Architecture des ordinateurs : TD3

Université de Tours

`Département informatique de Blois`

Processeur et jeu d'instructions

*
* *

Problème 1

Un processeur possède les registres $\mathtt{r}_{i\in\{0,\dots,9\}}$. On souhaite calculer l'expression suivante :

$$\mathtt{r}_0 \leftarrow 2\left(\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4}+\mathtt{r}_5\times\mathtt{r}_6\right)+(\mathtt{r}_1+\mathtt{r}_2)\times(\mathtt{r}_3-\mathtt{r}_4)$$

1. Écrire une suite d'instructions qui calcule cette expression à l'aide des mnémoniques ci-dessus en minimisant le nombre de commandes possibles. On précise que l'on dispose au maximum des registres $\mathtt{r}_0$ à $\mathtt{r}_9$.

   Combien d'accès mémoire sont effectués ?

   $\mathtt{r}_1+\mathtt{r}_2$: `add` $\mathtt{r}_7,\mathtt{r}_1,\mathtt{r}_2$

   $\mathtt{r}_3-\mathtt{r}_4$: `add` $\mathtt{r}_8,\mathtt{r}_3,\mathtt{r}_4$

   $\mathtt{r}_5\times\mathtt{r}_6$: `add` $\mathtt{r}_5,\mathtt{r}_5,\mathtt{r}_6$

   $\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4}$: `div` $\mathtt{r}_7,\mathtt{r}_8$

   $\mathtt{r}_0 \leftarrow \frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4}$: `mflo` $\mathtt{r}_0$

   $\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4}+\mathtt{r}_5\times\mathtt{r}_6$: `add` $\mathtt{r}_0,\mathtt{r}_0,\mathtt{r}_5$

   $2\left(\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4}+\mathtt{r}_5\times\mathtt{r}_6\right)$: `muli` $\mathtt{r}_0,\mathtt{r}_0,2$

   $(\mathtt{r}_1+\mathtt{r}_2)\times(\mathtt{r}_3-\mathtt{r}_4)$: `mul` $\mathtt{r}_7,\mathtt{r}_7,\mathtt{r}_5$

   $2\left(\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4}+\mathtt{r}_5\times\mathtt{r}_6\right)+(\mathtt{r}_1+\mathtt{r}_2)\times(\mathtt{r}_3-\mathtt{r}_4)$: `add` $\mathtt{r}_0,\mathtt{r}_0,\mathtt{r}_7$

   On a 3 accès mémoire par commande sauf pour :

   - `div` $\mathtt{r}_7,\mathtt{r}_8$: 4 accès mémoire

     2 pour $\mathtt{r}_7,\mathtt{r}_8$ et 2 pour placer respectivement le résultat de la division entière (registre lo) et le reste de la division (registre hi).
   - `mflo` $\mathtt{r}_0$: 2 accès mémoire

     Un accès au registre lo et un stokage dans $\mathtt{r}_0$.
   - `muli` $\mathtt{r}_0,\mathtt{r}_0,2$: 2 accès mémoire

   On a donc 26 accès mémoire en tout.

2. Le processeur possède un registre accumulateur `acc` qui est la source et destination de toutes les opérations arithmétiques et logiques. On donne les nouvelles instructions suivantes :

- `add` $\mathtt{r}_x$ $(\mathtt{acc} \leftarrow \mathtt{acc} + \mathtt{r}_x)$
- `sub` $\mathtt{r}_x$ $(\mathtt{acc} \leftarrow \mathtt{acc} - \mathtt{r}_x)$
- `mul` $\mathtt{r}_x$ $(\mathtt{acc} \leftarrow \mathtt{acc} \times \mathtt{r}_x)$
- `div` $\mathtt{r}_x$ $(\mathtt{acc} \leftarrow \mathtt{acc}/\mathtt{r}_x)$
- `load` $\mathtt{r}_x$ $(\mathtt{acc} \leftarrow \mathtt{r}_x)$
- `store` $\mathtt{r}_x$ $(\mathtt{r}_x \leftarrow \mathtt{acc})$

Écrire une suite d'instructions correspondant au calcul voulu à l'aide des mnémoniques ci-dessus en minimisant le nombre de commandes. Le résultat sera placé dans $\mathtt{r}_0$. Combien d'accès mémoire sont effectués ? Justifier l'utilité de l'accumulateur.

$\mathtt{load}\,\mathtt{r}_5$

$\mathtt{r}_5 \times \mathtt{r}_6$: $\mathtt{mul}\,\mathtt{r}_6$

$\mathtt{store}\,\mathtt{r}_5$

$\mathtt{load}\,\mathtt{r}_3$

$\mathtt{r}_3 - \mathtt{r}_4$: $\mathtt{sub}\,\mathtt{r}_4$

$\mathtt{store}\,\mathtt{r}_4$

$\mathtt{load}\,\mathtt{r}_1$

$\mathtt{r}_1 + \mathtt{r}_2$: $\mathtt{add}\,\mathtt{r}_2$

$\mathtt{store}\,\mathtt{r}_1$

$\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4}$: $\mathtt{div}\,\mathtt{r}_4$

$\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4} + \mathtt{r}_5 \times \mathtt{r}_6$: $\mathtt{add}\,\mathtt{r}_5$

$2\left(\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4} + \mathtt{r}_5 \times \mathtt{r}_6\right)$: $\mathtt{muli}\,2$

$\mathtt{store}\,\mathtt{r}_0$

$\mathtt{load}\,\mathtt{r}_1$

$(\mathtt{r}_1 + \mathtt{r}_2) \times (\mathtt{r}_3 - \mathtt{r}_4)$: $\mathtt{mul}\,\mathtt{r}_4$

$2\left(\frac{\mathtt{r}_1+\mathtt{r}_2}{\mathtt{r}_3-\mathtt{r}_4} + \mathtt{r}_5 \times \mathtt{r}_6\right) + (\mathtt{r}_1 + \mathtt{r}_2) \times (\mathtt{r}_3 - \mathtt{r}_4)$: $\mathtt{add}\,\mathtt{r}_0$

$\mathtt{store}\,\mathtt{r}_0$

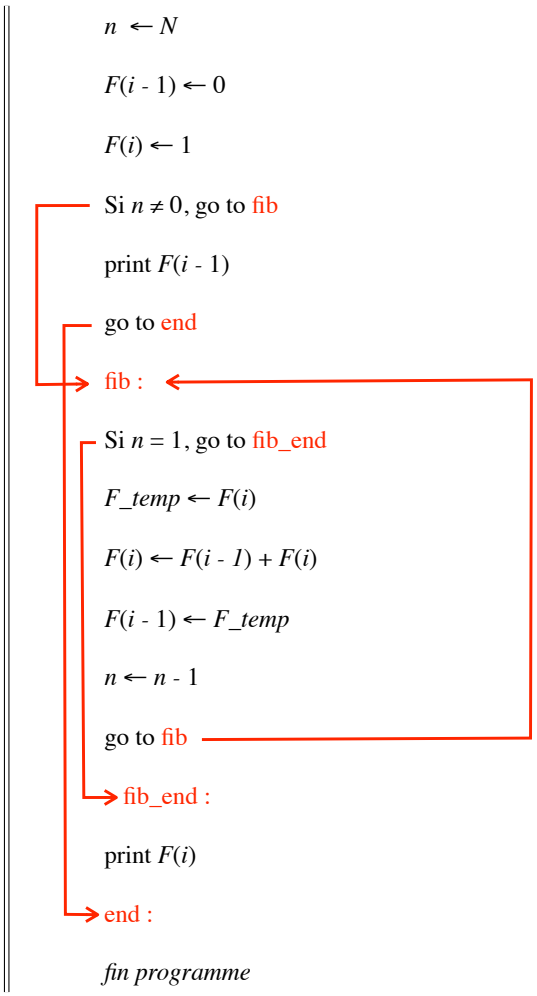
On a 16 accès mémoire.

Problème 2

Soit la suite de Fibonacci $(\mathcal{F}_n)_{n\in\mathbb{N}}$ définie telle que :

$$\begin{cases} \mathcal{F}_0 & = 0 \\ \mathcal{F}_1 & = 1 \\ \mathcal{F}_{n+2} & = \mathcal{F}_{n+1} + \mathcal{F}_n \end{cases}$$

1. Donner la forme linéaire de l'algorithme du calcul de $\mathcal{F}_n$ sous forme itérative.



2. Donner le programme assembleur MIPS correspondant au calcul itératif de $\mathcal{F}_n$.
3. On s'intéresse au calcul de la suite de Fibonacci sous selon la version récursive.
   (a) Donner le programme assembleur MIPS correspondant au calcul récursif de $\mathcal{F}_n$.
   (b) Dessiner l'évolution mémoire de la pile pour l'exécution du programme pour $n = 3$.

Problème 3

On considère un ensemble d'instructions $\mathcal{I}$ correspondant à l'exécution d'un programme. On suppose $|\mathcal{I}| = n$. La table suivante décrit les caractéristiques de ce dernier et de l'ensemble $\mathcal{I}$:

Instruction i	Proportion p_i	Nb cycles C_i
Opérations UAL	0.43	1
Chargement	0.21	2
Stockage	0.12	2
Branchement	0.24	2

On suppose que le programme s'exécute sur une machine dont la période de cycle horloge est de $20ns$.

1. Calculer le MIPS et le CPU_TIME du programme non-optimisé.

Calculons dans un premier la fréquence de notre processeur. La machine est cadensée à $20ns$, soit une période $P = 20 \times 10^{-9} = 2 \times 10^{-8}$.

On sait que : $f = \frac{1}{P} = \frac{1}{2\times10^{-8}} = 2 \times 10^8 = 200 \times 10^6 = 200MHz$.

Le CPI (ou Cycle Per Instruction) moyen est donné par $\sum_i p_i c_i$, dans notre cas : CPI $= 0.43 + 0.21 \times 2 + 0.12 \times 2 + 0.24 \times 2 = 1.57$

Le MIPS (Million Instructions Per Second) est une unité permettant de quantifier la performance globale d'une machine : MIPS $= \frac{f}{\text{CPI}\times10^6} = \frac{200}{1.57} = 127.39$

Enfin, le CPU_TIME permet de quantifier le temps processeur pour un ensemble d'instructions donné : CPU_TIME $= |\mathcal{I}| \times \text{CPI} \times P = 1.57 \times 2 \times 10^{-8} n = 3.14 \times 10^{-8} n$

2. Pour optimiser le programme, le compilateur réduit de 50% la fréquence d'utilisation des opérations UAL de $\mathcal{I}$ cependant, il ne réduit pas le nombre de chargements, stockages ou branchements.

   (a) Calculer les nouvelles proportions d'utilisation p_i des unités.

   Pour rappel, soit Ω, un ensemble d'éléments et $E \subseteq \Omega$. La proportion p d'éléments de E parmi Ω est donnée par $p = \frac{|E|}{|\Omega|}$.

   Soit $(E_1, ..., E_m)$, une partition de Ω telle que $\bigcup_{i=1}^{m} E_i = \Omega$ et telle que pour tout $i \neq j$, on a $E_i \cap E_j = \emptyset$. Alors $\sum_{i=1}^{m} \frac{|E_i|}{|\Omega|} = \sum_{i=1}^{m} p_i = 1$. (La somme de toutes les proportions doit être égale à 1).

   Dans notre cas, Ω est notre ensemble d'instructions $\mathcal{I}$ et ses sous-ensembles sont les différents postes d'instructions (UAL, Load, Stockage et Branchement).

   Notons $|\mathcal{I}^{(op)}|$ l'ensemble optimisé des instructions. L'énoncé nous affirme que le compilateur a réussi à optimiser les instructions UAL en diminuant leur proportion de moitié.

   Dès lors, il vient que $|\mathcal{I}^{(op)}| = n - \frac{0.43n}{2} = n(1 - 0.215) = 0.785n$

   On note $p_i^{(op)}$ les nouvelles proportions optimisées, il vient que :

   - (UAL) : $p_1^{(op)} = \frac{0.43n}{2} \times \frac{1}{|\mathcal{I}^{(op)}|} = 0.215n \times \frac{1}{0.785n} = 0.27$
   - (Chargement) : $p_2^{(op)} = \frac{0.21n}{|\mathcal{I}^{(op)}|} = \frac{0.21n}{0.785n} = 0.27$
   - (Stockage) : $p_3^{(op)} = \frac{0.12n}{0.785n} = 0.15$
   - (Branchement) : $p_4^{(op)} = \frac{0.24n}{0.785n} = 0.31$

   (b) Calculer le MIPS et le CPU_TIME du code optimisé. Que constatez-vous ?

   On re-calcule les mesures traitées question 1.

   - CPI$^{(op)} = 0.27 + 0.27 \times 2 + 0.15 \times 2 + 0.31 \times 2 = 1.57 = 1.73$
     $\Rightarrow$ *Interprétation* : Sur le programme optimisé, les instructions en moyenne durent plus longtemps.
   - MIPS$^{(op)} = \frac{200}{1.73} = 115.61$
     $\Rightarrow$ *Interprétation* : Globalement, les performances de la machine semblent moins efficaces car le MIPS est plus faible. Cela vient du fait que le MIPS ne tient pas compte du nombre d'intructions effectuées.
   - CPU_TIME$^{(op)} = |\mathcal{I}^{(op)}| \times \text{CPI} \times P = 1.73 \times 2 \times 10^{-8} \times 0.785n = 2.72 \times 10^{-8} n$
     $\Rightarrow$ *Interprétation* : Plus le CPU_TIME est faible, meilleur est l'exécution d'un programme. On constate bien ici que le compilateur a réussi à optimiser notre programme

car celui-ci est plus faible que précédemment.
On peut calculer le gain de performance $\gamma_{perf} = \frac{|\text{CPU\_TIME}^{(op)} - \text{CPU\_TIME}|}{\text{CPU\_TIME}} = \frac{|2.72-3.14|}{3.14} = 13\%$.
Le compilateur a optimisé le programme d'environ 13% au niveau du temps processeur.

Problème 4

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 considérée. 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'UAL 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.

1. On considère l'exécution suivante :

Instr − Cycle	1	2	3	4	5	6	7	8	...
`lb $r1, $r2`	IF	ID	EXE	MEM	WB				
i_1		IF	ID	EXE	MEM	WB			
i_2			IF	ID	EXE	MEM	WB		
i_3				IF	ID	EXE	MEM	WB	

Quel est le problème ? Proposer une solution

2. Deux instructions peuvent utiliser les mêmes données. Plusieurs types de dépendances peuvent alors survenir :
   (a) Read-After-Write (RAW) : dépendance vraie,
   (b) Write-After-Read (WAR) : anti-dépendance,
   (c) Write-After-Write (WAW) : dépendance de sortie,
   (d) Read-After-Read (RAR) : dépendance d'entrée.

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

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

4. Pour chacun des codes MIPS ci-dessous,
   (a) Déterminer les dépendances.
   (b) 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, etiq
```

   (c) Soit le programme suivant :

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

   Déterminer les dépendances et dessiner le pipeline puis calculer le MIPS et le CPU_Time pour un processeur cadencé à 1.6 GHz.

   On considèrera que les instructions UAL consomment 1 cycle, les chargements et stockage 2 cycles et les branchements 1 cycle.

Problème 5

1. On considère le calcul $Y = aX + Y$ où X et Y sont deux vecteurs de dimension $n > 0$ et $a \in \mathbb{F}$ une constante flottante.

   On donne le programme suivant SAXPY[1] :

[1] Ce programme standard d'algèbre linéaire est très souvent utilisé lors des benchmark pour évaluer les performances des cartes graphiques.

```
1. void SAXPY(float[] X, float[] Y, float a, unsigned int n) {
2.   int i;
3.   for(i = 0; i < n; i++) {
4.      Y[i] = Y[i] + a * X[i];
5.   }
6. }
```

On considère l'état des registres suivants : $\texttt{r}_8$: `i`; $\texttt{r}_0$: `X`; $\texttt{r}_1$: `Y`; $\texttt{r}_2$: `n`; $\texttt{F}_0$: `a`;

De plus on considère que les registres flottants $\texttt{F}_1, \texttt{F}_2, \texttt{F}_3, \texttt{F}_4$ sont libres.

Traduire le programme ci-dessus en une suite d'instructions assembleur en utilisant le jeu d'instructions MIPS décrit dans le cours.

On précise que les instructions flottantes utilisent des mnémoniques similaires que ceux vus pour les entiers mais sont caractérisées par le suffixe `.s`.

Ex : `add $t0, $t1, $t2` devient pour les flottants `add.s $F0, $F1, $F2`.

2. Décrire les aléas pipeline produits lors son l'exécution pour une division $\mathcal{E}$ en cinq étages telle que décrite dans le **Problème 4**.