Placement/Routage de Cellules Précaractérisées

Contents

1. Introduction 1
1.1 Circuit **addaccu** 1
1.2 La bibliothèque **sxlib** 2
1.3 Schéma des blocs 3
1.3.1 Multiplexeur 3
1.3.2 Registre 3
1.3.3 Additionneur 3
2. Travail à effectuer 5
2.1 Initialisation de l'Environnement 5
2.2 Générer un Modèle -- **buildModel** 5
2.3 Bloc **mux** 6
2.4 Bloc Registre 6
2.5 Bloc Additionneur 7
2.6 Circuit Addaccu 7
2.7 Description de Patterns 7
2.8 Circuit **addsubaccu** 7
2.9 Bibliothèque DpGen 7
3. Compte Rendu 8

Dans ce TP, nous souhaitons réaliser un générateur de circuit **addaccu** amélioré avec comme paramètre, entre autres, le nombre de bits. Ce générateur sera, dans un premier temps, conçu avec les cellules de **sxlib** , puis avec les générateurs d'opérateurs vectorisés de DpGen.

Nous verrons dans ce TP comment Stratus permet de décrire des netlists paramétrables et de les utiliser. Les netlists seront placés-routés de différentes manières pour montrer l'intérêt du placement procédural.

Note

Documentation de Stratus n'est accessible que depuis l'*intérieur* du département.

1. Introduction

1.1 Circuit addaccu

Dans le circuit **addaccu** sont instanciés trois blocs **mux** , **reg** et **add** .

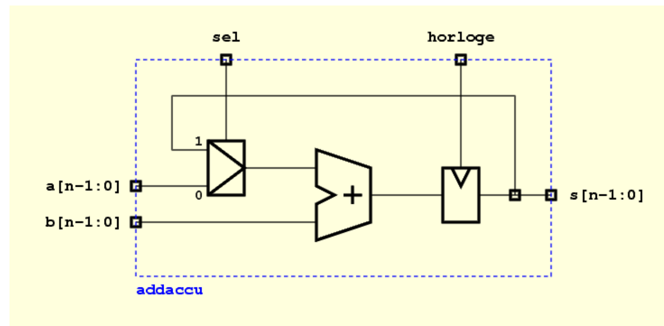

Figure 1: Figure 1 -- Schéma de l'**addaccu**

Les blocs **mux** et **reg** sont des générateurs paramétrables décrits dans le langage Stratus, ce sont des interconnexions de portes de bases, fournies par la bibliothèque de cellules pré-caractérisées **sxlib** . Ils sont une simple *vectorisation* d'une unique cellule (mx2_x2 pour **mux** et sff1_x4 pour **reg**).

Le bloc **add** sera plus complexe car il répète non pas une unique cellule mais le motif plus complexe d'un **full_adder** . Pour simplifier l'écriture de ce bloc on créera une fonction qui instanciera le motif **full_adder** , cette fonction étant ensuite appelée en boucle. Nous parlons bien ici d'une **fonction** et non pas d'un sous-modèle.

Le circuit **addaccu** a donc deux niveaux de hiérarchie.

Note

Nom du signal d'horloge
Pour un circuit aussi petit, nous n'utiliserons pas de stratégie spécifique pour router le signal d'horloge. Pour que ce signal soit routé comme un signal ordinaire, il est nécessaire de lui donner un nom ne contenant pas **ck** , on prendra **horloge** .

1.2 La bibliothèque sxlib

Une cellule pré-caractérisée (en anglais *standard cell* est une fonction élémentaire pour laquelle on dispose des différentes *vues* permettant son utilisation par des outils CAO :

- **Vue physique** : dessin des masques, permettant d'automatiser le placement et le routage.
- **Vue logique** : schéma en transistors permettant la caractérisation (surface, consommation, temps de propagation),
- **Vue comportementale** : description VHDL permettant la simulation logique des circuits utilisant cette bibliothèque.

La bibliothèque de cellules utilisée dans ce TP est la bibliothèque **sxlib** , développée par le laboratoire LIP6 , pour la chaîne de CAO ALLIANCE . La particularité de cette bibliothèque est d'être *portable* : le dessin des masques de fabrication utilise une technique de dessin symbolique, qui permet d'utiliser cette bibliothèque de cellules pour n'importe quel procédé de fabrication CMOS possédant au moins trois niveaux d'interconnexion.

Évidemment, les caractéristiques physiques (surface occupée, temps de propagation) dépendent du procédé de fabrication. Les cellules que vous utiliserez dans ce TP ont été caractérisées pour un procédé de fabrication CMOS 0.35 micron.

La liste des cellules disponibles dans la bibliothèque **sxlib** peut être obtenue en consultant la page man:

```
ego@pc:TME2> man sxlib
```

Comme vous pourrez le constater, il existe plusieurs cellules réalisant la même fonction logique. Les deux cellules na2_x1 et na2_x4 réalisent toutes les deux la fonction NAND à 2

entrées, et ne diffèrent entre elles que par leur puissance électrique : la cellule `na2_x4` est capable de charger une capacité de charge 4 fois plus grande que la cellule `na2_x1`. Évidemment, plus la cellule est puissante, plus la surface de silicium occupée est importante. Vous pouvez visualiser le dessin des masques de ces cellules en utilisant l'éditeur graphique de la chaîne ALLIANCE , **graal** .

1.3 Schéma des blocs

1.3.1 Multiplexeur

Un multiplexeur 4 bits peut être réalisé en utilisant 4 cellules `mx2_x2` suivant le schéma ci-dessous :

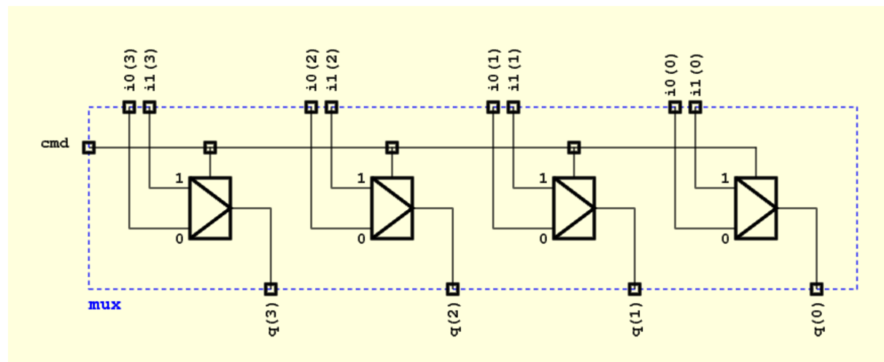

Figure 2: Figure 2 -- Schéma du multiplexeur

Vous pouvez consulter le modèle comportemental de la cellule mx2_x2.vbe.

1.3.2 Registre

Un registre 4 bits peut être réalisé en utilisant 4 cellules `sff1_x4` suivant le schéma ci-dessous:

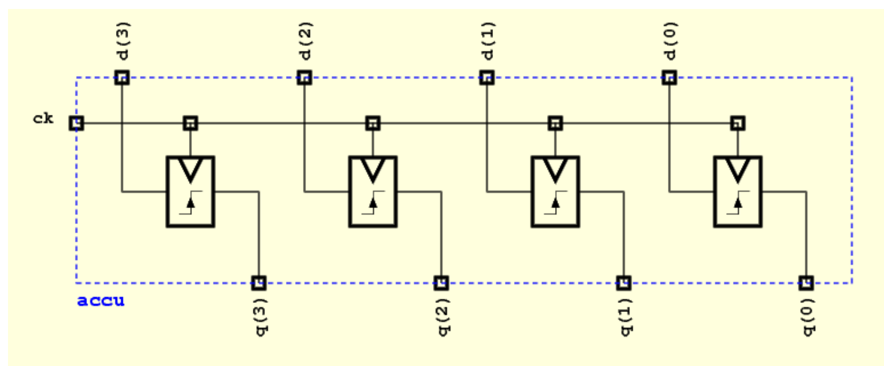

Figure 3: Figure 3 -- Schéma du registre

La cellule `sff1_x4` est une bascule D à échantillonnage sur front montant. Vous pouvez consulter le modèle comportemental de cette cellule : sff1_x4.vbe

1.3.3 Additionneur

Un additionneur 4 bits peut être réalisé en interconnectant 4 additionneurs 1 bit, suivant le schéma ci-dessous:

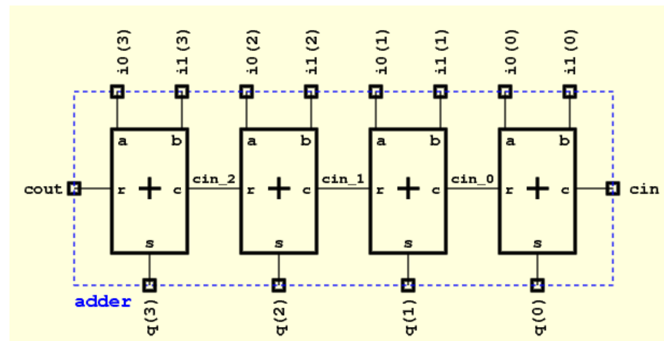

Figure 4: Figure 4 -- Schéma de l'additionneur

Un additionneur 1 bit (encore appelé `Full Adder`) possède 3 entrées a,b,c, et deux sorties s et r. La table de vérité est définie par le tableau ci-dessous. Le bit de *somme* s vaut 1 lorsque le nombre de bits d'entrée égal à 1 est impair. Le bit de *report* est égal à 1 lorsqu'au moins deux bits d'entrée valent 1.

a	b	c	s	r
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

Ceci donne les expressions suivantes:

```
s <= a XOR b XOR c
r <= (a AND b) OR (a AND c) OR (b AND c)
```

Il existe plusieurs schémas possibles pour réaliser un `Full Adder`. Nous vous proposons d'utiliser le schéma ci-dessous, qui utilise trois cellules `na2_x1` (NAND 2 entrées), et deux cellules `xr2_x1` (XOR 2 entrées) :

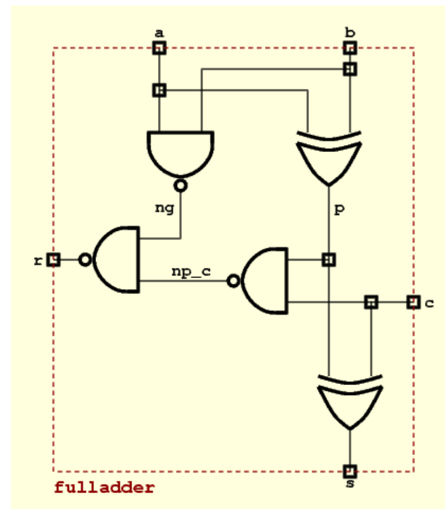

Figure 5: Figure 5 -- Schéma du Full Adder

2. Travail à effectuer

2.1 Initialisation de l'Environnement

Afin de pouvoir travailler avec ALLIANCE et CORIOLIS il vous faut *sourcer* les deux scripts suivant:

```
ego@pc:TME2> . /soc/alliance/etc/profile.d/alc_env.sh
ego@pc:TME2> . /soc/coriolis2/etc/coriolis2/coriolis2.sh
```

Note

Différence entre sourcer et exécuter un script.
Lorsque vous exécutez un script ou un programme, celui-çi va être lancé dans un processus séparé, fils du processus courant. L'environnement du processus fils est une copie de celui du père et les modifications n'affecteront pas le processus parent (i.e. le *shell*).
Lorsque vous sourcez un script, il n'y a pas de création de processus fils, les commandes contenues dans le script sont directement exécutés dans l'environnement courant, exactement comme si elles étaient tapées manuellement au prompt. Elles vont donc modifier l'environnement du *shell*.
La notation `.` est un raccourci pour source en `bash`.

2.2 Générer un Modèle -- `buildModel`

Les scripts que vous allez écrire, vont comporter deux parties :

1. La description du générateur lui-même, sous la forme d'une classe dérivée de `stratus.Model` (avec les surcharges d'une ou plusieurs méthodes `Interface()`, `Netlist()`, `Layout()` et `Pattern()`).
2. L'appel au générateur pour créer une version déterminée du modèle pour un paramétrage donné. Par exemple un modèle de **mux** 32 bits.

La fonction **buildModel** vous permet de réaliser simplement cette opération. Dans le fichier d'exemple fourni, on trouve :

```
buildModel( "mux"          # Nom du générateur.
          , DoNetlist      # flags (vues).
          , modelName="mux_%d"%self.n
          , parameters={ 'nbit':self.n } )
```

Les arguments de **`buildModel`** sont :

1. `"mux"` : le nom du module python dans lequel se trouve le modèle à créer. Un composant *de même* nom doit être défini dans ce module (i.e. `"mux"`).
2. `flags` : quelles vues doivent être générées, parmis `DoNetlist`, `DoLayout` ou `DoPattern`. Il s'agit de *flags* binaires qui peuvent être combinés entre eux (en **OU** bit à bit).

   Le flag `RunSimulator` quand à lui, demande de lancer la simulation avec **`asimut`** (il ne *génére* pas de vue).
3. `modelName="mux_%d"%self.n` : le nom du modèle généré.
4. `parameters={ 'nbit':self.n }` : le dictionnaire de paramètres passé au générateur de modèle. Il se retrouve dans `self._params`

2.3 Bloc mux

- Récupérer et étudier le fichier mux.py écrit avec le language Stratus et décrivant le bloc **`mux`**

  Ce bloc a la fonctionnalité suivante :

  ```
  si (cmd==0) alors s <= i0 sinon s <= i1
  ```

  Les signaux io, i1 et s ayant un nombre de bits paramétrable.

Note

Patterns & Simulation
Les fichiers fournis contiennent aussi la génération des patterns et l'appel au simulateur. Ce point est détaillé en 2.7 Description de Patterns et peut être ignoré ici.

- Créer une instance de mux sur 4 bits.

  Pour ce faire, il faut exécuter le script fourni avec le bon paramètre. Deux méthodes sont possibles :

  1. Soit en faisant exécuter le script par l'interpréteur Python :

     ```
     ego@pc:TME3> python mux.py -n 2
     ```
  2. Soit en appelant directement le script, après avoir changés ses droits pour qu'il soit exécutable:

     ```
     ego@pc:TME3> chmod u+x mux.py
     ego@pc:TME3> ./mux.py -n 2
     ```

  Si le script s'effectue sans erreur, un fichier `.vst` est normalement généré. Vous pouvez vérifier qu'il décrit bien le circuit voulu.

2.4 Bloc Registre

- En s'inspirant du multiplexeur, écrire le bloc **`reg`** avec Stratus en utilisant exclusivement les cellules de la bibliothèque **`sxlib`** . Ce bloc prend lui aussi comme paramètre le nombre de bits. En outre, il vérifie que son paramètre est compris entre 2 et 64 (ce n'est pas fait dans mux).
- Ecrire le script python permettant de créer l'instance du registre.

2.5 Bloc Additionneur

- Écrire le bloc **full_adder** en utilisant exclusivement les cellules de la bibliothèque **sxlib** .
- Écrire le bloc **adder** instanciant le **full_adder** créé. Ce bloc prend lui aussi comme paramètre le nombre de bits et vérifie que son paramètre est compris entre 2 et 64 (ce n'est pas fait dans mux).

2.6 Circuit Addaccu

- Écrire le circuit **addaccu** avec Stratus. Ce circuit instancie les trois blocs précédents (**mux** , **reg** et **adder**). Le circuit **addaccu** prend également comme paramètre le nombre de bits.
- Générer le circuit sur 4 bits.
- Visualiser la netlist obtenue avec **xsch** .

2.7 Description de Patterns

La chaîne de CAO ALLIANCE fournit un outil permettant de décrire des séquences de stimuli : l'outil **genpat** . Stratus fournit le même service pour la chaîne de CAO CORIOLIS . De plus, Stratus encapsule l'appel au simulateur **asimut** .

- Le fichier d'exemple mux.py contient déjà un exemple de génération de *patterns*.
- Créer les patterns et effectuer la simulation des deux autres blocs de la même façon.
- Une fois tous les sous blocs validés, créer les patterns et effectuer la simulation du bloc **addaccu** .

2.8 Circuit addsubaccu

Maintenant, nous souhaitons que l'addaccu puisse effectuer soit des additions, soit des soustractions. Un nouveau paramètre devra donc être ajouté pour choisir la fonction à effectuer (Vous avez le choix pour le nom et les valeurs possibles de ce paramètre). Ce nouveau composant sera sur le même schéma que le précédent, avec des modifications à apporter au circuit et/ou ses composants.

- Créer un nouveau composant, appelé **addsubaccu** qui prend en compte cette nouvelle contrainte.
- Écrire les patterns du composant **addsubaccu** et valider le bloc.

2.9 Bibliothèque DpGen

Une bibliothèque d'opérateurs vectorisés existe en Stratus, la bibliothèque DpGen. C'est une bibliothèque pour chemins de données.

Pour connaitre les générateurs dont vous disposez, consultez la documentation.

- Écrire un nouveau composant **addsubaccu** en utilisant les générateurs paramétrables de cette bibliothèque à votre disposition.
- Les générateurs de DpGen fournissent non seulement une *netlist* mais aussi un placement (une colonne empilée donc chaque *tranche* est associée à un bit). Pour cette version d'**addsubaccu** on créera donc un layout pour le générateur (surcharge de la méthode `Layout()`). Les colonnes des différents opérateurs seront posées côte à côte horizontalement.
- Valider ce bloc avec les mêmes patterns que le bloc précédent.

3. Compte Rendu

Vous rédigerez un compte-rendu d'une page maximum pour ce TME.

- Vous présenterez un schéma de la hiérarchie du circuit **addaccu** .
- Vous expliciterez les choix que vous avez fait pour modifier le circuit **addaccu** et/ou ses composants de façon à créer le circuit **addsubaccu** .
- Vous décrirez quels générateurs de la bibliothèque Dpgen vous avez utilisé et pourquoi.
- Comparer les surfaces et les longueur totale de fils pour les deux versions d'**addsubaccu** (**sxlib** et DpGen). Quelles conclusions peut-on en tirer ?

Vous fournirez tous les fichiers écrits, avec les `Makefile` permettant d'effectuer la génération des deux circuits (et l'effacement des fichiers générés).