ELECINF102
Processeurs et Architectures Numériques

Contrôle de connaissances

Vendredi 13 juin 2014 à 8h30

Document autorisé : une feuille recto-verso

Durée: 1h30 minutes

Ce contrôle comporte trois parties **indépendantes** :

1. Une horloge avec compensation de dérive en fréquence
2. Un multiplieur itératif
3. Un compilateur C pour le nanoprocesseur

Consignes importantes : Si des **schémas** sont demandés dans les différents exercices, ils doivent être impérativement clairs, lisibles et sans ambiguïté. Les dimensions des bus doivent être indiquées. Si nécessaire le sens des signaux doit être précisé. Pour la logique synchrone, les signaux d'horloge et d'initialisation asynchrone (`reset_n`) ne seront pas représentés dans ces schémas.

N'oubliez pas d'inscrire nom, prénom, et numéro de casier sur votre copie.

Bon courage !

1 Horloge avec compensation de dérive en fréquence

Pour répondre aux questions de cette exercice vous pouvez au choix :
- *faire des schémas en utilisant les symboles de portes et opérateurs vus en cours,*
- *écrire du code SystemVerilog.*

Nous voulons concevoir un chronomètre comptant les secondes, nous disposons pour cela d'un oscillateur à quartz générant une horloge principale à une fréquence nominale $F = 32\,768\,\text{Hz}$ à une température $T = 25°$.

Question 1 : À cette fréquence nominale, combien de bits sont nécessaires pour compter durant une seconde ?
Question 2 : Proposez alors un système de division de fréquence générant une impulsion toute les secondes. La durée de cette impulsion doit être exactement une période de l'horloge principale.

En pratique, la fréquence de l'oscillateur varie légèrement avec la température. Dans des conditions normales d'utilisation, la valeur de la fréquence peut devenir $F' = (1 + \Delta)F$ où $\Delta \in [-10^{-3} : 10^{-3}]$. La dérive en fréquence Δ a été caractérisée. Il s'agit donc d'une constante connue. Par ailleurs, le chronomètre dispose d'un capteur de température, et nous voulons ajouter un mécanisme pour compenser ces variations et toujours obtenir une durée constante d'une seconde. Il faut alors modifier le compteur du diviseur de fréquence pour prendre en compte cette dérive. La valeur maximale du compteur C_{max} doit pouvoir être changée en suivant la formule :

$$C_{max} = \left\lfloor \frac{32\,768}{1 + \Delta} \right\rfloor$$

où $\lfloor x \rfloor$ représente la partie entière de x. Cette valeur sera calculée à l'extérieur de notre système et lui sera transmise en entrée pour faire varier le nombre de cycles comptés dans le diviseur de fréquence.

Question 3 : Combien de bits sont maintenant nécessaires pour le compteur du diviseur de fréquence ?
Question 4 : Modifiez le système de division de fréquence pour prendre en compte cette valeur d'arrêt variable.

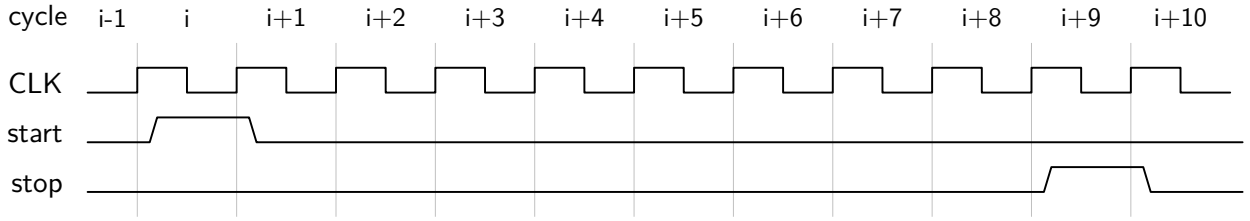
2 Un multiplieur itératif

Nous souhaitons construire un module de calcul **Operateur**, disposant d'une entrée **start**, d'une horloge **clk** et d'un sortie **stop**, dont les rôles sont :

- le signal **start** signale le démarrage d'un calcul à effectuer par **Operateur** ;
- le signal **start** est une impulsion d'une durée d'un cycle de l'horloge ;
- le signal **stop** généré par **Operateur** signale la fin du calcul ;
- le signal **stop** est une impulsion d'une durée d'un cycle de l'horloge ;
- le signal **stop** doit avoir un retard de 9 cycles par rapport au signal de départ ;
- si un nouveau **start** intervient avant l'arrivée de **stop**, alors la procédure en cours est réinitialisée pour générer **stop** 9 cycles plus tard.

On suppose qu'initialement toutes les bascules D sont à '0'.

Le chronogramme suivant illustre le fonctionnement d'**Operateur** :



Question 1 : Déterminez une architecture de traitement pour **Operateur** et complétez le code SystemVerilog correspondant :

```
module Operateur(
                input logic clk,
                input logic start,
                output logic stop
                );

// code a compléter

endmodule
```

Operateur doit réaliser le produit **P** de deux nombres **A** et **B** de manière itérative en 9 cycles d'horloge. On considère que :

- les nombres **A** et **B** sont deux entiers non signés codés sur 8 bits ;
- les nombres **A** et **B** sont fournis à **Operateur** lorsque **start** vaut '1' ;
- les nombres **A** et **B** sont maintenus pendant toute la durée du calcul ;
- le nombre **P** est un entier codé sur 16 bits ;
- le résultat **P** doit être généré par **Operateur** lorsque **stop** vaut '1'.

L'algorithme de calcul (en pseudo langage) est le suivant

```
P = 0
tmpA = A
pour i allant de 0 jusqu'à 7 faire
debut
  si B[i] = 1 alors P = P + tmpA
  tmpA = 2*tmpA
fin
```

Question 2 : Complétez le code SystemVerilog précédent pour obtenir le comportement souhaité. N'oubliez pas de préciser la taille (nombre de bits) des signaux internes éventuellement créés. On rappelle qu'en SystemVerilog on peut accéder au bit `i` du bus `A` par la syntaxe `A[i]`.

```
module Operateur(
                input logic clk,
                input logic start,
                input logic [7:0] A ,
                input logic [7:0] B ,
                output logic stop,
                output logic [15:0] P
                );

// code a compléter

endmodule
```

Question 3 : Modifiez **Operateur** pour qu'il fonctionne en 8 cycles.

3 Un compilateur C pour le nanoprocesseur

Dans cet exercice nous allons travailler sur le nano-processeur vu en cours et en TP. Les programmes à éxecuter sont stockés en mémoire en créant deux zones bien distinctes :

- Les instructions sont placées dans une première zone de la mémoire.
- Les données sont placées dans une deuxième zone de la mémoire.

Question 1 : Quels sont les avantages d'avoir des zones séparées pour les instructions et pour les données ?

Dans la suite de l'exercice les zones "instructions" et "données" seront définies de la façon suivante :

- La mémoire contient 256 octets.
- Les instructions sont placées dans la plage d'adresses de 0 à 127.
- Les données sont placées dans la plage d'adresses de 128 à 255.

Langage utilisé pour décrire un programme éxécuté par le nano-processeur s'appelle langage **assembleur**. Le langage assembleur n'est que très rarement utilisé en pratique. On préfère utiliser des langages de plus haut niveau comme C, C++ ou Java. C'est un compilateur qui se charge de traduire les instructions C en instructions assembleur compréhensibles par le processeur.

Question 2 : En s'inspirant des exemples situés en annexe, traduisez en assembleur le code C suivant. Vous préciserez le contenu de chaque adresse mémoire utilisée, en zone instructions et en zone données. Les types **char** du langage C sont des mots de 8 bits.

```
char a = 13;
char b = 127;
char c;

...
c = a + b;
...
```

Question 3 : Traduisez en assembleur le code C suivant. Vous préciserez le contenu de chaque adresse mémoire utilisée, en zone instructions et en zone données.

```
char a, b, c, d, e, f;

...
 if (a == b)
    c = d + e;
 else
    c = d + f;
...
```

Question 4 : En utilisant le jeu d'instructions vu en cours, traduisez en assembleur le code C suivant. Vous préciserez le contenu de chaque adresse mémoire utilisée, en zone instructions et en zone données.

```
char a, b, c, d, e, f;

...
 if (a < b)
    c = d + e;
 else
    c = d + f;
...
```

Question 5 : On cherche à faire fonctionner le processeur à une vitesse supérieure à sa vitesse limite, sans changer de technologie ni de tension d'alimentation. Comment procéder ? Quels sont les avantages et inconvénients de cette méthode ?
Question 6 : On réduit maintenant la taille des transistors d'un facteur α, tout en gardant une tension d'alimentation constante. Comment évolue la vitesse maximale de fontionnement du processeur ? Que devient sa consommation à cette vitesse maximale ?

Annexe :

Nous rappelons dans le tableau suivant la liste des instructions du nano-processeur

Inst.	Fonction	Inst.	Fonction
NOP	Ne fait rien	ROL	Rotation à gauche
XOR	Ou exclusif	ROR	Rotation à droite
AND	Et	LDA	Transfert mémoire vers accumulateur
OR	Ou	STA	Transfert accumulateur vers mémoire
ADD	Addition	OUT	Positionne le port de sortie
ADC	Addition avec retenue entrante	JMP	Saut inconditionnel
SUB	Soustraction	JNC	Saut si le dernier calcul a généré une retenue sortante
SBC	Soustraction avec retenue entrante	JNZ	Saut si le dernier calcul a généré un résultat non nul

Un programme en langage **assembleur** peut s'écrire de la façon suivante :

- Chaque **instruction** du programme est décrit par 3 champs :
  - <adresse de l'instruction> <instruction> <adresse de l'opérande>
- Chaque **donnée** du programme est décrite par 2 champs :
  - <adresse de la donnée> <valeur de la donnée>

Ainsi la portion de programme **assembleur** suivant :

```
...
34 XOR 141
36 AND 212
...
141 23
...
212 56
...
```

Doit être interprétée de la façon suivante :

- A l'adresse **34** se trouve l'**instruction XOR**
- A l'adresse **35** se trouve l'adresse de l'opérande (**141**)
- A l'adresse **141** se trouve une **donnée** de valeur **23**.
- Le microprocesseur doit calculer le "ou-exclusif" entre la valeur courante de l'accumulateur et la valeur 23 et stocker le résultat dans l'accumulateur.
- A l'adresse **36** se trouve l'**instruction** suivante : **AND**
- A l'adresse **37** se trouve l'adresse de l'opérande : (**212**)
- A l'adresse **212** se trouve une **donnée** de valeur **56**.
- Le microprocesseur doit calculer le "et" entre la valeur courante de l'accumulateur et la valeur 56 et stocker le résultat dans l'accumulateur