

# LU2IN014 : Machine et Représentation

---

Cours 2 : Addition, Soustraction et Représentation des Entiers Relatifs

---

quentin.meunier@lip6.fr

- 1 **Addition et soustraction d'entiers naturels**
- 2 **Représentation des entiers relatifs et arithmétique entière**
- 3 **Représentation des caractères alphanumériques**

## 1 Addition et soustraction d'entiers naturels

- Addition

- Addition et soustraction en décimal
- En base 2
- En base 16
- Addition de mots
- Circuit logique pour l'addition de 2 mots de 1 bit
- Additionneur n bits

- Soustraction

- Multiplication et division

## 2 Représentation des entiers relatifs et arithmétique entière

## 3 Représentation des caractères alphanumériques

# Addition de naturels : en décimal

## Principe et règles

- Addition des chiffres de même rang, en commençant par la droite
- Quand l'addition de deux chiffres dépasse 10, propagation d'une retenue à gauche
- La retenue entrante (sortante du rang précédent) est à sommer avec les chiffres du rang courant

## Exemple

$$\begin{array}{r} 7^1 \quad 5 \\ + \quad 1 \quad 7 \\ \hline 9 \quad 2 \end{array}$$

- Pour les unités : “ $7 + 5 = 12$  : je pose 2 et je retiens 1”
- Pour les dizaines : il faut intégrer la retenue  
“ $7 + 1 + 1 = 9$  : je pose 9 (et je retiens 0)”

# Addition en base 2

## Principe (1)

Addition des bits de même rang

## Règle sur les bits

- $0 + 0 = 0$ ;  $0 + 1 = 1$ ;  $1 + 0 = 1$
- $1 + 1 = 2_d = 10_b$  donc "je pose 0 et je retiens 1"

## Principe (2)

- Lorsqu'il y a une retenue, il faut l'intégrer dans le calcul sur les bits de gauche
- On a donc non pas 2 mais 3 chiffres à sommer

## Règle supplémentaire sur les bits

- Addition de trois 1, et  $1 + 1 + 1 = 3_d = 11_b$ , soit "je pose 1 et je retiens 1"

## Exemples

### Premier exemple

$$\begin{array}{r}
 1\ 0^1\ 1^1\ 0^1\ 1\ 0\ 1\ 1 \\
 + \\
 \hline
 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1
 \end{array}$$

- Les 2 opérandes s'écrivent sur 8 et 6 bits respectivement
- Le résultat s'écrit sur 8 bits

### Deuxième exemple

$$\begin{array}{r}
 1\ 1^1\ 1\ 0\ 0 \\
 + \\
 \hline
 1\ 0\ 0\ 1\ 0
 \end{array}$$

- Les 2 opérandes s'écrivent sur 4 et 3 bits respectivement
- Le résultat s'écrit sur 5 bits : il n'est pas représentable sur 4 bits

# Addition en base 16

## Principe

- Addition réalisée en sommant les chiffres/symboles de même rang
- Propagation d'une retenue lorsque le résultat est strictement supérieur à  $F_h = 15$

## Exemples

$$\begin{array}{r}
 \phantom{+} \phantom{5} \phantom{2} \phantom{9} \phantom{D} \phantom{2} \\
 + \phantom{5} \phantom{2} \phantom{9} \phantom{D} \phantom{2} \\
 \hline
 5 \phantom{F} \phantom{B}
 \end{array}$$

$$\begin{array}{r}
 \phantom{+} \phantom{7} \phantom{3} \phantom{4} \phantom{1^1} \phantom{F} \phantom{A} \\
 + \phantom{7} \phantom{3} \phantom{4} \phantom{1^1} \phantom{F} \phantom{A} \\
 \hline
 9 \phantom{2} \phantom{E}
 \end{array}$$

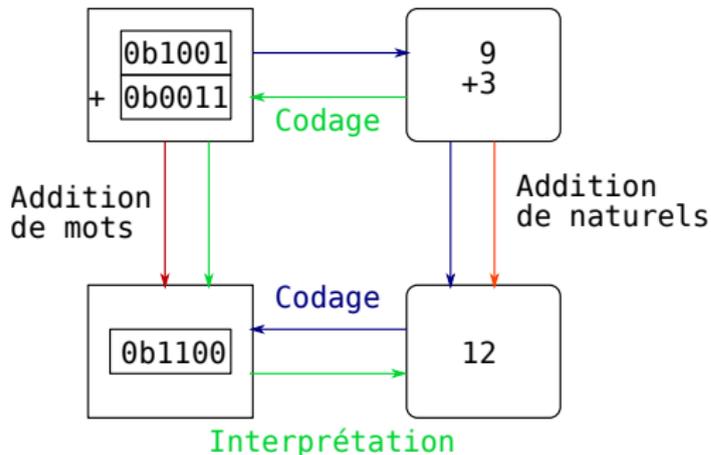
$$\begin{array}{r}
 \phantom{+} \phantom{1} \phantom{5} \phantom{6^1} \phantom{5} \\
 + \phantom{1} \phantom{5} \phantom{6^1} \phantom{5} \\
 \hline
 1 \phantom{1} \phantom{9} \phantom{3}
 \end{array}$$

# Addition de mots

## Principe

- On définit l'addition sur des mots pour que l'interprétation du résultat comme entier naturel corresponde à la somme des interprétations des opérandes
- Cela correspond à l'addition binaire

### Interprétation



- Flèches bleues et rouge : définition de l'addition de mots binaires
- Flèche orange : ce qu'on veut faire
- Flèches vertes : calcul réalisé

# Addition de mots

## Problème : dépassement de capacité

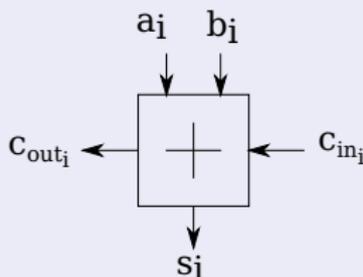
- On dit qu'il y a un **dépassement de capacité** sur entiers naturels quand l'addition de 2 mots ne donnent pas le bon résultat lorsque l'on interprète ceux-ci comme des entiers naturels
- **Exemple** :  $0b1001 + 0b1000$  correspond à  $9 + 8 = 17$  ; or 17 n'est pas représentable sur un mot de 4 bits
- Equivalence avec le fait que le résultat appartienne à l'intervalle représentable : lorsque le résultat théorique de l'opération appartient à l'intervalle de représentation, l'addition de mots est toujours correcte
- **Si dépassement** : la retenue sortante est ignorée
- $\Rightarrow$  Correspond à une arithmétique modulaire sur  $2^n$  pour un mot de  $n$  bits
- **Exemple** :  $0b1001 + 0b1000 = 0b0001 \rightarrow 1 = 17 \% 16$

# Circuit logique pour l'addition de 2 mots de 1 bit

## Additionneur 1 bit

- 3 entrées :  $a_i$ ,  $b_i$  et  $c_{in_i}$  la retenue entrante (sortant de l'addition des bits de droite)
- Deux fonctions à calculer (2 sorties)
  - $s_i$  pour le bit de somme
  - $c_{out_i}$  pour le bit de retenue

## Interface d'un additionneur 1 bit avec 1 bit de retenue entrante



# Table de vérité des fonctions somme et retenue

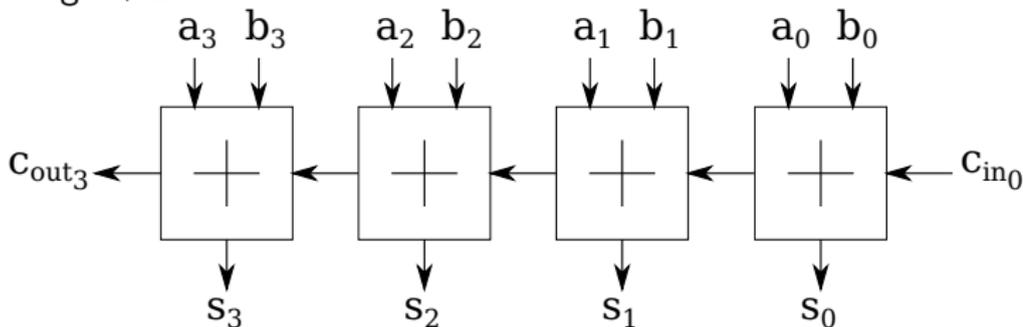
## Addition 1 bit : fonctions somme et retenue

$a_i$	$b_i$	$c_{in_i}$	somme $s_i$	retenue $c_{out_i}$
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Circuit logique pour ces fonctions à partir d'une expression logique pour chacune d'elles : cf. cours 1 pour traduire une fonction en une expression algébrique puis en circuit

## Additionneur $n$ bits

- Un additionneur  $n$  bits est obtenu en connectant  $n$  additionneurs 1 bit de telle sorte que la retenue sortante du rang  $i$  soit la retenue entrante du rang  $i + 1$



- Le circuit final a  $2n$  bits d'entrées ( $n$  pour  $A$  +  $n$  pour  $B$ ) et  $(n + 1)$  sorties ( $n$  bits pour la somme + la retenue sortante du dernier rang)
- Il y a un dépassement de capacité lors de l'addition de deux entiers naturels si  $c_{out_{n-1}} = 1$

## Soustraction en décimal

- On soustrait un chiffre au chiffre de même rang, en commençant par la droite :

$$\begin{array}{r} 7 \quad 5 \\ - \quad 1^1 \quad 7 \\ \hline 5 \quad 8 \end{array}$$

- Pour les unités : “7 est plus grand que 5, je pique une dizaine à gauche”
- Il y a une retenue à propager et à soustraire à gauche/rang supérieur
- Pour les dizaines : il faut soustraire la dizaine piquée “7 - 1 - 1”
- Deux valeurs à soustraire et non pas une seule avec la retenue entrante
- Remarque** : la soustraction  $A - B$  avec  $A$  et  $B$  entiers naturels n'est pas définie si  $B > A$ , i.e. si le nombre à soustraire est supérieur au nombre dont on le soustrait

# Soustraction en binaire

## Principe

- Même principe que la soustraction décimale

## Règles sur les bits

- $0 - 0 = 0$  ;  $1 - 0 = 1$  ;  $1 - 1 = 0$
- $0 - 1 =$  impossible : on pique une “deuzaine” à gauche, une retenue, qu’il faut soustraire ensuite à gauche  
 $0 - 1 \Rightarrow 2 - 1 = 1$  avec 1 retenue à retrancher à gauche

## soustraction binaire

- Lors de la soustraction de  $b_i$  à  $a_i$  dans le calcul de  $A - B$ , il y a 2 valeurs à soustraire : la retenue sortante du rang  $i - 1$  et  $b_i$
- La soustraction de deux bits produit donc aussi une sortie et une retenue

# Exemples de soustraction sur entiers naturels

## Sans dépassement

$$\begin{array}{r}
 \phantom{-} \phantom{0^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{0} \phantom{1} \phantom{1} \\
 - \phantom{0^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{0} \phantom{1} \phantom{1} \\
 \hline
 0 \phantom{1} \phantom{0} \phantom{1} \phantom{0} \phantom{1} \phantom{0} \phantom{1} \phantom{1}
 \end{array}$$

## Avec dépassement

$$\begin{array}{r}
 \phantom{-} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{0} \phantom{0} \\
 - \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{1^1} \phantom{0} \phantom{0} \\
 \hline
 1 \phantom{1} \phantom{0} \phantom{0} \phantom{0}
 \end{array}$$

- Il y a un dépassement si le nombre à soustraire est plus grand que le nombre duquel on le soustrait

## Multiplication et division

- Opérations de multiplication et division de 2 entiers complexes  $\Rightarrow$  circuits les réalisant pas dans l'ALU

### Multiplication et division par une puissance de 2 en binaire

- Multiplier ou diviser par  $2^n \equiv$  décalage à gauche ou à droite de  $n$  bits
  - $N_d = (a_{p-1} \dots a_1 a_0)_b \Rightarrow 2N_d = (a_{p-1} \dots a_1 a_0 0)_b$
  - $N_d = (a_{p-1} \dots a_1 a_0)_b \Rightarrow (N/2)_d = (0a_p a_{p-1} \dots a_1)_d$

### Multiplication et division par une puissance de B en base B

- Décaler à gauche de 1 (respectivement  $n$ ) revient à multiplier par la base B (respectivement par  $B^n$ )
- Décaler à droite de 1 (respectivement  $n$ ) à diviser par la base B (respectivement par  $B^n$ ).

# Décalage et multiplication/division

## Exemple en base 10

- $1024/10 = 102$  : décalage à droite de 1 (division par  $10^1$ )
- $1024/100 = 10$  : décalage à droite de 2 (division par  $10^2$ )
- $1024 \times 10 = 10240$  : décalage à gauche de 1

## Exemple en base 2

- $1100_b \ggg 1_d = 110_b = 6_d = 12_d/2_d$
- $110_b \ggg 2_d = 1_b = 1_d = 6_d/4_d$
- $11_b \lll 1_d = 110_b = 6_d = 3_d \times 2_d$
- $11_b \lll 2_d = 1100_b = 12_d = 3_d \times 4_d$

## 1 Addition et soustraction d'entiers naturels

## 2 Représentation des entiers relatifs et arithmétique entière

- Représentation des entiers relatifs
  - Représentation usuelle
  - Représentation en complément à 1
  - Représentation en complément à 2
- Codage entiers relatifs en complément à 2
  - Détermination de l'opposé d'un nombre
  - Extension du format d'un nombre
- Addition et soustraction d'entiers relatifs
  - Principe
  - Dépassement de capacité
  - Détermination d'un dépassement de capacité lors d'une addition
  - Détection de dépassement de capacité
  - Instruction d'addition et de soustraction MIPS

## 3 Représentation des caractères alphanumériques

# Représentation usuelle des entiers relatifs

## Définition

- C'est la représentation courante qui utilise un signe et une valeur absolue :  $+3$ ,  $-3$ ,  $+2$ ,  $-2$ , ...
- Un bit code le signe, les autres bits la valeur absolue

## Addition/soustraction

- Complexe
- Il faut déterminer le signe pour savoir s'il faut faire une addition ou une soustraction

## Autre problème

- Il y a deux représentations pour la valeur 0 : rend la comparaison de deux mots compliquée
- $\Rightarrow$  il faut interdire une des deux valeurs ( $-0$ )

# Représentation en complément à 1

## Complément à 1

- En complément à 1 si  $N_b = n_d$  alors  $-n_d = \overline{N_b}$ .

## Conséquences

- L'opposé en binaire est simplement le complément
- Il y a deux zéros (que des 1 et que des 0) à gérer
- Les opérations d'addition et de soustraction sont encore plus complexes

# Représentation en complément à 2

## En complément à 2

- Un seul zéro
- Les traitements à appliquer pour réaliser une addition sont identiques que les nombres soient positifs ou négatifs

## Utilisation

- C'est la représentation adoptée dans tous les processeurs pour représenter les entiers relatifs
- C'est la seule "bonne" représentation : désormais, on n'utilisera plus que celle-là
- Appelée dans la suite représentation ou codage **entiers relatifs en complément à 2** ou simplement **entiers relatifs**

# Représentation entiers relatifs

## Codage

- En complément à deux, tout nombre est représenté par deux parties :
  - un terme négatif constant ou nul
  - une correction positive

## Interprétation de $a_{n-1} \dots a_1 a_0 b$

$$a_{n-1} \dots a_1 a_0 b = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

$$a_{n-1} \dots a_1 a_0 b = -a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \dots + a_1 2 + a_0$$

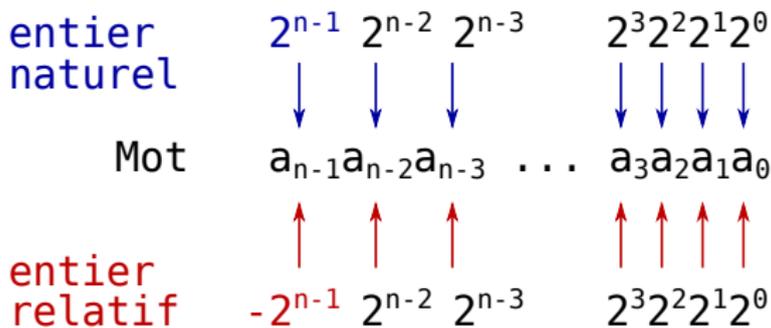
## Codage entiers relatifs

$$(a_{n-1} \dots a_1 a_0)_b = (-a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \dots + a_1 2 + a_0)_d$$

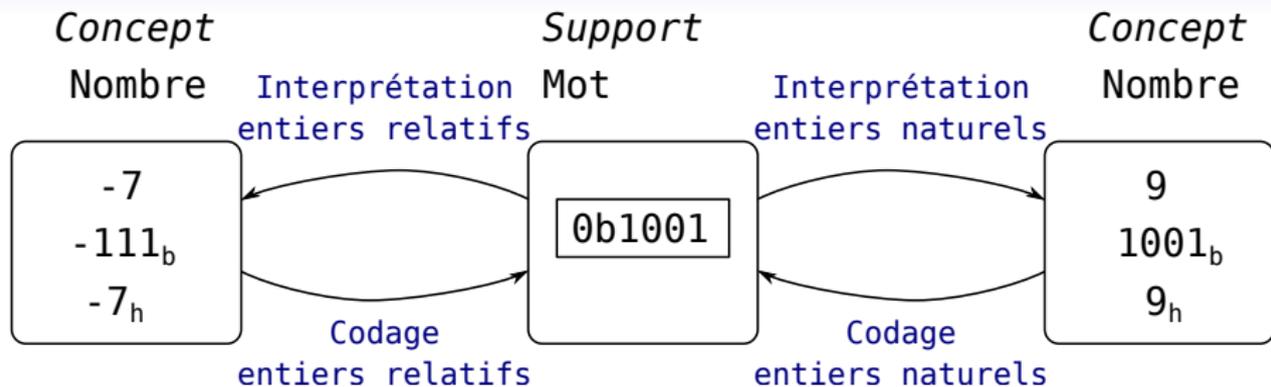
- $-a_{n-1} 2^{n-1}$  valeur négative constante ou nulle  $\Rightarrow a_{n-1}$  appelé **bit de signe** : sa valeur indique le signe de l'entier

- $\sum_{i=0}^{n-2} a_i 2^i$  correspond à la correction positive

- Différence des interprétations entiers naturels et relatifs



# Codage entiers relatifs



## Exemple

- Sur 4 bits, le mot  $0b1001$  s'interprète comme la valeur  $9$  selon le codage **entiers naturels**, et  $9 = 1001_b$
- Sur 4 bits, le mot  $0b1001$  s'interprète comme la valeur  $-7$  selon le codage **entiers relatifs**

# Exemples

## Exemple sur 3 bits

Soit un mot de 3 bits  $a_2a_1a_0$ , son **interprétation en complément à 2** est :

# Exemples

## Exemple sur 3 bits

Soit un mot de 3 bits  $a_2a_1a_0$ , son **interprétation en complément à 2** est :  $-a_2 \times 2^2 + a_1 \times 2 + a_0$

- $N = 0b000 =$

# Exemples

## Exemple sur 3 bits

Soit un mot de 3 bits  $a_2a_1a_0$ , son **interprétation en complément à 2** est :  $-a_2 \times 2^2 + a_1 \times 2 + a_0$

- $N = 0b000 = 0_d$
- $N = 0b101 =$

# Exemples

## Exemple sur 3 bits

Soit un mot de 3 bits  $a_2a_1a_0$ , son **interprétation en complément à 2** est :  $-a_2 \times 2^2 + a_1 \times 2 + a_0$

- $N = 0b000 = 0_d$
- $N = 0b101 = -2^2 + 1 = -3$
- $N = 0b110 =$

# Exemples

## Exemple sur 3 bits

Soit un mot de 3 bits  $a_2a_1a_0$ , son **interprétation en complément à 2** est :  $-a_2 \times 2^2 + a_1 \times 2 + a_0$

- $N = 0b000 = 0_d$
- $N = 0b101 = -2^2 + 1 = -3$
- $N = 0b110 = -2^2 + 2 = -2$
- $N = 0b111 =$

# Exemples

## Exemple sur 3 bits

Soit un mot de 3 bits  $a_2a_1a_0$ , son **interprétation en complément à 2** est :  $-a_2 \times 2^2 + a_1 \times 2 + a_0$

- $N = 0b000 = 0_d$
- $N = 0b101 = -2^2 + 1 = -3$
- $N = 0b110 = -2^2 + 2 = -2$
- $N = 0b111 = -2^2 + 2 + 1 = -1$
- $N = 0b011 =$

# Exemples

## Exemple sur 3 bits

Soit un mot de 3 bits  $a_2a_1a_0$ , son **interprétation en complément à 2** est :  $-a_2 \times 2^2 + a_1 \times 2 + a_0$

- $N = 0b000 = 0_d$
- $N = 0b101 = -2^2 + 1 = -3$
- $N = 0b110 = -2^2 + 2 = -2$
- $N = 0b111 = -2^2 + 2 + 1 = -1$
- $N = 0b011 = 2 + 1 = 3$
- Repérer
  - Entier positif  $\Leftrightarrow$  le bit de signe  $a_2$  vaut 0
  - Entier négatif  $\Leftrightarrow$  le bit de signe  $a_2$  vaut 1

# Intervalle de représentation

## Intervalle de représentation sur $n$ bits

- Sur  $n$  bits, en complément à 2, l'intervalle de représentation est :

$$[-2^{n-1}, 2^{n-1} - 1]$$

- Si  $a_{n-1} = 0 \Rightarrow$  entier naturel sur  $n - 1$  bits  
 $\Rightarrow$  dans l'intervalle  $[0, 2^{n-1} - 1]$  (cf. cours 1)
- Si  $a_{n-1} = 1$  :  $N = -2^{n-1} + corr$  avec  $corr$  la correction et  $corr \in [0, 2^{n-1} - 1]$ 
  - si  $corr = 0 \Rightarrow$  c'est le plus petit entier négatif possible, il vaut  $-2^{n-1}$
  - si  $corr = 2^{n-1} - 1$  valeur entière maximale  $\Rightarrow$  c'est le plus grand entier négatif possible, il vaut  $-1$
- Ainsi, l'ensemble des entiers relatifs représentés est en complément à 2 sur  $n$  bits est bien  $[-2^{n-1}, 2^{n-1} - 1]$

# Détermination de l'opposé d'un nombre

## Opposé d'un entier en complément à 2

- Soit un mot  $M = m_{n-1} \dots m_1 m_0$  encodant la valeur  $v$  selon le codage **entiers relatifs**
- La valeur  $-v$  s'obtient alors par le calcul de  $M' = \overline{M} + 1 = \overline{m_{n-1} \dots m_1 m_0} + 1$
- $M'$  s'appelle le **complément à 2** de  $M$  (noté  $\text{Cà2}(M)$ )
- Il ne faut pas confondre le calcul du complément à 2 (opération sur un mot) avec la représentation en complément à 2 (codage)

## Remarques

- L'intervalle des nombres représentables en complément à 2 sur  $n$  bits n'est pas symétrique : on ne peut pas coder l'opposé de  $-2^{n-1}$  sur  $n$  bits (si on calcule le complément à 2 du mot, c'est l'identité)
- Preuve : au tableau

# Extension de la représentation d'un entier relatif

## Extension de $p$ à $n$ bits

- Soit un mot de  $p$  bits contenant une valeur  $v$ , interprété selon le codage entier relatif par le nombre  $d$
- Pour que le mot  $v'$  de  $n$  bits ( $n > p$ ) encode également le nombre  $d$ , il faut :
  - Copier les bits de  $v$  dans les bits de poids faible de  $v'$
  - Mettre les bits de poids fort restant de  $v'$  à la valeur du bit de poids fort de  $v$
- Si  $N = a_{p-1}a_{p-2}\dots a_1a_0$  sur  $p$  bits, alors sur  $n$  bits  
 $N = a_{p-1}\dots(n-p \text{ fois})\dots a_{p-1}a_{p-1}a_{p-2}\dots a_1a_0$
- On appelle cette opération l'**extension signée**

## Preuve

- Au tableau

## Exemples d'extensions signées

### Extension de 4 à 8 bits

- $N_1 = 0b1001 =$

## Exemples d'extensions signées

### Extension de 4 à 8 bits

- $N_1 = 0b1001 = 0b11111001$
- $N_2 = 0b0110 =$

## Exemples d'extensions signées

### Extension de 4 à 8 bits

- $N_1 = 0b1001 = 0b11111001$
- $N_2 = 0b0110 = 0b00000110$

### Extension de 16 à 32 bits en hexadécimal

- $N_3 = 0x1110 =$

## Exemples d'extensions signées

### Extension de 4 à 8 bits

- $N_1 = 0b1001 = 0b11111001$
- $N_2 = 0b0110 = 0b00000110$

### Extension de 16 à 32 bits en hexadécimal

- $N_3 = 0x1110 = 0x00001110$
- $N_4 = 0x90B2 =$

## Exemples d'extensions signées

### Extension de 4 à 8 bits

- $N_1 = 0b1001 = 0b11111001$
- $N_2 = 0b0110 = 0b00000110$

### Extension de 16 à 32 bits en hexadécimal

- $N_3 = 0x1110 = 0x00001110$
- $N_4 = 0x90B2 = 0xFFFF90B2$

# Addition et soustraction d'entiers relatifs

## Propriété

- L'addition sur les mots peut être utilisée pour sommer des entiers relatifs
- Seule la condition de débordement change
- Propriété du codage des relatifs en complément à 2

# Addition et soustraction d'entiers relatifs

## Soustraction

- On pourrait utiliser un circuit soustracteur, mais en réalité pas besoin
- On tire partie du fait que  $A - B = A + (-B)$ , avec  $-B = \text{C}à2(B)$
- $\Rightarrow$  On calcule  $-B$ , et on réutilise l'additionneur comme pour l'addition des entiers relatifs

# Exemples d'addition et soustraction

## Exemples

- Au tableau...

# Dépassement de capacité

## Résultat représentable

- Le résultat d'une addition/soustraction de 2 mots de  $n$  bits est dit **représentable** sur  $n$  bits si l'interprétation des bits  $n - 1$  à  $0$  dans la même représentation représente bien le résultat théorique de l'opération.
- Si ce n'est pas le cas : **dépassement de capacité**
- **Remarque** : le fait qu'il y ait un dépassement ou non dépend de l'interprétation des mots

## Addition de $A$ et $B$ et dépassement de capacité en entiers relatifs

- **Règle** : Lors d'une addition de deux mots, il y a un dépassement de capacité en entiers relatifs si et seulement si  $cout_{n-1} \neq cout_{n-2}$
- 3 cas en fonction du signe de  $A$  et  $B$ 
  - $A$  et  $B$  de signes différents
  - $A \geq 0$  et  $B \geq 0$
  - $A < 0$  et  $B < 0$

# Détermination d'un dépassement de capacité lors d'une addition

## A et B de signe différent, A négatif

- Au tableau
- **Conclusion** : pas de dépassement possible et  $cout_{n-1} = cout_{n-2}$

# Détermination d'un dépassement de capacité lors d'une addition

## A et B de même signe et négatifs

- Au tableau
- **Conclusion** : dépassement ssi  $cout_{n-1} \neq cout_{n-2}$

# Détermination d'un dépassement de capacité lors d'une addition

## A et B de même signe et positifs

- Au tableau
- **Conclusion** : dépassement ssi  $cout_{n-1} \neq cout_{n-2}$

# Détermination d'un dépassement de capacité lors d'une addition

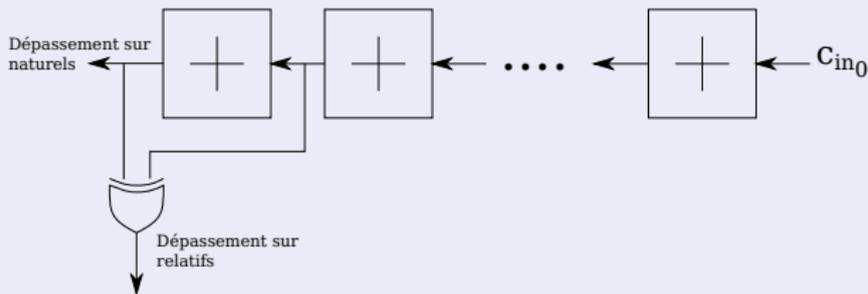
## Synthèse de l'analyse des différents cas

- Quelques soient les signes de A et B, il y a donc bien un dépassement de capacité sur entiers relatifs ssi  $c_{out_{n-1}} \neq c_{out_{n-2}}$ , soit  $c_{out_{n-1}} \oplus c_{out_{n-2}} = 1$ .

## Rappel

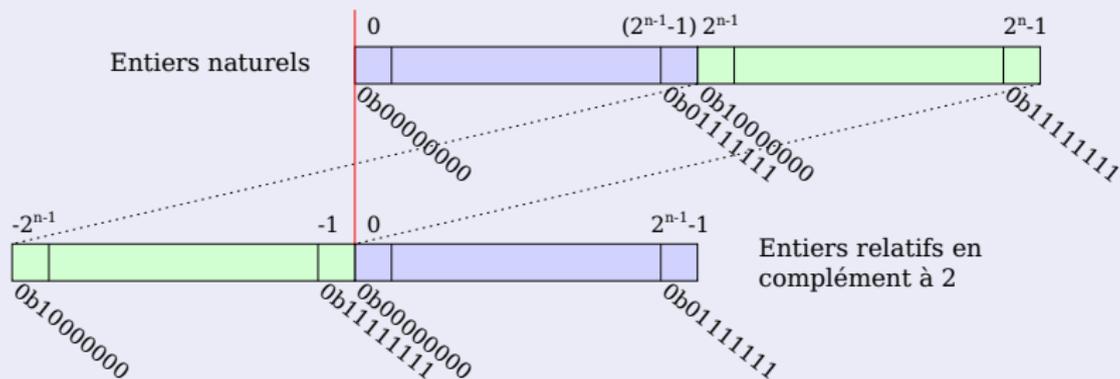
- La détection d'un dépassement de capacité lors de l'addition sur entiers relatifs est différente de celle lors de l'addition d'entiers naturels

# Détection d'un dépassement de capacité dans l'ALU



# Pourquoi ça marche aussi bien ?

## Visualisation des intervalles des 2 représentations



# Soustractions d'entiers non signés

## Soustraction d'entiers non signés

- A priori, pour faire une soustraction d'entiers non signés sur  $n$  bits, si on ne veut pas rajouter un soustracteur, il faut un additionneur  $n + 1$  bits :
  - Sur  $n + 1$  bits en complément à 2, on peut représenter les valeurs de  $-2^n$  à  $2^n - 1$
  - $\Rightarrow$  inclut l'intervalle  $[0; 2^n - 1]$  des entiers naturels sur  $n$  bits (avec le bit de signe = 0)
  - On peut ainsi calculer l'opposé des nombres et faire l'addition  $A + (-B) = A - B$
- ...ou pas : exemple avec  $15 - 10$

## Conclusion

- Pour faire une soustraction d'entiers non signés sur  $n$  bits, on peut utiliser le même additionneur/soustracteur  $n$  bits que pour les signés en faisant le calcul  $A - B = A + C\text{à}2(B)$  (la condition de débordement n'est pas la même que pour les relatifs)

# Instructions d'addition et soustraction MIPS

## Instructions d'addition et de soustraction MIPS

- *add*  $r_d, r_s, r_t$  ou *addu*  $r_d, r_s, r_t$  avec  $r_i$  des registres et réalisant  $r_d \leftarrow r_s + r_t$
- *addi*  $r_d, r_s, imm$  ou *addiu*  $r_d, r_s, imm$  avec  $r_i$  des registres, *imm* un immédiat sur 16 bits et réalisant  $r_d \leftarrow r_s + imm$  avec l'immédiat étendu sur 32 bits (de manière signée)
- *sub*  $r_d, r_s, r_t$  ou *subu*  $r_d, r_s, r_t$  avec  $r_i$  des registres et réalisant  $r_d \leftarrow r_s - r_t$

## Détection de capacité et interprétation des valeurs

- Sémantique : lorsque le nom se termine par *u* il n'y a pas de détection de dépassement de capacité, en représentation complément à 2
- L'immédiat et les valeurs contenues dans les registres sont **toujours** interprétés en complément à 2 et vus comme des valeurs signées, qu'il y ait ou non détection de dépassement de capacité
- Comment ça marche alors ?

# Instructions d'addition et soustraction MIPS

## Conclusion

- Extrait de la doc Mips officielle (addu, subu, addiu) : *The term “unsigned” in the instruction name is a **misnomer** ; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.*
- En C, il n'y a pas de débordement d'entiers, même pour les signés ⇒ Utilisation des instructions *addu* et *subu* pour les signés et les non signés
- En Mips, si on veut détecter un débordement lors d'une addition/soustraction d'entiers non signés, il faut le faire en logiciel (i.e. rajouter des instructions qui font des tests sur les opérandes/le résultat)
- Attention : cette remarque n'est pas valable pour les autres instructions (ex : *slt*, *lb*, ...) pour lesquelles le *u* est indispensable !

- 1 Addition et soustraction d'entiers naturels
- 2 Représentation des entiers relatifs et arithmétique entière
- 3 Représentation des caractères alphanumériques**
  - Codage ASCII
  - Codage de chaînes

# Représentation des caractères alphanumériques

## Utilité

Les caractères alphanumériques sont utilisés pour les mots en langage naturel/écrits par utilisateur (texte, programme, commande, email,...)

## Codage des caractères

- Les caractères sur un clavier sont désignés par leur position
- Une table de codage associe à chaque position (éventuellement en combinaison avec d'autres touches) un caractère
- Ce caractère lorsqu'il est rangé en mémoire ou lorsqu'il est manipulé par un programme est codé suivant un codage
- Il existe plusieurs codages différents (UTF8, ASCII, etc.)

# Codage ASCII

Dans ce codage, un caractère est codé sur 1 octet, une table donne le codage Seul 7 des 8 bits sont utilisés, le bit de poids fort vaut toujours 0

Hex	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



Quartet de poids fort

# Extension du codage ASCII

## Extension du codage ASCII

- Des extensions de codage incorporant le 8ème bit de poids fort permettent d'avoir un sous ensemble commun (caractères codés de  $00_h$  à  $7F_h$ ) puis des caractères spécifiques à une langue (codé de  $80_h$  à  $FF_h$ )
- L'extension pour les caractères français (accents, cédille) est latin-1 (ISO 8859-1)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A		;	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	­	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Extension iso-latin

# Codage de chaînes

## Codage d'une chaîne de caractères

- Une chaîne de caractère est composée de plusieurs lettres et symboles
- Codage d'une chaîne = codage de chacune des lettres et symboles la composant + caractère de fin de chaîne : `'\0'` (code 0x00)

## Codage de chaînes en ASCII

- "Lundi" =

# Codage de chaînes

## Codage d'une chaîne de caractères

- Une chaîne de caractère est composée de plusieurs lettres et symboles
- Codage d'une chaîne = codage de chacune des lettres et symboles la composant + caractère de fin de chaîne : `'\0'` (code 0x00)

## Codage de chaînes en ASCII

- "Lundi" = 0x4C ('L') 0x75 ('u') 0x6E ('n') 0x64 ('d') 0x69 ('i') et 0x00 ('\0').
- "123" =

# Codage de chaînes

## Codage d'une chaîne de caractères

- Une chaîne de caractère est composée de plusieurs lettres et symboles
- Codage d'une chaîne = codage de chacune des lettres et symboles la composant + caractère de fin de chaîne : `'\0'` (code `0x00`)

## Codage de chaînes en ASCII

- "Lundi" = `0x4C` ('L') `0x75` ('u') `0x6E` ('n') `0x64` ('d') `0x69` ('i') et `0x00` ('\0').
- "123" = `0x31` ('1') `0x32` ('2') `0x33` ('3') `0x00` ('\0').

## Attention

$$123_d = 64 + 32 + 16 + 8 + 2 + 1$$

Donc sur 8 bits,  $123_d = 0b01111011 = 0x7B$ , et sur 32 bits,  $123_d = 0x0000007B$

$123_d \neq \text{"123"}$