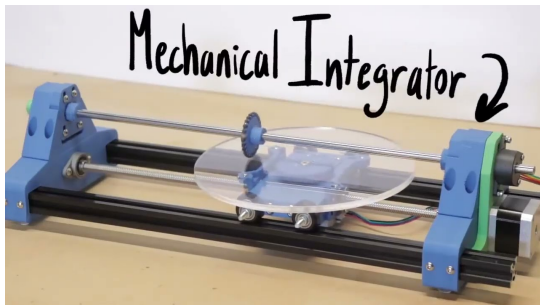


Architecture des ordinateurs (X31I050)

Frédéric Goualard

Laboratoire d'Informatique de Nantes-Atlantique, UMR CNRS 6241
Bureau 112, bât. 11
Frederic.Goualard@univ-nantes.fr

Représentation de l'information



Ordinateur analogique :

- ▶ Transformation d'un problème en un autre *analogue* (exemple : addition de nombres → addition de résistances) résolvable directement et physiquement sur des *valeurs continues*

Ordinateur numérique :

- ▶ Transformation d'un problème en une suite d'instructions sans analogie sur des *valeurs discrètes* puis exécution du code

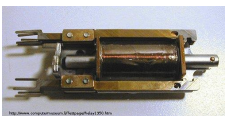
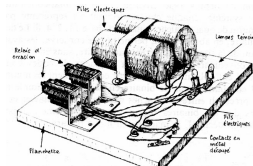


George Boole, Algèbre booléenne, 1847

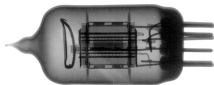
George Stibitz, model K 1937 (« kitchen ») :

Claude Shannon, 1937

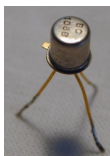
John V. Atanasoff, 1937–1942 (ABC)



Relais électromécanique (Henry, 1835)



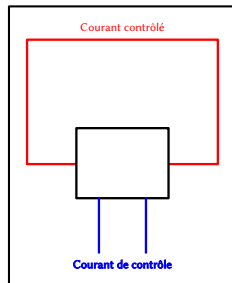
Tube thermionique
"lampe à vide" (De Forest, 1907)



Transistor
(Shockley, Bardeen & Brattain, 1947)

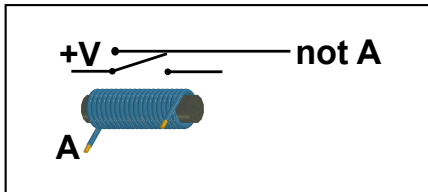
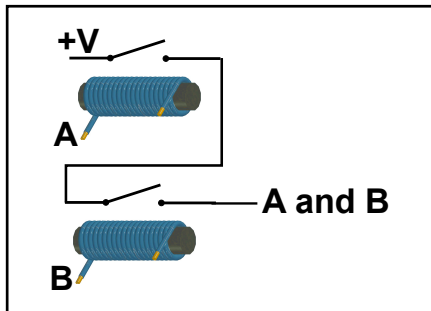
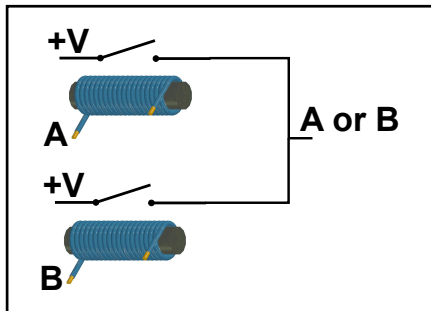


Circuit intégré (Kilby, 1958)



Ordinateur numérique (\neq analogique) : architecture utilisant l'absence ou la présence de courant (relais, tubes ou transistors)

→ 2 états (0 et 1) : représentation *binnaire*



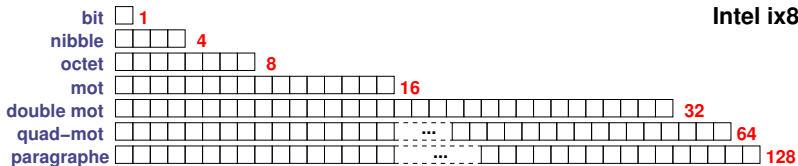
- ▶ Deux états internes symbolisés par 0 et 1
- ▶ Informations :

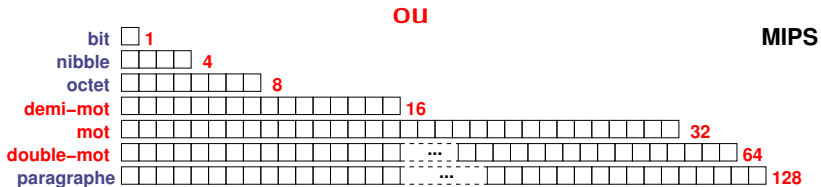
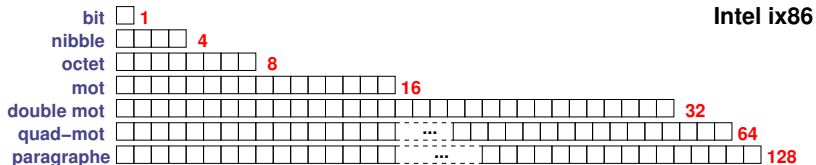
Nom	Valeur
<i>Bit</i>	0 ou 1
<i>Octet</i>	00000000 à 11111111

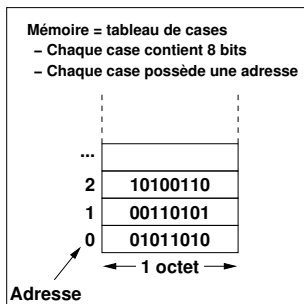
Multiples définis depuis 1998 :

Nom	Notation	Valeur
1 kibibit	1 Kibit	$2^{10} = 1\,024$ bits
1 kilobit	1 kbit	$10^3 = 1\,000$ bits
1 mebioctet	1 MiB/1 Mio	$2^{20} = 1\,048\,576$ octets
1 megaoctet	1 MB/1 Mo	$10^6 = 1\,000\,000$ octets
1 gibioctet	1 GiB/1 Gio	$2^{30} = 1\,073\,741\,824$ octets
1 gigaoctet	1 GB/1 Go	$10^9 = 1\,000\,000\,000$ octets

Intel ix86





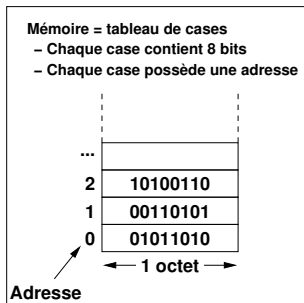


On veut stocker :

- ▶ des entiers positifs (12, 534256, ...)
- ▶ des entiers négatifs (-56, -435345, ...)
- ▶ des caractères ('a', 'Z', '5', '+', ...)
- ▶ des chaînes de caractères ("bonjour", ...)
- ▶ des réels (12.34, -670.5552, ...)
- ▶ des instructions

Mais :

Une case contient uniquement des bits



On veut stocker :

- ▶ des entiers positifs (12, 534256, ...)
- ▶ des entiers négatifs (-56, -435345, ...)
- ▶ des caractères ('a', 'Z', '5', '+', ...)
- ▶ des chaînes de caractères ("bonjour", ...)
- ▶ des réels (12.34, -670.5552, ...)
- ▶ des instructions

Mais :

Une case contient uniquement des bits

⇒ tout coder sous forme d'*entiers positifs en binaire*

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
xxxx	illégal

Décimal :

3406

BCD :

001101000000110

14 bits

- ▶ Codage BCD gourmand
(12 bits seulement en binaire pour coder 3406)
- ▶ Opérations arithmétiques compliquées et pas efficaces
- ▶ Entrées/sorties facilitées

Représentation usuelle : 12

$$= 1 \times 10^1 + 2 \times 10^0$$



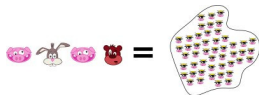
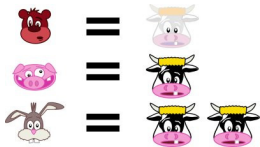
→ Représentation *positionnelle* :

- ▶ Choix d'une base b : (ex. : 10, 2, ...)
- ▶ Choix de b symboles

Exemples :

Base 2 (0, 1) : $1100_2 (= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 12_{10})$

Base 3 « Toons » :



$$1 \times 3^3 + 2 \times 3^2 + 1 \times 3^1 + 0 \times 3^0 = 48$$

Expression d'un nombre a en base b :

$$\begin{aligned}
 a_b &= (a_n a_{n-1} \dots a_2 a_1 a_0 . a_{-1} a_{-2} \dots a_{-m})_b \\
 &= a_n b^n + \dots + a_2 b^2 + a_1 b + a_0 + a_{-1} b^{-1} + \dots a_{-m} b^{-m}
 \end{aligned}$$

a_n : chiffre *le plus significatif*

a_{-m} : chiffre *le moins significatif*

Exemples :

$$\begin{aligned}
 98_{10} &= 9 * 10^1 + 8 * 10^0 \\
 101_2 &= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 &&= 5_{10} \\
 136_8 &= 1 * 8^2 + 3 * 8^1 + 6 * 8^0 &&= 94_{10} \\
 3A_{16} &= 72_8 &&= 58_{10} \\
 110.01_2 &= 1 * 2^2 + 1 * 2^1 + 0 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} &&= 6.25_{10}
 \end{aligned}$$

Avec $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, ...

Passage de 23_{10} en base 10, 2, 3?

Division par 10, 2, 3 itérées.

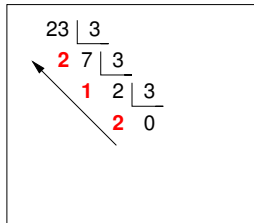
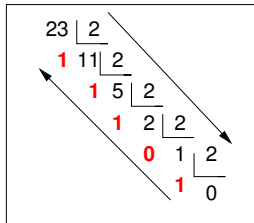
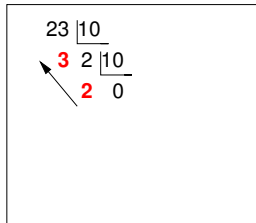
$$\begin{array}{r}
 23 \overline{)10} \\
 \underline{32} \\
 20 \\
 \underline{20} \\
 0
 \end{array}$$

$$\begin{array}{r}
 23 \overline{)2} \\
 \underline{111} \\
 15 \\
 \underline{12} \\
 12 \\
 \underline{01} \\
 10
 \end{array}$$

$$\begin{array}{r}
 23 \overline{)3} \\
 \underline{27} \\
 12 \\
 \underline{12} \\
 0
 \end{array}$$

Passage de 23_{10} en base 10, 2, 3?

Division par 10, 2, 3 itérées.



Résultat : $(23) = 23_{10} = 10111_2 = 212_3$

Passage de la représentation de u en base r à sa représentation en base R ?

$$\begin{aligned} u &= (x_{k-1}x_{k-2}\dots x_0)_r \\ &= (X_{K-1}X_{K-2}\dots X_0)_R \end{aligned}$$

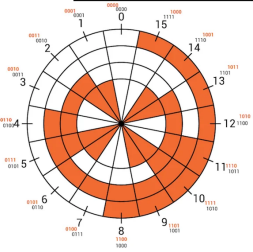
Représentation de Horner :

$$\begin{aligned} u &= (X_{K-1}R^{K-1} + X_{K-2}R^{K-2} + \dots + X_0)_r \\ &= (R(\dots R(R \times 0 + X_{K-1}) + X_{K-2}) + X_{K-3}) \dots + X_0)_r \end{aligned}$$

en exprimant toutes les constantes et en faisant toutes les opérations dans la base r .

⇒ Algorithme : divisions itérées par R en base r

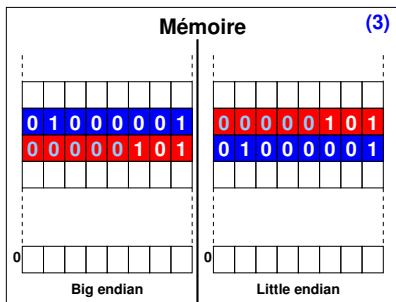
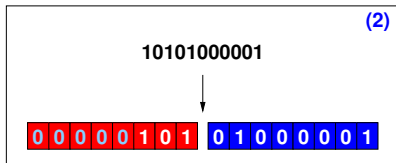
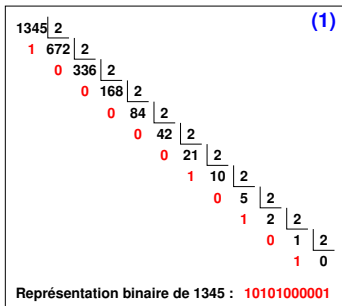
- ▶ Choix d'un ordre pour entiers binaires tel que chaque élément diffère par un seul bit du suivant et du précédent :

	Code de Gray BRGC	Ordre naturel	Nombre
	000	000	0
	001	001	1
	011	010	2
	010	011	3
	110	100	4
	111	101	5
	101	110	6
	100	111	7

- ▶ Ordre non unique (*Binary Reflected Gray Code*)
- ▶ Utilisation : roue codeuse, tableau de Karnaugh, code de correction d'erreur, ...

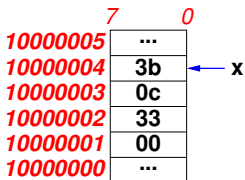
Stockage d'un entier positif en mémoire :

1. Représentation du nombre en binaire
2. Découpage de la représentation binaire en octets
3. Stockage de chaque octet consécutivement

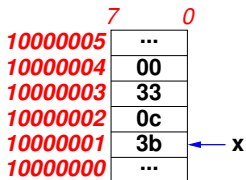


- ▶ Architecture ix86 : adressage par octet *little-endian*
- ▶ Stockage d'infos sur plus d'un octet :
 - ▶ msb (*Most Significant Byte*) à l'adresse la plus petite
 - ➔ big-endian
 - ▶ lsb (*Least Significant Byte*) à l'adresse la plus petite
 - ➔ little-endian

unsigned long int x = 3345467;
 { => x = 0x00330c3b; }



big-endian




little-endian

Entiers représentables sur 1 octet :

Base 2	Base 10	Base 16
11111111	255	FF
11111110	254	FE
⋮	⋮	
00010001	17	11
00010000	16	10
00001111	15	0F
00001110	14	0E
00001101	13	0D
00001100	12	0C
00001011	11	0B
00001010	10	0A
00001001	9	09
00001000	8	08
00000111	7	07
00000110	6	06
00000101	5	05
00000100	4	04
00000011	3	03
00000010	2	02
00000001	1	01
00000000	0	00

Entiers représentables sur 1 octet :

Base 2	Base 10	Base 16
11111111	255	FF
11111110	254	FE
⋮	⋮	⋮
00010001	17	11
00010000	16	10
00001111	15	0F
00001110	14	0E
00001101	13	0D
00001100	12	0C
00001011	11	0B
00001010	10	0A
00001001	9	09
00001000	8	08
00000111	7	07
00000110	6	06
00000101	5	05
00000100	4	04
00000011	3	03
00000010	2	02
00000001	1	01
00000000	0	00



Passage de base 2 à base 16 et inversement :

Base 2 : 1000101001

Base 16 : 2 2 9

B2	B16
1111	F
1110	E
1101	D
1100	C
1011	B
1010	A
1001	9
1000	8
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0

Entiers non-signés : ensemble d'entiers positifs

Entiers signés : ensemble d'entiers positifs et négatifs

Comment représenter des entiers négatifs ?

Entiers non-signés : ensemble d'entiers positifs

Entiers signés : ensemble d'entiers positifs et négatifs

Comment représenter des entiers négatifs ?

Convention de recodage des chaînes de bits

- ▶ Magnitude signée
- ▶ Complément à 1
- ▶ Complément à 2
- ▶ Biaisée

Dans un entier de k bits, le bit de poids fort code le signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Exemples (sur 8 bits) :

- ▶ $+25_{10} = 00011001_2$
- ▶ $-25_{10} = 10011001_2$

Inconvénient = deux représentations pour 0 :

- ▶ $+0_{10} = 00000000_2$
- ▶ $-0_{10} = 10000000_2$

Sur 8 bits : -127..+127

Dans un entier de k bits, le bit de poids fort code le signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Exemples (sur 8 bits) :

- ▶ $+25_{10} = 00011001_2$
- ▶ $-25_{10} = 10011001_2$

Inconvénient = deux représenta

- ▶ $+0_{10} = 00000000_2$
- ▶ $-0_{10} = 10000000_2$

Sur 8 bits : -127..+127

chaîne de bits	non signé	magnitude signée
1111	15	-7
1110	14	-6
1101	13	-5
1100	12	-4
1011	11	-3
1010	10	-2
1001	9	-1
1000	8	-0
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0

Le bit de poids fort correspond au signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en complémentant bit à bit sa valeur absolue avec 1 (cf. complément à 9 de la *Pascaline*)

Exemple : représentation de -25_{10} sur 8 bits :

- ▶ $25_{10} = 00011001_2$

$$\begin{array}{r}
 1111111 \\
 \text{▶ D'où : } - 00011001 \\
 \hline
 = 11100110
 \end{array}$$

Deux représentations pour 0 : 00000000_2 et 11111111_2

Nombres représentés sur 8 bits : $-127..+127$

Le bit de poids fort correspond au signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en complément
absolue avec 1 (cf. complément à 9 d'abord)

Exemple : représentation de -25_{10} sur 8 bits

- ▶ $25_{10} = 00011001_2$

$$\begin{array}{r} \phantom{D'o\grave{u} : -} 11111111 \\ D'o\grave{u} : - \quad 00011001 \\ \hline = 11100110 \end{array}$$

Deux représentations pour 0 : 00000001 et 11111111

Nombres représentés sur 8 bits : -127 à 127

chaîne de bits	non signé	complément à 1
1111	15	-0
1110	14	-1
1101	13	-2
1100	12	-3
1011	11	-4
1010	10	-5
1001	9	-6
1000	8	-7
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0

Le bit de poids fort indique le signe :

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en ajoutant 1 au complément à 1 de sa valeur absolue (et inversement).

Exemple : représentation de -25_{10} ?

- ▶ $+25_{10} = 00011001_2$
- ▶ Complément à 1 de $+25_{10} = 11100110_2$
- ▶ Ajout de 1 : $11100110_2 + 1 = 11100111_2$

Une seule représentation pour 0 : 00000000_2

Nombres représentés sur 8 bits : $-128..+127$

Le bit de poids fort indique le s

- ▶ 0 = positif
- ▶ 1 = négatif

Un nombre négatif s'obtient en valeur absolue (et inversement)

Exemple : représentation de -2

- ▶ $+25_{10} = 00011001_2$
- ▶ Complément à 1 de $+25_{10}$
- ▶ Ajout de 1 : $11100110_2 + 1$

Une seule représentation pour 0

Nombres représentés sur 8 bits

chaîne de bits	non signé	complément à 2
1111	15	-1
1110	14	-2
1101	13	-3
1100	12	-4
1011	11	-5
1010	10	-6
1001	9	-7
1000	8	-8
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0

de sa

Représentation des nombres négatifs par ajout d'un biais les rendant positifs.

Le biais est ajouté aussi aux nombres positifs

Exemple : codage sur 8 bits avec un biais de 127

- ▶ -12_{10} est codé par $-12 + 127 = 115$ i.e. 01110011_2
- ▶ 30_{10} est codé par $30 + 127 = 157$ i.e. 10011101_2

Nombres représentés sur 8 bits avec biais de 127 : $-127..128$

Représentation des nombres négatifs rendant positifs.

Le biais est ajouté aussi aux nombres positifs.

Exemple : codage sur 8 bits avec un biais de 7

- ▶ -12_{10} est codé par 1100
- ▶ 30_{10} est codé par 1110

Nombres représentés sur 8 bits

chaîne de bits	non signé	codage avec biais de 7
1111	15	8
1110	14	7
1101	13	6
1100	12	5
1011	11	4
1010	10	3
1001	9	2
1000	8	1
0111	7	0
0110	6	-1
0101	5	-2
0100	4	-3
0011	3	-4
0010	2	-5
0001	1	-6
0000	0	-7

binaire	décimal	signe + magnitude	complément à 1	complément à 2	représentation biaisée
0000	0	0	0	0	-7
0001	1	1	1	1	-6
0010	2	2	2	2	-5
0011	3	3	3	3	-4
0100	4	4	4	4	-3
0101	5	5	5	5	-2
0110	6	6	6	6	-1
0111	7	7	7	7	0
1000	8	-0	-7	-8	1
1001	9	-1	-6	-7	2
1010	10	-2	-5	-6	3
1011	11	-3	-4	-5	4
1100	12	-4	-3	-4	5
1101	13	-5	-2	-3	6
1110	14	-6	-1	-2	7
1111	15	-7	-0	-1	8

(biais = 7)

binaire	décimal	signe + magnitude	complément à 1	complément à 2	représentation biaisée
0000	0	0	0	0	-7
0001	1	1	1	1	-6
0010	2	2	2	2	-5
0011	3	3	3	3	-4
0100	4	4	4	4	-3
0101	5	5	5	5	-2
0110	6	6	6	6	-1
0111	7	7	7	7	0
1000	8	-0	-7	-8	1
1001	9	-1	-6	-7	2
1010	10	-2	-5	-6	3
1011	11	-3	-4	-5	4
1100	12	-4	-3	-4	5
1101	13	-5	-2	-3	6
1110	14	-6	-1	-2	7
1111	15	-7	-0	-1	8

(biais = 7)

Quelle est la valeur de la chaîne de bits : 1010 ?

Plusieurs formats pour représenter des caractères (imprimables et de contrôle) sous forme binaire :

- ▶ EBCDIC (*Extended Binary-Coded Decimal Interchange Code*)
 - ▶ Représentation sur 8 bits (256 caractères possibles)
 - ▶ Utilisé autrefois sur les *mainframes* IBM
- ▶ ASCII (*American Standard Code for Information Interchange*)
 - ▶ Représentation sur 7 bits (pas d'accents)
 - ▶ ASCII étendu : sur 8 bits (plusieurs possibilités. Ex. : ISO 8859-1)
- ▶ Unicode : encodage sur 8–48 bits (ex. : UTF-8) pour représenter tous les caractères de toutes les langues

Chaîne de caractères : suite de caractères stockés consécutivement en mémoire (terminateurs?)

ISO-8859-1																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¸
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ø	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ



Standard :

- ▶ Code point
- ▶ Encodage

Encodage UTF-8 :

Domaine	O_1	O_2	O_3	O_4
[U+0000, U+007F]	0xxxxxxx			
[U+0080, U+07FF]	110xxxxx	10xxxxxx		
[U+0800, U+FFFF]	1110xxxx	10xxxxxx	10xxxxxx	
[U+10000, U+10FFFF]	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Code point de «é» : U+00E9

⇨ 0000 0000 1110 1001

⇨ 110000 11 1010 1001

⇨ 0xC3 0xa9

⇨ "Ã©" avec l'encodage 8859-1

Opérations $+$, $-$, \times , \div sur :

- ▶ Nombres non-signés
- ▶ Nombres signés *en complément à 2*

Le calcul se fait indépendamment de l'interprétation des chaînes de bits

L'addition se fait classiquement avec les règles :

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0 \quad \text{avec retenue de 1}$$

Exemples :

$$\begin{array}{r}
 00011010 \quad \mathbf{26} \\
 + 00001100 \quad \mathbf{12} \\
 \hline
 11 \\
 \hline
 00100110 \quad \mathbf{38}
 \end{array}$$

$$\begin{array}{r}
 00010011 \quad \mathbf{19} \\
 + 00111110 \quad \mathbf{62} \\
 \hline
 11111 \\
 \hline
 01010001 \quad \mathbf{81}
 \end{array}$$

Résultat sur 9 bits :

- ▶ Non signé : dépassement de capacité
- ▶ Signé : pas de signification

La soustraction suit les règles suivantes :

$$0 - 0 = 0$$

$$0 - 1 = 1 \quad \text{et on prend 1 à gauche}$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

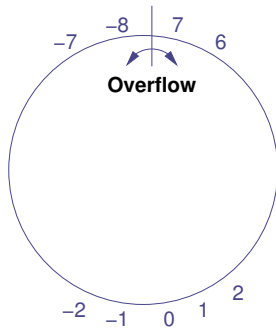
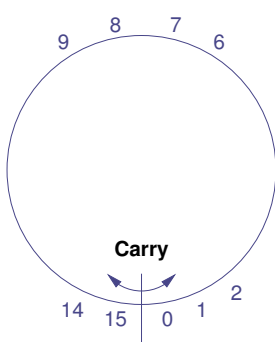
Exemples :

$$\begin{array}{r}
 001^100101 \quad \mathbf{37} \\
 -00010001 \quad \mathbf{17} \\
 \hline
 - \quad \mathbf{1} \\
 \hline
 00010100 \quad \mathbf{20}
 \end{array}$$

$$\begin{array}{r}
 001^110011 \quad \mathbf{51} \\
 -00010110 \quad \mathbf{22} \\
 \hline
 - \quad \mathbf{111} \\
 \hline
 00011101 \quad \mathbf{29}
 \end{array}$$

On peut aussi faire une addition avec le complément à 2 du deuxième opérande.

LS2M Carry vs. Overflow



Non signé

$$\begin{array}{r} 1110 \\ + 1011 \\ \hline 11001 \end{array}$$

14
11
25 > 15

Signé

$$\begin{array}{r} 1110 \\ + 1011 \\ \hline \cancel{11001} \end{array}$$

-2
-5
-7

Non signé

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$

7
1
8

Signé

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$

7
1
-8

Détection :
1 bit de plus en sortie

Détection :
*Opérandes de même signe
Résultat de signe différent*

La multiplication suit les règles suivantes :

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

Exemple :

$$\begin{array}{r}
 00101001 \quad \mathbf{41} \\
 \times 00000110 \quad \mathbf{6} \\
 \hline
 00000000 \\
 00101001 \\
 00101001 \\
 \hline
 0011110110 \quad \mathbf{246}
 \end{array}$$

On peut aussi faire des additions itérées

Division obtenue par itération de soustractions jusqu'à ce que le résultat de la soustraction soit inférieur au diviseur :

- ▶ Quotient = nombre de soustractions
- ▶ Reste = résultat de la dernière soustraction

Exemple : division de 7 par 3

$$\begin{array}{r}
 00000111 \quad 7 \\
 -00000011 \quad 3 \\
 \hline
 00000100 \quad 4 \\
 \hline
 00000001 \quad 1
 \end{array}$$

Diagram illustrating the binary division of 7 by 3. The dividend is 7 (00000111) and the divisor is 3 (00000011). The quotient is 2 (00000010) and the remainder is 1 (00000001). The diagram shows the iterative subtraction process. The first subtraction (7 - 3) results in 4 (00000100). The second subtraction (4 - 3) results in 1 (00000001). The quotient is 2, and the remainder is 1. The diagram uses blue and red colors to highlight the numbers and the subtraction steps.

Résultat : quotient = 2 et reste = 1

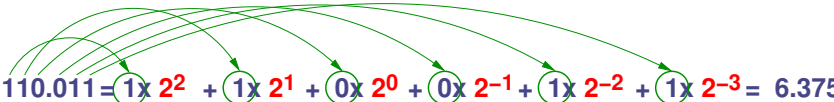
On peut aussi faire comme une division classique en décimal

- ▶ Infinité de nombres entiers
 - ➡ *Mais* représentation correcte dans un intervalle
- ▶ Infinité de nombres réels
 - ▶ Impossibilité de représentation correcte même d'un petit intervalle :

$$\forall a, b \in \mathbb{R} \quad \exists c \in \mathbb{R} \quad \text{t.q.} \quad a \leq c \leq b$$

⇒ Représentation d'un sous-ensemble de \mathbb{Q}

- Nombres en virgule fixe :

$$110.011 = \underbrace{1}_{\text{1}} \times 2^2 + \underbrace{1}_{\text{1}} \times 2^1 + \underbrace{0}_{\text{0}} \times 2^0 + \underbrace{0}_{\text{0}} \times 2^{-1} + \underbrace{1}_{\text{1}} \times 2^{-2} + \underbrace{1}_{\text{1}} \times 2^{-3} = 6.375$$


- Nombres en virgule flottante (notation scientifique) :

$$101010.10 = 1.0101010 \times 2^5$$

$$0.00100011 = 1.00011 \times 2^{-3}$$

⇒ Usage de l'arithmétique en virgule flottante majoritaire

Passage d'un nombre réel de base 10 vers base 2 en virgule fixe :

- ▶ Partie entière : comme pour les entiers
- ▶ Partie décimale : multiplications itérées par 2

Exemple : conversion de 14.375_{10} en base 2 ?

- ▶ $14_{10} = 1110_2$ (divisions itérées par 2)
- ▶ $0.375_{10} = ???_2$

$$0.375 \quad \times 2 \quad = 0 + 0.75$$

$$0.75 \quad \times 2 \quad = 1 + 0.5$$

$$0.5 \quad \times 2 \quad = 1 + 0.0$$

Résultat : $14.375_{10} = 1110.011_2$

1936 : machines Z de K. Zuse

60's–70's : Design des FPU = anarchie totale

→ Portabilité nulle

→ Peu de propriétés ($a = b \Leftrightarrow a - b = 0$)

1976 : création du i8087 par Intel (*best arithmetic*)

→ Rapport Kahan–Coonen–Stone

1985 : Rapport K-C-S devient la norme *IEEE 754*

Aujourd'hui : norme implantée de fait sur tous les ordinateurs (sauf certains Cray).

Nombre flottant x en binaire :

- ▶ un bit de signe s
- ▶ un exposant E
- ▶ un *signifiant* m

$$x = (-1)^s \times m \cdot 2^E$$

Représentations équivalentes :

- (a) $0.0000111010 \cdot 2^0$
- (b) $0.000000111010 \cdot 2^2$
- (c) $1.110100 \cdot 2^{-5}$

Taille de signifiant fixée \Rightarrow forme c plus précise

- ▶ Représentation *normalisée* (forme **c**) ;
- ▶ Toujours un 1 avant la virgule \Rightarrow pas codé (*hidden bit*)

$$m = 1.00101 \quad \longrightarrow \quad f = 00101$$
- ▶ exposants négatifs et positifs : codage par biais :

$$E \quad \longrightarrow \quad e = E + \text{biais}$$

Intérêt : comparaison lexicographique

[

single (1,8,23)

double (1,11,52)

ix87 reg. (1,15,64)
]

s	e	f
1	10101010101	101010101010...10100101
<i>Signe</i>	<i>Exposant biaisé</i>	<i>Partie fractionnaire</i>

Exemple : format tiny sur 5 bits (1, 2, 2) de biais 1 :

Nombres positifs représentables :

$$0\ 00\ 00 \rightsquigarrow 1.00 \times 2^{-1} = 0.5$$

$$0\ 00\ 01 \rightsquigarrow 1.01 \times 2^{-1} = 0.625$$

$$0\ 00\ 10 \rightsquigarrow 1.10 \times 2^{-1} = 0.75$$

$$0\ 00\ 11 \rightsquigarrow 1.11 \times 2^{-1} = 0.875$$

$$0\ 01\ 00 \rightsquigarrow 1.00 \times 2^0 = 1$$

$$0\ 01\ 01 \rightsquigarrow 1.01 \times 2^0 = 1.25$$

$$0\ 01\ 10 \rightsquigarrow 1.10 \times 2^0 = 1.5$$

$$0\ 01\ 11 \rightsquigarrow 1.11 \times 2^0 = 1.75$$

$$0\ 10\ 00 \rightsquigarrow 1.00 \times 2^1 = 2$$

$$0\ 10\ 01 \rightsquigarrow 1.01 \times 2^1 = 2.5$$

$$0\ 10\ 10 \rightsquigarrow 1.10 \times 2^1 = 3$$

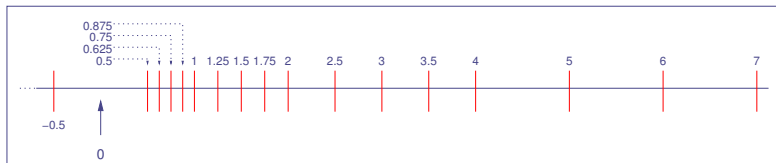
$$0\ 10\ 11 \rightsquigarrow 1.11 \times 2^1 = 3.5$$

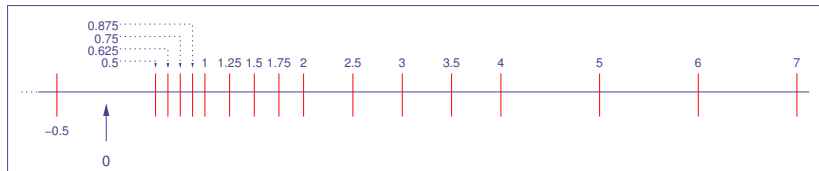
$$0\ 11\ 00 \rightsquigarrow 1.00 \times 2^2 = 4$$

$$0\ 11\ 01 \rightsquigarrow 1.01 \times 2^2 = 5$$

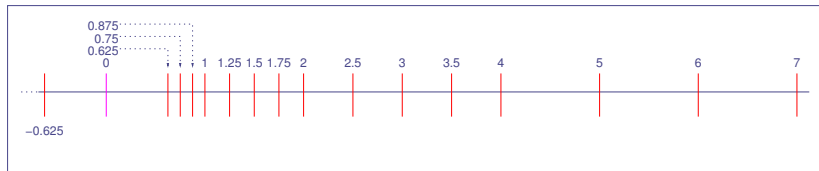
$$0\ 11\ 10 \rightsquigarrow 1.10 \times 2^2 = 6$$

$$0\ 11\ 11 \rightsquigarrow 1.11 \times 2^2 = 7$$

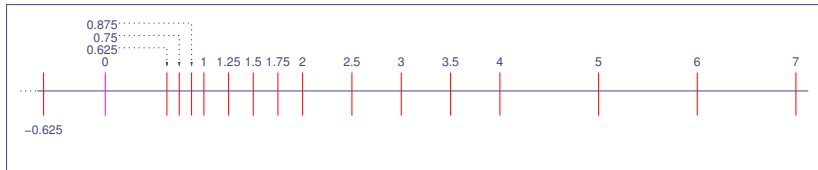




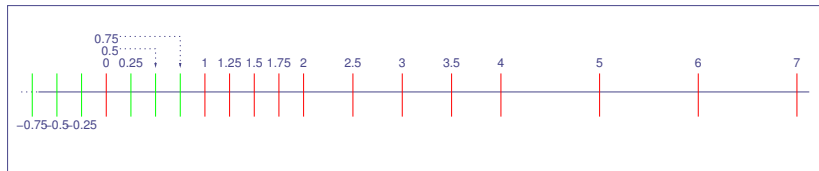
- ▶ Pas de codage pour 0



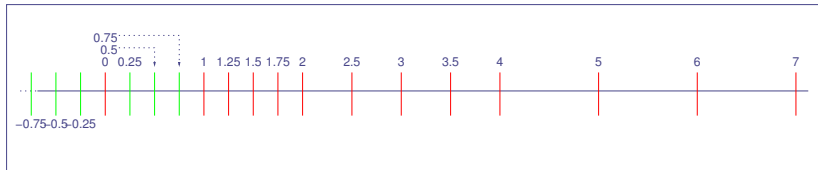
- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)



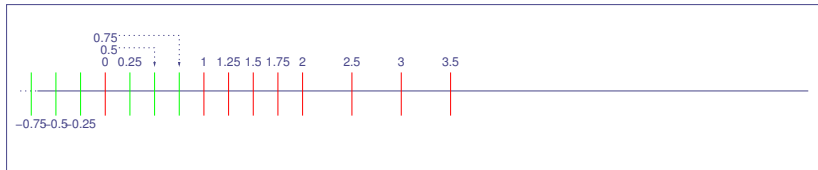
- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0



- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0
 - ▶ Réserver $e = 0$ pour les nombres *dénormalisés*
 - ➡ Plus de *hidden bit* à 1



- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0
 - ▶ Réserver $e = 0$ pour les nombres *dénormalisés*
 - ➡ Plus de *hidden bit* à 1
- ▶ Notions d'infinis mathématiques et de résultat indéfini :



- ▶ Pas de codage pour 0
 - ▶ Réserver 0 00 00 et 1 00 00 pour ± 0 (perte de ± 0.5)
- ▶ Grand trou autour de 0
 - ▶ Réserver $e = 0$ pour les nombres *dénormalisés*
 - ➡ Plus de *hidden bit* à 1
- ▶ Notions d'infinis mathématiques et de résultat indéfini :
 - ▶ Réserver $e = 3$

Interprétation des bits :

$$\begin{cases}
 e = 3, & f \neq 0 & : v = \mathbf{NaN} \\
 e = 3, & f = 0 & : v = (-1)^s \times \infty \\
 0 < e < 3 & & : v = (-1)^s \times (1.f) \cdot 2^{e-1} \\
 e = 0, & f \neq 0 & : v = (-1)^s \times (0.f) \cdot 2^0 \\
 e = 0, & f = 0 & : v = (-1)^s \times 0
 \end{cases}$$

00000	0.5	00000	0
00001	0.625	00001	0.25
00010	0.75	00010	0.5
00011	0.875	00011	0.75
00100	1	00100	1
00101	1.25	00101	1.25
00110	1.5	00110	1.5
00111	1.75	00111	1.75
01000	2	01000	2
01001	2.5	01001	2.5
01010	3	01010	3
01011	3.5	01011	3.5
01100	4	01100	$+\infty$
01101	5	01101	NaN
01110	6	01110	NaN
01111	7	01111	NaN

- ▶ *Not a Number* : cas indéfinis (*non ordonnés*)

▶ Infinis :
$$\begin{cases} 1/+\infty & = +0 \\ 1/-\infty & = -0 \\ 1/-0 & = -\infty \\ 1/0 & = +\infty \end{cases}$$

$x + y$	NaN	$-\infty$	0	$+\infty$
NaN	✓	✓	✓	✓
$-\infty$	✓			✓
0	✓			
$+\infty$	✓	✓		

$x - y$	NaN	$-\infty$	0	$+\infty$
NaN	✓	✓	✓	✓
$-\infty$	✓	✓		
0	✓			
$+\infty$	✓			✓

\sqrt{x}		$x \times y$	NaN	$-\infty$	0	$+\infty$
NaN	✓	NaN	✓	✓	✓	✓
$-\infty$	✓	$-\infty$	✓	✓	✓	
$x < 0$	✓	0	✓	✓		✓
0		$+\infty$	✓		✓	
$+\infty$						

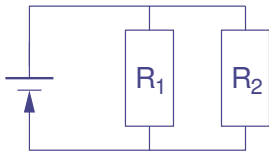
x/y	NaN	$-\infty$	0	$+\infty$
NaN	✓	✓	✓	✓
$-\infty$	✓	✓		✓
0	✓		✓	
$+\infty$	✓	✓		✓

Cas spéciaux pour les opérateurs non définis par la norme IEEE 754 :

x	$\exp(x)$	$\log(x)$	$\sin(x)$	$\cos(x)$	$\tan(x)$	$\operatorname{acos}(x)$	$\operatorname{asin}(x)$	$\operatorname{atan}(x)$
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$-\infty$	0.0	NaN	NaN	NaN	NaN			
$x < -745.13$	0.0							
$x < -1$						NaN	NaN	
$x < 0$		NaN						
0		$-\infty$						
$x > 1$						NaN	NaN	
$x > 709.78$	$+\infty$							
$+\infty$	$+\infty$	$+\infty$	NaN	NaN	NaN			

Source : *librairie fdlibm sur les doubles*

Calculer la résistance totale du circuit :



Formule :

$$T = \frac{1}{1/R_1 + 1/R_2}$$

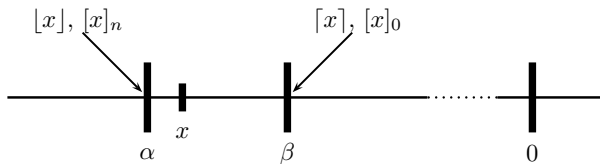
Cas où l'une des résistances est nulle ?

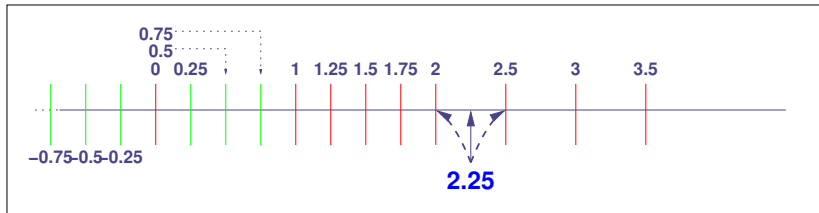
⇒ Résistance totale nulle

- ▶ Ensemble des flottants \mathbb{F} non clos pour les opérations de base
- ▶ conversion exacte décimal/binaire pas toujours possible

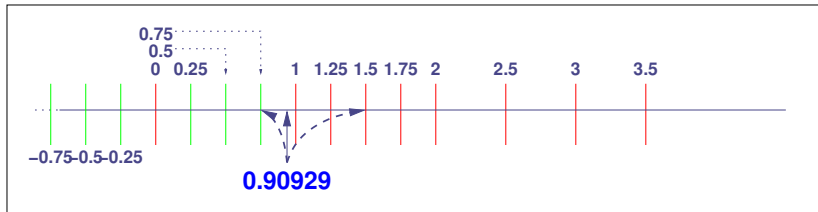
⇒ Fonctions d'arrondi :

$$\left\{ \begin{array}{l} \lfloor x \rfloor \quad \text{arrondi vers } -\infty \\ \lceil x \rceil \quad \text{arrondi vers } +\infty \\ [x]_0 \quad \text{arrondi vers } 0 \\ [x]_n \quad \text{arrondi au plus proche pair} \end{array} \right.$$





- ▶ Opérations $\{+, -, \times, /, \sqrt{\}$:
arrondi correct suivant le mode courant
 (erreur $< 1 \text{ ulp.}$);
 - ▶ Exemple : $2.0 + 0.25 = 2.25$ arrondi à 2 ou 2.5




- ▶ Opérations $\{+, -, \times, /, \sqrt{\}$:
arrondi correct suivant le mode courant
 (erreur < 1 ulp.);
 - ▶ Exemple : $2.0 + 0.25 = 2.25$ arrondi à 2 ou 2.5
- ▶ fonctions transcendantes : *aucune garantie*
 - ▶ Exemple : $\sin(2.0) = 0.90929$ arrondi vers 0.75 ou 1.5

- ▶ Addition possible si et seulement si les opérandes ont même exposant
- ▶ Exposants différents \Rightarrow décalage du nombre de *plus petit* exposant

Exemple :

$$\begin{array}{r}
 10.375 \\
 + 6.34375 \\
 \hline
 16.71875
 \end{array}
 \quad \xrightarrow{\text{dashed arrow}} \quad
 \begin{array}{r}
 1.0100110 \times 2^3 \\
 + 1.1001011 \times 2^2 \\
 \hline
 \end{array}
 \quad \xrightarrow{\text{dashed arrow}} \quad
 \begin{array}{r}
 1.0100110 \times 2^3 \\
 + 0.1100101 \times 2^3 \\
 \hline
 10.0001011 \times 2^3
 \end{array}$$



Attention : bit de garde

- ▶ Soustraction possible si et seulement si les opérandes ont même exposant
- ▶ Exposants différents \Rightarrow décalage du nombre de *plus petit* exposant

Exemple :

$$\begin{array}{r}
 16.75 \\
 - 15.9375 \\
 \hline
 0.8125
 \end{array}
 \quad \rightarrow \quad
 \begin{array}{r}
 1.0000110 \times 2^4 \\
 - 1.1111111 \times 2^3 \\
 \hline
 \end{array}
 \quad \rightarrow \quad
 \begin{array}{r}
 1.0000110 \times 2^4 \\
 - 0.1111111 \times 2^4 \\
 \hline
 0.0000111 \times 2^4 \\
 \downarrow \\
 0.875
 \end{array}$$

Diagram illustrating the subtraction of floating-point numbers. The first column shows the decimal subtraction of 16.75 and 15.9375, resulting in 0.8125. The second column shows the binary representation of the same numbers: 1.0000110×2^4 and 1.1111111×2^3 . The third column shows the result after aligning the exponents: 1.0000110×2^4 minus 0.1111111×2^4 , resulting in 0.0000111×2^4 , which is 0.875. A red dotted arrow points from the result 0.8125 to 0.875, with a large red 'X' over it, indicating that the result is incorrect due to the loss of precision (guard bit).

Attention : bit de garde

- Multiplication des significatifs et ajout des exposants

Exemple :

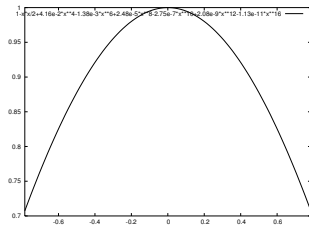
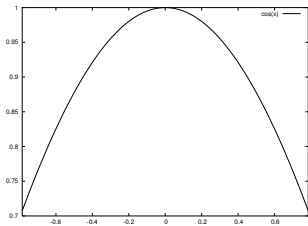
$$\begin{array}{r}
 10.375 \\
 \times 2.5 \\
 \hline
 25.9375
 \end{array}
 \quad \dashrightarrow \quad
 \begin{array}{r}
 1.0100110 \times 2^3 \\
 \times 1.0100000 \times 2^1 \\
 \hline
 10100110 \\
 10100110 \\
 \hline
 1.10011110000000 \times 2^4
 \end{array}$$
$$25.875 \quad \dashleftarrow \quad 1.1001111 \times 2^4$$

- ▶ Procédure plus compliquée
- ▶ Parfois implémentée par utilisation de la méthode de Newton

- ▶ Fonctions cos, sin, exp, ... :
 - ▶ Utilisation d'approximations polynomiales
 - ▶ Implémentation dans une librairie (e.g. en C)
 - ▶ Implémentation codée dans l'*unité arithmétique flottante*

Exemple : Pour $x \in [-\frac{\pi}{4}, \frac{\pi}{4}]$:

$$\cos x \approx 1 - \frac{x^2}{2} + 4.16 \cdot 10^{-2} x^4 - 1.38 \cdot 10^{-3} x^6 + 2.48 \cdot 10^{-5} x^8 - 2.75 \cdot 10^{-7} x^{10} + 2.08 \cdot 10^{-9} x^{12} - 1.13 \cdot 10^{-11} x^{16}$$



▶ *Arrondi.*
$$\left[\begin{array}{l} x \rightarrow \tilde{x} = x(1 + \delta), \quad \delta \leq \varepsilon/2 \\ x + y \rightarrow x \oplus y = (x + y)(1 + \delta) \end{array} \right.$$

▶ *Absorption.*

Addition opérandes de magnitudes différentes :

$$1.345 \cdot 10^5 + 1.45 \cdot 10^1 = 1.345 \cdot 10^5$$

Alignement \Rightarrow chiffres significatifs de $1.45 \cdot 10^1$ éliminés.

▶ *Cancellation bénigne.* Voir arrondis.

▶ *Cancellation catastrophique.*

Soustraction opérandes entachés d'erreurs :

$$a = 1.22, b = 3.34, c = 2.28, b^2 - 4ac = 11.2 - 11.1 = .1 \neq .0292$$

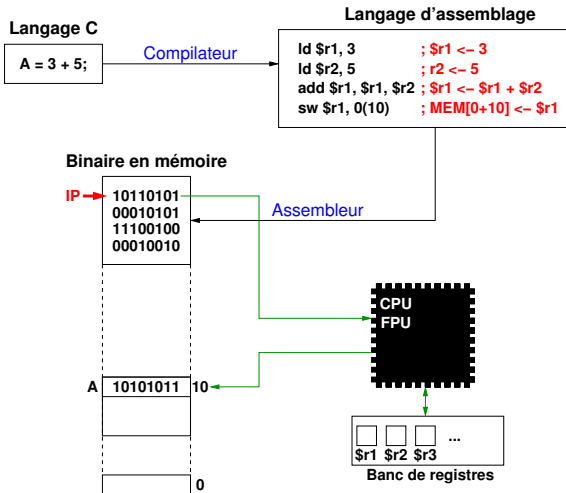
$$x \oplus y = y \oplus x \quad \text{(commutativité de l'addition)}$$

$$x \otimes y = y \otimes x \quad \text{(commutativité de la multiplication)}$$

$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z \quad \text{(non associativité)}$$

$$x \otimes (y \oplus z) \neq x \otimes y \oplus x \otimes z \quad \text{(non distributivité)}$$

⇒ ordre des calculs pas indifférent. (e.g. $\sum_{i=1}^n x_i$)



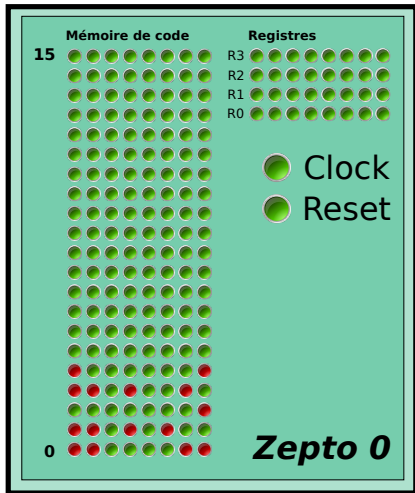
Deux types d'architecture :

	RISC <i>Reduced Instruction Set Computer</i>	CISC <i>Complex Instruction Set Computer</i>
Nb. d'instr.	Peu	Beaucoup
Taille d'instr.	Fixe	Variable
Arité des instr.	Fixe	Variable
Adressage	Peu	Beaucoup

- ▶ RISC (Sparc, MIPS) : instructions simples et rapides
 - ▶ Optimisations faciles
 - ▶ Programmes complexes longs
- ▶ CISC (ix86) : instructions plus ou moins complexes
 - ▶ Programmes plus petits
 - ▶ Optimisation plus difficiles

$A \leftarrow 3 + 5$?

Machine à pile	Machine à accumulateur	Machine à registres
PUSH X $top(pile) \leftarrow X$	LOAD X $acc \leftarrow X$	LOAD R_i, X $R_i \leftarrow X$
POP X $X \leftarrow top(pile)$	STORE X $X \leftarrow acc$	STORE R_i, X $X \leftarrow R_i$
ADD POP t_2 ; POP t_1 PUSH $t_1 + t_2$	ADD X $acc \leftarrow acc + X$	ADD R_i, R_j, R_k $R_i \leftarrow R_j + R_k$
push 3 push 5 add pop A	load 3 add 5 store A	load R1, 3 load R2, 5 add R3, R1, R2 store R3, A

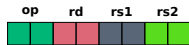


Zepto 0

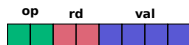
- 4 registres de 8 bits
- Mémoire de code de 16 octets
- 4 instructions : add, sub, mul, load

Format d'instructions

Instr	op
add	00
sub	01
mul	10
load	11



Format 1



Format 2

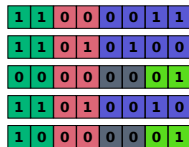
Exemple de programme

Calcul de $(3+4)*2$

```
load R0, 3      # R0 <- 3
load R1, 4      # R1 <- 4
add R0, R0, R1  # R0 <- R0 + R1
load R1, 2      # R1 <- 2
mul R0, R0, R1  # R0 <- R0 x R1
```

Résultat dans R0

Codage du programme



Circuit du Zepto-0