TD 5 : Assembleur MIPS et Pipelines

Exercice 1 - Un peu d'assembleur MIPS

pourquoi faire simple quand on peut faire compliqué ?

1. Écrivez un programme assembleur fournissant l'indice du plus grand élément d'un tableau de dix entiers.

   Dans les questions suivantes, on s'intéresse à la fonction mystere détaillée ci-dessous.

```
void mystere( A : liste d'entiers ) {
  n = taille(A);
  repeat
    newn = 0;
    for i=1 to n-1 do
      if (A[i-1] > A[i] then
        swap(A[i-1],A[i]);
        newn = i;
      end if;
    end for;
    n = newn;
  until n = 0;
}
```

2. Que fait la fonction mystere ?
3. Écrivez un programme assembleur implémentant une version **itérative** de la fonction mystere.
4. (**Bonus**) : Même question pour une version **récursive** de la fonction mystere.

Exercice 2 - Les pipelines

<insert a nasty joke here>

Le but de cet exercice est de comprendre l'intérêt du *pipelining*, une opération utilisée lors de la conception d'ordinateurs afin d'optimiser leur rendement et leur efficacité. Cet exercice est découpé comme suit : après avoir brièvement expliqué le principe et l'intérêt du *pipelining*, nous pointerons plusieurs défauts pouvant apparaître en pratique. À vous d'imaginer des solutions répondant à ces problèmatiques successives. Vous serez d'ailleurs confrontés à certaines d'entre elles lors de la réalisation du projet d'ASR1.

Introduction au *pipelining*

Comme nous l'avons vu, un processeur fonctionne par cycles de temps cadencés par une horloge. Bien entendu, pour des raisons de performance, en pratique un processeur ne se contente par d'effectuer une tâche par cycle. Plusieurs tâches peuvent ainsi être effectuées de manière concurrente (en même temps) ou parallèle (par des circuits différents).

Clairement, la longueur d'un cycle est fortement influencée par la longueur des tâches effectuées en parallèle. Ainsi, si une tâche A demande beaucoup plus de temps que d'autres pour être effectuée, il s'en suit un asynchronisme qui ne sera réglé qu'au prochain cycle suivant la fin de l'exécution de A.

Pour s'assurer de l'"uniformité" des tâches à exécuter (et donc d'une utilisation optimale du processeur), on recourt au *pipelining*. Concrètement, il s'agit de "découper" chaque tâche en plusieurs

sous-tâches, celles-ci pouvant être traitées par des parties différentes du *hardware*. Cela permet notamment de fixer la longueur du cycle d'horloge : il suffit alors de se référer aux sous-tâches demandant le plus grand temps d'exécution.

Le nombre de sous-tâches résultant du *pipelining* dépend de l'architecture considée. Ici, nous nous fixons un découpage de chaque instruction en certaines des sous-instructions (ou *étages*) ci-dessous.

— **Instruction Fetch (IF) :** l'instruction suivante est placée dans le registre d'instruction depuis la mémoire ;
— **Instruction Decode (ID) :** les registres nécessaires sont lus ;
— **Execution (EXE) :** l'opération de calcul de l'instruction est effectuée (= l'ALU pour une opération arithmétique ou logique, calcul d'adresse mémoire, etc.) ;
— **Memory Access (MEM) :** si l'instruction concerne la mémoire (écriture/lecture), celle-ci est exécutée ;
— **Write Back (WB) :** les registres-cibles sont mis à jour.

Les sous-tâches doivent alors être exécutées suivants les deux règles suivantes :
— chaque étage s'exécute en un cycle,
— un étage exécute au plus une instruction par cycle.

Problèmes structurels

Problème 1

Considérez l'exécution suivante, où chaque instruction n'utilise que les étages nécessaires à son exécution.

Instr − Cycle	1	2	3	4	5	6	7	8	...
load	IF	ID	EXE	MEM	WB				
add		IF	ID	EXE	WB				

1. Quel est le problème ? Proposez une solution.

Problème 2

Considérez l'exécution suivante :

Instr − Cycle	1	2	3	4	5	6	7	8	...
load	IF	ID	EXE	MEM	WB				
instr1		IF	ID	EXE	MEM	WB			
instr2			IF	ID	EXE	MEM	WB		
instr3				IF	ID	EXE	MEM	WB	

1. Quel est le problème ? Proposez une solution.

Problème de données

Deux instructions peuvent utiliser les mêmes données. Plusieurs types de dépendances peuvent alors survenir :

1. Read-After-Write (RAW) : dépendance vraie,
2. Write-After-Read (WAR) : anti-dépendance,
3. Write-After-Write (WAW) : dépendance de sortie,
4. Read-After-Read (RAR) : dépendance d'entrée.

1. Pour chaque type de dépendance, donnez un exemple concret la faisant apparaître. Indiquez si ce type de dépendance pose problème.
2. Comment peut-on résoudre ce(s) problème(s) ?

3. Supposons que l'on choisisse la solution du *bypass*. Listez tous les *bypass* utiles et illustrez leur utilisation par des exemples.

Problèmes de contrôle

Delay slots

1. Dans le cas d'une instruction de branchement ou de saut, à quel étage a-t-on déterminé l'adresse de destination ?
2. Combien de cycles sont donc perdus ?
3. Proposez une solution.

Stratégies de branchement

Par la solution retenue précédemment, un *delay slot* reste incompressible.

1. Proposez une ou plusieurs solutions.

Pipeline hazards

Pour chacun des codes MIPS ci-dessous, (i) déterminer les dépendances, (ii) dessiner le pipeline simplifié.

```
add $r3, $r2, $r1
j $r3
```

```
lw $r3, 0($r5)
add $r5, $r4, $r3
```

```
lw $r3, 0($r5)
beq $r3, $r0, L
```

Pour le code MIPS ci-dessous, faire (i), (ii) et (iii) calculer le CPI et CPIutile.
CPI = #cycle / #instructions
CPIutile = #cycle / #instructions_utile (sans NOP)

```
lw $r3, 0($r5)
addiu $r8, $r8, 1
sll $r3, $r3, 1
sw $r3, 0($r5)
bne $r8, $r9, loop
addiu $r5, $r5, 4
```