ES102/PC2 : énoncé et corrigé

1) Soustraction entière et entiers négatifs (~20')

1a) Calculer 8-6 sur 5 bits, d'abord directement, puis en recourant au complément.

Dans les calculs ci-dessous, les retenues produites sont montrées en italique : pour la soustraction (1), ce sont des emprunts, placés en bas ; pour les additions (2) à (4), ce sont des reports, placés en haut. La soustraction directe (1) n'appelle pas d'autres commentaires. Le CM2 a montré qu'on peut calculer 8-6 comme 8+6'+1, où 6' est le *complément à 1* (où chaque bit est complémenté) de 6. C'est l'objet de l'addition (2) ci-dessous, où le +1 a été intégré à la ligne des retenues (il s'agit de c_0=1) mais pas en italique puisque c'est une entrée. On obtient bien le même résultat. La retenue sortante à 1 (en gras), de poids 2^5 constitue un débordement normal de ce calcul modulo 2^5 (cf. CM2).

```
(1)          (2)          (3)        (4)

 01000       11  11
-00110        01000          11        11
  11         +11001       +11011    +00011
=00010       =00010        11100     00100
```

1b) Quel est le code de -4 en tant qu'entier signé sur 5 bits ? Le vérifier avec les poids.

Chez les entiers signés sur 5 bits, l'entier -4 se voit attribuer le code de l'entier non signé 2^5-4, soit 28, d'où 11100. Mais, électroniquement, c'est directement au niveau binaire qu'il faut travailler et *la technique idoine pour passer du code d'un entier à celui de son opposé est de calculer son complément à 2*. Ici, c'est le résultat de l'addition 4'+1. Réalisée en (3) ci-dessus, elle fournit bien le même résultat. Cela peut aussi être vérifié par les poids, sachant que celui du MSB est ici de -16. On a bien : -16+8+4 = -4.

1c) Retrouver le code de l'entier signé +4 en calculant celui de -(-4).

Il suffit de réappliquer le complément à 2 au code juste obtenu, comme fait ci-dessus en (4).

1d) Trouver à la main le code de l'entier 235 sur 8 bits. Puis l'écrire en hexadécimal.

Le CM2 a présenté une méthode systématique de division itérative par 2 pour obtenir la décomposition en base 2 de tout entier décimal. Mais on remarque ici que 235 est proche de 255 = 2^8-1 = $(1111\ 1111)_2$. En fait, 235=255-20=255-16-4. On peut en déduire directement que le code de 235 est $(1110\ 1011)_2$. Il revient au même de dire que 235 est le complément à 1 de 16+4. En hexadécimal, cela donne $(EB)_{16}$.

2) Représentation flottante (~20')

2a) Calculer la décomposition du nombre réel 1/6 en puissances négatives de 2.

En base 10, le réel 1/6 s'écrit 0,166··· : la première puissance négative de 10 (c'est-à-dire 10^{-1}) reçoit le chiffre 1, et toutes les autres le chiffre 6. Par ailleurs, la notation scientifique décimale de 1/6 est 1,666···x10^{-1}. Pour représenter le fait que le chiffre 6 se répète à l'infini, écrivons-le une seule fois, mais souligné : 1/6=0,1$\underline{6}$=1,$\underline{6}$x10^{-1}.

Pour obtenir la décomposition en base 2 de 1/6, on effectue la division itérative de 1 par 6 en multipliant à chaque fois le reste par 2 (au lieu de 10 en version décimale) :

- étape 0* : 1 divisé par 6 → quotient 0, reste 1.

Le quotient obtenu est le facteur binaire associé au poids 2^0 (* d'où le numéro 0 donné à l'étape) : il s'agit du chiffre (bit) situé juste avant la virgule, qui est évidemment nul puisque 1/6<1. Les étapes suivantes vont fournir les puissances négatives. Les quotients obtenus, binaires, correspondent aux bits successifs après la virgule :

- étape 1 : 2 divisé par 6 → quotient 0, reste 2.
- étape 2 : 4 divisé par 6 → quotient 0, reste 4.
- étape 3 : 8 divisé par 6 → quotient 1, reste 2.
- étape 4 : 4 divisé par 6 → quotient 0, reste 4.
- ···

L'étape 4 étant identique à l'étape 2, le processus va dès lors se répéter, selon un cycle d'ordre 2. On a donc 1/6 = $(0,0\underline{01})_2$, où le bloc 01 se répète à l'infini.

2b) En déduire la représentation du réel 1/6 en format flottant 64 bits.

On réécrit 1/6 = $(0,0\underline{01})_2 \times 2^0$, puis on normalise la mantisse en la multipliant par 8, tout en compensant par une valeur –3 pour l'exposant : 1/6 = $(1,\underline{01})_2 \times 2^{-3}$. Cette normalisation revient en fait à écrire 1/6=4/3 $\times 2^{-3}$, où l'on a effectivement $1 \leq 4/3 < 2$.

La représentation flottante sur 64 bits de 1/6 est donc : 0 11111111101 01···01, où le bloc 01 est répété 26 fois, à droite. Cette représentation comporte successivement un bit nul pour le signe positif, 11 bits pour l'exposant -3 en tant qu'entier signé et enfin 52 bits qui sont ceux situés après la virgule dans l'expression binaire de la mantisse normalisée : 1,$\underline{01}$.

A l'attention des plus curieux, la représentation ci-dessus n'est pas exactement celle utilisée en pratique : il y a un décalage de +1023 sur l'exposant (cf. norme IEEE 754 double précision).

2c) Quel est l'ensemble des réels ayant la même représentation flottante sur 64 bits que 1/6 ?

Le représentant naturel de notre nombre flottant est le réel $r = 2^{-3} + 2^{-5} + 2^{-7} + \cdots + 2^{-55}$. Toutefois, tout réel supérieur à r, mais distant de moins de 2^{-55} (poids du dernier bit) aurait la même représentation selon la méthode de division itérative adoptée en question a). Donc, notre nombre flottant représente en réalité l'intervalle réel $[a , a+2^{-55}[$, qui contient notamment 1/6.

3) Réduction de délai par manipulations algébriques (~1h10')

On désigne par *commutation* (ou *transition*) le changement de valeur d'un signal binaire. Le délai d'un circuit combinatoire se mesure par le temps maximal qui s'écoule entre des commutations en entrée et les commutations qui en résultent en sortie. Parmi les multiples scénarios possibles, c'est le plus lent qui définit le temps de réponse du circuit.

L'esprit général de l'exercice est de mettre les possibilités de manipulation algébrique des formules booléennes au service de la réduction du délai combinatoire, pour calculer plus vite.

On dispose de la *bibliothèque* suivante de portes logiques. Le délai de chacune apparaît sur son symbole. Il s'agit d'un délai typique normalisé par rapport à celui de l'inverseur, qui est actuellement de quelques picosecondes (10^{-12} s) sur les technologies les plus avancées.

En fait, le délai réel dépendra aussi de ce qui se trouve en sortie de la porte, plus précisément de la capacité à charger. Mais ce point sera secondaire dans la suite de l'exercice.

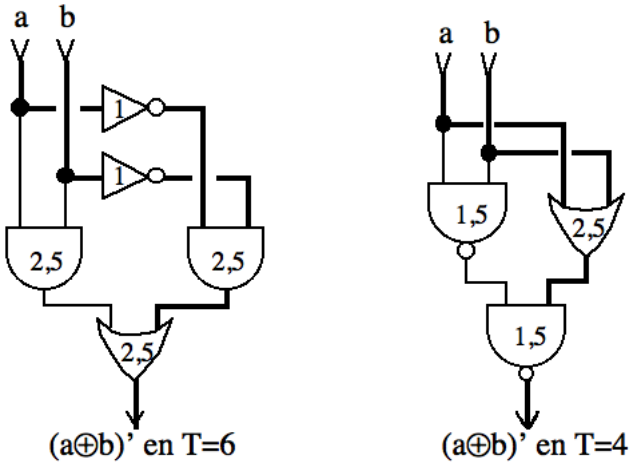
Les portes à 2 entrées les plus rapides de cette bibliothèque sont NAND et NOR (en réalité, la NOR est soit moins rapide soit moins compacte que la NAND, comme expliqué au CM4). Une porte AND présente le même délai qu'une porte NAND suivie d'un inverseur : c'est

effectivement ainsi qu'elle est réalisée en pratique. Sa présence au sein de la bibliothèque n'est donc en fait qu'une commodité. De même pour la porte OR. Au contraire, XOR et XNOR présentent une rapidité inégalable par combinaison des autres portes.

3a) Essayons pour la porte XNOR. Commencer par établir et implanter directement sa FDM. Puis procéder par « jeu de bulles » pour exploiter les portes les plus rapides, en particulier la porte NAND, et retranscrire sous forme algébrique la transformation ainsi réalisée. Enfin, déterminer le délai de chacune des 2 réalisations.

Les 1 du *non ou exclusif* étant isolés, sa FDM est égal à sa FND : $(a \oplus b)' = a \cdot b + a' \cdot b'$. L'implantation directe apparaît sur la figure ci-dessous à gauche. Comme vu en CM2, un « jeu de bulles » appliqué en commençant par la sortie permet de convertir ce circuit en celui de droite (où la porte OR est en fait la recombinaison d'une NAND avec les 2 inverseurs qui la précédaient, à délai identique). La transformation algébrique correspondante s'appuie sur les lois de De Morgan : $a \cdot b + a' \cdot b' = [\, a \cdot b + a' \cdot b' \,]'' = [\, (a \cdot b)' \cdot (a' \cdot b')' \,]' = [\, (a \cdot b)' \cdot (a+b) \,]'$.

Le « jeu » de bulles a permis d'abaisser fortement le délai, de 6 à 4. On ne voit pas comment on pourrait faire mieux. Le délai de 3 est annoncé pour la porte XNOR de la bibliothèque suggère qu'elle est conçue selon un autre principe, à découvrir dès la prochaine séance.



Plus généralement, comment accélérer un circuit combinatoire trop lent pour l'application visée ? Pour cela, il est utile d'identifier son « chemin critique ». Un chemin est ici une liste de connexions qui se suivent au sein du circuit, franchissant les portes dans le sens entrée-sortie. A chaque chemin est associé le total des délais des portes franchies. On considère alors l'ensemble de tous les chemins qui relient entrées et sorties du circuit considéré. Le chemin critique est celui qui présente le plus long délai. Il n'est pas forcément unique.

Remarque : sur un plan algorithmique, il s'agit de trouver le plus long chemin sur un graphe orienté acyclique (dont les nœuds sont les portes et les arêtes/arcs sont les connections). C'est un problème classique (très utile en ordonnancement, notamment), solutionnable en temps linéaire par rapport au nombre de nœuds et arêtes (contrairement à son équivalent sur graphe quelconque qui, lui, ne connaît pas de solution en temps polynomial).

3b) Identifier les chemins critiques des circuits proposés à la question précédente.

Les chemins critiques sont montrés en gras sur la figure ci-dessus. Il y en a 2 dans chaque cas. Ils constituent un support rigoureux à la détermination du délai.

La mise en série de n FAs (*Full Adder*) réalise un additionneur numérique, dit à retenues propagées. Constitué de quelques portes et destiné à être abouté en de multiples exemplaires, le FA est qualifié de « cellule ». Notons τ_{xy} le temps de réponse de la cellule FA entre son entrée x = a, b ou $c[in]$ (retenue entrante) et sa sortie y = s ou $c[out]$ (retenue sortante).

3c) Rappeler les équations de la cellule FA, liant ses sorties *s* et *cout* à ses entrées *a*, *b* et *cin*.

s = a⊕b⊕cin (fonction parité) et cout = a·b + a·cin + b·cin (fonction majorité)

3d) Que peut-on dire du chemin critique de l'additionneur à retenues propagées ?

La structure de cet additionneur numérique *n* bits a été présentée au CM1 et au CM2. Le chemin reliant c_0 à c_n (en passant par toutes les retenues intermédiaires) est appelé « chaîne de propagation des retenues ». L'additionneur en tire son nom. Sa longueur (temporelle) est $n \cdot \tau_{cc}$. Difficile de faire plus long, sauf éventuellement dans le premier FA (0) vers a_0 si $\tau_{ac} > \tau_{cc}$ ou vers b_0 si $\tau_{bc} > \tau_{cc}$, et dans le dernier FA (n-1) vers s_{n-1} si $\tau_{cs} > \tau_{cc}$. Mais les τ_{xy} étant a priori comparables, cet éventuel allongement sera insignifiant par rapport à $n \cdot \tau_{cc}$ pour les grandes valeurs classiques de *n*, telles que 32 ou 64. Le chemin critique est donc à très peu de choses près la chaîne de propagation des retenues.

Cela correspond bien au temps mis par l'additionneur pour faire son calcul. En effet, le dernier FA doit attendre que c_{n-1} soit valide pour produire c_n, et cela lui prend τ_{cc}. Mais, pour que c_{n-1} soit valide, il faut laisser τ_{cc} au FA précédent, une fois c_{n-2} valide. Et ainsi de suite jusqu'à c_0. Cette chaîne de dépendances entre retenues demande effectivement un délai total de $n \cdot \tau_{cc}$.

3e) Du coup, dans le cadre d'une implantation optimisée de la cellule FA, quelle est la caractéristique à minimiser en priorité ?

C'est τ_{cc} bien sûr, à savoir le délai pour produire une retenue sortante valide à partir d'une retenue entrante valide (c'est-à-dire qui vient de prendre une valeur stable).

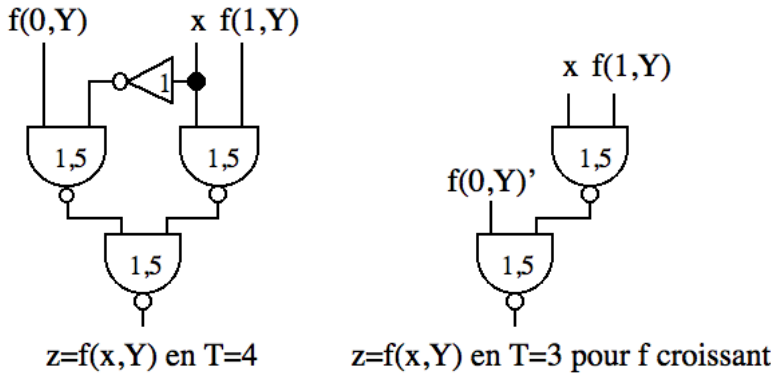
3f) Dans un FA, pour quelles valeurs de *a* et *b* la sortie *cout* dépend-elle effectivement de *cin* ? Ainsi, pour quelles paires d'entiers A et B l'additionneur présente-t-il vraiment le délai imposé par son chemin critique ?

Vu la fonction majorité, pour que *cout* dépende de *cin*, il faut avoir a+b=1 et ab=0, c'est-à-dire a≠b. Dans ce cas, cout=cin. Le FA doit alors reproduire son entrée *cin* sur sa sortie *cout*. Les paires d'entiers A et B concernées sont celles pour lesquelles $a_i \neq b_i$ quel que soit *i*, en d'autres termes où B est le complément à 1 de A. Exemples : $0 + 2^n - 1$ chez les entiers non signés ou encore 0 + (-1) chez les entiers signés.

Le besoin de minimiser le délai entre une entrée *x* et une sortie *z* particulières d'une cellule est général. Traitons le comme tel. Soit donc une fonction booléenne f telle que z=f(x,Y) où Y est la liste des autres entrées. Cherchons donc un montage où *x* intervienne au plus près de la sortie *z*.

3g) Exploiter la formule d'expansion de Boole pour aboutir à une première solution générique.

L'expansion de Boole de la fonction *f* par rapport à la variable *x* se traduit par l'égalité suivante : f(x,Y) = x'·f(0,Y) + x·f(1,Y). Comme on cherche à utiliser de préférence la porte NAND, on réécrit cette égalité sous la forme suivante : f(x,Y) = [(x'·f(0,Y))' . (x·f(1,Y))']'. Cette égalité correspond au montage de gauche de la figure ci-dessous, caractérisé par un délai T=4 entre *x* et *z*.



3h) Il est possible de faire mieux si f(x,Y) est croissante par rapport à *x* : ∀Y, f(0,Y) ⇒ f(1,Y). Sous cette condition, démontrer l'égalité f(x,Y) = f(0,Y) + x·f(1,Y) puis l'exploiter.

Adoptons les notations n=f(0,Y) pour le cofacteur **n**égatif de *f* par rapport à *x* et p=f(1,Y) pour son cofacteur **p**ositif. L'expansion de Boole de *f* par rapport à *x* fournit f = x'n+xp. Or il faut montrer que f = n+xp. On constate qu'il manque un terme xn pour y parvenir. Or la croissance de *f* s'exprime par n⇒p, qui équivaut à p=n+p. Repartant de l'expansion, il vient :

$$f = x'n+xp = x'n+x(n+p) = (x'+x)n+xp = n+xp \qquad \text{(CQFD)}$$

L'équivalence (n⇒p) ⇔ p=n+p, évidente d'un point de vue ensembliste (N⊂P ⇔ P=N∪P), peut se démontrer formellement comme suit. Par définition, (n⇒p) ⇔ n'+p=1 ⇔ np'=0. Donc (n⇒p) implique que p = p+np' = p+n (par absorption).

Pour utiliser des portes NAND, le résultat se réécrit : f(x,Y) = [f(0,Y)' . (x·f(1,Y))']', égalité qui correspond au montage de droite sur la figure ci-dessus, avec un délai T=3.

On notera que les montages retenus sont spécifiquement destinés à minimiser la longueur temporelle des chemins reliant l'entrée *x* à la sortie *z*, aux dépens des longueurs des chemins entre les autres variables d'entrée et de sortie du sous-circuit considéré.

3i) Appliquer les techniques développées ci-dessus pour optimiser la cellule FA et ainsi rendre l'additionneur à retenues propagées le plus rapide possible.

Comme conclu précédemment, il s'agit de reconcevoir la cellule FA pour minimiser son délai τ_{cc}. Or la fonction majorité est une fonction croissante par rapport à *cin* : en effet, pour *a* et *b* arbitraires fixés, une transition montante (0→1) de *cin* ne peut pas provoquer de transition descendante (1→0) de *cout*. En fait, symétrique, la fonction majorité est croissante par rapport à ses 3 variables et cela se voit en outre à sa FDM sans variable complémentée.

Finalement, le montage de droite sur la figure ci-dessus peut être exploité. Pour cela, on réécrit ainsi la fonction parité cout = a·b + (a+b)·cin et on aboutit au montage de gauche sur la figure ci-dessous, avec un petit τ_{cc} de 3, qui raccourcit fortement le délai total n·τ_{cc} de l'additionneur.

A posteriori, on peut réaliser qu'il suffisait de jouer avec les bulles sur l'implantation naïve de la fonction majorité pour parvenir au même résultat. Mais nous avons appris d'autres choses…

Il s'agit ensuite d'achever le montage pour réaliser le calcul du bit somme *s*. Or, (a·b)' et a+b étant déjà calculés, il s'avère qu'il suffit de les combiner par une porte NAND pour obtenir (a⊕b)'. Puis on combine ce résultat et *cin* avec une porte XNOR pour obtenir la fonction parité, d'où la cellule FA ci-dessous à gauche. Celle-ci est présentée prête à être aboutée de la droite vers la gauche pour réaliser l'additionneur à retenues propagées dans le sens naturel (LSB à droite, MSB à gauche).