

Fonctionnement des Ordinateurs

TP2 - Programmation en langage d'assemblage MIPS

B. QUOTIN
Faculté des Sciences
Université de Mons

Résumé

L'objectif de ce TP est de renforcer votre compréhension du jeu d'instructions MIPS. Il s'agit de bien comprendre le rôle des différentes instructions : arithmétiques/logiques, load/store, branchements (in)conditionnels et appels de procédures.

Il vous est proposé à cet effet d'implémenter un certain nombre de petits programmes en langage d'assemblage MIPS et de les exécuter dans un simulateur appelé SPIM ou QTSPIM développé par James Larus de Microsoft Research.

Table des matières

1	Démarrer avec SPIM	1
1.1	Interface graphique de SPIM	1
1.2	Votre premier programme	2
1.3	Initialisation de SPIM	3
1.4	Points d'arrêt (<i>breakpoints</i>)	4
1.5	Langage d'assemblage	4
2	Exercices	6
2.1	Boucle à 5 itérations	6
2.2	Affichage d'un entier	7
2.3	Affichage d'une chaîne de caractères	8
2.4	Lecture d'entiers	9
2.5	Somme de deux entiers	10
2.6	Conversion en binaire	11
2.7	Structure de contrôle <i>switch</i>	12
2.8	Conversion en hexadécimal	14
2.9	Appel de fonction	16
2.10	Passage d'arguments et résultat	17
2.11	Manipulation de chaînes de caractères	18
2.12	Multiplication... récursive	20
2.13	Recherche dans un tableau	22
2.14	Idées d'exercices supplémentaires	23

A	Annexes	24
A.1	Appels système	24
A.2	Types d'instructions MIPS	24
A.3	Directives de l'assembleur	25

1 Démarrer avec SPIM

Le simulateur que nous allons utiliser lors de ces travaux pratiques est QTSPIM, une version graphique du simulateur SPIM développé par James Larus de Microsoft Research. Ce simulateur est déjà installé dans les salles informatiques, mais vous pouvez l'installer sur votre propre ordinateur. Il est disponible gratuitement à l'adresse suivante : <https://sourceforge.net/projects/spimsimulator/files/>.

1.1 Interface graphique de SPIM

Une copie de l'interface graphique de SPIM est présentée à la Figure 1. A gauche, dans le cadre **rouge**, le contenu des registres est mis à jour en temps-réel, au fur et à mesure de l'exécution des instructions. Il est possible d'y consulter la valeur de tous les registres discutés lors de ce cours : *program counter* (PC), banque de registres GPR (R[i]), registres LO et HI.

Au centre, dans le cadre **bleu**, le contenu de la mémoire est accessible. Les onglets "Data" et "Text" permettent de consulter respectivement la mémoire de données ou d'instructions. Dans la mémoire d'instructions, visible sur la Figure, chaque ligne représente une instruction et contient son emplacement en mémoire (adresse), son code binaire sur 32 bits, et son expression en langage d'assemblage. La partie mémoire d'instructions est découpée en deux zones, celle destinée aux programmes utilisateurs que nous allons écrire (*User Text Segment*) se situe entre les adresses 0x00400000 et 0x00440000, tandis que la mémoire réservée au système (*Kernel Text Segment*) se situe entre les adresses 0x80000000 et 0x80010000. Le simulateur démarre l'exécution des programmes à l'adresse 0x00400000.

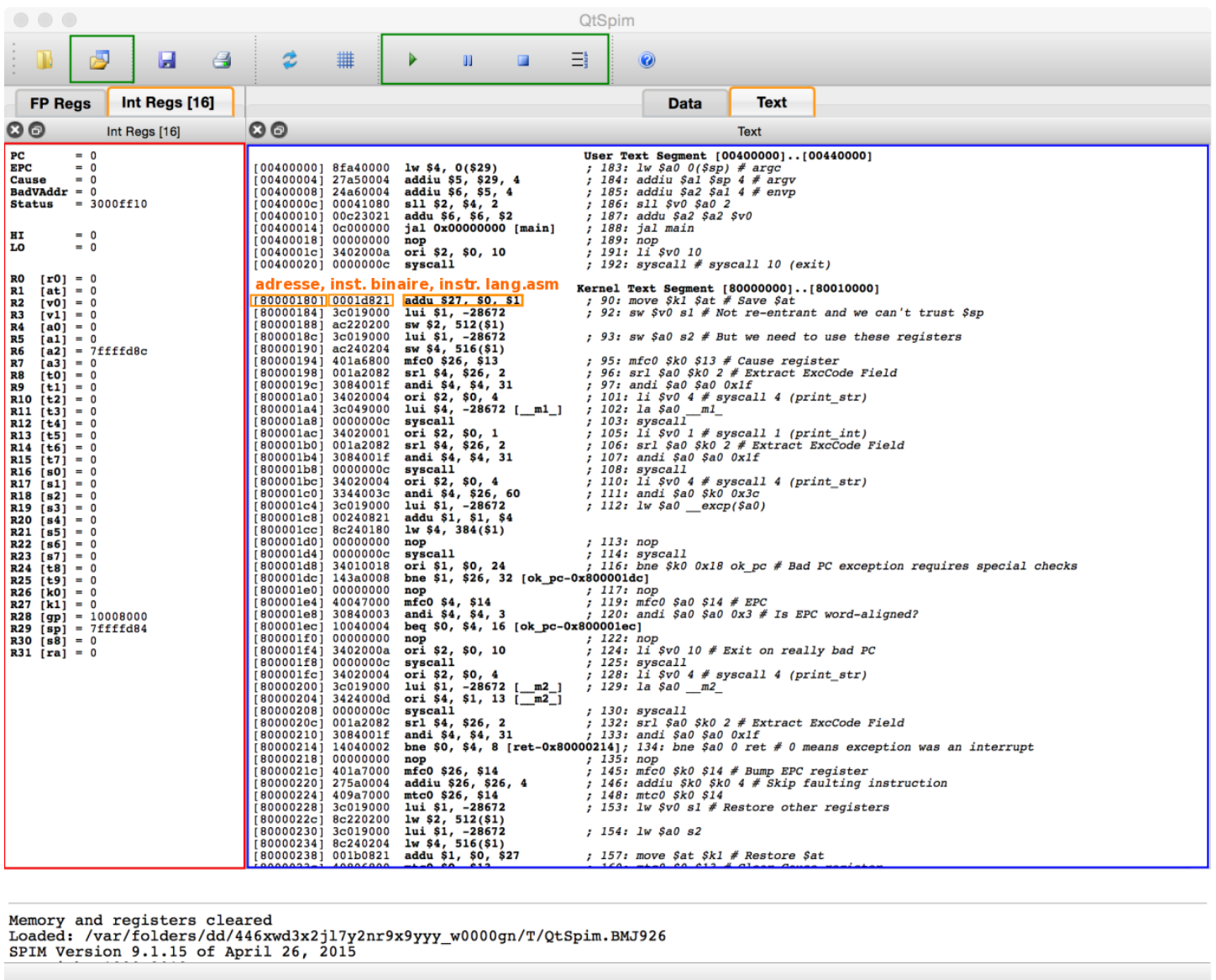


FIGURE 1 – Fenêtre principale de QTSPIM.

Dans la partie haute de l'écran, plusieurs boutons importants sont entourés de cadres **verts** : ré-initialiser et charger un programme

(📁), lancer l'exécution (▶), mettre l'exécution en pause (⏸), arrêter l'exécution (■) et avancer d'une seule instruction (⏮). La Table 1 fournit une brève description du rôle de chaque bouton.









Bouton	Description
	Charger un fichier.
	Charger un fichier et ré-initialiser le simulateur.
	Remettre à zéro les registres.
	Ré-initialiser le simulateur.
	Exécuter le programme jusqu'à sa fin.
	Mettre le programme en pause.
	Arrêter l'exécution du programme.
	Avancer d'une instruction.

TABLE 1 – Boutons de QTSPIM.

1.2 Votre premier programme

Pour découvrir le fonctionnement de SPIM, votre première mission consiste à créer un programme MIPS minimal, à le faire exécuter *pas-à-pas* par le simulateur et à observer les informations que le simulateur peut vous fournir durant l'exécution

Le premier programme à réaliser exécute une boucle sans fin. Il est illustré à la Figure 2. Le programme définit le label `main`. Il contient deux instructions exécutées séquentiellement : `nop` et `j main`. L'instruction `nop` ne fait rien. L'instruction `j main` effectue un branchement inconditionnel (*jump*) vers le label `main`.

Note importante : tous vos programmes doivent commencer au label `main`. Si ce label n'est pas défini, le simulateur générera une erreur.

Note importante : il est recommandé d'utiliser l'indentation pour rendre votre programme plus facile à lire. Utilisez également des lignes vides pour séparer des suites d'instructions qui réalisent des tâches différentes (par exemple des fonctions différentes).

```
1 | main:
2 |     nop
3 |     j     main
```

FIGURE 2 – Programme en langage d'assemblage MIPS qui effectue une boucle sans fin.

Avant d'exécuter le programme de la Figure 2, vous devez d'abord le copier ou le re-transcrire dans un fichier. Appelez celui-ci `spim-loop.s`. Vous pouvez utiliser à cet effet votre éditeur de texte préféré (p.ex. `gedit`, `atom` ou `vim`).

Pour exécuter ce programme, utilisez le bouton 📁 pour ré-initialiser le simulateur et charger le programme `spim-loop.s`. SPIM est capable de lire directement le code en langage d'assemblage. Il se charge de le convertir en langage machine. Vous pouvez observer que votre programme est chargé dans la mémoire d'instructions¹. Les deux instructions de votre programme se situent aux adresses suivantes :

```
0x00400024    main :  nop
0x00400028                j  main
```

Lancez l'exécution de ce programme avec le bouton ▶. Le simulateur est maintenant en train d'exécuter votre programme, mais celui-ci ne se terminera jamais car il effectue une boucle infinie. Pour interrompre l'exécution, utilisez le bouton ⏸. Vous devriez constater que le programme est arrêté sur l'une des deux instructions de votre programme (vérifiez la valeur du registre PC). Vous pouvez continuer à exécuter le programme pas à pas en utilisant le bouton ⏮. Vous devriez observer une exécution "en boucle" : la valeur du registre PC doit alterner entre `0x00400024` et `0x00400028`.

1. Pour trouver où votre programme se situe, vous pouvez demander au simulateur quelle est l'adresse du label `main`. Pour cela, vous pouvez utiliser le menu *Simulator / Display Symbols*. Vous devriez voir dans le bas du simulateur une ligne contenant `g main at 0x00400024`.

Questions. Vous pouvez observer les codes binaires des instructions dans la partie centrale de la fenêtre du simulateur, dans l'onglet *Text*. Toutes les instructions MIPS sont encodées sur 32-bits, cependant il existe plusieurs formats d'instructions : R, I ou J (appelés en Section A.2). Il est possible de décoder manuellement les instructions, comme le processeur le ferait. A cette fin, il suffit d'identifier d'abord l'*opcode* et sur base de celui-ci, déterminer le format (R, I ou J) de l'instruction. Il est alors possible d'extraire les opérandes de l'instruction.

Notez ci-dessous les codes des instructions `nop` et `j main` de votre programme. Identifiez l'*opcode*, le format et les opérandes. Pour l'instruction `j main`, vérifiez que l'adresse résultant du branchement correspond bien à l'adresse du label `main`.

Q1) Décodage de l'instruction `nop`.

Le code de l'instruction `nop` est `0x00000000`. L'*opcode* de cette instruction vaut `000000`, ce qui nous informe qu'il s'agit d'une instruction de type R. Toutes ses opérandes, `rs`, `rt`, `rd`, `shamt` et `function` valent 0. Le code de fonction indique que l'instruction est en fait `sll` (*shift left logical*) appliquée sur le registre `$zero`, avec un décalage de 0, ce qui n'a aucun effet.

Q2) Décodage de l'instruction `j main`.

Le code de l'instruction `nop` est `0x08100009`. L'*opcode* de cette instruction vaut `000010`₂, ce qui correspond à l'instruction `j` (*jump*). Il s'agit d'une instruction de type J. Elle ne possède qu'une opérande, `target`, qui vaut `0x100009`.

Pour déterminer l'adresse du saut, il faut calculer `PC31...28 || target || 02` où `PC31...28` représente les 4 bits de poids fort de l'actuel *program counter* et les deux bits `02` ajoutés à la fin correspondent à la multiplication par 4 (toutes les adresses d'instruction sont multiples de 4). Le registre PC vaut `0x400028` au moment où `j main` est exécuté. La nouvelle adresse vaut donc `0 || 0x100009 || 02 = 0x400024` ce qui provoque un retour à l'instruction précédente.

1.3 Initialisation de SPIM

La valeur du *programme counter* (PC) de SPIM vaut `0x00400000` suite à une ré-initialisation. Il commence donc à exécuter des instructions à cette adresse là. Pourtant le programme commence à l'adresse `0x00400024`, ce qui correspond au label `main`. Il est utile de comprendre l'origine de cette différence. Le simulateur ajoute des instructions supplémentaires à votre programme qui sont exécutées avant d'arriver à `main`.

Questions. Essayez de déterminer ce que font ces instructions. Il est probable que vous ne puissiez comprendre l'objectif exact de toutes les instructions. Cependant, vous devriez notamment y trouver

- une instruction `jal main`, ce qui signifie que `main` est une fonction,
- un appel système mettant fin au programme (voir `syscall` en Section A.1).

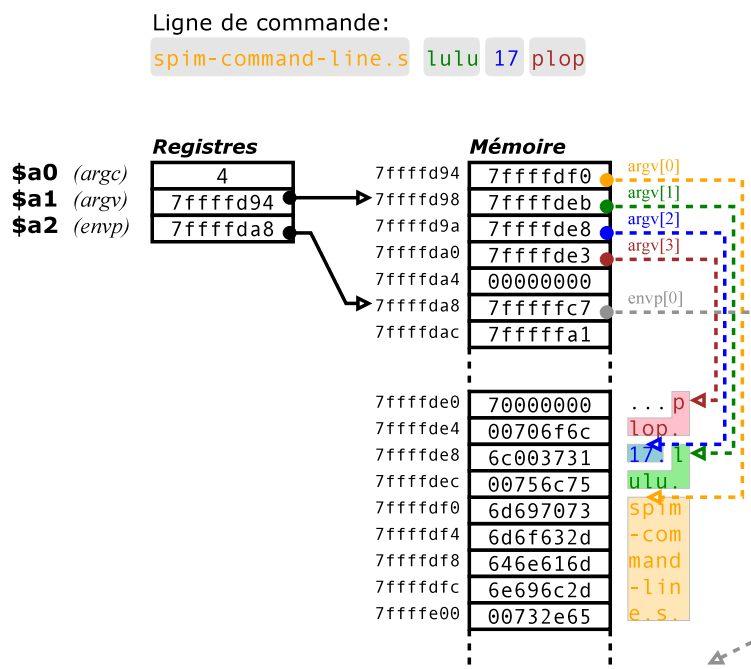
Q3) Notez ici les instructions ajoutées par SPIM.

```
lw    $a0, 0($sp)    # argc
addiu $a1, $sp, 4    # argv
addiu $a2, $a1, 4    # envp
sll   $v0, $a0, 2
addu  $a2, $a2, $v0
jal   main
nop
li    $v0, 10
syscall
```

Q4) Décrivez ce que vous avez compris des instructions ajoutées par SPIM.

La partie la plus importante pour nous est l'appel à la fonction `main` réalisé à la ligne 6 avec `jal main`. Il est important de noter que l'adresse de retour de `main` est placée dans le registre `$ra` par cette instruction. Les instructions aux lignes 8 et 9 mettent fin au programme à l'aide d'un appel système.

La compréhension des autres instructions est réservée aux plus curieux d'entre vous. Vous pouvez donc passer le paragraphe suivant. Les lignes 1-5 préparent les arguments de la fonction `main`, point d'entrée du programme. La signature de cette fonction est `main(int argc, char ** argv, char ** envp)`, un héritage du langage C. Les arguments de la fonction sont ceux passés en ligne de commande ainsi que l'environnement du programme. Ainsi, `$a0` contient `argc`, le nombre d'arguments. `$a1` contient l'adresse (`argv`) d'un tableau dont les cellules sont les adresses des chaînes de caractères correspondant aux arguments de la ligne de commande. De manière similaire, `$a2` contient l'adresse (`envp`) d'un tableau référençant les chaînes de caractères contenant les variables d'environnement. Ces tableaux sont situés au sommet de la pile. Cette organisation est illustrée ci-dessous sur un exemple, pour la partie `argc` et `argv`.



Pour passer des arguments en ligne de commande à votre programme, il faut aller dans le menu *Simulator / Run parameters*.

1.4 Points d'arrêt (breakpoints)

Une fonctionnalité importante de SPIM est celle des « points d'arrêt » (*breakpoints*). Il s'agit d'un moyen d'arrêter le simulateur lorsqu'il s'apprête à exécuter une instruction située à une adresse particulière.

La Figure 3 illustre comment définir un breakpoint à l'adresse de `main`, c'est-à-dire `0x00400024`. Il suffit d'effectuer un « clic droit » sur la ligne de l'instruction en question. Un menu contextuel apparaît dans lequel il faut sélectionner *Set Breakpoint*. Une petite main rouge apparaît à la gauche de l'instruction sur laquelle un point d'arrêt a été défini.

Une première utilisation des breakpoints est le passage des instructions ajoutées par SPIM à votre programme (voir Section 1.3). Avec un breakpoint défini sur `main`, il suffit de lancer l'exécution avec . L'exécution s'arrêtera dès que le simulateur arrivera à l'adresse de `main`. Il sera alors possible de continuer pas-à-pas avec .

1.5 Langage d'assemblage

Le simulateur supporte directement le langage d'assemblage MIPS. Il se charge de convertir un programme en langage d'assemblage vers le langage machine et de charger les données et instructions définies dans le programme vers les zones mémoire correspondantes. Les particularités du langage d'assemblage supporté par le simulateur sont résumées ci-dessous.

- Chaque instruction occupe une ligne : le mnémonique de l'instruction vient en premier, suivi des opérandes. Le registre destination est généralement le premier opérande.
- Des commentaires peuvent être écrits à la suite du caractère `#`. Le commentaire s'étend jusqu'à la fin de la ligne.
- Les noms des registres doivent être préfixés par le caractère `$`.

```

User Text Segment [00400000]..[00440000]
[00400000] 8fa40000 lw $4, 0($29) ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004 addiu $5, $29, 4 ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004 addiu $6, $5, 4 ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080 sll $2, $4, 2 ; 186: sll $v0 $a0 2
[00400010] 00c23021 addu $6, $6, $2 ; 187: addu $a2 $a2 $v0
[00400014] 0c100009 jal 0x00400024 [main] ; 188: jal main
[00400018] 00000000 nop ; 189: nop
[0040001c] 3402000a ori $2, $0, 10 ; 191: li $v0 10
[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)
[00400024] 3c011001 lui $1, 4097 [msg] ; 11: la $a0, msg
[00400028] 3424000c ori $4, $1, 12 [msg]
[0040002c] 34020004 ori $2, $0, 4 ; 12: li $v0, 4
[00400030] 0000000c syscall ; 13: syscall
[00400034] 34020005 ori $2, $0, 5 ; 15: li $v0, 5
[00400038] 0000000c syscall ; 16: syscall
[0040003c] 3c0a1001 lui $10, 4097 [jump_table]; 18: la $t2, jump_table
[00400040] 00021080 sll $2, $2, 2 ; 20: sll $v0, $v0, 2
[00400044] 01424020 add $8, $10, $2 ; 21: add $t0, $t2, $v0

User Text Segment [00400000]..[00440000]
[00400000] 8fa40000 lw $4, 0($29) ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004 addiu $5, $29, 4 ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004 addiu $6, $5, 4 ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080 sll $2, $4, 2 ; 186: sll $v0 $a0 2
[00400010] 00c23021 addu $6, $6, $2 ; 187: addu $a2 $a2 $v0
[00400014] 0c100009 jal 0x00400024 [main] ; 188: jal main
[00400018] 00000000 nop ; 189: nop
[0040001c] 3402000a ori $2, $0, 10 ; 191: li $v0 10
[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)
[00400024] 3c011001 lui $1, 4097 [msg] ; 11: la $a0, msg
[00400028] 3424000c ori $4, $1, 12 [msg]
[0040002c] 34020004 ori $2, $0, 4 ; 12: li $v0, 4
[00400030] 0000000c syscall ; 13: syscall
[00400034] 34020005 ori $2, $0, 5 ; 15: li $v0, 5
[00400038] 0000000c syscall ; 16: syscall
[0040003c] 3c0a1001 lui $10, 4097 [jump_table]; 18: la $t2, jump_table
[00400040] 00021080 sll $2, $2, 2 ; 20: sll $v0, $v0, 2
[00400044] 01424020 add $8, $10, $2 ; 21: add $t0, $t2, $v0

```

FIGURE 3 – Définir un « point d’arrêt » avec SPIM.

- Le point d’entrée du programme doit être désigné par le *label* `main`. La fin du programme se traduit par un retour vers l’appelant de `main`, en utilisant `jr $ra`.
- Plusieurs pseudo-instructions sont disponibles telles que
 - `li` (*load immediate*) : charge une constante dans un registre
 - `la` (*load address*) : charge une adresse dans un registre, typiquement à partir d’un *label*.
- Plusieurs directives sont supportées, voir Section A.3
- Plusieurs appels systèmes sont supportés, voir Section A.1

Attention ! Le simulateur est parfois assez permissif et se permet de traduire automatiquement (et silencieusement) certaines instructions qui ne devraient pas être permises. Dans certains cas, cela pourrait causer une confusion de votre part. Par exemple,

- L’instruction `add $a0, $a1, 5` ne devrait pas être permise car le troisième opérande devrait être un registre et non une constante. Pourtant, SPIM va la traduire automatiquement en `addi $a0, $a1, 5`.
- L’instruction `jr loop` sera traduite en `j loop`, mais il est important de comprendre que `jr` (*jump register*) devrait normalement prendre un registre comme opérande !

2 Exercices

2.1 Boucle à 5 itérations

Ecrivez un programme en langage d'assemblage MIPS qui effectue une boucle de 5 itérations. Le programme doit être placé dans un fichier nommé `spim-loop-5.s`. Le programme maintient le nombre d'itérations restant à effectuer à l'aide du registre `a0`. Ce registre est décrémenté à chaque itération de la boucle.

Exécutez votre programme pas à pas avec SPIM. Vérifiez que votre boucle se termine. Observez l'évolution du registre `a0` au cours des itérations successives de la boucle.

Les détails suivants sont importants pour résoudre cet exercice :

1. Les noms des registres doivent être préfixés du caractère `$`. Par exemple, l'instruction qui charge la valeur 5 dans le registre `a0` s'écrira `li $a0, 5`.
2. Il est nécessaire de retourner à l'appelant de `main` une fois la boucle exécutée. Ceci doit être réalisé avec l'instruction `jr $ra`.

Q5) Programme en langage d'assemblage MIPS effectuant une boucle à 5 itérations.

Deux solutions sont proposées ci-dessous. Celle de gauche correspond à une boucle `do...while` qui effectue toujours sa première itération. La solution de droite correspond à une boucle `while`. La première solution ne fonctionne pas correctement si le nombre d'itérations à effectuer est nul.

```
1 main:
2     li    $a0, 5
3 loop:
4     addi  $a0, -1
5     bgtz  $a0, loop
6     jr    $ra
```

```
1 main:
2     li    $a0, 5
3 loop:
4     bne   $a0, $zero, loop_end
5     addi  $a0, -1
6     b     loop
7 loop_end:
8     jr    $ra
```

L'instruction `li` est une pseudo-instruction. En quelle(s) instruction(s), SPIM l'a-t-il traduite ?

Q6) Traduction de la pseudo-instruction `li $a0, 5`.

La pseudo-instruction a été traduite en `ori $a0, $zero, 5`.

Si vous omettez l'instruction `jr` à la fin de votre programme, que se passe-t-il ? Confirmez votre réponse en exécutant une version de votre programme dans laquelle l'instruction `jr` est omise.

Q7) Comportement du programme sans instruction `jr`.

Des instructions indéfinies sont exécutées car le programme se poursuit à des adresses pour lesquelles nous n'avons pas défini les codes d'instruction !! Une erreur pourrait être générée par le simulateur.

D'où provient la valeur de `ra` utilisée dans l'instruction `jr $ra` ?

Q8) Comment la valeur de `ra` a-t-elle été obtenue ?

La valeur de `$ra` a été déterminée lors de l'appel à la fonction `main` effectué par l'appel `jal main` (voir code ajouté par le simulateur, en Section 1.3).

Décodez l'instruction de branchement conditionnel utilisée dans votre programme. De ce code d'instruction, dérivez la valeur immédiate `offset`. A quoi correspond cette valeur ? Y a-t-il une différence par rapport à ce qui a été expliqué durant le cours ?

Q9) Décodage de l'instruction de branchement conditionnel.

Le code de l'instruction est `0x1c80ffff`. Son *opcode* est `0001112`, ce qui correspond à l'instruction `bgtz` (*branch on greater than zero*). Il s'agit d'une instruction de type I. Ses opérandes sont `rs=($a0)`, `rt=0 ($zero)` et `offset=0xffff`.

La nouvelle valeur de PC est obtenue comme $PC + (\text{offset} \times 4)$, ce qui donne $PC - 4$. En effet, `0xffff` représente -1 en complément à deux sur 16 bits. Un retour de 4 octets en arrière fait donc passer à l'instruction précédente (`add`) si la condition du branchement est satisfaite, i.e. tant que $\$a0 > 0$.

Dans le cours, nous avons vu que lors d'un branchement, l'adresse est calculée comme $PC + 4 + (\text{offset} \times 4)$. La différence vient du fait que le simulateur ne représente pas parfaitement l'architecture MIPS. Le +4 de l'architecture réelle de MIPS vient d'une particularité de son implémentation du *pipeline* d'instructions^a.

Pour les étudiants curieux, voici le code qui serait généré pour un vrai processeur MIPS avec l'assembleur GNU `as`. La première colonne donne l'adresse de l'instruction et la deuxième colonne en donne le code. Notez l'ajout des deux instructions `nop`. Elles sont nécessaires car lors d'un branchement, l'instruction suivante est **toujours** exécutée !

```
400a54          loop:
400a54 2084ffff      addi $a0, $a0, -1
400a58 1c80ffff      bgtz $a0, 0x400a54 <loop>
400a5c 00000000      nop
400a60 03e00008      jr  ra
400a64 00000000      nop
```

Dans le **code de l'instruction `bgtz`**, on remarque que la valeur d'`offset` est `0xfffe` cette fois, ce qui correspond à -2. La valeur de PC est bien calculée comme $PC + 4 + (\text{offset} \times 4) = 0x400a58 + 4 + (-8) = 0x400a54$.

a. L'option *Delayed branches* de SPIM permet de mieux coller à l'architecture MIPS.

2.2 Affichage d'un entier

L'objectif de cet exercice est d'utiliser un appel système pour afficher un entier à la console. La façon dont un appel système est réalisé ainsi que la liste des appels système sont décrits en Section A.1. Pour afficher un entier à la console, l'appel système 1 (aussi appelé `print int`) va être invoqué à l'aide de l'instruction `syscall`. Le numéro de l'appel système doit avoir été placé **préalablement** dans le registre `v0`. Il en va de même pour la valeur de l'entier à afficher qui doit être placée dans le registre `a0`.

L'extrait de programme montré à la Figure 4 illustre comment afficher l'entier 603 à la console. Cet entier est placé dans le registre `a0` et le numéro d'appel système (1) est placé dans le registre `v0`. L'appel système est invoqué avec l'instruction `syscall`.

```
1 | # entier a afficher
2 | li    $a0, 603
3 | # numero de l'appel systeme
4 | li    $v0, 1
5 | syscall
```

FIGURE 4 – Invocation de l'appel système "print_int".

Dans cet exercice, vous devez modifier le programme précédent (boucle à 5 itérations) de sorte qu'à chaque itération la valeur du registre `a0` soit affichée à la console. Le programme résultant doit être sauvegardé dans le fichier `spim-loop-5-print.s`. Exécutez ce programme pour en vérifier le bon fonctionnement.

Note importante : la console de SPIM est placée dans une fenêtre séparée. Si celle-ci n'est pas visible, il suffit d'aller dans le menu *Window* et de cocher *Console*.

Q10) Programme en langage d'assemblage effectuant 5 itérations et affichant le nombre d'itérations restant à la console.

```
1 | main:
2 | li    $a0 5
```

```

3
4 loop5:
5     li      $v0, 1
6     syscall
7
8     addi    $a0, -1
9     bgtz    $a0, loop5
10
11     jr      $ra

```

2.3 Affichage d'une chaîne de caractères

Dans cet exercice, il vous est demandé d'utiliser l'appel système 4 (`print_string`, Section A.1) afin d'afficher la chaîne de caractères de votre choix à la console. Cet appel système prend un seul argument qui est l'adresse du premier caractère de la chaîne à afficher. Cet argument est passé dans le registre `a0`.

La première étape pour résoudre cet exercice est de définir une chaîne de caractères en mémoire. La Figure 5 illustre la représentation en mémoire de la chaîne de caractères "Hello World of MIPS". Chaque caractère est représenté par son code ASCII et occupe un octet en mémoire. Par exemple, le caractère 'H' est représenté par le code ASCII `0x48`. Les codes des caractères successifs de la chaîne sont placés en mémoire à des adresses consécutives. Ainsi, dans l'exemple, le premier caractère est stocké à l'adresse `0x10010000`, le second à l'adresse `0x10010001`, etc. Attention, ces adresses peuvent être différentes dans votre programme. Un *label* `str` permet de désigner l'adresse du premier caractère de la chaîne, soit `0x10010000`.

Il est important de remarquer que la longueur de la chaîne n'est pas stockée en mémoire. En revanche, la fin de la chaîne est indiquée par le 0. Ce code est visible à la Figure 5 : le dernier caractère de la chaîne, 'S', est suivi par le code 0. Cette représentation de chaînes de caractères est appelée *chaîne à zéro terminal* (*nul/zero-terminated string*).

str: 0x10010000	'H'
0x10010001	'e'
0x10010002	'l'
0x10010003	'l'
0x10010004	'o'
0x10010005	' '
0x10010011	'P'
0x10010012	'S'
0x10010013	0

FIGURE 5 – Chaîne de caractères à zéro terminal en mémoire.

```

1      .data
2  str:  .ascii "Hello World of MIPS\n"
3
4      .text
5  main:
6      # suite du programme ...
7      la    $a0, str
8      li    $v0, 4
9      syscall

```

FIGURE 6 – Définition d'une chaîne de caractères, obtention de son adresse et invocation de l'appel système `print_string` (4).

Définir une telle chaîne en mémoire peut être réalisé à l'aide de directives du langage d'assemblage (voir Section A.3), comme montré à la Figure 6. La directive `.ascii` crée une chaîne à zéro terminal en mémoire, sur base d'un littéral chaîne de caractères entouré de guillemets ("). La chaîne est définie dans le segment de données du programme, grâce à la directive `.data`. Ce qui est déclaré dans le segment de données est placé en mémoire RAM. De même, la directive `.text` indique que ce qui suit est placé dans le segment d'instructions (et est typiquement placé en mémoire ROM/Flash). Afin de pouvoir faire référence à la chaîne de caractères, elle est précédée d'un label (`str`). L'adresse mémoire de la chaîne de caractères peut ainsi être récupérée et copiée dans un registre à l'aide de la pseudo-instruction `la` (*Load Address*).

Question Veuillez écrire un programme nommé `spim-print-str.s` qui affiche une chaîne de caractères de votre choix à la console. Vérifiez le bon fonctionnement de votre programme en le chargeant et l'exécutant à l'aide de SPIM.

Q11) Programme affichant une chaîne de caractères à la console.

```

1      .data
2      .globl msg
3  msg:  .ascii "Hello World of MIPS\n"
4
5      .text
6  main:

```

```

7      la      $a0, msg
8      li      $v0, 4
9      syscall
10     jr      $ra

```

Question Afin de vérifier votre compréhension de l'encodage d'une chaîne de caractères en mémoire, veuillez donner la valeur du mot de 32 bits stocké à l'adresse désignée par le *label* `str`. Afin d'obtenir l'adresse à laquelle correspond le *label* `str`, vous pouvez exécuter votre programme jusqu'à passer la pseudo-instruction `la $a0, str`. L'adresse sera alors chargée dans le registre `a0`. Une autre possibilité consiste à définir `str` comme un symbole global (voir directive `.globl` en Section A.3) et lister les symboles avec le menu *Simulator / Display Symbols*.

Allez ensuite voir dans la zone de mémoire de données via le panneau central du simulateur, onglet *Data*. Dans ce panneau, chaque ligne donne le contenu de 16 octets consécutifs en mémoire. L'adresse du premier de ces octets est en début de ligne.

Notez ci-dessous, en hexadécimal, l'adresse correspondant à `str` et le mot de 32 bits situé à l'adresse à cette adresse. D'où vient la valeur de ce mot ?

Q12) Valeur du mot de 32 bits situé à l'adresse `str`, en hexadécimal.

Le *label* `str` désigne l'adresse `0x10010000`. Le mot de 32 bits (4 octets) situé à cette adresse vaut `6c6c6548`. Il s'agit des codes ASCII des 4 premiers caractères de la chaîne.

Caractère	Code ASCII
H	0x48
e	0x65
l	0x6c

Ces caractères semblent apparaître en ordres opposés dans le mot de 32 bits et dans la chaîne. Ceci est dû au fait que sur les processeurs que nous utilisons, le simulateur SPIM utilise la convention *little-endian* pour représenter des mots de plusieurs octets. L'octet de plus petite adresse (H / 0x48) est donc celui de **plus faible poids** dans le mot `6c6c6548`.

2.4 Lecture d'entiers

Dans cet exercice, il vous est demandé de produire un programme qui demande, en boucle, un entier à l'utilisateur. Si cet entier est inférieur à 0, la boucle se termine. Si l'entier est strictement supérieur à 10, la chaîne de caractères "trop grand" est affichée à la console. Sinon, la chaîne de caractères "valeur acceptée" est affichée à la console.

Pour demander un entier à l'utilisateur, utilisez l'appel système `read_int (5)`. Celui-ci retourne l'entier lu à la console dans le registre `v0`. Attention, si la chaîne ne correspond pas à un entier, l'appel système retourne 0.

Veuillez écrire un programme nommé `spim-read-int.s` répondant à la description donnée ci-dessus. Vérifiez le bon fonctionnement de votre programme en le chargeant et l'exécutant à l'aide de SPIM. Fournissez des valeurs différentes permettant de tester toutes les conditions du programme.

Attention, ce programme commence à devenir plus complexe. Il devrait s'étaler sur environ 30 lignes. Assurez-vous de bien le structurer, en employant l'indentation à bon escient, en insérant des lignes vides pour séparer les différentes tâches et en ajoutant éventuellement des commentaires (en utilisant le caractère spécial `#`).

Q13) Programme lisant des entiers à la console.

```

1      .data
2      msg1: .asciiz "Entrez un entier:\n"
3      msg2: .asciiz "Trop grand.\n"
4      msg3: .asciiz "Valeur acceptee: "
5      endl: .asciiz "\n"
6
7      .text
8      main:
9          la      $a0, msg1
10         li      $v0, 4
11         syscall
12
13         li      $v0, 5
14         syscall

```

```

15      move    $t0, $v0
16
17      bltz    $v0, end_loop
18      bgt     $v0, 10, too_large
19
20      la      $a0, msg3
21      li      $v0, 4
22      syscall
23
24      move    $a0, $t0
25      li      $v0, 1
26      syscall
27
28      la      $a0, endl
29      li      $v0, 4
30      syscall
31      j       main
32
33 too_large:
34      la      $a0, msg2
35      li      $v0, 4
36      syscall
37      j       main
38
39 end_loop:
40      jr      $ra

```

Fournissez une variante du programme ci-dessus qui, en plus, affiche la valeur donnée par l'utilisateur lorsque celle-ci est acceptée, c'est-à-dire lorsqu'elle est située dans l'intervalle $[0, 10]$.

Q14) Programme lisant des entiers à la console (variante).

2.5 Somme de deux entiers

Dans cet exercice, il vous est demandé d'écrire un programme qui demande deux entiers à l'utilisateur et qui en affiche la somme. Placez le programme résultant dans un fichier nommé `spim-add-int.s`. Testez ce programme avec plusieurs couples d'entiers.

Q15) Programme calculant la somme de deux entiers lus à la console.

```

1  # Effectue la somme de deux entiers demandés à l'utilisateur.
2  # Subtilité : il faut parfois sauvegarder les entiers lus
3  #           car ils sont écrasés par des syscalls suivants.
4  #           ici, on utilise $t0 pour cette sauvegarde.
5
6      .data
7  msg1: .asciiz "Entrez le premier entier: "
8  msg2: .asciiz "Entrez le second entier: "
9  msg3: .asciiz "La somme vaut : "
10 endl: .asciiz "\n"
11
12      .text
13 main:
14      la      $a0, msg1
15      li      $v0, 4
16      syscall
17
18      li      $v0, 5
19      syscall
20      move    $t0, $v0
21
22      la      $a0, msg2
23      li      $v0, 4
24      syscall
25
26      la      $a0, msg3
27      li      $v0, 4
28      syscall
29
30      jr      $ra

```

```

23      syscall
24
25      li    $v0, 5
26      syscall
27
28      add    $t0, $t0, $v0
29
30      la     $a0, msg3
31      li     $v0, 4
32      syscall
33
34      move   $a0, $t0
35      li     $v0, 1
36      syscall
37
38      la     $a0, endl
39      li     $v0, 4
40      syscall
41
42      jr     $ra

```

2.6 Conversion en binaire

Dans cet exercice, il vous est demandé d'écrire un programme qui demande un entier à l'utilisateur et qui affiche cet entier en binaire. Utilisez à cet effet la méthode effectuant des divisions euclidiennes successives, vue au Chapitre 2 du cours. Pour résoudre cet exercice, vous aurez besoin de l'instruction `divu x , y` . Pour rappel, cette instruction calcule le quotient et le reste de la division de x par y (en considérant que ces nombres sont non-signés). Le quotient est placé dans le registre `LO` tandis que le reste est placé dans le registre `HI`. Votre algorithme peut produire les bits de la représentation binaire en commençant par le bit de poids le plus faible.

Placez le programme résultant dans un fichier nommé `spim-to-binary.s`. Testez ce programme avec plusieurs nombres naturels.

Q16) Programme convertissant un naturel en binaire.

Deux solutions différentes sont proposées ci-dessous. A gauche, on utilise la division euclidienne par 2. A noter que dans cette seconde solution, le nombre est affiché en commençant par les bits de poids faible. De plus, pour effectuer la division par 2, le reste est obtenu avec `andi` et le quotient avec `srl`. A droite, on utilise un masque binaire valant 2^i et l'on effectue un ET bit-à-bit pour faire ressortir la valeur du bit b_i .

```

1  # Conversion en binaire avec div. euclid.
2
3  .data
4  endl: .asciiz "\n"
5
6  .text
7  main:
8      li $v0, 5
9      syscall
10     move $a1, $v0
11
12     bne $a1, $zero, non_zero
13     li $v0, 1
14     li $a0, 0
15     syscall
16     b     end
17
18 non_zero:
19     andi $a0, $a1, 1
20     li $v0, 1
21     syscall
22     li $a2, 1
23     srl $a1, $a1, $a2
24     bne $a1, $zero, non_zero
25
26 end:
27     la $a0, endl
28     li $v0, 4
29     syscall
30
31     jr $ra

```

```

1  # Conversion en binaire avec masque binaire
2
3  .data
4  endl: .asciiz "\n"
5
6  .text
7  main:
8      li $v0, 5
9      syscall
10     move $a1, $v0
11
12     li $a2, 1
13     li $a3, 31
14     sll $a2, $a2, $a3
15 loop:
16     and $a3, $a1, $a2
17     bne $a3, $zero, is_one
18 is_zero:
19     li $a0, 0
20     b     print_int
21 is_one:
22     li $a0, 1
23
24 print_int:
25     li $v0, 1 # print_int
26     syscall
27
28     srl $a2, $a2, 1
29     bne $a2, $zero, loop
30
31     la $a0, endl
32     li $v0, 4
33     syscall
34
35     jr $ra

```

2.7 Structure de contrôle *switch*

Le chapitre 4 du cours a montré comment une structure de contrôle *switch* peut être exprimée en langage d'assemblage, en utilisant une séquence d'instructions de test avec la variable de décision. Dans cette approche, il y a une instruction de branchement conditionnel par cas du *switch*. Cette approche est illustrée à la Figure 7.

Si les cas sont consécutifs, il est possible d'utiliser une autre approche plus économe en terme d'instructions : l'utilisation d'une table de branchement. Un exemple d'un *switch* qui se prête à une telle implémentation est illustré à la Figure 8. Une table de branchement contient les adresses auxquelles il faut brancher pour chacun des cas. L'index dans la table est obtenue à partir de la variable de décision (*a* dans l'exemple). La Figure 8 illustre une table de branchement composée de 4 adresses correspondant aux cas 5 à 8 (via les labels *case5* à *case8*). La directive *.word* permet de définir un ou plusieurs mode de 32 bits en mémoire. Pour effectuer le branchement, il faut d'abord déterminer l'index dans la table. Dans le cas de l'exemple, cet index sera $a - 5$. Cet index est converti en adresse mémoire, de façon à aller lire l'adresse de branchement correspondante dans la table, à l'aide de l'instruction *lw* (*Load Word*). Dans l'exemple, cette adresse a la forme $tab + 4 \times (a - 5)$. Il suffit ensuite de brancher à l'adresse lue dans la table, à l'aide de l'instruction *jr* (*Jump Register*).

Placez le programme résultant dans un fichier nommé *switch-table.s*. Testez ce programme avec plusieurs valeur de *a* de façon à vérifier le bon fonctionnement de votre implémentation.

```

1 switch (a) {
2 case 5:
3     /* code du cas 5 */
4     break;
5 case 13:
6     /* code du cas 13 */
7     break;
8 case 37:
9     /* code du cas 37 */
10    break;
11 default:
12    /* code autre */
13 }

```

```

1
2 case5:
3     li $t0, 5
4     bne $a0, $t0, case13
5     # code du cas 5
6     j end_switch
7
8 case13:
9     li $t0, 13
10    bne $a0, $t0, case37
11    # code du cas 13
12    j end_switch
13
14 case37:
15    li $t0, 37
16    bne $a0, $t0, case_default
17    # code du cas 37
18    j end_switch
19
20 case_default:
21     # code autre
22
23 end_switch:

```

FIGURE 7 – Implémentation possible d'une structure switch en langage d'assemblage MIPS.

```

1 switch (a) {
2 case 5:
3     /* code du cas 5 */
4     break;
5 case 6:
6     /* code du cas 6 */
7     break;
8 case 7:
9     /* code du cas 7 */
10    break;
11 case 8:
12    /* code du cas 8 */
13    break;
14 default:
15    /* code autre */
16 }

```

```

1 .data
2 tab: .word case5, case6, case7, case8
3
4 .text
5 main:
6     # chercher adresse branchement
7     # dans table
8     lw ...
9     # brancher vers adresse obtenue,
10    jr ...
11 case5:
12
13 case6:
14
15 case7:
16
17 case8:
18
19 end_switch
20     jr $ra

```

FIGURE 8 – Structure switch avec cas consécutifs.

Q17) Programme implémentant une structure switch avec une table de branchement.

```

1 .data
2 jump_table:
3     .word case5, case6, case7, case8
4
5 msg: .asciiz "Entrez un nombre: "
6 msg_case5: .asciiz "Cas 5.\n"

```

```

7 msg_case6: .ascii "Cas 6.\n"
8 msg_case7: .ascii "Cas 7.\n"
9 msg_case8: .ascii "Cas 8.\n"
10 msg_case_default: .ascii "Cas par défaut.\nFin.\n"
11
12 .text
13 main:
14     la $a0, msg
15     li $v0, 4
16     syscall
17
18     li $v0, 5
19     syscall
20
21     # verifier ici si 5 <= $v0 <= 8, sinon case "default"
22     li $a0, 5
23     blt $v0, $a0, case_default
24     li $a0, 8
25     bgt $v0, $a0, case_default
26
27     # charger adresse de branchement depuis table
28     la $t2, jump_table
29     addiu $v0, $v0, -5
30     sll $v0, $v0, 2
31     add $t0, $t2, $v0
32     lw $t2, 0($t0)
33     jr $t2
34 case5:
35     la $a0, msg_case5
36     li $v0, 4
37     syscall
38     j      end_case
39 case6:
40     la $a0, msg_case6
41     li $v0, 4
42     syscall
43     j      end_case
44 case7:
45     la $a0, msg_case7
46     li $v0, 4
47     syscall
48     j      end_case
49 case8:
50     la $a0, msg_case8
51     li $v0, 4
52     syscall
53     j      end_case
54 case_default:
55     la $a0, msg_case_default
56     li $v0, 4
57     syscall
58     j      end
59 end_case:
60
61     j main
62
63 end:
64     jr $ra

```

2.8 Conversion en hexadécimal

Adaptez le programme de la Section 2.6 de façon à effectuer la conversion en hexadécimal. Deux adaptations sont nécessaires. La première est facile : il faut changer le diviseur utilisé dans l'algorithme. La seconde est plus complexe : il faut pouvoir afficher des symboles hexadécimaux à la console, en fonction des restes produits par l'algorithme.

Supposons que vous disposiez de 16 chaînes de caractères, correspondant à chacun des symboles hexadécimaux, telles qu'illustrées

à la Figure 9. Ces chaînes sont consécutives en mémoire. Par exemple, l'adresse de "0" est *hex*, alors que "1" est *hex+2*, "2" est *hex+4*, etc. Il est donc possible de calculer l'adresse de chaque chaîne en fonction de l'adresse de la première ("0") et d'un décalage proportionnel à la valeur du symbole à afficher... Le décalage est un multiple de 2 car chaque chaîne est composée du caractère hexadécimal et du caractère 0 signalant la fin de chaîne.

```
1 | .data
2 | hex: .asciiz "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"
```

FIGURE 9 – Définition de 16 chaînes consécutives en mémoire, chacune correspondant à un symbole hexadécimal.

Une autre possibilité serait de définir une chaîne contenant les 16 caractères hexadécimaux. Le caractère voulu peut alors être lu en utilisant l'adresse de base de la chaîne plus un décalage correspondant au symbole à afficher. Pour aller lire un caractère de la chaîne, l'instruction de lecture en mémoire *lb* (*Load Byte*) peut être utilisée. Celle-ci fonctionne exactement de la même manière que *lw* (*Load Word*) à l'exception qu'elle ne lit qu'un octet plutôt qu'un mot de 32 bits.

```
1 | .data
2 | hex: .asciiz "0123456789ABCDEF"
```

FIGURE 10 – Définition d'une chaîne de 16 symboles hexadécimaux.

Placez le programme résultant dans un fichier nommé `spim-to-hex.s`. Testez ce programme avec plusieurs nombres naturels.

Q18) Programme convertissant un naturel en hexadécimal.

Deux solutions sont proposées ci-dessous. Celle de gauche utilise la division euclidienne par 16 et affiche les symboles hexadécimaux en commençant par le poids faible. Celle de droite utilise le même algorithme mais stocke les symboles dans une chaîne de caractères (`result`). Les symboles sont donc affichés en commençant par le poids fort.

```

1      .data
2 msg1: .ascii "Entrez un naturel:\n"
3 hex:  .ascii "0123456789ABCDEF"
4 endl: .ascii "\n"
5
6      .text
7 main:
8      la $a0, msg1
9      li $v0, 4
10     syscall
11
12     li $v0, 5
13     syscall
14
15     la $a2, hex
16
17     bne $v0, $zero, non_zero
18     li $a0, 0
19     li $v0, 1
20     syscall
21     j end
22
23 non_zero:
24     li $a0, 16
25     div $v0, $a0
26     mfhi $a0 # reste
27     mflo $a1 # quotient
28
29     add $a0, $a0, $a2
30     lb $a0, 0($a0)
31     li $v0, 11 # print_char
32     syscall
33
34     move $v0, $a1
35     bne $v0, $zero, non_zero
36
37 end:
38     la $a0, endl
39     li $v0, 4
40     syscall
41     jr $ra

```

```

1      .data
2 msg1: .ascii "Entrez un naturel:\n"
3 hex:  .ascii "0123456789ABCDEF"
4 endl: .ascii "\n"
5 result: .ascii "....."
6
7      .text
8 main:
9
10     la $a0, msg1
11     li $v0, 4
12     syscall
13
14     li $v0, 5
15     syscall
16
17     la $a2, hex
18     la $a3, result
19     addi $a3, $a3, 8
20
21     bne $v0, $zero, non_zero
22     li $a0, 0
23     li $v0, 1
24     syscall
25     j end
26
27 non_zero:
28     li $a0, 16
29     div $v0, $a0
30     mfhi $a0 # reste
31     mflo $a1 # quotient
32
33     add $a0, $a0, $a2
34     lb $a0, 0($a0)
35     addi $a3, $a3, -1
36     sb $a0, 0($a3)
37
38     move $v0, $a1
39     bne $v0, $zero, non_zero
40
41     move $a0, $a3
42     li $v0, 4 # print_string
43     syscall
44
45 end:
46     la $a0, endl
47     li $v0, 4
48     syscall
49     jr $ra

```

2.9 Appel de fonction

Dans cet exercice, il vous est demandé d'effectuer un appel à une fonction appelée `fct` qui ne prend pas d'argument et qui ne retourne pas de résultat. Cette fonction affiche un message pré-déterminé à la console et retourne à l'appelant. Votre programme doit donc définir deux fonctions : la première, appelée `main` correspond au programme principal, et la deuxième appelée `fct`. Cet exercice nécessite l'usage des instructions `jal` et `jr` ainsi que du registre `ra` pour conserver l'adresse de retour.

Attention, il y a en réalité deux appels de fonctions imbriqués dans cet exercice : appel de `main` réalisé par SPIM et appel de `fct` réalisé par `main`. Ceci implique qu'il faille sauvegarder correctement l'adresse de retour. Il vous est proposé d'utiliser un autre registre à cet effet.

Placez le programme résultant dans un fichier nommé `spim-function-call.s`. Vérifiez le comportement de votre programme en l'exécutant pas-à-pas. Observez, en particulier, l'évolution du registre *Program Counter* (`pc`) et le registre *Return Address* (`ra`). Exécutez à cet effet votre programme *pas-à-pas*.

Q19) Programme effectuant un appel de fonction.

```

1 main:
2     addiu    $sp, $sp, -4
3     sw      $ra, 0($sp)
4     jal     fct
5     lw      $ra, 0($sp)
6     addiu    $sp, $sp, 4
7     jr      $ra
8
9 fct:
10    # corps de la fonction
11    jr      $ra

```

2.10 Passage d'arguments et résultat

L'objectif de cette question est de réaliser un appel de fonction dans lequel des paramètres sont passés en arguments et un résultat est retourné. La fonction à implémenter est nommée `mul6`. Elle prend 6 entiers en arguments, en effectue la multiplication et retourne le résultat. Il vous est demandé de suivre la convention MIPS pour le passage des paramètres : les 4 premiers entiers sont passés dans les registres `a0` à `a3` et les 2 arguments excédentaires sont passés sur la pile. Le résultat de la multiplication doit être retourné dans le registre `v0`.

Placez votre programme dans le fichier `spim-mul6.s`. Ce programme doit contenir l'implémentation de la fonction `mul6` ainsi que le programme principal (fonction `main`) effectuant un appel à `mul6` avec des paramètres que vous aurez choisis. Vérifiez le résultat retourné par la fonction.

Q20) Programme effectuant un appel de fonction.

```

1 main:
2     addiu    $sp, $sp, -12
3     sw      $ra, 0($sp)
4     li      $a0, 1 # arg no 1
5     li      $a1, 2 # arg no 2
6     li      $a2, 3 # arg no 3
7     li      $a3, 4 # arg no 4
8     li      $t0, 5 # arg no 5
9     sw      $t0, 4($sp)
10    li      $t0, 6 # arg no 6
11    sw      $t0, 8($sp)
12    jal     mul6
13    lw      $ra, 0($sp)
14    addiu    $sp, $sp, 12
15
16    move     $a0, $v0
17    li      $v0, 1
18    syscall # print_int
19
20    jr      $ra
21
22 mul6:
23    mult     $a0, $a1
24    mflo     $v0
25    mult     $v0, $a2
26    mflo     $v0
27    mult     $v0, $a3
28    mflo     $v0
29    lw      $a0, 4($sp)
30    mult     $v0, $a0
31    mflo     $v0
32    lw      $a0, 8($sp)

```

```

33      mult    $v0, $a0
34      mflo    $v0
35      jr      $ra

```

2.11 Manipulation de chaînes de caractères

L'objectif de cet exercice est d'implémenter deux fonctions utiles lors de la manipulation de chaînes de caractères. Les chaînes de caractères sont de type « à zéro terminal ». Les fonctions à implémenter sont les suivantes :

- `strlen` : détermine la longueur d'une chaîne à zéro terminal. La fonction prend un seul argument : l'adresse du premier caractère de la chaîne de caractères, passé dans le registre `a0`. La fonction retourne dans le registre `v0` la longueur de la chaîne. Note : une solution à cet exercice a été discutée durant la partie théorique du cours.
- `strcmp` : compare deux chaînes de caractères selon l'ordre lexicographique. La fonction prend 2 arguments : les adresses `s1` et `s2` de deux chaînes de caractères, passées dans les registres `a0` et `a1` respectivement. La fonction retourne dans le registre `v0` la valeur 0 si les deux chaînes sont identiques ; la valeur -1 si la chaîne `s1` est inférieure à la chaîne `s2` ; la valeur +1 sinon.

L'implémentation des deux fonctions nécessite de lire une partie ou la totalité des caractères des chaînes. L'instruction `lb` (*Load Byte*) permet de charger dans un registre l'octet situé à une adresse donnée.

Placez vos fonctions ainsi qu'un programme de test dans le fichier `spim-strlen-strcmp.s`. Assurez-vous de tester vos fonctions sur plusieurs exemples permettant de vérifier les différents cas de figure possibles.

Q21) Fonction `strlen`.

```

1      .data
2      str:    .ascii "Ceci est un test"
3      endl:   .ascii "\n"
4
5      .text
6      main:
7          la   $a0, str
8
9          addiu $sp, $sp, -4
10         sw   $ra, 0($sp)
11         jal  strlen
12         lw   $ra, 0($sp)
13         addiu $sp, $sp, 4
14
15         move  $a0, $v0
16         li    $v0, 1
17         syscall
18
19         la   $a0, endl
20         li    $v0, 4
21         syscall
22
23         jr   $ra
24
25      strlen:
26         li    $v0, 0
27      strlen_loop:
28         lb    $t0, 0($a0)
29         beq   $t0, $zero, strlen_loop_end
30         addiu $v0, $v0, 1
31         addiu $a0, $a0, 1
32         b     strlen_loop
33      strlen_loop_end:
34         jr   $ra

```

Q22) Fonction `strcmp`.

```

1      .data
2      str1: .asciiz "Abracadabra"
3      str2: .asciiz "Abrac0dab"
4      str3: .asciiz "Abraca"
5      endl: .asciiz "\n"
6
7      .text
8      main:
9          addiu    $sp, $sp, -4
10         sw      $ra, 0($sp)
11
12         # "Abracadabra" > "Abrac0dab" (code caractere index 5 superieur)
13         la      $a0, str1
14         la      $a1, str2
15         jal     strcmp
16         move    $a0, $v0
17         li      $v0, 1
18         syscall
19         la      $a0, endl
20         li      $v0, 4
21         syscall
22
23         # "Abraca" < "Abracadabra" (longueur inferieure)
24         la      $a0, str3
25         la      $a1, str1
26         jal     strcmp
27         move    $a0, $v0
28         li      $v0, 1
29         syscall
30         la      $a0, endl
31         li      $v0, 4
32         syscall
33
34         # "Abracadabra" = "Abracadabra"
35         la      $a0, str1
36         la      $a1, str1
37         jal     strcmp
38         move    $a0, $v0
39         li      $v0, 1
40         syscall
41         la      $a0, endl
42         li      $v0, 4
43         syscall
44
45         lw      $ra, 0($sp)
46         addiu    $sp, $sp, 4
47         jr      $ra
48
49      strcmp:
50         lb      $t0, 0($a0)
51         lb      $t1, 0($a1)
52         blt     $t0, $t1, strcmp_smaller
53         bgt     $t0, $t1, strcmp_greater
54         beq     $t0, $zero, strcmp_end
55         addiu    $a0, $a0, 1
56         addiu    $a1, $a1, 1
57         b       strcmp
58      strcmp_end:
59         li      $v0, 0
60         jr      $ra
61      strcmp_smaller:
62         li      $v0, -1
63         jr      $ra
64      strcmp_greater:

```

```

65 |    li    $v0, 1
66 |    jr    $ra

```

2.12 Multiplication... récursive

Dans cet exercice, il s'agit d'effectuer la multiplication de deux nombres naturels de manière récursive. La fonction `mult_rec` prend en arguments deux entiers positifs ou nuls x et y placés dans les registres `a0` et `a1` respectivement. Le produit de x et y sera retourné dans le registre `v0` et affiché à la console.

Il est interdit d'utiliser les instructions de multiplication telles que `mult`. La fonction doit être implémentée récursivement, c'est-à-dire que la fonction doit s'appeler elle-même un certain nombre de fois. Remarquez à cet effet que $x \times y$ peut être obtenu comme suit

$$p = \begin{cases} y + (x - 1) \times y & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases}$$

Posez-vous également la question « y a-t-il un avantage à effectuer la récursion sur x ou sur y ? »

Placez votre fonction `mult_rec` ainsi qu'un programme de test dans le fichier `spim-mult-rec.s`. Observez l'exécution de votre programme *pas-à-pas*. Prêtez particulièrement attention aux registres *Program Counter* (`pc`), *Return Address* (`ra`) et au *Stack Pointer* (`sp`) lors des appels récursifs.

Q23) Fonction `mult_rec`.

Ci-dessous, deux solutions. La solution de gauche est la plus simple. La solution de droite échange les valeurs de x et y si $x > y$, afin de minimiser le nombre d'appels récursifs.

```

1      .data
2 msg1:  .asciiz "Entrez le naturel x: "
3 msg2:  .asciiz "Entrez le naturel y: "
4 endl:  .asciiz "\n"
5
6      .text
7 main:
8      addiu $sp, $sp, -4
9      sw $ra, 0($sp)
10
11     la $a0, msg1
12     li $v0, 4
13     syscall
14
15     li $v0, 5
16     syscall
17     move $t0, $v0
18
19     la $a0, msg2
20     li $v0, 4
21     syscall
22
23     li $v0, 5
24     syscall
25
26     move $a0, $t0
27     move $a1, $v0
28     jal mult_rec
29
30     move $a0, $v0
31     li $v0, 1
32     syscall
33
34     la $a0, endl
35     li $v0, 4
36     syscall
37
38     lw $ra, 0($sp)
39     addiu $sp, $sp, 4
40     jr $ra
41
42
43 mult_rec:
44     beq $a0, $zero, mult_rec_base
45     addi $a0, $a0, -1
46     addiu $sp, $sp, -4
47     sw $ra, 0($sp)
48     jal mult_rec
49     lw $ra, 0($sp)
50     addiu $sp, $sp, 4
51     add $v0, $v0, $a1
52     jr $ra
53
54 mult_rec_base:
55     li $v0, 0
56     jr $ra

```

```

1      .data
2 msg1:  .asciiz "Entrez le naturel x: "
3 msg2:  .asciiz "Entrez le naturel y: "
4 endl:  .asciiz "\n"
5
6      .text
7 main:
8      addiu $sp, $sp, -4
9      sw $ra, 0($sp)
10
11     la $a0, msg1
12     li $v0, 4
13     syscall
14
15     li $v0, 5
16     syscall
17     move $t0, $v0
18
19     la $a0, msg2
20     li $v0, 4
21     syscall
22
23     li $v0, 5
24     syscall
25
26     move $a0, $t0
27     move $a1, $v0
28     jal mult_rec
29
30     move $a0, $v0
31     li $v0, 1
32     syscall
33
34     la $a0, endl
35     li $v0, 4
36     syscall
37
38     lw $ra, 0($sp)
39     addiu $sp, $sp, 4
40     jr $ra
41
42
43 mult_rec:
44     # optionnel: swap x et y si x > y
45     ble $a0, $a1, mult_rec_no_swap
46     move $t0, $a0
47     move $a0, $a1
48     move $a1, $t0
49 mult_rec_no_swap:
50
51     beq $a0, $zero, mult_rec_base
52     addi $a0, $a0, -1
53     addiu $sp, $sp, -4
54     sw $ra, 0($sp)
55     jal mult_rec_no_swap
56     lw $ra, 0($sp)
57     addiu $sp, $sp, 4
58     add $v0, $v0, $a1
59     jr $ra
60
61 mult_rec_base:
62     li $v0, 0
63     jr $ra

```

2.13 Recherche dans un tableau

Cet exercice vise à rechercher un entier dans un tableau d'entiers triés. A cet effet, votre programme doit déclarer un tableau trié d'entiers en mémoire. Utilisez pour cela la directive `.word` comme illustré ci-dessous. Dans l'exemple, un tableau de 5 entiers de 32 bits est créé et ses cellules sont initialisées avec les valeurs -17, 5, 9, 123 et 1024. L'adresse de la première cellule du tableau est désignée par le *label* `tableau`.

```
1 | tableau:
2 |     .word -17, 5, 9, 123, 1024
```

Le programme doit également contenir la définition d'une fonction `find_int`. Cette fonction prend comme premier argument l'adresse de la première cellule d'un tableau, comme second argument la taille du tableau et comme troisième argument l'entier recherché. La fonction retourne la position dans le tableau de l'entier recherché si celui-ci s'y trouve. La fonction retourne -1 si l'entier ne se trouve pas dans le tableau. Vous pouvez bien entendu tirer parti du fait que le tableau est trié, de façon à en diminuer la complexité algorithmique.

Placez votre fonction `find_int`, la déclaration d'un tableau trié d'entiers ainsi qu'un programme de test dans le fichier `spim-find-int.s`. Assurez-vous de rechercher des entiers qui sont ou pas dans le tableau, en positions début, fin et milieu, de façon à tester les différents cas auxquels votre implémentation sera exposée.

Q24) Fonction `find_int`.

```
1 | .data
2 | tab: .word -17, 5, 9, 123, 1024
3 | msg: .asciiz "Entrez l'entier recherche: "
4 | endl: .asciiz "\n"
5 |
6 | .text
7 | main:
8 |     addiu $sp, $sp, -4
9 |     sw $ra, 0($sp)
10 |
11 |     la $a0, msg
12 |     li $v0, 4
13 |     syscall
14 |
15 |     li $v0, 5
16 |     syscall
17 |
18 |     la $a0, tab
19 |     li $a1, 5
20 |     move $a2, $v0
21 |     jal find_int
22 |
23 |     move $a0, $v0
24 |     li $v0, 1
25 |     syscall
26 |
27 |     la $a0, endl
28 |     li $v0, 4
29 |     syscall
30 |
31 |     lw $ra, 0($sp)
32 |     addiu $sp, $sp, 4
33 |     jr $ra
34 |
35 | find_int:
36 |     li $v0, 0
37 |     beq $a1, $zero, find_int_end
38 | find_int_loop:
39 |     lw $t0, 0($a0)
40 |     beq $a2, $t0, find_int_match
41 |     bgt $t0, $a2, find_int_end # Tire parti du tableau trie
```



```
42      addiu    $a0, $a0, 4 # Adresse prochaine cellule tableau
43      addiu    $a1, $a1, -1 # Taille restante tableau
44      addiu    $v0, $v0, 1 # Index cellule tableau
45      bgtz     $a1, find_int_loop
46 find_int_end:
47      li       $v0, -1
48      jr       $ra
49 find_int_match:
50      jr       $ra
```

2.14 Idées d'exercices supplémentaires

- Implémentez une fonction qui accepte un nombre quelconque (variable) de paramètres. Les paramètres passés à la fonction sont des entiers. La fonction calcule la moyenne de ces entiers et retourne la moyenne dans `v0`.
- Implémentez une fonction `strprint` qui prend en entrée une chaîne de caractères à zéro terminal et qui affiche la chaîne caractère par caractère en utilisant l'appel système 11 (`print_char`).
- Implémentez la fonction factorielle de façon itérative.
- Implémentez la fonction factorielle de façon récursive. Note : cet exercice a été résolu lors de la partie théorique du cours.
- Implémentez une fonction calculant de façon récursive le nombre de façons de prendre un sous-ensemble de taille k parmi un ensemble de taille n . Implémentez éventuellement une version mettant en oeuvre la mémorisation (triangle de Pascal).

A Annexes

A.1 Appels système

Le simulateur SPIM permet aux programmes qu'il exécute d'interagir avec l'utilisateur au travers d'une console. Pour cela, il est possible d'effectuer la lecture ou l'écriture de données de et vers la console : entiers, flottants et chaînes de caractères. Ces interactions sont réalisées à l'aide d'« appels systèmes » qu'un programme en langage d'assemblage MIPS peut invoquer. Les appels systèmes supportés par SPIM sont résumés à la Table 2.

Service	Numéro de l'appel système	Arguments	Résultats
print int	1	\$a0 = integer	
print float	2	\$f12 = float	
print double	3	\$f12 = double	
print string	4	\$a0 = string	
read int	5		integer (dans \$v0)
read float	6		float (dans \$f0)
read double	7		double (dans \$f0)
read string	8	\$a0 = adresse du buffer, \$a1 = longueur du buffer	
sbrk	9	\$a0 = quantité	address (dans \$v0)
exit	10		
print char	11	\$a0 = char	
read char	12		char (dans \$v0)

TABLE 2 – Liste des appels système supportés par SPIM. Le numéro de l'appel doit être placé dans le registre `v0` avant d'invoquer l'instruction `syscall`.

Pour invoquer un appel système, l'instruction spéciale `syscall` est utilisée. Cette instruction n'a pas d'opérande. Ses arguments sont passés au travers de registres. Le numéro de l'appel système est toujours passé dans le registre `v0`. Chaque appel système transfère ses autres arguments ainsi que ses résultats dans des registres qui varient d'un appel système à l'autre et qui sont documentés à la Table 2.

A.2 Types d'instructions MIPS

Le jeu d'instructions MIPS repose sur un encodage de taille fixe, 32-bits. En revanche, trois formats différents d'instructions sont définis : R, I et J. Ces formats sont illustrés à la Figure 11. Il est possible de déterminer le format d'une instruction sur base de son *opcode*, toujours situé dans les 6 bits de poids fort du code.

- Les instructions de **type R** permettent de spécifier 3 registres ainsi qu'un code de fonction (*funct*) et une quantité de décalage (*shamt*). Les instructions `add` et `sll` par exemple utilisent ce format. Toutes ces instructions ont un *opcode* qui vaut `b000000` et sont donc distinguées sur base du champ *funct*.
- Les instructions de **type I** permettent de spécifier 2 registres ainsi qu'une valeur immédiate de 16-bits (*imm*). Les instructions `addi` et `bgt` par exemple utilisent ce format.
- Les instructions de **type J** permettent de spécifier une valeur immédiate de 26-bits (*target*). Seules les instructions `j` et `jal` utilisent ce format.

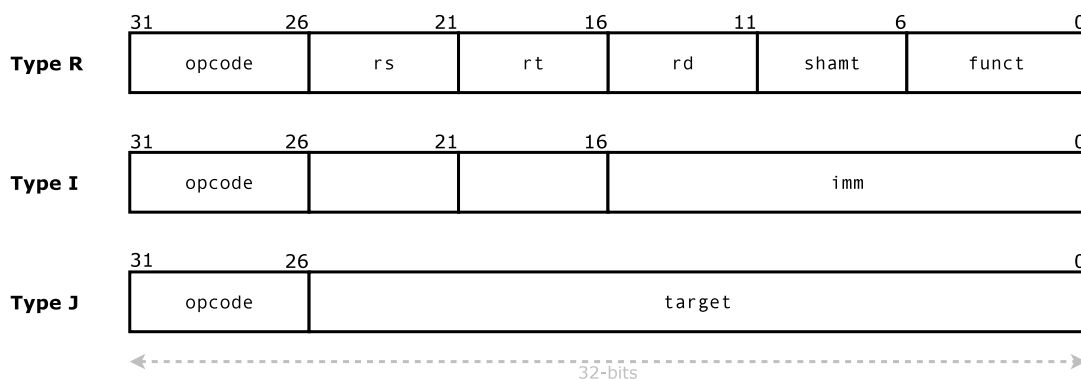


FIGURE 11 – Formats des instructions MIPS.

A.3 Directives de l'assembleur

Le langage d'assemblage MIPS supporte plusieurs directives permettant par exemple de définir des chaînes de caractères en mémoire ou des tableaux d'entiers. La Table 3 donne une brève description des directives qui pourraient être utiles dans ce TP.

Directive	Description
<code>.data [addr]</code>	Indique que les déclarations suivantes (données et instructions) sont mises dans un segment <u>mémoire de données</u> . Il est possible d'imposer l'adresse mémoire à laquelle le segment commence, en utilisant l'argument optionnel <i>addr</i> .
<code>.text [addr]</code>	Indique que les déclarations suivantes (données et instructions) sont mises dans un segment <u>mémoire d'instructions</u> . Il est possible d'imposer l'adresse mémoire à laquelle le segment commence, en utilisant l'argument optionnel <i>addr</i> .
<code>.ascii str</code>	Stocke le littéral chaîne de caractères <i>str</i> en mémoire selon le codage ASCII et sans zéro terminal.
<code>.asciiz str</code>	Stocke le littéral chaîne de caractères <i>str</i> en mémoire selon le codage ASCII et avec zéro terminal.
<code>.byte b_1, \dots, b_n</code>	Stocke les n littéraux entiers b_1, \dots, b_n sous forme de mots de 8 bits à des emplacements consécutifs en mémoire.
<code>.half s_1, \dots, s_n</code>	Stocke les n littéraux entiers s_1, \dots, s_n sous forme de mots de 16 bits à des emplacements consécutifs en mémoire.
<code>.word w_1, \dots, w_n</code>	Stocke les n littéraux entiers w_1, \dots, w_n sous forme de mots de 32 bits à des emplacements consécutifs en mémoire.
<code>.space n</code>	Réserve une zone de n octets.
<code>.float f_1, \dots, f_n</code>	Stocke les n littéraux flottants f_1, \dots, f_n sous forme de mots de 32 bits au format IEEE 754 simple précision à des emplacements consécutifs en mémoire.
<code>.double d_1, \dots, d_n</code>	Stocke les n littéraux flottants d_1, \dots, d_n sous forme de mots de 64 bits au format IEEE 754 double précision à des emplacements consécutifs en mémoire.
<code>.align n</code>	Force l'alignement des données suivantes à une adresse qui est un multiple de 2^n .
<code>.extern sym size</code>	Déclare le symbole <i>sym</i> comme étant global et de taille <i>size</i> . Cela peut être utilisé pour déclarer un symbole défini ailleurs, p.ex. dans un autre fichier.
<code>.globl sym</code>	Déclare le symbole <i>sym</i> comme étant global. Cela permet de référencer ce symbole à partir d'autres fichiers. Dans le simulateur SPIM, cela permet aussi que ce symbole soit listé via le menu <i>Simulator / Display Symbols</i> .
<code>.kdata [addr]</code>	Similaire à <code>.data</code> mais sera placé dans une zone mémoire réservée au système.
<code>.ktext [addr]</code>	Similaire à <code>.text</code> mais sera placé dans une zone mémoire réservée au système.

TABLE 3 – Liste de directives utiles de l'assembleur MIPS.

Remerciements

Merci à V. DHEUR, M. CHARLIER, A. BUYS et D. HAUWEELE pour leurs remarques sur ce document ou une version antérieure.