Corrigé de l'épreuve d'informatique de Polytechnique (2005)

I. Génération de circuits PLA

Question 1 Si $x = 2^p$ alors $x-1 = 2^p - 1 = 1+2+\ldots+2^{p-1} \Rightarrow x \wedge (x-1) = (0.1)2^0 + \ldots (0.1)2^{p-1} + (1.0)2^p + (0.0)2^{p+1} + \ldots + (0.0).2^n = 0$.

Réciproque : par contraposition, si x n'est pas de la forme 2^p alors x est de la forme $2^q + \sum_{k=q+1}^{p-1} x_k 2^k + 2^p$ (avec $q < p$) et donc $x-1 = \sum_{k=0}^{q-1} 2^k + \sum_{k=q+1}^{p-1} x_k 2^k + 2^p$ ce qui nous donne $x \wedge (x-1) = \sum_{k=q+1}^{p-1} x_k 2^k + 2^p \neq 0$. Donc par contraposition, $x \wedge (x-1) = 0 \Rightarrow x$ de la forme 2^p.

conclusion : $\boxed{x \wedge (x-1) = 0 \iff x \text{ est de la forme } 2^p}$

Question 2 Si x et y diffèrent d'un seul bit, dans l'expression de $x \oplus y = (x_0 \oplus y_0)2^0 + (x_1 \oplus y_1)2^1 + \cdots + (x^{29} \oplus y^{29})2^{29}$, un et un seul des $x_i \oplus y_i$ est non nul et donc $x \oplus y$ est de la forme 2^p. La réciproque étant aussi vraie, on a une condition nécessaire est suffisante. Compte tenu de la question 1, la distance de Hamming entre x et y vaut 1 si et seulement si $(x \oplus y) \wedge ((x \oplus y) - 1) = 0$ ce qui nous amène à écrire le programme suivant (il faut faire attention au cas $x \oplus y \neq 0$) :

```
1  let aDistUn x y =
2  let x_oplus_y = x ⊕ y in
3      (x_oplus_y <> 0) && (x_oplus_y ∧ (x_oplus_y − 1) = 0);;
```

Question 3 $\boxed{h(x_0, x_1, x_2) = x_0 . \overline{x_1} \cdot \overline{x_2} \dot{+} x_0 \cdot x_1 \cdot \overline{x_2} \dot{+} \overline{x_0} \cdot \overline{x_1} \cdot x_2 \dot{+} \overline{x_0} \cdot x_1 \cdot x_2}$

Question 4 $h(x_0, x_1, x_2) = x_0 \cdot \overline{x_2} \cdot (\overline{x_1} \dot{+} x_1) \dot{+} \overline{x_0} \cdot x_2 \cdot (\overline{x_1} \dot{+} x_1) = x_0 \cdot \overline{x_2} \dot{+} \overline{x_0} \cdot x_2$

conclusion : $\boxed{h(x_0, x_1, x_2) = x_0 \cdot \overline{x_2} \dot{+} \overline{x_0} \cdot x_2}$

Question 5 Si p et q vérifient $p = p'.x_i.p''$ et $q = p'.\overline{x_i}.p''$ ou $p = p'.\overline{x_i}.p''$ et $q = p'.x_i.p''$, c'est qu'ils ont le même masque et que leurs valeurs sont à une distance de Hamming de 1. Ce qui nous donne la fonction `varEliminable` suivante :

```
1  let varEliminable p q =
2    (aDistUn p.v q.v) && (p.m = q.m);;
```

Pour la fonction `unifier`, le résultat a comme masque $p'p''$ (suivant les notations de l'énoncé), il s'agit donc de retirer le "i" du masque de p (ou de q). L'indice i est le seul indice de $p.v$ ou de $q.v$ qui n'est pas dans l'autre valeur, on peut donc le repérer par la formule $p.v \oplus q.v$. L'expression $\neg(p.v \oplus q.v)$ nous donne tous les autres indices et $p.m \wedge (\neg(p.v \oplus q.v))$ nous donne le masque $p'p''$. La valeur de $p'p''$ est donnée par $p.v \wedge q.v$ (on élimine le i dans $p.v$) ce qui nous donne la fonction suivante :

```
1  let unifier p q =
2  match varEliminable p q with
3      true -> {m=p.m ∧ (¬ (p.v ⊕ q.v)); v=p.v ∧ q.v}
4    | false -> failwith "produit_non_unifiable";;
```

Les fonctions &&, $\wedge$, =, $\oplus$, $\neg$ s'exécutant à temps constant, les fonctions `varEliminable` et `unifier` s'exécutent à temps constant, la complexité est en $O(1)$.

Question 6

```
let rec supprime x = function
  [] -> []
 | y::ys when y=x -> supprime x ys
 | y::ys -> y::(supprime x ys);;

let rec unique = function
  [] -> []
| x::xs -> x::(unique (supprime x xs));;
```

La fonction `supprime` a une complexité proportionnelle à la longueur de la liste passée en argument. La fonction `unique` appelle au plus ℓ fois la fonction `supprime` sur des listes de plus en plus petites. Dans le pire des cas, c'est à dire quand les éléments sont présents une et une seule fois dans la liste, la complexité sera en $\sum_{k=1}^{\ell} k = \frac{\ell(\ell+1)}{2} = O(\ell^2)$.

conclusion : La complexité de cette fonction est quadratique par rapport à la longueur ℓ (en $O(\ell^2)$).

Question 7 La fonction `Produits p l` permet de trouver tous les produits qu'il est possible d'obtenir entre p et les éléments de l et qui permettent l'élimination d'une variable.

```
let rec Produits p = function
      [] -> []
  | q::qs -> if varEliminable p q
                 then unifier p q::(Produits p qs)
                 else Produits p qs;;
```

On peut alors écrire la fonction `nouveauxProduits` sous la forme suivante :

```
let rec nouveauxProduits = function
      [] -> []
  | p::ps -> (Produits p ps)@(nouveauxProduits ps);;
```

La complexité des fonctions `varEliminable` et `unifier` étant à temps constant, la complexité de `Produits` est en temps linéaire par rapport à la longueur de la liste passée en argument. Donc la complexité de `nouveauxProduit` est en $O\left(\sum_{k=1}^{\ell-1} k\right) = O\left(\frac{\ell(\ell-1)}{2}\right) = O(\ell^2)$.

La complexité en temps de `nouveuxProduits` par rapport à la longueur ℓ de la somme a est en $O(\ell^2)$.

Question 8 La suite des a_i obtenus pour h est :

$a_0 = \langle$`001,011,100,110`$\rangle$, $a_1 = \langle$`0-1,1-0`$\rangle$ et on s'arrête.

Question 9

```
let rec developper = function
  [] -> failwith "Erreur : liste vide"
  | x::xs -> let y = unique(nouveauxProduits x) in
      if y=[] then x::xs
              else developper (y::x::xs);;
```

On prend soin d'enlever les doublons de la liste de produits retournée par la fonction `nouveauxProduits` pour avoir des ℓ_i les plus petits possibles.

La complexité de "`nouveauxProduits` a_i" est en $O(\ell_i^2)$ et celle de "`unique`" appliquée au résultat en $O(\ell_{i+1}^2)$ d'où la complexité de la fonction `developper` par rapport aux longueurs ℓ_i est en $O\left(\sum_{i=0}^{k} \ell_i^2\right)$.

Question 10 En fait, p est couvert par q si et seulement si le masque de p est inclus dans le masque q (ce qui va se traduire en caml par `q.m` $\wedge$ `p.m = p.m`) et les positions des bits à 1 de p du masque de p doivent coïncider avec les positions des bits à 1 de q (ce qui se traduit en caml par `p.v = q.v` $\wedge$ `p.m`).

```
let couvertPar p q =
  (q.m ∧ p.m = p.m) && (p.v = (q.v ∧ p.m));;
```

Tous les opérateurs utilisés travaillant à temps constant, la complexité en temps de cette fonction est à temps constant ($O(1)$).

Question 11 La fonction `simplifie2 p l` prend comme argument un produit p et une liste de produits `l` et retourne la liste obtenue à partir de la liste `l` en supprimant tous les produits couverts par `p`.

```
let rec simplifie2 p = function
  [] -> []
  | q::qs -> if couvertPar p q then simplifie2 p qs
                               else q::(simplifie2 p qs);;
```

La fonction `aux l s` prend comme argument `l` une liste de produits et `s` une liste de sommes et retourne la liste des sommes `s'` obtenue à partir de `s` en supprimant tous les produits des sommes qui sont couverts par l'un des produits de `l`.

```
let rec simplifie1 = fun
    [] lq -> lq
  | (p::ps) lq -> simplifie2 p (simplifie1 ps lq);;
```

La fonction `reduit` prend comme argument une liste de listes de produits $\langle a_k, a_{k-1}, ..., a_0\rangle$ et retourne la liste de listes de produits $\langle a_k, a'_{k-1}, ..., a'_0\rangle$ où a'_i est obtenu à partir de a_i en supprimant les produits couverts par l'un des produits de a_{i+1}

```
let rec reduit = function
    [] -> []
  | [a] -> []
  | a::b::suite -> (simplifie1 a b)::(reduit (b::suite));;
```

On suppose que l'argument de `reduire` est une liste de listes de produits fournie par la fonction `developper`. Il ne peut donc y avoir de produits qui se couvrent dans un même a_i. L'idée est de prendre chaque produit p de a_k et de supprimer dans les a_{k-1} les produits q couverts par p. La transitivité de la relation « *est couvert par* » nous assure que l'on supprimera tous les produits des a_i pour $i < k$ couverts par p.

```
let reduire l =
   (hd l)::(reduit l);;
```

La complexité de cette fonction par rapport aux longueurs ℓ_0, $\ell_1, \dots \ell_k$ des listes a_0, $a_1, \dots a_k$ composant la représentation $\langle a_k, \dots a_2, a_1, a_0\rangle$ de s est en $O\left(\sum_{i=1}^{k} \ell_i \ell_{i-1}\right)$. En effet, on appelle la fonction `simplifie1` pour chaque couple de liste (a_i, a_{i-1}) qui nécessite de parcourir ℓ_i fois la liste a_{i-1} avec à chaque étape une opération de coût constant (dûe à `couvertPar` dans la fonction `simplifie2`).

Question 12 Si on a n arguments à notre fonction f, dans a_0, on a au plus 2^n éléments ($\ell_0 \leqslant 2^n$) ce qui est le cas de la fonction constante égale à 1. Les éléments de a_k ont un masque de longueur $n-k$ et on a C_n^{n-k} choix de masques possibles et pour chaque masque, on a 2^{n-k} choix de valeurs possibles. Donc $\ell_k \leqslant C_n^{n-k}.2^{n-k}$ (si on a pris soin d'éviter les doublons en utilisant la fonction `unique`).

Pour trouver la f.n.d. réduite d'une fonction f, il faut commencer par calculer a_0 ce qui se fera en $O(2^n)$ puis développer ce qui se fera au maximum en $O\left(\sum_{k=0}^{n}(C_n^{n-k}.2^{n-k})^2\right) = O\left(\sum_{k=0}^{n}(C_n^k)^2.4^k\right)$. La fonction de `reduire` aura une complexité majorée par $O\left(\sum_{i=1}^{k} C_n^{n-i}.2^{n-i}.C_n^{n-i+1}.2^{n-i+1}\right) = O\left(\sum_{k=0}^{n-1} C_n^k.C_n^{k+1}.2^{2k+1}\right)$.
Comme $a.b \leqslant \frac{1}{2}(a^2+b^2)$, on peut majorer de la manière suivante :

$$\sum_{k=0}^{n-1} C_n^k.C_n^{k+1}.2^{2k+1} \leqslant \frac{1}{2}\left(\sum_{k=0}^{n-1}(C_n^k.2^k)^2 + \sum_{k=1}^{n}(C_n^k.2^k)^2\right) \leqslant \sum_{k=0}^{n}(C_n^k.2^k)^2$$

Conclusion : la borne supérieure de la complexité du calcul du résultat de la f.n.d reduite est en $O\left(\sum_{k=1}^{n}(C_n^k.2^k)^2\right)$

Question 13 a) Dans le PLA, on câble le résultat de la f.n.d. réduite de chaque fonction f_i. Dans la partie-*Et*, on câble les produits qui compose une *f.n.d* de chaque fonction f_i et dans la partie-OU, on les assemble à travers un porte OU pour reconstituer chaque fonction f_i.

Pour $f_0(x_0,x_1,x_2) = g(x_0,x_1) = \overline{x_0} \dot{+} x_0.x_1 = \overline{x_0} \dot{+} x_1$ et $f_1(x_0,x_1,x_2) = h(x_0,x_1,x_2) = x_0 \cdot \overline{x_2} \dot{+} \overline{x_0} \cdot x_2$, on obtient le PLA donnée par la figure 1.

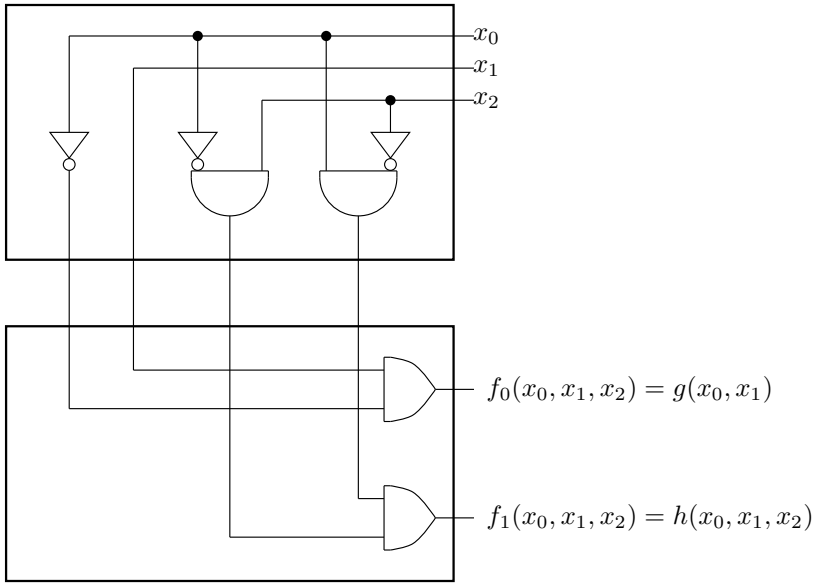

FIG. 1 – Circuit PLA de la question 13.

b) k correspond au nombre de fils entre la partie ET et la partie OU, c'est à dire à la somme des nombres de produits de littéraux formant les formes normales disjonctives des fonctions f_i. Pour réduire k, il est important d'utiliser les *f.n.d.* minimales pour chaque f_i et de repérer les produits identiques des différents *f.n.d* pour ne pas les câbler plusieurs fois.

II. Génération de circuits combinatoires

Question 14 On établit le tableau des valeurs des fonctions $r(x_2, x_1, x_0)$ et $q(x_2, x_1, x_0)$ qui retournent respectivement le reste modulo 2 de $x_2 + x_1 + x_0$ et le quotient de la division de $x_2 + x_1 + x_0$ par 2 :

x_2	x_1	x_0	$r(x_2, x_1, x_0)$	$q(x_2, x_1, x_0)$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$r(x_2, x_1, x_0) = x_0.\overline{x_1}.\overline{x_2} \dot{+} \overline{x_0}.x_1.\overline{x_2} \dot{+} \overline{x_0}.\overline{x_1}.x_2 \dot{+} x_0.x_1.x_2$ qui n'est pas simplifiable ce qui nous donne :

```
let bitAdd x y z =
Ou(Et(Non(x),Et(Non(y),z),
    Ou(Et(Non(x),Et(y,Non(z))),
        Ou(Et(x,Et(Non(y),Non(z))),
            Et(x,Et(y,z))))));;
```

$q(x_2, x_1, x_0) = x_0.x_1.\overline{x_2} \dot{+} x_0.\overline{x_1}.x_2 \dot{+} \overline{x_0}.x_1x_2 \dot{+} x_0.x_1.x_2 = x_0.x_1 \dot{+} x_0.x_2 \dot{+} x_1.x_2$, ce qui nous donne :

```
let bitRetenue x y z =
  Ou(Et(x,y),Ou(Et(x,z),Et(y,z)));;
```

Question 15

```
let addSerie (x : mot) (y : mot) =
let n = vect_length x in
let retenue = ref(Bit(0)) in
let res = make_vect (n+1) (Bit(0)) in
for i = 0 to n-1 do
  res.(i) <- bitAdd x.(i) y.(i) !retenue;
  retenue := bitRetenue x.(i) y.(i) !retenue;
done;
res.(n) <- !retenue;
(res : mot);;
```

Voici une version avec moins de portes en simplifiant les portes au niveau des poids faibles.

```
let addSerie (x : mot) (y : mot) =
let n = vect_length x in
let res = make_vect (n+1) (Bit(0)) in
let retenue = ref(Et(x.(0),y.(0))) in
res.(0) <- Ou(Et(Non(x.(0)),y.(0)),Et(x.(0),Non(y.(0))));
for i = 1 to n-1 do
  res.(i) <- bitAdd x.(i) y.(i) !retenue;
  retenue := bitRetenue x.(i) y.(i) !retenue;
done;
res.(n) <- !retenue;
(res : mot);;
```

La fonction `bitAdd` génère 17 portes et la fonction `bitRetenue` 5 portes. Ces fonctions étant appelées n fois dans la fonction `addSerie`, l'ordre de grandeur du nombre de portes est de $22.n$ portes.

À cause de la propagation de la retenue, on ne peut exécuter les calculs en parallèle et donc l'ordre de grandeur du temps de calcul sera directement proportionnel au nombre de portes soit en $O(22.n)$.

Question 16 La question n'est pas très claire. On nous dit que la fonction `mux` doit retourner un circuit alors que dans le typage de la fonction, elle retourne un mot ! Voici l'interprétation que j'ai donné de cette question, on commence par écrire une fonction d'évaluation d'un circuit puis on écrit la fonction `mux` :

```
let rec valeur = function
  Bit(b) -> b
| Non(c) -> 1-(valeur c)
| Et(c1,c2) -> (valeur c1)*(valeur c2)
| Ou(c1,c2) -> min 1 (valeur c1 + valeur c2);;

let mux s (x : mot) (y : mot) =
  if valeur s = 1 then x else y
;;
```

Question 17

```
let addPar (x : mot) (y : mot) (r : circuit) =
let n = vect_length x in
let retenue = ref(r) in
let res = make_vect (n+1) (Bit(0)) in
for i = 0 to n-1 do
  res.(i) <- bitAdd x.(i) y.(i) !retenue;
  retenue := bitRetenue x.(i) y.(i) !retenue;
done;
res.(n) <- !retenue;
(res : mot);;
```

Il n'y a pas de changement fondamental par rapport à la fonction `addSerie` de la question 15 et donc l'ordre de grandeur du nombre de porte reste $O(20.n)$ ainsi que le temps d'exécution. Si on veut utiliser les remarques qui précèdent la question, il faut réécrire la fonction `addSerie` de la manière suivante (en coupant les mots en deux) :

```
let addSeriebis (x : mot) (y : mot) =
let n = vect_length x in
let p = n/2 in
let x1 = sub_vect x 0 p and x2 = sub_vect x (p+1) (n-1) in
let y1 = sub_vect y 0 p and y2 = sub_vect y (p+1) (n-1) in
let circuit1 = addSerie x1 y1 in
let retenu = circuit1.(p-1) in
let circuit1' = sub_vect circuit1 0 (p-2) in
let circuit2 = addPar x2 y2 (Bit(1)) in
let circuit3 = addPar x2 y2 (Bit(0)) in
concat_vect circuit1' (mux retenu circuit2 circuit3);;
```

Le nombre de portes est à peu près multiplié par 1,5 tandis que le temps d'exécution est à peu près divisé par deux.

Question 18 Le problème pour utiliser un PLA est le nombre de portes logiques à utiliser. Grosso modo, l'usage d'un PLA revient à câbler toutes les additions $p+q$ pour tous les couples $(p,q) \in [0, 2^n]$. On a alors une croissance exponentielle du nombre de portes en fonction de n. L'avantage est l'exécution se fait à temps constant, chaque somme $p+q$ étant associé à un et un seul produit de la *f.n.d.* de la fonction d'addition contrairement aux additionneurs de la partie II qui ont des temps d'exécution proportionnel à n.