ENSSAT - EII3

TP de Systèmes sur Silicium (SOC)

Synthèse d'architecture de circuits intégrés VLSI Algorithmes de multiplication complexe et de filtrage numérique (RIF)

Utilisation de l'outil Behavioral Compiler de Synopsys

Note : n'oubliez pas d'apporter votre cours sur la synthèse de haut niveau, ainsi que votre cours et projet de conception d'ASIC de deuxième année, pendant ce TP.

1. Installation du TP

Modifiez votre fichier **.cshrc** en y écrivant la ligne suivante afin de configurer les outils Synopsys utilisés dans la suite du TP.

```
source /usr/local/synopsys/environ.csh
```

- Copiez le fichier suivant : tp_soc.tar à partir de http://r2d2.enssat.fr/enseignements/Cao/Cao.php
- Décompressez le avec la commande : `tar xfv tp_soc.tar`

Votre répertoire TP_BC de travail contient maintenant les fichiers de configurations de Synopsys, les scripts de synthèse, les sources en VHDL et en SystemC comportemental et RTL.

2. Les fichiers sources VHDL et SystemC

La première partie de ce TP se base sur un algorithme qui réalise la multiplication de deux nombres complexes.
L'algorithme est le suivant :

```
résultat = (a+ib) * (c + id);
         = (ac - bd) + i (ad + bc);
```

Cette opération nécessite 4 multiplications et 2 additions. **Dessinez le graphe flot de données**.

Pour la suite du TP, pensez à suivre les étapes en faisant la conception manuelle de ce graphe flot en même temps que ne le fait l'outil. Vous pouvez vous aider du cours où cet exemple est présenté dans le chapitre sur la synthèse d'architecture.

2.1 SystemC de niveau RTL

Les fichiers **cmult_rtl.cc** et **cmult_rtl.h** dans le répertoire TP_BC/SystemC sont la description RTL

en SystemC de la multiplication complexe. Le script de synthèse RTL est donné dans **cmult_rtl.scr**.

2.2 VHDL de niveau comportemental

Le fichier **multx.vhd** dans le répertoire TP_BC/VHDL contient la description comportementale de la multiplication complexe décrite en VHDL. Le fichier **multx_types_pkgs.vhd** contient tous les types et les constantes dont vous pouvez avoir besoin. Vous pouvez aussi y mettre des fonctions ou procédures.
Ce rattachement se réalise par les commandes ci dessous dans le fichier VHDL.

```
library multx_lib;
use multx_lib.multx_types_pkgs_all;
```

Le script de synthèse d'architecture est donné dans **multx.scr**.

2.3 SystemC de niveau comportemental

Les fichiers **cmult.cc** et **cmult.h** dans le répertoire TP_BC/SystemC sont la description comportementale en SystemC de la multiplication complexe. Le script de synthèse d'architecture est donné dans **cmult.scr**.

Les autres fichiers (**Makefile*, display.*, main.cc, stimulus.***) servent à la simulation comportementale de cet algorithme.

3. Synthèse RTL

1. Editez les fichiers cmult_rtl.cc, cmult_rtl.h et cmult_rtl.scr.
2. Lancez les outils de synthèse par la commande `dc_shell`. Puis lancez le script de synthèse par :
   ```
   dc_shell> include cmult_rtl.scr
   dc_shell> quit
   ```
3. Quelle est la surface et la fréquence maximale de fonctionnement de ce circuit ? Vous pouvez visualiser le résultat avec `design_analyzer`.

4. Simulation comportementale en SystemC

1. Editez éventuellement les fichiers cités dans le 2.3.
2. Lancez la commande `make`. Puis lancez la simulation par `./run.x`
3. Il semble que ca marche! Non ?

5. Synthèse architecturale

5.1 Principe de la synthèse architecturale

La suite de ce TP vous propose d'utiliser soit une description en SystemC soit une description en VHDL. En fonction de votre choix, vous vous placerez soit dans le répertoire TP_BC/SystemC soit dans TP_BC/VHDL.

L'utilitaire de synthèse se lance à partir de la commande **`bc_shell`** ou **`dc_shell`** dans le répertoire de travail choisi. Il est aussi possible de garder une trace du rapport de synthèse. Pour cela, tapez la commande : **`bc_shell | tee rapport1`**. Dans ce cas, une copie de tout ce qui s'écrit à l'écran sera sauvegardée dans le fichier **rapport1**.
Le shell comprend toutes les commandes spécifiques à Synopsys, mais aussi, certaines commandes shell unix classiques tel que cd, ls, history, etc..

La première étape consiste à analyser les fichiers sources pour vérifier les erreurs de syntaxes.

VHDL

On analyse d'abord le fichier *package*.

```
bc_shell> analyze -format vhdl multx_types_pkg.vhd
bc_shell> analyze -format vhdl multx.vhd
```

La deuxième étape va construire notre design à partir du fichier précédemment créé.

```
bc_shell> elaborate -schedule multx
```

multx est le nom donné à l'entité dans le fichier VHDL.
L'option **-schedule** précise que le design est destiné à être ordonnancé. Notons qu'un reset synchrone a été généré automatiquement.

SystemC

```
bc_shell> compile_systemc -cpp "gcc -E" cmult.cc
```

ou si problème avec la version de gcc

```
bc_shell> compile_systemc -cpp "cc -E -xCC -Xc" cmult.cc
```

Vous allez maintenant vérifier si le design est bien ordonnancable. C'est à dire, si les **wait until clk** (ou les **wait()** en systemc) sont bien placés dans le code. En effet, ces synchronisations doivent répondre à des règles bien précises qui sont fonctions du mode d'ordonnancement.
Pour cela, on va d'abord définir la période de l'horloge:

```
bc_shell> create_clock clk -period 1
```

Vous avez défini une période de 1 ns. Vous pouvez maintenant faire votre vérification.

```
bc_shell> bc_check_design -io_mode superstate
```

L'option **-io_mode** définit le mode d'ordonnancement qui est ici **superstate**. C'est le mode le plus courant.

La quatrième étape réalise l'estimation de tous les temps de traversée des ressources utiles. Ce sont ces estimations qui seront utilisées pour ordonnancer les opérations dans les cycles et pour allouer les ressources aux opérations.

```
bc_shell> bc_time_design
bc_shell> report_resource_estimates
```

Quel est la latence d'un multiplieur ?
Quel est la latence d'un additionneur ?
Quel est la latence d'un soustracteur ?

Vous allez maintenant sauvegarder le design. En effet, les étapes précédentes s'étant bien déroulées, il est inutile de les recommencer pour chaque ordonnancement.

```
bc_shell> write -hierarchy -output multx_timed.db
```

Le suffixe **timed** indique que le design a été sauvegardé après l'analyse du timing. La commande `read` permettra de relire le design.
Vous pouvez enfin réaliser l'étape d'ordonnancement :

```
bc_shell> schedule -io_mode superstate
```

L'architecture est créée. Vous pouvez sauvegarder les informations relatives à celle-ci par la commande suivante.

```
bc_shell> report_schedule -variables -operations -summary -abstract_fsm >
archi1_sch.rpt
```

Editez le fichier et répondez aux questions suivantes :

En combien de cycles l'algorithme est il exécuté ?
Quelle est la surface de la partie combinatoire ?
Quelle est la surface de la partie séquentielle ?
Quelle est la surface totale ?
Combien de multiplieurs sont utilisés ?

Nous pouvons maintenant sauvegarder notre design.

```
bc_shell> write -hier -o multx_sch.db
```

Le suffixe est ici **sch** pour un design après ordonnancement. Remarquez que les options peuvent être abrégées. Ainsi **-hier** suffit pour **-hierarchy** et **-o** pour **–output** ;

5.2 BC_view

Il existe aussi une interface graphique pour visualiser les résultats : `bc_view`. Cet outil peut être lancé soit dans bc_shell, soit directement sous unix (il faut dans ce cas avoir créé un projet **multx.proj** par la commande `bc_view -dont_start`).

```
bc_shell> bc_view
```

Au premier lancement l'interface met quelque temps à s'installer, mais par la suite cela sera plus rapide. bc_view est composé de quatre fenêtres :
HDL Browser : contient le code VHDL.
Selection Inspector : contient des informations à propos de l'objet sélectionné.
FSM Viewer : contient le diagramme de la machine d'états.
Reservation Table : indique comment les opérations ont été allouées à chaque cycle.

Cliquez sur la boucle extérieure dans la fenêtre **Reservation Table**. Des informations relatives aux surfaces et à la période de l'horloge apparaissent alors dans la fenêtre **Selection Inspector**.
Remarquez les liens qui existent entre les fenêtres : quand on clique sur un objet dans une fenêtre, les objets correspondant sont en surbrillance dans les autres.
Lorsque vous quitterez **bc_view** appuyez sur **return** pour quitter la fenêtre de lancement de l'interface.

En combien de cycles sont exécutées les différentes opérations ?
En combien de cycles l'algorithme est il exécuté ?
Quel est le pourcentage d'utilisation de l'additionneur ?
Quel est l'optimisation choisie pour l'ordonnancement ?

Une fois que l'on a fini de travailler sur un design il faut l'effacer ou quitter bc_shell.

```
bc_shell> remove_design -designs
```

Vérifiez que vous l'avez bien sauvegardé avant d'exécuter cette commande.

5.3 Le fichier VHDL de niveau RTL généré et la synthèse logique au niveau porte

Relisons notre design.

```
bc_shell> read multx_sch.db
```
Mettons à jour la variable qui indique que le fichier à créer est en VHDL.
```
bc_shell> vhdlout_levelize = true
```
Et créons le fameux fichier :
```
bc_shell> write -hier -f vhdl -o multx_sch.vhd
```
Vous pouvez maintenant examiner ce fichier et comparer le nombre de ligne avec le fichier initiale. Il est à noter que ce fichier est utile principalement pour la simulation. En effet, la poursuite de la synthèse est réalisée à partir du fichier **.db**.

La suite de la conception nécessite de faire la synthèse logique du circuit et d'écrire la netlist de niveau porte aux formats **db** et **VHDL**.

bc_shell> **compile -map_effort medium**
bc_shell> **write -hier -o multx_netlist.db**
bc_shell> **write -hier -f vhdl -o multx_netlist.vhd**
bc_shell> **remove_design -designs**

Vous pouvez visualiser la *netlist* de niveau porte en lançant `design_analyzer&` et en chargeant le fichier **multx_netlist.db** (voir tutorial synopsys de deuxième année).

5.4 Les options de synthèse

5.4.1 Changement de la période de l'horloge

Dans l'architecture précédente, les opérations sont exécutées sur plusieurs cycles. Cela n'offre ici aucun avantage et complique donc inutilement l'unité de contrôle. Vous pouvez redéfinir la période de l'horloge à une valeur supérieure à la latence d'un multiplieur.

```
bc_shell> read multx_timed.db
bc_shell> create_clock clk -period ??
bc_shell> schedule -io super
bc_shell> bc_view
```

Quel est maintenant le nombre de cycles ?
Comparez le temps d'exécution de l'algorithme avec celui de la précédente architecture.

```
bc_shell> remove_design -d
```

5.4.2 Optimisation en surface

Comme vous l'avez sûrement remarqué, l'ordonnancement par défaut est optimisé en vitesse. Il est aussi possible, si on ne possède pas de fortes contraintes de temps de réaliser une optimisation en surface. Ceci est spécifié par l'option **-extend_latency**.

```
bc_shell> read multx_timed.db
bc_shell> schedule -io_mode superstate -extend_latency
bc_shell> write -hier -o multx_area_sch.db
bc_shell> bc_view
```

Quel est maintenant le nombre d'opérateurs utilisés?
En combien de cycles l'algorithme est il exécuté?

```
bc_shell> remove_design -d
```

5.4.3 Compromis temps/surface

L'architecture recherchée est souvent un compromis entre les deux optimisations. En effet, on recherche l'architecture de plus faible surface mais qui exécute l'algorithme dans le temps imparti parle cahier des charges. Pour cela on dispose d'une commande qui permet de fixer le nombre de cycles alloués à l'exécution d'une boucle.

```
bc_shell> read DB/multx_timed.db
bc_shell> set_cycles 17 -from_begin multx/reset_loop/main -to_end
multx/reset_loop/main
bc_shell> schedule -io super -extend_latency
```

En ouvrant **bc_view** vous pouvez constater que la contrainte a bien été appliquée et que le nombre de ressources a varié en conséquence.
reset_loop/main est un nom donné par l'outil pour désigner les différents objets du code. On peut en obtenir la liste complète grâce à la commande suivante.

```
bc_shell> find cell -hier "*"
```

Il est aussi possible de fixer un nombre maximum de ressources. Dans ce cas, l'outil est susceptible de choisir un nombre de cycle inferieur à celui spécifié si il juge cela intéressant:

```
bc_shell> set_max_cycles 14 -from_begin multx/reset_loop/main -to_end
multx/reset_loop/main
```

5.4.4 Les efforts de synthèse

Avec Behavioral Compiler le temps de synthèse est souvent loin d'être négligeable. C'est pourquoi, il peut être intéressant de diminuer ce temps en jouant sur l'effort de synthèse. Cela se fait bien sur au détriment de la qualité des résultats.

```
bc_shell> schedule -io_mode super -effort schedule_effort
```

schedule_effort peut prendre les valeurs **zero, low, medium, high.**

5.5 La synthèse par script

Comme vous vous en êtes aperçu, la synthèse d'une architecture demande de taper un grand nombre de commandes. Cela peut s'avérer très fastidieux lorsque l'on désire générer un grand nombre d'architecture afin de sélectionner à posteriori celle qui nous convient le mieux.
Ouvrez le fichier **script.scr** (ou **cmult.scr** en systemc) et analysez son contenu.
A l'aide de ce fichier on peut lancer une série de synthèse à l'aide de la commande unix suivante.

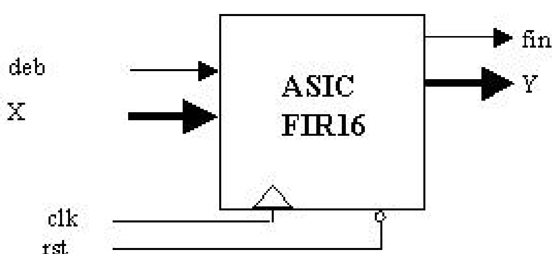
```
bc_shell -f script.scr | tee rapport_script.rpt
```

6. Conception d'un filtre numérique RIF

1. Conception du code comportemental d'un filtre numérique RIF

Cette troisième partie reprend l'étude légèrement modifiée du projet de deuxième année. On considère le filtre numérique à réponse impulsionnelle finie (RIF) de taille 16.

La vue extérieure du circuit est donnée ci contre. On dispose d'une horloge rapide et d'un reset général asynchrone.



Le signal x(n) arrive par un port parallèle, le résultat y(n) est également transmis en parallèle. Ces données sont codées sur 8 bits. La fréquence d'échantillonnage est fixée à 8kHz, elle correspond à l'arrivée des échantillons x(n). Un signal deb respecte cette fréquence et indique qu'un nouvel échantillon est disponible. Un signal *fin* indique que l'ASIC a terminé son calcul. Le protocole de ces 22 signaux est libre.

Vous trouverez dans le répertoire TP_BC 2 fichiers vous aidant à spécifier votre filtre. **test_fir16.vhd** est le *testbench* de simulation. Il permet de vérifier votre code aux niveaux comportemental, RT et porte. **fir16.vhd** est le code comportemental du filtre. La vue extérieure et la boucle principale sont fournies, il ne vous reste plus qu'à compléter le coeur de l'algorithme.

Après avoir complété le fichier, analysez le code, puis vérifiez par simulation le bon fonctionnement du filtre. Vous vous reporterez pour cela au tutorial de l'outil Synopsys.

```
vhdlan fir16.vhd test_fir16.vhd
vhdldbx &
```

Puis on lance la simulation de la configuration **cfg_fir16**.

2. Synthèse de haut niveau du filtre numérique RIF

2.1 Allocation

A l'aide de `bc_shell`, nous devons nous faire une idée du nombre d'opérateurs à utiliser. La contrainte de temps étant fixée à 125us (8kHz). Chargez le code, puis effectuez les premières estimations par les commandes `analyse, elaborate, bc_check_design, bc_time_design`. Vous pouvez également utiliser un fichier *script* fourni effectuant la synthèse totale.

```
bc_shell> include fir16.scr
```

Lancez ensuite `bc_view` pour vous aider à répondre aux questions ci dessous.

Vous remarquerez qu'une contrainte sous forme de *pragma* dans le code VHDL a été donnée à l'outil BC de ne pas dérouler la boucle principale de calcul (**attribute dont_unroll of filtrage_loop : label is true;**)

Quel est la latence d'un multiplieur ?
Quel est la latence d'un additionneur ?
Quel est donc le nombre d'opérateurs nécessaires ?

2.2 Ordonnancement

Faites la synthèse du filtre. Vous utiliserez **-io_mode free_floating** comme mode de compilation ou **-extended_latency**. Si vous utilisez le script l'ordonnancement est lancé automatiquement.

En combien de cycles l'algorithme est il exécuté ?
Quel est le pourcentage d'utilisation des ressources ?

Après analyse des résultats essayez plusieurs optimisations de l'ordonnancement afin d'optimiser l'architecture.

2.3 Synthèse logique et simulation

Utilisez rapidement l'outil `design_analyzer` afin de visualiser la structure de l'architecture. Pour cela charger le fichier **fir16_sch.db**.

> **La structure de l'architecture correspond elle à vos prévisions ?**
> **Quelle est la surface après synthèse logique ?**
> **Quelle est la puissance moyenne dissipée par le circuit ?**

Si vous avez du temps ;-), utilisez l'outil `vhdldbx` afin de valider la structure de l'architecture.