Représentation et codage de l'information

M1103 - Architecture des équipements informatiques

Sebastien.Kramm@univ-rouen.fr

IUT R&T Rouen, site d'Elbeuf

2018-2019

Codage des entiers positifs

- Pour coder des entiers, on utilise le codage binaire naturel, en spécifiant par des moyens annexes :
  - le nombre de bits utilisés pour chaque symbole,
  - l'ordre dans lequel on envoie poids fort / poids faible (notion d'*endianess*).
- Par exemple, le message suivant :
  0001.0000.0000.0010.0000.0100.0000.0001
  pourra représenter :
  - Si les valeurs sont codées sur 4 bits, le message aura une longueur de 8 mots et sera : 1 ;0 ;0 ;2 ;0 ;4 ;0 ;1
  - Si les valeurs sont codées sur 8 bits, le message aura une longueur de 4 mots et sera : 16 ;2 ;4 ;1
  - Si les valeurs sont codées sur 16 bits, le message aura une longueur de 2 mots et sera : 4098 ;1025

Comment représenter en binaire un nombre relatif ?

- Pour encoder le signe, on réserve un bit, appelé le bit de signe, et qui sera **toujours** en position de poids fort (MSB).
  - 0 : nombre positif
  - 1 : nombre négatif
- Plusieurs solutions sont envisageables :
  1. bit de signe + binaire naturel
     $4 = 0000.0100 \rightarrow$ -4 = 1000.0100
  2. bit de signe + complément à un
     $4 = 0000.0100 \rightarrow$ -4 = 1111.1011
  3. complément à deux
- Problème des méthodes 1 et 2 : deux représentations possibles pour 0 :
  1. bit de signe + binaire naturel
     $+0_{10} = 0000.0000 \leftrightarrow -0_{10} = 1000.0000$
  2. bit de signe + complément à un
     $+0_{10} = 0000.0000 \leftrightarrow -0_{10} = 1111.1111$
- De plus, seul le complément à deux permet de faire de l'arithmétique correcte avec la même unité de calcul.

Notation en complément à deux

- Cette représentation s'entend pour un **nombre de bits donné**.
- Pour une valeur sur n bits, on dispose de $n-1$ bits pour coder la valeur (MSB : bit de signe).
- Nombre positif : codage binaire naturel.
- Nombre négatif : complément à deux :
  1. Complémentation de tous les bits
  2. Addition de 1 à la valeur obtenue
- Exemple : codage de la valeur -14 sur 8 bits :
  $14_{10} = 8 + 4 + 2 = (0000.1110)_2$
  1. Complément à 1 : $\overline{0000.1110} = 1111.0001$
  2. Ajout de 1 : $1111.0001 + 1 = 1111.0010 =$ `0xF2`
- Exemple : codage de la valeur -14 sur 16 bits :
  $14_{10} = (0000.0000.0000.1110)_2$
  1. Complément à 1 : 1111.1111.1111.0001
  2. Ajout de 1 : 1111.1111.1111.0010 = `0xFFF2`

Intérêt du complément à deux

- Opération : $5 + 4$, sur 8 bits

	binaire	décimal
	0000.0101	5
+	0000.0100	4
	0000.1001	9

- Opération : $5 - 4$, sur 8 bits
  Le processeur va en fait exécuter l'opération : 5 + (-4)

	binaire	décimal
	0000.0101	5
+	1111.1100	-4
	1.0000.0001	1

- Remarque : Le bit 1 en position 2^9 est la **retenue**, dont il ne faut pas tenir compte ici (calcul sur 8 bits).

Plage de valeurs en complément à deux

- Rappel : en binaire naturel, pour n bits, on peut encoder 2^n valeurs, qui représentent les valeurs de 0 à $2^n - 1$.

Par exemple, $n = 8 \Rightarrow 256$ valeurs, de 0 à 255.

- En complément à deux, on ne dispose plus que de $n - 1$ bits...
  ... mais on a toujours 2^n valeurs possibles !
- Ces 2^n valeurs sont pour moitié positives et négative.

Par exemple, $n = 8 \Rightarrow 128$ valeurs positives et 128 valeurs négatives.

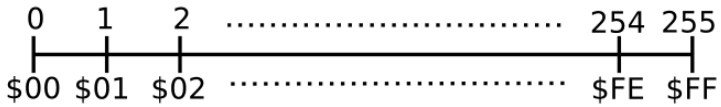
- Comme la valeur 0 est (arbitrairement) positive, l'étendue de chaque zone est donc différente.
  Par exemple, si $n = 8 \rightarrow -128 < N < +127$

Généralisation

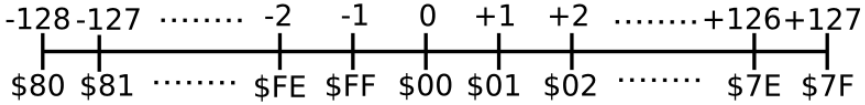
En complément à deux sur n bits, la valeur numérique aura une étendue de -2^{n-1} à $+2^{n-1} - 1$

Etendue numérique

- en non-signé sur 8 bits :



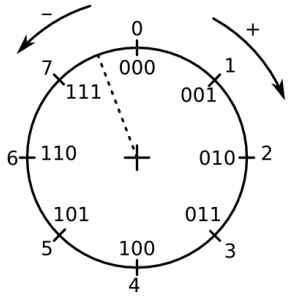
- en signé sur 8 bits :



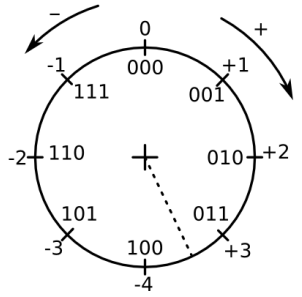
Représentation circulaire

- On peut montrer les deux représentations sous la forme d'un cercle, par exemple pour 3 bits :

- en "non-signé" :



- en "signé" :



- On aura un débordement lorsqu'une opération va **franchir** le trait pointillé.

Encodage des nombres réels

- Soit le nombre (base 2) 1.0010.0111.0001.0000.1010,1110.0010.1011.101
- Question : comment encoder cette valeur dans un ordinateur ?
- Deux approches envisageables :
  - Codage en virgule fixe : on impose un format, n bits pour la partie entière, m bits pour la partie fractionnaire.
  - Codage en virgule flottante : on utilise la notation $N = M \cdot b^E$ vue précedemment, et on assigne n bits pour coder M et m bits pour coder E, plus 1 bit pour le signe.

Représentation en virgule fixe

- Principe : on décide (arbitrairement ou en fonction d'un contexte) d'une position fixe pour la virgule.
- Exemple (en binaire et sur 8 bits) : XXXXX,XXX
  (5 bits pour la partie entière, 3 bits pour la partie fractionnaire)
- calcul :

	base 2	base 10
	01000,100	8,5
+	00100,001	4,125
	01100,101	12,625

Inconvénients

- Erreur d'arrondi importante.
  Exemple : soit l'opération $4,125/2 \times 2$
  $\rightarrow$ 00100,001 / 2 = 00010,000
  $\rightarrow$ $00010,000 \times 2 = 00100,000 = 4,0$
- Impossible de représenter à la fois des très grand nombres et des très petits.

Représentation en virgule flottante

- Pour eviter les erreurs d'arrondi inhérent au codage en virgule fixe, la plupart des machines proposent un codage en virgule flottante.
- Un nombre sera codé sous la forme $N = M \cdot 2^E$, avec :
  - M : mantisse **normalisée**,
  - E : exposant
- On pourra ainsi facilement comparer des nombres entre eux.

En pratique

La grande majorité des machines modernes utilisent la norme **IEEE 754** pour le codage des nombres.

Norme IEEE 754

- Cette norme définit (principalement) deux formats de stockage des nombres représentés en virgule flottante.
  - *Simple précision* : 32 bits (4 octets)
    - 1 bit de signe : $N >= 0 \rightarrow S = 0$, $N < 0 \rightarrow S = 1$
    - 8 bits d'exposant (décalé) : -128 à +127
    - 23 bits de mantisse : 0 à $2^{23} - 1$
  - *Double précision* : 64 bits (8 octets)
    - 1 bit de signe
    - 11 bits d'exposant (décalé) : - 1024 à +1023
    - 52 bits de mantisse 0 à $2^{52} - 1$

Codage en IEEE 754 - Mantisse

- Mantisse : on code la mantisse normalisée **sans** le premier 1.
  En effet, comme **tout** nombre normalisé aura un '1' comme premier chiffre[1], on le considère implicite (on ne le mémorise pas), et on gagne un bit !
- Exemple : soit à encoder le nombre 1,25
  En binaire : $1,25 = 1 + 1/4 = 2^0 + 2^{-2} = (1,01)_2$
  La mantisse à coder sera : ,01
  La mantisse stockée sur 23 bits sera : 0100.0000.0000.0000.0000.000

Au décodage...

... Penser à **ajouter** ce '1' !

- Example : soit la mantisse encodée suivante (23 bits) :
  1111.0000.1111.0000.0000.000
  $\Rightarrow$ la mantisse **réelle** du nombre sera :
  1,1111.0000.1111

1. Sauf la valeur 0, évidemment...

Codage en IEEE 754 - Exposant

- L'exposant (E) peut être positif ou négatif, $\forall$ la base.
  (Exemple en base 10 : 5 mm $= 5 \cdot 10^{-3}$ m. ; 5 km $= 5 \cdot 10^3$ m.)
- Afin d'éviter d'avoir à coder ce signe, la norme IEEE-754 prévoit d'ajouter un *offset* (décalage) à l'exposant avec une valeur fixe.
- Au décodage, il faudra bien sur **retrancher** cette même valeur.
- Les valeurs de ce décalage sont :
  - Simple précision : d=127
  - Double précision : d= 1023
- Exemple 1 (simple précision) : E=3 $\Rightarrow$ on encode la valeur $127 + 3 = 130 = (1000.0010)_2$
- Exemple 2 (simple précision) : E=-19 $\Rightarrow$ on encode la valeur $127 - 19 = 108 = (0110.1100)_2$

IEEE 754 - nombres particuliers

- Certaines valeurs donnent lieu à un codage particulier :
  (X : valeur quelconque)

Valeur	S	Mantisse	Exposant
Zéro	0	0...0	0...0
NAN	X	X...X, sauf 0...0	1...1
$+\infty$	0	0...0	1...1
$-\infty$	1	0...0	1...1

- NAN (*Not A Number*) est utilisé pour signaler des erreurs de calcul.
  (C'est par exemple la valeur renvoyée en cas de tentative de calcul d'une racine carrée d'un nombre négatif.)
- ∞ (souvent noté "inf") est la valeur renvoyée par exemple en cas de division par 0.

Code ASCII

- 1963, codage des caractères latins non accentués uniquement.
- Codage sur 7 bits : `00h` à `7Fh`.
- Les caractères de numéro 0 à 31 (0 à `1Fh`) et le 127 (`7Fh`) ne sont pas affichables ; ils correspondent à des commandes de contrôle de terminal informatique.

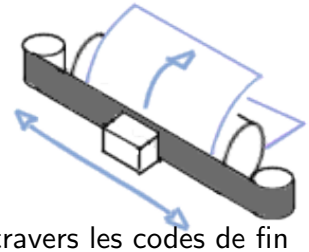
En hexadécimal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Exemple : 'A' = 65 = $41

Codes de contrôle de l'ASCII

- DEL (*Delete*) : effacement. Correspond sur un clavier contemporain à la touche du même nom. (retour arrière en effaçant).
- Fin de ligne : deux codes sont utilisés
  - LF (*Line Feed*), saut de ligne, code 10 (0x0A)
  - CR (*Carriage Return*), retour chariot, code 13 (0x0D)
- Historiquement, ceci correspondait aux deux opérations nécéssaires sur une imprimante à rouleaux :
  - faire tourner le rouleau d'un cran,
  - ramener la tête d'impression à gauche.



- Ceci perdure dans l'informatique moderne, à travers les codes de fin de ligne dans les fichiers texte :
  - Certains OS (Windows) utilisent les 2 codes (CR suivi de LF).
  - D'autres (Linux, Mac OS X, etc.) n'en utilisent qu'un seul (LF).
- D'où des problèmes parfois lors de transferts de fichiers entre machines.

Extensions aux code ASCII

- Pour pouvoir coder les caractères de chaque langue, on a d'abord utilisé les 127 autres valeurs (0x80 à 0xFF).
- Problème : insuffisant pour tout coder.
- Solution : notion de **page de code** : on spécifie en amont quel jeu de caractère est utilisé pour les valeurs entre 128 et 255.

Page de code 850 (DOS Latin 1) — mode standard

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8x	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9x	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
Ax	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
Bx	░	▒	▓	│	┤	Á	Â	À	©	╣	║	╗	╝	¢	¥	┐
Cx	└	┴	┬	├	─	┼	ã	Ã	╚	╔	╩	╦	╠	═	╬	¤
Dx	ð	Ð	Ê	Ë	È	ı	Í	Î	Ï	┘	┌	█	▄	¦	Ì	▀
Ex	Ó	ß	Ô	Ò	õ	Õ	µ	þ	Þ	Ú	Û	Ù	ý	Ý	¯	´
Fx	SHY	±	‗	¾	¶	§	÷	¸	°	¨	·	¹	³	²	■	NBSP

Extensions aux code ASCII

- Quelques pages de codes usuelles :
  - CP 850
  - CP 1252
  - ISO 8859-1 (Latin-1)
  - ISO 8859-15 (Latin-9)
  - Windows-1252
  - etc.
- Problème : multiples pages de codes, et interopérabilité limitée.
- Impossible d'avoir dans un même document des symboles spécifiques à deux pages de code différentes !

Unicode

- Standard informatique qui permet des échanges de textes dans différentes langues, à un niveau mondial.
- 109 000 caractères couvrant 93 écritures.
  - Chaque caractère abstrait est identifié par un nom unique et associé à un nombre entier positif appelé son **point de code** (différent de son codage !)
  - L'espace de codage est divisé en 17 zones de 65 536 points de codes. Ces zones sont appelées **plans**. (Référence)
  - Le point de code est noté U+xxxx où xxxx est en hexadécimal, et comporte 4 à 6 chiffres :
    - 4 chiffres pour le premier plan, appelé plan multilingue de base (donc entre U+0000 et U+FFFF) ;
    - 5 chiffres pour les 15 plans suivants (entre U+10000 et U+FFFFF) ;
    - 6 chiffres pour le dernier plan (entre U+100000 et U+10FFFF).
  - Exemple : le symbole ê a pour point de code U+00EA

Unicode : Encodage

- La transformation d'un point de code passe par une **représentation** en UTF-8, UTF-16 ou UTF-32.
- Le nombre après "UTF" spécifie le nombre de bits **minimal** avec lequel un caractère est codé.
  - Windows utilise en interne UTF-16 : chaque caractère est codé sur 2 octets (au minimum).
  - Linux utilise l'UTF-8.
  - Le Web moderne est en UTF-8.

Avantage de l'UTF-8

Les caractères ASCII sont encodés de façon identique, sur 1 octet :
'A' en ASCII = $61 = 'A' en UTF-8

UTF-8

- Code de taille variable, 4 formats selon le nombre de bits du point de code à encoder (de 1 à 4 octets)

Représentation binaire UTF-8	Signification
`0xxxxxxx`	1 octet codant 1 à 7 bits
`110xxxxx 10xxxxxx`	2 octets codant 8 à 11 bits
`1110xxxx 10xxxxxx 10xxxxxx`	3 octets codant 12 à 16 bits
`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	4 octets codant 17 à 21 bits

- Remarques :
  - Tout octet de bit de poids fort **nul** désigne un point de code assigné à un caractère ASCII (un seul octet) ;
  - Tout octet de bits de poids fort valant 11 est le **premier** octet d'une séquence représentant un point de code codé sur plusieurs octets ;
  - Tout octet de bits de poids fort valant 10 est un des octets **suivants** d'une séquence unique représentant un point de code codé sur plusieurs octets ;

UTF-8 : Exemple d'encodage

- Soit par exemple le point de code `U+03F0`
  1. On l'écrit en binaire pour avoir le nombre de bits nécessaires :
     `03F0` = `0000.0011.1111.0000` = `11.1111.0000`
     ⇒ 10 bits nécessaires, on utilise donc la **2e** ligne du tableau :
     `110x.xxxx - 10xx.xxxx`
  2. On place les 10 bits dans le masque (en ajoutant un 11e bit à 0 à gauche) :
     5 sur le 1er octet, 6 sur le 2e :
     `1100.1111 - 1011.0000`
  3. L'encodage de ce point de code sera donc : `CF - D0`
- Autres exemples de points de code :

	point de code	Nb de bits	binaire	encodage utf8
é	U+00E9	8	1110.1001	C3 A9
€	U+20AC	14	10.0000.1010.1100	E2 82 AC

Décodage UTF-8

Question : je prends un octet au hasard dans un fichier texte encodé en UTF-8.
Comment savoir s'il s'agit d'un caractère ASCII ou "multi-octets" ?
Reponse :

- **Si** le premier bit est 0 → caractère ASCII
- **Si** les deux premiers bits sont "11"
  → 1eroctet d'un caractère codé sur plusieurs octets
- **Si** les deux premiers bits sont "10"
  → fait partie d'un caractère codé sur plusieurs octets, mais n'est **pas** le 1er

UTF-8 : Exemple de décodage

- Soit par exemple le caractère encodé en UTF8 sur deux octets : `C6 - A5`

1. On l'écrit en binaire :
   `1100.0110 - 1010.0101`
   Ceci correspond à la 2ème ligne du tableau, le masque utilisé est :
   `110xxxxx - 10xxxxxx`
2. On "extrait" les 5 bits de droite du 1er octet et les les 6 bits de droite du 2e octet :
   `11000110 - 10100101`
   Ce qui donne les 11 bits suivants : `00110.100101`
   Répartis par 4 : `001.1010.0101` soit en hexa : `1A5`
   Le point de code est donc : `U+01A5`

Base64

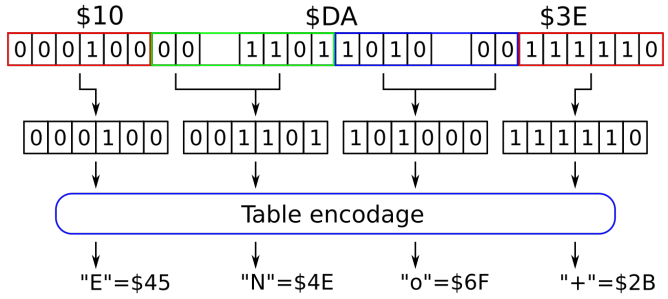
- Comment faire passer un fichier binaire (exécutable, image, vidéo, ...) dans un canal qui ne permet que du texte en ASCII 7 bits (mail) ?
- Solution : encodage "base64" : on découpe la source par groupe de 3 octets (24 bits) et on associe à chaque groupe de 6 bits un symbole (via une table d'encodage), encodé en ASCII.

```
--------------722B9F7E8BCDF3CE442C5EB1
Content-Type: image/jpeg;
 name="20170621_191701_resized.jpg"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
 filename="20170621_191701_resized.jpg"

/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAIBAQEBAQIBAQECAgICAgQDAgICAgUEBAMEBgUG
BgYFBgYGBwkIBgcJBwYGCAsICQoKCgoKBggLDAsKDAkKCgr/2wBDAQICAgICAgUDAwUKBwYH
CgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgr/wAARC
CAgQBIkDASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAA
AgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkK
```

ref : `https://fr.wikipedia.org/wiki/Base64`

Base64 : Exemple



Valeur	Codage	Valeur	Codage	Valeur	Codage	Valeur	Codage
0	000000 A	17	010001 R	34	100010 i	51	110011 z
1	000001 B	18	010010 S	35	100011 j	52	110100 0
2	000010 C	19	010011 T	36	100100 k	53	110101 1
3	000011 D	20	010100 U	37	100101 l	54	110110 2
4	000100 E	21	010101 V	38	100110 m	55	110111 3
5	000101 F	22	010110 W	39	100111 n	56	111000 4
6	000110 G	23	010111 X	40	101000 o	57	111001 5
7	000111 H	24	011000 Y	41	101001 p	58	111010 6
8	001000 I	25	011001 Z	42	101010 q	59	111011 7
9	001001 J	26	011010 a	43	101011 r	60	111100 8
10	001010 K	27	011011 b	44	101100 s	61	111101 9
11	001011 L	28	011100 c	45	101101 t	62	111110 +
12	001100 M	29	011101 d	46	101110 u	63	111111 /
13	001101 N	30	011110 e	47	101111 v		
14	001110 O	31	011111 f	48	110000 w	(complément)	=
15	001111 P	32	100000 g	49	110001 x		
16	010000 Q	33	100001 h	50	110010 y		