Minimisation de fonction par ESPRESSO [1],[2]

1 Introduction

Lors du développement de l'électronique digitale, la problématique suivante s'est posée : nous avons des fonctions à 50 variables à minimiser. Comment faire ? Avec les méthodes graphiques comme Karnaugh, on est vite limité (on peut aller à 5 ou 6 variables si on a une bonne représentation 3D). Avec les méthodes par tableaux comme Quine-McCluskey, on liste tout. Si on confie cela à un PC, il va mettre énormément de temps à lister tous les minterms et impliquants. Il faut donc trouver une méthode rapide et efficace : Espresso est né.

Espresso est un algorithme de minimisation de fonction booléenne développé par IBM [3] et amélioré ensuite par l'université de Berkley [4]. IBM a réalisé sa publication sur le sujet en 1984, Berkley en 1986.

Depuis lors, cette algorithme a subit de nombreuses améliorations et est utilisé dans les logiciels de conception de ASIC (Application-Specific Integrated Circuit) ou FPGA (Field-Programmable Gate Array) qui sont des circuits logiques combinatoires et/ou séquentiels qui peuvent comporter des centaines de millions de portes.

Par exemple, voici un processeur [1] spécialisé dans le traitement du signal réalisé par la société Imec. Imec est une entreprise spécialisée en micro-électronique dont la maison mère se situe en Belgique, près de Leuven.

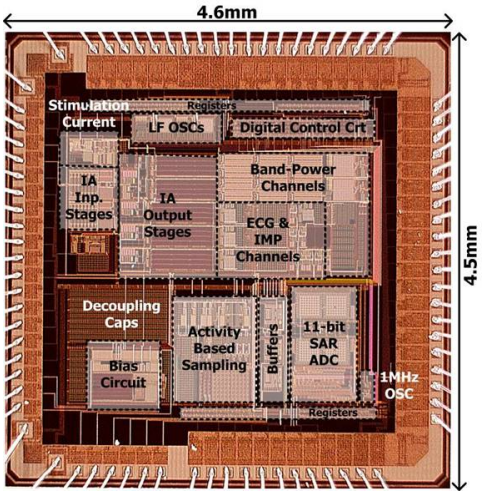

FIGURE 1 – Un processeur de signal intégrant diverses composants

1. Imec and Holst Centre's $30\mu W$ analog signal processor ASIC for biopotential signal monitoring, 2010, available at http ://start.artoos.eu/dmp/printflo/content/107/96/default.aspx

2 Principes généraux de la boucle Reduction-Expansion-Irredondance

La boucle Reduction-Expansion-Irredondance (*Reduce-Expand-Irredundant Loop*) constitue le coeur de l'algorithme Espresso. Cette boucle est réalisée plusieurs fois jusqu'à obtention d'une fonction minimisée. On ne cherche pas la meilleure solution car cela prendrait trop de temps, mais simplement une solution assez bonne.

Partons d'un diagramme de Karnaugh représentant une fonction non-minimale à 5 impliquants.

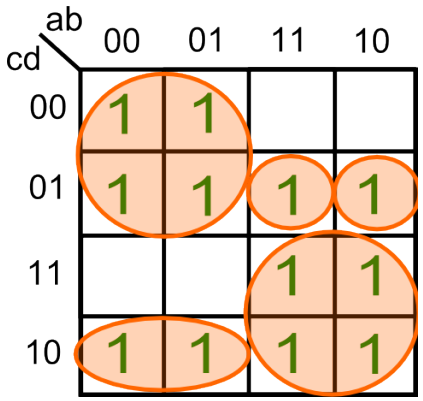

FIGURE 2 – Fonction de départ non-minimisée

Expansons tous les impliquants au maximum afin de n'obtenir plus que des impliquants premiers. Ainsi, l'impliquant $ab\bar{c}d$ devient $\bar{c}d$, l'impliquant $a\bar{b}\bar{c}d$ devient ad et l'impliquant $\bar{a}c\bar{d}$ devient $c\bar{d}$.

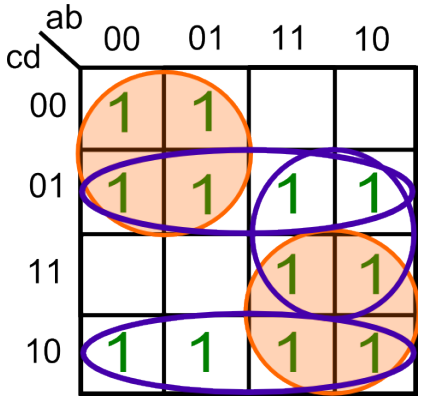

FIGURE 3 – Première phase d'expansion

Supprimons les impliquants redondants . Les 1 de l'impliquant ad sont tous couverts par un autre impliquant. L'impliquant n'est donc pas utile et est redondant. On peut donc le retirer.

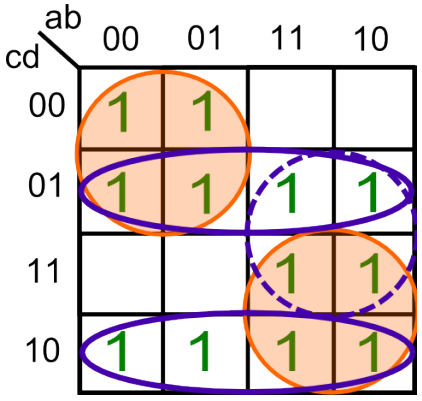

FIGURE 4 – Première phase d'irredondance

Réduisons les impliquants au minimum . Les impliquants sont réduits afin de ne couvrir que les 1 absolument indispensables, que d'autre impliquants ne couvrent pas. L'impliquant $\bar{c}d$ devient $a\bar{c}d$ et l'impliquant $c\bar{d}$ devient $\bar{a}c\bar{d}$.

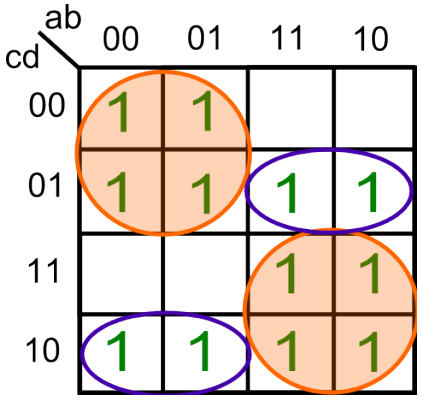

FIGURE 5 – Première phase de réduction

La première boucle est terminée. On en commence une deuxième.

Expansons à nouveau tous les impliquants au maximum afin de n'obtenir plus que des impliquants premiers. Ainsi, l'impliquant $a\bar{c}d$ devient ad et l'impliquant $\bar{a}c\bar{d}$ devient $c\bar{d}$.

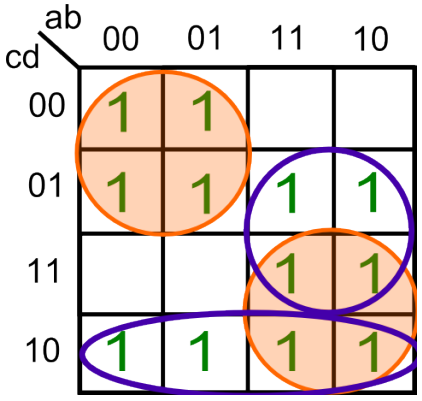

FIGURE 6 – Deuxième phase d'expansion

Supprimons les impliquants redondants. Nous pouvons supprimer ac.

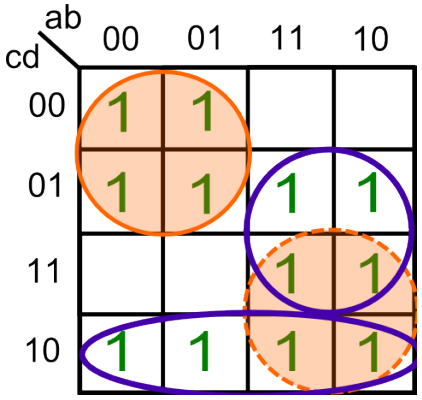

FIGURE 7 – Deuxième phase d'irredondance

Réduisons les impliquants au minimum... (il n'y a plus moyen de réduire), on y est !

Il faut généralement moins de 5 itérations pour trouver une solution minimale, irredondante, dans lequel on ne retrouve que des impliquants premiers.

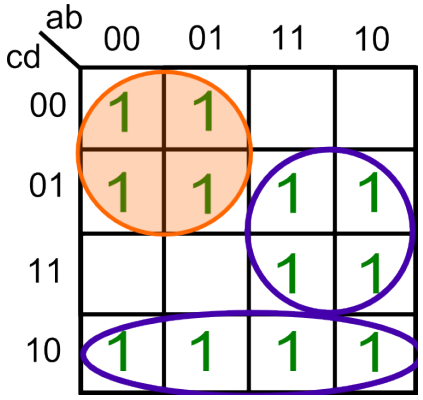

FIGURE 8 – Deuxième phase de réduction et fin

Afin de pouvoir analyser la mise en œuvre de cet algorithme en software, nous devons commencer par étudier quelques notions de représentation, traitement et analyse de fonction booléenne.

3 Représentation, analyse et traitement de fonction booléenne

3.1 Positional Cube Notation (PCN)

Le PCN (*Positional Cube Notation*) est une technique permettant de décrire informatiquement un impliquant appelé cube. Cette méthode est assez simple et permet d'effectuer des opérations sur des fonctions représentées par des listes d'impliquants ou cubes.

Le principe est le suivant :

- **Pour chaque impliquant, on crée un vecteur ligne** comprenant autant de colonne qu'il y a de variable dans la fonction.

  Exemple :
  - pour le minterm $a\bar{b}\bar{c}d$: pcn = [.. .. .. ..] où chaque .. est une valeur à définir.
  - pour l'impliquant $x\bar{y}$ (la fonction à laquelle appartient cet impliquant contient les variables x, y et z) : pcn = [.. .. ..] où chaque .. est une valeur à définir.

- **Chaque colonne représente une variable.**

  Exemple
  - pour le minterm $a\bar{b}\bar{c}d$:

    $a\ b\ c\ d$
    ↓ ↓ ↓ ↓
    pcn = [.. .. .. ..]
  - pour l'impliquant $x\bar{y}$ (la fonction à laquelle appartient cet impliquant contient les variables x, y et z) :

    $x\ y\ z$
    ↓ ↓ ↓
    pcn = [.. .. ..]

- **Le vecteur est rempli en regardant si chaque variable apparait, niée ou non dans l'impliquant.** Si la variable apparait non niée dans l'impliquant, écrire '01' dans la colonne de la variable. Si la variable apparait niée dans l'impliquant, écrire '10' dans la colonne de cette variable. Si la variable n'apparait pas du tout, alors écrire '11' dans la colonne de la variable.

  En résumé
  - Variable non niée => 01
  - Variable niée => 10
  - Variable absente => 11

Exemple
- pour le minterm $a\bar{b}\bar{c}d$: a et d apparaissent non niés (donc '01'), b et c apparaissent niés (donc '10').

  $a \quad b \quad c \quad d$
  $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
  $pcn = [\,01 \;\; 10 \;\; 10 \;\; 01\,]$
- pour l'impliquant $x\bar{y}$ (la fonction à laquelle appartient cet impliquant contient les variables x, y et z) : x apparait non nié ('01'), y apparait nié ('10') et z n'apparait pas ('11').

  $x \quad y \quad z$
  $\downarrow \quad \downarrow \quad \downarrow$
  $pcn = [\,01 \;\; 10 \;\; 11\,]$

Grâce à cette notation, nous allons pouvoir exprimer nos fonctions booléennes comme étant des listes d'impliquants ou cubes.

Exemple
Si $f = a\bar{b} + \bar{a}\bar{b}c + bc$, la liste sera : [01 10 11], [10 10 01], [11 10 10]

3.2 L'expansion de Shannon

L'expansion de Shannon est une théorie développée par Claude Shannon (bien connu pour son théorème de l'échantillonnage) qui permet de décomposer une fonction booléenne, dans le même sens que la décomposition en série de Taylor en algèbre classique ou encore en série de Fourrier dans le cas de signaux périodiques.

Ce théorème fait apparaitre la notion de cofacteur, négatif ou positif :

Soit la fonction F(x1, x2, ..., xi, ...xN) , F(xi = 0) est le cofacteur négatif de F selon xi

Soit la fonction F(x1, x2, ..., xi, ...xN) , F(xi = 1) est le cofacteur positif de F selon xi

Le cofacteur de F selon xi est simplement F pour laquelle on a remplacé xi par 1 (pour le cofacteur positif) et par 0 (pour le cofacteur négatif).

Exemple
Si $f = a\bar{b} + \bar{a}\bar{b}c + bc$,
le cofacteur positif selon a, $f(a = 1)$, vaut $\bar{b} + bc$
le cofacteur négatif selon a, $f(a = 0)$, vaut $\bar{b}c + bc$

L'expansion de Shannon pose l'égalité suivante :

$$\mathbf{F(x1, x2, \ldots, xi, \ldots.xN) = xi * F(xi = 1) + \bar{xi} * F(xi = 0)}$$

Exemple :
$f = a\bar{b} + \bar{a}\bar{b}c + bc$ est il égal à $a * f(a = 1) + \bar{a} * f(a = 0)$?

En sachant que $f(a = 1)$ vaut $\bar{b} + bc$ et que $f(a = 0)$ vaut $\bar{b}c + bc$ (cfr exemple précédent)
$a * f(a = 1) + \bar{a} * f(a = 0)$
$= a * (\bar{b} + bc) + \bar{a} * (\bar{b}c + bc)$
$= a\bar{b} + abc + \bar{a}\bar{b}c + \bar{a}bc$ (distributivité)
$= a\bar{b} + \bar{a}\bar{b}c + bc(\bar{a} + a)$ (mise en évidence)
$= a\bar{b} + \bar{a}\bar{b}c + bc$ (complément)

Une fois l'expansion réalisée sur une variable, on peut le refaire pour une autre. On aura alors que :

$$\mathbf{F(x1, x2, \ldots, xi, xj, \ldots.xN) = xi * xj * F_{xi,xj} + \bar{xi} * xj * F_{\bar{xi},xj} + xi * \bar{xj} * F_{xi,\bar{xj}} + \bar{xi} * \bar{xj} * F_{\bar{xi},\bar{xj}}}$$

avec
$F_{xi,xj} = F(xi = 1, xj = 1)$
$F_{\bar{xi},xj} = F(xi = 0, xj = 1)$
$F_{xi,\bar{xj}} = F(xi = 1, xj = 0)$
$F_{\bar{xi},\bar{xj}} = F(xi = 0, xj = 0)$

Exemple :
Si $F = a\bar{b} + \bar{a}\bar{b}c + bc$ alors $F_{a,\bar{c}} = \bar{b}$

Comment faire pour décomposer une liste de cube en cofacteur ?

Pour obtenir **le cofacteur positif** d'une liste de cube selon une variable x, regarder dans la colonne de x de chaque cube :
- Si on trouve un **'10'**, c'est que x apparait niée dans le cube. On doit donc supprimer ce cube. En effet, si j'ai $\bar{x}yz$ et que je met x à 1, mon cube vaut 0.
- Si on trouve un **'01'**, c'est que x apparait non nié dans le cube. On doit donc remplacer ce '01' par '11', pour signifier que x ne doit plus apparaitre dans le cube. En effet, si j'ai xyz et que je met x à 1, mon cube vaut yz.
- Si on trouve un **'11'**, c'est que x n'apparait pas dans le cube. Il ne faut donc rien faire. En effet, si j'ai yz et que je met x à 1, mon cube reste yz.

Pour obtenir **le cofacteur négatif** d'une liste de cube selon une variable x, regarder dans la colonne de x de chaque cube :
- Si on trouve un **'01'**, c'est que x apparait non niée dans le cube. On doit donc supprimer ce cube. En effet, si j'ai xyz et que je met x à 0, mon cube vaut 0.
- Si on trouve un **'10'**, c'est que x apparait niée dans le cube. On doit donc remplacer ce '10' par '11', pour signifier que x ne doit plus apparaitre dans le cube. En effet, si j'ai $\bar{x}yz$ et que je met x à 0, mon cube vaut yz.
- Si on trouve un **'11'**, c'est que x n'apparait pas dans le cube. Il ne faut donc rien faire. En effet, si j'ai yz et que je met x à 0, mon cube reste yz.

3.3 Tautologie

Une fonction F est une tautologie si cette fonction vaut 1.

F est une tautologie si F == 1

Si vous avez une fonction à 50 variables, c'est assez difficile de se dire si elle vaut 1 ou pas. C'est pourquoi on peut utiliser les cofacteurs pour découper le problème en sous-problèmes plus simples à résoudre.

F sera une tautologie si ces cofacteurs positifs et négatifs selon une variable x sont aussi tous les deux une tautologie.

F est une tautologie si $F_x = 1$ ET $F_{\bar{x}} = 1$

En effet, si $F_x = 1$ et $F_{\bar{x}} = 1$, alors
$F = x * F_x + \bar{x} * F_{\bar{x}}$
$= x * 1 + \bar{x}i * 1$
$= x + \bar{x}i$
$= 1$

3.4 Fonction monoforme (*unate*) et biforme (*binate*)

Une fonction est dite monoforme (*unate*) si elle ne contient qu'une polarité de chaque variable, c'est à dire que chaque variable n'apparait soit qu'en polarité nié, soit qu'en polarité non niée. Si elle n'est pas monoforme, elle est dite biforme (*binate*).

Exemple

$$f = abd + b\bar{c}d + a\bar{c}$$

Cette fonction est **monoforme** car a est toujours non niée, b est toujours non niée, c est toujours niée et d est toujours non niée.

$$f = \bar{a}d + bcd + a\bar{c}$$

Cette fonction est **biforme** car a et c apparaissent à la fois niées et non niées.

Comment faire pour vérifier si une fonction est monoforme ou non dans une liste de cube ?

Il est assez facile de vérifier dans une liste de cube si la fonction est monoforme ou pas. Il suffit de vérifier que chaque variable n'est présente que dans une polarité (10 ou 01) dans la liste de cube.

Exemple

Soit la fonction f décrite par cette liste de cube :
[11 10 10]
[01 10 11]
[11 11 10]
↓ ↓ ↓
a b c
f est **monoforme** car a n'apparait qu'en polarité non niée (01), b en polarité niée (10) et c en polarité niée (10). Les '11' signifient que la variable ne se trouvent pas dans le cube, il ne faut donc pas en tenir compte.
Soit la fonction f décrite par cette liste de cube :
[10 10 10]
[01 10 11]
[11 11 10]
↓ ↓ ↓
a b c
f est **biforme** car a apparait dans les deux polarités (10 et 01).

3.5 Unate Recursive Paradigm (URP)

L'URP (Unate Recursive Paradigm) est une méthode d'analyse de fonction booléenne permettant par exemple, de vérifier une tautologie, de calculer le complément d'une fonction ou encore de déterminer les impliquants premiers.

Son principe est d'utiliser les cofacteurs pour décomposer la fonction et de profiter des caractéristiques des fonctions et variables monoformes pour l'analyse et la décomposition de la fonction.

Voici son principe de fonctionnement dans le cadre d'une vérification de tautologie.

1. Utiliser les cofacteurs de manière récursive pour diviser le problème en sous-problème. On sait que, pour qu'il y ait tautologie, chaque cofacteur selon une variable doit être aussi une tautologie. Pour choisir quelle variable utiliser pour décomposer en cofacteur, il faut choisir la variable la plus biforme (celle qui apparait le plus grand nombre de fois sous la forme 10 ou 01). Si deux variables sont autant biformes l'une que l'autre, choisir celle qui est la plus balancée en terme de polarité (|nombre de variable niée - nombre de variable non niée| le plus petit possible).

   Exemple
   Soit la fonction suivante :
   [11 01 10 11]
   [01 01 10 10]
   [10 10 01 11]
   [10 10 10 10]
   ↓ ↓ ↓ ↓
   a b c d
   La variable a est biforme pour 3 cubes. La variable b est biforme pour 4 cubes. La variable c est biforme pour 4 cubes. La variable d est monoforme. Comme nous devons choisir la variable la plus biforme, nous avons le choix entre le b et la c. Pour se décider entre les deux, il faut choisir la plus balancée : pour la variable b, nous avons 2 niées et 2 non-niées : 2-2=0. Pour la c, nous avons 3 niées et 1 non-niées : 3-1=2. Nous choisissons donc de factoriser selon b.

   Selon la méthode présentée au paragraphe 3.2, nous obtenons :

   pour F_b :
   [11 11 10 11]
   [01 11 10 10]

   pour $F_{\bar{b}}$:
   [10 11 01 11]
   [10 11 10 10]

2. Utiliser la propriété suivante : la fonction sera une tautologie si et seulement si un des cubes ne comprend que des *dont' care* (11). Le cube [11 11 11] correspond bien à 1. Si j'ai ce cube dans une fonction, la fonction devient ... + 1 + ... = 1.

Exemple
Soit la fonction suivante :
[11 01 10 11]
[01 01 10 10]
[11 11 11 11]
[10 10 10 10]
↓ ↓ ↓ ↓
a b c d
Cette fonction correspond à $b\bar{c} + ab\bar{c}\bar{d} + 1 + \bar{a}\bar{b}\bar{c}\bar{d} = 1$

3. Profiter des caractéristiques des fonctions monoformes. En effet, si j'ai une liste de cube monoforme et qu'il n'y a pas de cube ne comprenant que des '11', alors on peut conclure que la fonction n'est pas une tautologie. Il n'est pas possible de créer une fonction égale à 1 en utilisant des variables n'apparaissant que dans une seule polarité.

Exemple
Soit la fonction suivante :
[11 01 10 11]
[01 01 10 10]
[11 10 01 11]
[10 10 10 10]
↓ ↓ ↓ ↓
a b c d
Cette fonction est monoforme et ne comprend pas cube rempli de *don't care*. C'est donc impossible qu'elle soit une tautologie.

4. Utiliser la propriété suivante : si dans la liste de cube, j'ai deux cubes d'une seule variable compléments l'un de l'autre, alors ma fonction est une tautologie. Par exemple, [11 11 01] et [11 11 10] correspondent à c et $\bar{c}$. Ma fonction va dont contenir ... + c + $\bar{c}$ + ... = ... + 1 + ... = 1.

Exemple
[11 11 11 01]
[01 01 10 10]
[11 10 01 11]
[11 11 11 10]
↓ ↓ ↓ ↓
a b c d
Cette fonction vaut $d + ab\bar{c}\bar{d} + \bar{b}c + \bar{d} = ab\bar{c}\bar{d} + \bar{b}c + 1 = 1$

L'algorithme *URP Tautology* est donc le suivant :

```
1
2  IsTautology(f in cubelist form){
3  //Vérifier si la fonction a un cube rempli de don't care.
4  //Si oui, la fonction est une tautologie
5  if (f has an all don´t care cube){
6     return 1;
7  }
8  //Sinon, vérifier si la fonction est monoforme
9  //Si elle l'est, la fonction n'est pas une tautologie
10 else if (f is unate){
11    return 0;
12 }
13 //Sinon, vérifier si deux cubes sont complémentaires
14 //Si oui, la fonction est une tautologie
15 else if (f has 2 complement cube ){
16    return 1;
17 }
18 //Sinon
19 else{
20    //Choisir la variable à partir de laquelle on va
21    //  décomposer la fonction.
22    //Cette variable doit être la variable la plus biforme
23    //  et la plus balancée
24    x = most binate variable in f;
25    //Appeler récursivement la fonction pour les deux cofacteurs.
26    return IsTautology(fx) && IsTautology(fnot(x));
27 }
28 }
```

Exemple

Soit la fonction suivante :

$$ab + ac + a\bar{b}\bar{c} + \bar{a}$$

Elle est représentée par la liste de cube suivant :

$[\,01\;\;01\;\;11\,] \rightharpoonup ab$
$[\,01\;\;11\;\;01\,] \rightharpoonup ac$
$[\,01\;\;10\;\;10\,] \rightharpoonup a\bar{b}\bar{c}$
$[\,10\;\;11\;\;11\,] \rightharpoonup \bar{a}$
$\downarrow\quad\downarrow\quad\downarrow$
$a\quad b\quad c$

1. Y a-t-il un cube rempli de don't care ? Non. Testons un autre cas.
2. La fonction est elle monoforme ? Non. Testons un autre cas.
3. Y a-t-il deux cubes compléments ? Non. Il faut factoriser.

4. Quelle variable choisir pour factoriser ? C'est a car elle est la plus biforme (elle apparait 4 fois, les autres 2 ou 3).

Il faut donc factoriser suivant a. Cela donne :
- pour f_a :
  [11 01 11] ⇀ b
  [11 11 01] ⇀ c
  [11 10 10] ⇀ $\bar{b}\bar{c}$

- pour $f_{\bar{a}}$:
  [11 11 11] ⇀ $\bar{a}$

Pour que f soit une tautologie, il faut que ces deux cofacteurs selon a le soient aussi.

Est-ce que $f_{\bar{a}}$ est une tautologie ?

1. Y a-t-il un cube rempli de don't care ? Oui. C'est une tautologie.

Est-ce que f_a est une tautologie ?

1. Y a-t-il un cube rempli de don't care ? Non. Testons un autre cas.
2. La fonction est elle monoforme ? Non. Testons un autre cas.
3. Y a-t-il deux cubes compléments ? Non. Il faut factoriser.
4. Quelle variable choisir pour factoriser ? On peut choisir indifféremment b ou c. Choisissons b.

Cela donne :
- pour f_{ab} :
  [11 11 11] ⇀ 1
  [11 11 01] ⇀ c

- pour $f_{a\bar{b}}$:
  [11 11 01] ⇀ c
  [11 11 10] ⇀ $\bar{c}$

Pour que f_a soit une tautologie, il faut que ces deux cofacteurs selon b le soient aussi.

Est-ce que f_{ab} est une tautologie ?

1. Y a-t-il un cube rempli de don't care ? Oui. C'est une tautologie.

Est-ce que $f_{a\bar{b}}$ est une tautologie ?

1. Y a-t-il un cube rempli de don't care ? Non. Testons un autre cas.
2. La fonction est elle monoforme ? Non. Testons un autre cas.
3. Y a-t-il deux cubes compléments ? Oui. C'est une tautologie.

En conséquence, f_a est une tautologie car f_{ab} et $f_{a\bar{b}}$ le sont.

Au final, f est une tautologie car f_a et $f_{\bar{a}}$ le sont.

4 La réduction en détails

La phase de réduction sert à réduire au minimum le nombre de 1 couvert par un impliquant, afin qu'il y ait un minimum de 1 qui soient couvert par plusieurs impliquants. On part donc d'une fonction F décrite par une liste de cube qui contient l'impliquant R que l'on souhaite réduire. Cela signifie qu'il faut ajouter une ou des variables dans l'impliquant.

Cette phase comprend les étapes suivantes :

1. On retire l'impliquant R de la fonction F pour former la fonction $F - \{R\}$.

Exemple
Soit la fonction F suivante :
[11 01 11 01] ⇀ bd
[01 11 11 11] ⇀ a
[11 11 10 11] ⇀ $\bar{c}$
↓ ↓ ↓ ↓
a b c d
dont on souhaite réduite l'impliquant R : [11 11 10 11] ⇀ $\bar{c}$
La fonction $F - \{R\}$ est donc :
[11 01 11 01] ⇀ bd
[01 11 11 11] ⇀ a

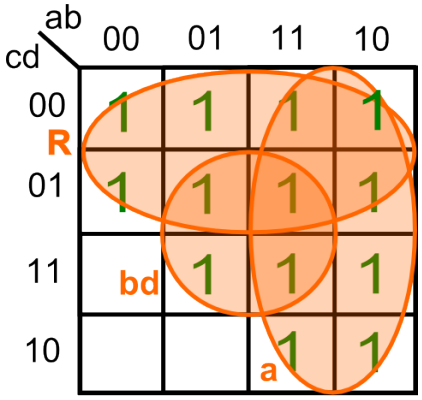

FIGURE 9 – Les 3 cubes de la fonction F : a, bd et R, l'impliquant à réduire

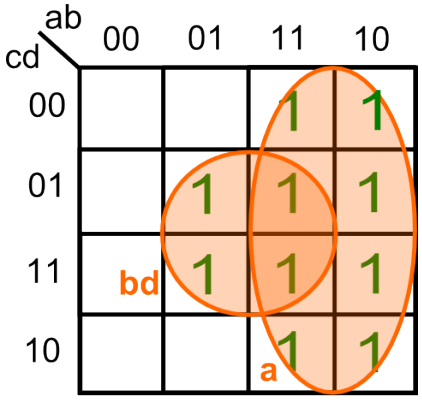

FIGURE 10 – La fonction $F-\{R\}$

2. On calcule le complément de cette fonction par URP (dédié à la complémentation). Cela permet d'identifier les minterms qui ne sont pas compris dans la fonction $F-\{R\}$.

Exemple
Si la fonction $F-\{R\}$ est la suivante :
[11 01 11 01] ⇀ bd
[01 11 11 11] ⇀ a

alors son complément peut être décrite par :
[10 10 11 11] ⇀ $\bar{a}\bar{b}$
[10 11 11 10] ⇀ $\bar{a}\bar{d}$

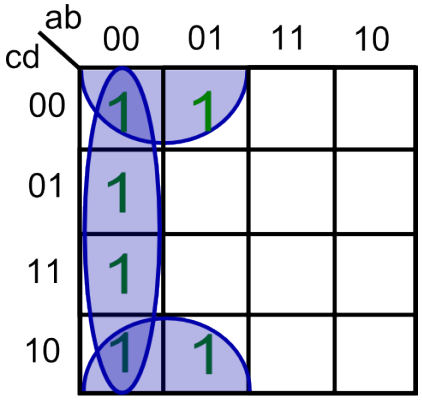

FIGURE 11 – La fonction complémentée de $F-\{R\}$

3. Faire l'intersection de chaque cube de la fonction $F-\{R\}$ complémentée avec l'impliquant R. Cela permet de mettre en évidence les minterms qui sont couvert par R et pas par la fonction $F-\{R\}$. Pour faire l'intersection de deux cubes décrits en PCN, il suffit de faire un ET logique entre les bits de la description. Ainsi l'intersection de [11 10 11 01] avec [01 11 11 01] est [01 10 11 01]. Si jamais vous trouver un

'00' dans le fruit de l'intersection, c'est qu'il n'y a pas d'intersection possible.

Exemple
Le complément de la fonction $F - \{R\}$ comprend 2 cubes :
[10 10 11 11] ⇀ $\bar{a}\bar{b}$
[10 11 11 10] ⇀ $\bar{a}\bar{d}$

L'intersection de l'impliquant R = [11 11 10 11] avec $\bar{a}\bar{b}$ = [10 10 11 11] donne [10 10 10 11].
L'intersection de l'impliquant R = [11 11 10 11] avec $\bar{a}\bar{d}$ = [10 11 11 10] donne [10 11 10 10].

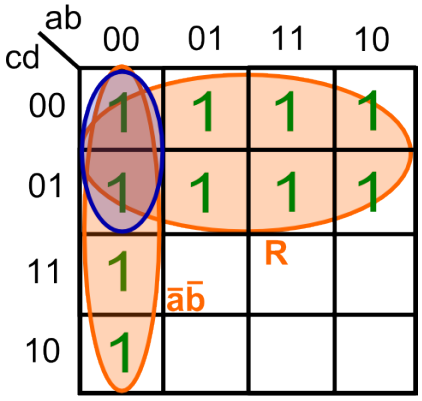

FIGURE 12 – L'intersection entre R et le cube $\bar{a}\bar{b}$ de la fonction complémentée de $F - \{R\}$

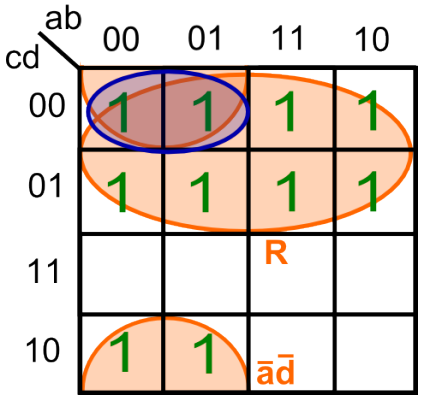

FIGURE 13 – L'intersection entre R et le cube $\bar{a}\bar{d}$ de la fonction complémentée de $F - \{R\}$

4. Faire l'union de tous les cubes qui sont le fruit des intersections. Cela permet de créer un unique cube qui reprend tous les minterms qui sont uniquement couvert par R. Ce cube unique est la forme réduite de R. Pour faire l'union de deux cubes décrits en PCN, il suffit de faire un OU logique entre les bits de la description. Ainsi l'union de [11 10 11 01] avec [01 11 11 01] est [11 11 11 01].

Exemple
L'intersection de R avec la fonction $F - \{R\}$ complémentée donne deux cubes : [10 10 10 11] et [10 11 10 10].
Pour trouver la version réduite de R, il faut réaliser l'union de ces deux cubes. Nous obtenons [10 11 10 11].

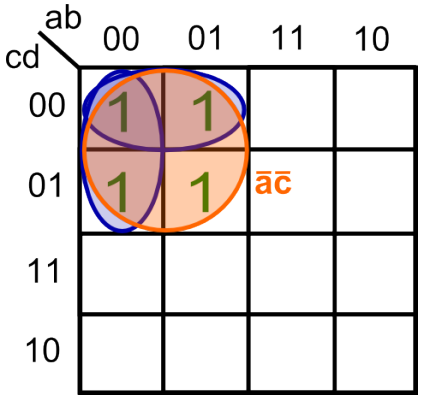

FIGURE 14 – L'union entre les deux cubes issus des intersections

5 L'expansion en détails

La phase d'expansion consiste à agrandir les impliquants d'une fonction de base au maximum pour qu'ils deviennent premiers. Cela signifie donc qu'il faut retirer une ou des variables dans l'impliquant.

Elle comprend les étapes suivantes :

1. On commence par établir la fonction complémentaire (OFF-SET) de la fonction de base (ON-SET) (dans la fonction, les 1 sont remplacés par des 0 et les 0 par des 1). Cette fonction va nous dire où sont les '0' de notre fonction base et donc, là où nous ne pouvons pas étendre nos impliquants. Pour établir la fonction complémentaire, on utilise simplement l'algorithme URP dédié à la complémentation.

Exemple
Soit la fonction suivante : $f = \bar{w}\bar{y}\bar{z} + xz + \bar{x}y\bar{z} + \bar{w}xy\bar{z}$
Le complément de cette fonction sera : $\bar{f} = \bar{x}z + wx\bar{z} + w\bar{y}\bar{z}$

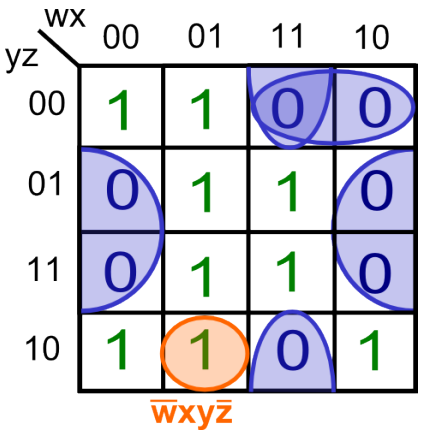

FIGURE 15 – Fonction de départ en vert, fonction complémentaire en bleu et impliquant à expandre en orange

2. On établit ensuite ce qu'on appelle la matrice bloquante (*blocking Matrix*). Elle est construite de la manière suivante : une ligne pour chaque variable (avec sa polarité) de l'impliquant que l'on souhaite étendre, une colonne pour chaque cube de l'OFF-SET. On place un '1' dans la matrice si la variable de la ligne apparait dans une polarité différente que dans le cube colonne et un '0' si ce n'est pas le cas. Cette matrice a la signification suivante : si on retire une variable contenant un '1' pour un cube alors on risque de heurter ce cube. A l'inverse, si on garde une variable contenant un '1' pour un cube, alors on s'assure qu'on ne heurtera pas (jamais) ce cube.

Exemple
Pour l'impliquant $\bar{w}xy\bar{z}$ de la fonction de la figure 15 : une ligne pour chaque variable de l'impliquant $\bar{w}xy\bar{z}$, une colonne pour chaque cube de la fonction complémentaire $\bar{f} = \bar{x}z + wx\bar{z} + w\bar{y}\bar{z}$.

	$\bar{x}z$	$wx\bar{z}$	$w\bar{y}\bar{z}$
$\bar{w}$	0	1	1
x	1	0	0
y	0	0	1
$\bar{z}$	1	0	0

On retrouve la version niée de $\bar{w}$ dans $wx\bar{z}$ et $w\bar{y}\bar{z}$, la version niée de x dans $\bar{x}z$, la version niée de y dans $w\bar{y}\bar{z}$ et la version niée de $\bar{z}$ dans $\bar{x}z$.

3. Finalement, on fait face à un problème de couverture : il faut trouver un set minimal de variables qui contient un '1' pour chaque colonne. En effet, si on garde une variable, on s'assure qu'on ne va pas heurter tous les cubes notés '1' pour cette variable ligne. Si on choisit une combinaison de variable qui couvrent tous les cubes colonnes, on s'assure qu'on ne va heurter aucun cube de la matrice bloquante si on s'expand. On réalise donc le produit des variables présentes dans cette combinaison pour obtenir une expansion possible de l'impliquant de départ.

Exemple
Pour l'impliquant $\bar{w}xy\bar{z}$ de la fonction de la figure 15 :

	$\bar{x}z$	$wx\bar{z}$	$w\bar{y}\bar{z}$
$\bar{w}$	0	1	1
x	1	0	0
y	0	0	1
$\bar{z}$	1	0	0

Les variables $\bar{w}$ et x couvrent tous les cubes colonne, on peut donc expandre l'impliquant $\bar{w}xy\bar{z}$ en $\bar{w}x$. Les variables $\bar{w}$ et $\bar{z}$ couvrent tous les cubes, on peut donc expandre l'impliquant $\bar{w}xy\bar{z}$ en $\bar{w}\bar{z}$.

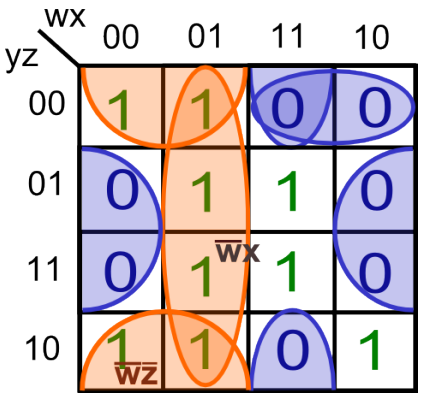

FIGURE 16 – Les deux possibilités d'expansion : $\bar{w}x$ et $\bar{w}\bar{z}$

6 L'irredondance en détails

Dans la partie irredondance, l'algorithme Espresso retire les impliquants ou cubes redondants. Comment le savoir ?

Le principe est assez simple :

Soit un impliquant S, on souhaite vérifier si la fonction F couvre ou non cet impliquant S. On commence par cofactoriser F par rapport à S, cela signifie qu'on cofactorise F en utilisant les variables présentes dans l'impliquant S suivant leur polarité. Pour cofactoriser F par rapport à S, on remplace dans F toutes les variables présentes dans S par 1 ou 0 : par 1 si la variable est présente non niée dans l'impliquant, par 0 si la variable est présente niée dans l'impliquant. Cela permet de cofactoriser F en une fonction FS.

Exemple
Soit la fonction $F = \bar{c}d + ad + \bar{a}bc$ et l'impliquant $S = \bar{a}bd$. On remplace a par 0,b et d par 1 dans la fonction F. On obtient la fonction $FS = \bar{c} + 0 + c = 1$

Si FS vaut 1, cela signifie que la fonction FS vaut 1 pour la même combinaison d'entrées que l'impliquant. La fonction faut donc le job de l'impliquant (imposer un 1 en détection d'une combinaison d'entrée). La présence de l'impliquant dans la fonction n'est pas nécessaire et si on le met, cela crée une redondance.

Exemple
Soit la fonction $F = \bar{c}d + ad + \bar{a}bc$ et l'impliquant $S = \bar{a}bd$. La fonction vaut 1 ($FS = \bar{c} + 0 + c = 1$) et donc l'impliquant $S = \bar{a}bd$ est déjà couvert par F.

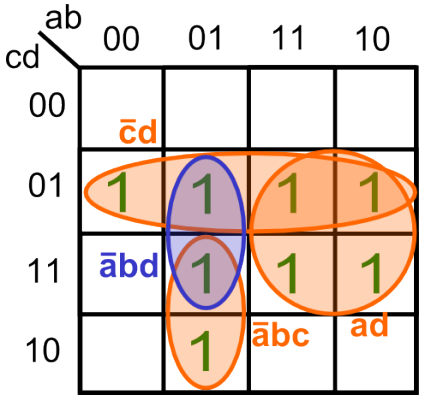

FIGURE 17 – L'ensemble de la fonction en orange couvre bien l'impliquant $S = \bar{a}bd$ en bleu

Si l'on souhaite retirer les cubes redondants d'une liste de cubes, on peut donc procéder de la manière suivante, pour chaque cube :

1. Retirer le cube de la fonction

2. Remplacer par 1 ou 0 les variables du cube en fonction de leur polarité dans la fonction.
3. Vérifier si la fonction vaut 1 (par URP Tautology Checking)
4. Si la fonction vaut 1, le cube est redondant. On ne modifie pas la fonction (on y avait déjà retiré le cube au départ). On passe donc à un autre cube. Si la fonction vaut 0, le cube n'est pas redondant, on le remet dans la fonction.

On conçoit aisément que le résultat de ce genre de méthode va être dépendant de l'ordre dans lequel on travaille. Dans Espresso, un algorithme heuristique est mis en place permettant de chercher un ensemble minimal d'impliquants irredondants.

7 Exercices

1. Unate Recursive Complement Algorithm Le projet consiste à utiliser l'algorithme URP pour déterminer le complément d'une fonction. Voici le fonctionnement de cet algorithme. Chacune de ces lignes sera ensuite expliquée.

```
1  Complement(f in cubelist form, nbrvar){
2  //Vérifier si la liste est vide (f = 0)
3  //Si oui, le complément vaut donc 1 (f'=1)
4  if (f is empty){
5          // f´ est un all don´t care cube dont le nombre de colonne est nbrvar
6          compf = [11 11 ... 11] ;
7          return compf;
8  }
9  //Sinon, vérifier si la liste a un cube rempli de don´t care.
10 //Si oui, la fonction vaut 1, son complément vaut 0
11 else if (f has an all don´t care cube){
12         compf = [empty]; //f´ est vide
13         return compf;
14 }
15 //Sinon, vérifier si la liste contient un seul cube
16 //Si elle l'est, appliquer Demorgan
17 else if (f contains one cube){
18         compf = comp(f);//Déterminé par Demorgan
19         return compf;
20 }
21 //Sinon
22 else{
23    //Choisir la variable à partir de laquelle on va décomposer la fonction.
24    //Chercher s'il y a des variables biformes
25    //Si oui, choisir la variable la plus biforme et la plus balancée
26    if (there are binate variables){
27       x = most binate variable in f;
28    }
29    //Si pas de biforme, choisir la variable monoforme qui apparait
30    //   dans le plus de cube.
31    else {
32      x = most unate variable in f;
33    }
34    //Calculer les cofacteurs selon cette variable x
35    P = positiveCofactor(f,x);
36    N = negativeCofactor(f,x);
37    //Appliquer l'expansion de Shannon en version 'Complement'
38    //f' = x.P'+x'.N'
39    P = Complement(P,nbrvar); //Appel récursif
40    N = Complement(N,nbrvar); //Appel récursif
41    P = AND(x,P); //P =x.P';
42    N = AND(x´,N); // N =x'.N';
43    compf = OR(P,N); //f'= x.P'+ x'.N';
44    return compf;
45 }}
```

Ligne 1 : La fonction *Complement* reçoit en argument une liste de cube et renvoie en sortie une liste de cube représentant la fonction d'entrée complémentée.

Lignes 2 à 8 : Si la liste de cube est vide, c'est que la fonction f vaut 0 . Dès lors, sa version complémentée vaut 1. Pour représenter f' = 1, il suffit de créer une liste contenant un *all don't care cube*. Par exemple, si j'ai une liste de cube contenant 4 variables, ce *all don't care cube* sera [11 11 11 11]. Si j'ai 2 variables, ce *all don't care cube* sera [11 11]. C'est pour cela qu'il faut absolument donner le nombre de variable à la fonction 'Complement'.

Lignes 9 à 14 : Si la fonction comprend un *all don't care cube*, alors c'est que f = 1. En effet, comme un *all don't care cube* représente '1', si j'ai d'autres cubes dans la liste, 1 + ... + ... + ... donnera toujours 1. La fonction complémentée de '1' est f' = 0. Pour représenter ce '0', il suffit de rendre une liste de cube vide.

Lignes 15 à 20 : Si la fonction ne contient qu'un seul cube, on peut la complémenter en utilisant les lois de Demorgan. Par exemple, avec les variables xyzw, le cube [11 01 10 01] représentant $y\bar{z}w$ a pour complément $\bar{y} + z + \bar{w}$ représenté par [11 10 11 11],[11 11 01 11],[11 11 11 10].

Lignes 21 à 22 : Si on ne se trouve pas dans les 3 cas ci-dessus, on va cofactoriser la fonction afin de découper la fonction en deux fonctions plus simples et y appliquer la fonction *Complement*.

Lignes 23 à 30 : Pour diviser en cofacteur, il est idéal de réaliser la cofactorisation sur une variable biforme (qui apparait en version niée et non niée dans la liste de cube) afin que la cofactorisation soit la plus équilibrée (le nombre de cube dans le cofacteur positif et le même que le nombre de cube dans cofacteur négatif). L'idée est qu'il faut mieux couper équilibré afin d'arriver plus vite à des fonctions simples. Pour choisir avec quelle variable réaliser la cofactorisation, vérifier qu'il y a au moins une variable biforme et si c'est le cas, choisir celle qui est le plus biforme (celle qui apparait le plus en version niée ou non niée). Si il y a un litige, choisir celle qui est la plus balancée (|nombre version non niée - nombre version niée| = minimum). Si il y a encore une égalité, prendre la variable dont l'index est le plus faible.

Lignes 31 à 33 : S'il n'y a pas de variable biforme, choisir la variable monoforme qui apparait le plus dans la liste de cube. Si il y a un litige, choisir celle dont l'index est le plus faible.

Lignes 34 à 36 : Calculer les cofacteurs selon la variable déterminée avant.

Lignes 37 à 45 : Pour déterminer le complément, utiliser l'expansion de Shannon en version complémentée :

$$\bar{F} = x.\overline{F_x} + \bar{x}.\overline{F_{\bar{x}}}$$

Pour réaliser l'opération AND entre la variable et la liste de cube, il suffit de remplacer la colonne de la matrice correspondant à la variable par '01' si on ajoute la variable non

niée et par '10' si on ajoute la variable niée. Par exemple, avec les variables xyz, si on réalise l'opération $x.(\bar{y}.z + \bar{z})$ (x . [11 10 01,11 11 01]) on obtient $x.\bar{y}.z + x.\bar{z}$ ([01 10 01,01 11 01]). Toute la colonne de la variable x a été remplacée par '01'.

Pour réaliser l'opération OR entre deux matrices, il suffit de les concaténer (mettre les cubes à la suite des autres).

Avant de vous lancer dans le code, commencer par imaginer le principe que vous allez utiliser pour réaliser chaque étape.

Réaliser la fonction étape par étape en validant chaque étape avant de passer à la suivante. Si vous ne le faites pas, vous risquez d'accumuler les bugs. Pour valider une étape, servez-vous des fichiers *.cubes* (disponible sur Claco) suivant :

- **Vérifier si la liste est vide** : *etape1OK.cubes* (5 variables) est vide et votre fonction doit donc renvoyer [3 3 3 3 3].
- **Vérifier si la liste a un cube rempli de don´t care** : *etape2OK.cubes* (4 variables) contient un 'all don't care cube'. Votre fonction doit donc renvoyer [] (une liste vide).
- **Vérifier si la liste contient un seul cube** : utiliser *etape3OK.cubes*(6 variables). Le résultat doit être :
  [3 1 3 3 3 3]
  [3 3 2 3 3 3]
  [3 3 3 1 3 3]
  [3 3 3 3 3 2]

- **Choisir la variable à partir de laquelle on va décomposer la fonction** :
  - la liste *etape4bif.cubes* (3 variables) est biforme et il faut choisir la variable ayant l'index numéro 2.
  - la liste *etape4mono.cubes* (3 variables) est monoforme et il faut choisir la variable ayant l'index numéro 2.
- **Calculer les cofacteurs selon une variable** : *etape5.cubes* (3 variables) doit être décomposé selon la variable d'index 2. On obtient alors :
  - Cofacteur N :
    [3 3 1]
    [3 3 3]
    [2 3 3]

  - Cofacteur P :
    [1 3 1]

- **Appliquer l'expansion de Shannon en version 'Complement'** : le complement de *etape5.cubes* (3 variables) est :
  [2 1 3]
  [3 1 2]

Les fichiers .cubes contiennent une matrice qui est une liste de cube : chaque ligne de la matrice est un cube.

Pour lire ces fichiers, mettez-les dans le répertoire courant de Matlab et utilisez la commande *dlmread()* comme suit :

```
1 A = dlmread('etape1.cubes');
```

Cette commande va lire la matrice du fichiers etape1.cubes et la mettre dans la variable A.

Lorsque vos étapes sont validées, vous pouvez vous lancer dans l'analyse de fonction plus importante : function1.cubes (6 variables) , function2.cubes (5 variables), function3.cubes (6 variables) et function4.cubes (6 variables). Pour chaque fichier, chronométrez le temps mis par Matlab et essayer d'optimiser votre fonction en utilisant les fonctions tic (remise à zéro du compteur interne de Matlab) et toc (affichage du temps du compteur). Un résultat de complémentation pour chacune de ces fonctions est disponible dans functionXcomp.cubes, où X est le numéro de la fonction.

Une version Opensource de Matlab, **Octave** est disponible sur le serveur *alexandrie* à l'adresse suivante : \\alexandrie\Cours\Outils de calcul numérique 2BE.

Références

[1] Giovanni DeMicheli : *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[2] University of Illinois at Urbana-Champaign Rob A. Rutenbar : Vlsi cad : Logic to layout. *https://www.coursera.org/course/vlsicad*, 2013. En ligne ; consulté le 10/06/2013.

[3] McMullen Sangiovanni-Vincentelli Brayton, Hachtel : *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Press, 1984.

[4] Richard L. Rudell : *Multiple-Valued Logic Minimization for PLA Synthesis*. Memo No. UCB/ERL M86-65, 1986.