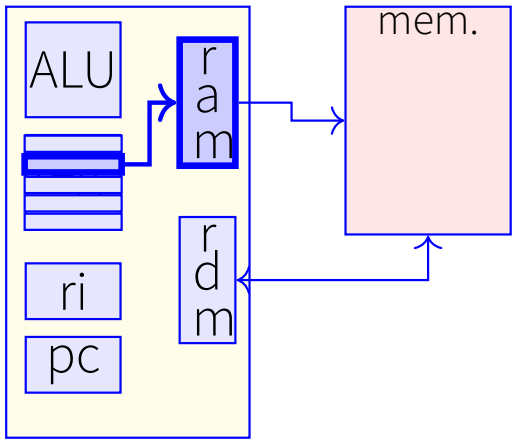


Lecture de la mémoire et recopie du contenu de la case adressée par le registre `ram` dans le registre de données mémoire (`rdm`)

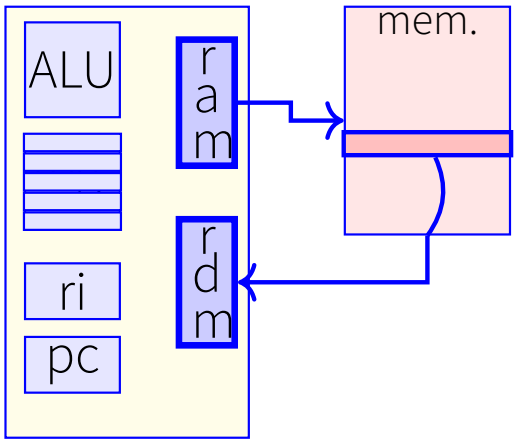


Recopie de l'instruction (dans le registre `rdm`) vers le registre d'instruction `ri`.

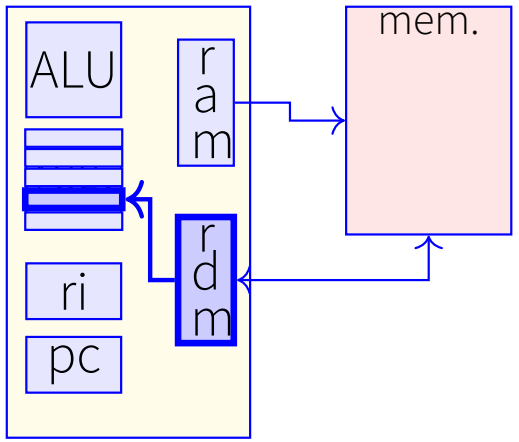
Exemple de l'instruction : `ld r4, (r2)` (registre `r4` ← mémoire[`r2`])



copier `r2` dans le registre `ram`



lire la case mémoire pointée par le registre `ram` et copier le contenu dans le registre `rdm`



copier le contenu du registre `rdm` vers le registre destination `r4`

Grandes familles de processeurs

L'efficacité d'un ordinateur pour exécuter un programme dépend de divers facteurs :

- nombre d'instructions exécutées
- rapidité d'exécution d'une instruction (nombre de cycles)
- nombre d'accès à la mémoire
- fréquence de fonctionnement
- etc

Elle dépend largement des choix du *jeu d'instructions*.

Grandes familles de processeurs (cont.)

Premiers processeurs (1950–1980) :

- programmés principalement en assembleur
- on favorise un jeu d'instruction contenant des instructions sophistiquées : processeurs CISC (*complex instruction set computer*) (IBM370, vax, x86 (intel), 68000 (motorola), etc)

Les instructions CISC peuvent :

- comporter des calculs avec opérandes/résultat en mémoire
- manipuler des zones mémoires de taille quelconque (copie d'une zone mémoire, recherche d'un caractère dans une chaîne, etc).
- itérer une ou plusieurs instructions
- être codées par un nombre quelconque d'octets

Ces caractéristiques posent de nombreux problèmes, sont peu utilisées par les compilateurs et interdisent une mise en oeuvre architecturale efficace.

Grandes familles de processeurs (cont.)

A partir des années 80 :

- la quasi totalité de la programmation se fait en langage de haut niveau
- on favorise un jeu d'instruction simple et efficace, l'optimisation étant faite par le compilateur. Processeurs RISC (*Reduced Instruction Set Computer*) (MIPS, SPARC, PowerPC, Arm, RISC-V, Nios, etc).
  - les instructions de calcul ne manipulent que des registres
  - les accès mémoire ne sont pas associés à des calculs (jeu d'instructions *load–store*)
  - les instructions sont de taille identiques et *alignées* en mémoire

Cette simplification permet de mettre en oeuvre des mécanismes architecturaux accélérant *largement* les traitements : pipeline (lancement d'une instruction/cycle), superscalaire (plusieurs instructions parallèles), etc... Tous les processeurs actuels reposent sur des mécanismes RISC[2].

[2] Le pentium a un jeu d'instructions complexe (appelé x86 ou IA-32), mais un coeur de traitement à base d'instructions RISC (μops)

Caractéristiques des instructions

Adressage des opérandes

Adressage = spécification d'un opérande (registre ou mémoire)

mode	exemple	interprétation	utilisation
registre	`add r1,r2,r3`	`r1←r2+r3`	données en registre
immédiat	`add r1,r2,#5`	`r1←r2+5`	constante entière
direct	`add r1,r2,(1234)`	`r1←r2+mem[1234]`	var. globale (statique)
ind. registre	`add r1,r2, (r3)`	`r1←r2+mem[r3]`	déréférence de pointeur
basé	`add r1,r2,16(r3)`	`r1←r2+mem[r3+16]`	accés aux variables locales. base = pointeur de pile. (calculé à la compilation)
indexé	`add r1,r2,(r3+r4)`	`r1←r2+mem[r3+r4]`	accés systématique aux éléments d'un tableau
ind. mém.	`add r1,r2, @(r3)`	`r1←r2+mem[mem[r3]]`	déréférence de pointeur
étendu	`add r1,r2,4(r3)[r4]`	`r1←r2+mem[4+r3+r4]`	
post-incrémenté	`add r1,r2,(r3)+`	`r1←r2+mem[r3]` `r3←r3+1`	parcours de tableau ou accès a une pile
pré-décrémenté	`add r1,r2,−(r3)`	`r3←r3−1` `r1←r2+mem[r3]`	idem

Caractéristiques des instructions (cont.)

Adressage des opérandes

Tous les types d'adressage n'ont pas la même utilité :

- Registre : indispensable (50% des instructions au moins). Permet des calculs entre registres du processeur.

  Pour les instructions restantes, sont utiles :

- Immédiat : indispensable pour des instructions contenant des constantes (`x=5; if(x>2)...; i++,` etc) (20–40% des instructions)
- Basé : permet notamment la manipulation des données dans la pile (arguments de fonction, variables locales), les accès aux champs d'un *struct*, etc. Très utile (large majorité des instructions d'accès à la mémoire)
- Indirect-registre : cas particulier d'un adressage basé avec base nulle. Indispensable pour déréférencer un pointeur (`*x`)
- Les autres instructions sont moins utiles et rarement présentes dans les processeurs RISC

Les instructions peuvent manipuler des données de différents types : octets (`char`, double octet (`short`), mot de 32 bit (`int` ou `long`), mot de 64 bits (`long long`), flottant simple ou double précision.

Les données *autres que les octets* sont sujettes à des contraintes :

alignement : adresses des données de plus d'un octet

arrangement des octets d'un mot : ordre des octets des données

Les contraintes dépendent des choix architecturaux.

leur non-respect peut conduire à programmes non portables entre architectures différentes.

Caractéristiques des instructions (cont.)

alignement des données

Alignement : Une donnée de 2^k octets doit être à une adresse multiple de 2^k.
Exemple : un `int` (4o) **doit** être à une adresse multiple de 4.
Un `struct` est aligné sur son plus grand champ.
Pour respecter l'alignement, ajout d'octets de remplissage (*padding*).

```
struct s1 { char c1;   // 1 octet + 3 pour l'alignement de i1
            int  i1;   // 4 octets
            char c2;   // 1 octet + 3 pour l'alignement de i2
            int  i2;   // 4 octets
};  // sizeof(struct s1)==16
struct s2 { int  i1;   // 4 octets
            int  i2;   // 4 octets
            char c1;   // 1 octet
            char c2;   // 1 octet + 2 pour alignement de struct s2
};  // sizeof(struct s2)==12
```

L'alignement assure que tous les octets d'une donnée élémentaire sont
- dans la même rangée mémoire
- dans la même ligne de cache

Toutes les architectures récentes nécessitent des données alignées.
Le pentium peut traiter des données non alignées, mais les performances sont meilleures avec alignement.

Caractéristiques des instructions (cont.)

boutisme

arrangement des octets d'un mot Différents choix sont possibles pour l'emplacement en mémoire d'un mot sur plusieurs octets.

Exemple : `0xfedcba98` à l'adresse `0x1000`

adr.	`1000`	`1001`	`1002`	`1003`	
	`fe`	`dc`	`ba`	`98`	***big-endian*** ou gros-boutiste (poids fort en tête) (SPARC,MC68000, TCP-IP)
	`98`	`ba`	`dc`	`fe`	***little-endian*** ou petit-boutiste (poids faible en tête) (x86, nios)

Sur de nombreux processeurs modernes, choix *au démarrage* du mode petit-boutiste ou gros-boutiste (Arm, power-PC, Mips, etc) (*bi-endian*).

Attention à la portabilité des programmes !!
On peut tester si `__BYTE_ORDER__` vaut `__ORDER_LITTLE_ENDIAN__` ou `__ORDER_BIG_ENDIAN__`

Caractéristiques des instructions (cont.)

boutisme

Des problèmes peuvent apparaitre lors d'équivalence entre pointeurs vers des types différents.

Exemple : extraction des octets de poids faible et de poids fort d'un mot

```
// ***mauvaise*** extraction des LSB et MSB par équivalence de pointeurs
int ii;
char *cc, lsb, msb;
ii = 0xfedcba98;
cc = (char *) &ii; // A éviter !!! Équivalence entre un char* et un int*
lsb = cc[0];
msb = cc[3] ;
// cc[0] et cc[3] ne sont le LSB et MSB qu'en little-endian
// programme non portable (et ne respectant pas le ``strict aliasing'')
```

Mais le boutisme ne concerne que des données *en mémoire*.
Le plus simple et portable est d'utiliser un registre.

```
int ii=0xfedcba98;
char lsb,msb;
// on manipule les données dans un registre : parfaitement portable
lsb=ii&0xff; // ou simplement lsb=ii;
msb=(ii>>24)&0xff ;
```

Instructions de contrôle

Les instruction de contrôle sont de quatre types :

- branchements conditionnels (80–90% des instructions de contrôle)
- sauts inconditionnels
- appels de procédure
- retour de procédure

Les branchements servent à réaliser les tests (`if`) et les boucles (`for`, `while`) des langages de haut niveau. Généralement réalisés par un déplacement *relatif* au compteur de programme.

Ils résultent souvent d'une instruction de comparaison

- instruction explicite de comparaison vers un registre banalisé.
- utilisation d'informations sur les résultats d'opérations positionnées implicitement par l'UAL dans un *registre d'état*
- instructions combinant test et branchement

Loi d'Amdahl

Supposons un programme avec deux parties de temps d'exécution $t = t_1 + t_2$.
Un « dispositif » accélére la partie 2 par un facteur $A > 1$ ($t_2' = t_2/A < t_2$).

L'accélération globale A' du programme sera :

$$\begin{aligned} A' &= \frac{t_1+t_2}{t_1+t_2/A} \\ &= A \times \frac{1}{A \times \frac{t_1}{t_1+t_2} + \frac{t_2}{t_1+t_2}} \end{aligned}$$

Donc $A' < A$, d'autant plus que t_1 est important par rapport à t_2.

Exemple : $t_1 = 90, t_2 = 10, A = 10 \rightarrow A' = 1.098$

Il est donc inutile d'accélérer des fonctions dont l'occurrence est faible.

"Make the common case fast"

Loi de Hennessy-Patterson

$$t_{prog} = n_i \times T_c \times \overline{CPI} \quad (1)$$

Trois facteurs déterminent les performances d'un ordinateur pour l'exécution d'un programme :

- nombre d'instructions nécessaires n_i
- période de l'horloge du processeur T_c,
- nombre moyen de cycles nécessaires pour exécuter une instruction $\overline{CPI}$.

Si on a n, et T et un $\overline{CPI} = 5$ pour un processeur CISC, généralement pour un processeur RISC de même technologie, on aura :

- $n' = 1.2n$ (légère augmentation du nombre d'instructions)
- $T' = 0.7T$ (instructions plus simples $\Rightarrow$ T horloge$\searrow$)
- $\overline{CPI}' = 1.4$ (et même $\overline{CPI} < 1$ pour les processeurs modernes)
  $t' = 1.2 \times 0.7 \times 1.4 \times n \times T = 1.176 \times n \times T < t = 5 \times n \times T$

Le processeur NIOS II

Alain MÉRIGOT

Université Paris Saclay

Le processeur NIOS II

Processeur 32 bits

Présent dans les FPGA Altera/Intel

Générable sous différentes microarchitectures

Jeu d'instruction RISC simple (très proche de MIPS)

Les accès mémoire données et instructions sont strictement alignés.

L'arrangement des données est *petit boutiste* (*little endian*).

Le processeur NIOS II (cont.)

Le NIOS comprend 32 registres de 32 bits nommés `r0` à `r31`

Certains registres ont un rôle spécifique imposé par le matériel.

- `r0` est toujours à 0 et toute écriture vers `r0` n'a aucune action (`nop`).
- `r31` contiendra l'adresse de retour des appels de procédure

D'autres registres ont un rôle imposé par des conventions liées à la chaine logicielle (assembleur, compilateur, éditeur de lien, chargeur, système d'exploitation) et il faut respecter ces conventions pour assurer un bon fonctionnement des programmes.

Jeu d'instructions

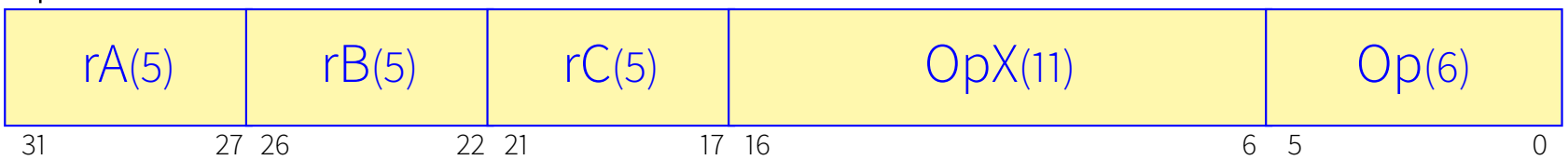
Jeux d'instruction *chargement-rangement*

- **Instructions unité arithmétique et logique** réalisent un calcul entre registres et/ou opérandes immédiats
- **Instructions d'accès mémoire** transfert entre registres et mémoire avec adressage basé
- **Instruction de contrôle de flot** branchement conditionnels, sauts, appels de procédure

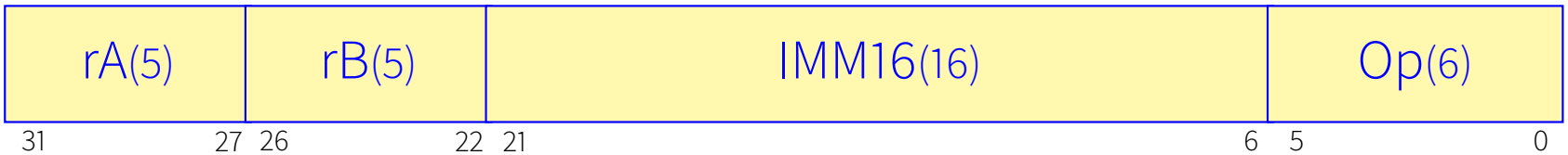
Formats des instructions

Toutes les instructions sont codées sur 32 bits et sont alignées.
3 formats :

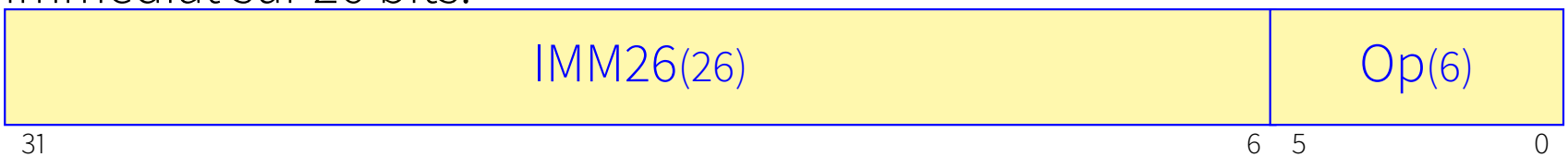
Type R (**r**egistre): décrit des instructions n'ayant que des registres comme opérandes



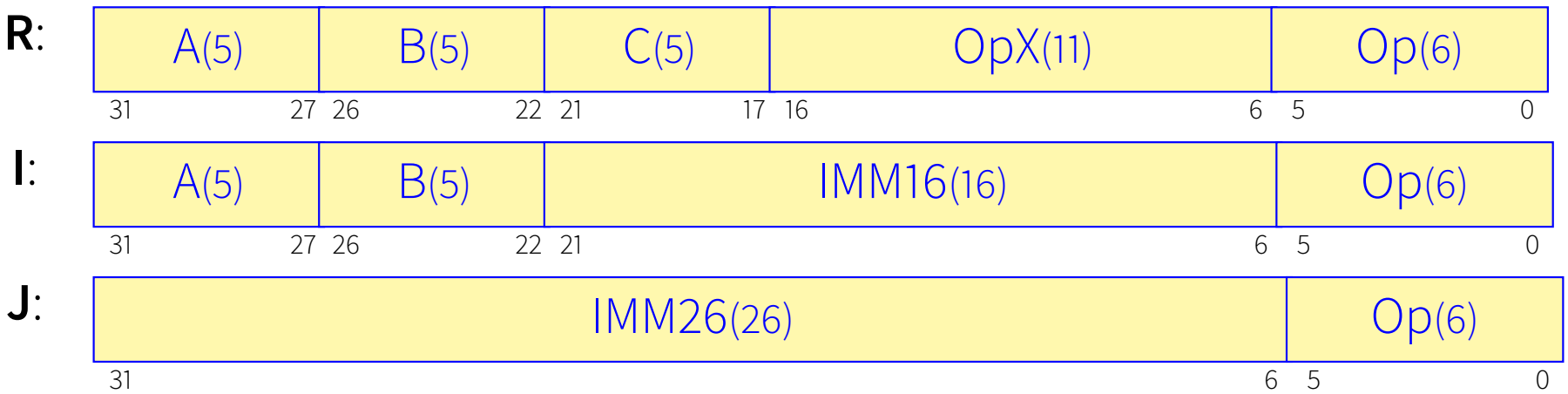
Type I (**i**mmédiat): instructions de calcul ou d'accès mémoire ayant un immédiat sur 16 bits comme opérande



Type J (***j**ump*): utilisé uniquement pour les sauts/appels avec opérande immédiat sur 26 bits.



Formats des instructions (cont.)



Les 6 bits de poids faible (champs `Op`) représentent le *code opération* qui indique le type d'instruction.

Le code `Op=0x3a` indique un format `R`. Le type d'instruction (`add`, `and`, etc) est alors codé par les 11 bits du champ `OpX`.

Les champs `A`, `B` et `C` donnent un numéro de registre sur 5 bits.

Notations :

Imm_n	immédiat sur n bits
$(\mathtt{X})_{n..m}$	bits $n..m$ de X
`rA, rB, rC`	registres 32 bits
`X:Y`	concaténation de X et Y
$\sigma(\mathtt{X})$	extension signée de `X`
`%lo(X)`	16 bits de poids faibles de `X`
`%hi(X)`	16 bits de poids forts de `X`
$\mathrm{Mem}_n[\mathtt{X}]$	n bits à l'adresse mémoire `X`
(signed)`rX`	interprétation signée de `rX`
(unsigned)`rX`	interprétation non signée de `rX`
`pc`	*program counter* ou compteur de programme

Instructions arithmétiques et logiques

Instructions arithméthiques :

add	`add`	`rC,rA,rB`	`rC`$\leftarrow$ `rA`+ `rB`	[R]
addi	`addi`	`rB,rA,Imm16`	`rB`$\leftarrow$ `rA`+ σ(`Imm`$_{16}$)	[I]
sub	`sub`	`rC,rA,rB`	`rC`$\leftarrow$ `rA`$-$ `rB`	[R]
subi	`subi`	`rB,rA,Imm16`	`rB`$\leftarrow$ `rA`$-$ σ(`Imm`$_{16}$)	[I]

Copies de données :

mov	`mov`	`rB,rA`	`rB`$\leftarrow$ `rA`	[M]
			(== `add rB,rA,r0`)	
movi	`movi`	`rB,imm16`	`rB`$\leftarrow$ 0x0000:`Imm`$_{16}$	[M]
			(== `addi rB,r0,Imm16`)	
movhi	`movhi`	`rB,imm16`	`rB`$\leftarrow$ `Imm`$_{16}$:0x0000	[M]
			(== `orhi rB,r0,Imm16`)	
movia	`movia`	`rB,label`	`rB`$\leftarrow$ adr(label)	[M]
			(== `orhi rB,r0, %hi(label)`)	
			(+ `addi rB,r0, %lo(label)`)	

Instructions arithmétiques et logiques (cont.)

Opérations logiques

nor	nor	rC,rA,rB	rC←~(rA|rB)	[R]
			nor rB,rA,r0≡rB←~rA	
and	and	rC,rA,rB	rC←rA&rB	[R]
andi	andi	rB,rA,imm16	rB←rA&(0x0000:Imm_{16})	[I]
andhi	andhi	rB,rA,imm16	rB←rA&(Imm_{16}:0x0000)	[I]
or	or	rC,rA,rB	rC←rA|rB	[R]
ori	ori	rB,rA,imm16	rB←rA|(0x0000:Imm_{16})	[I]
orhi	orhi	rB,rA,imm16	rB←rA|(Imm_{16}:0x0000)	[I]
xor	xor	rC,rA,rB	rC←rA^rB	[R]
xori	xori	rB,rA,imm16	rB←rA^(0x0000:Imm_{16})	[I]
xorhi	xorhi	rB,rA,imm16	rB←rA^(Imm_{16}:0x0000)	[I]

Instructions arithmétiques et logiques (cont.)

Comparaisons

cmpeq	cmpeq rC,rA,rB	rC←(rA== rB? 1 : 0)	[R]
cmpeqi	cmpeqi rB,rA,imm16	rB←(rA== σ(Imm$_{16}$) ? 1 : 0)	[I]
cmpne	cmpne rC,rA,rB	rC←(rA!= rB? 1 : 0)	[R]
cmpnei	cmpnei rB,rA,imm16	rB←(rA!= σ(Imm$_{16}$) ? 1 : 0)	[I]
cmpge	cmpge rC,rA,rB	rC←((signed)rA>= (signed)rB? 1 : 0)	[R]
cmpgei	cmpgei rB,rA,imm16	rB←((signed)rA>= σ(Imm$_{16}$) ? 1 : 0)	[I]
cmpgeu	cmpgeu rC,rA,rB	rC←((unsigned)rA>= (unsigned)rB? 1 : 0)	[R]
cmpgeui	cmpgeui rB,rA,imm16	rB←((unsigned)rA>= (0x0000:Imm$_{16}$) ? 1 : 0)	[I]

cmpgt (et cmpgti, cmpgtu, cmpgtui) pour l'opérateur > [M]

cmple (et cmplei, cmpleu, cmpleui) pour l'opérateur <= [M]

cmplt (et cmplti, cmpltu, cmpltui) pour l'opérateur < [R]

Instructions arithmétiques et logiques (cont.)

Multiplication/division

mul	mul rC,rA,rB	$\texttt{rC} \leftarrow (\texttt{rA} \times \texttt{rB})_{31..0}$	[R]
muli	muli rB,rA,imm32	$\texttt{rB} \leftarrow (\texttt{rA} \times \sigma(\texttt{Imm}_{16}))_{31..0}$	[I]
mulxss	mulxss rC,rA,rB	$\texttt{rC} \leftarrow ((\text{signed})\texttt{rA} \times (\text{signed})\texttt{rB})_{63..32}$	[R]
mulxsu	mulxsu rC,rA,rB	$\texttt{rC} \leftarrow ((\text{signed})\texttt{rA} \times (\text{unsigned})\texttt{rB})_{63..32}$	[R]
mulxuu	mulxuu rC,rA,rB	$\texttt{rC} \leftarrow ((\text{unsigned})\texttt{rA} \times (\text{unsigned})\texttt{rB})_{63..32}$	[R]
div	div rC,rA,rB	$\texttt{rC} \leftarrow ((\text{signed})\texttt{rA} \div (\text{signed})\texttt{rB})_{31..0}$	[R]
divu	divu rC,rA,rB	$\texttt{rC} \leftarrow ((\text{unsigned})\texttt{rA} \div (\text{unsigned})\texttt{rB})_{31..0}$	[R]

Le reste de la division doit être calculé par $R = Dd - Q \times Dv$

```
div rC, rA, rB
mul rD, rC, rB
sub rD, rA, rD ; rD == rA % rB
```

Instructions arithmétiques et logiques (cont.)

Rotations/décalages

rol	rol	rC,rA,rB	$rC \leftarrow (rA_{(31-rB_{4..0})..0} : rA_{31..(32-rB_{4..0})})$ rotation gauche de $(rB_{4..0})$ bits	[R]
roli	roli	rC,rA,imm5	$rC \leftarrow (rA_{(31-Imm_5)..0} : rA_{31..(32-Imm_5)})$	[I]
ror	ror	rC,rA,rB	$rC \leftarrow (rA_{(rB_{4..0}-1..0)} : rA_{31..(rB_{4..0})})$ rotation droite de $(rB_{4..0})$ bits	[R]
sll	sll	rC,rA,rB	$rC \leftarrow (rA << (rB_{4..0}))$	[R]
slli	slli	rC,rA,imm5	$rC \leftarrow (rA << (Imm_5))$	[I]
srl	srl	rC,rA,rB	$rC \leftarrow ((\text{unsigned})rA >> (rB_{4..0}))$	[R]
srli	srli	rC,rA,imm5	$rC \leftarrow ((\text{unsigned})rA >> (Imm_5))$	[I]
sra	sra	rC,rA,rB	$rC \leftarrow ((\text{signed})rA >> (rB_{4..0}))$	[R]
srai	srai	rC,rA,imm5	$rC \leftarrow ((\text{signed})rA >> (Imm_5))$	[I]

`srl` fait un décalage *logique* avec des 0 en poids fort,
`sra` fait un décalage *arithmétique* avec extension de signe.

Accès mémoire

Chargements (`ld`) et rangements (`st`)
sur des octets (***b****yte*), des demi-mots (***h****alf*), ou des mots (***w****ord*).

ldb	`ldb rB,imm16(rA)`	$rB \leftarrow \sigma(Mem_8[rA + \sigma(Imm_{16})])$	[I]
ldbu	`ldbu rB,imm16(rA)`	$rB \leftarrow 0x000000:Mem_8[rA + \sigma(Imm_{16})]$	[I]
ldh	`ldh rB,imm16(rA)`	$rB \leftarrow \sigma(Mem_{16}[rA + \sigma(Imm_{16})])$	[I]
ldhu	`ldhu rB,imm16(rA)`	$rB \leftarrow 0x0000:Mem_{16}[rA + \sigma(Imm_{16})]$	[I]
ldw	`ldw rB,imm16(rA)`	$rB \leftarrow Mem_{32}[rA + \sigma(Imm_{16})]$	[I]
stb	`stb rB,imm16(rA)`	$Mem_8[rA + \sigma(Imm_{16})] \leftarrow rB_{7..0}$	[I]
sth	`sth rB,imm16(rA)`	$Mem_{16}[rA + \sigma(Imm_{16})] \leftarrow rB_{15..0}$	[I]
stw	`stw rB,imm16(rA)`	$Mem_{32}[rA + \sigma(Imm_{16})] \leftarrow rB$	[I]

Toutes les instructions `ld` et `st` existent en version entrée-sortie (`io`), qui évitent la mémoire cache (`ldbio, ldbuio`, etc).
Ceci permet d'accéder à des périphériques.

Instructions de contrôle de flot

Branchements

br	br	imm16	pc← pc+4+σ(Imm$_{16}$)	[I]
beq	beq	rA,rB,imm16	si (rA== rB)	[I]
			alors pc← pc+4+σ(Imm$_{16}$)	
			sinon pc← pc+4	
bne	bne	rA,rB,imm16	idem si (rA!= rB)	[I]
bge	bge	rA,rB,imm16	idem si ((signed)rA>= (signed)rB)	[I]
bgeu	bgeu	rA,rB,imm16	idem si ((unsig.)rA>= (unsig.)rB)	[I]
bgt	bgt	rA,rB,imm16	idem si ((signed)rA> (signed)rB)	[M]
bgtu	bgtu	rA,rB,imm16	idem si ((unsig.)rA> (unsig.)rB)	[M]
ble	ble	rA,rB,imm16	idem si ((signed)rA<= (signed)rB)	[M]
bleu	bleu	rA,rB,imm16	idem si ((unsig.)rA<= (unsig.)rB)	[M]
blt	blt	rA,rB,imm16	idem si ((signed)rA< (signed)rB)	[I]
bltu	bltu	rA,rB,imm16	idem si ((unsig.)rA< (unsig.)rB)	[I]

Instructions de contrôle de flot (cont.)

Sauts/appels de procédure

jmpi	`jmpi imm26`	$pc \leftarrow (pc_{31..28} : Imm_{26} \times 4)$	[J]
jmp	`jmp rA`	$pc \leftarrow rA$	[R]
call	`call imm26`	$ra \leftarrow pc + 4$ //r31 $pc \leftarrow (pc_{31..28} : Imm_{26} \times 4)$	[J]
callr	`callr rA`	$ra \leftarrow pc + 4$ //r31 $pc \leftarrow rA$	[R]
ret	`ret`	$pc \leftarrow ra$ //r31	[R]

Opérations flottantes

On peut ajouter des extensions pour le calcul flottant, uniquement en simple précision.
Les nombres flottants sont stockés dans les mêmes registres que les entiers[1].
Les instructions sont de type R.
Le support flottant comprend :

- des instructions arithmétiques et logiques : `fsubs fadds fmuls fdivs fsqrts`
- des conversions entier-flottant : `floatis fixsi round`
- des opérations simples : `fmins fmaxs fnegs fabss`
- des comparaisons : `fcmplts fcmples fcmpgts fcmpges fcmpeqs`[2] `fcmpnes`[2]

[1] contrairement à la plupart des processeurs
[2] Attention aux erreurs d'arrondi dans les comparaisons flottantes d'égalité/inégalité

Les registres Nios II

`r0`	`zero`	toujours à 0
`r1`	`at`	*assembler temporary* (utilisé par certaines macros)
`r2-r3`		valeur de retour des procédures
`r4-r7`		arguments des procédures
`r8-r15`		registres volatiles (non préservés)
`r16-r23`		registres non volatiles (à sauvegarder par la procédure appelée avant usage)
`r24`	`et`	*exception temporary*
`r25`	`bt`	*breakpoint temporary*
`r26`	`gp`	*global pointer*
`r27`	`sp`	*stack pointer*
`r28`	`fp`	*frame pointer*
`r29`	`ea`	*exception return address*
`r30`	`sstatus`	*status register/breakpoint return address*
`r31`	`ra`	*return address*

Ecriture d'un programme assembleur

Un programme assembleur comprend des **instructions** et des **directives** permettant notamment de définir des données.
Principales directives :

```
.data              # début d'un segment de données
x: .word 10        ; définit une variable de 4 octets x
                   ; et lui affecte la valeur 10 (ou 0xa)
                   ; idem pour .byte(1o), .half(2o), .dword(8o)
                   ; les données sont alignées
y: .space 16       # 16 octets non initialisés à l'adresse y
z: .ascii "hello, world!"  # Une chaine de caractères à l'adresse z
                   # Pour une chaîne C standard (terminée par un '\0'),
                   # utiliser .asciz
.equ N,12          # N sera remplacé par 12
.include "file.asm"  # inclut un fichier
.text              # début des instructions
...
.end               # fin du programme. Ignore le reste du fichier
```

Exemples de programmes assembleur

Produit scalaire de deux vecteurs

```
int v1[2]={10, -5};
int v2[2]={-3, 8};
int ps;
....
ps =
     v1[0]*v2[0]
      +v1[1]*v2[1];
...
```

```
.data
v1: .word 10,-5
v2: .word -3,8
ps: .space 4
.text
movia r8,v1     #r8=&v1[0]
ldw r9,0(r8)    #r9=v1[0]
ldw r10,4(r8)   #r10=v1[1]
movia r8,v2     #r8=&v2[0]
ldw r11,0(r8)   #r11=v2[0]
ldw r12,4(r8)   #r12=v2[1]
mul r9, r9,r11 #v1[0]*v2[0]
mul r10,r10,r12#v1[1]*v2[1]
add r9, r9,r10 #r9=ps
movia r8,ps     #r8=&ps
stw r9,0(r8)    #write ps
```

Exemples de programmes assembleur (cont.)

Mise en oeuvre de tests: maximum de deux nombres

```
      ....
int a,b, max;

if (a>b)
  max = a;
else
  max = b;
 ...
```

```
  # on suppose a->r8, b->r9,
  #     max->r10
  ble r8, r9, else#a<=b?->else
  mov r10, r8 # max <= x
  br fin
else:
  mov r10, r9 # max <= y
fin: ...
```

ou (en plus optimisé)

```
  mov r10, r8
  bgt r8,  r9, fin
  mov r10, r9
fin: ...
```

Exemples de programmes assembleur (cont.)

Mise en oeuvre de boucles.

```
for(initialisation; condition; increment) {
  corps_de_boucle;
}
```

Mise en oeuvre directe

```
        initialisation
bcle:   test condition
        si faux -> fin
        corps_de_boucle
        increment
        -> bcle (incond.)
fin:    ...
```

Version optimisée (1 seul brcht/boucle) forme « *do .. while()* »

```
        initialisation
        -> cond (incond.)
debut:  corps_de_boucle
        increment
cond:   test condition
        si vrai -> debut
fin:    ...
```

si il y au moins une itération, le premier saut peut être supprimé

Exemples de programmes assembleur (cont.)

Boucle : mise à zéro d'un tableau

```
int N;
short t[N];
for(int i=0; i<N; i++)
  t[i]=0;
```

```
// équivalent à :
int N;
short t[N], *pt;
for(int i=0, pt=&t[0];
          i<N;
          i++,pt++)
  *pt=0;
```

```
  # on suppose @t->r8, N->r9
### initialisation boucle
  mov  r10,r8 #r10=@t=&t[0]
  mov  r11,r0   #i=0
boucle:
### corps de la boucle
  sth  r0,0(r10) #*t=0
### incrément de la boucle
  addi r11,r11,1 #i++
  addi r10,r10,2 #t++(@t+=2)
  # 2==sizeof(short)
### test condition
  blt  r11,r9,boucle
       #si i<N -> boucle
fin: ...
```

Exemples de programmes assembleur (cont.)

Boucle : ondelette de Haar

```
int N;
int t[N], v[N];

for(int i=0;i<N;i++){
  if (i%1) // i impair
    t[i]=v[i-1]-v[i];
  else     // i pair
    t[i]=v[i]+v[i+1];
}
```

```
# @t->r13 @v->r14 N->r15
    mov  r16,r13  #r16=@t=&t[0]
    mov  r17,r14  #r17=@v=&v[0]
    mov  r8,r0    #i=0
bcl:ldw  r9,0(r17)#r9=v[i]
    andi r11,r8,0x01
    beq  r11,r0, pair
    ldw  r10,-4(r17)#r10=v[i-1]
    sub  r9,r10,r9#v[i-1]-v[i]
    br   cnt
pair:ldw  r10,4(r17)#r10=v[i+1]
    add  r9,r9,r10#v[i]+v[i+1]
cnt:stw  r9,0(r16)#range t[i]
    addi r8,r8,1  #i++
    addi r16,r16,4#t++ (@t+=4)
    addi r17,r17,4#v++ (@v+=4)
    blt  r8,r15,bcl # i<N?
fin: ...
```

Exemples de programmes assembleur (cont.)

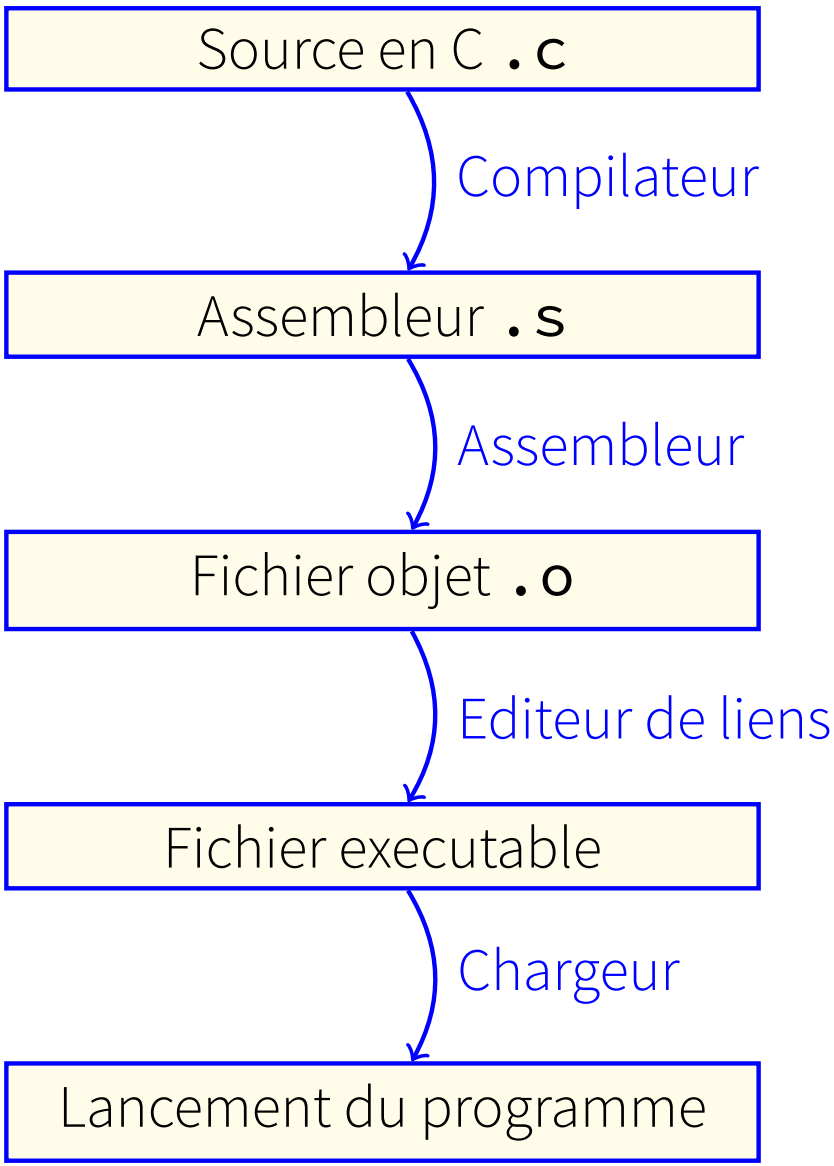
Boucle : ondelette de Haar (version optimisée)

```
int N;
int t[N], v[N];

for(int i=0;i<N;i+=2){
  t[i]=v[i]+v[i+1];
  t[i+1]=v[i]-v[i+1];
}
```

```
# @t->r13 @v->r14 N->r15
  mov  r16,r13    #r16=@t=&t[0]
  mov  r17,r14    #r17=@v=&v[0]
  mov  r8,r0      #i=0
boucle:
  ldw  r9,0(r17) #v[i]
  ldw  r10,4(r17)#v[i+1]
  add  r11,r9,r10#v[i]+v[i+1]
  stw  r11,0(r16)#t[i]
  sub  r11,r9,r10#v[i]-v[i+1]
  stw  r11,4(r16)#t[i+1]
  addi r8,r8,2    #i+=2
  addi r16,r16,8 #t+=2 (@t+=8)
  addi r17,r17,8 #v+=2 (@v+=8)
  blt  r8,r15,boucle#i<N?
fin: ...
```

Processus de développement logiciel



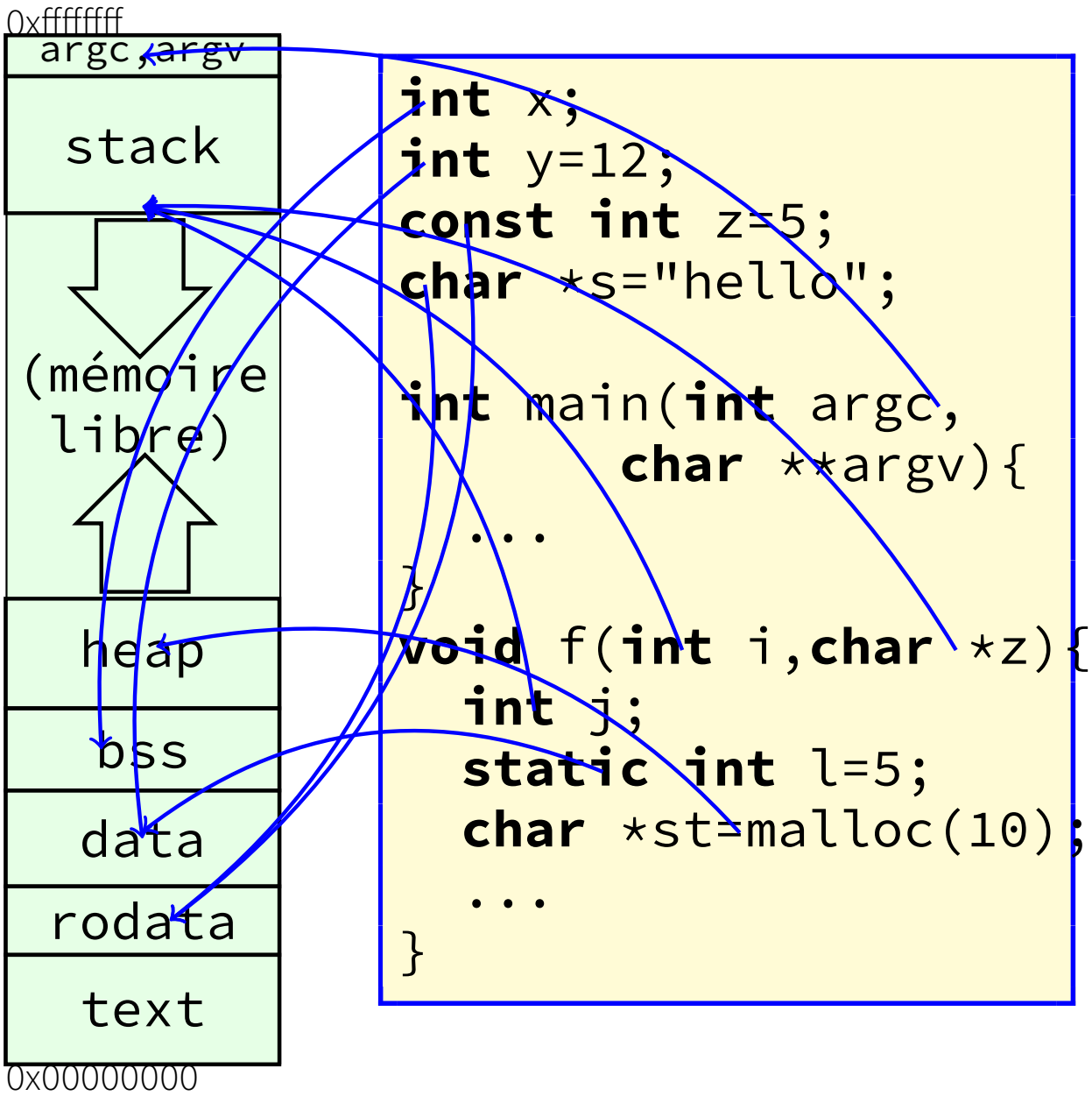
Un **compilateur** va traduire en assembleur un fichier C

L'**assembleur** convertit le programme en fichier objet. Il contient le code binaire des instructions et des symboles donnant les noms des variables, fonctions, etc

L'**éditeur de lien** (ou *linker*) rassemble les fichiers objets et les bibliothèques. Il positionne les variables globales et statiques en mémoire, ainsi que les fonctions, et modifie le code pour générer les adresses correspondantes.

Le **chargeur** (ou *loader*) demande des ressources au système d'exploitation. Il positionne divers registres et tables et lance l'exécution du programme.

Segments mémoire d'un exécutable



Segments mémoire (créés par le *loader*)

pile (stack) gestion des appels de fonctions

tas (heap) allocations dynamiques (malloc())

bss données non initialisées

data données initialisées

rodata données constantes

text (code) instructions du programme

Appels de procédure

Lors d'un appel de procédure, l'adresse de retour est sauvegardée dans `ra` (`r31`).
Mais il est nécessaire de gérer une *pile* pour :

- sauvegarder les adresses de retours lors d'appels imbriqués de fonctions
- accéder aux paramètres des fonctions
- mettre les variables locales qui ne tiennent pas dans des registres
- sauvegarder des registres non volatiles avant de les modifier

La pile est gérée au moyen de deux pointeurs :

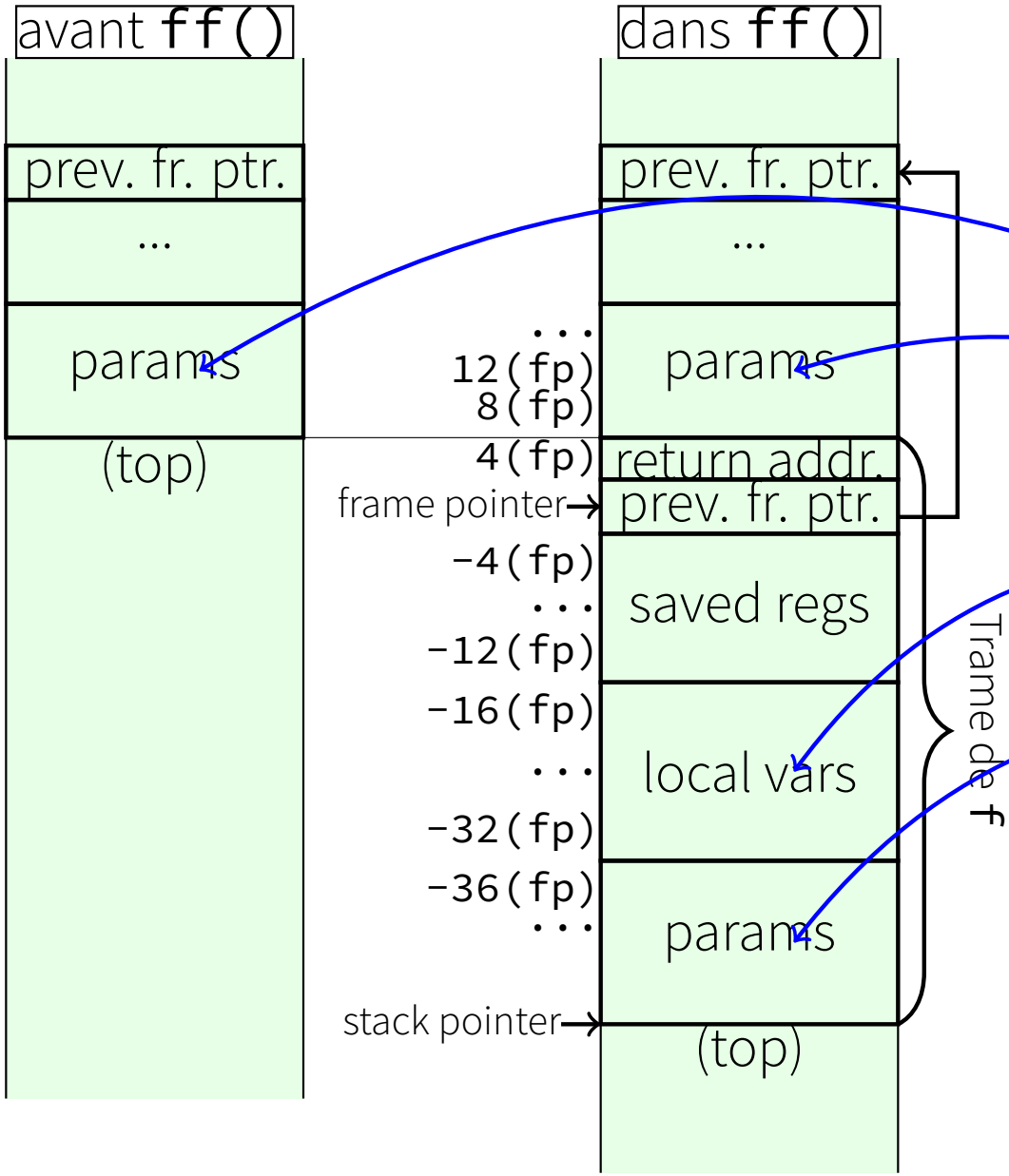
- le **pointeur de pile** (*stack pointer*) (`sp` ou `r27`) qui pointe sur la valeur courante du haut de pile
- le **pointeur de trame** (*frame pointer*) (`fp` ou `r28`) qui permet de revenir à la trame précédente (utile pour le *debug*)

Appels de procédure (cont.)

Pour assurer la compatibilité des différentes fonctions, spécification de **conventions d'appels** (*calling conventions*).

- En début de fonction, on réserve un espace suffisant dans la pile pour la trame de la fonction et les arguments des fonctions appelées. Cet espace sera libéré en fin de fonction.
- les 4 premiers arguments d'une fonction sont passés dans les registres `r4`–`r7`. Les arguments suivants sont dans la pile.
- la valeur de retour d'une fonction est dans le registre `r2` (et `r3` si 64 bits sont nécessaires).
- les registres `r8`–`r15` peuvent être utilisé sans contrainte et doivent être sauvegardés par la fonction appelante si nécessaire (registres *volatiles*)
- les registres `r16`–`r23` ne peuvent être utilisés par une fonction que s'ils sont préalablement sauvegardés (registres *non volatiles*).

Appels de procédure (cont.)




```
int g(int a, int b
  int x, int y, int z);
int h() {
  ...
  ff(1,2);
  ...
}
int ff(int u,int v){
  int e,f;
  e=2*u;
  f=2*v;
  f=g(f,u,e,f,5);
  return f;
}
```

Accès aux paramètres et variables locales par `fp` (ou `sp`) + un décalage.

Appels de procédure (cont.)

En début de fonction:

- réserver un espace suffisant dans la pile pour la trame de la fonction
- empiler `ra` si la fonction appelle une autre fonction (fonction *non terminale*)
- empiler `fp`
- `fp` = adr(`fp`)
- si nécessaire, sauvegarder les registres non volatiles (`r16-r23`) que l'on va utiliser

```
int ff(int u,int v){
  int e,f;
  x=2*u;
  y=2*v;
  f=g(5,u,v,x,y);
  return f;
}
```

```
ff:# début fonction ff()
   addi sp,sp,-24# alloc. trame
   stw  ra,20(sp)#sauv. adr. ret
   stw  fp,16(sp)#sv. fp. prec.
   addi fp,sp,16 # nouveau fp
   stw  r16,12(sp) # sauv. r16
     # ou    stw  r16,-4(fp)
   stw  r17,8(sp)  # sauv. r17
     # ou stw  r17,-8(fp)
   ...
```

Appels de procédure (cont.)

corps de la fonction

- `u` est dans `r4` et `v` dans `r5`[1]
- on met `x` dans `r16` et `y` dans `r17`
- calcul de `x` et `y`
- (si nécessaire, on sauvegarde les registres volatiles `r8`–`r15`)[1]
- on copie les paramètres[1] :
  - 5→`r4`
  - u→`r5`
  - v→`r6`
  - x→`r7`
  - y→pile
- appel `g()`

[2] conformément aux conventions d'appel

```
int ff(int u,int v){
  int e,f;
  x=2*u;
  y=2*v;
  f=g(5,u,v,x,y);
  return f;
}
```

```
ff:# suite fonction ff()
   slli r16,r4,1 #2*u
   slli r17,r5,1 #2*v
   mov  r6,r5
   mov  r5,r4
   addi r4,r0,5
   mov  r7,r16
   stw  r17,4(sp)
   call g
   ...
```

Appels de procédure (cont.)

fin de la fonction `ff()`

- la valeur de retour de `g()` est dans `r2`[1] et on la copie dans `f` (`r17`)
- on copie `f` de `r17` vers le registre `r2` (valeur de retour de la fonction[1]) (largement optimisable!)
- on restaure les registres `ra` et `fp`[1]
- on libère la trame
- retour de la fonction

NB : l'utilisation d'un *frame pointer* est optionnelle, mais `fp` permet d'utiliser un *debugger*.

```
int ff(int u,int v){
  int x,y;
  x=2*u;
  y=2*v;
  f=g(5,u,v,x,y);
  return f;
}
```

```
ff:# fin fonction ff()
   mov  r17,r2
   mov  r2, r17
   ldw  ra, 20(sp)
   ldw  fp, 16(sp)
   addi sp,sp,24
   ret
```

[2] conformément aux conventions d'appel

Appels de procédure (cont.)

Noter que si :

- la fonction est terminale
- on n'utilise pas de *frame pointer*
- on n'utilise que des registres volatiles

il est inutile de gérer une pile.
On peut toujours utiliser `sp` pour récupérer des arguments de la fonction, si nécessaire.

```
int g(int a, int b,
      int x, int y, int z)
{
  return a+b+x+y+z;
}
```

```
g:
    add   r2,r4,r5 # r2=a+b
    add   r2,r2,r6 # r2+=x
    add   r2,r2,r7 # r2+=y
    # z est dans la pile
    # (5e argument de g)
    # Noter que sp n'a pas bougé
    # et z est toujours en sp+4
    ldw   r8, 4(sp)# r8=z
    add   r2,r2,r8 # r2+=z
    ret
```

Mise en oeuvre multicycle du processeur NIOS II

Alain MÉRIGOT

Université Paris sud

Introduction

Le formalisme *multicycle* a été employé pour réaliser des processeurs jusque dans les années 80.

Aujourd'hui, il est encore employé pour des microcontroleurs ou même des processeurs en version basses performances (comme le Nios II/e).

On définit un *chemin de données* et chaque instruction est décomposée en étapes élémentaires sur ce chemin.

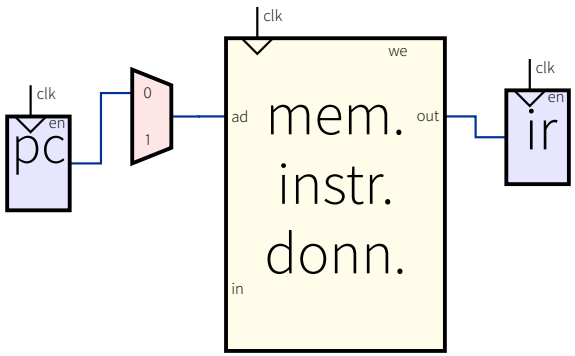
L'exécution des instructions est pilotée par un automate qui va contrôler les différents éléments du chemin de données (registres, multiplexeurs, opérateurs) en fonction des caractéristiques de l'instruction.

Phase d'acquisition

La première phase de toute instruction est la phase d'**acquisition des instruction** (*fetch*).

Nous avons besoin d'introduire :

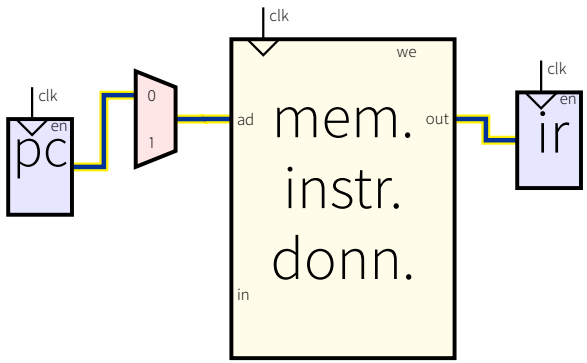
- la *mémoire*, qui contiendra instructions et données
- le *compteur de programme* `pc`, qui contiendra l'adresse de la prochaine instruction à exécuter
- le *registre d'instruction* `ir`, qui stocke le contenu de l'instruction en cours d'exécution



Phase d'acquisition (cont.)

L'action à réaliser durant cette phase est : `ir <- mem[pc]`
Pour cela, il faut :

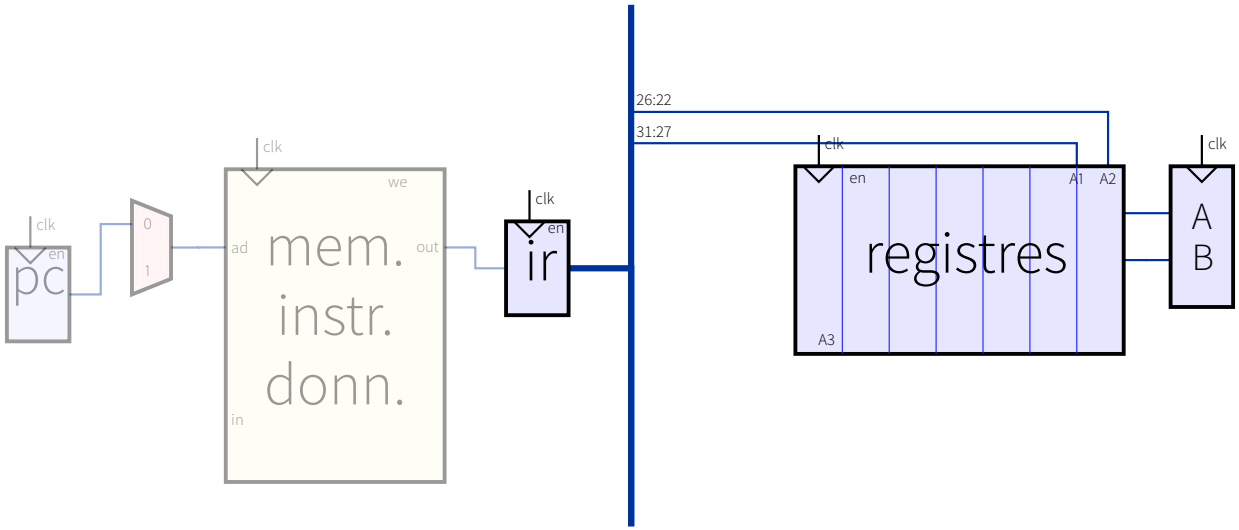
- aller lire l'instruction dans la mémoire à une adresse contenue dans le compteur de programme (`pc`)
- copier la sortie de la mémoire dans le registre d'instructions (`ir`)



Il est également nécessaire de préparer l'instruction suivante en incrémentant le compteur de programme.
Nous utiliserons pour ce calcul l'unité arithmétique et logique.

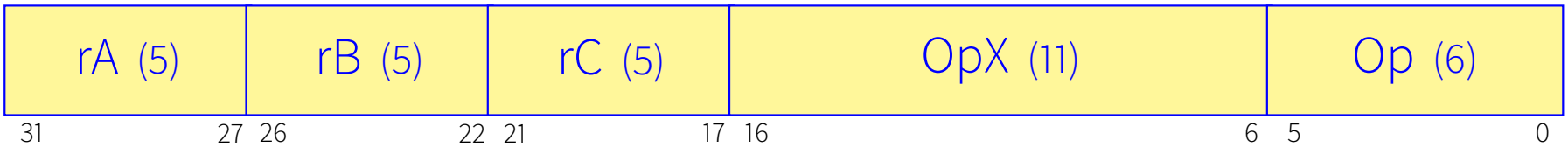
Décodage des instruction

Au début de la deuxième phase, l'instruction est écrite dans `ir`.
Durant cette phase, le contenu de l'instruction est disponible sur le bus instruction, mais la phase **doit** être identique pour toutes les instructions.
Réalisation du *décodage des instructions* (phase *decode*).
Pour préparer les phases d'exécution suivantes, extraction des opérandes du *banc de registre*
A chaque cycle, le banc de registres peut lire deux opérandes (`A1, A2`) et en écrire un troisième (`A3`).

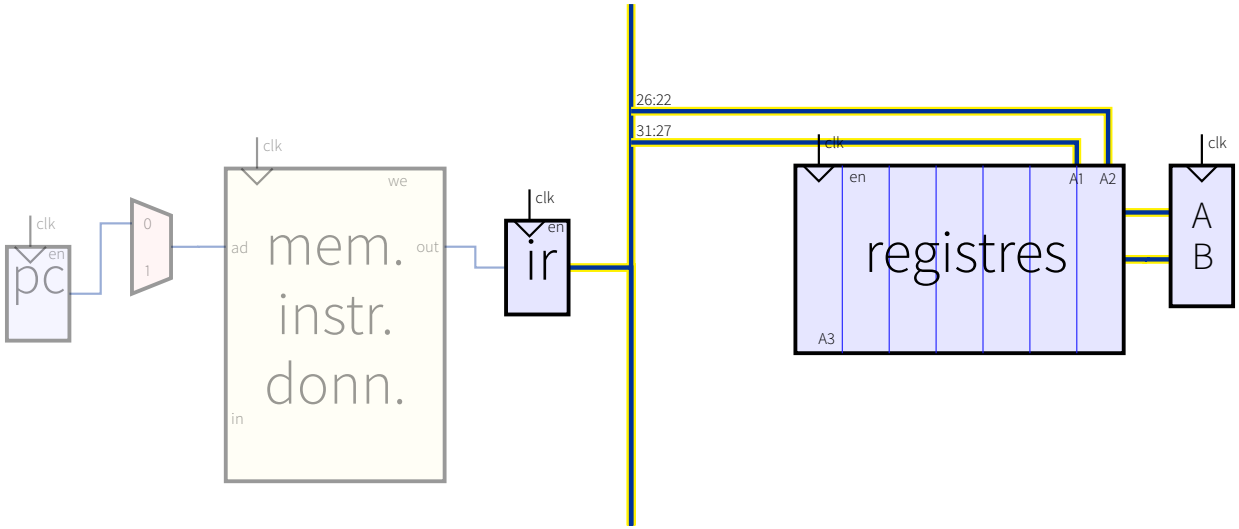


Décodage des instruction (cont.)

Les opérandes spécifiés par les champs **rA** (31:27) et **rB** (26:22) des instructions sont copiés dans des *registres tampon* **A** et **B**.

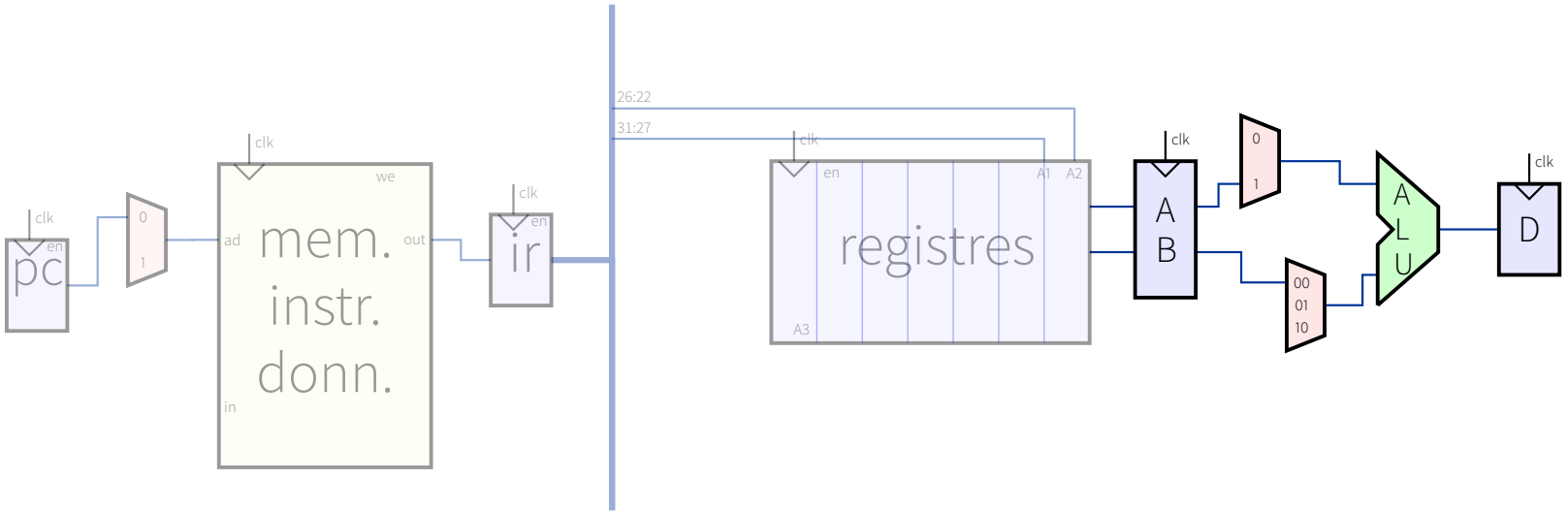



```
A <- rA
B <- rB
```



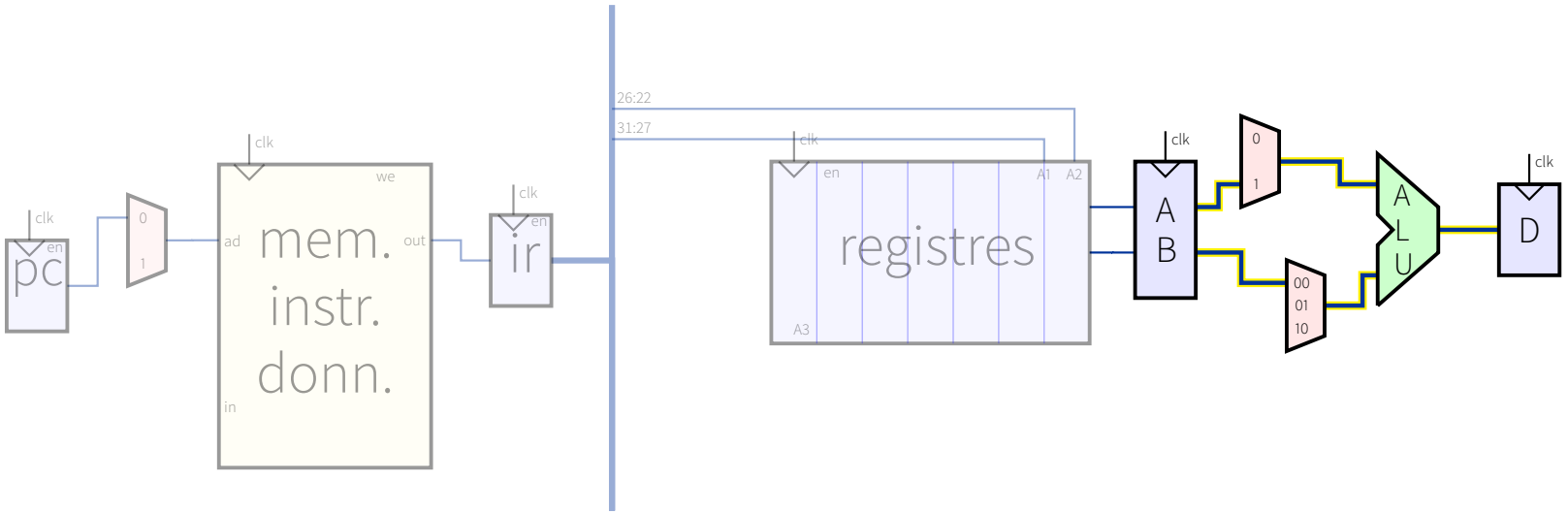
Instructions arithmétiques et logiques

La troisième phase est la phase d'*exécution* des instructions.
Pour les instructions arithmétiques et logiques, il faut appliquer l'opération spécifiée par l'instruction (**add**, **sub**, **and**, etc) aux opérandes copiés dans la phase *decode*.
Introduction de l'unité arithmétique et logique **ALU** et du registre tampon **D**.



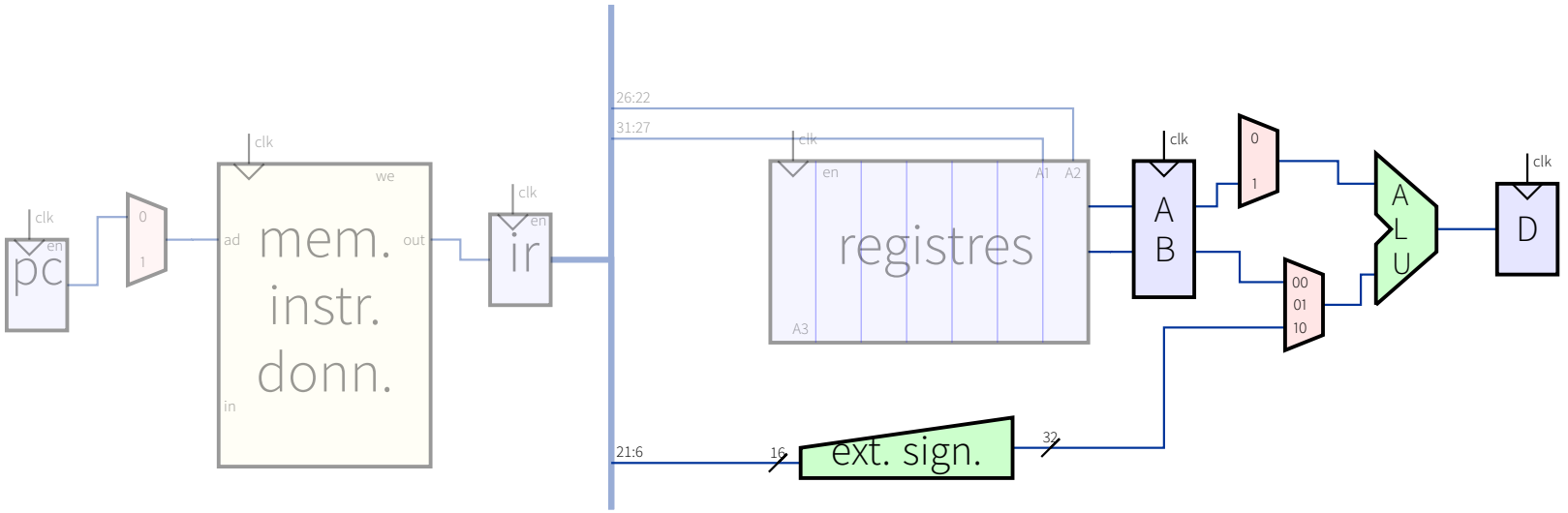
Instructions arithmétiques et logiques (cont.)

L'unité arithétique et logique utilise comme opérandes les registres tampons **A** et **B**.
Le résultat de l'opération effectuée par l'ALU est copié dans le registre tampon **D**.



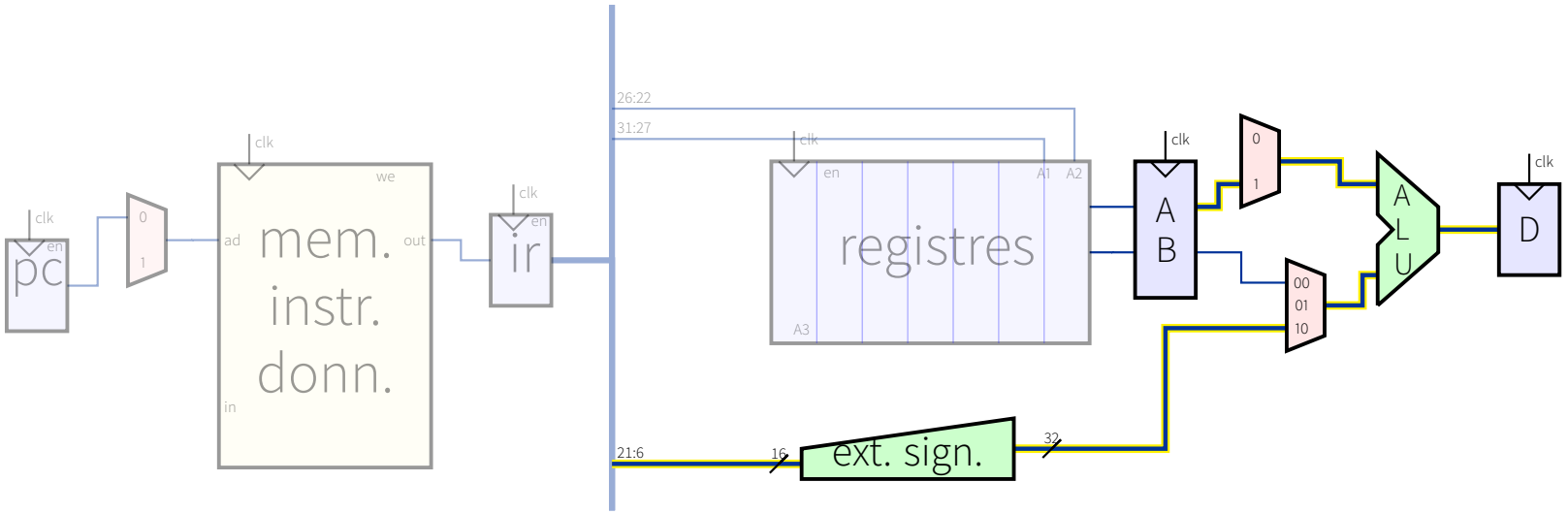
Instruction avec immédiat

Pour les instructions avec un immédiat (**`addi, subi,`** etc), l'opération doit porter entre le registre **`rA`** et l'immédiat présent sur les 16 bits (21:6).
Extraction des bits 21:6 de **`ir`** et extension signée à 32 bits.



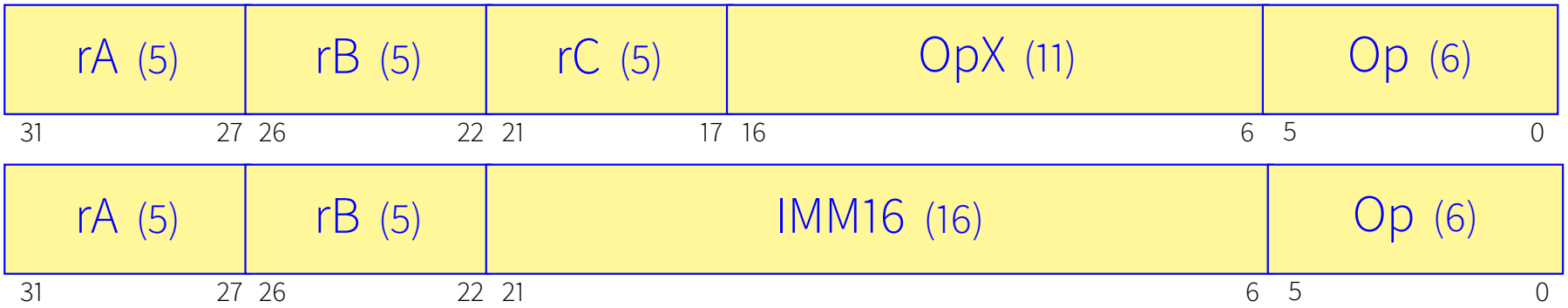
Instruction avec immédiat (cont.)

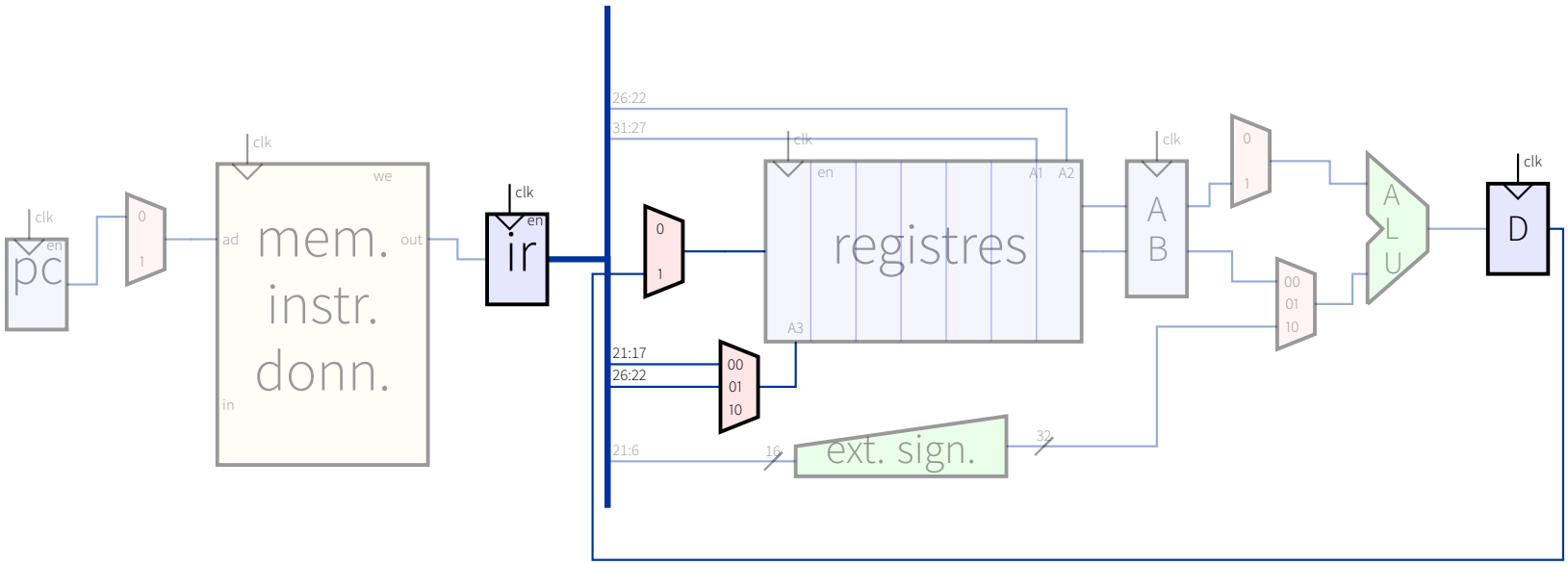
Comme précédemment le résultat du calcul est copié dans le registre **D**.



Rangement du résultat

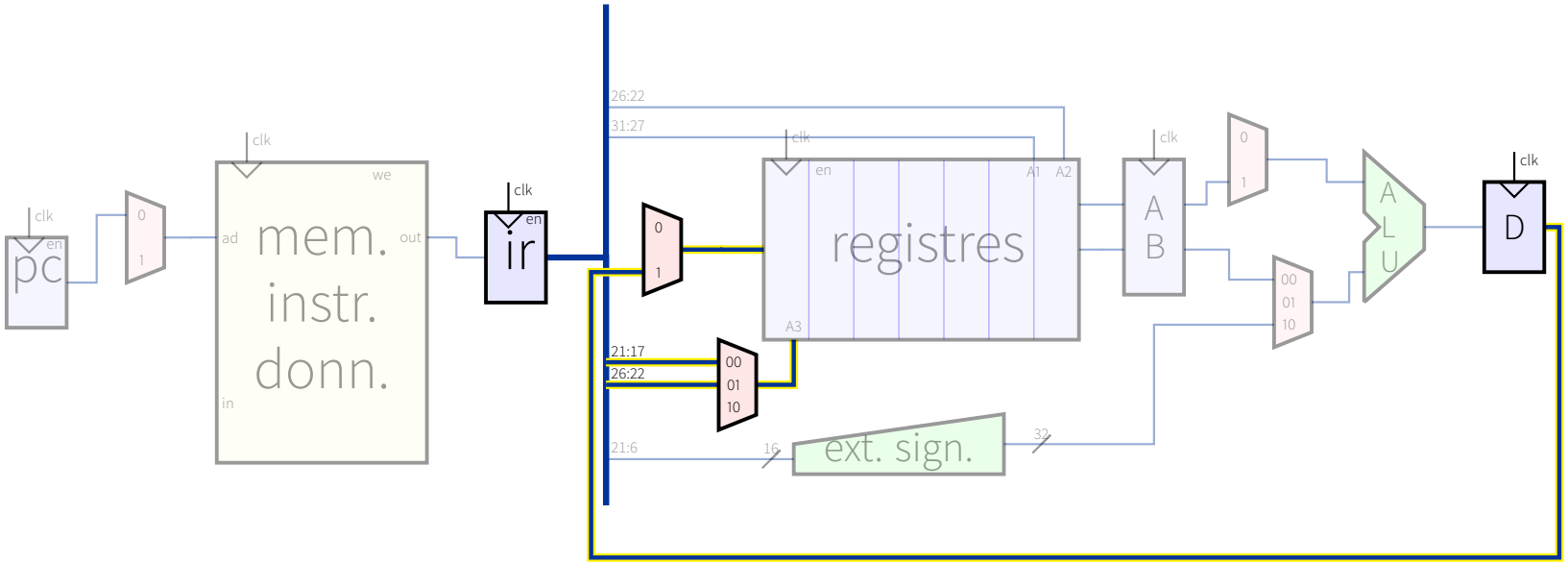
Une fois le calcul effectué et copié dans **D**, il faut ranger le résultat dans le banc de registres.
Suivant le type d'instruction, le registre destination est décrit par `rC` ou `rB`.





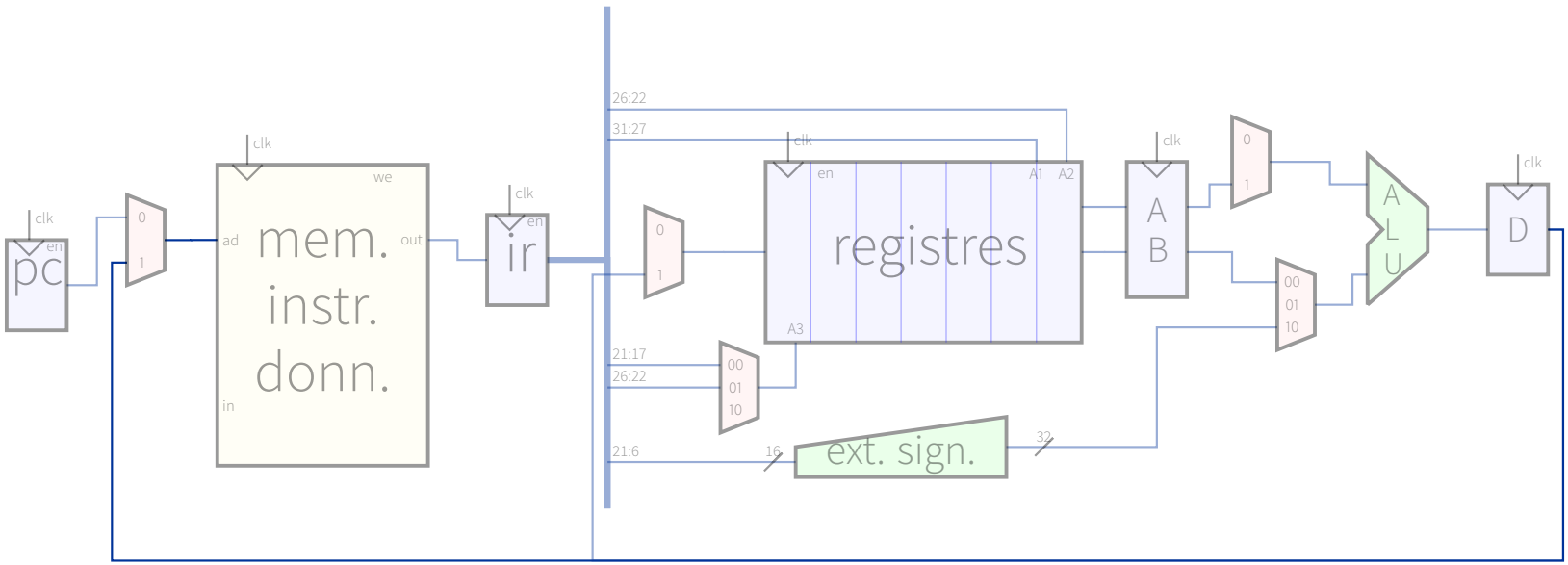
Rangement du résultat (cont.)

Pour les instructions entre registres, recopie du résultat dans le registre `rC` (décrit par les bits 21:17 de l'instruction)
Pour les instructions avec un immédiat, recopie du résultat dans le registre `rB` (décrit par bits 26:22 de l'instruction)



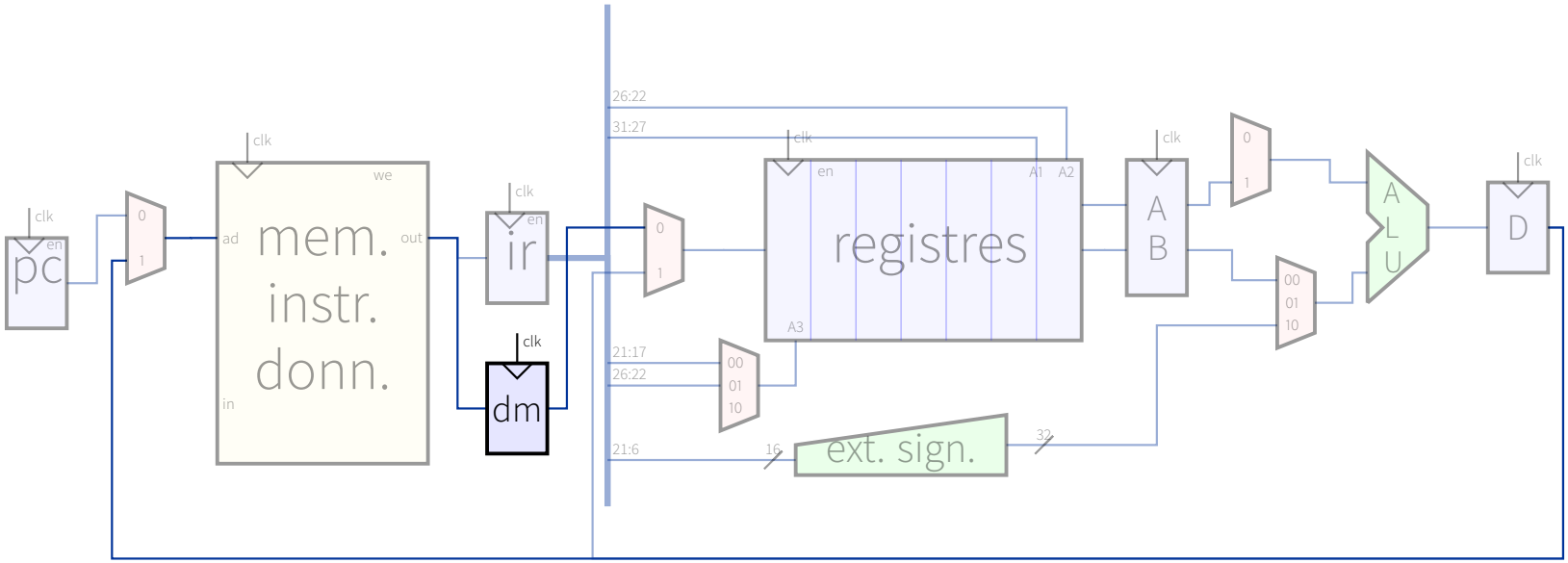
Accès mémoire

Les accès mémoire nécessitent de calculer l'adresse en ajoutant le registre `rA` à un immédiat.
Le mécanisme de calcul est identique aux instructions ALU avec un immédiat.
Le résultat du calcul servira ensuite à adresser la mémoire.
Ajout d'un chemin entre **D** et l'entrée **ad** de la mémoire.



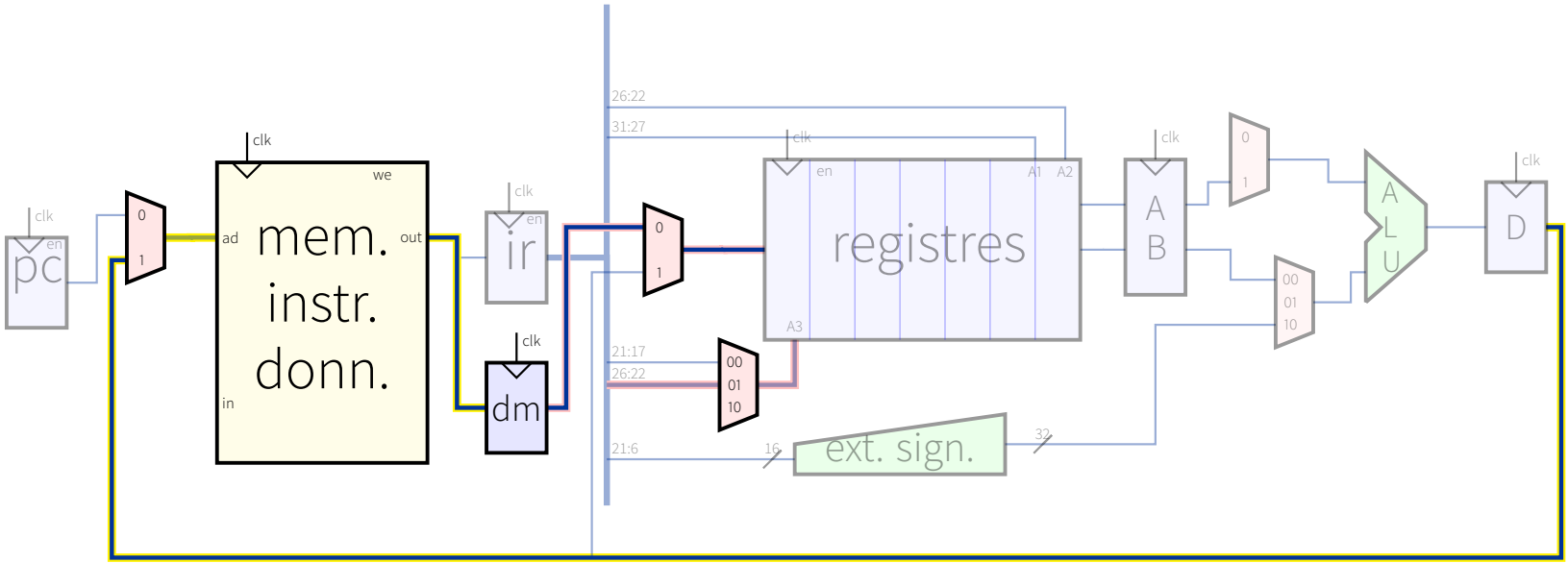
Accès mémoire : chargements

Dans le cas d'un chargement (**`ldb, ldh, ldw, etc`**), la mémoire est ensuite lue et son résultat est écrit dans un *registre de données mémoire* (**`dm`**).



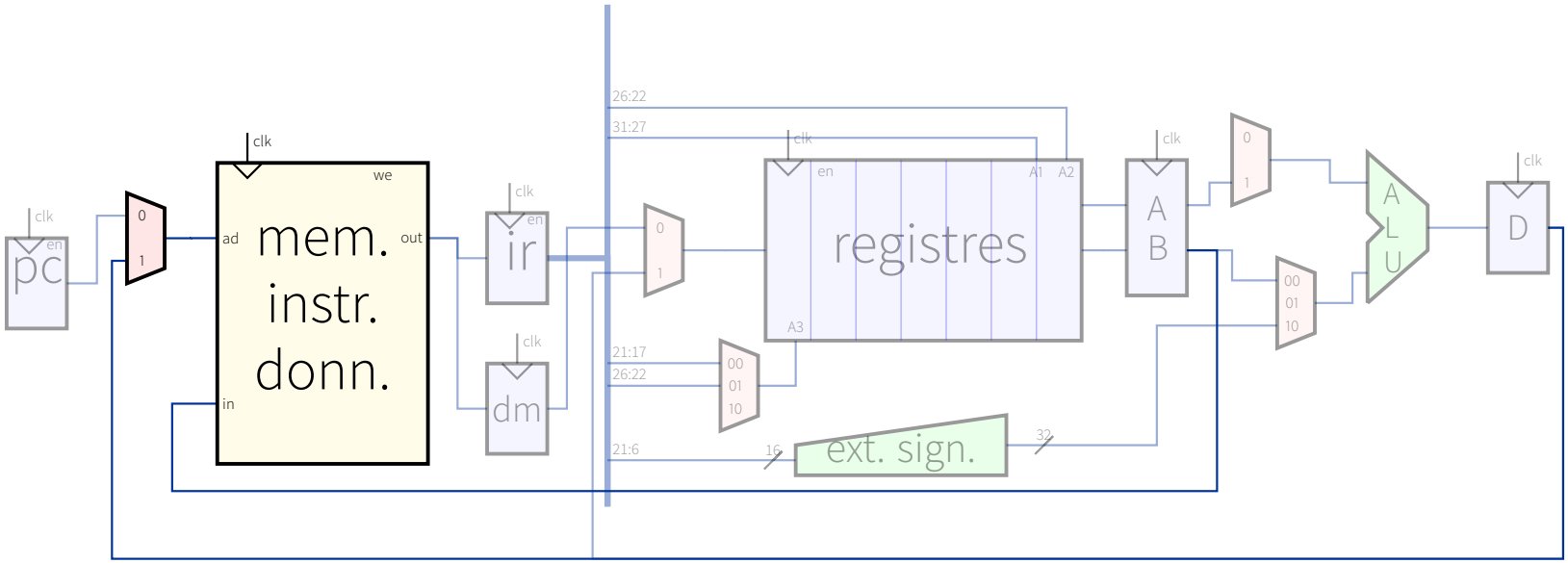
Accès mémoire : chargements (cont.)

Une fois la mémoire lue, dans un deuxième temps, le contenu du registre **dm** est copié vers le registre **rB** du banc de registres.



Accès mémoire : rangements

La partie calcul d'adresse est identique pour les instructions de rangement *store* (`stb, sth,` etc).
Il faut ensuite copier le registre de numéro `rB` dans la mémoire.
Celui-ci a été chargé dans le registre tampon **B** lors de la phase *decode*.
Chemin de **B** vers l'entrée de la mémoire.

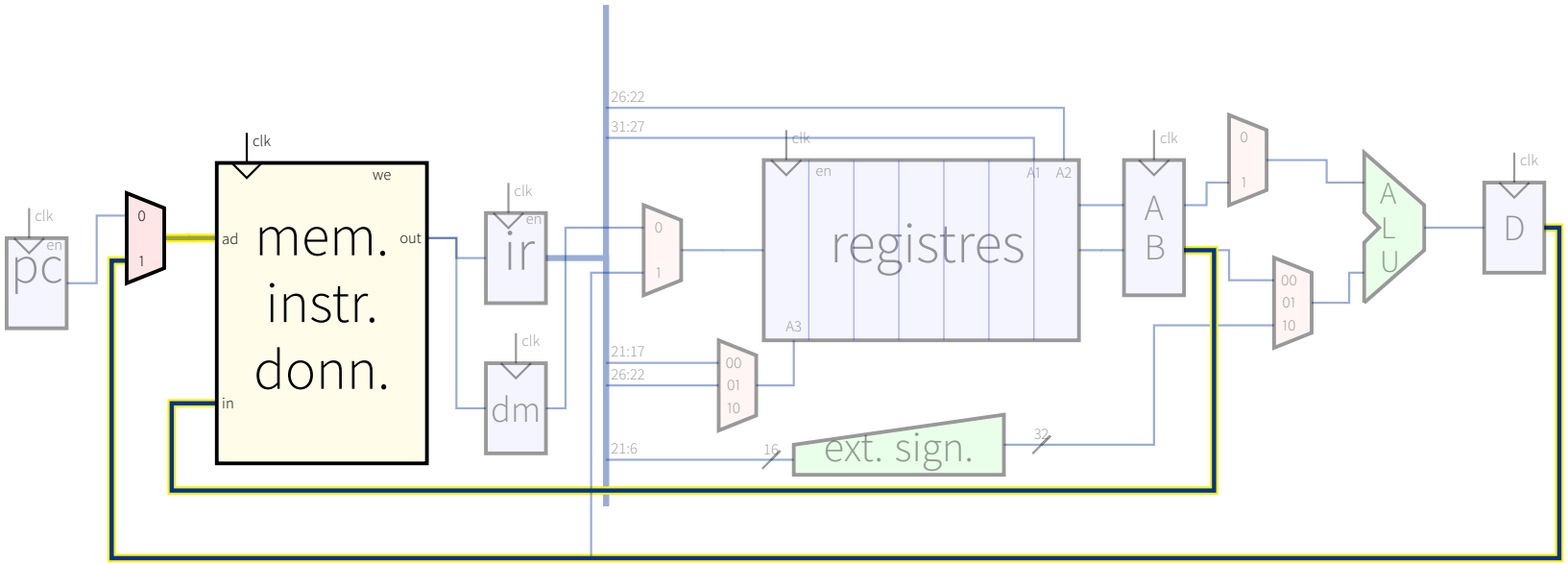


Accès mémoire : rangements (cont.)

Pour effectuer l'écriture, on transfère simultanément :

- le registre **D** qui contient le résultat `rA + immₙ₁₆` vers l'entrée d'adresse de la mémoire.
- le registre **B** (qui est une copie de `rB`) vers l'entrée de données de la mémoire.

et on donne un ordre d'écriture à la mémoire.



Mise à jour du compteur de programme

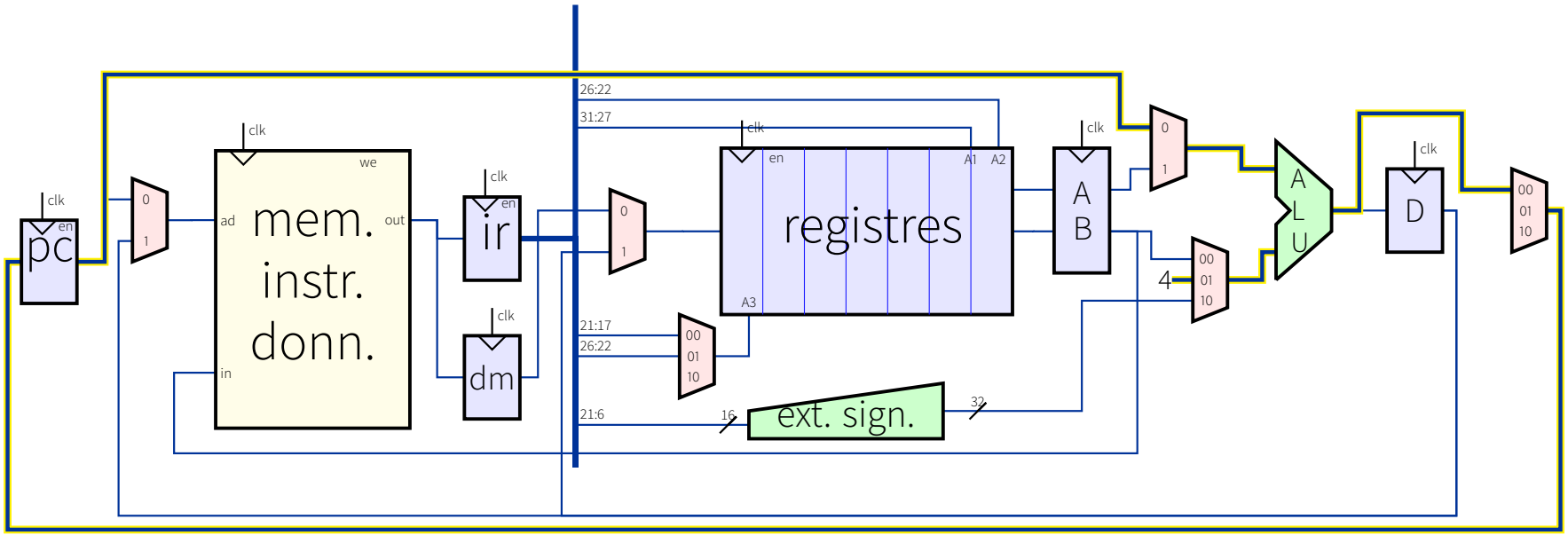
Lors de la phase de *fetch*, incrémentation du compteur de programme **pc**.

`pc <- pc+4`

Chemin de **pc** vers l'entrée de l'ALU

Immédiat "4" à la deuxième entrée de l'ALU

Chemin de la sortie de l'ALU vers l'entrée de **pc**

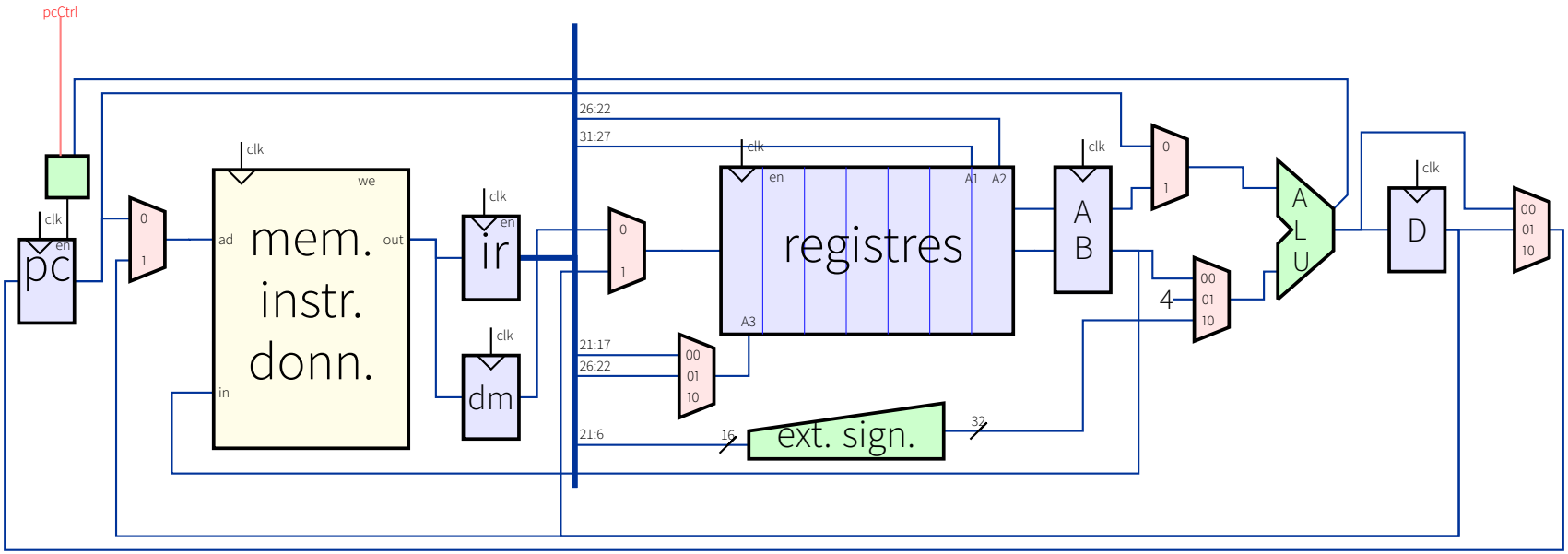


Branchements

Les branchements nécessitent :

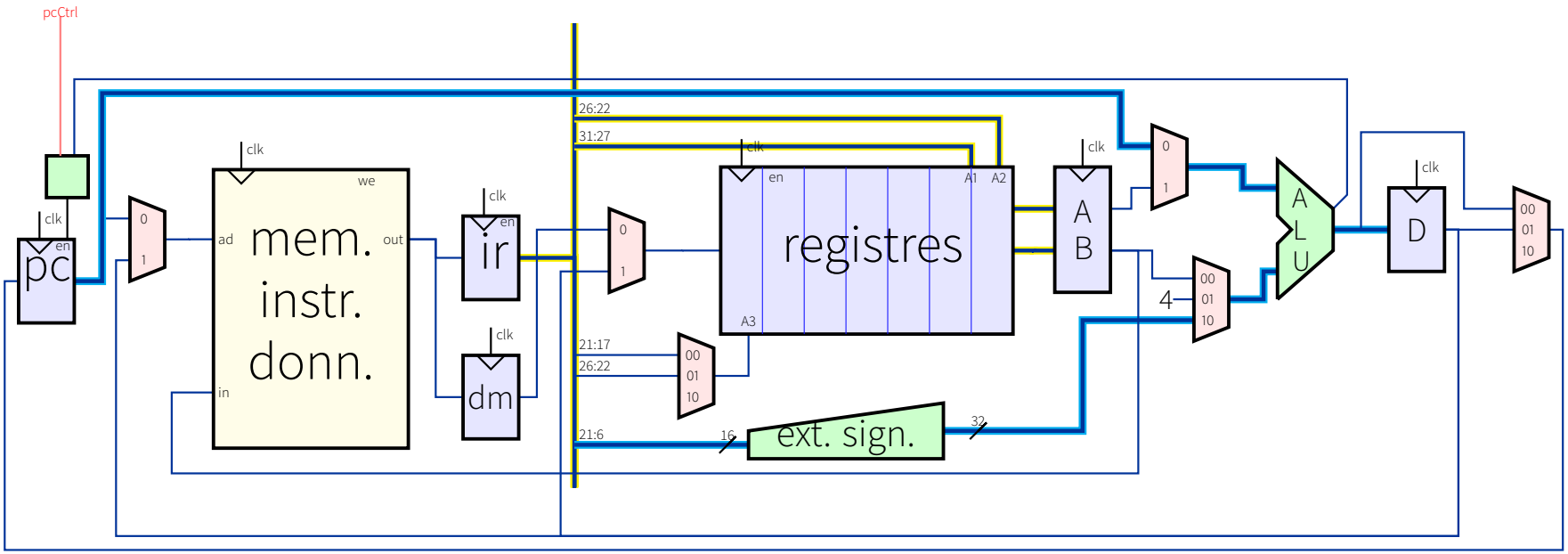
- de calculer la somme de **pc** et d'un $\texttt{imm}_{16}$ en recopiant le résultat dans **pc**. Ceci peut se faire par l'ALU avec les chemin de données existant.
- d'utiliser l'ALU pour effectuer une comparaison. On sortira un bit indiquant le résultat de la comparaison (**eq,ne, gt,** etc) de l'ALU.

Par contre, comme l'ALU est utilisée deux fois, il n'est pas possible de faire ces deux opérations dans une même phase.



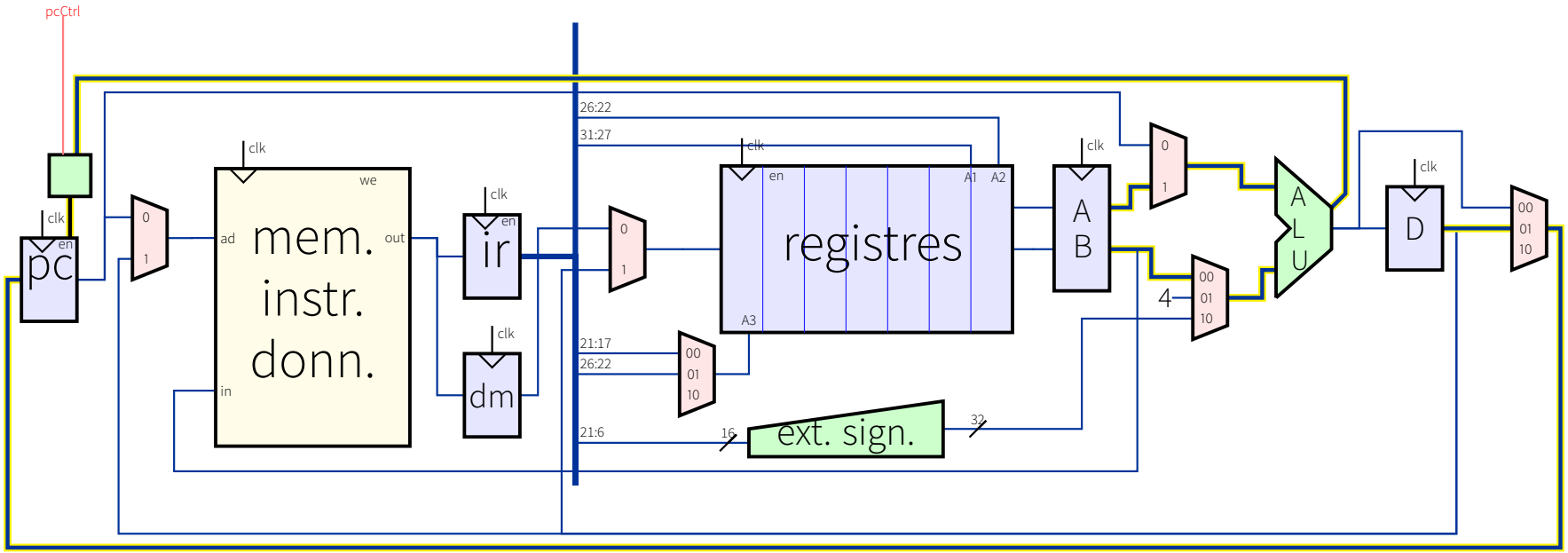
Branchements (cont.)

On modifie la phase *decode*.
En plus des opérations normales de la phase *décode* (___)
on calcule *toujours* la somme de *pc* et de l'immédiat (___)
L'adresse (éventuelle) de branchement est dans le registre **D**.



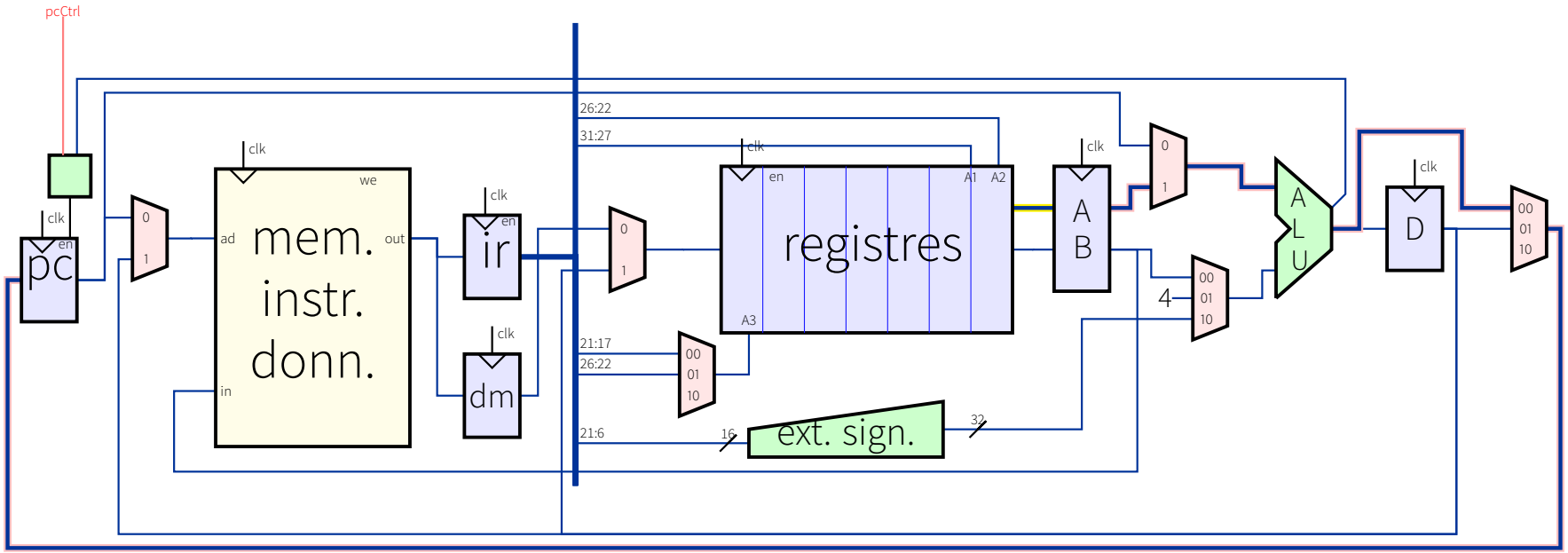
Branchements (cont.)

On peut ensuite effectuer le branchement.
On utilise l'ALU pour faire la comparaison entre les registres **A** et **B** et valider (ou non) l'autorisation d'écriture de **pc**
Simultanément, on amène le contenu du registre D vers l'entrée du registre **pc**.



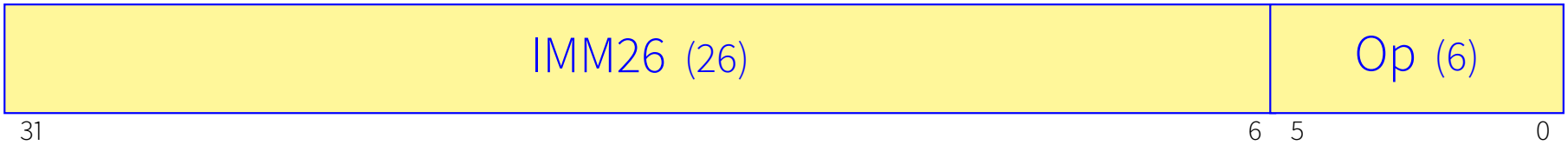
Sauts

Les instructions de sauts existent sous deux formes.
Dans la version registre **`jmp`**, il faut copier le contenu du registre **`rA`** dans PC.
Peut se faire en phase 3 sans modification architecturale.
En phase 2 (**`decode`**), rA est copié vers A
En phase 3, on le transfère vers pc



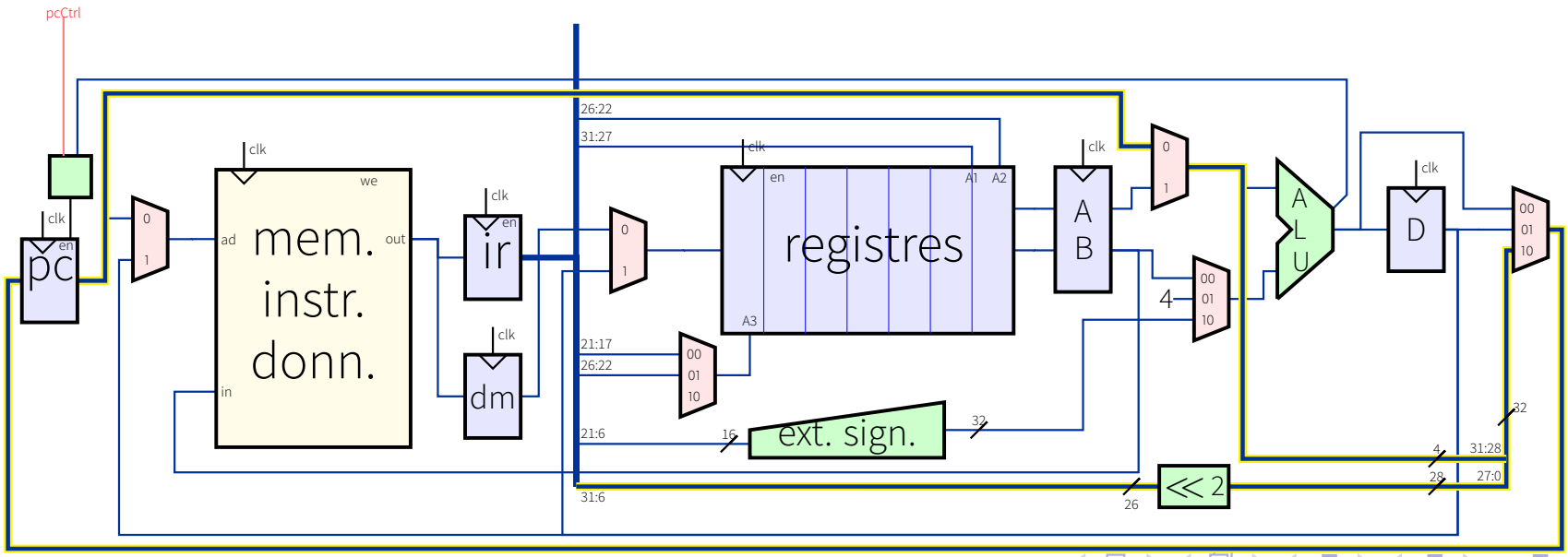
Sauts (cont.)

Dans la version avec immédiat **`jmpi`**, utilisation d'un immédiat sur 26 bits.



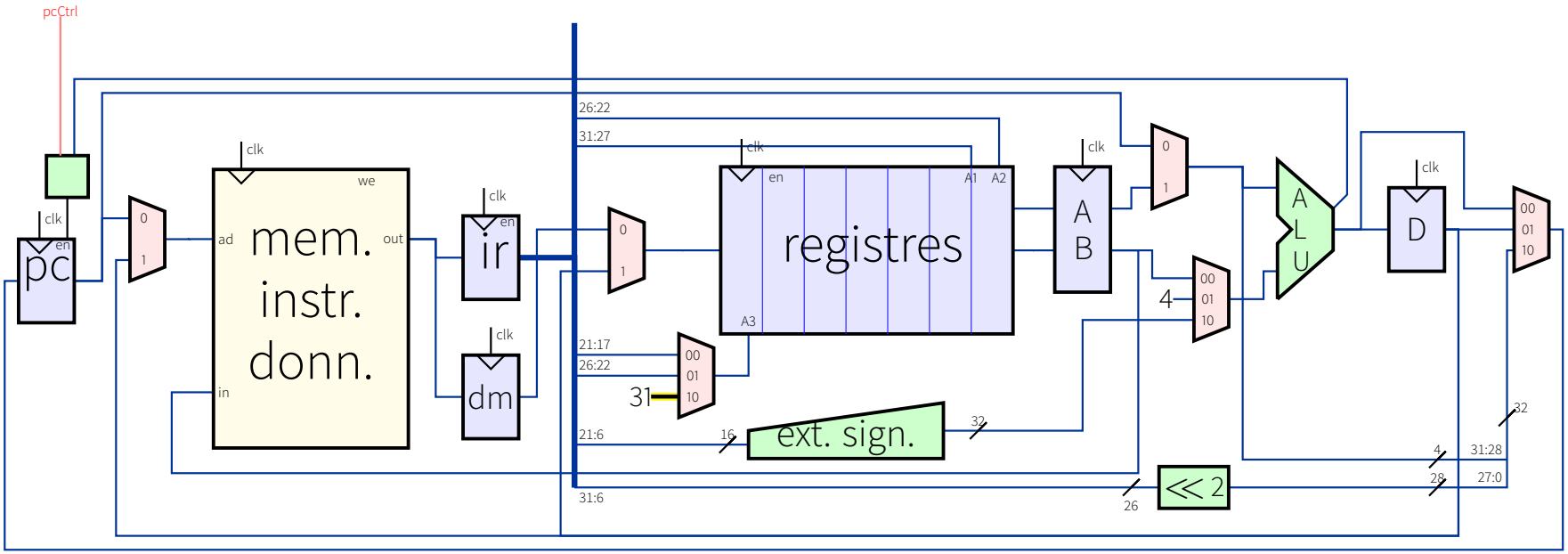
Les 26 de poids fort de l'instruction forment un immédiat, qui doit être :

- multiplié par 4
- complété en poids fort par les 4 bits de poids forts de **pc**
- copié dans **pc**



Appels de procédure

Similaires aux sauts, mais il faut pouvoir réécrire `pc` dans le registre `ra` (`r31`)
Ajout de la génération d'une adresse à 31 dans la sélection du registre d'écriture.[1]

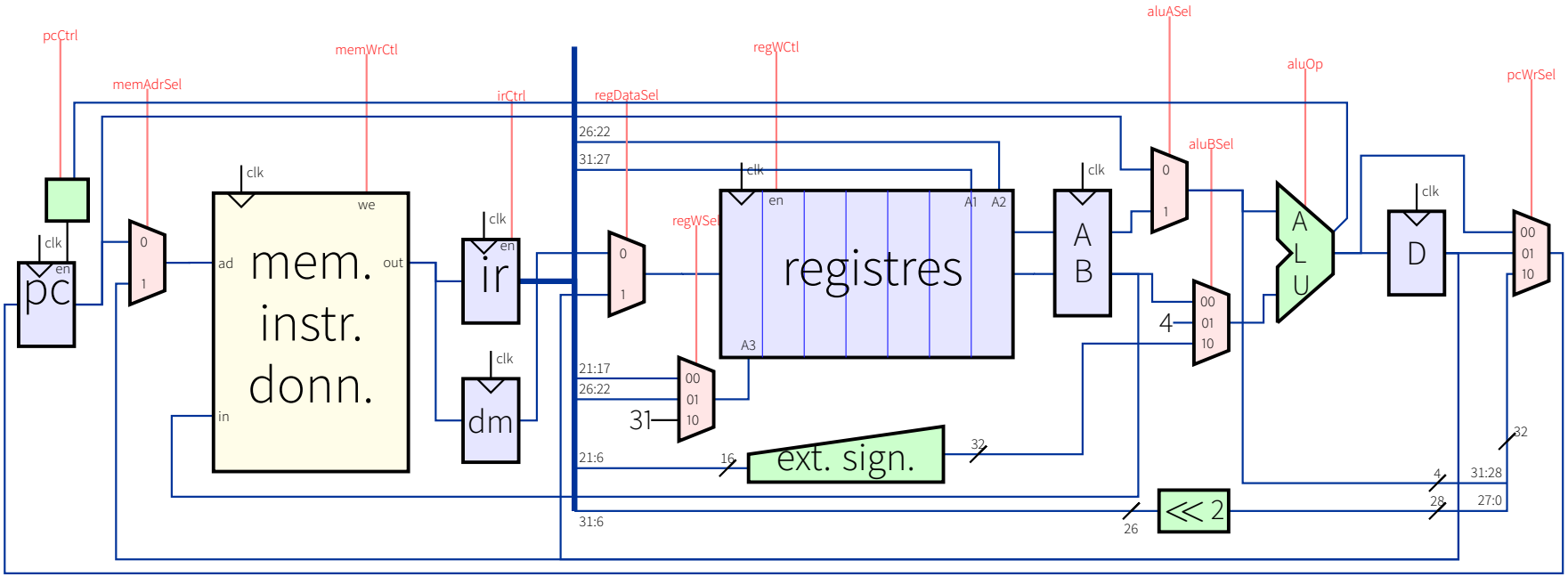


[1] La mise en oeuvre sera étudiée en TD

Contrôle du processeur

Nécessité de générer des commandes pour :

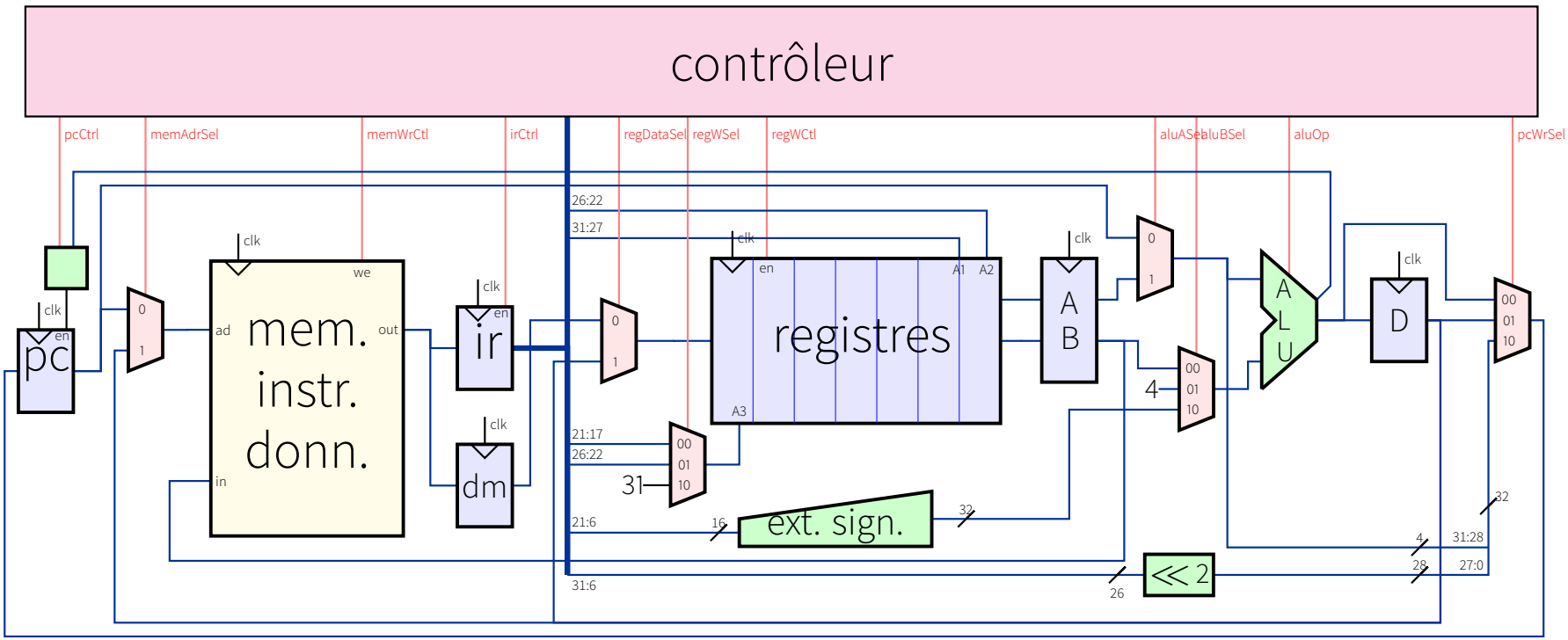
- registres avec contrôle d'écriture (*enable*)
- multiplexeurs
- unité arithmétique et logique



Contrôle du processeur (cont.)

Se fait au moyen d'un séquenceur (ou *contrôleur*) entrée : registre d'instruction, sorties : signaux de contrôle de l'architecture :

- **`memAdrSel`** sélection adresse mémoire et **`memWrCtl`** contrôle écriture
- **`irCtrl`** écriture registre instruction
- **`regWCtl`** contrôle écriture du banc de registres, **`regDataSel`** sélection source de données et **`regWSel`** sélection du registre écrit
- **`aluOp`** opération ALU, **`aluASel`** et **`aluBSel`** sélection des opérandes
- **`pcCtrl`** contrôle écriture de pc, **`pcWrSel`** sélection source

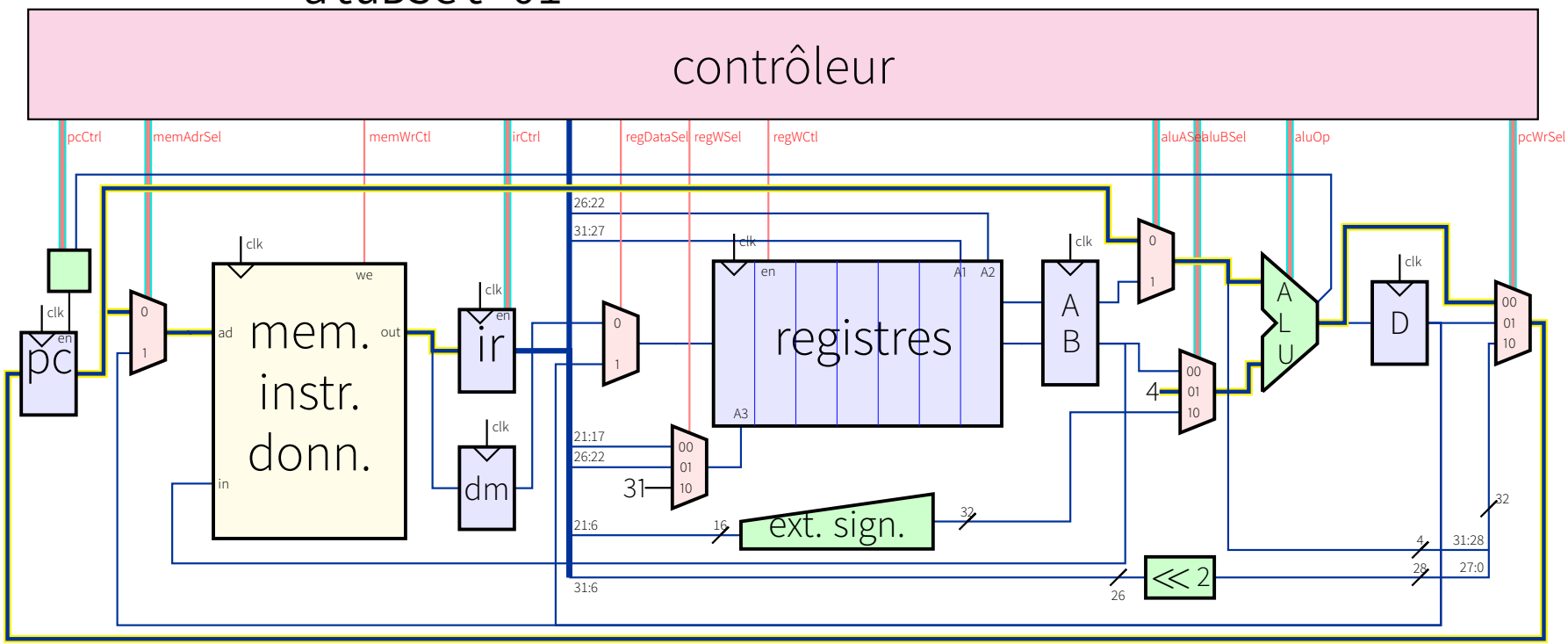


Contrôle du processeur (cont.)

Premier état : acquisition de l'instruction *fetch*

Correspond à
```
pcWrSel=00
pcCtrl=wr
memAdrSel=0
irCtrl
aluOp=add
aluASel=0
aluBSel=01
```

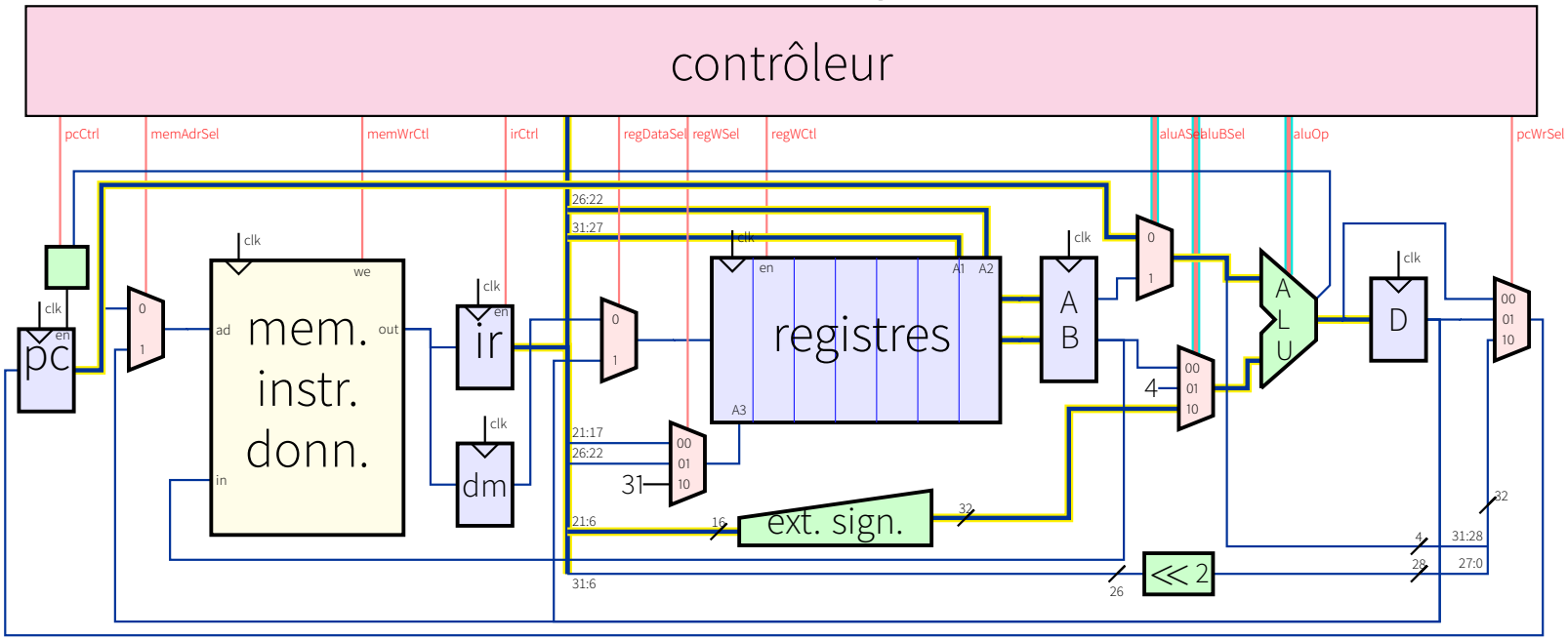
fetch:
```
ir <- mem[pc]
pc <- pc + 4
```



Contrôle du processeur (cont.)

Deuxième état : décodage de l'instruction *decode*
identique pour toutes les instructions
Ecriture dans les registres **A** et **B** systématique (aucun contrôle).
Le calcul `D <- pc + imm16` est nécessaire pour les branchements.
Correspond à `aluOp=add`
`aluASel=0`
`aluBSel=10`

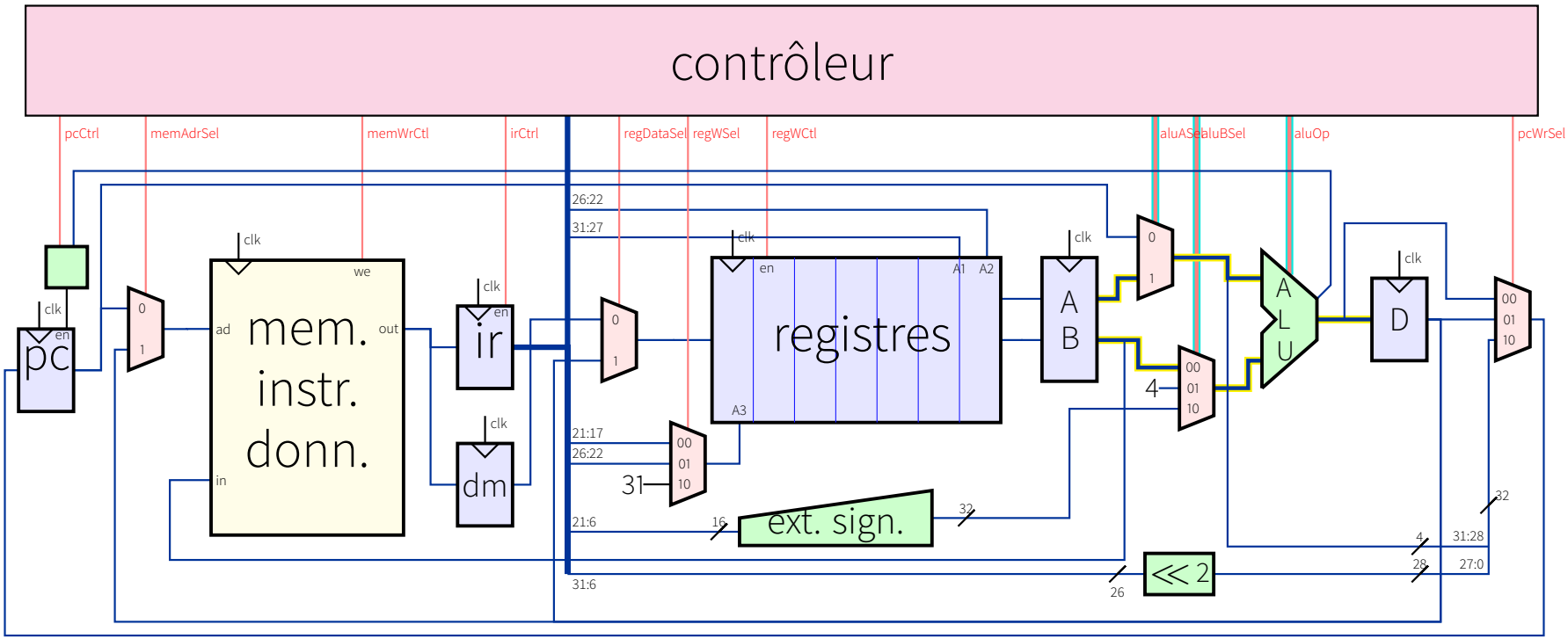
decode:
`A <- rA`
`B <- rB`
`D <- pc + imm16`



Contrôle du processeur (cont.)

Les autres états dépendent des types d'instruction.
Exemple : addition entre registres (et toutes des instructions arithmétiques et logiques de type R). *add*
Correspond à `aluOp=add`
`aluASel=1`
`aluBSel=00`

add-execute:
`D <- A + B`

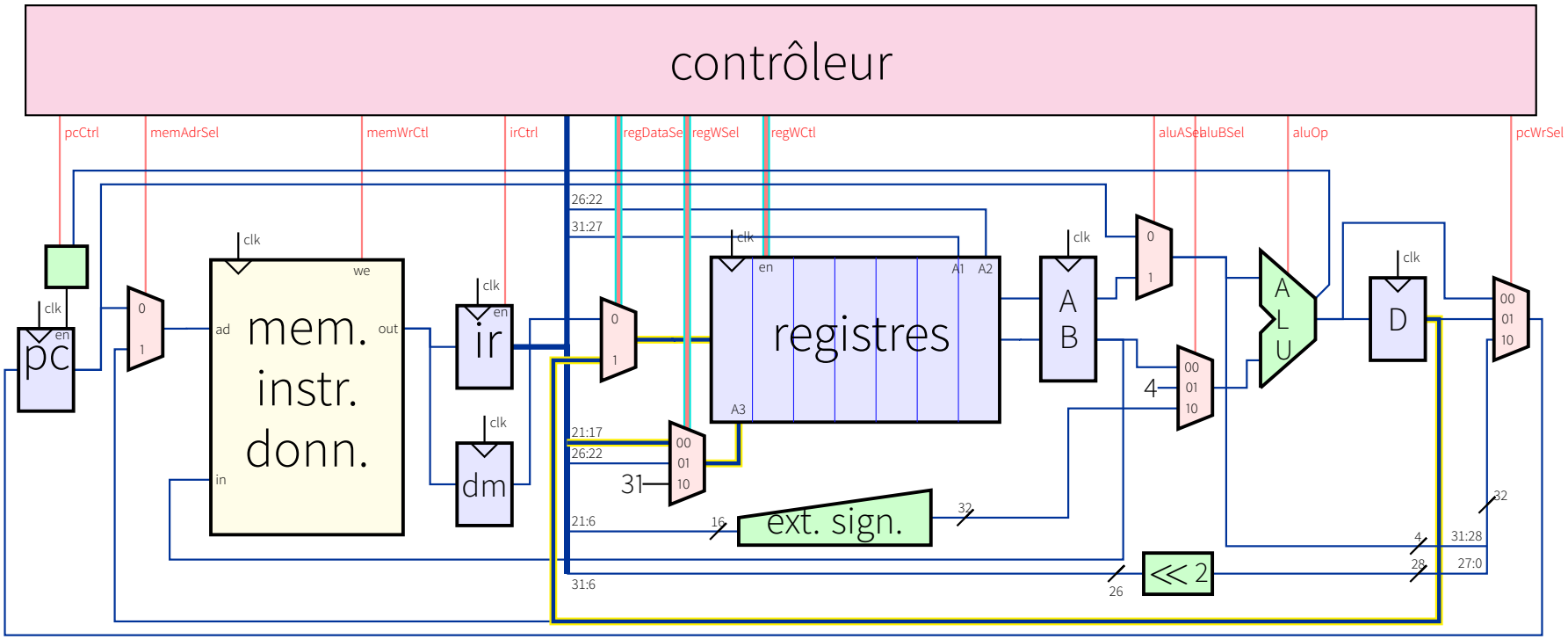


Contrôle du processeur (cont.)

La dernière phase de l'exécution d'une addition est une réécriture du résultat dans le registre `rC`

Correspond à `regWCtl`
`regDataSel=1`
`regWSel=00`

add-rr:
`rC <- D`



Contrôle du processeur (cont.)

On peut donc décrire l'instruction `add` par

1. `fetch`
2. `decode`
3. `D <- A + B`
4. `rC <- D`

On peut faire de même pour toutes les instructions :
`addi` (et toutes instruction arithmétiques et logiques avec un immédiat)

1. `fetch`
2. `decode`
3. `D <- A + imm`$_{16}$
4. `rB <- D`

Contrôle du processeur (cont.)

Instructions d'accès mémoire

`load`

1. `fetch`
2. `decode`
3. `D <- A + imm16`
4. `dm <- mem[D]`
5. `rB <- dm`

`store`

1. `fetch`
2. `decode`
3. `D <- A + imm16`
4. `mem[D] <- B`

Contrôle du processeur (cont.)

Instructions de contrôle de flot

`br`	`jmp`	`jmpi`
1 `fetch`	1 `fetch`	1 `fetch`
2 `decode`	2 `decode`	2 `decode`
3 `if(pcComp==1)` `pc <- A+imm`$_{16}$	3 `pc <- rA`	3 `pc<-pc`$_{31:28}$`:(imm`$_{26}$`<<2)`

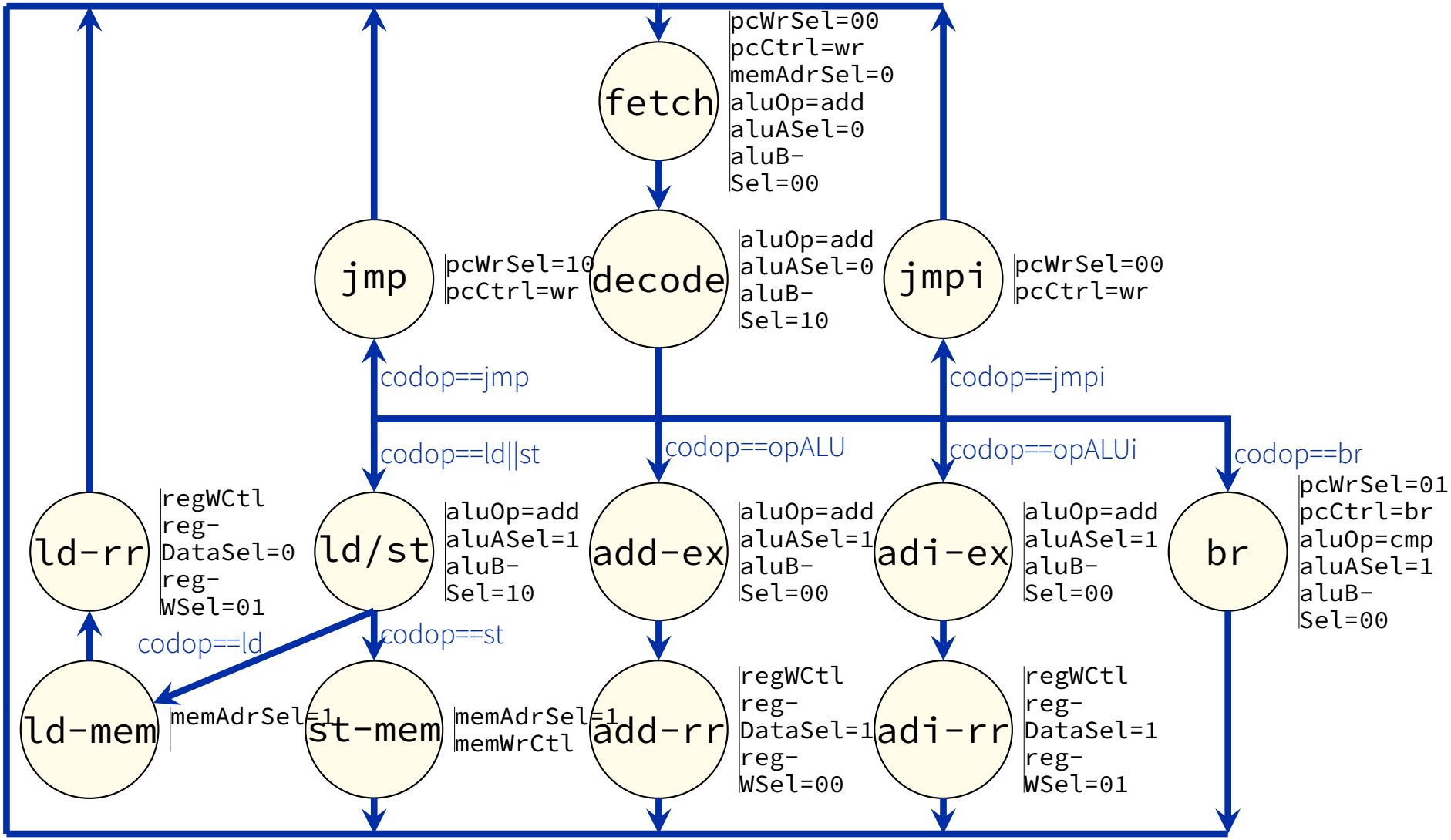
NB : Le signal de contrôle d'écriture de `pc` doit être à 1 :

- dans la phase `fetch`
- dans la phase 3
  - pour un `jmp` ou un `jmpi`
  - pour un `br` si la comparaison est vraie

Le contrôleur génère 2 bits : `wrpc` et `br` et l'écriture de `pc` est vraie si (`wrpc` || (`br` && `pcComp`))

Contrôle du processeur (cont.)

On peut en déduire l'automate de contrôle



Mise en oeuvre pipeline du processeur NIOS II

Alain MÉRIGOT

Université Paris Saclay

Principe du traitement pipeline

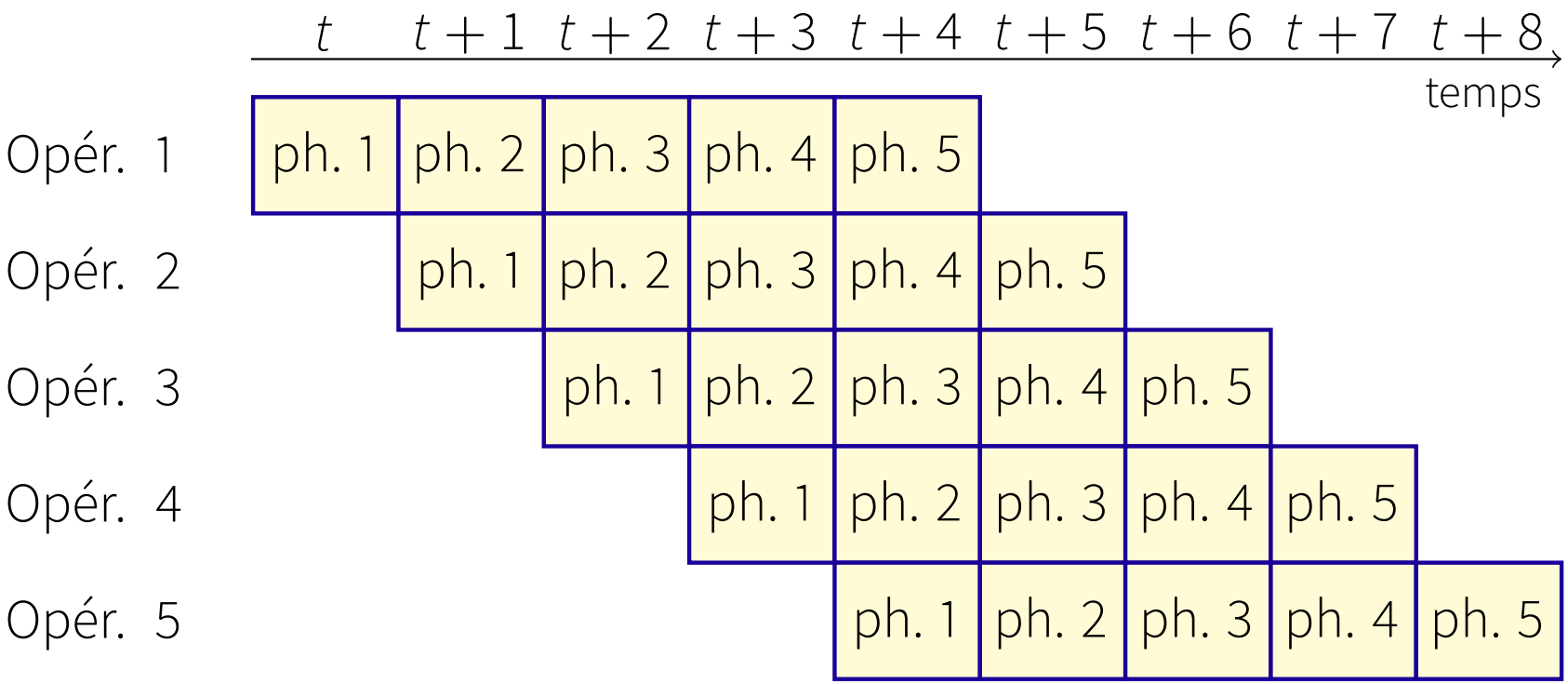
- traitement décomposé en n phases
- tous les traitements ont le même nombre de phases
- pas d'interférences entre phases

$\rightarrow$ possibilité de pipeline

On démarre une instruction, sans attendre la fin de l'exécution de la précédente.

Lancement d'une opération par cycle
(mais latence inchangée)

Principe du traitement pipeline (cont.)



Après une phase de démarrage, 5 opérations sont simultanément effectuées.

Principe du traitement pipeline (cont.)

Si le pipeline a *n* étages, les performances sont (*presque*) multipliées par *n*.

Contraintes :

- *Toutes* les instructions doivent comporter le même nombre de phases.
- Aucune phase ne doit utiliser des ressources dont une autre phase aurait aussi besoin. (*conflit structurel*)
  Chaque ressource (opérateur, bus, registre) est liée à une phase spécifique.

Nécessité d'adapter l'architecture.

Nécessité de gérer les intéractions éventuelles

Version *pipeline* du nios

Dans le cas du Nios II, différentes durées d'instructions (3 phases pour `br, jmp, ...`, 4 pour instructions ALU `add, addi,...` et `st` et 5 pour `ld`).
→ On passe toutes les instructions à **5 phases**

Des ressources sont utilisées plusieurs fois :

mémoire utilisée par *toutes* les instructions en phase *fetch* et par `ld,st` en phase 4
→ séparation mémoire instructions – mémoire données

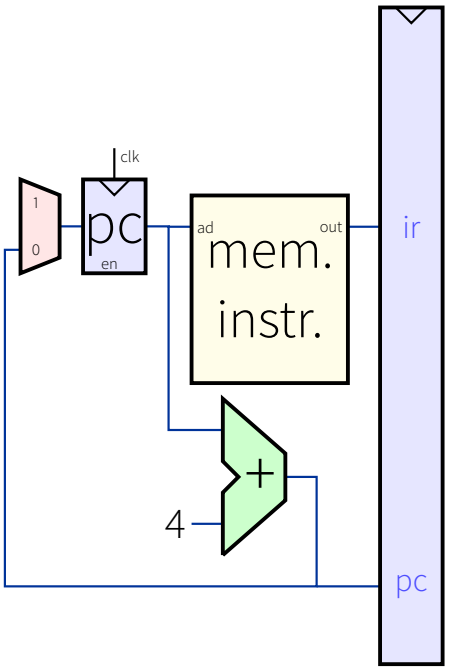
ALU utilisées pour la phase *execute*, et les calculs `pc+4`, `pc+imm16`...
→ ajout d'additionneurs pour les calculs sur `pc`

registre d'instruction `ri` nécessaire à chaque phase pour le contrôle

compteur de programme `pc` nécessaire en phase *fetch* et dans d'autres phases pour `br, jmpi`
→ on copie tous les registres à préserver (`ri`, `pc`, registres tampons `A`, `B`, etc) dans de grands ***registres pipeline***

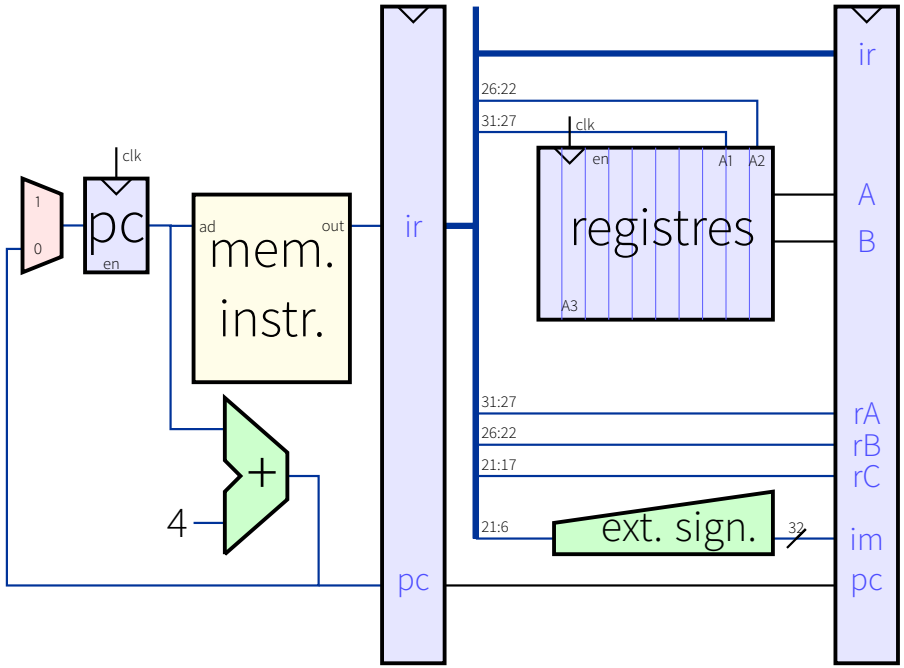
Acquisition de l'instruction

Phase *fetch* ou **lecture instruction** (LI)
On ajoute un additionneur pour l'incrémentation de `pc`.
`ir` et `pc` sont copiés dans un registre pipeline à la fin de cette phase.



Décodage de l'instruction

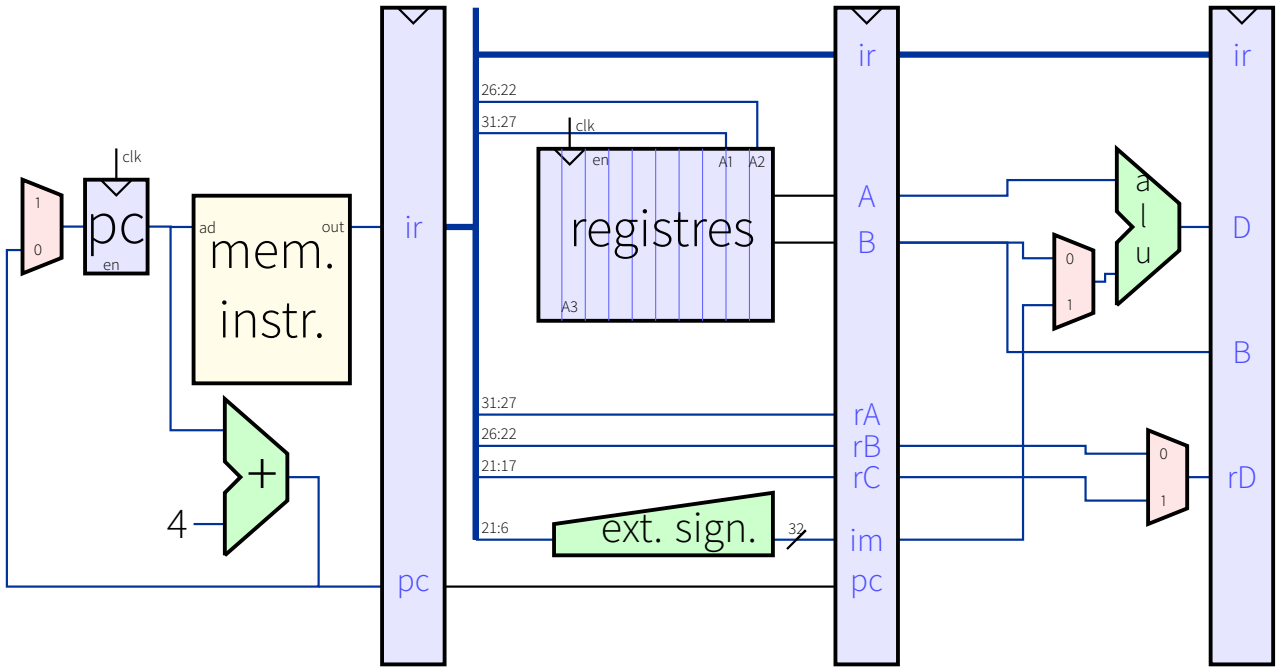
Phase *decode* ou **décodage instruction** (DI)
Extraction des opérandes (`A`, `B`, `imm16`) et copie dans le registre pipeline
Les numéros des registres concernés par l'instruction (`ra`, `rb`, etc) sont également transmis.
`ir` est copié dans le registre pipeline sous forme décodée (commandes pour chaque opérateur/registre)



Exécution de l'instruction

Phase *execute* (EX)
On effectue le calcul requis par l'instruction entre les opérandes : A et soit B, soit `imm16`.
On détermine également le numéro du registre où sera écrit le résultat :

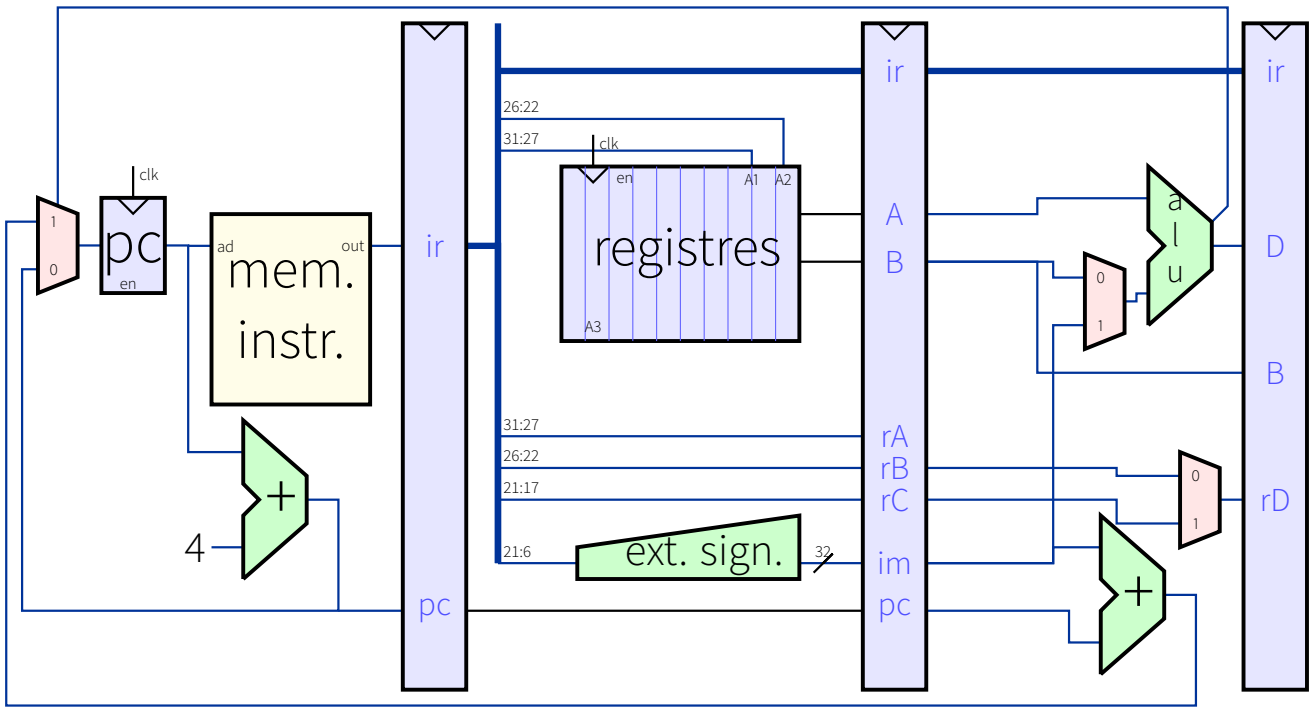
- `rC` pour une instruction de type R
- `rB` pour une instruction de type I



Exécution de l'instruction (cont.)

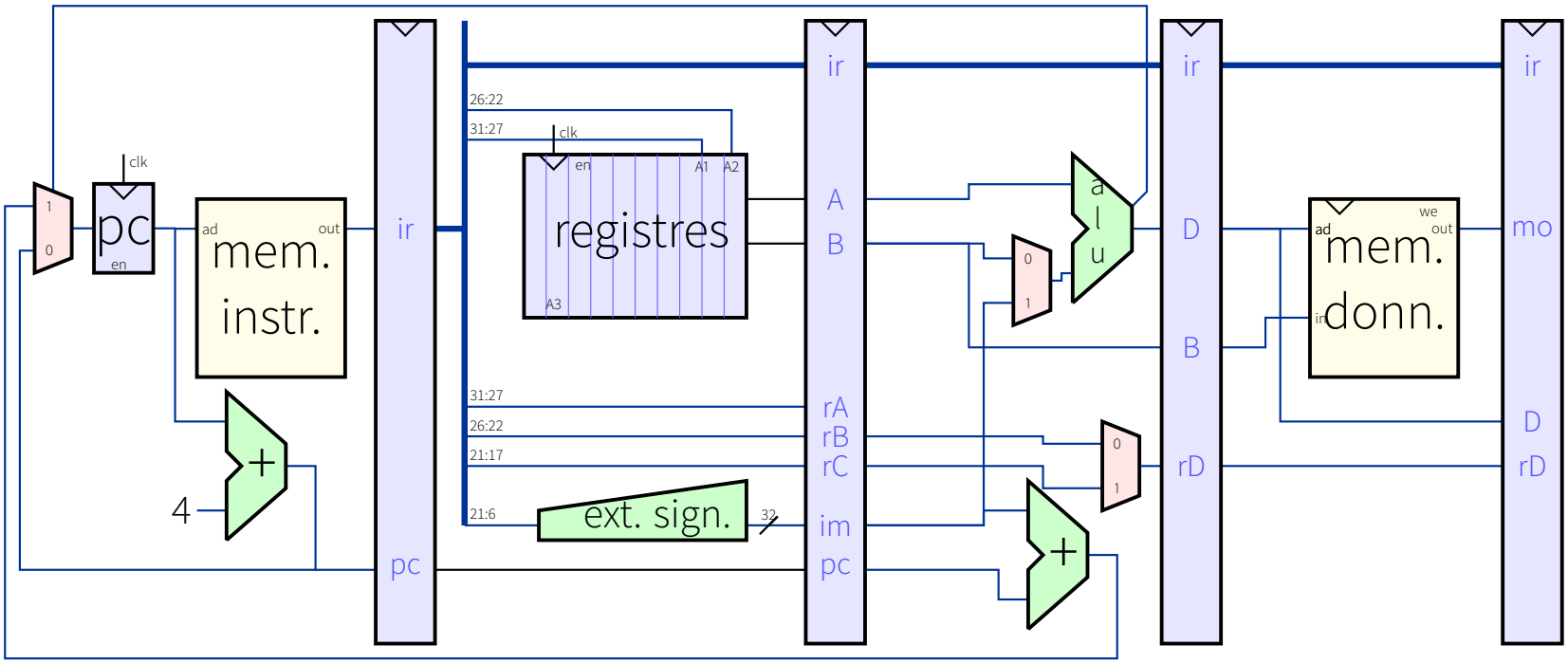
Durant la phase EX, on effectue également les opérations nécessaires pour un branchement :

- calcul de l'adresse de branchement `pc+imm16`
- comparaison entre les opérandes **A** et **B**. Le résultat détermine si `pc` est mis à jour par le mécanisme normal (`pc<-pc+4`) ou par l'adresse de branchement (`pc<-pc+imm16`) suivant la comparaison (ALU).



Accès mémoire

Tous les accès à la mémoire de données se font durant la phase MM.
L'adresse mémoire pour `ld` ou `st` résulte du calcul `A+imm16` et est dans `D`.
Pour un `st`, la donnée à écrire en mémoire est dans `B`
Pour les instructions autres que `ld` ou `st`, on recopie durant cette phase le registre `D` et le numéro du registre destination `rD`.

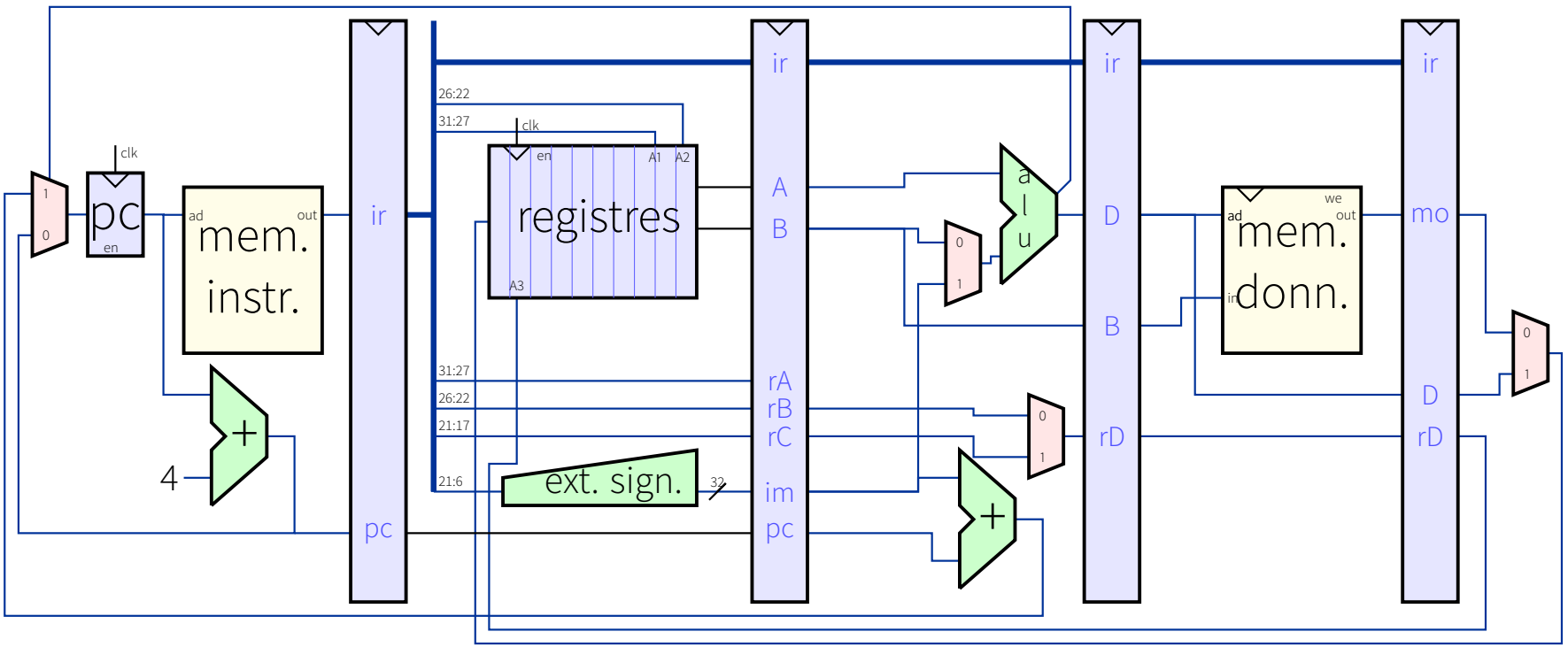


Réécriture des résultats

La phase RR permet la réécriture des résultats vers le banc de registres.
La donnée à réécrire provient :

- de la mémoire pour un `ld`
- de l'ALU (via **D**) pour une instruction arithmétique et logique

`rD` contient le numéro de registre à écrire.



Récapitulatif

Pipeline en 5 étages

LI Lecture de l'instruction. On va chercher l'instruction dans `Mem[pc]` et on incrémente `pc`

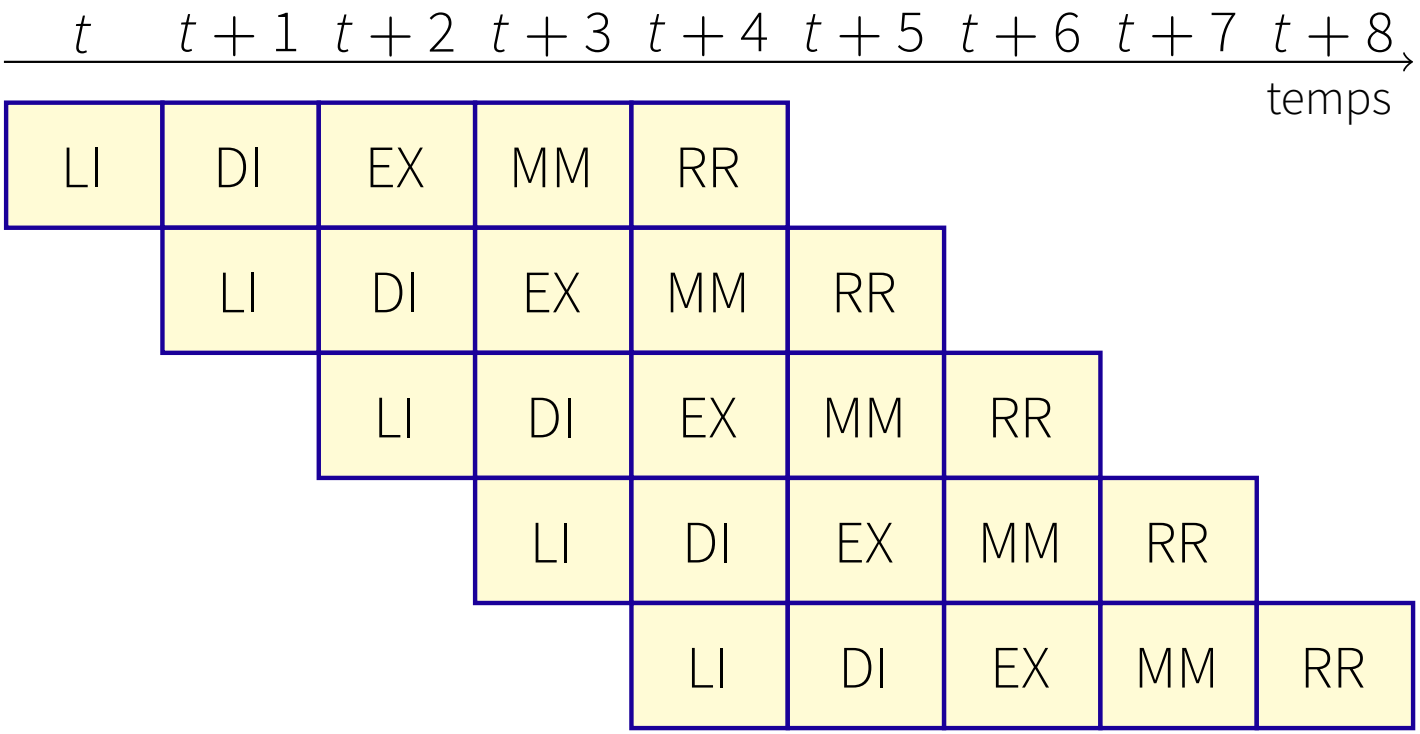
DI Décodage de l'instruction. Pendant ce temps, on lit les données dont on aura besoin dans le banc de registres et dans le registre d'instruction

EX Exécution de l'instruction. On utilise l'unité arithmétique pour effectuer le calcul nécessaire ou calculer l'adresse mémoire (adressage basé).

MM Accés MéMoire. Pour les instructions mémoire on lit ou on écrit la mémoire. Pour les autres instructions, aucune opération, mais transfert des registres tampon.

RR Rangement des résultats. On réécrit les données calculées ou lues en mémoire dans le registre destination.

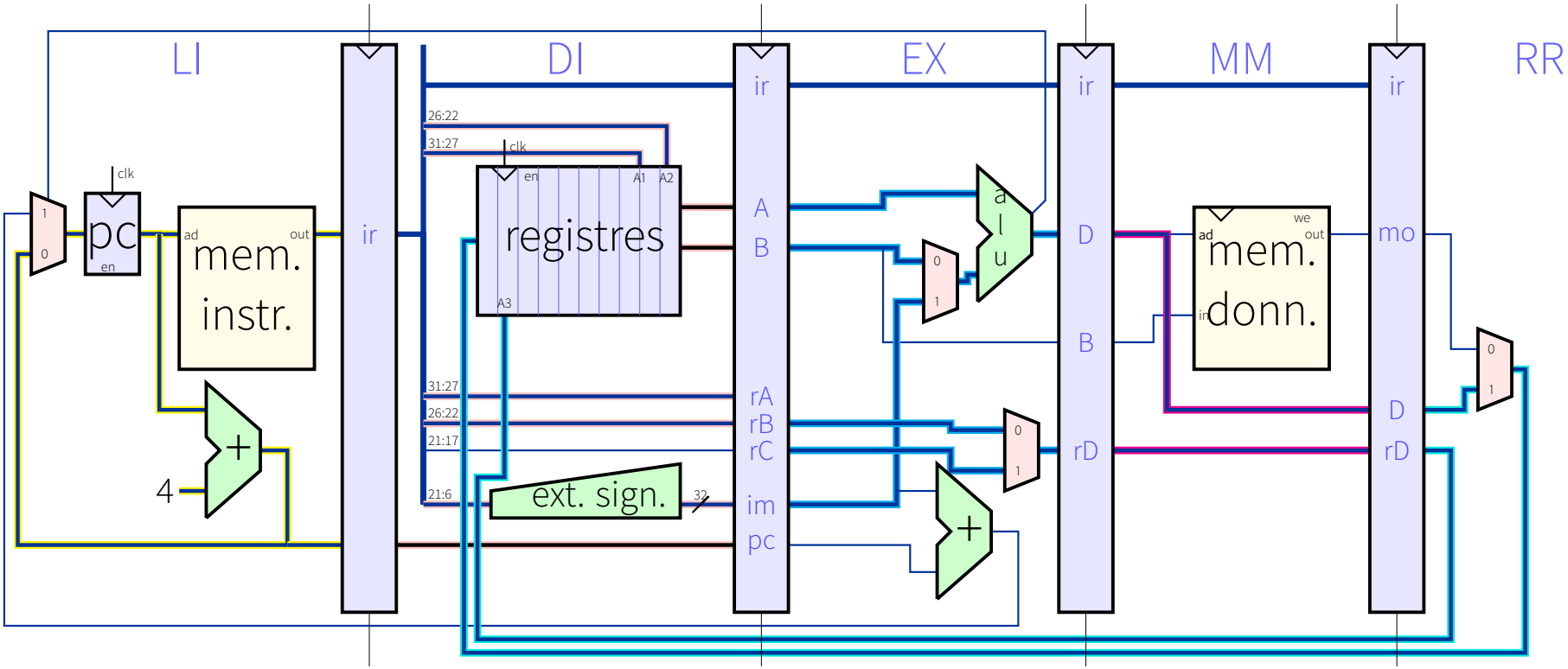
Le pipeline nios



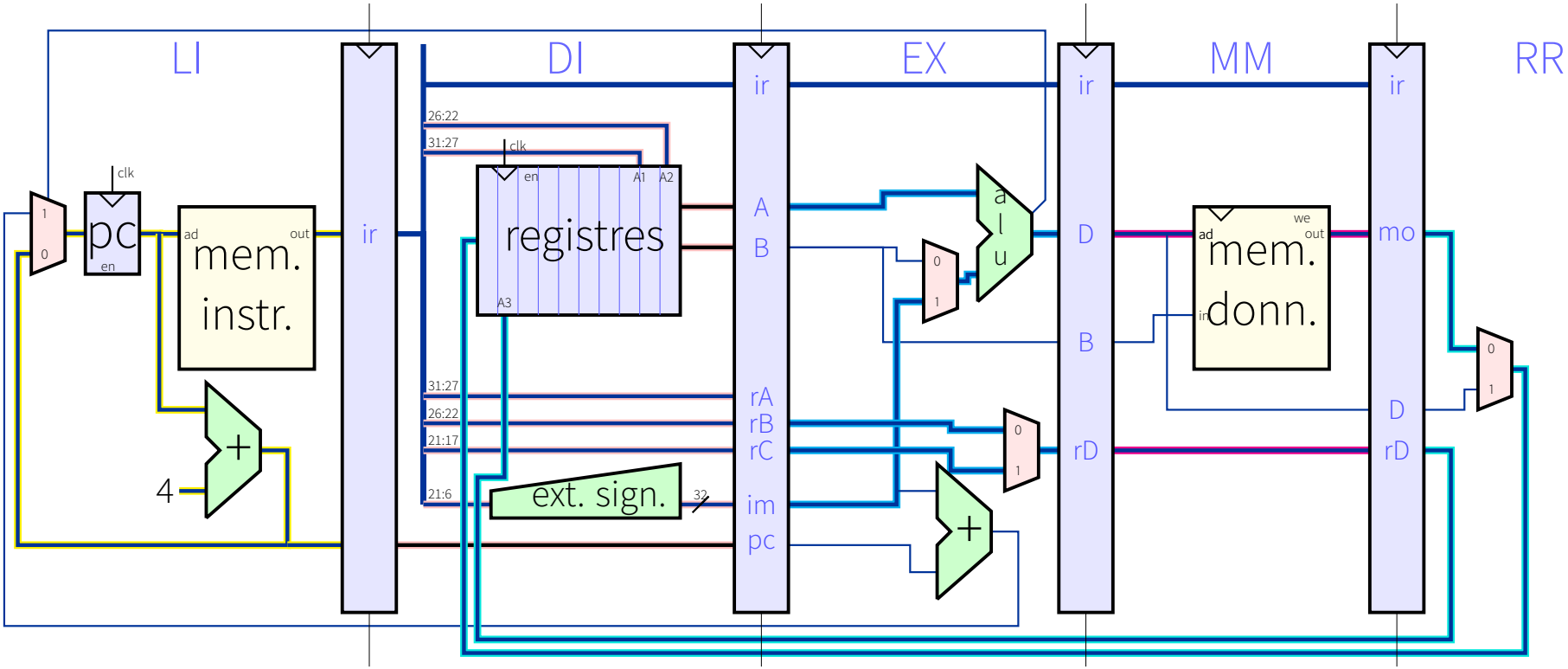
« pipeline RISC simple »

Instructions arithmétiques et logiques

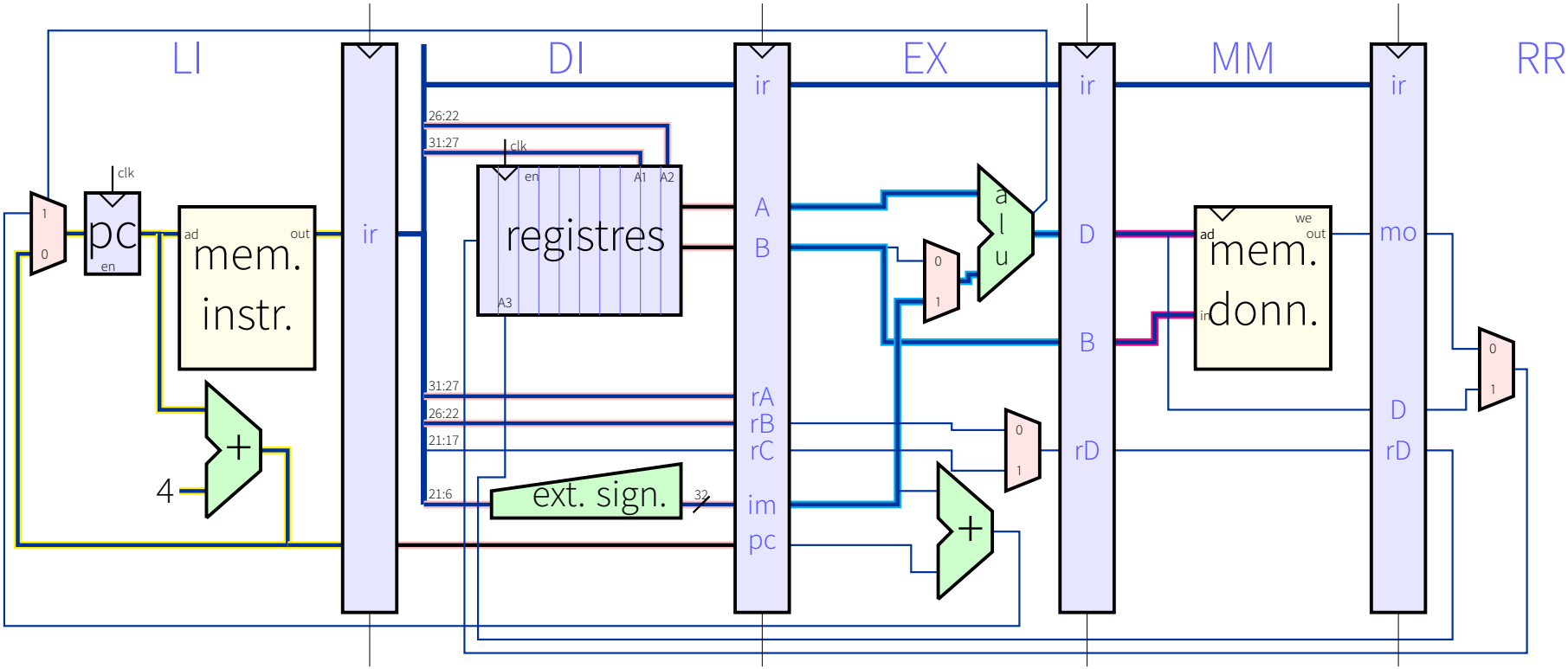
(LI ——, DI ——, EX ——, MM ——, RR ——)



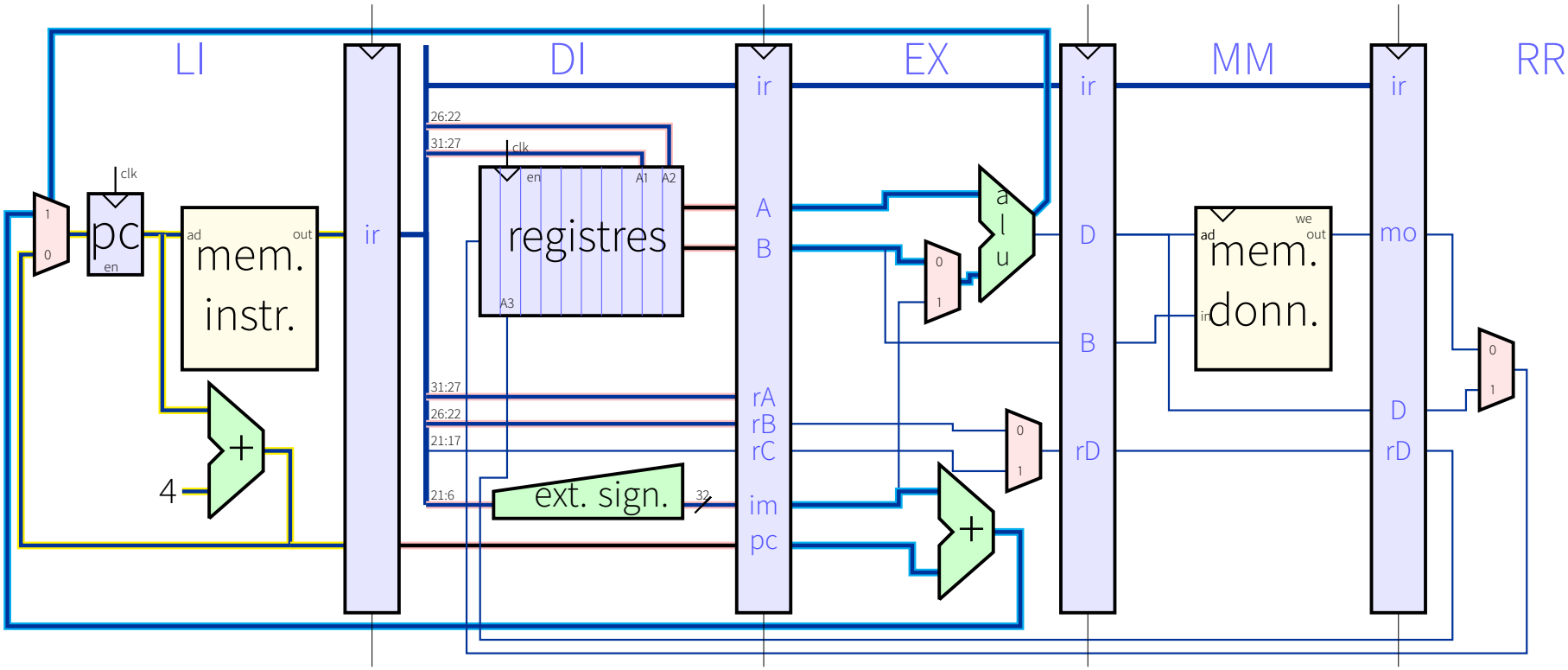
Lectures mémoire (`ld`)



Ecritures mémoire (`st`)



Branchements



Récapitulatif (cont.)

Le nios pipeliné existe en deux versions dans les FPGA Atera/Intel: nios II/s (5 étages de pipeline) et nios II/f (6 étages). On retrouve des pipelines similaires de 5 à 7 étages dans les processeurs embarqués simples actuels (arm-v6, mips4).

Pour les processeurs plus performants (pentium, power-PC, arm-v7, arm-v8, etc), le pipeline est plus important (10 à 20 étages).

- permet la mise en oeuvre d'opérations complexes (flottant)
- permet d'augmenter la fréquence de fonctionnement

Aléas dans les pipeline

Un *aléa* est un événement qui perturbe le bon fonctionnement d'un pipeline.

Trois types d'aléas :

aléa de données lié à la non disponibilité d'une donnée pour éxécuter une certaine opération à cause du délai d'exécution

aléa de contrôle lié aux ruptures du flot d'exécution d'un programme (branchements, sauts, etc)

aléa structurel lié à l'existence d'une ressource matérielle partagée par deux étages ou plus

Aléa de données

Du fait qu'une instruction est lancée alors que la précédente n'est pas achevée, il y a un risque d'utiliser une donnée qui n'est pas encore mise à jour.

```
add  r1, r2, r3
add  r5, r4, r1
```

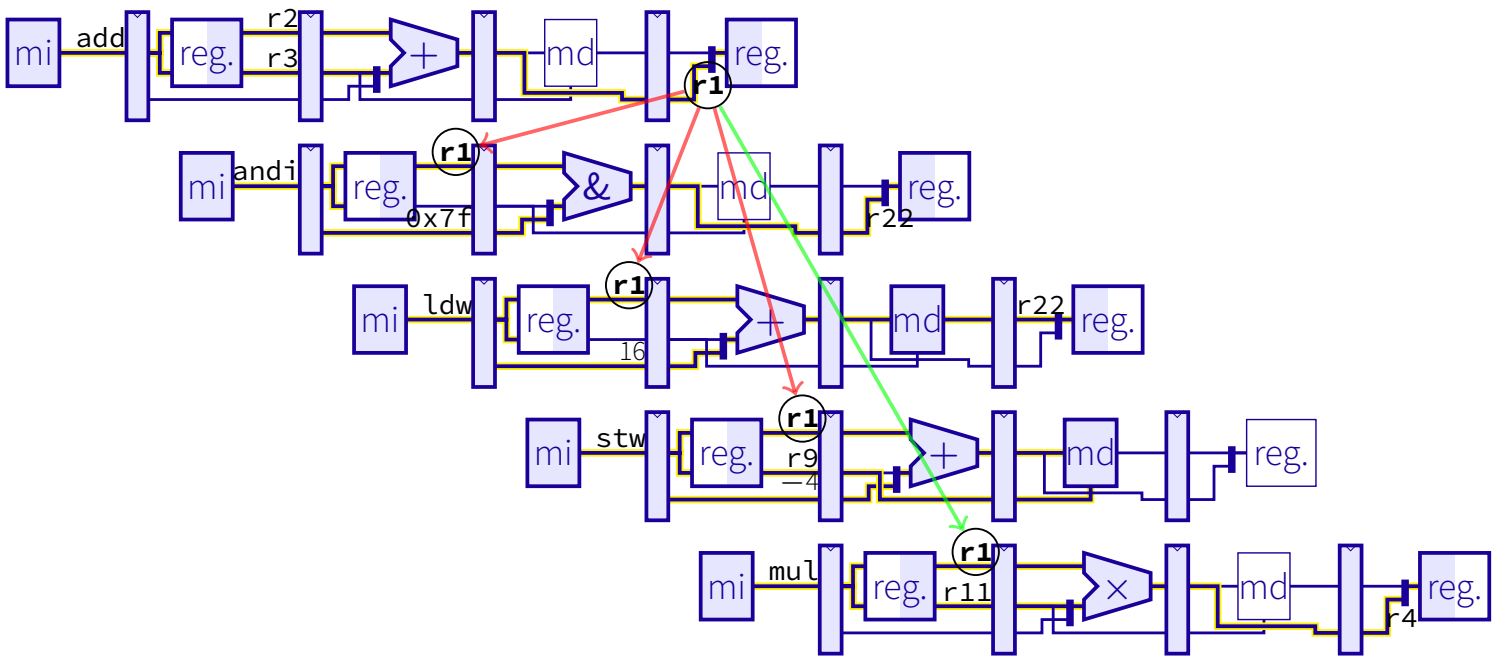
L'instruction 2 va utiliser *l'ancienne* valeur de **`r1`** et non celle qui vient d'être calculée.

Aléa de données

Aléa de données (cont.)

Le contenu des registres est chargé dans les registres pipeline **A** et **B** à la fin de l'étage **LI**.
Les registres **r1–r31** sont écrits à la fin de la phase **RR**.

1 add **r1**,r2,r3
2 andi r22,**r1**,0x7f
3 ldw r22,(16)**r1**
4 stw r9,(-4)**r1**
5 mul r4,**r1**,r11



L'instruction 1 intéragit sur les instructions 2, 3 et 4.
NB : *Tous* les registres sont :

- lus pendant une phase
- écrits à la fin de la phase

Aléa de données (cont.)

On peut supprimer certains aléas de données en allant chercher la nouvelle donnée là où elle est présente avant de faire le calcul lors la phase `EX`.

```
1 add r1,r2,r3
2 add r3,r1,r2
3 add r4,r1,r4
4 add r5,r1,r5
```

`add r1,r2,r3`

`add r3,r1,r3` la nouvelle valeur de `r1` est dans le registre pipeline `EX/MM.D`

`add r4,r1,r4` la nouvelle valeur de `r1` est dans le registre pipeline `MM/RR.D`

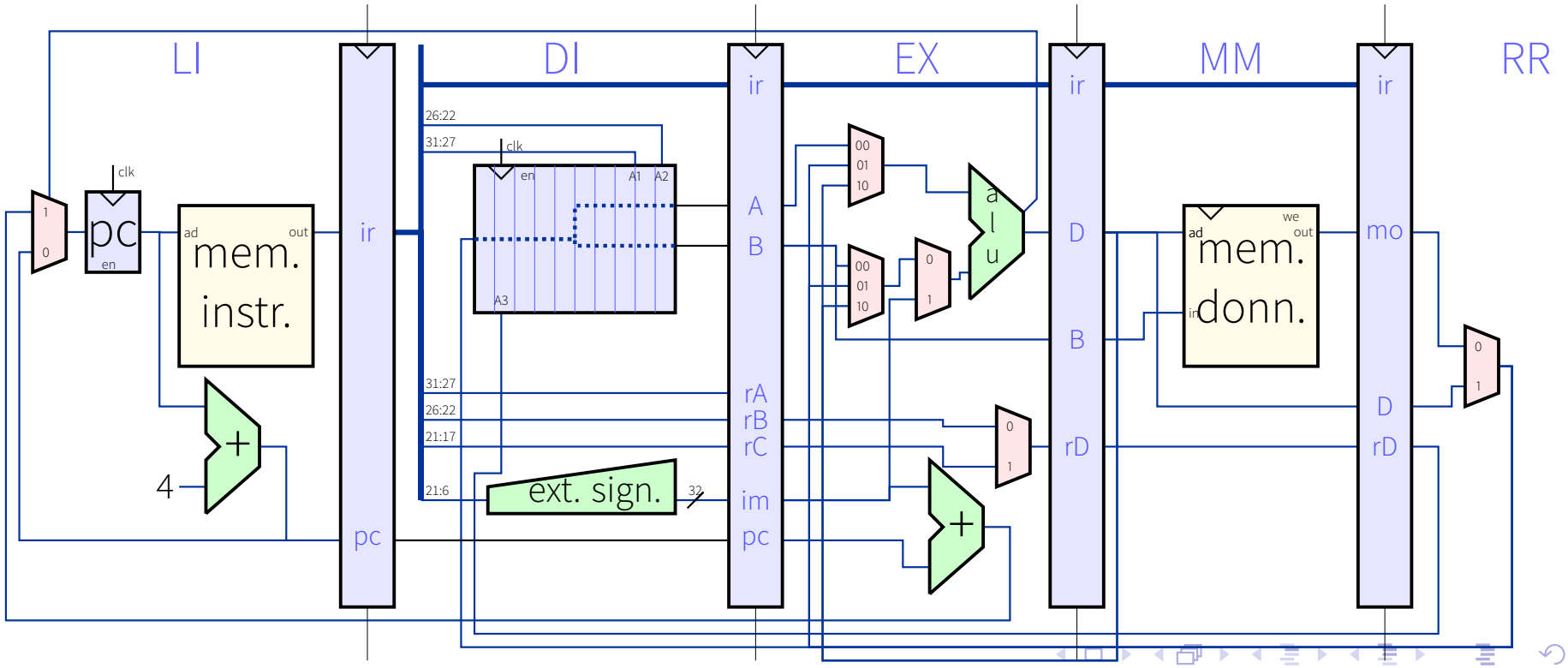
`add r5,r1,r5` le registre `r1` est *en train* d'être écrit dans le banc de registre

Mécanisme d'**envoi** (*forwarding*) (également appelé court-circuit (*bypass*)). On rajoute des chemins entre les registres pipeline où le résultat peut être présent et l'entrée de l'ALU.

Aléa de données (cont.)

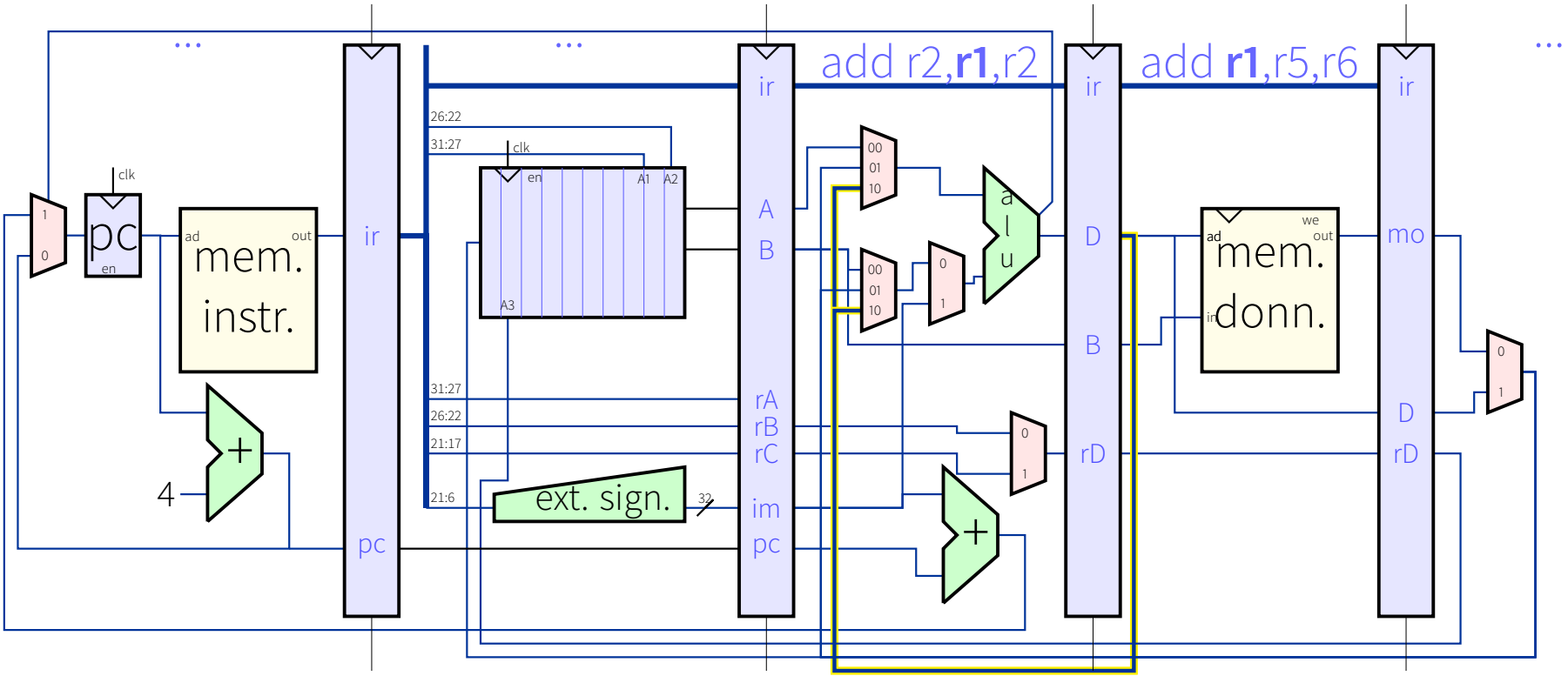
Modifications architecturales :

- ajouts de multiplexeurs aux entrées **A** et **B** de l'ALU
- chemin de EX/MM.D vers les entrées **A** et **B** de l'ALU
- chemin de MM/RR.D vers les entrées **A** et **B** de l'ALU
- transfert direct entre entrée du banc de registres et ses sorties



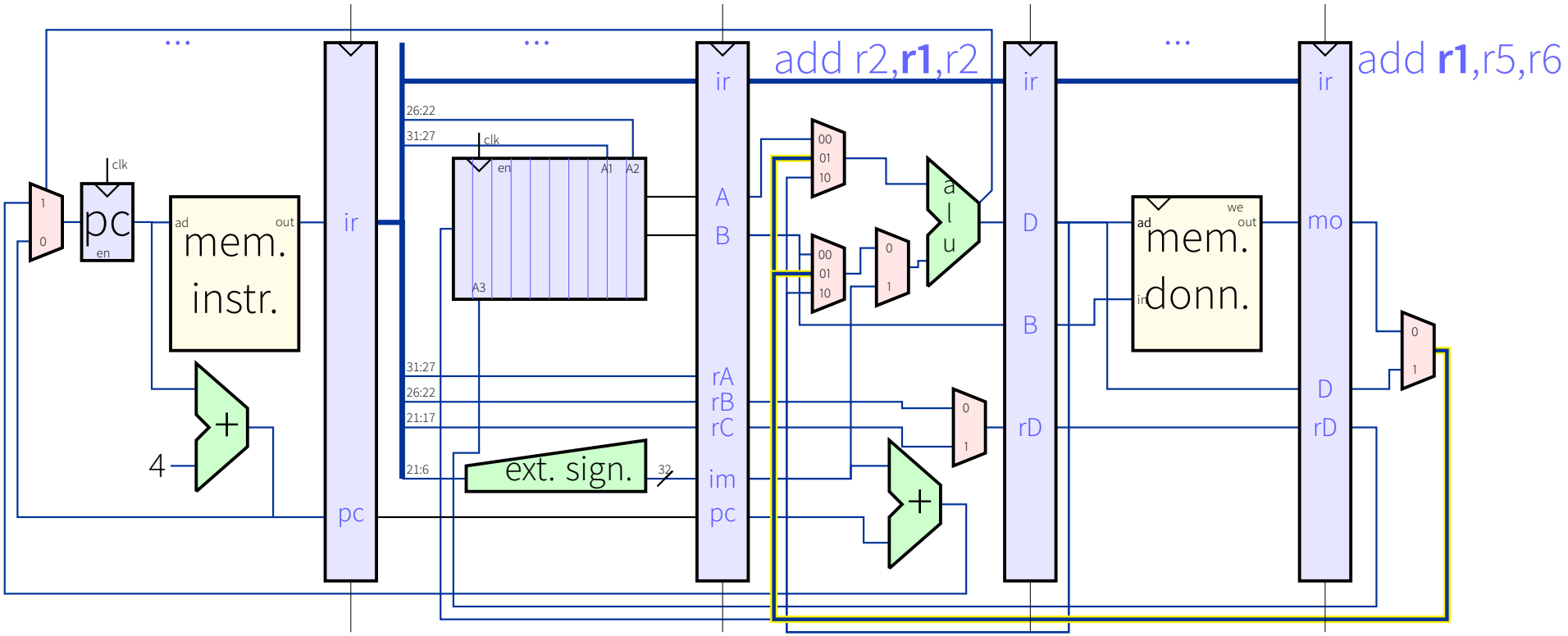
Aléa de données (cont.)

Cas d'une intéraction entre les étages MM et EX.



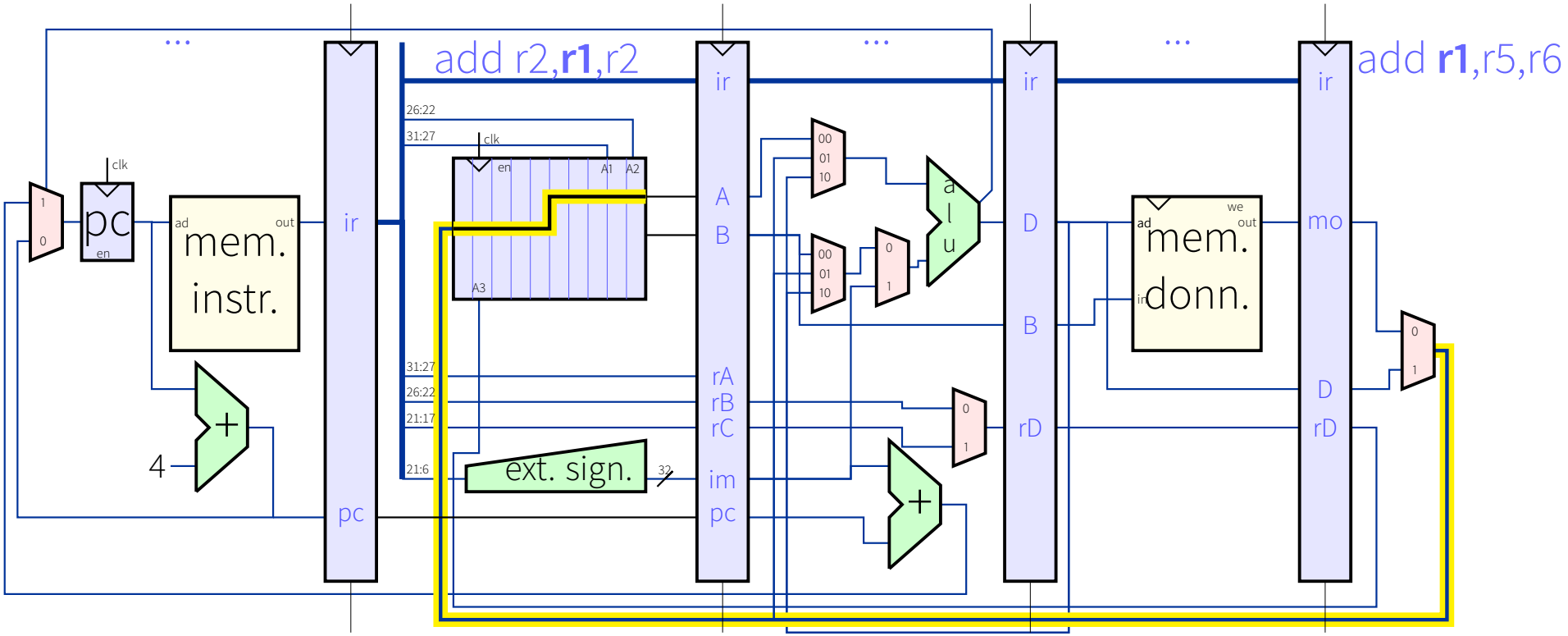
Aléa de données (cont.)

Cas d'une intéraction entre les étages RR et EX.



Aléa de données (cont.)

Cas d'une intéraction entre les étages RR et LI.



Le transfert entrée → sortie pour le banc de registres peut se faire :

- en ajoutant des multiplexeurs aux sorties du banc de registres pour sélectionner entre ce qui est écrit et ce qui est en train d'être écrit
- en écrivant dans le banc de registre sur la phase descendante de l'horloge. On lit ainsi dans tous les cas ce qui vient d'être écrit.

Aléa de données (cont.)

L'envoi est réalisé en comparant les champs `rD` des registres pipeline `EX/MM` et `MM/RR` et les champs `rA` (ou `rB`) du registre pipeline `DI/EX`.

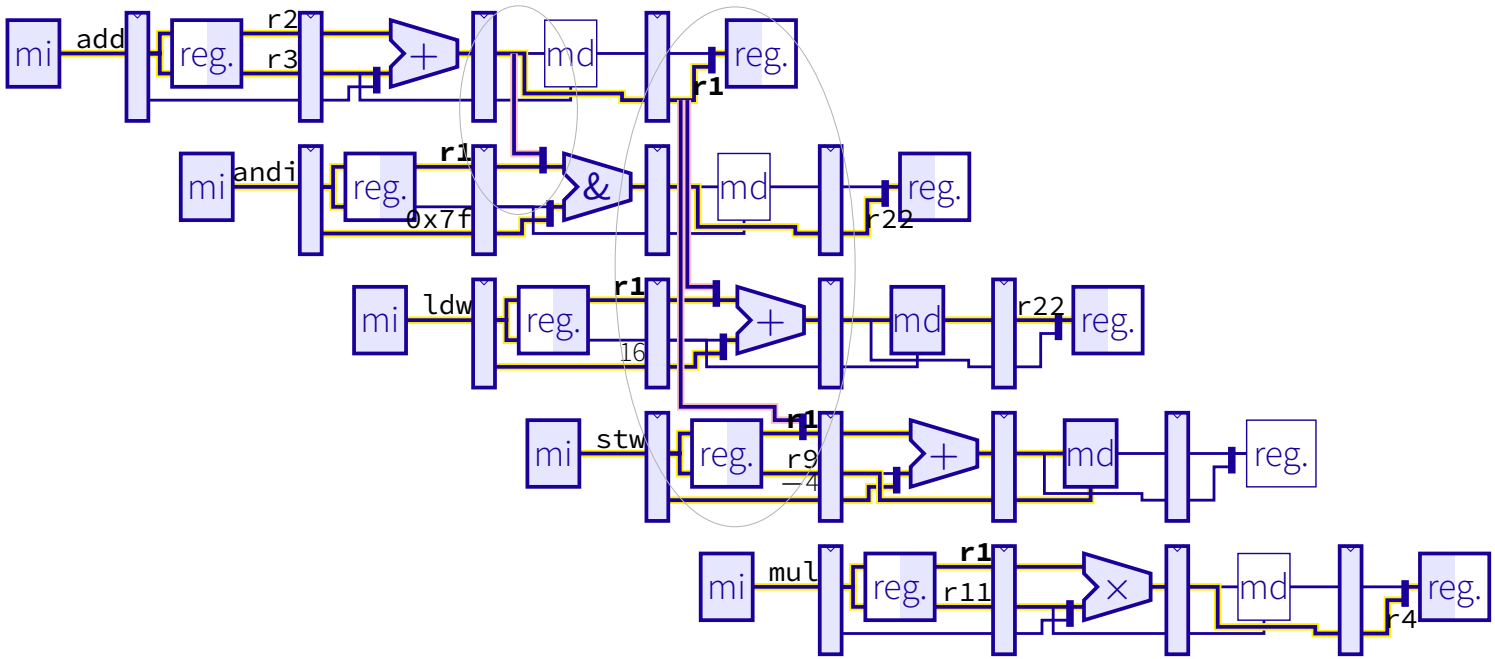
si `EX/MM.rD` == `DI/EX.rA` alors
`muxFwdASel`=10
sinon si `MM/RR.rD` == `DI/EX.rA` alors
`muxFwdASel`=01
sinon
`muxFwdASel`=00

NB1 : `r0` est toujours à 0 et n'est jamais *envoyé*.
NB2 : il faut aussi vérifier que les instructions dans les étages `MM` ou `RR` vont bien réécrire dans le banc de registres (toutes sauf `st` et sauts/branchements).

Aléa de données (cont.)

Avec les mécanismes d'envoi le fonctionnement est correct, sans perte de performances

1 add **r1**,r2,r3
2 andi r22,**r1**,0x7f
3 ldw r22,(16)**r1**
4 stw r9,(-4)**r1**
5 mul r4,**r1**,r11



Mais...

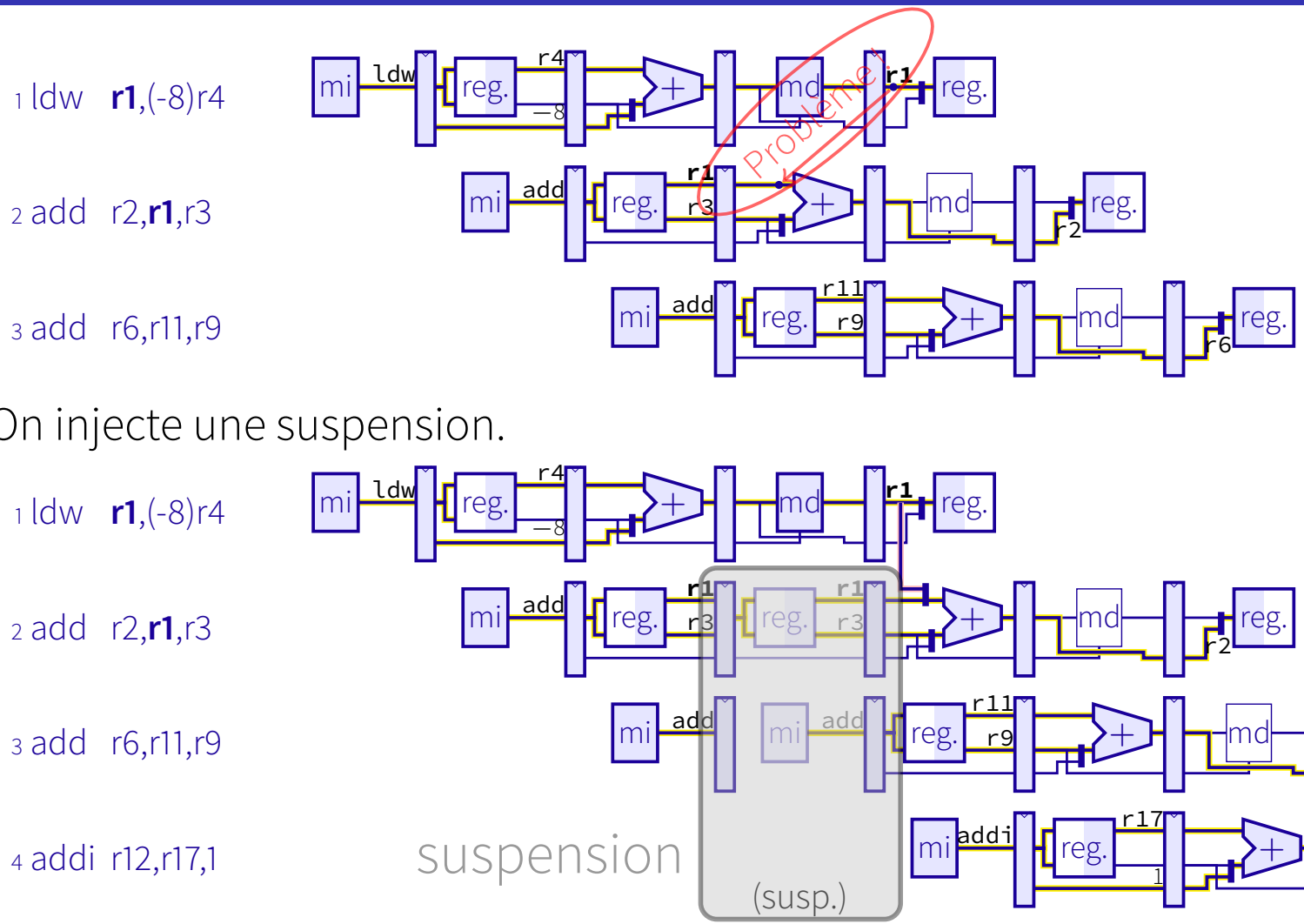
Aléa de données (cont.)

...le mécanisme d'envoi n'est possible que quand la donnée est *dans le processeur*.
Lors d'une lecture mémoire (`ld`), elle n'est *pas* accessible au processeur avant la fin de la phase `MM`.

```
ld     r1,4(r2)
add    r3,r1,r4
```

La solution dans ce cas est de détecter le problème et de faire une **suspension** (*stall*) du pipeline.
L'acquisition et la progression des instructions sont bloquées tant que la donnée n'est pas présente dans le processeur (dans `MM/RR.mo`).
Les mécanismes d'envoi prennent alors le relai.

Aléa de données (cont.)



On injecte une suspension.

Une suspension est mise en oeuvre en interdisant toute écriture dans les registres pipeline ou **pc**, ce qui empêche la progression du pipeline.
Duplication des phases suspendues.

Aléa de données (cont.)

Un `ld` suivi d'une instruction utilisant comme source le registre écrit par le `ld` prendra alors 2 cycles.

En plus des accès mémoire, il existe d'autres situations où la résolution d'un aléa de données nécessite une suspension.

C'est le cas quand les opérations ont une **latence** importante (>1).
Exemple : opérations flottantes

```
fmul r1,r2,r3
fadd r5,r1,r6
```

Si la multiplication flottante a une latence de 4, 3 cycles de suspension seront nécessaires.

Les compilateurs essaient de *réordonnancer* les instructions pour réduire ces suspensions.

Aléas de controle

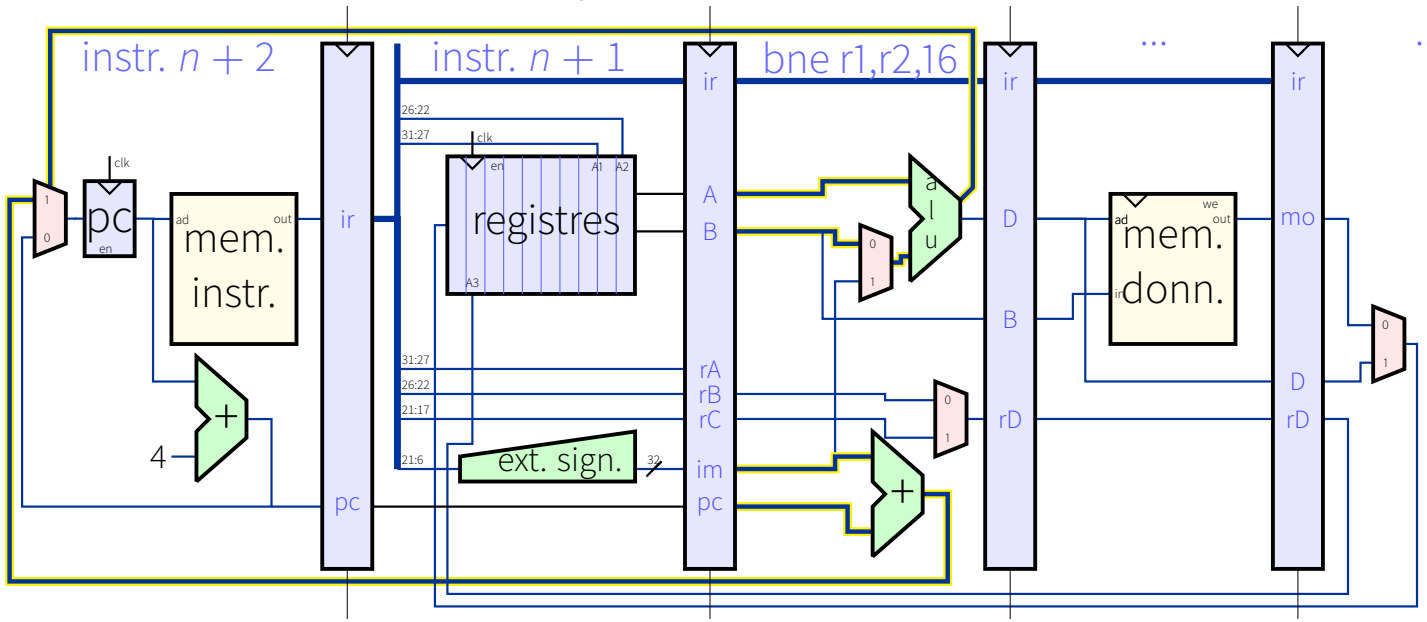
Lors d'une rupture de séquence (branchement, saut), on ne connait le type de l'instruction qu'*après* la phase de décodage d'instruction (`DI`).
Or on a déja acquis l'instruction suivante...

aléa de contrôle

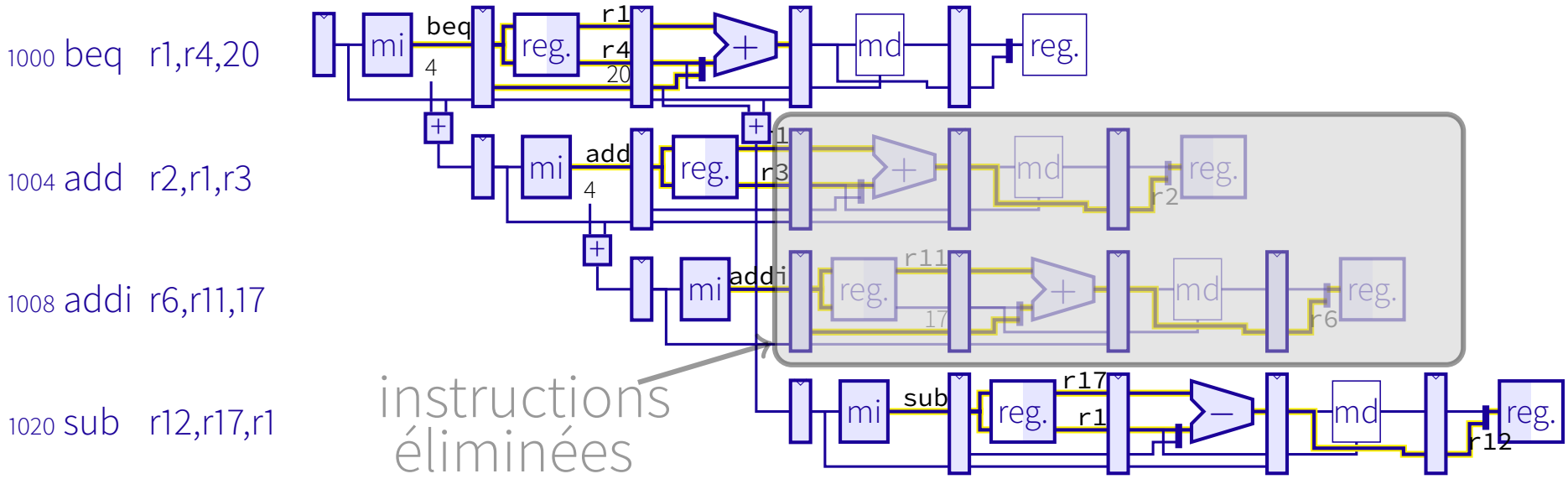
Perte systématique de cycle(s) lors d'un branchement ou d'un saut.
Dans la cas du nios, le branchement se fait en phase `EX`.

Pénalité de branchement de deux cycles.

Instructions $n+1$ et $n+2$ déja acquises au moment du branchement.



Aléas de controle (cont.)



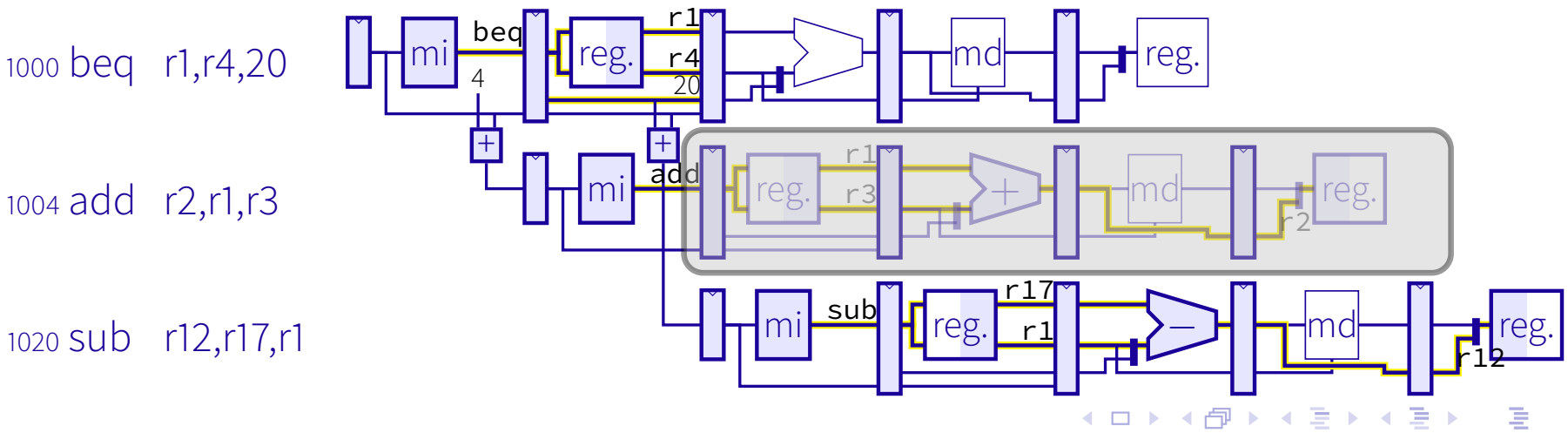
Les instruction aux adresses 1004 et 1008 ont été chargées à tort.
Elles sont éliminées (*flushed*).
En pratique, on met à zéro les registres pipeline pour remplacer les instructions éliminées par des **nop**.
Pénalité de branchement de 2 cycles.

Aléas de controle (cont.)

On peut réduire la pénalité de branchement à un cycle en effectuant le branchement en phase `DI`.
Il faut modifier l'architecture pour :

- calculer `pc+imm16` en phase `DI` (déplacement de l'additionneur de `EX` vers `DI`)
- effectuer les comparaisons en phase `DI` (ajout d'un comparateur aux sorties `A` et `B` du banc de registres)

Mais on ne peut pas supprimer totalement la pénalité par une action faite durant de la phase `DI`.



Aléas de controle (cont.)

Solutions possibles pour réduire la pénalité de branchement :

– Changement de l'action réalisée par un branchement : `brx` (*branch and execute*)

On exécute toujours l'instruction qui suit le `brx`, puis on fait le branchement (*branchement retardé*).

Si on peut exécuter une instruction utile après le branchement, pas de pénalité.

Sinon, on met un **nop** après le branchement

Utilisé dans MIPS et les premiers processeurs pipeline.

Aléas de controle (cont.)

Solutions possibles (suite) :

– Méthodes sophistiquées de *prédiction de branchement* basées sur l'*adresse* de l'instruction lors de la phase `LI`.

Les branchements ont souvent un comportement répétitif (boucles).

Quand on détecte un branchement, on note son adresse et son comportement (pris/non pris).

Quand on détecte une instruction à cette adresse, on cherche à prédire son comportement en fonction de son comportement passé.

Si la prédiction est correcte, pas de pénalité.

Si la prédiction est incorrecte, on efface les instructions acquises et on a la pénalité habituelle.

Aléas de controle (cont.)

Solutions possibles : (suite)
– Instructions conditionnelles
Les instructions conditionnelles présentes dans certaines architectures permettent d'éviter des branchements.

```
if(R1)
  R3=R2;
```

Traduit sans instructions conditionnelles

```
    breq r1,r0,L1
    add r3,r0,r2
L1:  ...
```

Avec instructions conditionnelles

```
movcond r3,r2,r1
```

Très efficace pour de petits branchements.
`cmov` dans pentium, instructions conditionnelles dans ARM

Aléas structurels

Quand deux étages différents d'un pipeline ont besoin d'une même ressource il y a *aléa structurel*.

Idéalement, la plupart sont supprimés par l'architecture.

Mais, certains peuvent arriver.

Exemples :

- si on utilise une mémoire unique pour instructions et données.
  Présence d'un aléa structurel lors de la phase `MM` d'une instruction `ld` ou `st` et de la phase `LI` d'une autre instruction.
- certains opérateurs complexes et rarement utilisés (division flottante, racine carrée) sont *non pipelinés*.
  Ils sont alors indisponibles pendant plusieurs cycles. Aléa structurel potentiel en cas d'autre demandes.
- file d'attente pleine, etc

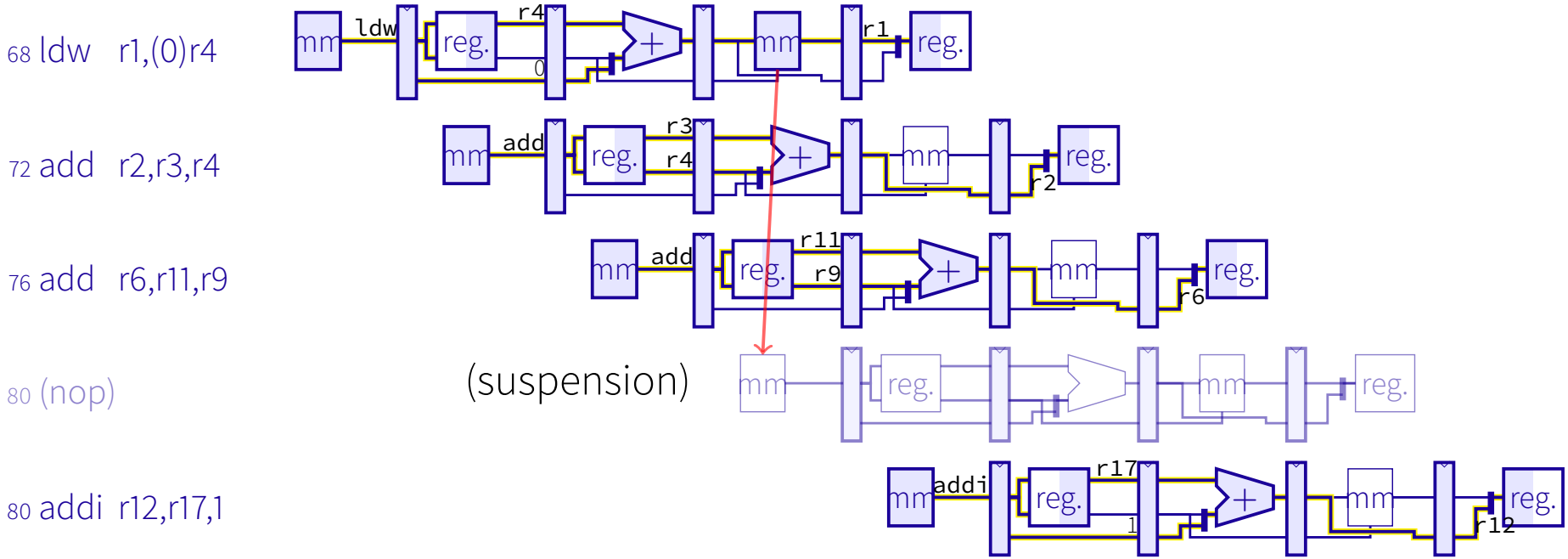
Aléas structurels (cont.)

Solutions :

- duplication de la ressource.
  - mémoire de données et mémoire d'instructions séparées (au moins au niveau des mémoires cache).
  - mise en pipeline d'opérateurs
  - tampons, files d'attentes, etc, plus importantes
- *suspension* (interruption) (*stall*) du pipeline
  La suspension permet d'attendre que la ressource se libère et entraîne la perte d'un ou plusieurs cycles

Aléas structurels (cont.)

Exemple : cas d'une *mémoire unique instructions/données*



Utilisation simultanée de la mémoire par LI et MM.
La phase LI est remplacée par l'injection d'un **nop** sans lecture de la mémoire, ni incrémentation de **pc**.
On reprend ensuite le cours normal d'exécution.
Tout se passe comme si **ld** (ou **st**) prenait 2 cycles.

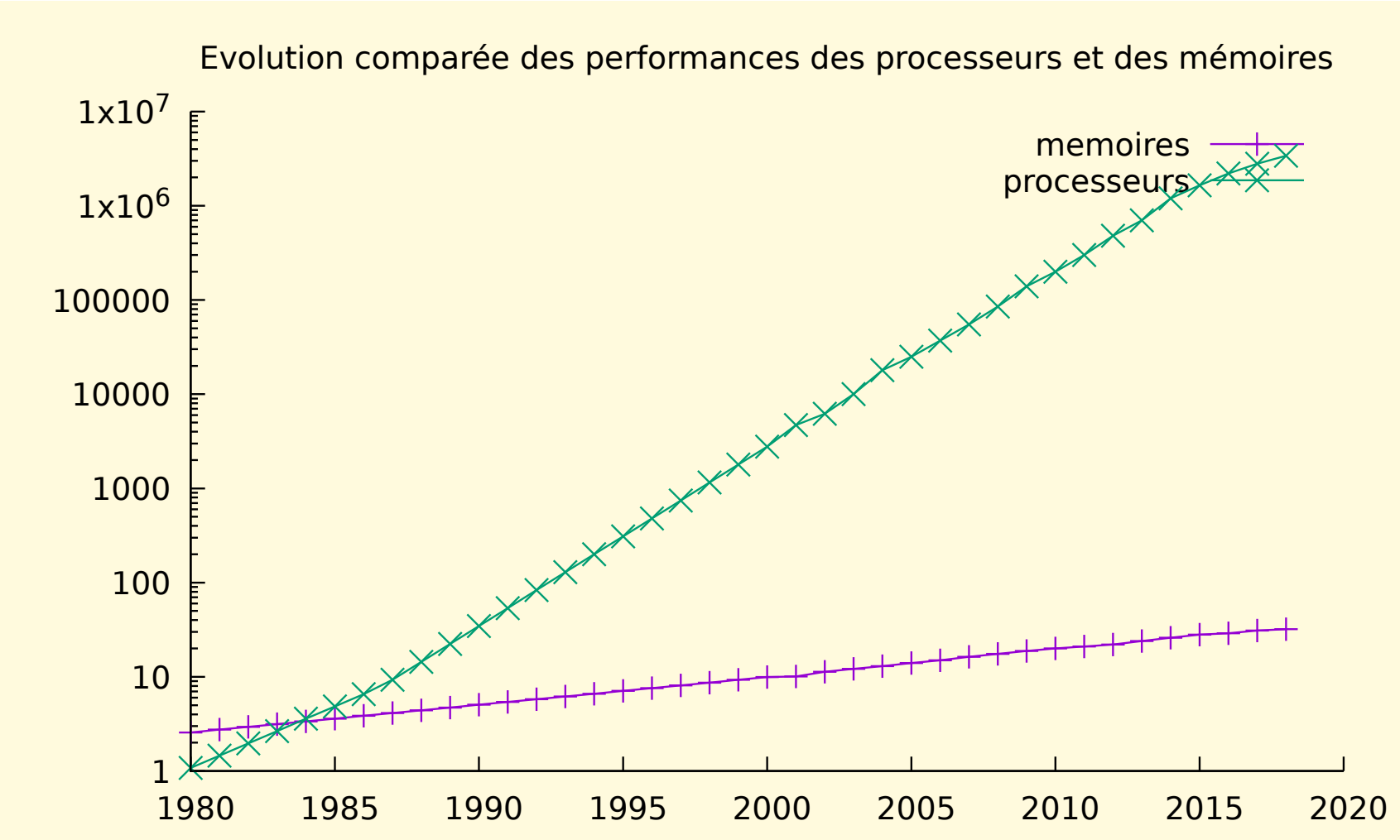
Hiérarchie mémoire

Alain MÉRIGOT

Université Paris Saclay

Position du problème

Evolution technologique comparée des processeurs et des mémoires



Principes de localité

– **Localité temporelle**

Si une information (instruction ou donnée) est manipulée par le processeur, il est très probable qu'elle sera également utilisée peu de temps après

– **Localité spatiale**

Si une information est manipulée par le processeur, il est très probable qu'une information située dans une emplacement mémoire proche sera également utilisée peu de temps après

La hiérarchie mémoire

Différents niveaux hiérarchiques :

– Processeur

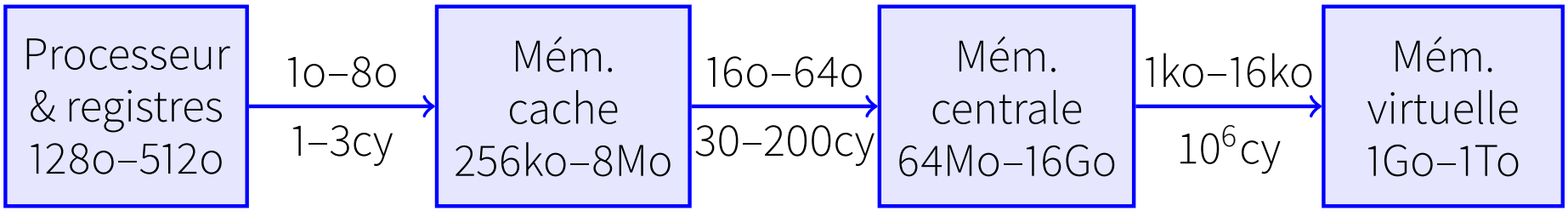
– Mémoire *cache*

– Mémoire centrale

– Mémoire vituelle

Echange entre cache et mémoire centrale : *lignes de cache*

Echange entre mémoire centrale et mémoire virtuelle : *pages*

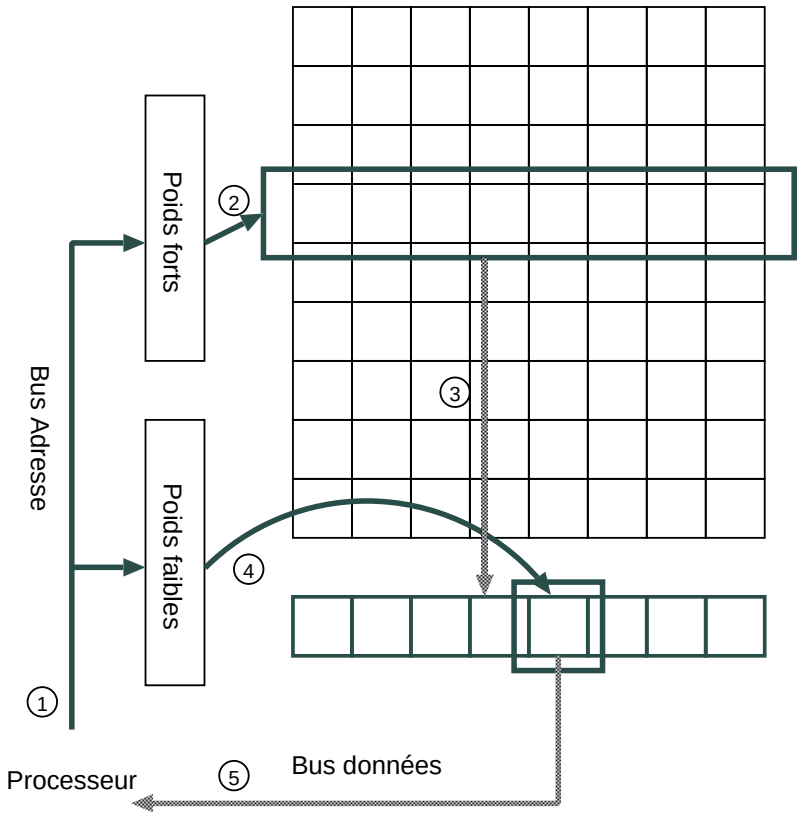


La mémoire centrale

Organisée comme une matrice de points mémoire élémentaires.
La lecture d'un mot mémoire procède en plusieurs étapes:

1. envoi de l'adresse par le processeur
2. sélection d'une ligne avec la partie haute de l'adresse
3. lecture de la ligne
4. sélection du bit (ou mot) dans la ligne avec la partie basse de l'adresse
5. envoi de l'information au processeur

La phase 3 est de très loin la plus coûteuse



La mémoire centrale (cont.)

Améliorations des accès mémoire par accès à plusieurs mots contigus

- Mémoire élargie : utilisation de plusieurs boitiers lus simultanément et envoyés en parallèle sur un bus élargi
- Mémoire entrelacée : utilisation de plusieurs boitiers lus simultanément et envoyés séquentiellement
- Accès en rafale (ou *burst mode*) : boitier mémoire pouvant envoyer séquentiellement n mots en réponse à une demande de lecture. La quasi totalité des boitiers mémoire ont actuellement cette fonctionnalité

Les mémoires actuelles (DDR SDRAM) possèdent également une sorte de *cache* interne mémorisant la ligne récemment lue.

Les mémoires *cache*

Stockent une partie des données de la mémoire centrale

Organisée en *lignes de cache*

Lors d'un accès mémoire :

1. On cherche si la donnée est présente dans le cache
2. Si oui, on envoie la donnée au processeur.
   Temps t_a (temps d'accès réussi).
3. Si non
   1. On lit la ligne contenant la donnée en mémoire et on la copie dans le cache.
   2. on envoie la donnée au processeur au processeur.

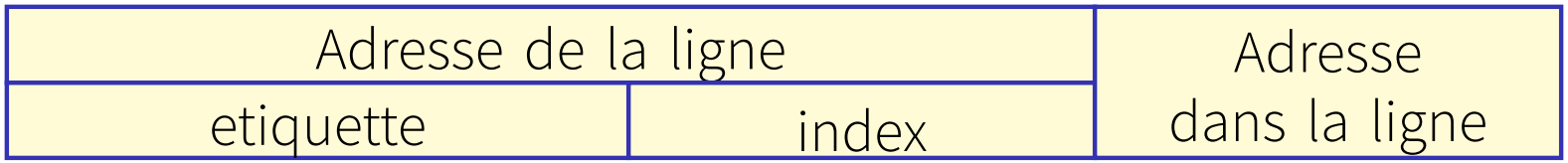
   Temps t_e temps d'échec
   Probabilité d'un échec τ_e.

Ou stocker une ligne dans le cache ?

Plusieurs politiques :

- En un emplacement unique dépendant de son adresse : **cache à correspondance directe**.
  Par exemple on prendra les poids faibles de l'adresse pour déterminer l'emplacement dans le cache
- En un emplacement quelconque : **cache associatif**
  Evite que des lignes avec les mêmes poids faibles d'adresse ne s'*écrasent* mutuellement, mais nécessite une recherche complexe
- En un emplacement quelconque dans un ensemble déterminé par l'adresse : **cache associatif par ensemble**.
  Bon compromis, les ensembles font quelques lignes (4, 8...)

Comment repérer si une ligne est présente dans le cache ?



On distingue différent champs dans une adresse de donnée:

déplacement l'adresse de l'octet dans la ligne

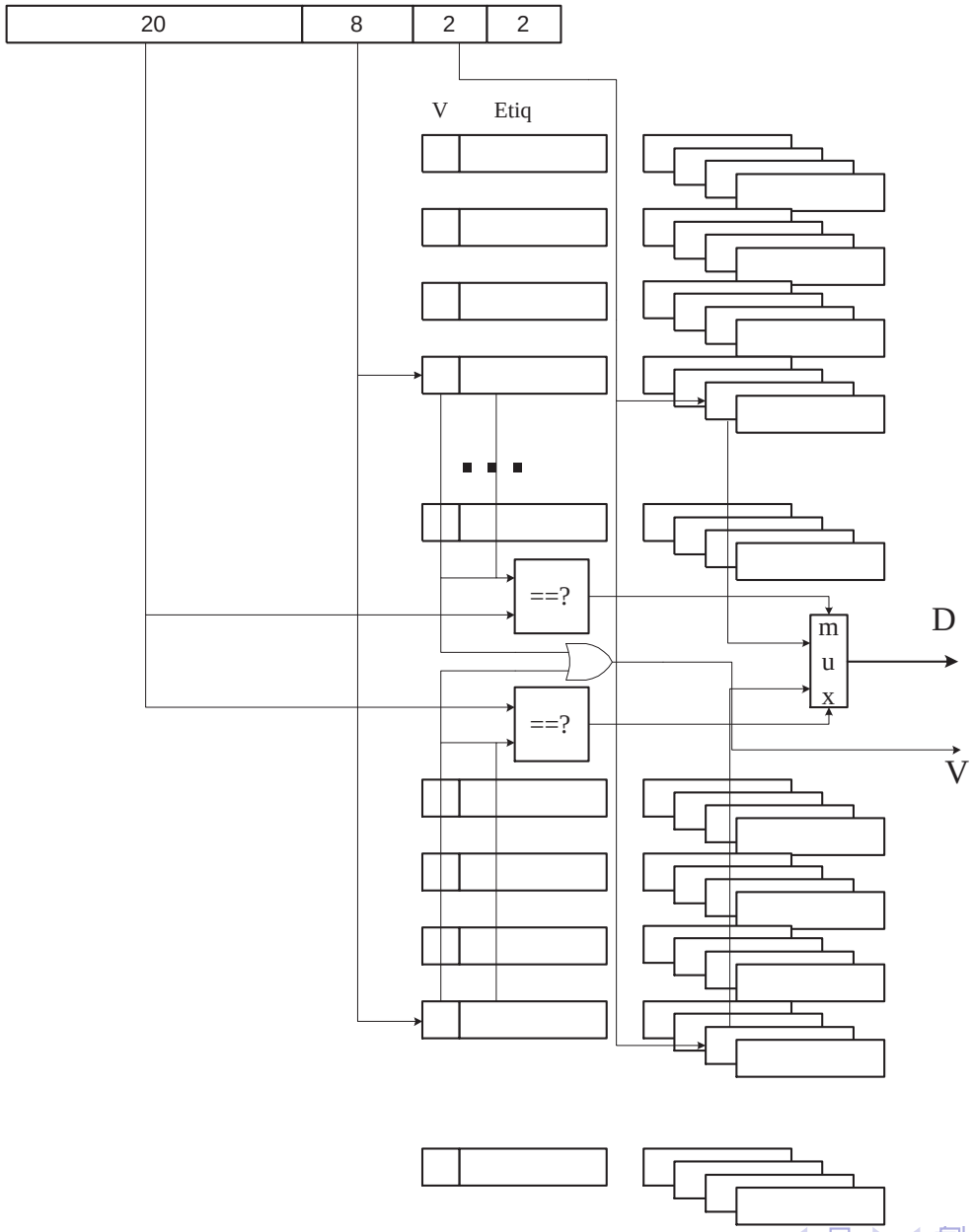
index la partie servant à déterminer la place de la ligne dans le cache (pour les caches à correspondance directe ou associatifs par ensemble)

étiquette Les poids forts qui servent à identifier la ligne de cache

Exemple : cache de 1024 ensembles contenant 4 lignes de 4 mots

- adresse de l'octet dans le mot (2b) + adresse mot dans la ligne (2b) : deplacement 4 bits
- index : 10 bits (repère un ensemble parmi 1024)
- étiquette : $32 - 10 - 4 = 18$ bits

Exemple de cache associatif : 256 ensembles de 2 lignes



Comment remplacer une ligne ?

Si on insère une nouvelle ligne dans le cache, il faut en supprimer une.

Si le cache est (partiellement) associatif, comment choisir la ligne à supprimer ?

Différentes politiques :

- aléatoirement
- la plus ancienne
- la moins récemment utilisée

Cas des écritures

Deux politiques dans le cas d'une écriture :

- **Cache à écriture simultanée (*write-through*)** On écrit simultanément dans le cache et la mémoire pour assurer la cohérence des information.
  Problème : encombrement du bus
- **Cache à réécriture (*write-back*)** On récrit la ligne en mémoire au moment où elle est supprimée du cache.
  Plus complexe, mais charge moins le bus.

Cache instruction et données

On peut avoir ou non les instructions et les données dans la même cache.

- **Caches séparés :** un cache pour les instructions et un pour les données.
  Permet d'éviter les conflits d'accès à la mémoire dans le processeur
- **Caches unifiés :** un cache unique stockant instructions et données
  Permet une meilleure utilisation de la capacité du cache

Amélioration des caches

Idéalement, il faudrait réduire t_a, t_e et τ_e.
Conduit à des exigences la plupart du temps contradictoires.

Réduction du taux d'échec τ_e

- Taux d'échec lors de la première utilisation d'une donnée
  - augmenter la taille des lignes
  - anticiper la demande (*prefetch*)
- Taux d'échec du à une capacité trop faible: augmenter la taille
- Taux d'échec du à des conflits : augmenter l'associativité

Réduction du temps d'accès Utiliser des caches simples (associativité faible, écriture simultanée)

Réduction du temps d'échec

- Caches non bloquant :
  - exécution des requètes dans le désordre
  - priorité aux lectures sur les écritures
- Caches à plusieurs niveaux Deux voire trois niveaux sont courants

Amélioration des caches (cont.)

On utilise *plusieurs niveau de caches* successifs pour permettre de respecter les exigences simultanées (et incompatibles) de réduction du temps d'accès t_a, du temps d'échec t_e et du taux d'échec τ_e.

le cache de niveau 1 (L1) est le plus proche du processeur. Il doit fournir des données en 1 cycle et pour cela être le plus simple possible, quitte à avoir un fort taux d'échec. Il est toujours dans le circuit du processeur.

les caches de niveau supérieur (L2 et souvent L3) doivent limiter les accès mémoire. Il sont de plus grande taille et plus sophistiqués, mais leur temps d'accès est plus important (2–4 cy typ.) Le cache L2 est actuellement intégré au processeur, alors que L3 peut être externe ou intégré.

Caches à plusieurs niveaux

	Cache L1	Cache L2
Objectifs	Cache rapide pour alimenter le processeur (sur le même circuit)	Réduire le temps d'échec du cache L1 et limiter au maximum les accès à la mémoire
Type de cache	deux caches séparés pour données et instructions	unifié instructions/données
Temps accès	1 cy	2-3 cy
Temps d'échec	2-3 cy	30-200 cy
Taille	2×16–32 ko	256 ko (qq Mo pour L3)
Associativité	Corr. directe ou faible associativité (pour $\searrow t_a$)	Associatif par ensemble (2-8 lignes/ens.)
Écriture	Écriture simultanée (pour simplifier le cache)	Réécriture (pour limiter le trafic mémoire)

Evaluation de performances

Il est nécessaire de rendre en compte les caches dans les évaluation de performances.

$$t_{prog} = n_i \times T_c \times (CPI + MPI \times \tau_e \times t_e) \quad (1)$$

CPI : nombre de cycles par instruction en moyenne
MPI : nombres d'accès mémoire par instruction en moyenne
τ_e taux d'échec du cache L1
t_e temps d'échec (en cycles) du cache L1
Dans les cas des caches à plusieurs niveaux, t_e prend en compte les différentes caractéristiques des caches :

$$t_e = t_{eL1} + t_{eL2} \times \tau_{eL2}$$

Mémoire virtuelle

Dans la plupart des systèmes modernes, on peut utiliser une partie de la mémoire de masse (disque) pour étendre la mémoire centrale. **mémoire virtuelle**

En fonction des besoins, les informations sont transférées au disque ou recopiées en mémoire centrale par *pages* (typ. 1-4 ko).

Vus les temps **très** importants d'accès à un disque, on cherche à limiter le taux d'échec (*associativité totale*).

Un programme et ses données ne sont donc pas toujours au même emplacement en mémoire centrale suivant la place disponible.

Par contre, ils gardent toujours les mémes adresses vues du processeur (*adresses virtuelles*), alors que les emplacement en mémoire ont changés (*adresses physiques*).

Mémoire virtuelle (cont.)

Le système d'exploitation crée une correspondance entre les adresses virtuelles et les adresses physiques dans une *table des pages*.

Ces tables peuvent être de grande taille (qq Mo) et résident en mémoire.

Tous les accès mémoire ont besoin de faire une traduction entre les espaces virtuels et physiques avec la table des pages.

Pour accélérer cette traduction, un mécanisme du processeur permet de faire les correspondances entre adresses virtuelles et adresses physiques *avant* l'accès au cache.

Il s'agit d'une petite table qui mémorise les dernières traductions *Translation Lookahead buffer TLB*

Architecture avancée

Alain MÉRIGOT

Université Paris Saclay

Architecture avancée

Les processeurs actuels augmentent leurs performances grace à plusieurs mécanismes :

- le parallélisme d'instructions qui permet d'améliorer l'exécution d'une instruction et/ou de permettre l'exécution d'instructions simultanée
- le parallélisme de données, et notamment les extension dites SIMD qui permettent en une seule instruction de transformer plusieurs données de manière similaire.
- la possibilité de basculer rapidement d'un *thread* à un autre (processeurs *multithreads*).
- l'utilisation de plusieurs processeurs simultanés, avec une mémoire partagée (processeurs *multicores*).

Le parallélisme d'instructions

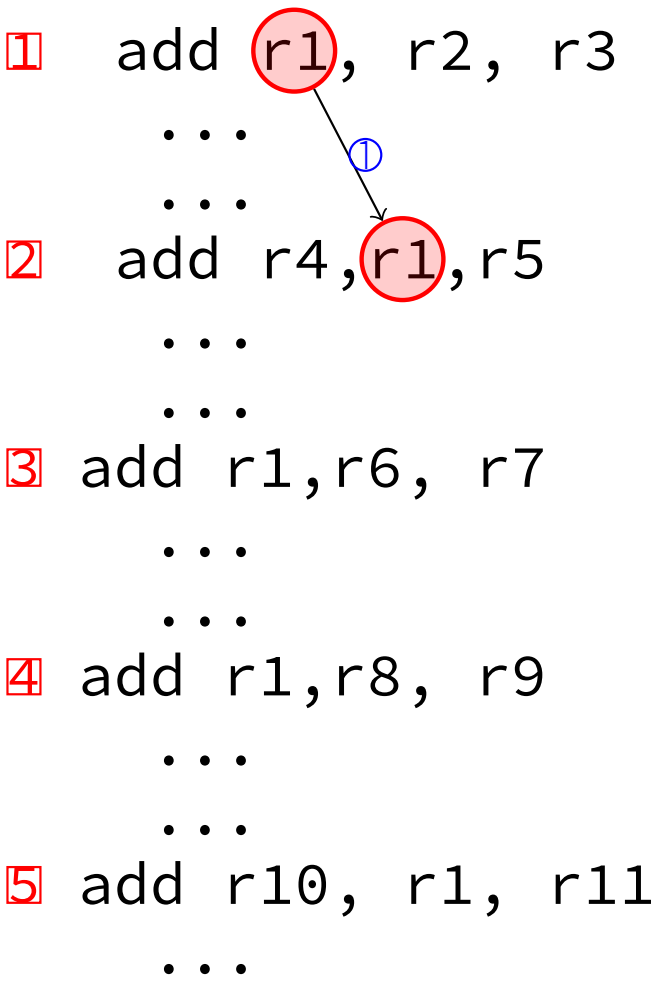
La parallélisation d'instructions repose sur une analyse des *dépendances* entre instructions.
Une dépendance reflète la séquentialité, le fait qu'une instruction doive s'exécuter *avant* une autre pour que le programme soit correct.
Ces mécanismes peuvent ensuite être utilisées :

- par un compilateur qui va modifier l'ordre des instructions pour améliorer l'exécution
- par un processeur qui va modifier *en cours d'exécution* l'ordonnancement des instructions (*exécution dans le désordre*)
- par un processeur qui va exécuter *plusieurs instructions* simultanément. *Processeurs superscalaires*
- par un processeur capable d'éxécuter des instructions en parallèle, la parallélisation étant faite préalablement par un compilateur. Processeurs VLIW

Différents types de dépendances de données

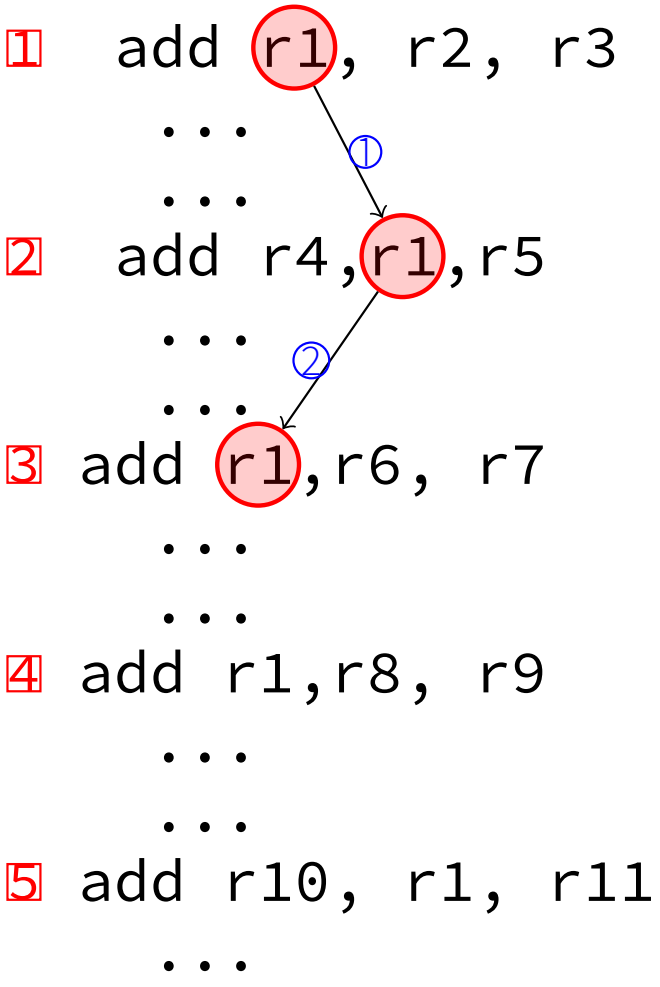
```
1  add r1, r2, r3
     ...
     ...
2  add r4,r1,r5
     ...
     ...
3 add r1,r6, r7
     ...
     ...
4 add r1,r8, r9
     ...
     ...
5 add r10, r1, r11
     ...
```

Différents types de dépendances de données



① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

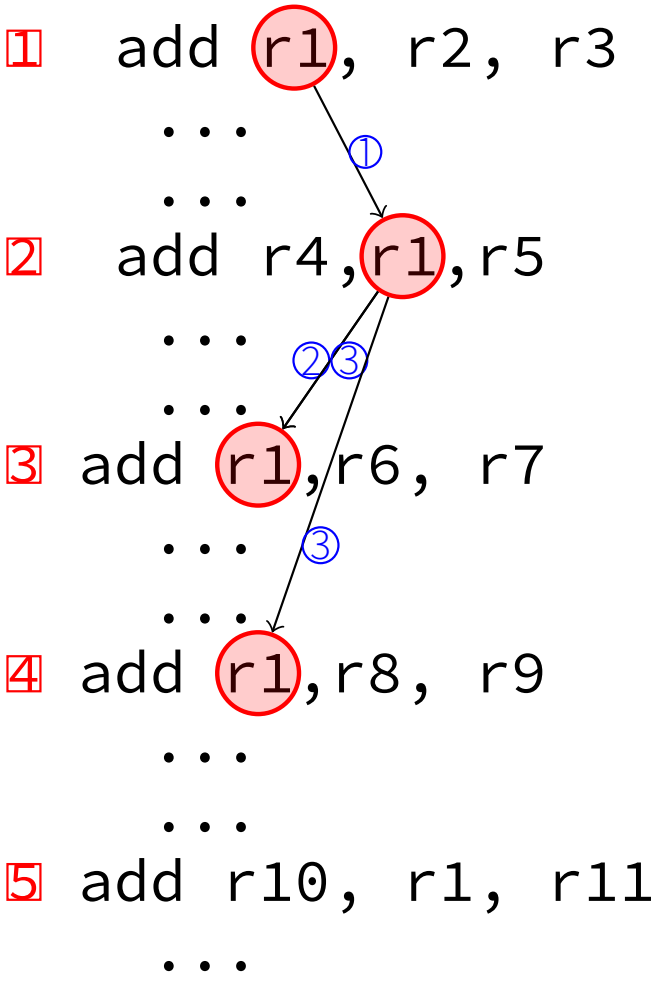
Différents types de dépendances de données



① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

② Le registre `r1` est écrit par l'instruction 3 et 2 doit lire son contenu *avant*. Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse. **antidépendance** ou **dépendance de nom**

Différents types de dépendances de données

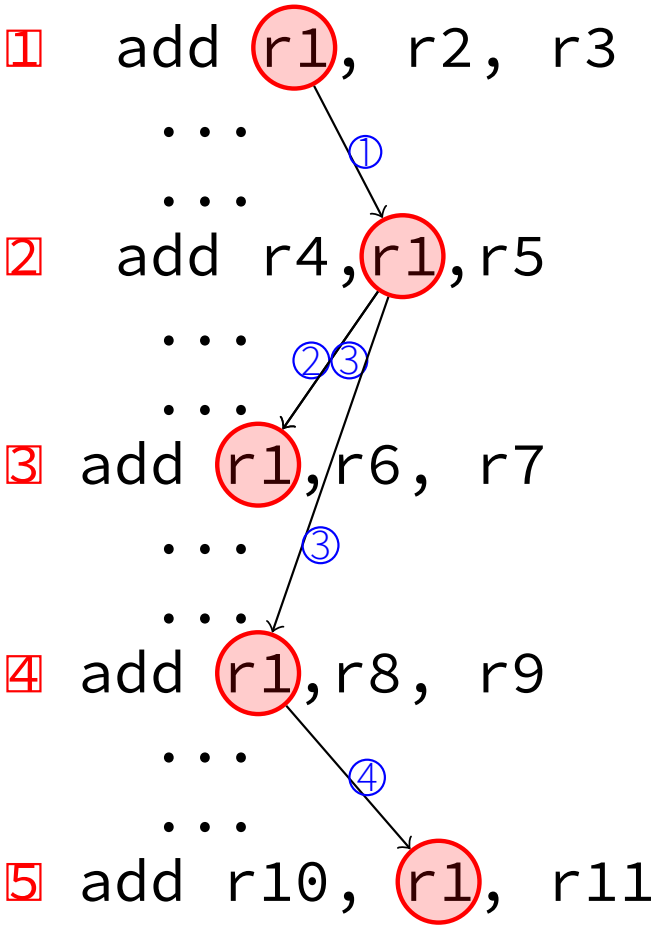


① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

② Le registre `r1` est écrit par l'instruction 3 et 2 doit lire son contenu *avant*. Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse. **antidépendance** ou **dépendance de nom**

③ Les instructions 3 et 4 écrivent successivement le registre `r1`. L'ordre des écritures doit être respecté. **dépendance de sortie**

Différents types de dépendances de données



① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

② Le registre `r1` est écrit par l'instruction 3 et 2 doit lire son contenu *avant*. Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse. **antidépendance** ou **dépendance de nom**

③ Les instructions 3 et 4 écrivent successivement le registre `r1`. L'ordre des écritures doit être respecté. **dépendance de sortie**

④ A nouveau une **dépendance de données** entre 4 et 5.

Réordonnancement d'instructions

On suppose qu'un `ld` suivi de l'utilisation du registre chargé depuis la mémoire provoque une suspension du pipeline d'un cycle (et même beaucoup plus, en cas de défaut de cache et/ou de pipeline profond).

Programme à exécuter

```
ld r1, 0(r2)
add r2, r1, r3
ld r4, 0(r5)
add r6, r4, r7
```

Sans optimisation

```
ld r1, 0(r2)
....susp...
add r2, r1, r3
ld r4, 0(r5)
....susp...
add r6, r4, r7
```

Présence de suspensions du pipeline 6 cy.

Après réordonnancement

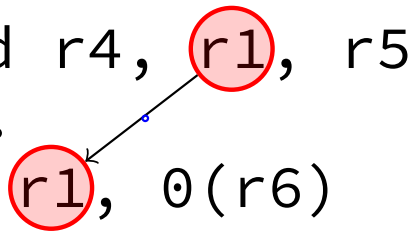
```
ld r1, 0(r2)
ld r4, 0(r5)
add r2, r1, r3
add r6, r4, r7
```

La permutation des instructions 3 et 4 ramène le temps à 4 cycles.

Dépendances de nom

Soit le programme ci-dessous

```
add r1, r2, r3
...
add r4, r1, r5
...
ld r1, 0(r6)
...
add r7, r1, r8
```



Dépendance de nom du fait de l'utilisation du même registre `r1`

Il suffit de *renommer* le (deuxième) registre `r1` en `r9`

```
add r1, r2, r3
...
add r4, r1, r5
...
ld r9, 0(r6)
...
add r7, r9, r8
```

La dépendance de nom a disparu sans que cela change le programme.

Le renommage est fait automatiquement par la plupart des processeurs. Ils disposent en interne de beaucoup plus de registres que dans le jeu d'instruction et à chaque nouvelle écriture dans un registre il est renommé en fonction des registres disponibles.

Prise en compte des aléas de contrôle

Pour limiter les aléas de contrôle, il y a deux mécanismes :

Prédiction de branchement On analyse le comportement des instructions de branchement et quand on retombe sur une instruction de rupture de séquence, on va supposer qu'elle aura le même comportement (même adresse de branchement).

Exécution spéculative On peut exécuter des instructions *postérieures* à un branchement, sans connaitre le résultat du branchement. Les calculs effectués de manière *spéculative* ne seront recopiés dans les registres que quand on connaitra effectivement le comportement de l'instruction (branchement pris ou exécution en séquence).

Exécution d'instructions en parallèle

Processeurs superscalaires Le processeur analyse *dynamiquement* les dépendances et exécute en parallèle 2–6 instructions (1–2 inst entières, 2 flottantes, un accès mémoire, un branchement). La quasi totalité des processeurs généralistes actuels sont superscalaires. En pratique 2–3 instr en //.

Processeurs VLIW (*very long instruction word*) On arrive à augmenter le nombre d'instructions parallèles en faisant une analyse **par le compilateur** qui génère un code avec des instructions déjà parallélisées. 8–16 inst en //. Actuellement surtout pour processeurs de traitement de signal et serveurs.

Processeurs *multithread*

En cas d'attente longue pour une instruction (défaut de cache), on bascule sur un autre fil d'exécution (*thread*).

Permet de masquer les attentes de données en mémoire, si le programme comprend plusieurs *threads*.

Nécessite plusieurs jeux de registres sur le processeur.

Dans certains processeurs superscalaires (Pentium IV et suivants), on peut même exécuter **en parallèle** des instructions appartenant à des *threads* différents.

Parallélisme de données

Dans de nombreux cas, un processeur est amené à effectuer un traitement identique sur un ensemble de données (image, son, etc).
Pour améliorer le temps d'exécution dans cette situation, de nombreux processeurs possèdent des *extensions SIMD* (*single instruction, multiple data*).

- On considère qu'un registre de 64 bits contient en fait 8 données de 8 bits.
- une instruction SIMD va opérer en parallèle sur ces 8 données (reconfiguration de l'ALU)

Il existe différentes configurations possibles des registres 64 bits (8x8, 4x16, 2x32) et les versions actuelles supportent des données flottantes et même un parallélisme très important (AVX512 16x32bits).
Instructions diverses de compactage/décompactage, instructions *masquées*, mouvements de données (permutations, etc).

Multicoeurs

‘

Actuellement plusieurs processeurs par circuits (multicoeurs)

- Progrès des technologies d’intégration
- Limitations des progrès en architecture des processeurs (parallélisme d’instruction)
- Problèmes de consommation

Actuellement

- ordinateurs “grand public” 4 à 8 coeurs
- serveurs 8 à 16
- systèmes embarqués jusqu’à 256 !

Multicoeurs (cont.)

Un ordinateur multicoeur permet d'exécuter

- plusieurs process sur les différents coeurs
- plusieurs *threads* dans un process (parallèlisation d'un même programme sur plusieurs coeurs)

Le parallélisme permet d'améliorer la rapidité d'exécution d'un programme, mais il nécessite une réécriture complète des applications.

Les différents coeurs travaillent sur les mêmes données en mémoire centrale. (modèle à *mémoire partagée*)

Par contre, chacun a sa propre mémoire cache. Nécessité de mécanismes assurant la *cohérence des caches*