Chapitre 2

Représentation de l'information

2.1 Le problème de la fiabilité

Pour que la fiabilité de l'ensemble que constitue un ordinateur atteigne un niveau raisonnable, il faut que la fiabilité des composants du plus bas niveau soit très élevée. Pour obtenir cette très grande fiabilité de bas niveau, le choix du binaire a été fait ; de plus une certaine redondance peut être introduite dans les circuits de bas niveau avec des codes de détection et de correction d'erreurs.

2.2 Codes binaires

Il faut un *alphabet* à deux symboles pour représenter les deux états possibles :

- **0** et **1**, par référence à la numération décimale,
- **L** et **H** (Low-High), par référence à un niveau électrique,
- **F** et **T** (False-True), par référence à l'algèbre booléenne.

Un objet qui peut prendre l'une de ces deux valeurs est appelé un *bit* (en anglais : **Bi**nary digi**t**). Un "appareil" matériel qui peut conserver un tel objet — et qui peut donc prendre deux états stables physiquement discernables — est une mémoire de un bit.

L'information codée sur un bit est extrêmement rudimentaire : il n'y a que **deux** possibilités. C'est en assemblant des bits qu'on peut agrandir l'espace des possibilités. Ainsi, avec deux bits, on peut coder quatre états : `00`, `01`, `10` et `11`. D'une façon générale, avec n bits, 2^n états différents sont représentables.

> **Notations** Pour notre part, nous utiliserons les symboles **0** et **1**. Pour éviter les confusions entre une chaîne de bits et un nombre en binaire ou en toute autre base qui ne s'écrirait qu'avec des **0** et des **1**, nous utiliserons les conventions suivantes :
> - une chaîne de bits sera toujours écrite dans un rectangle individuel, par exemple `10`,
> - par commodité, les longues chaînes de bits seront écrites "en hexadécimal" précédé de `0x`, par exemple, on écrira `0x3A` plutôt que `0011 1010`,
> - les nombres décimaux sont écrits normalement, ainsi le nombre dix s'écrit 10,
> - dans toute autre base, la base est écrite en indice à la fin du nombre, ainsi deux s'écrit 10_2, et seize s'écrit 10_{16}.

Dans un codage binaire, il s'agit de représenter, donc d'associer, les éléments d'un ensemble (peut-être pas tous : ce n'est pas nécessairement une application[1]) avec des chaînes de bits. Cette association fonctionne dans deux sens, soit on a un élément x de l'ensemble à représenter et on cherche sa *représentation* (ou son *codage*) $C(x)$ qui est une chaîne de bits. Nous appelerons une telle fonction un codage, et nous noterons une telle fonction avec la lettre C. Par exemple, C_{PB} est une fonction de l'ensemble des entiers naturels dans l'ensemble des chaînes de bits. Réciproquement, si

[1] Il y a même des cas où ce n'est pas une fonction, c'est-à-dire que le même objet x a plusieurs codages. Nous en verrons un exemple avec le codage en complément à 1, où zéro a deux représentations.

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	˜
1111	SI	US	/	?	O	_	o	DEL

TAB. 2.1 – Code ASCII

on a une chaîne de bits, on peut l'*interpréter* et chercher quel est son antécédent par la fonction C. Nous noterons cette fonction réciproque avec la lettre I. Par exemple, I_{PB} est une fonction de l'ensemble des chaînes de bits vers l'ensemble des entiers naturels.

2.3 Codage des caractères

2.3.1 L'ASCII

Dans l'alphabet *ordinaire* nous avons 26 lettres majuscules et minuscules, les 10 chiffres décimaux, les caractères de ponctuation et quelques *caractères de contrôle* comme l'espace et la fin de ligne. Pour cet alphabet *ordinaire* un codage sur 7 bits est suffisant et a été normalisé : le code *ASCII* (en anglais : American Standard Code for Information Interchange). Mais cette longueur de 7 bits ayant peu de rapport avec les autres longueurs employées dans les ordinateurs, ce code est souvent étendu sur 8 bits.

La table 2.1 donne le code ASCII, les 3 bits de poids forts sont sur la première ligne, les quatre bits de poids faibles sont dans la première colonne. Nous noterons C_{ascii} la fonction de codage et I_{ascii} la fonction d'interprétation. On a par exemple :

$$C_{ascii}(\text{'A'}) = \boxed{\texttt{0100 0001}}$$
$$I_{ascii}(\boxed{\texttt{0010 0011}}) = \text{'\#'}$$

Historiquement, ce code était destiné aussi à contrôler des *télétypes*, autrement dit des machines à écrire connectées sur les ordinateurs de l'époque. Ces télétypes possédait un chariot d'impression portant le dessin des *caractères imprimables* sur des marteaux. Ces caractères imprimables sont ceux des 6 dernières colonnes du tableau 2.1. Les deux premières colonnes contiennent des *caractères de contrôle*. Ces caractères de contrôle comme leur nom l'indique était destiné à contrôler le fonctionnement du télétype.

Ces caractères de contrôle étaient et sont encore accessibles au clavier soit parcequ'ils ont une touche qui leur est dédiée (typiquement la touche Enter pour le caractère **CR**) ou en combinant la touche Control avec le caractère imprimable mentionné 4 colonnes plus loin dans le tableau 2.1. Ainsi par exemple, **CR** est aussi accessible avec Ctrl-M, **HT** avec Ctrl-I.

Voici la signification de quelques uns de ces caractères de contrôle pour le contrôle des télétypes :

BEL Ctrl-G (en anglais : *Bell*) faisait tinter une cloche installée dans le télétype de façon à attirer l'attention de l'opérateur ;

BS Ctrl-H (en anglais : *Back Space*) faisait reculer d'un cran le chariot d'impression ;

HT Ctrl-I (en anglais : *Horizontal Tabulation*) faisait avancer le chariot à la prochaine position tabulée ;

LF Ctrl-J (en anglais : *Line Feed*) (aussi appelé **NL** (en anglais : *New Line*)) faisait avancer le papier pour passer à la ligne suivante ;

FF Ctrl-L (en anglais : *Form Feed*)(aussi appelé **NP** (en anglais : *New Page*)) faisait avancer le papier pour passer à la page suivante ;

CR Ctrl-M (en anglais : *Carriage Return*) faisait revenir le chariot d'impression au début de la ligne ;

Et voici encore des significations de quelques uns de ces caractères de contrôle dans le cadre de la transmission d'informations entre ordinateurs :

EOT Ctrl-D (en anglais : *End of Text*) indiquait la fin d'un fichier transféré ;

DC1 Ctrl-Q indiquait une demande de reprise de transmission ;

DC3 Ctrl-S indiquait une demande d'interruption de transmission ;

On peut reconnaître au passage que certains ont des significations analogues sur les fenêtres d'émulation de terminaux ou pour les shells de nos systèmes modernes.

2.3.2 IsoLatin-1

Pour les langues européennes, l'Iso a défini 12 normes d'extension de l'ASCII à 8 bits — connues sous le nom Iso/Iec8859-n (avec n de 1 à 12). Ainsi dans chacun des cas, on peut coder 128 caractères supplémentaires. En fait seule l'extension pour l'europe occidentale[2] Iso/Iec8859-1, aussi appellée IsoLatin-1 est bien implantée. Cependant, son utilisation n'est pas universelle et il est encore bien difficile de s'échanger des courriers électroniques avec des lettres accentuées.

2.3.3 Unicode et Iso-10646

La norme Iso/Iec 10646 propose des codages à deux ou quatre octets. Unicode développé par un consortium, est un sous-ensemble 16 bits de Iso/Iec 10646. Ce dernier comprend actuellement 34168 codes distincts pris dans 24 systèmes d'écriture différents. Windows NT fonctionne directement avec Unicode (mais avec une police qui ne contient que 1750 symboles).

2.4 Codage des entiers

2.4.1 Codage en *Décimal Codé Binaire*

Dans le *codage Décimal Codé Binaire* (DCB) (en anglais : **B**inary **C**oded **D**ecimal, BCD) des nombres entiers, chaque chiffre de la notation décimale usuelle est codé en binaire sur un quartet. Par exemple :

$$C_{BCD}(255) = \boxed{0010}\boxed{0101}\boxed{0101}$$

2.4.2 Codage en binaire pur

Le *codage binaire pur* C_{PB} est le seul qui soit naturel : chaque bit de la représentation est interprété comme un chiffre binaire. Sur n bits, ce codage permet de représenter les entiers naturels

[2] Les langues codables sont : allemand, anglais, danois, espagnol, féroïen, finnois, français [bien qu'il manque le caractère 'œ'], islandais, italien, néerlandais, norvégien, portugais, suédois.

compris entre 0 et $2^n - 1$.

$$\begin{array}{rcrcccc} C_{PB}(0) & = & C_{PB}(0_2) & = & \boxed{\texttt{0000 0000}} & = & \boxed{\texttt{0x00}} \\ C_{PB}(16) & = & C_{PB}(10000_2) & = & \boxed{\texttt{0001 0000}} & = & \boxed{\texttt{0x10}} \\ C_{PB}(255) & = & C_{PB}(11111111_2) & = & \boxed{\texttt{1111 1111}} & = & \boxed{\texttt{0xFF}} \end{array}$$

En ajoutant deux nombres représentables, on n'obtient pas nécessairement un nombre représentable. En fait, en ajoutant deux représentations sur n bits, le nombre obtenu est représentable sur $n+1$ bits. Dans un processeur dont les registres sont sur n bits, ce $n+1^{\text{ème}}$ bit est le *bit de retenu* (en anglais : **C**arry). Ce bit est (éphémèrement) stocké dans le *registre d'état*, c'est un des indicateurs.

2.4.3 Codage en complément à un

Le *codage en complément à un* (en anglais : one's complement) C_{1C} est défini comme suit :

$$\begin{array}{rcll} C_{1C}(x) & = & C_{PB}(x), & x \in [0, 2^{n-1} - 1] \\ C_{1C}(x) & = & \overline{C_{PB}(-x)}, & x \in [-2^{n-1} + 1, 0] \end{array}$$

On a par exemple :

$$\begin{array}{rcccc} C_{1C}(-127) & = & \boxed{\texttt{1000 0000}} & = & \boxed{\texttt{0x80}} \\ C_{1C}(-0) & = & \boxed{\texttt{1111 1111}} & = & \boxed{\texttt{0xFF}} \\ C_{1C}(0) & = & \boxed{\texttt{0000 0000}} & = & \boxed{\texttt{0x00}} \\ C_{1C}(127) & = & \boxed{\texttt{0111 1111}} & = & \boxed{\texttt{0x7F}} \end{array}$$

Réciproquement, pour interpréter un mot en complément à un, soit le bit de poids fort est à zéro, et alors le nombre est positif, et donc sa valeur est la même que dans l'interprétation en binaire pur ; soit le bit de poids fort est à un, et alors le nombre est négatif, sa valeur absolue est l'interprétation en binaire pur du complément à un du mot. Pour trouver le complément à un d'un mot, on complémente ses bits un à un.

Par exemple, on veut interpréter $\boxed{\texttt{0010 0011}}$. Cet octet représente en complément à un un nombre positif dont la valeur est :

$$I_{PB}(\boxed{\texttt{0010 0011}}) = 100011_2 = 35$$

et donc :

$$I_{1C}(\boxed{\texttt{0010 0011}}) = 35$$

Autre exemple, on veut interpréter $\boxed{\texttt{1101 1101}}$. Cet octet représente en complément à un un nombre négatif. Calculons le complément à un de cet octet. En inversant tous les bits, on obtient :

$$\boxed{\texttt{0010 0010}}$$

La valeur absolue du nombre que l'on cherche est donc $I_{PB}(\boxed{\texttt{0010 0010}})$, soit 34, et on a donc :

$$I_{1C}(\boxed{\texttt{1101 1101}}) = -34$$

Deux grands inconvénients à cette représentation, d'une part zéro a deux représentations, et d'autre part l'algorithme d'addition est complexe.

2.4.4 Codage en complément à deux

L'interprétation mathématique du *codage en complément à deux* (en anglais : two's complement) C_{2C} est relativement simple : puisqu'on peut représenter 2^n états sur n bits, et qu'il y a une infinité de nombres entiers relatifs, il est naturel de chercher une relation d'équivalence dans l'ensemble des entiers relatifs qui ait 2^n classes d'équivalence. Or les mathématiques nous en fournissent une que l'on connaît bien : la congruence modulo 2^n.

Dans le codage en complément à deux, seuls les nombres de -2^{n-1} jusqu'à $2^{n-1}-1$ sont représentés, et on le définit par :

$$\begin{aligned} C_{2C}(x) &= C_{PB}(x), & x \in [0, 2^{n-1}-1] \\ C_{2C}(x) &= C_{PB}(x+2^n), & x \in [-2^{n-1}, -1] \end{aligned}$$

Sur 8 bits, on a par exemple :

$$\begin{aligned} C_{2C}(-128) &= \boxed{\texttt{1000 0000}} = \boxed{\texttt{0x80}} \\ C_{2C}(-1) &= \boxed{\texttt{1111 1111}} = \boxed{\texttt{0xFF}} \\ C_{2C}(0) &= \boxed{\texttt{0000 0000}} = \boxed{\texttt{0x00}} \\ C_{2C}(127) &= \boxed{\texttt{0111 1111}} = \boxed{\texttt{0x7F}} \end{aligned}$$

Réciproquement, pour interpréter un mot en complément à deux, soit le bit de poids fort est à zéro, et alors le nombre est positif, et donc sa valeur est la même que dans l'interprétation en binaire pur ; soit le bit de poids fort est à un, et alors le nombre est négatif, et il vaut 256 de moins que l'interprétation du mot en binaire pur.

Par exemple, on veut interpréter $\boxed{\texttt{0010 0011}}$. Cet octet représente en complément à deux un nombre positif dont la valeur est :

$$I_{PB}(\boxed{\texttt{0010 0011}}) = 100011_2 = 35$$

et donc :

$$I_{2C}(\boxed{\texttt{0010 0011}}) = 35$$

Autre exemple, on veut interpréter $\boxed{\texttt{1101 1101}}$. Cet octet représente en complément à deux un nombre négatif. L'interprétation en binaire pur de ce mot est :

$$I_{PB}(\boxed{\texttt{1101 1101}}) = 221$$

et donc :

$$I_{2C}(\boxed{\texttt{1101 1101}}) = 221 - 256 = -35$$

L'addition ordinaire est correcte si on interprète les représentations comme des classes d'équivalence de $Z/2^nZ$. Par contre, dans Z, la somme de deux nombres représentables en complément à deux sur n bits n'est pas nécessairement un nombre représentable en complément à deux sur n bits. En fait, il est facile de montrer que :

- la somme d'un nombre strictement négatif et d'un nombre positif est correcte,
- la somme de deux nombres de même signe n'est correcte que si le résultat est aussi du même signe.

Un processeur range dans le bit de *dépassement de capacité* (en anglais : o**V**erflow) la validité de la dernière opération arithmétique qu'il a réalisée. Ce bit est encore un indicateur et se trouve dans le registre d'état.

2.4.5 Codage en signe-valeur absolue

Pour le *codage en signe-valeur absolue* (en anglais : sign-magnitude) sur n bits, le *bit de poids le plus fort* (en anglais : **M**ost **S**ignificant **B**it, MSB) code le signe du nombre avec la valeur 1 pour le signe négatif, les $n-1$ autres bits codent la valeur absolue en binaire pur. Ainsi les nombres entre $-2^{n-1}+1$ et $2^{n-1}-1$ sont représentables.

$$\begin{aligned} C_{SM}(-127) &= \boxed{\texttt{1111 1111}} = \boxed{\texttt{0xFF}} \\ C_{SM}(-0) &= \boxed{\texttt{1000 0000}} = \boxed{\texttt{0x80}} \\ C_{SM}(0) &= \boxed{\texttt{0000 0000}} = \boxed{\texttt{0x00}} \\ C_{SM}(127) &= \boxed{\texttt{0111 1111}} = \boxed{\texttt{0x7F}} \end{aligned}$$

2.4.6 Codage par excès

On définit le codage par excès de M par :

$$C_{+M}(n) = C_{PB}(n + M)$$

pour un codage sur 8 bits, une valeur usuelle de M est 127, une autre est 128. Ainsi sur 8 bits, les nombres représentables en excès de 128 sont ceux compris entre -128 et 127. On a par exemple :

$$\begin{array}{rcccc} C_{+128}(-128) & = & \boxed{\texttt{0000 0000}} & = & \boxed{\texttt{0x00}} \\ C_{+128}(0) & = & \boxed{\texttt{1000 0000}} & = & \boxed{\texttt{0x80}} \\ C_{+128}(127) & = & \boxed{\texttt{1111 1111}} & = & \boxed{\texttt{0xFF}} \end{array}$$

2.5 Codage des nombres flottants

Pour la norme IEEE-754, un nombre flottant x s'écrit sous la forme $x = sm2^e$, où :

s est le signe et vaut 1 ou -1 ;

m est la mantisse ; elle est supérieure ou égale à 0.5 et strictement inférieure à 1 ;

e est l'exposant.

La valeur 1 de s est représentée par une valeur nulle du bit de signe. La valeur -1 de s est représentée par la valeur un dans le bit de signe. L'exposant est représenté sur huit bits avec un excès de 127. Les exposants valides vont de -126 à $+127$. Les extrêmes sont utilisés pour coder les conditions "exceptionnelles" selon le tableau suivant :

Signification	Exposant	Mantisse	Signe
Not-A-Number (NaN)	FF	$\neq 0$	0/1
$+\infty$	FF	0	0
$-\infty$	FF	0	1
$+0$	0	0	0
-0	0	0	1

La mantisse s'écrit en binaire

$$m = 0.1m_2m_3m_4m_5\ldots$$

et seul les bits m_2 à m_{24} sont codés dans la représentation. Les mantisses non-nulles avec un exposant nul indiquent des *nombres dénormalisés*. Les différents champs de bits sont dans l'ordre suivant :

31	30 — 23	22 — 0
s	exp.	mantisse

Par exemple, on a :

$$1.375 = 0.6875 \times 2^1 = (\frac{1}{2} + \frac{1}{8} + \frac{1}{16}) \times 2^1 = 0.1011_2 \times 2^1$$

et la représentation de 1.375 en IEEE-754 est donc :

31	30 — 23	22 — 0
0	10000000	01100000000000000000000

2.6 Exercices

Exercice 2.1 (Conversions de base) *Écrire en base 16 ce nombre (écrit en base 2) :*

- $110\,0101\,0100\,0011\,0010\,0001_2 =$ $\qquad\qquad_{16}$

Écrire en base 8 ce nombre (écrit en base 2) :

- $111\,110\,101\,100\,011\,010\,001_2 =$ $\qquad\qquad_{8}$

Écrire en base 2, 8, et 10 ce nombre (écrit en base 16) :

- $82C_{16}$

Faire les conversions suivantes de la base 10 vers les base 2, 8, et 16 :

1_{10}	2	8	16
2_{10}	2	8	16
4_{10}	2	8	16
8_{10}	2	8	16
16_{10}	2	8	16
32_{10}	2	8	16
64_{10}	2	8	16
128_{10}	2	8	16
256_{10}	2	8	16
512_{10}	2	8	16
1024_{10}	2	8	16
1048576_{10}	2	8	16

Exercice 2.2 (Binaire pur) ***Question 2.2.1*** *Coder en binaire pur sur 8 bits, puis en binaire pur sur 16 bits les nombres* 0 1 23 32 1025 32767 32768 32769 65535 65536 65537. *Tous ces nombres ne sont pas nécessairement représentables dans l'une, voire les deux représentations, dans ce cas, rayez les cases correspondantes. On écrira les représentations en hexadécimal sous la forme* `0xXX` *pour le cas huit bits, et sous la forme* `0xXXXX` *pour le cas seize bits :*

	Binaire pur sur 8 bits	*Binaire pur sur 16 bits*
0		
1		
23		
32		
1025		
32767		
32768		
32769		
65535		
65536		
65537		

Exercice 2.3 (Codages sur huit bits) ***Question 2.3.1*** *Donner les représentations (ou codages) sur un octet écrit en hexadécimal sous la forme* `0xXX` *des nombres suivants en : binaire pur, signe/valeur absolue, excès de 128, complément à 1, et complément à 2 :*

	binaire pur	*signe/v.a.*	*excès de 128*	*compl. à 1*	*compl. à 2*
-256					
-255					
-128					
-127					
-16					
-2					
-1					
0					
1					
2					
16					
127					
128					
255					
256					

Exercice 2.4 (Complément à deux et autres représentations) ***Question 2.4.1*** *Faire apparaître tous les bits des deux octets* `0xFF` *et* `0xC9`.

- `0xFF` = ☐
- `0xC9` = ☐

Question 2.4.2 *Interpréter ces octets en supposant une représentation binaire pur, une représentation par signe-valeur absolue, puis une représentation par excès de $M/2$ (ici 128), une représentation en complément à un, et finalement une représentation en complément à deux. Il s'agit donc de trouver les images de ces deux octets par les fonctions I_{PB}, I_{SM}, I_{+128}, I_{1C}, et I_{2C}.*

	binaire pur	*signe/v.a.*
`0xFF`		
`0xC9`		

	excès de 128	*compl. à 1*	*compl. à 2*
`0xFF`			
`0xC9`			

Question 2.4.3 *Ajouter ces deux octets en binaire comme le fait un processeur, c'est-à-dire que le résultat de l'addition de deux octets donne un octet et positionne les indicateurs* Carry Flag *et* Overflow Flag *:*

`0xFF` + `0xC9` = ☐ + ☐ $\longrightarrow$ ☐ *(Overflow)* ☐ *(Carry)* ☐

Question 2.4.4 *Ecrire les différentes interprétations du résultat, c'est-à-dire donner l'interprétation de l'octet résultat trouvé par le processeur.*

	binaire pur	*signe/v.a.*
"`0xFF` + `0xC9`"		

	excès de 128	*compl. à 1*	*compl. à 2*
"`0xFF` + `0xC9`"			

Question 2.4.5 *Est-ce que l'interprétation de la somme est toujours la somme des interprétations ?*

Exercice 2.5 (Arithmétique en complément à deux) *Pour un processeur qui calcule sur des représentations en complément à 2 sur un octet :*

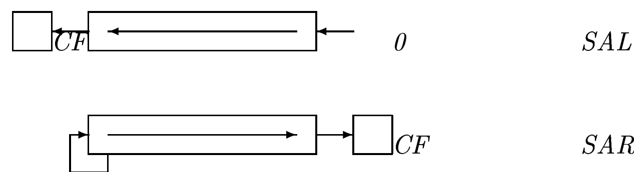
Question 2.5.1 *Le nombre* -128 *est-il représentable ?*

Question 2.5.2 *Quel est l'opposé de la représentation* -128 *?*

Question 2.5.3 *Quelle est la valeur absolue de la représentation* -128 *?*

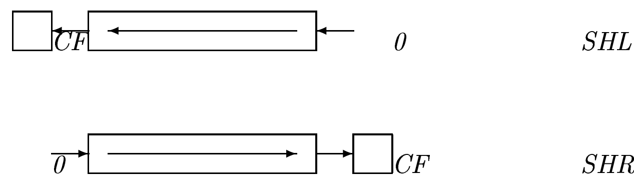
Question 2.5.4 *Mêmes questions pour le nombre* 128.

Exercice 2.6 (Décalages) *Voici décrites graphiquement deux instructions* ***SAL*** *(Shift Arithmetic Left) et* ***SAR*** *(Shift Arithmetic Right). Leur syntaxe est* `SAL count, reg`*, et elles effectuent un décalage de* `count` *bits vers la gauche (ou la droite si c'est* ***SAR****) du registre* `reg`.



Question 2.6.1 *Si la donnée de* `EAX` *est interprétée en complément à deux, qu'effectuent les instructions* `SAL 1, %EAX`, `SAL 3, %EAX`, `SAR 1, %EAX`, `SAR 3, %EAX` *?*

De même, voici les instructions ***SHL****, et* ***SHR****, qui ont la même syntaxe que* ***SAL****, et* ***SAR****. (Remarque :* ***SHL*** *est synonyme de* ***SAL****.)*



Question 2.6.2 *Sur quelles types de données s'interprètent-elles facilement ?*

Exercice 2.7 (Opérations logiques) ***Question 2.7.1*** *Effectuer les opérations suivantes :*

```
0x12345678 AND 0x000000FF
0x12345678 AND 0x00007F00
0x0002 OR 0x0004 OR 0x0400
0x12345678 XOR 0x00007F00
0x12345678 XOR 0x12345678
```

Question 2.7.2 *Quel est l'élément neutre de AND ?*

Question 2.7.3 *Quel est l'élément neutre de OR ?*

Question 2.7.4 *Quel est l'élément absorbant de AND ?*

Question 2.7.5 *Quel est l'élément absorbant de OR ?*

Exercice 2.8 (Débordement en langage de haut niveau) ***Question 2.8.1*** *Que penser de l'extrait de programme suivant si i est un entier en complément à deux sur 8 bits ? En Pascal :*

```
i:=-128;
while i < 128 do begin writeln(i); i:=i+1 end;
```

et en langage C :

```
i= -128;
while (i < 128) { printf("%d\n", i); i=i+1; }
```

Question 2.8.2 *ou de celui-ci : En Pascal :*

```
i:=-128;
while i <= 127 do begin writeln(i); i:=i+1 end;
```

et en langage C :

```
i= -128;
while (i <= 127) { printf("%d\n", i); i=i+1; }
```

Question 2.8.3 *Sur votre ordinateur et avec votre compilateur, comment sont représentés les entiers ?*

Exercice 2.9 (Conversion d'entiers) ***Question 2.9.1*** *Comment convertir un entier en complément à deux sur m bits en un entier en complément à deux sur n bits si :*

- $m < n$
- $m > n$

Question 2.9.2 *Mêmes questions avec des entiers en binaire pur.*

Exercice 2.10 (Représentation des flottants) ***Question 2.10.1*** *Ecrire en simple précision IEEE-754 les nombres suivants :*

- -1.375
- -0.6875
- -0.375
- -0.34375

Question 2.10.2 *Ajouter les nombres en simple précision* 1.375 *et* 0.6875.

Exercice 2.11 (Représentation des flottants) ***Question 2.11.1*** *Représenter sur un axe réel l'ensemble des nombres flottants représentés avec les mêmes conventions que la norme IEEE-754, mais avec seulement 3 bits d'exposants et 4 bits de mantisse.*

Frobnitz, pl. Frobnitzem (frob'nitsm) n. :
An unspecified physical object, a widget. Also refers to electronic black boxes. This rare form is usually abbreviated to FROTZ, or more commonly to FROB. Also used are FROBNULE, FROBULE, and FROBNODULE. Starting perhaps in 1979, FROBBOZ (fruh-bahz'), pl. FROBBOTZIM, has also become very popular, largely due to its exposure via the Adventure spin-off called Zork (Dungeon). These can also be applied to non-physical objects, such as data structures.