Architecture 1
TP 5 : fonctionnement d'un processeur

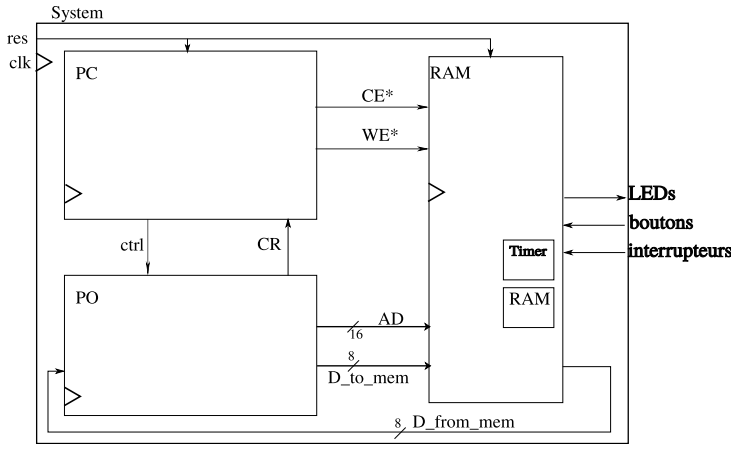
Il est impératif d'avoir effectué le travail préparatoire avant de venir en séance.

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.

Ex. 1 : Travail préparatoire

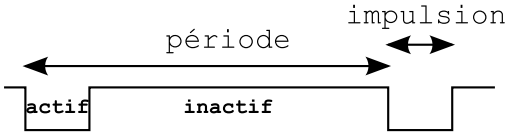
Repartons de l'environnement du processeur vu en TD (8 à 11). Pour interagir avec le processeur, il faut ajouter à l'environnement des périphériques (LED, interrupteurs...). Notre processeur peut accéder à ses périphériques en utilisant les instructions d'accès mémoire (ld, st) dans une plage d'adresses dédiée aux périphériques. La table des périphériques donnée en annexe précise les accès possibles. 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.

Un timer est un composant matériel qui mesure le temps. Le timer fourni respecte le chronogramme ci-dessous. Par défaut, l'impulsion est d'une durée de 18 cycles du processeur (pour rappel, le processeur étudié traite les instructions de chargement, stockage et saut en 5 cycles et toutes les autres en 3 cycles). Par défaut, la période du timer est d'environ 500ms :



Afin de détecter l'activation de ce timer, le processeur doit faire une boucle de *scrutation*, c'est-à-dire une boucle dans laquelle on exécute une instruction de lecture du timer.

Question 2

— Comment déterminer la durée d'un cycle du processeur en millisecondes ?

— On considère des programmes en assembleur de la forme suivante :

```
attente:
    ld 0x0010, r0
    -- instructions diverses...
    jmp attente
```

Dans le cas où il n'y a pas d'instruction entre le `ld` et le `jmp`, quelle durée (en millisecondes) sépare deux instants où le processeur lit le timer ? Même question si les instructions `ld` et `jmp` sont séparées par les 4 instructions suivantes : `and r0,r0` puis `st r3, 0x0001` puis `li 16,r2` et finalement `shl r1`.

— Dessiner des chronogrammes sur lequels on peut observer le comportement du timer (comme sur la figure ci-dessus), et les instants où le programme lit le timer.

— Si les lectures du timer sont trop espacées dans le temps, on risque de "rater" une impulsion du timer. Donner un exemple de programme de la forme ci-dessus, où ce cas se produit.

— Inversement, si les lectures du timer sont très proches dans le temps, il se peut qu'on fasse deux lectures du timer pendant une même impulsion. Pouvez-vous donner un exemple de programme de la forme ci-dessus, où ce cas se produit ?

Question 3

— En utilisant le timer, écrire un programme en langage d'assemblage qui fait clignoter toutes les LED à 1Hz.

— Calculer le temps qui sépare deux lectures du timer par le processeur, dessiner les chronogrammes correspondants, et expliquer pourquoi ça marche.

— Trouver des valeurs pour la fréquence du processeur, la durée des impulsions du timer, la période du timer, pour lesquelles le même programme ne marche plus.

Au cours du TP, on vous demandera d'exécuter un programme permettant d'afficher un chenillard sur les LED de la carte. C'est-à-dire que les LED allumées se décalent d'une position à intervalles réguliers (toutes les 0,5s pour notre exemple). L'algorithme ci-contre permet de réaliser un chenillard sur un afficheur à 4 LED :

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

Question 4

— Écrivez un programme en langage d'assemblage réalisant cet algorithme.

— Comme à la question précédente, dessinez des chronogrammes pour montrer les impulsions du timer et les instants où le processeur lit le timer. Expliquez pourquoi ça marche.

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 « *Instances and Processes* » 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 les tester sur carte, il vous suffit d'utiliser la commande :

```
$ make run_fpga PROG=nom
```

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

Question 2 Regardez le programme fournit `exo2.as`. A votre avis, que fait-il ?
Implantez le sur la carte et vérifiez.

Question 3 Ecrivez dans un fichier `src/nom.as` chacun des programmes obtenus en travail préparatoire et testez les sur carte. 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.

Question 4 Il vous reste du temps. Pourquoi ne pas s'amuser avec vos propres programmes ? Quelques idées en vrac :

- Coder une multiplication.
- Coder un compteur qui affiche le résultat de plus en plus lentement.

Ex. 4 : 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)
		0x0040	Change la durée de l'impulsion du timer (défaut 18 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

Jeux 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
Chargement : $rd := MEM(AD)$								
`ld AD, rd`	1	0	0	0	0	0	rd_1	rd_0
	ADH							
	ADL							
Stockage : $MEM(AD) := rs$								
`st rs, AD`	1	1	0	0	0	0	rs_1	rs_0
	ADH							
	ADL							
Branchement inconditionnel : $PC := AD$								
`jmp AD`	1	0	0	0	0	1	0	0
	ADH							
	ADL							
Chargement de constante : $rd := imm8$								
`li imm8, rd`	1	0	1	0	0	0	rd_1	rd_0
	imm8							
Branchements conditionnels : si $COND$ alors $PC := AD$								
`jz AD`	1	0	0	0	0	1	0	1
	ADH							
	ADL							