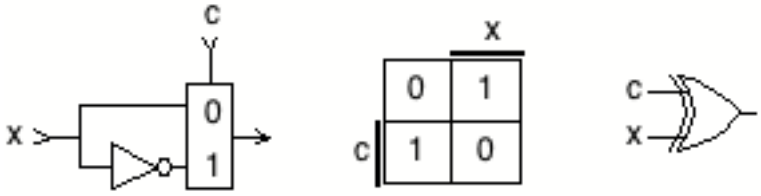
ES102/PC1 : énoncé et corrigé

1) To be or not to be… (~15')

1a) Soit deux valeurs binaires *c* et *x* et un multiplexeur 2 vers 1 laissant passer *x* si c=0 et *x'* si c=1. Dessiner le montage correspondant et sa table de Karnaugh *fonctionnelle*. Que reconnaît-on ?



C'est un OU exclusif, alias XOR.

1b) Quel usage de la porte reconnue cela suggère-t-il ?

Un OU exclusif peut servir d'*inverseur à volonté* (inverse ou n'inverse pas) sur une entrée servant de donnée, l'autre de commande. Cette dualité donnée-commande sera à la base de l'organisation des *chemins de données* du CM6, eux-mêmes fondement des processeurs.

1c) Compléter les équations suivantes : 0⊕x = ? 1⊕x = ? x'⊕y = ?

0⊕x = x et 1⊕x = x', comme mis en évidence ci-dessus.

Pour la dernière égalité, on pourrait raisonner sur la table de vérité et ses symétries mais une approche algébrique est pertinente également, sachant que le OU exclusif est un opérateur associatif (cf. CM2). On peut ainsi écrire : x'⊕y = (1⊕x)⊕y = 1⊕(x⊕y) = (x⊕y)'. C'est aussi égal à x⊕y', en faisant jouer la commutativité. Bref, le « rond » (alias la « bulle », qui représente l'inversion) de la porte XNOR peut être librement déplacé sur l'une quelconque des deux entrées. On « jouera » de nouveau avec les bulles au CM2 pour imager les lois de De Morgan et plus.

2) Full adder (~35')

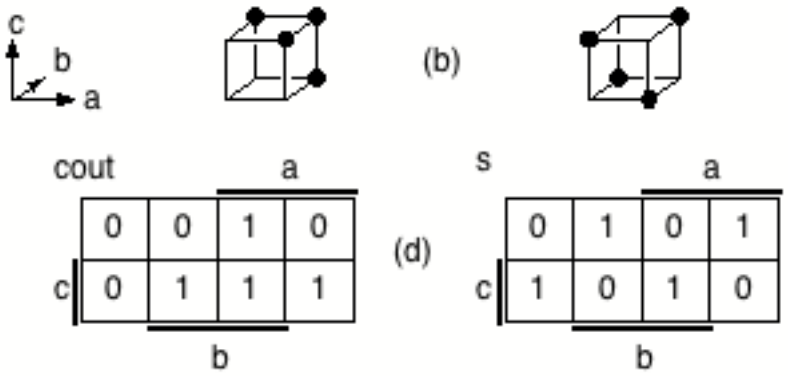
Un FA, alias *Full Adder* (additionneur binaire complet), prend en entrée 3 bits *a*, *b* et *c* et produit en sortie deux bits *s* et *cout*, tel que 2·cout **+** s = a **+** b **+** c (le signe **+** est mis en gras pour représenter l'addition, évitant la confusion avec le OU logique +). En d'autres termes, un FA exprime en base 2 la somme arithmétique ∑ des 3 valeurs binaires *a*, *b* et *c*.

2a) Dresser une table de vérité 1D de ∑, *cout* et *s* en fonction de *a*, *b* et *c*.

2b) Donner une représention géométrique 3D de *s* et *cout* en fonction de *a*, *b* et *c*.

a	b	c	∑	Cout	s
0	0	0	**0**	0	0
0	0	1	**1**	0	1
0	1	0	**1**	0	1
0	1	1	**2**	1	0
1	0	0	**1**	0	1
1	0	1	**2**	1	0
1	1	0	**2**	1	0
1	1	1	**3**	1	1

(a)



2c) Pourquoi *s* et *cout* sont-elles respectivement appelées fonctions *parité* et *majorité* de *a*, *b* et *c* ?

s = (a + b + c)%2 (où % désigne le modulo). En fait, s devrait plutôt s'appeler *imparité* puisqu'elle vaut 1 lorsqu'il y a un nombre impair d'entrées à 1. Attention à ce faux ami.

cout = 1 lorsqu'il y a une majorité d'entrées à 1 (c'est-à-dire au moins 2 sur 3) : c'est limpide.

2d) Représenter *s* et *cout* sous forme de tables de Karnaugh *fonctionnelles*, en fonction de *a*, *b* et *c*.

2e) À l'aide des deux représentations précédentes, exprimer *s* et *cout* en fonction de *a*, *b* et *c*, sous forme de formule booléenne disjonctive aussi concise que possible.

En regroupant les 1 par arêtes sur la représentation 3D, il vient : cout = (a·b) + (a·c) + (b·c). Mais la précédence de . sur + rend les parenthèses inutiles. En outre, · est généralement omis, d'où : cout = ab + ac + bc. C'est la *Forme Disjonctive Minimale* (FDM), forme ∑∏ la plus concise. Les mêmes regroupements peuvent être facilement réalisés sur la table de Karnaugh fonctionnelle : c'est sa raison d'être et on l'utilisera généralement dans ce but désormais.

Pressentant la distributivité de + sur · (cf. CM2), on peut en outre faire une factorisation, telle que : cout = ab + c(a+b). Il s'agit d'une formule plus concise encore que la FDM, mais d'un type plus complexe : ∑∏∑.

Quant à s, ses 1 sont tous isolés, interdisant tout regroupement. Sa forme disjonctive la plus concise est donc : s = a'b'c + a'bc' + ab'c' + abc. Au CM2, cette forme lourde car « éclatée » sera nommée *Forme Normale Disjonctive* (FND).

2f) Repartant de la forme ∑∏∑ de *cout* évoquée en marge de la réponse précédente, réétablir la table de Karnaugh fonctionnelle lui correspondant (et constater comme cela est simple).

La forme ∑∏∑ en question est : ab + c(a+b). Soit A (resp. B, C) le sous-ensemble de la table où *a*=1 (resp. *b*, *c*). Au bord de la table, le trait surmonté de la variable *a* indique clairement A : il s'agit de la moitié droite de la table. De même, B et C sont clairement indiqués. Il suffit alors de réaliser que ab + c(a+b) vaut 1 à l'intérieur de (A∩B) ∪ [C∩(A∪B)], et 0 ailleurs…

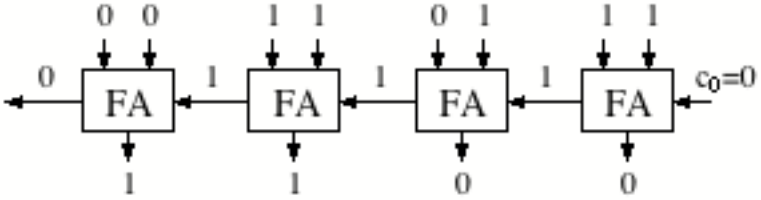
2g) Montrer que, en fait, s = a⊕b⊕c.

Prouver cette égalité est trivial en comparant les tables de vérité. Mais on peut aussi remarquer que c=0 ⇒ s=a⊕b et c=1 ⇒ s=(a⊕b)'. L'égalité découle alors de l'exercice 1.

2h) Traduire directement les formules précédentes en une implantation.

Trivial… Juste pour se forcer à utiliser les symboles des portes.

2i) Simuler l'addition des entiers 5 et 7 sur un additionneur à retenue propagée comportant 4 FA.



En outre, on remarque que chaque FA ne peut vraiment réaliser son calcul que lorsque son voisin de droite a fini le sien, d'où l'expression de « propagation des retenues ».

2j) *Prérequis pour l'exercice suivant*. En tant que somme arithmétique modulo 2, la fonction *parité* se généralise à *n* variables. Montrer qu'elle est égale au OU exclusif entre ses *n* variables.

Nommons P_n la fonction parité des *n* variables x_1 à x_n. L'égalité est vérifiée pour P_2 et P_3 (cf. question 2g) Elle se généralise par récurrence. On réalise que, en passant de *n* à *n*+1 variables, la parité est respectivement inchangée ou complémentée selon que la variable supplémentaire vaut 0 ou 1. Donc $P_{n+1} = P_n \oplus x_{n+1}$, d'après l'exercice 1. CQFD.

3) Codes de Gray (~40')

Les 3 premières questions sont à traiter rapidement, au profit des 4 dernières qui portent sur les concepts de complexité.

Le système de numération binaire classique sur *n* bits est une bijection de l'intervalle entier $[\![0, 2^n-1]\!]$ vers B^n (où B = {0, 1}) Une telle bijection est appelée *code* sur *n* bits. Mais sa valeur pour un entier particulier est aussi appelée *code* : ainsi, on dit que le code binaire classique sur 4 bits de l'entier 3 est 0011.

3a) Avec le code classique sur n=32 bits, quel est le nombre maximal de bits qui changent du code d'un entier *k* au code de son successeur *k*+1 ? Et en moyenne ?

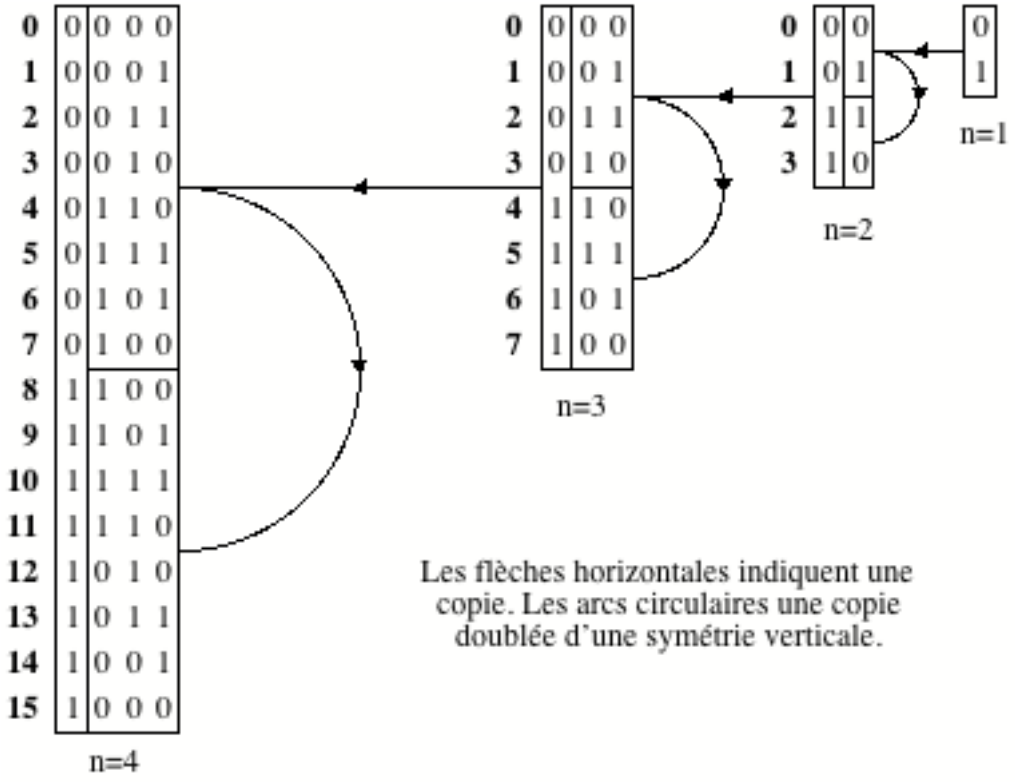
Avec n=3 bits, la pire situation se produit entre $(3)_{10}=(011)_2$ et $(4)_{10}=(100)_2$., c'est-à-dire au milieu de l'intervalle entier représenté : chaque bit change de valeur à cette occasion. Pour n=32 bits, le phénomène équivalent a lieu entre $k = 2^{31}-1 = (01\cdots1)_2$ et $k+1 = 2^{31} = (10\cdots0)_2$.
Par ailleurs, il est souvent pertinent de regarder $\{0, 1, \cdots, 2^{32}-1\}$ comme $\mathbb{Z}/2^{32}\mathbb{Z}$, l'ensemble des entiers modulo 2^{32}. Alors, tous les bits changent aussi lorsqu'on passe de $k = 2^{32}-1 = (1\cdots1)_2$ à $k+1 = 0 = (0\cdots0)_2$.

En moyenne, le bit de poids faible change dans tous les cas, le suivant dans un cas sur 2, le suivant dans un cas sur 4, etc. Donc il y a presque 2 bits qui changent au total en moyenne.

Exigeant le chargement de capacités, les changements de bits consomment courant et énergie. Il peut être intéressant d'en réduire le nombre dans certaines situations. Il existe justement un code dit *de Gray*, tel qu'un seul bit change du code de chaque entier à celui de son successeur.

3b) Comptant à partir de l'entier 0, représenté par 0···0 (chaîne de *n* '0'), le code de Gray sur *n* bits se construit en complémentant toujours le bit situé le plus à droite possible, sans jamais revenir sur un code (*n*-uplet de bits) déjà exploité. Le construire à la main pour *n*=2, puis 3, puis 4.

Voir la figure ci-dessous, qui commence par la droite. Les règles de construction induisent des propriétés de symétrie qui aboutissent à une structure originale mais simple à générer.



Comment passer du code classique d'un entier k à son code de Gray, et réciproquement ? Ce problème de codage/décodage (*codec*) n'a rien d'évident a priori. Notons b_i le bit de poids 2^i du code binaire classique de k, et g_i celui occupant la même position dans son code de Gray. On considèrera le cas n=4 ci-dessous.

3c) Exprimer algébriquement le fait qu'un seul bit change entre deux codes de Gray d'entiers successifs et, sur cette base, observer une relation entre b_0 et les g_i.

Le changement d'un (et un seul) bit provoque un changement de parité des g_i quand on passe de k à k+1. Cette parité alterne donc entre 0 et 1 quand on parcourt les entiers k par incrément de 1. Or il en est de même pour b_0 et il y a égalité en k=0. Donc $g_3 \oplus g_2 \oplus g_1 \oplus g_0 = b_0$.

3d) Implanter cette relation à l'aide de portes XOR à 2 entrées (OU exclusif), en choisissant la solution la plus rapide. Chaque porte XOR présente un même temps de réponse (alias délai) τ_{XOR}.

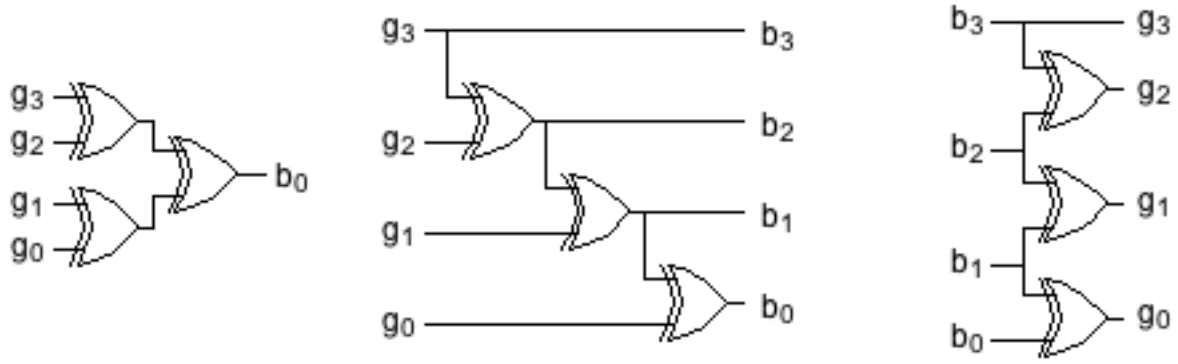
L'implantation la plus rapide consiste en un arbre binaire équilibré à 2 étages de portes XOR, correspondant au parenthésage suivant : $(g_3 \oplus g_2) \oplus (g_1 \oplus g_0)$. Elle est montrée plus bas à gauche. Le délai est de $2 \cdot \tau_{XOR}$.

3e) Pour n plus grand, comment se comporterait le délai d'obtention de b_0 ?

Augmentons n progressivement : chaque fois que n franchit une puissance de 2, il faut ajouter un étage supplémentaire à l'organisation en arbre ci-dessus, ce qui accroît le délai de τ_{XOR}. Donc le délai total T se comporte comme $\log_2(n) \cdot \tau_{XOR}$, à l'arrondi entier près. On dit que le montage fonctionne en *temps logarithmique* (sous-entendu par rapport à n) et l'on écrit $T=\Theta(\log(n))$ suivant la notation de Landau (cf. article Wikipedia *comparaison asymptotique*). Ceci caractérise sa *complexité en temps*.

3f) Exprimer les autres b_i en fonction des g_i. *Indice : examiner la parité des g_i hors g_0.* Implanter avec des portes XOR le décodage Gray vers classique, de façon aussi compacte que possible.

On remarque que, g_0 mis à part, les autres g_i changent de parité avec une fréquence de 2, comme b_1 ! En fait, il y a égalité : $b_1 = g_3 \oplus g_2 \oplus g_1$. De même, g_1 et g_0 mis à part, on observe que $b_2 = g_3 \oplus g_2$. Enfin, $b_3 = g_3$. Du coup, parenthéser ainsi $[\ (g_3 \oplus g_2) \oplus g_1\ ] \oplus g_0$ aboutit au montage ci-dessous au centre qui fournit l'ensemble des b_i à coût matériel minimal en nombre de portes à 2 entrées (impossible effectivement de faire mieux que 3 avec 4 variables). Mais le délai est désormais de $3 \cdot \tau_{XOR}$ (où 3=n-1). Ce montage, qui réalise le décodage complet, fonctionne en *temps linéaire* : $T=\Theta(n)$. Il serait donc bien plus lent que le précédent pour de grandes valeurs de n. On pourrait l'accélérer, mais il faudrait mobiliser plus de portes XOR (une de plus ici pour n=4). Cet exemple illustre une règle générale : compacité et rapidité sont *in fine* des exigences antagonistes. Le concepteur privilégiera l'une ou l'autre selon l'application visée.

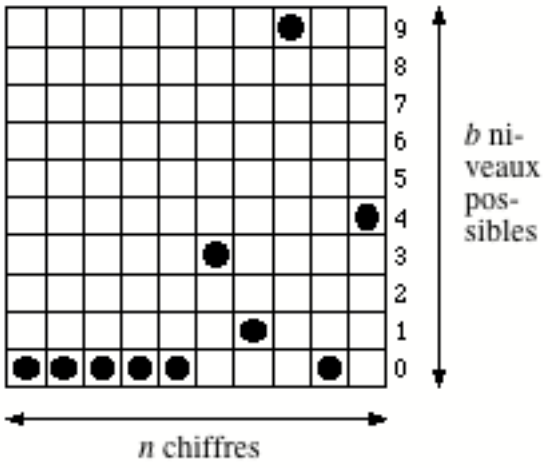


3g) Calculer $b_1 \oplus b_0$ en fonction des g_i. En déduire les relations de codage, les implanter et qualifier les complexités en temps et en espace obtenues.

$b_0 \oplus b_1 = g_3 \oplus g_2 \oplus g_1 \oplus g_3 \oplus g_2 \oplus g_1 \oplus g_0$. Par associativité et commutativité du OU exclusif, cela se réécrit $g_3 \oplus g_3 \oplus g_2 \oplus g_2 \oplus g_1 \oplus g_1 \oplus g_0$. Or $\forall x,\ x \oplus x = 0$. Finalement, $b_0 \oplus b_1 = g_0$. De même, il vient $b_1 \oplus b_2 = g_1$ et $b_2 \oplus b_3 = g_2$. Enfin, $g_3 = b_3$. Le codage binaire vers Gray s'implante donc très simplement, comme montré ci-dessus à droite. Ce montage se généralise de façon évidente à *n* variables. Il fonctionne en *temps constant* (par rapport à *n*) : $T = 1 \cdot \tau_{XOR} = \Theta(1)$. Comportant n-1 portes (à 2 entrées), sa complexité en espace (alias surface) est quant à elle *linéaire* : $A = \Theta(n)$.

4) Base optimale pour représenter les nombres (~30')

Considérons un moyen ancestral de représentation des nombres, constitué d'une tablette comme celle ci-contre. Chaque colonne comporte *b* cases, représentant de bas en haut les chiffres de 0 à *b*-1, et l'une de ces cases contient un caillou (*calculus* en latin…) qui indique le chiffre représenté. Cette tablette permet donc de manipuler des nombres de *n* chiffres en base *b*. Ci-contre, c'est donc l'entier 31904 qui est représenté, en base 10.



On mesure le niveau de performance d'une tablette par sa *portée P*, c'est-à-dire le nombre P(b, n) d'entiers qu'elle peut représenter. Par ailleurs, on mesure son *coût C* par son nombre C(b, n) de cases.

4a) Exprimer les fonctions *C* et *P* en fonction de *b* et *n*. Comparer le cas de la figure ci-dessus, où *b*=10 et *n*=10, avec le cas *b*=5 et *n*=20. Pour mémoire, $\log_{10}(5) \approx 0{,}7$.

$P(b,n) = b^n$ et $C(b,n) = b \cdot n$.
$P(10,10) = 10^{10}$, $C(10,10) = 100 = C(5,20)$, $P(5,20) = 5^{20} \approx 10^{0,7 \cdot 20} = 10^{14} >> 10^{10}$.
A coût identique de 100, la base 5 apparaît largement préférable à la base 10.
Mais peut-être est-il possible de faire encore mieux ? C'est l'objet de la question suivante.

4b) Comment choisir la base *b* pour minimiser le coût *C* à portée *P* donnée ? Pour le savoir, exprimer *C* en fonction de *P* et *b*, en faisant disparaître *n*. Finalement, que choisir ?

$P=b^n \Rightarrow n = \log(P)/\log(b)$. Comme $C=n \cdot b$, on obtient $C = b/\log(b) \cdot \log(P)$. Cette équation a un sens quelle que soit la base de la fonction logarithme considérée. Retenons le logarithme népérien, car le plus simple à dériver. Alors l'équation ci-dessus signifie simplement que si l'on souhaite une portée *e* fois plus grande, il faudra b/ln(b) cases en plus. Il s'avère aussi que la minimisation du coût est indépendante de P. Cette propriété remarquable montre l'existence d'une base idéale quelle que soit la portée visée. Cela n'avait rien d'évident a priori. La dérivée par rapport à *b* de b/ln(b) est une fraction dont le numérateur vaut ln(b)-1. Donc *la base idéale est le nombre e*. Malheureusement, ce n'est pas un entier : il faut choisir entre les deux entiers l'encadrant : 2 ou 3. La calculatrice fournit les valeurs suivantes : $2/\ln(2) \approx 2{,}89$; $e/\ln(e) \approx 2{,}72$; $3/\ln(3) \approx 2{,}73$. Le choix 3 apparaît significativement meilleur que le choix 2, et très proche de la valeur idéale. Les meilleures tablettes sont donc de hauteur 3 !

4c) Selon ce qui précède, la base 2 ne serait pas la meilleure. Mais le raisonnement mené ne vaut que pour un modèle de coût bilinéaire $C(b,n)=b \cdot n$. Une version électronique de notre tablette comporterait typiquement *n* dispositifs présentant chacun *b* états possibles. Si la proportionnalité de *C* par rapport à *n* est plausible, elle l'est moins par rapport à *b*. Considérons donc un modèle

de coût plus général C(b,n)= n·f(b), où *f* est une fonction inconnue. Pour quelle propriété de *f* la base 2 est-elle préférable à la base 3 ?

Suivant la même démarche que précédemment, on obtient C = f(b)/ln(b) · ln(P). La base 2 sera alors préférable à la base 3 si f(3)/ln(3) > f(2)/ln(2), c'est-à-dire si f(3) > 1,6 · f(2). En d'autres termes, il suffit que le surcoût relatif lié à l'exploitation de 3 états au lieu de 2 excède 60% pour que la base 2 soit préférable à la base 3.

En pratique, cette marge de 60% paraît faible. Par exemple, si un dispositif à 3 états (0, 1, 2) repose sur une grandeur physique scalaire (tension, réflectivité, …), il faudra sans doute un premier mécanisme pour distinguer l'état 0 de l'état 1 et un deuxième pour distinguer l'état 1 de l'état 2 : de quoi occasionner un surcoût important par rapport à un dispositif à 2 états...