INF3500 - Conception et réalisation de systèmes numériques

Labo 2 - Circuits combinatoires
22 janvier 2020

par Olivier Dion

Table des matières

1. Objectifs

Ce laboratoire sert à confirmer votre compréhension des circuits combinatoires. Le but sera donc d'apprendre à développer un circuit combinatoire synthétisable en **VHDL** avec **Vivado** . Pour ce faire, 5 objectifs sont à accomplir.

- Approfondir la syntaxe et la sémantique de *VHDL*
- Pratiquer la description d'un circuit combinatoire
- Pratiquer la conception d'un module selon le modèle de flot des données
- Implémenter un circuit combinatoire sur une carte **FPGA**
- Apprendre à extraire les ressources utilisées et en faire l'analyse

Vous serez évalué sur ces objectifs ainsi que votre compréhension de l'exercice.

2. Conseils

Avant de commencer, voici quelques conseils.

- Assurez-vous de ne pas travailler dans un dossier temporaire !
- Utiliser **Git** (*GitBash* sur votre poste de travail) pour sauvegarder l'historique de vos travaux. Indexer seulement les fichiers code sources et autre fichier **non binaire**.
- Écrivez toujours le nom des autrices et matricules étudiant aux débuts de vos fichiers en commentaire. Ajoutez optionnellement une licence pour votre code source.
- La synthèse et l'implémentation sont des procédés assez longs. Imaginez un programme `C++` qui compile de 10 à 30 minutes. Relisez-vous pour vous sauver du temps !
- Dans le doute, consultez la **FAQ** du cours (`faq.pdf`).
- Écrivez vos rapports en `LaTeX`. La qualité de ceux-ci sera supérieure.

3. Contexte

Les crypto monnaies, telles que **Bitcoin**, sont des technologies qui agitent souvent les médias, notamment en raison de leur volatilité. D'un point de vue technique, *Bitcoin* est une technologie qui utilise des algorithmes empruntés à la cryptographie, notamment afin de valider les transactions. Dans ce laboratoire on s'intéresse à la fonction de hachage **SHA-256**, utilisées notamment lors de la validation des transactions *Bitcoin*. *SHA-256* est une fonction de hachage de la famille des **SHA-2** ("Secure Hash Algorithm 2"), générant une "empreinte", ou un "hachage", de 256 bits. Ainsi, en appliquant l'algorithme *SHA-256* sur un mot encodé sur une longueur $l \leq 2^{64}$ bits, une empreinte de 256 bits est générée. La spécification complète de l'algorithme *SHA-256* se trouve dans le fichier `fips180-4.pdf`. La lecture de ce document n'est pas requise pour ce laboratoire, mais contient l'ensemble des définitions de l'algorithme.

L'algorithme *SHA-256* calcule l'empreinte d'un mot en plusieurs itérations. À chaque itération, des fonctions dites de "compressions" sont appliquées sur une fraction du mot initial. Dans le cadre de ce laboratoire, vous allez implémenter les fonctions de compressions uniquement. Donc, il n'y a **pas** d'itération à faire pour ce laboratoire. L'implémentation complète de l'algorithme *SHA-256* sera l'objet d'un autre laboratoire.

4. Fonctions de compressions

Vous devez pour cette partie du laboratoire, implémentez 6 fonctions de compression en utilisant le paradigme de description par flot de données.

La table 4.1 énumère ces fonctions. La colonne des noms représente le nom de l'entité qui fait référence à cette fonction en *VHDL*. La colonne des fichiers représente dans quel fichier doit se trouver la fonction. Enfin, la colonne des équations représente les équations binaires de ces fonctions. Aussi, les entrées et sorties de ces fonctions sont sur 32 bits (`std_logic_vector(31 downto 0)`).

NOTE ! La raison pour laquelle on énumère les fonctions *sigma* de cette façon est pour éviter d'utiliser des caractères grecs dans le code source. Donc, rappelez-vous que `sigma0` et `sigma1` sont respectivement $\Sigma_0^{\{256\}}$ et $\Sigma_1^{\{256\}}$, mais que `sigma2` et `sigma3` sont respectivement $\sigma_0^{\{256\}}$ et $\sigma_1^{\{256\}}$

Nom	**Fichier**	**Équation**
ch	`ch.vhd`	$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
maj	`maj.vhd`	$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
sigma0	`sigma0.vhd`	$\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
sigma1	`sigma1.vhd`	$\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
sigma2	`sigma2.vhd`	$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
sigma3	`sigma3.vhd`	$\sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

TABLE 4.1 – Fonctions de compression utilisées dans l'algorithme SHA-256

La table 4.2 vous rappelle les symboles du tableau 4.1. Notez que pour utilisez `rotate_right` et `shift_right`, vous devez importez la library `IEEE` et utiliser le package `NUMERIC_STD.ALL`. Aussi, les paramètres de ces fonctions sont des entiers, et non des signaux. Ainsi, il faut convertir le signal `x` en entier, puis reconvertir le résultat en signal, voir 4.1.

FIGURE 4.1 – Exemple d'utilisation de rotate_right

```
signal bar : std_logic_vector(31 downto 0);
signal foo : std_logic_vector(31 downto 0);

-- rotate_right(bar, 2)
foo <= std_logic_vector(rotate_right(unsigned(bar), 2));
```

Symbole	VHDL
$\neg$	not
$\vee$	or
$\wedge$	and
$\oplus$	xor
$ROTR^n(x)$	`rotate_right(x, n)`
$SHR^n(x)$	`shift_right(x, n)`

TABLE 4.2 – Opérateurs binaires.

Vous devez par la suite implémenter un module `top` qui va cascader vos 6 fonctions pour produire un résultat. Vous devez donc compléter le fichier `top.vhd` et le remettre.

Le module lit en entrée 16 bits (les 16 commutateurs) et écrit en sortie 16 bits (les 16 LED). Vous devez utiliser le fichier `top.xdc` pour votre implémentation et ne **pas** le modifier.

Deux signaux sont mis à votre disposition. `input` qui convertit l'entrée de 16 bits en signal de 32 bits et `output` qui convertit la sortie de vos modules de 32 bits vers la sortie des LED à 16 bits. Vous devez travailler avec ces signaux ainsi que d'autres que vous allez créer. Vous n'avez pas à toucher aux ports.

Le module top est décrit selon la cascade de compression suivante

$input \Rightarrow \sigma_1^{\{256\}} \Rightarrow \sigma_0^{\{256\}} \Rightarrow \Sigma_1^{\{256\}} \Rightarrow \Sigma_0^{\{256\}} \Rightarrow Maj \Rightarrow Ch \Rightarrow output$

Lorsque plusieurs entrées sont requises à une fonction, réutilisez plusieurs fois la sortie de la fonction précédente.

Enfin, voici un exemple de comment importer un module dans `top.vhd` pour vous aider.

```
CH_MODULE: entity work.ch
port map (
  x => input,
  y => input,
  z => input,
  o => output
);
```

```
SIGMA0_MODULE: entity work.sigma0
port map (
  x  => input, -- Map input to x in instance SIGMA0_MODULE
  o  => open   -- Output of SIGMA0_MODULE is left opened
);
```

Voyons cela en détail. `CH_MODULE:` est juste une étiquette, ce qui nous intéresse est `entity work.ch`. Cela ce traduit par instancier une entité de type `ch` dans `work`. Notez qu' ici `work` ne varie jamais, mais `ch` pourrait être une autre entité. Enfin on a le mappage avec `port map`. Les variables à gauche doivent correspondent aux noms des ports de l'entité et doivent être relier à l'aide du symbole `=>` à un signal dans le module.

Donc on peut résumer l'exemple précédent comme suit :

Reliez le signal `input` du module `top` aux ports d'entrées des instances `CH_MODULE` et `SIGMA0_MODULE` dont les types sont respectivement l'entité `ch` et `sigma0`. Enfin, reliez la sortie de l'instance `CH_MODULE` vers le signal `output` et laisser la sortie de l'instance `SIGMA0_MODULE` ouverte (`open` est un mot réservé en *VHDL*).

4.1 Livrables

- ch.vhd
- maj.vhd
- sigma0.vhd
- sigma1.vhd
- sigma2.vhd
- sigma3.vhd
- top.vhd

5. Simulation

Un banc d'essai vous est fourni dans `top-tb.vhd`. Utilisez-le pour vérifier que votre module `top` marche comme il le faut.

Regardez la `Tcl Console` pour voir les résultats du banc d'essai.

6. Synthèse et implémentation

Faites la synthèse du module `top` puis son implémentation à l'aide du fichier `top.xdc`. Demandez par la suite à ce que l'on vienne vous évaluer.

6.1 Livrables

- top.bit

7. Discussion

Générez l'estimation des ressources utilisées après la synthèse. Aidez-vous de la `faq.pdf` au besoin. Vous devez inclure une figure de l'estimation des ressources dans votre rapport et discuter des résultats.

8. Questions

Ces questions sont à répondre dans votre rapport.

a) (0.5) La fonction `maj` est le préfix de **maj**ority. Expliquez de façon concrète et concise pour quelle raison cette fonction est appelée ainsi dans le contexte où les entrées sont des vecteurs de bits (`std_logic_vector`).

b) (1.5) Donnez un avantage et un désavantage de tester les 6 fonctions de compression du `SHA-256` en utilisant un *pipeline* comme celui du module `top`.

9. Rapport

Vous devez écrire un rapport selon les directives dans le fichier `rapport.pdf`.

Seuls les fichiers de type `pdf`, `DjVu` et `ps` sont acceptés pour le rapport. Aucun fichier de type Word (doc, docx) n'est accepté.

Assurez-vous d'inclure tous les fichiers listés dans les sections `Livrable(s)`.

10. Barème

Critères	**Points**
Partie 4 : Conception des modules	7
Partie 6 : Synthèse et implémentation	7
Partie 8 : Questions	2
Discussion	2
Rapport : Présentation et qualité de la langue	2
Total	20