Master SESI

Architecture et réalisation des processeurs RISC

Collection de corrections & fiches

Ce document a été réalisé par Zak MEKHLOUFI basé sur les notes de Paul KLINT de cours et travaux dirigés de l'unité d'enseignement d'*architecture et réalisation des processeurs RISC* du master *Sytèmes Électroniques Systèmes Informatiques* (SESI) enseigné en première année par D[r] Pirouz BAZARGAN-SABET, directeur et professeur du master.

Table des matières

1 Avertissement

Ce document **n'est en aucun cas un document officiel** de l'unité d'enseignement (UE), ainsi celui-ci peut contenir des erreurs, des explications ou schémas erronés. Il est conseillé d'utiliser ce document avec un certain regard critique et en cas de doute, se réferrer directement au professeur.
Nous encourageons le lecteur de ce document à échanger avec ses camarades, et d'aider ceux qui sont en difficultés dans cette UE car c'est un très bon moyen d'auto évaluer ses connaissances et sa compréhension du cours. Ainsi, le lecteur ayant compris le cours ne trouvera aucune difficulté à l'expliquer au profane.

2 Memento de l'architecture du MIPS

2.1 Les registres du MIPS

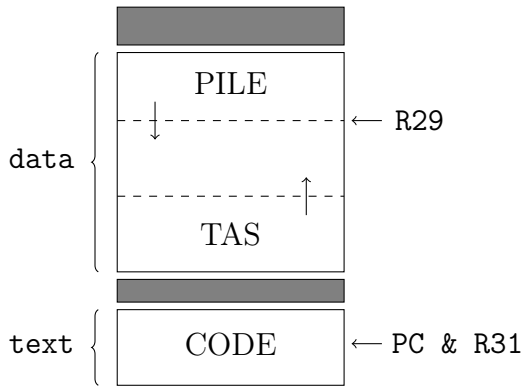
Le MIPS R3000 contient deux bancs de registres de 32 bits. L'un est destiné à un usage en mode USER et l'autre en mode KERNEL (le système), ce dernier banc est contenu dans le coprocesseur 0 (noté `CP0`).

Status Register (`R12` du `CP0`) :
Il contient l'état du masque des interruptions et le mode de fonctionnement du processeur.

Cause Register (`R18` du `CP0`) :
En cas d'entrée en mode KERNEL il contient la cause pour laquelle on fait appel au système : Interruption, Syscall ou Exception.

Lors de son execution, un programme est représenté en mémoire par son *espace d'adressage* (noté EA). Il s'agit du code et des données enregistrées respectivement dans les segments `text` et `data` (qui sont situés en RAM).
La **pile** sert à la sauvegarde des registres permanents, aux variables locales et aux surplus d'argument des fonctions appelées.
Le **tas** contient les variables globales.

Schéma d'une partie de l'EA



Les registres USER utiles

R0	Contient toujours 0x0
R2	Valeur de retour
R4 ... R7	Arguments des fonctions appelées
R8 ... R15	Registres non-permanents
R16 ... R29	Registres permanents
R28	Pointeur des var. glob.
R29	Pointeur de pile
R31	Adresse de retour vers la fonction appelante
PC	Program Counter
HI LO	Utilisé pour les Mult. et Div.

-Dans le cas d'une Div :
HI contient le reste et
LO le quotient

-Dans le cas d'une Mult :
HI contient les MSB et
LO les LSB

2.2 Représentation et adressage en mémoire

Le MIPS R3000 utilise le boutisme "Little-endian" pour enregistrer les valeurs héxadécimales en mémoire i.e. il commence par **les octets de poids faibles** et les envoie ensuite vers la RAM.

Exemple de boutisme avec `0xa1b2c3d4`

Big-Endian

a1	b2	c3	d4
0	1	2	3

Little-Endian

d4	c3	b2	a1
0	1	2	3

L'hexadécimal permet une correspondance immédiate vers le binaire. Ainsi, pour convertir un mot en hexadécimal en binaire il suffit de connaître le codage des valeurs de `0` à `F`.

La représentation des valeurs négatives se fait par la méthode du **complément à 2** : $-x = \overline{x} + 1$

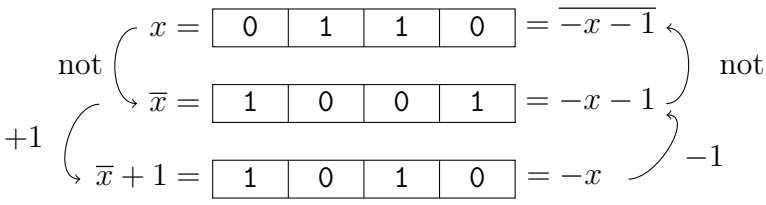
```
Correspondance hexa/binaire
 en logique et arithmétique

0 : 0000    8 : 1000 : -8
1 : 0001    9 : 0001 : -7
2 : 0010    A : 1010 : -6
3 : 0011    B : 1011 : -5
4 : 0100    C : 1100 : -4
5 : 0101    D : 1101 : -3
6 : 0110    E : 1110 : -2
7 : 0111    F : 1111 : -1

Sur N bits on a donc :
en logique : x ∈ ℕ
et x ∈ ⟦0; 2^N − 1⟧

en arth. : x ∈ ℤ
et x ∈ ⟦−2^(N−1); 2^(N−1) − 1⟧
```

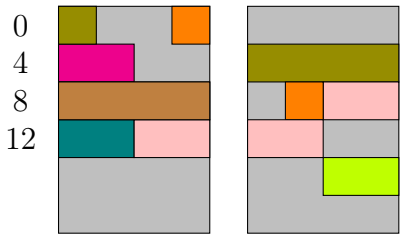
Exemple pour $x =$ `0x6` :



En logique : $x = \sum_{i=0}^{N-1}(2^i * x[i])$

En arth. : $x = -2^{N-1} * x[N-1] + \sum_{i=0}^{N-2}(2^i * x[i])$

Les données et les instructions doivent être **alignées en mémoire** le processeur part en exception si cette règle n'est pas respectée. Être aligné en mémoire signifie que l'adresse doit être un multiple de la taille de sa donnée i.e. que l'adresse d'un mot doit être un multiple de 4, celle d'un demi-mot doit être un multiple de 2 et par définition une donnée de 1 octet est toujours aligné en mémoire.



Ici dans la mémoire de droite les données représentées en rose ne sont pas alignées. Le bloc rose à une taille de 4 octets mais son adresse est à la 10ème case, 10 n'est pas multiple de 4. Dans la mémoire de gauche toute les données sont correctement alignées toutes les adresses sont multiples de la taille de leur donnée.

3 Corrigé TD1

3.1 Exercice 1 :

Écrire une suite d'instructions permettant de d'initialiser le registre R5 à 0.

```
add r5, r0, r0   # r5 <- 0 + 0
xor r5, r5, r5   # r5 <- r5 != r5
and r5, r0, r0   # r5 <- r0 & r0
sll r5, r0, 8    # r5 <- r0 >> 8
```

3.2 Exercice 2 :

Écrire une suite d'instructions permettant de copier le contenu du registre R6 dans R5.

```
or   r5, r6, r0 # r5 <- r6 || 0
addu r5, r6, r0 # r5 <- r6 + 0
sll  r5, r6, 0  # r5 <- r6 << 0
```

4 Corrigé TD2

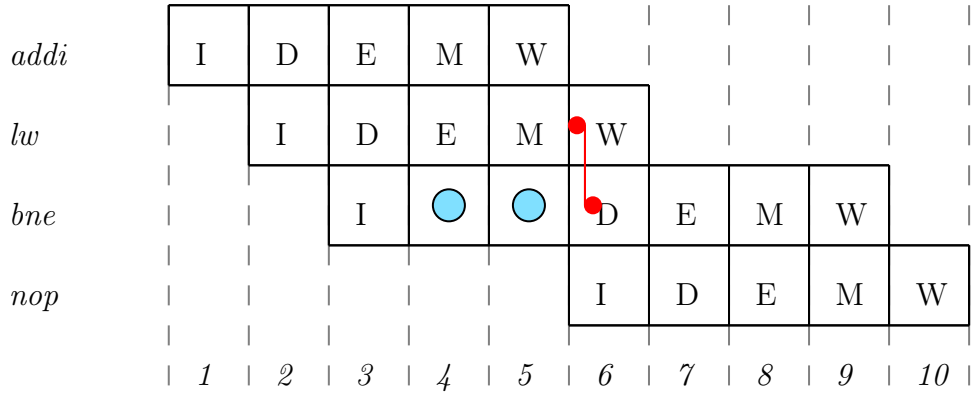
5 Corrigé TD3

6 Corrigé TD4

6.1 Exercice 1 :

Écrire en assembleur Mips-32 la boucle principale de GetListLength . On suppose que R4 contient l'adresse de la liste (pt). Analyser l'exécution de la boucle à l'aide d'un schéma simplifié. On considère que R4 contient *pt et R2 la variable i.

```
            ori     r2, r0, 0
            beq     r4, r0, endwhile
            nop
while :
            addi    r2, r2, 1
            lw      r4, 0(r4)
            bne     r4, r0, while
            nop
endwhile:
            jr      r31
            nop
```



#cycles/iter = 6
#inst/iter = 4
CPI = $\frac{6}{4}$ = 1,5 cycles/inst
CPI_{utile} = $\frac{6}{3}$ = 2 cycles/inst

6.2 Exercice 2 :

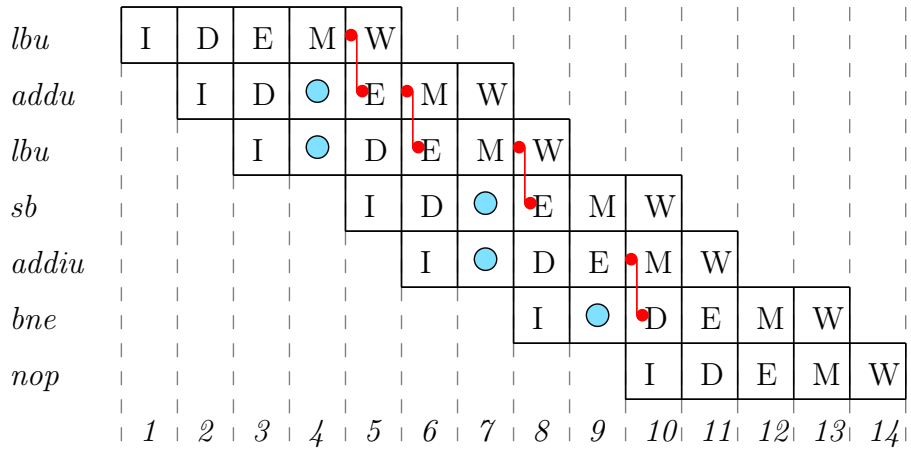
6.2.1 Question a

Modifier le programme pour qu'il soit exécutable sur le Mips-32.

```
1 loop:
2         lbu     r8, 0(r4)       # lire un pixel
3         addu    r8, r8, r5
4         lbu     r9, 0(r8)       # lire f(pixel)
5         sb      r9, 0(r4)
6
7         addiu   r4, r4, 1
8         bne     r4, r10, loop
9         nop                     # on ajoute un nop (delayed slot)
```

6.2.2 Question b

Analyser, à l'aide d'un schéma simplifié, l'exécution de ce programme dans le Mips. Calculer le nombre de cycles pour effectuer une itération.



#cycles/iter = 10
#inst/iter = 7
CPI = $\frac{10}{7}$ = 1,4 cycles/inst
CPI_{utile} = $\frac{10}{6}$ = 1,6 cycles/inst

6.2.3 Question c

Optimiser le code en changeant l'ordre des instructions de manière à obtenir un CPI et un CPI-utile de 1.

Étape 1 : Trouver les dépendances de données

```
loop:
    (1) lbu     r8, 0(r4)       # lire un pixel
    (2) addu    r8, r8, r5
    (3) lbu     r9, 0(r8)       # lire f(pixel)
    (4) sb      r9, 0(r4)

    (5) addiu   r4, r4, 1
    (6) bne     r4, r10, loop
    (7) nop                     # on ajoute un nop (delayed slot)
```

→ : "est dépendant de"

(2) → (1) [R8],
(3) → (2) [R8],
(4) → (3) [R9],
(6) → (5) [R4]

Étape 2 : Identification des cycles perdus
(2) → (1) : 1 cycle perdu,
(4) → (3) : 1 cycle perdu,
(6) → (5) : 1 cycle perdu,
(7) : 1 cycle perdu

Étape 3 : Schéma des dépendances
Attention ici la flèche n'a pas le même sens qu'à l'étape précédente ! Chaque ligne désigne une "chaîne" de dépendance. Les instructions qui ne sont pas reliées entre elles sont indépendantes, l'une de l'autre.
Il est conseillé de dessiner ce graphe à la vertical à côté du code.

→ : " **dépend de**"
Le numéro par dessus la flèche désigne le nombre de cycle perdu.

(1) $\xrightarrow{1}$ (2) → (3) $\xrightarrow{1}$ (4)
(5) $\xrightarrow{1}$ (6)
(7)

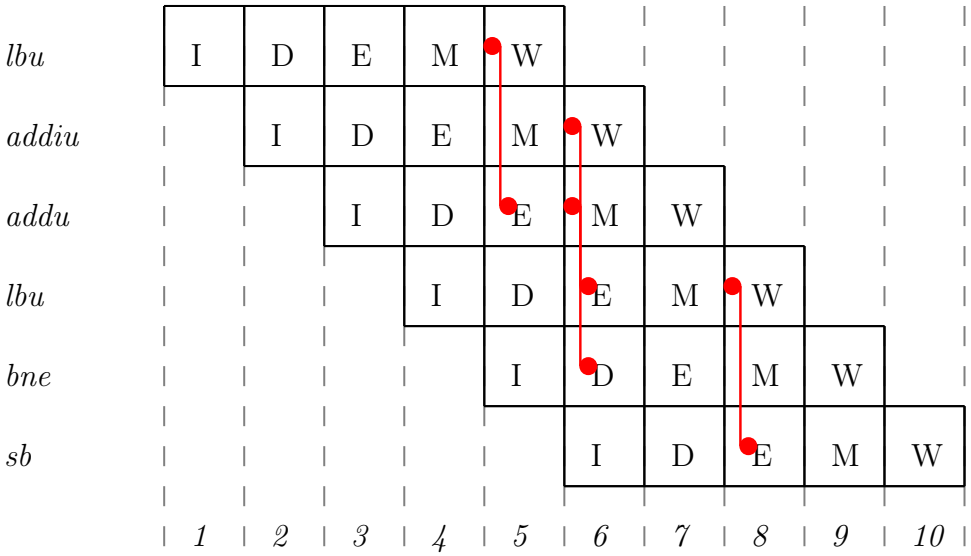
Ainsi (4) et (5) ne sont pas dépendantes.

Étape 4 : Réordonnacement du code
Le but est de combler les cycles perdus par des instructions.

Le schéma des dependances donne ainsi :
(1) → (5) → (2) → (3) → (6) → (4)

```
loop:
    (1) lbu     r8, 0(r4)       # lire un pixel
    (5) addiu   r4, r4, 1
    (2) addu    r8, r8, r5
    (3) lbu     r9, 0(r8)       # lire f(pixel)
    (6) bne     r4, r10, loop
    (4) sb      r9, -1(r4)      # r4 est incremente avant le sb d'ou -1
```

L'incrémentation de r4 (5) se fait avant qu'on y ait enregistré la valeur de r9, ainsi, lors de l'execution de (4), r4 contient la prochaine adresse d'où le `-1(r4)` et non `0(r4)`.



#cycles/iter = 6
#inst/iter = 6
CPI = CPI_{utile} = $\frac{6}{6}$ = 1 cycles/inst

6.2.4 Question d

Optimiser le code en utilisant la technique du "pipeline logiciel". Calculer le nombre de cycles par itération.

Optimisation par déroulage de boucle :
À chaque itération, dans le code non-optimisé, on ne traitait qu'un seul élèment. Si l'on en traite 2 par itération, le temps d'execution de la boucle en sera subséquement réduit. De même que si l'on en traite 3 et ainsi de suite. Le nombre d'instruction se trouve quand à lui augmenter et permet plus de réordonnancement.

Pourquoi traiter 2 élèments par itérations est plus interessant que d'en traiter 3, 4 ou plus ?
Une des raisons est lorsque le nombre d'éléments traités par itération augmente, la proportion des overheads (tels que les instructions de contrôle de la boucle et les instructions de branchement) par rapport au calcul réel peut diminuer. Cela signifie que l'impact relatif de l'overhead de la boucle sur le temps d'exécution total devient plus faible à mesure que l'on traite plus d'éléments par itération. Par conséquent, le bénéfice tiré de la réduction de l'overhead de la boucle peut ne pas être aussi prononcé lorsque l'on passe de 3 éléments à 4 éléments par itération.

```
loop:
        lbu     r8, 0(r4)
        addu    r8, r8, r5
        lbu     r9, 0(r8)
        sb      r9, 0(r4)

        lbu     r10, 1(r4)
        addu    r6, r6, r5
        lbu     r7, 0(r6)
        sb      r7, 1(r4)

        addiu   r4, r4, 2
        bne     r4, r10, loop
        nop
```

- Séparer le **traitement** de la gestion de la boucle
- Déterminer les dépendances de données dans le **traitement**
- Indiquer les pertes de performances (cycles de gel)
- Découper le traitement sur ces pertes (en étages qu'on nomera E1, E2, ..., Ei
- Vérifier que les règles du pipeline sont respectées :
  - Chaque étape et opération doit être séparée par un registre : si non respecté, on ajoute des instructions de transfert ou on modifie le découpage
  - Chaque matériel est utiliser dans un unique étage : si non respectés, renommer les registres (en utiliser d'autres)
- Réorganiser sous la forme $En_i, ..., E0_{n+i}$
- Effectuer le réordonnancement de la gestion de boucle

```
loop:
      i  lbu     r8, 0(r4)
    i-1  addu    r8, r8, r5
         lbu     r9, 0(r8)
    i-2  sb      r9, 0(r4)

         addiu   r4, r4, 1
         bne     r4, r10, loop
         nop
```

7 Corrigé TD5

8 Corrigé TD6

9 Astuce pour trouver le nombre de bypass

10 Liens utiles

https ://www.irisa.fr/caps/projects/TechnologicalSurvey/micro/PI-957-html/rapport.html