TP2 -- Placement/Routage de Cellules Précaractérisées

1. 1. 1 Introduction
      1. 1.1 Circuit addaccu
      2. 1.2 La bibliothèque sxlib
   2. 1.3 Schémas des Blocs
      1. 1.3.1 Multiplexeur
      2. 1.3.2 Registre
      3. 1.3.3 Additionneur
   3. 2 Travail à effectuer
      1. 2.1 Initialisation de l'Environnement
      2. 2.1 Bloc Mux
      3. 2.2 Bloc Registre (Accumulateur)
      4. 2.3 Bloc Additionneur
      5. 2.4 Circuit Addaccu
      6. 2.5 Fonction Generate
      7. 2.6 Description de Patterns
      8. 2.8 Bibliothèque `DpGen`
      9. 2.9 Placement & Routage
   4. 3 Compte-Rendu

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 cellules de `dp_sxlib`.

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`**

Elle est accessible à l'adresse suivante:

?file:///soc/coriolis2/share/doc/coriolis2/en/html/stratus/index.html

Ce lien n'est disponible que depuis l' *intérieur* du département.

1 Introduction

1.1 Circuit addaccu

Dans le circuit **`addaccu`** sont instanciés trois blocs **`mux`**, **`accu`** et **`adder`**.

Les deux blocs **`mux`** et **`accu`** 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`**.

Le bloc **`adder`**, également décrit dans le langage **`Stratus`**, répète un motif **`full_adder`**. Le motif **`full_adder`** sera créé à l'aide d'une fonction `Python`. *Ce n'est pas une instantiation*.

Le circuit **`addaccu`** comporte donc deux niveaux de hiérarchie:

1. - Le niveau des blocs (`mux`, `accu`, `adder`).

2. - Le niveau du circuit complet (`addaccu`).

Nom du signal d'horloge

Note — 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 TME 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 TME 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 de manuel:

```
> 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 Schémas des Blocs

1.3.1 Multiplexeur

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

Vous pouvez consulter le modèle comportemental de la cellule *`mx2_x2`*: 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:

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:

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 *retenue* 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) :

2 Travail à effectuer

2.1 Initialisation de l'Environnement

Afin de pouvoir travailler avec **Alliance** et **Coriolis 2** il vous faut *sourcer* les deux scripts suivant:

```
> . /soc/alliance/etc/alc_env.sh
> . /soc/coriolis2/etc/coriolis2/coriolis2.sh
```

Différence entre *sourcer* et *exécuter* un script.

Note

- 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`.

2.1 Bloc Mux

1. Récupérer les deux fichiers permettant de créer le bloc **mux** et les étudier:
   - Netlist en Stratus du bloc mux
   - Script pour la création de la netlist

Patterns & Simulation

Note — Les fichiers fournis contiennent aussi la génération des *patterns* et l'appel au simulateur. Ce point est détaillé en 2.6 et peut être ignoré ici.

Ce bloc a la fonctionnalité suivante :

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

`i0`, `i1` et `s` ayant un nombre de bit paramétrable

2. Créer une instance de mux sur 4 bits. Pour ce faire, il faut exécuter le script fourni avec le bon paramètre :
   - soit en exécutant la commande suivante :

     ```
     > python gen_mux.py -n 2
     ```
   - soit en modifiant les droits du fichier :

     ```
     > chmod u+x generate_mux.py
     > ./generate_mux.py -n 2
     ```

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

2.2 Bloc Registre (Accumulateur)

- En s'inspirant du multiplexeur, écrire le bloc **`accu`** 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 véri?e que son paramètre est compris entre 2 et 64 (ce n?est pas fait dans mux).
- Écrire le script `Python` permettant de créer l'instance du registre.

2.3 Bloc Additionneur

- Écrire la fonction **`fullAdder()`** en utilisant exclusivement les cellules de la bibliothèque **`SxLib`**.
- Écrire le bloc **`adder`** appelant la fonction **`fullAdder()`** . Ce bloc prend lui aussi comme paramètre le nombre de bits et véri?e que son paramètre est compris entre 2 et 64 (ce n?est pas fait dans **`mux`**).
- Écrire le script `Python` permettant de créer l'instance de l'additionneur.

2.4 Circuit Addaccu

- Ecrire le circuit **`addaccu`** avec **`Stratus`**. Ce circuit instancie les trois blocs précédents (**`mux`**, **`accu`** et **`adder`**). Le circuit **`addaccu`** prend également comme paramètre le nombre de bits.
- Écrire le script python permettant de créer des instances de l'addaccu.
- Écrire un ?chier *`Makefile` paramétrable* permettant de produire chaque composant et le circuit **`addaccu`** en choisissant le nombre de bits.
- Générer le circuit sur 4 bits.
- Visualiser la netlist obtenue avec **`xsch`**.

2.5 Fonction Generate

Il n'est pas toujours très pratique d'avoir à générer avec plusieurs scripts les différents blocs d'un circuit. Le langage **Stratus** fournit donc une alternative: la fonction **Generate**.

Par exemple, pour générer une instance du multiplexeur fourni, il suffit d'ajouter la ligne suivante dans le fichier **addaccu** :

```
Generate ( "mux.mux", "mux_%d" % self.n, param={'nbit':self.n} )
```

Dans cette fonction, le premier argument représente la classe **Stratus** créée (format: `nom_de_fichier.nom_de_classe`), le deuxième argument est le nom du modèle généré, le dernier argument est un dictionnaire initialisant les différents paramètres de cette classe.

- Modifier le fichier décrivant l' **addaccu** et le `Makefile` de façon à pouvoir créer les instances de ce circuit en n'ayant besoin que d'un script.

2.6 Description de Patterns

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

- Récupérer les deux fichiers décrivant le bloc mux avec création du fichier de patterns et simulation, et les étudier :
  - ♦ **mux.py** contient la génération des *patterns*.
  - ♦ **generate_mux.py** contient l'appel au simulateur.
- Créer les patterns et effectuer la simulation des deux autres blocs de la même façon.nce
- Une fois tous les sous blocs validés, créer les patterns et effectuer la simulation du bloc **addaccu**.

2.8 Bibliothèque `DpGen`

Stratus propose aussi une bibliothèque d'opérateurs de chemins de données (*datapath*). Sa documentation est accessible ?ici

- Ré-écrire un **addaccu** paramétrable en utilisant les opérateurs de chemins de données.
- Valider ce bloc avec les mêmes patterns que le bloc précédent.

2.9 Placement & Routage

A l'aide de `cgt`, effectuer un placement/routage des circuits. Pour rendre les différences plus significatives, générer des **addaccu** à 64 bits.

Procéder aux essais suivants:

- Circuit *glue logique* placé/routé avec les paramètres par défauts.
- Circuit *glue logique* avec 10% de marge de surface.
- Circuit *glue logique* avec 10% de marge de surface et le recuit simulé traditionnel activé.
- Circuit *chemin de données*.

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 décrirez quels générateurs de la bibliothèque **`DpGen`** vous avez utilisé et pourquoi.
- Vous commenterez les différence en longueur de fils et surfaces des approches chemins de données et *standard cells*.
- Vous fournirez tous les fichiers écrits, avec les **`Makefile`** permettant d'effectuer la génération des deux circuits (et l'effacement des fichiers générés).