Projet Info LDD2 – TD 10

Objectifs du TD : Circuits booléens aléatoires; Additionneur.

Circuit aléatoire

On peut depuis le TD3 générer des graphes aléatoires avec différents paramètres. On peut en particulier générer un graphe dirigé acyclique. Ça n'est malheureusement pas suffisant pour générer un circuit booléen pour trois raisons :

- on peut avoir des nœuds sans parents (ceux-là sont ok si on autorise les constantes 0 et 1, uniquement s'ils ont un unique successeur.)

- on n'a pas assigné d'étiquette aux nœuds

- on peut avoir des nœuds avec plus de 2 parents et plus de 2 enfants

On propose la méthode suivante pour générer un circuit booléen :

1. générer un graphe dirigé acyclique sans inputs ni outputs dans un premier temps

2. ajouter un input vers chaque nœud sans parent, et un output depuis chaque nœud sans enfant

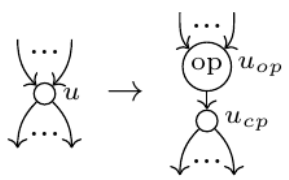
3. pour chaque nœud u dans le graphe ouvert ainsi obtenu :

- si deg+(u)=deg-(u)=1 : donner à u comme label un opérateur unaire

- si deg+(u)=1 et deg-(u)>1 : ne rien faire (le nœud représente une copie)

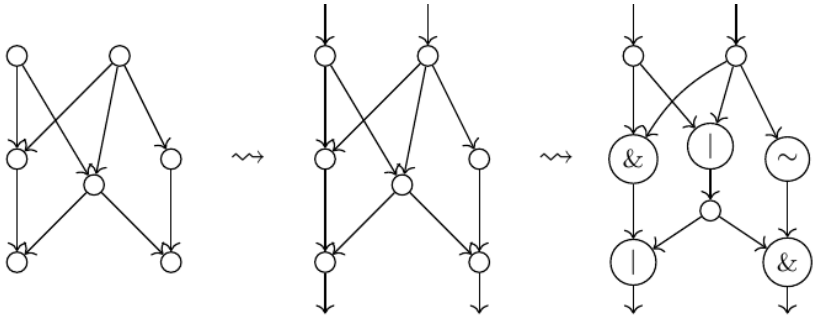
- si deg+(u)>1 et deg-(u)=1 : donner à u comme label un opérateur binaire

- si deg+(u)>1 et deg-(u)>1 : séparer le nœud en 2 nœuds u_op et u_cp tels que : il y ait une flèche de u_op vers u_cp, u_op est pointé par tous les parents de u, et u_cp pointe vers tous les enfants de u. Pour u_op choisir comme label un opérateur binaire :



avec op un opérateur binaire.

Voici par exemple ce qu'on peut obtenir à partir du graphe de gauche :



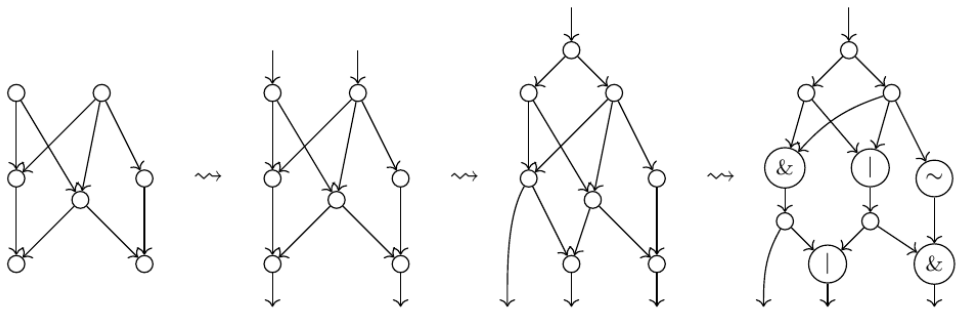
avec les opérateurs tirés aléatoirement parmi une liste donnée de symboles autorisés.

Exercice 1 :

Implémenter une méthode de bool_circ qui génère un circuit booléen aléatoirement (à partir d'un graphe de taille donnée).

On peut avoir envie de spécifier pour notre circuit booléen un nombre particulier d'entrées et de sorties. On peut faire ça en rajoutant une étape 2bis entre 2 et 3 : si le nombre d'entrées voulu est plus grand que celui obtenu, choisir des nœuds au hasard qui vont être pointés par des (nouveaux) inputs ; si le nombre voulu est plus petit, ajouter un nœud qui va relier deux inputs pris au hasard tant que nécessaire. On peut évidemment faire la même chose pour les outputs.

Par exemple, si on voulait 1 entrée et 3 sorties à partir du graphe de départ précédent, on pourrait obtenir :



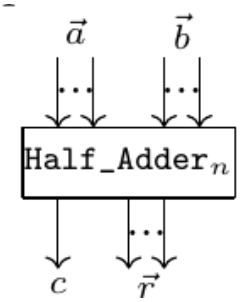
Exercices 2 :

Modifier l'algo précédent pour pouvoir spécifier les nombres d'inputs et d'outputs voulus. On considère que le nombre d'inputs/outputs demandé sera toujours plus grand ou égal à 1.

Additionneur

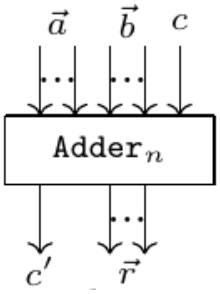
Pour avoir d'autres types de circuits que ceux aléatoires sur lesquels tester nos méthodes, on propose de créer des circuits booléens qui vont calculer la somme de deux registres (de taille une puissance de 2). On va manipuler deux familles de circuits assez proches l'une de l'autre : (Adder_n)_n et (Half_Adder_n)_n.

Half-Adder_n prend deux registres de tailles 2^n et renvoie un registre de taille 2^n (qui contient la somme des deux nombres donnés en entrée, modulo 2^n), plus un bit appelée retenue ("carry"), qui indique si le calcul a dépassé la taille du registre :

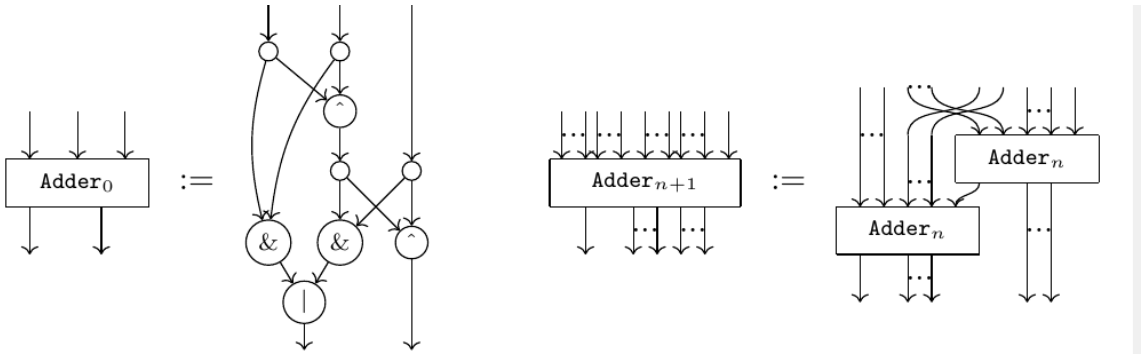


(On considère ici et dans la suite que le bit de poids fort est à gauche.)

Adder_n fait essentiellement la même chose, mais en prenant en compte un bit de retenue additionnel :

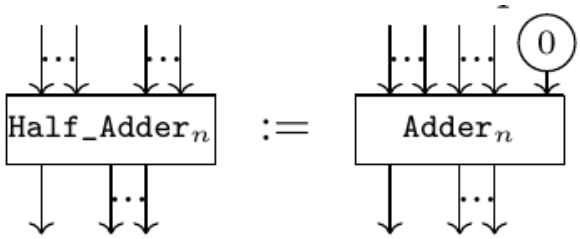


On peut définir ce dernier inductivement :



avec ~ le OU EXCLUSIF.

Et le half-adder d'après le précédent où la retenue est initialisée à 0:



Exercice 3 :

Implémenter deux méthodes de bool_circ qui construisent les deux circuits précédents en fonction de la taille du registre donnée en argument.

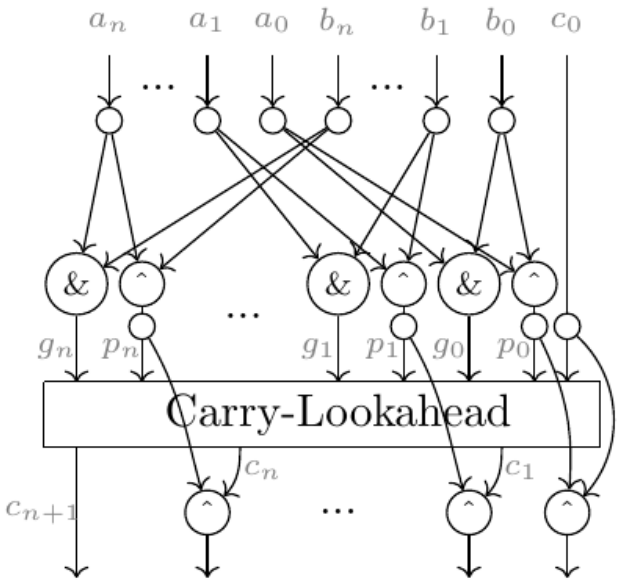
Expérimenter avec les adders, et vérifier qu'ils somment bien deux entiers donnés en représentation binaire.

Bonus

Cette version de l'additionneur a une belle définition inductive, et utilise assez peu de portes (un nombre linéaire en la taille des registres). Par contre, la profondeur est elle aussi linéaire (chaque bit a besoin d'attendre que tous les bits de poids plus faible soient traités avant d'être lui-même traité); alors qu'une approche à base de tables de Karnaugh par exemple, donne quelque chose de profondeur constante (mais au prix d'une explosion a priori du nombre de portes). Il existe plusieurs propositions pour obtenir quelque chose

d'intermédiaire entre les deux. L'une d'entre elles s'appelle le Carry-Lookahead Adder (CLA).

Dans l'additionneur précédent, on voit que chaque bloc à besoin d'attendre la retenue du bloc précédent. L'idée ici va donc être de calculer rapidement ces retenues, avant que le reste du calcul ne soit fini. Ceci va être fait par le bloc carry-lookahead. Chaque pair de bits (a_i,b_i) va créer deux bits g_i et p_i appelés "generate" et "propagate". Ceux-ci vont, avec la première retenue, être utilisés dans le bloc carry-lookahead, pour générer toutes les retenues nécessaires, qui vont être utilisées pour réaliser le calcul~:



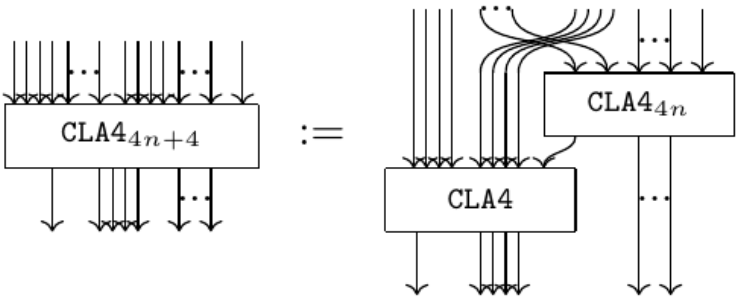
Considérons les valeurs possibles de (a_i,b_i) qui peuvent générer une retenue c_i+1. Si (a_i,b_i)=(1,1), on sait tout de suite qu'une retenue est générée. Si (a_i,b_i) = (0,0), il n'y a pas de retenue générée, quelle que soit la valeur de la retenue c_i. Si (a_i,b_i) = (1,0) ou (0,1), alors c'est la valeur de la retenue c_i qui va déterminer celle de c_i+1. Le bit g_i = a_1&b_i teste le cas de figure (a_i,b_i)=(1,1) (dans ce cas on génère une retenue), et p_i = a_i^b_i teste le dernier cas. Dans ce cas on va propager la retenue précédente dans la nouvelle. On a donc la relation inductive suivante~:

c_i+1 = g_i ^ (p_i & c_i)

avec c_0 qui est donné en entrée. Après distribution, les premières valeurs sont:

— $c_1 = g_0 \hat{} (p_0 \& c_0)$
— $c_2 = g_1 \hat{} (p_1 \& g_0) \hat{} (p_1 \& p_0 \& c_0)$
— $c_3 = g_2 \hat{} (p_2 \& g_1) \hat{} (p_2 \& p_1 \& g_0) \hat{} (p_2 \& p_1 \& p_0 \& c_0)$
— $c_4 = g_3 \hat{} (p_3 \& g_2) \hat{} (p_3 \& p_2 \& g_1) \hat{} (p_3 \& p_2 \& p_1 \& g_0) \hat{} (p_3 \& p_2 \& p_1 \& p_0 \& c_0)$

Le bloc carry-lookahead utilisé dans le circuit précédent construit exactement ça. Une telle famille de circuits a une profondeur constante, mais un nombre de portes qui évolue en O(n^2). Dans les faits, on va donc moduler et créer des blocs qui utilisent le carry-lookahead sur une petite taille de registre (e.g.~4), puis utiliser ce bloc de la même façon que pour le premier additionneur :



Exercice 4 :

Implémenter une méthode (de classe) qui crée un carry-lookahead additionneur sur des registres de taille 4. Implémenter une seconde méthode (de classe) qui utilise ce bloc pour construire un additionneur sur des registres de tailles 4n.

Exercice 5 :

Estimer la profondeur du circuit en fonction de n, et son nombre de portes.