Algorithmique et programmation
Travaux pratiques, 18 et 19 janvier 1999

Louis Granboulan

1 Introduction

1.1 Optimiser le calcul d'une fonction booléenne

Étant donnée une fonction binaire f ayant k entrées et n sorties, dont on connait la table de vérité, on aimerait en optimiser le calcul. Ce problème peut être abordé de nombreuses façons...

Si on choisit une base combinatoire, c'est-à-dire quelques opérateurs élémentaires, à 1, 2 ou 3 entrées et une sortie, on peut chercher à minimiser le nombres d'opérateurs pour le calcul de f. C'est le cas pour l'optimisation d'une implantation *bitslice* [1] d'un algorithme de chiffrement. On peut aussi regarder le circuit de calcul de f et chercher à en minimiser la hauteur. Ceci prend en compte la possibilité d'exécuter en parallèle plusieurs opérateurs.

1.2 Opérateurs élémentaires

On peut se limiter aux portes NOT, AND, OR et XOR, qui permettent d'exprimer toutes les fonctions booléennes. On peut aussi rajouter des portes binaires parfois connues par les microprocesseurs récents : NAND, NOR, NXOR, AND-NOT et OR-NOT.

Une alternative est d'exprimer la fonction boléenne à base de multiplexeurs MUX. On rappelle que le multiplexeur est une fonction ternaire qu'on peut définir par : $\mathrm{MUX}(c, b, a) = (c \text{ AND } b) \text{ OR } ((\text{NOT } c) \text{ AND } a)$.

1.3 Mise en œuvre de la recherche

Étant donnée une base combinatoire et un critère d'optimalité, il s'agit de mettre en œuvre la recherche d'un circuit optimal. Limitons nous à la recherche d'un circuit avec un nombre minimal d'opérateur.

Une première technique est la recherche exhaustive. On énumère tous les circuits, par nombre de portes croissant, jusqu'à trouver un circuit qui réalise notre fonction f. Le nombre de circuits à énumérer pouvant être très élevé, la limite de la recherche exhaustive est très vite atteinte. [2]

La seconde approche part d'un circuit qui réalise notre fonction f et modifie ce circuit pour minimiser au mieux le nombre de portes. Il s'agit donc de réécrire une formule en utilisant les règles de l'algèbre booléenne.

[1] Imaginons un algorithme de chiffrement qui transforme un bloc de n bits en un autre bloc de n bits, au moyen d'une clé de m bits. C'est une fonction booléenne à $m + n$ entrées et n sorties. Pour accélérer le chiffrement, la technique dite *bitslice* utilise un microprocesseur comme une machine SIMD exécutant autant d'opérations booléennes en parallèle que la taille des mots du microprocesseur. Cette idée a été introduite par Biham en cryptographie. Voir par exemple `http ://www.darkside.com.au/bitslice/index.html`.

[2] Pour les fonctions d'arité 6 avec une sortie, on peut espérer mettre en œuvre une recherche exhaustive en utilisant un processeur 64-bits qui manipule simultanément les $2^6 = 64$ éléments de la table de vérité. Ceci est difficile et on ne vous demande pas de la faire en deux heures de TP !

2 Optimisation par étapes

2.1 Base MUX

Soit f une fonction binaire d'arité k, la *formule de Shannon* $f(x_0, ..., x_{k-1}) = \mathrm{MUX}(x_0,\ f(1, x_1, ..., x_{k-1}), f(0, x_1, ..., x_{k-1}))$ exprime f à l'aide d'un MUX et de deux fonctions d'arité $k - 1$. Appliquée récursivement, elle permet d'exprimer f par un arbre binaire dont les 2^k feuilles contiennent la table de vérité de f et dont les 2^i MUX de hauteur i sont contrôlés par x_i.

On peut ensuite diminuer le nombre de MUX en regroupant les branches communes de l'arbre. On obtient alors un *diagramme de décision binaire* ou BDD. Voir par exemple la figure 1 ci-dessous. Cette compression donne en théorie un nombre de MUX *quasi optimal*. En réalité, changer l'ordre des variables peut être très utile.

À faire en TP

Le but de cet exercice est de réaliser un logiciel qui calcule le nombre de MUX d'un circuit BDD qui calcule une fonction donnée. On appliquera le logiciel à la formule $(x_1 \cap x_0) \cup (x_1 \cap x_3) \cup (\bar{x}_1 \cap x_2)$ puis à la multiplication modulo 2^8.

Le logiciel comporte un module de manipulation de BDD (définir les types de données et l'interface), un module qui crée l'arbre de Shannon et un module qui le simplifie en un circuit BDD pour diminuer le nombre de MUX.

Réfléchir ensuite aux optimisations possibles.

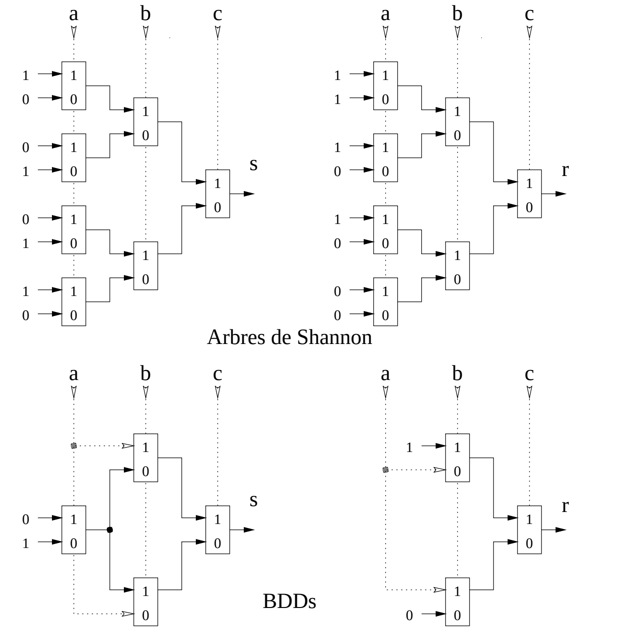

FIG. 1: Additionneur binaire complet : $a + b + c = s + 2r$.

2.2 Autres bases

On pourra par exemple partir d'une forme normale exclusive et utiliser les règles de l'anneau booléen.