Compte rendu séance 1

Expérience :

- a = 0.1, b = 0.2, c = 0.3 :
  - float : renvoir true
  - double : renvoir false
- a = 1.1, b = 1.2, c = 2.3 :
  - float : renvoir false
  - double : renvoie false

Les cas 2, 3 et 4 illustrent l'imprécision du calcul à virgule.

Exercice 1 :

a) 0 ^ 1 = 0 car faux et vrai = faux

b) En fixant x = 0, puis x = 1, on trouve deux tables de vérités que l'on combine en une grâce à cette expression :

((¬x) ^ y ^ (¬z)) v (x ^ (¬(y ^ z)))

Exercice 2 :

a) $c_0 = (a_0 \wedge (\neg b_0)) \vee ((\neg a_0) \wedge b_0)$; c'est le ou exclusif

b) Le bit b_0 est obtenu comme précédemment pour le bit c_0 : $c_0 = a_0 \oplus b_0$

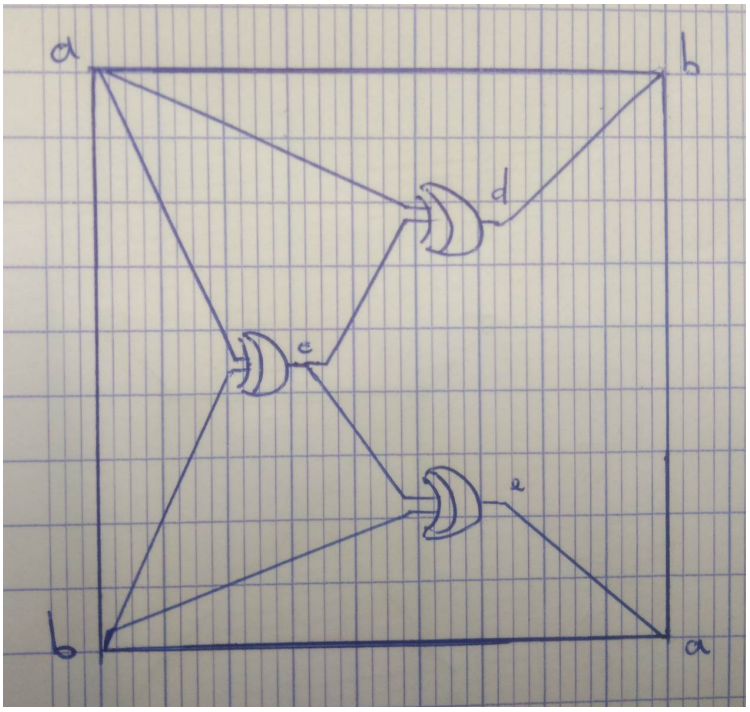
Le bit b_1 peut s'obtenir de deux façons : soit il n'y a pas de retenue sur les bits des unités et un seul bit parmi a_1 et b_1 vaut 1, soit il y a un bit de retenue et les bits a_1 et b_1 ont la même valeur
$d_1 = (a_1 \oplus b_1) \oplus (a_0 \wedge b_0)$

Le bit b_2 peut aussi s'obtenir de plusieurs façons. Une façon assez compacte est décrite dans la partie 1.1.2. et peut être écrite de la façon suivante
$d_2 = (a_1 \wedge b_1) \vee (a_1 \oplus b_1 \wedge a_0 \wedge b_0)$

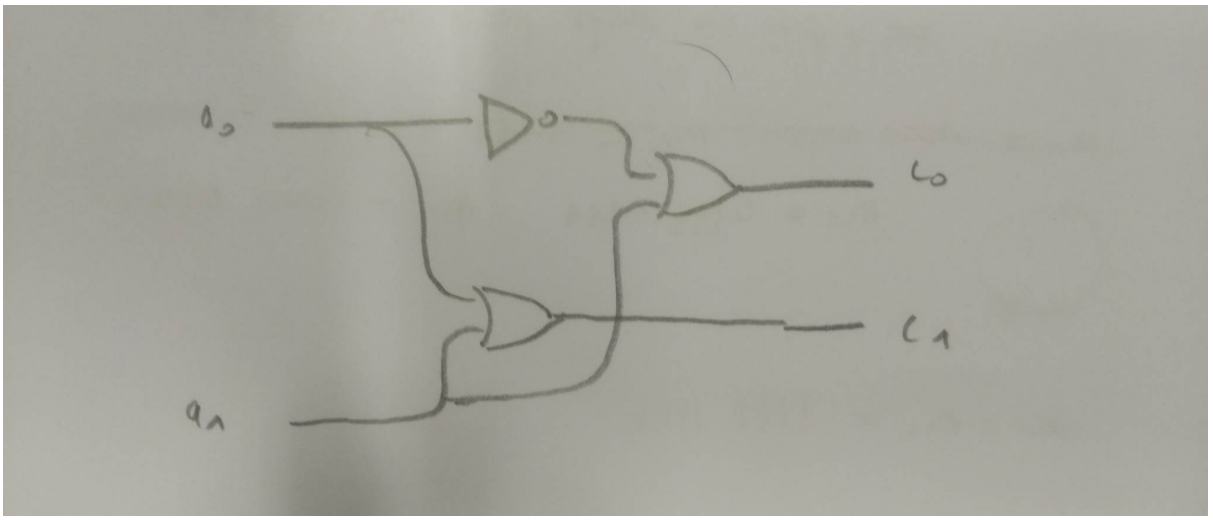
Exercice 3 :

a) On remarque que l'opérateur $\oplus$ est commutatif et associatif. On peut donc écrire :
$a \oplus c = (a \oplus a) \oplus b = 0 \oplus b = b$
Ainsi d = b et de la même façon e = a

b)



Exercice 4 :



Exercice 5 :

a) int : 32 bits - **short :** 16 bits - **char :** 8 bits

Code :

```c
int a;
short b;
char c;

printf("int : %ld, short : %ld, char : %ld\n", sizeof(a), sizeof(b), sizeof(c));
```

Output :
```
int : 4, short : 2, char : 1
```

b) Voir output et code ci-dessous
Nous avons illustré le principe de *dépassement de capacité*, qui se produit ici car nous tentons d'ajouter 1 à la plus grande valeur que l'on peut stocker pour chaque type.
Output :
```
Greatest unsigned int : 4294967295
Adding one : 0

Greatest unsigned short : 65535
Adding one : 0

Greatest char : 255
Adding one : 0
```
Code :
```c
#include<stdio.h>
#include <unistd.h>

int main () {
    // Exo 5
    unsigned int i = 0xffffffff;
    unsigned int i2 = i + 0x01;
    unsigned short s = 0xffff;
    unsigned short s2 = s+0x01;
    char c = 0x7f;
    char c2 = c + c + 0x02;

    printf("Greatest unsigned int : %u\n", i);
    printf("Adding one : %u\n\n", i2);

    printf("Greatest unsigned short : %u\n", s);
    printf("Adding one : %u\n\n", s2);

    printf("Greatest char : %u\n", c+c+0x01);
    printf("Adding one : %u\n", c2);

    return 0;
}
```

Exercice 6 :

0 : sur **4-bits :** 0000 ; sur **8-bits** : 00000000
1 : sur **4-bits :** 0001 ; sur **8-bits** : 00000001
-1 : sur **4-bits :** 1111 ; sur **8-bits** : 11111111
-2 : sur **4-bits :** 1110 ; sur **8-bits** : 11111110

Exercice 7 :

a) $m_1 = 0001$; $m_{-1} = 1001$
b) On a $m_0 = 1010$
c) m_0 code l'entier relatif -2 (en règle du bit de signe sur 4 bits)

Exercice 8 :

Sachant que les positifs sont codés de **0** à $\mathbf{2^{b-1}-1}$ pour un mot b-bits, le bit de poids fort vaut 0. En passant au complément à deux, le bit de poids fort vaut 1. Ou, en faisant $x \rightarrow x + 2^b$ pour coder les négatifs, on ajoute en fait le bit de poids le plus fort.

Exercice 9 :

Les deux programmes montrent que des cases mémoires contenant les mêmes données peuvent être interprétées de deux manières différentes.

Dans le premier, on compare directement les valeurs, dans le deuxième on compare à l'aide de pointeurs, mais le résultat est le même : avec la valeur *unsigned*, on retrouve bien 150, mais avec la valeur *signed*, on trouve 150 - 256 = 109.

Exercice 10 :

a) Considérons les mots 0x0000001 et 0x11111111.
Ces mots valent respectivement 1 et -1 en les interprétant comme entiers signés, et 1 et 255 en les interprétant comme des entiers non-signés.
Or lorsqu'on les additionne on obtient bien 0 dans le premier cas mais 0 au lieu de 256 dans le second : c'est un *dépassement de capacité* non signé, mais pas signé.

b) Considérons les mots 0x01111111 et 0x0000001.
Ces mots valent respectivement 127 et 1 en les interprétant comme entiers signés ou non signés.
Or lorsqu'on les additionne on obtient bien 128 dans le premier cas mais -128 dans le second : c'est un *dépassement de capacité* signé, mais pas non signé.

c) On considère désormais le mot 0x10000000.
Ce nombre vaut -128 dans le cas signé et 128 dans le cas non signé.
Lorsqu'on l'additionne avec lui-même, on obtient 0x00000000, c'est-à-dire 0.
Or sans dépassement de capacité on est censé obtenir -256 en signé et 256 en non signé.

Exercice 11 :

a) Soit k un entier naturel strictement supérieur à 1, $2^k/10 = 2^{k-1}/5$. Or un nombre pair n'est pas divisible par un nombre impair.

b) Ainsi, $2^l \times 0.1$ n'est pas entier. Donc d'après la propriété de 1.3.2 x n'est pas représentable sur k.l bits.

Exercice 12 :

On commence par multiplier 0.1 et 2^8 : on obtient 25.6 dont la partie entière vaut 25, qui s'écrit en binaire 0x00000000 00011001
Enfin on obtient 00000000.00011001