TP 5 : Simulation d'un processeur dans Diglog

L'objectif de ce TD machine est la réalisation d'un mini-processeur dans Diglog. Pour ce faire, nous allons utiliser :

- les circuits combinatoires (multiplexeurs, ALU) et séquentiels (registres, compteurs) que vous avez faits aux TDs précédents ;
- un langage machine pour notre mini-processeur, décrit ici : `http://www.ensiie.fr/~christophe.mouilleron/Teaching/ARMA/langage.pdf` ;
- un compilateur, transformant du code assembleur en fichiers de données à charger dans les blocs SRAM de Diglog, et dont les sources sont disponibles ici : `http://www.ensiie.fr/~christophe.mouilleron/Teaching/ARMA/digcomp_skel.tbz2` ;
- une première ébauche du processeur, que vous trouverez ici : `http://www.ensiie.fr/~christophe.mouilleron/Teaching/ARMA/digproc_skel.tbz2`.

Exercice 1 - Exécution du premier programme

Pour le moment, le processeur fourni ne supporte que les instructions `ldi` et `addi`. Faire appel à toute autre instruction conduit à un comportement non spécifié.

Le but de cet exercice est d'ajouter le support des sauts inconditionnels, afin de pouvoir simuler l'exécution du programme suivant :

```
        ldi r1, 42
        addi r0, r1, 17
end:    jmp end
```

1.1 Donner la valeur de chaque registre à la fin de l'exécution de ce programme.

1.2 Traduire à la main le code assembleur en langage machine, sachant que la première instruction sera placée à l'adresse `0000`.

1.3 Compiler le code assembleur avec *digcomp*. Comparer le contenu des fichiers ainsi produits à la réponse de la question précédente.

1.4 Charger les fichiers obtenus dans Diglog, et déterminer le chemin suivi par les données lors de l'exécution de l'instruction `ldi`.

Note : Pour faire une exécution pas à pas, on utilisera un générateur en guise d'horloge, sur lequel on pourra cliquer afin de passer au front (montant ou descendant) suivant.

1.5 Avancer d'un cycle, et déterminer le chemin suivi par les données lors de l'exécution de l'instruction `addi`.

1.6 Avancer à nouveau d'un cycle, et identifier comment récupérer l'adresse à laquelle il faudra être après le saut.

1.7 Déterminer le rôle des bits de contrôle *write_reg*, *arg2_imm* et *res_imm*, puis compléter le tableau suivant (mettre `x` si la valeur du bit de contrôle n'a aucune influence) :

instruction	*do_jmp_abs*	*write_reg*	*arg2_imm*	*res_imm*
`ldi`	0			
`addi`	0			
`jmp`	1			

1.8 Modifier le processeur afin de gérer l'instruction de saut `jmp`. Utiliser pour cela le bit de contrôle *do_jmp_abs* (le bit et *do_jcc* servira plus tard pour les sauts conditionnels).

Tester le résultat sur le code assembleur proposé précédemment.

Exercice 2 - Gestion complète de l'addition et de la soustraction

2.1 Donner la liste des instructions dont l'exécution nécessite d'utiliser l'ALU en tant que soustracteur.

2.2 Modifier le processeur afin d'obtenir la bonne valeur pour le bit de contrôle *do_sub*. Vérifier que le processeur gère désormais aussi l'instruction `subi`.

2.3 Modifier le banc de registre afin de pouvoir lire les valeurs de deux registres (pas forcément différents) à chaque cycle.

2.4 Pour chaque instruction, déterminer le nombre de lectures de registres à effectuer, ainsi que les bits contenant les numéros des registres à lire.

2.5 Ajouter le support des instructions `add` et `sub`, et proposer un code assembleur de test.

Vous devrez sûrement modifier le calcul des bits de contrôle introduits à l'exercice précédent.

Exercice 3 - Entrées/sorties et gestion du saut `jeq`

Afin de pouvoir procéder à de vrais tests, il nous manque deux choses :
- un coté interactif pour accélérer/faciliter les tests, c'est-à-dire la possibilité de saisir des entrées au clavier et d'afficher des résultats sur un écran ;
- notre première instruction de saut conditionnel, afin d'augmenter significativement l'expressivité dans les programmes codés en assembleur.

Pour le premier point, le fichier *io.lgf* fournit déjà un clavier et un écran. Il est possible de récupérer une touche saisie au clavier (fils *kb0* à *kb1*) à condition de positionner le bit de contrôle *in* à 1. De plus, il est possible d'afficher le caractère dont le code ASCII est la valeur RD (fils *RD0* à *RD7*) à condition de positionner le bit de contrôle *out* à 1.

3.1 Faire en sorte que les bits de contrôle *in* et *out* reçoivent la bonne valeur.

3.2 Tester le clavier et l'écran. Que se passe-t-il si on récupère des données du clavier alors qu'aucune touche n'a été frappée ? et si de nombreuses touches ont été frappées depuis la dernière récupération de touche ?

3.3 Rappeler comment on peut tester que deux valeurs entières sont égales. Modifier l'ALU afin d'avoir une nouvelle sortie, nécessaire à la réalisation du test.

3.4 Afin de gérer les instructions de saut conditionnel, nous allons utiliser le bit de contrôle *do_jcc*, qui vaudra 1 si et seulement si l'instruction courante est de type saut conditionnel et qu'il convient d'effectuer le saut en question.

Faire en sorte que ce bit de contrôle reçoive la bonne valeur lors de l'exécution d'un `jeq`.

3.5 Modifier le processeur afin de mettre correctement à jour le pointeur d'instruction PC lors de l'exécution d'une instruction `jeq`.

3.6 Tester le bon fonctionnement des sauts conditionnels dans le cas d'un saut :

1. en avant (vers une adresse plus grande que la valeur courante dans PC),
2. en arrière (vers une adresse plus petite que la valeur courante dans PC).

3.7 Écrire un code assembleur qui récupère les touches réellement saisies par l'utilisateur, les affichent, et s'arrête dès que l'utilisateur a appuyé sur la touche entrée (*Cr*, de code ASCII 13).

3.8 Écrire un code assembleur qui récupère un entier $n \in [1, 9]$ et un caractère c, puis qui affiche à l'écran un carré de taille n et composé de caractères c.

Exercice 4 - Gestion de la mémoire et des autres sauts conditionnels

4.1 Ajouter le support des instructions `ld` et `st`. Pour cela, il faudra identifier le parcours que les données devront suivre, ajouter des multiplexeurs au besoin, et utiliser au minimum un nouveau bit de contrôle : *write_mem*.

Note : Pour l'instant, on ne traitera pas la partie `imm5` de ces instructions.

4.2 Écrire un code assembleur qui récupère les touches saisies par l'utilisateur et les stocke en mémoire jusqu'à la saisie de la touche entrée, puis qui réécrit les données ainsi récupérées à l'écran en ordre inverse.

Exemple : La saisie de « 123<*entrée*> » donnera donc lieu à l'affichage de « 321 ».

4.3 Modifier l'ALU afin d'avoir 3 nouvelles sorties, correspondant aux drapeaux S (signe), C (carry), et O (overflow).

4.4 Démontrer que, pour a et b deux entiers signés sur 8 bits différents, $a \leq b$ est équivalent à $S = O$, où S et O sont obtenus suite au calcul de $b - a$ par l'ALU.

4.5 Utiliser le résultat précédent et ce qui a déjà été à la section précédente pour ajouter le support des instructions `jlt`, `jle` et `jne`.

4.6 Écrire un code assembleur qui effectue à l'aide d'une boucle la multiplication par 10 de la valeur stockée dans *r0*.

Comment peut-on faire ce calcul beaucoup plus efficacement ?

4.7 Écrire un code assembleur qui récupère les touches saisies par l'utilisateur et les stocke en mémoire jusqu'à la saisie de la touche entrée, puis qui réécrit à l'écran les données ainsi récupérées mais en majuscules.

Exemple : La saisie de « Abc1<*entrée*> » donnera donc lieu à l'affichage de « ABC1 ».

Exercice 5 - Améliorations diverses

Certaines questions de cet exercice impliquent de compléter le code source du compilateur.

5.1 Améliorer l'ALU et modifier le chemin de données de façon à supporter les instructions logiques (catégorie `001`).

5.2 Écrire un code qui lit un entier saisi au clavier.

5.3 Écrire un code qui effectue la division par 10 de la valeur stockée dans *r0*.

5.4 Écrire un code qui lit deux entiers saisis au clavier, puis affiche le résultat de leur multiplication (modulo 256) à l'écran.

5.5 Compléter le support des instructions `ld` et `st` afin de gérer la partie `imm5` de l'instruction.

5.6 Ajouter le support de l'instruction *jr*.