

# TP2 : Modélisation structurelle avec Stratus

1. 1 Introduction
  1. 1.1 Circuit addaccu
  2. 1.2 La bibliothèque sxlib
  3. 1.3 Schéma des blocs
    1. 1.3.1 Multiplexeur
    2. 1.3.2 Registre
    3. 1.3.3 Additionneur
2. 2 Travail à effectuer
  1. 2.1 Initialisation de l'environnement Coriolis 2
  2. 2.2 Bloc mux
  3. 2.3 Bloc registre
  4. 2.4 Bloc additionneur
  5. 2.5 Circuit addaccu
  6. 2.6 Fonction Generate
  7. 2.7 Description de patterns
  8. 2.8 Circuit addsubaccu
  9. 2.8 Bibliothèque dpgen
3. 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 dpsxlib.

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.

Pensez à consulter la documentation de **Stratus** en cas de besoin :

[?http://www-soc.lip6.fr/~jpc/coriolis2/en/html/stratus/index.html](http://www-soc.lip6.fr/~jpc/coriolis2/en/html/stratus/index.html)

## 1 Introduction

### 1.1 Circuit addaccu

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



Les deux 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.

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

### 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:

```
> 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 :



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 : [attachment:sff1\_x4.vbe 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 "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 :

- $s \leq a \text{ XOR } b \text{ XOR } c$
- $r \leq (a \text{ AND } b) \text{ OR } (a \text{ AND } c) \text{ OR } (b \text{ AND } c)$

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 Coriolis 2

Un script vous est fourni pour initialiser l'environnement pour coriolis 2. Il s'exécute de la façon suivante:

```
> . ~/jpc/M2-TOOLS/environment/coriolis2.sh
```

Il va:

1. Positionner les variables d'environnement: `PATH`, `LD_LIBRARY_PATH`, `PYTHONPATH`, ....
2. Copier les fichiers de configuration:
  - ◆ `.environment.alliance.xml`.
  - ◆ `.coriolis2.configuration.xml`.

### 2.2 Bloc mux

- 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

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

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

## 2.3 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.

## 2.4 Bloc additionneur

- Le bloc additionneur de N bits est composé de N **full adder**. Pour éviter une recopie fastidieuse de code, nous allons écrire, dans la description du modèle **adder** une fonction dédiée chargée d'instancier les cellules composant un **full adder**.

```
class adder ( Model ) :

    def fulladder ( self, bit, sig_a, sig_b, sig_cin, sig_s, sig_cout, sig_vdd, sig_vss ) :
        # Ici on instancie les cellules du full adder.
        # [...]
        return

    def Interface ( self ) :
        # [...]
        return

    def Netlist ( self ) :
        # Ici on utilise notre fonction fulladder.
        # [...]
        return
```

Attention, la fonction `fulladder()` ne crée pas de niveau de hiérarchie supplémentaire. C'est simplement une fonction utilitaire permettant de factoriser la création d'un bit de l'additionneur.

- Ecrire la fonction **fulladder** en utilisant exclusivement les cellules de la bibliothèque **sxlib**.
- Ecrire le bloc **adder** appelant la fonction **fulladder**. 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).
- Ecrire le script python permettant de créer l'instance de l'additionneur.

## 2.5 Circuit addaccu

- Ecrire 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.
- Ecrire le script python permettant de créer des instances de l'addaccu.
- Ecrire un fichier **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**.
- Placer & Router en utilisant **cgt**.

## 2.6 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.

## 2.7 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**.

- Récupérer les deux fichiers décrivant le bloc mux avec création du fichier de patterns et simulation, et les étudier :
  - ♦ [attachment:mux2.py Netlist en **Stratus** du bloc **mux** avec la description des patterns]
  - ♦ [attachment:genmux2.py Script pour la création de la netlist, du fichier de patterns et du lancement du simulateur]
- 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 sera donc à apporter 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.
- Ecrire le script python permettant de créer des instances de l'**addsubaccu**.
- Ecrire un fichier **Makefile** paramétrable permettant de produire le circuit **addsubaccu**.
- Ecrire les patterns du composant **addsubaccu** et valider le bloc.

## 2.8 Bibliothèque dpgen

Une bibliothèque d'opérateurs existe en **Stratus**, la bibliothèque **dpgen**.

Pour connaître les générateurs dont vous disposez, consultez la documentation fournie:

[?http://www-soc.lip6.fr/~jpc/coriolis2/en/html/dpgen/index.html](http://www-soc.lip6.fr/~jpc/coriolis2/en/html/dpgen/index.html)

- Ecrire un nouveau composant **addsubaccu** en utilisant les générateurs paramétrables de cette bibliothèque à votre disposition.
- 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 TP.

- Vous présenterez un schéma de la hiérarchie du circuit **addaccu**.
- Vous explicitez 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 **dpngen** vous avez utilisé et pourquoi.

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