TP 5 : Fonctionnement d'un processeur

Il est impératif d'avoir effectué le travail préparatoire avant de venir en séance.

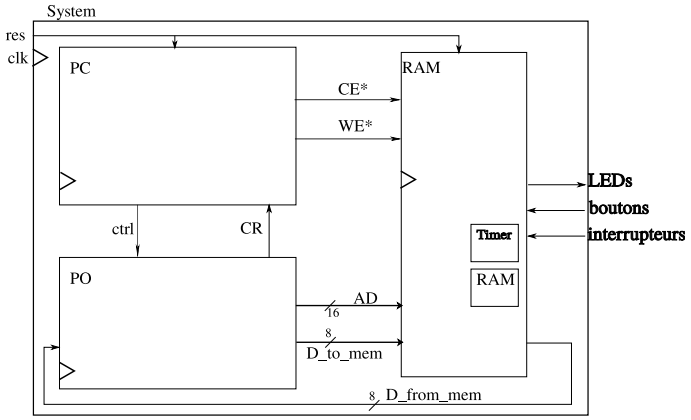
Le but de ce TP est de comprendre le fonctionnement d'un processeur très simple (celui vu en TD) et plus précisément :

— comment fonctionne sa mécanique interne (PC et PO),
— comment on peut le contrôler via le jeu d'instructions,
— comment l'intégrer dans un système.

Préparation
Ex. 1 : Travail préparatoire

Repartons de l'environnement du processeur vu en TD (8 à 11). Pour interagir avec le processeur, il faut ajouter à l'environnement des périphériques (LED, interrupteurs...). Notre processeur peut accéder à ses périphériques en utilisant les instructions d'accès mémoire (ld, st) dans une plage d'adresses dédiée aux périphériques. Vous trouverez en annexe le fonctionnement de ces périphériques et les adresses associées. Par exemple, on peut y constater que le processeur de ce TP peut accéder à de la mémoire sur la plage d'adresse $[0x4000 - 0x4FFF]$ et aux LED en écrivant à l'adresse $0x1$.

L'environnement du processeur complètement défini est représenté ci-dessous :



Question 1 En utilisant la table d'allocation des périphériques donnée en annexe, écrivez en langage d'assemblage un programme recopiant en boucle sur les LED la valeur présente sur les interrupteurs.

Ex. 2 : Simulation comportementale du processeur

La mémoire RAM est initialisée avec le contenu du fichier `boot_defaut.mem`. Le fichier fourni contient le programme vu durant le TD10 et calculant $A = 2 \times (B + C) - 18$.

Question 1 Ouvrez le fichier `boot_defaut.mem` pour en déduire comment y est stocké un programme.

Dans le répertoire du projet, lancez la simulation avec `make run_simu`. La simulation repose sur le banc de test `tb_system.vhd`, que vous pouvez regarder.

Question 2 Ajoutez à votre simulation les registres généraux (r0,r1,r2,r3) de la PO, puis relancez la simulation pour suivre l'évolution des données dans les registres au cours du calcul. Pour rappel, on ajoute des signaux à votre simulation en sélectionnant dans l'onglet « *Scope* » le composant contenant le signal recherché. Tous les signaux de ce composant apparaissent dans l'onglet « *Objects* ». Il suffit alors de faire glisser le signal désiré vers le chronogramme.

Question 3 Sans fermer le simulateur, ajoutez les registres PC et IR de la PO et le signal état courant de la PC. Relancez la simulation pour suivre l'évolution de ces signaux au cours du calcul. Pour conserver un chronogramme lisible, pensez à regrouper les signaux par thématique et à utiliser des séparateurs (clic droit dans le chronogramme et « *New Divider* »).

Question 4 Sauvegardez la configuration de votre simulation (Ctrl+S) dans le fichier `magic.wcfg`. Lors de vos prochaines simulations, la configuration sera automatiquement chargée.

Question 5 Modifiez le programme inclus dans le fichier `boot_defaut.mem` de manière à écrire le résultat sur les LED. Relancer la simulation et vérifier que la sortie LED du chronogramme est correcte (Pour vérifier les 8 bits du résultat vous pouvez observer le signal sLED du composant periph.)

Question 6 Que fait le processeur à la fin du programme ? Proposez une modification du programme afin de résoudre le problème et relancez le simulateur pour la tester.

Ex. 3 : Exécution du programme sur la carte FPGA

Question 1 Implantez sur la carte le processeur avec le programme de la question précédente en utilisant la commande :

```
$ make run_fpga
```

Comme il est fastidieux d'écrire un programme en hexadécimal dans le fichier `boot_defaut.mem`, nous utiliserons par la suite un assembleur qui se chargera de générer ce fichier depuis un fichier contenant le programme en langage d'assemblage. Pour générer le fichier `boot_defaut.mem` correspondant à votr programme, il vous suffit d'utiliser la commande :

```
$ make boot_defaut.mem PROG=nom
```

après avoir écrit votre programme dans le fichier `src/nom.as`.

Il vous suffira ensuite de programmer le FPGA avec la commande : `$ make run_fpga`

Question 2 Transcrivez votre travail préparatoire dans un fichier `src/recopie.as` et testez-le sur carte.

Question 3 Regardez le programme fourni `src/exo2.as`. En utilisant les annexes, prévoyez le fonctionnement de ce programme. Lancez ensuite une simulation pour déterminer le nombre de cycles d'horloge entre 2 passages dans la boucle d'attente, que vous comparerez à la taille par défaut de l'impulsion (Veillez à visualiser l'horloge du processeur et non celle fournie par la carte). Implantez ce programme sur la carte et vérifiez votre prédiction.

Question 4 En réutilisant la boucle d'attente du programme précédent, écrivez puis testez sur carte un programme en langage d'assemblage qui fait clignoter toutes les LED à 1Hz (c'est-à-dire que les LEDs restent allumées pendant 500ms, et ensuite elles sont éteintes pendant 500ms, etc.) Si ça ne marche pas comme vous l'espériez, vérifiez en simulation que votre programme est correct. Conseil : désactivez dans ce cas la boucle d'attente en changeant la destination du saut de rebouclage.

L'algorithme ci-contre permet de réaliser un chenillard sur un afficheur à 4 LED. C'est-à-dire que les LED allumées se décalent d'une position à intervalles réguliers (toutes les 0,5s pour notre exemple).

Question 5 Transcrivez cet algorithme en langage d'assemblage puis testez le sur carte.

```
motif ← 0x01;
while true do
    repeat
        timer ← lire_timer();
    until timer = 0;
    afficher(motif);
    motif ← motif ≪ 1;
    if (motif and 0x10) ≠ 0
     then
        motif ← motif + 1;
    end if
end while
```

Pour aller plus loin...

Question 6 Repartons du programme exo2. Ce programme fonctionne, car la durée de la boucle d'attente est exactement égale à la durée d'une impulsion et que la période du timer est bien plus grande que le temps nécessaire pour réaliser la fonction. Modifiez la durée de l'impulsion pour voir l'effet produit sur le programme. Qu'observez vous en réduisant la taille de l'impulsion ? en l'augmentant ? Justifiez.

Pour aller plus loin...

Question 7 Quelques idées de programmes :

a) Coder une multiplication.

b) Coder un compteur qui affiche le résultat de plus en plus lentement (en jouant sur la période du timer).

Pour aller plus loin...

Question 8 Avant de partir, pourquoi ne pas jeter un œil aux fichiers VHDL décrivant la PC, la PO et les périphériques. Le codage des états de la PC est par exemple très instructif.

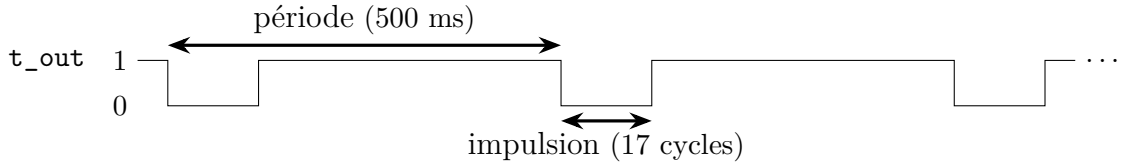
Annexes

Table des périphériques :

Périphérique	Accès	Adresse	Action
LEDs	écriture	0x0001	Affiche l'octet écrit par le processeur sur les 4 LED pour voir les 4 MSB, appuyer sur le BTN1
Timer	lecture	0x0010	récupère la valeur du timer (actif à 0)
	écriture	0x0020	Change la période du timer par pas de 13ms (défaut 500ms)
	écriture	0x0040	Change la durée de l'impulsion du timer (défaut 17 cycles)
Interrupteurs	lecture	0x0100	récupère la valeur sur les 4 interrupteurs.
Boutons poussoir	lecture	0x0200	récupère la valeur sur les 3 boutons poussoirs (BTN3,BTN2,BTN1)
Mémoire	lecture écriture	0x1000 – 0x1FFF	plage mémoire décrite après @0 dans le fichier `boot_defaut.mem` réservée pour les données
Mémoire	lecture écriture	0x4000 – 0x4FFF	plage mémoire décrite après @1000 dans le fichier `boot_defaut.mem` réservée pour les instructions

Fonctionnement détaillé du timer :

Un timer est un composant matériel qui permet d'avoir une notion de temps dans nos programmes. Le timer peut être vu comme une boîte qui a une seule sortie `t_out`. `t_out` vaut 0 ou 1, en suivant le chronogramme ci-dessous. On appelle *impulsion* le moment pendant lequel la sortie `t_out` est à 0. La période et l'impulsion sont paramétrables. Sauf extension, vous n'aurez pas à modifier les valeurs par défaut (500ms, 17 cycles).



Jeu d'instructions :

Instruction	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Opérations à 2 registres : $rd := rd\ op\ rs$								
`or rs, rd`	0	0	0	0	rs_1	rs_0	rd_1	rd_0
`xor rs, rd`	0	0	0	1	rs_1	rs_0	rd_1	rd_0
`and rs, rd`	0	0	1	0	rs_1	rs_0	rd_1	rd_0
`add rs, rd`	0	1	0	0	rs_1	rs_0	rd_1	rd_0
`sub rs, rd`	0	1	0	1	rs_1	rs_0	rd_1	rd_0
Opérations à 1 registre : $rd := op\ rd$								
`not rd`	0	0	1	1	0	0	rd_1	rd_0
`shl rd`	0	1	1	0	0	0	rd_1	rd_0
`shr rd`	0	1	1	1	0	0	rd_1	rd_0
Stockage : $MEM(AD) := rs$								
`st rs, AD`	1	1	0	0	0	0	rs_1	rs_0
					ADH			
					ADL			
Chargement : $rd := MEM(AD)$								
`ld AD, rd`	1	0	0	0	0	0	rd_1	rd_0
					ADH			
					ADL			
Chargement de constante : $rd := imm8$								
`li imm8, rd`	1	0	1	0	0	0	rd_1	rd_0
					imm8			
Branchement inconditionnel : $PC := AD$								
`jmp AD`	1	0	0	0	0	1	0	0
					ADH			
					ADL			
Branchement conditionnel : si $COND$ alors $PC := AD$								
`jz AD`	1	0	0	0	0	1	0	1
					ADH			
					ADL			