ES102/PC9 : énoncé et corrigé

Les objectifs pédagogiques de cette PC sont de découvrir :

- l'exécution de quelques fragments de code sur un microprocesseur de type MIPS et leur expression, voire compilation, dans le langage *assembleur* du même nom ;
- les solutions et artifices élaborés pour répondre efficacement à certains besoins logiciels (structuration des données, contrôle de flux, démarche procédurale, etc.) ;
- certains compromis matériels/logiciels pour obtenir la meilleure efficacité de calcul.

1) Multiplication logicielle

1a) Décaler de 1 bit vers la gauche un nombre entier stocké dans un registre 32 bits revient à le multiplier par 2, sous certaines conditions... Lesquelles ? On les supposera vérifiées par la suite.

Il ne faut pas que l'entier soit trop grand sinon une multiplication par 2 le fait « déborder » de sa plage. Ainsi le MSB d'un entier non signé doit être nul. Quant aux entiers signés, sur 32 bits, ils prennent normalement leur valeur sur l'intervalle [-2^{31}, $+2^{31}-1$]. Il faut le réduire de moitié, autour de 0. En fait, il faut se restreindre à l'intervalle [-2^{30}, $+2^{30}-1$]. Cela signifie que les 2 bits de poids (le plus) fort doivent être identiques : 00– si positif, 11– si négatif. Des décalages à plus grande portée vers la gauche contraindront un plus grand nombre de bits de poids forts.

La séquence d'instructions MIPS ci-dessous est présentée instructions à gauche et explications à droite. Chaque instruction vient avec une liste d'opérandes : le registre destination, le premier registre source et enfin le deuxième registre source ou un paramètre. Trois sortes d'instructions apparaissent ci-dessous : *mv* (move) pour copie, *add* pour addition, et *sll* (shift left logical) pour décalage vers la gauche, avec 0 entrant à droite. Les registres utilisés ont pour nom *s1* et *t1*.

```
mv $t1,$s1          t1 ← $s1
sll $t1,$t1,2       t1 ← $t1 << 2 (décalage de 2 bits vers la gauche)
add $t1,$t1,$s1     t1 ← $t1+$s1
sll $t1,$t1,1       t1 ← $t1 << 1
add $t1,$t1,$s1     t1 ← $t1+$s1
sll $t1,$t1,1       t1 ← $t1 << 1
add $t1,$t1,$s1     t1 ← $t1+$s1
```

1b) Que réalise le programme ci-dessus ?

Le registre *s1* contient manifestement une variable entière, que l'on nomme *i*. Le calcul réalisé peut être formulé comme suit : [(i×4+i)×2+i]×2+i. Il s'agit du produit de *i* par la constante 23. On a $(23)_{10}=(10111)_2$ et le facteur 4 ci-dessus correspond à l'écart de 2 bits entre les deux premiers 1 du côté des poids forts. Cette technique de calcul rappelle celle utilisée sur la calculette du CM7, améliorée ici grâce au décalage à portée variable effectué par `sll`. Le registre « accumulateur » R1 y jouait le même rôle que *t1* ici. Cela évite d'avoir à décaler *s1* lui-même.

1c) Le simplifier par recodage de Booth. L'instruction de soustraction est nommée/notée `sub`.

Le recodage de Booth transforme les chaînes de plusieurs 1 consécutifs précédées d'un 0 en 10-01 (où 1 désigne le « chiffre » -1), afin d'augmenter le nombre de 0. Pour 23, cela donne : $(10111)_2 \rightarrow (1100\underline{1})_2$. En d'autres termes : 23=16+8-1. D'où le code suivant, plus court :

```
mv $t1,$s1
sll $t1,$t1,1
add $t1,$t1,$s1
sll $t1,$t1,3
sub $t1,$t1,$s1
```

1d) En réalité, la première instruction, « `mv $t1,$s1` » est réalisée par « `addi $t1,$s1,0` », où *addi* est l'instruction d'addition immédiate. Pourquoi ?

C'est que *mv* n'existe pas : c'est une *pseudo-instruction*. Cela économise du matériel, dans le chemin de données comme dans l'unité de commande (et peut-être même de l'énergie).

2) Boucle while et tableau de données

Les registres constituent une ressource rare : une donnée stockée en mémoire n'est amenée dans le banc de registres que lorsqu'elle va être utilisée par l'ALU pour un calcul. Ensuite, une autre prendra sa place. De même, un résultat de calcul ne devant pas être réutilisé prochainement sera renvoyé en mémoire pour libérer de la place dans le banc de registres. Ces nécessités sont évidentes lorsqu'on fait des calculs sur les données d'un tableau, dont la taille est généralement bien plus grande que la vingtaine de registres disponibles en pratique pour faire des calculs.

Les instructions MIPS permettant les échanges de données 32 bits entre mémoire et banc de registres sont respectivement notées :

- lw (*load word*) pour un chargement, impliquant une lecture de la mémoire
- sw (*store word*) pour un stockage, impliquant une écriture en mémoire

Lecture ou écriture nécessitent la présentation d'une adresse à la mémoire. La donnée en jeu est souvent un élément d'un tableau dont l'adresse de base aura été préalablement chargée dans un registre. L'adresse de l'élément considéré est alors calculée par l'ALU, à partir de son indice. Soit `data[]` un tableau d'entiers déclaré dans un programme en langage C.

2a) Comment est compilée l'instruction « `i=data[2];` » en assembleur MIPS ?

On suppose l'adresse de base du tableau `data` déjà disponible dans un registre, par exemple *s1*. Un autre registre, par exemple *t1*, va accueillir la variable `i`. La donnée `data[2]` est disponible en mémoire de données à l'adresse *data+8*, car 2 entiers occupent 8 octets. L'instruction C ci-dessus peut alors être réalisée par une unique micro-instruction MIPS réalisant à la fois le calcul de *data+8*, la présentation de cette adresse à la mémoire de données, la récupération de la donnée `data[2]` et son stockage dans le registre *t1* par :

```
lw  $t1,$s1+8
```

Pour calculer *data+8*, la valeur 8 est passée en valeur immédiate dans les 16 derniers bits des 32 représentant la microinstruction, puis ajoutée à $s1 par l'ALU.

Pour information, la véritable syntaxe MIPS pour cette microinstruction est : `lw $t1,8($s1)`. Mais son étrangeté fait qu'on ne la respecte pas ici, dans un souci de simplification/lisibilité.

On continue ci-dessous à accéder au tableau `data[]`, mais les indices des éléments manipulés sont désormais variables, exigeant un calcul à la volée de leur adresse (pour lequel le mécanisme de valeur immédiate n'est d'aucune aide). Un compilateur C a transformé la ligne suivante :

```
while (data[i] == k) { i = i+j; }
```

en la séquence suivante d'instructions en langage assembleur MIPS :

```
Loop : sll $t1,$s1,2      t1 ← $s1 << 2 (décalage de 2 bits vers la gauche)
       add $t1,$t1,$s4    t1 ← $t1 + $s4
       lw  $t0,$t1+0      t0 ← contenu de la mémoire à l'adresse figurant dans t1
       bne $t0,$s3,Exit   si $t0≠$s3, sauter jusqu'à Exit (branch if not equal)
       add $s1,$s1,$s2    s1 ← $s1 + $s2
       j Loop             sauter (retourner) jusqu'à l'instruction étiquetée Loop
Exit : ------             instruction non précisée
```

Les instructions ci-dessus, chacune codées sur 32 bits, sont stockées l'une derrière l'autre en mémoire d'instructions. « Loop » et « Exit » sont des étiquettes attachées chacune à une instruction, vers lesquelles d'autres instructions peuvent pointer. C'est une commodité de notation pour désigner l'adresse en mémoire de l'instruction étiquetée.

2b) Ci-dessus, que contiennent les registres *s1*, *s2*, *s3*, *s4* ? Qui sert de pointeur sur `data[i]` ?

Dans le code C, seul data[i] exige un accès à la mémoire. Cette donnée réside 4*i octets plus loin que data[0]. Or, seule la troisième instruction réalise une lecture de la mémoire, à l'adresse désignée par *t1*. C'est donc *t1* qui joue le rôle de **pointeur** sur data[i].

Or sa valeur est 4*$s1+$s4. L'explication la plus plausible est que $s1=i et $s4=data, adresse du début du tableau data[]. La lecture charge data[i] dans *t0*. Puisque $t0 est ensuite comparé avec $s3, c'est que *s3* contient *k*. Enfin, *s2* contient manifestement *j*, qui vient incrémenter *i* lors de l'avant-dernière instruction.

2c) Comment appelle-t-on la dernière instruction du segment de code ci-dessus ? Quelles particularités de fonctionnement du processeur exige-t-elle ? Quelle est sa « portée » ?

L'instruction *j* (jump) est un saut.

Habituellement, le processeur se prépare à accéder à l'instruction suivante en mémoire de programme en incrémentant de 4 octets (une instruction) son CP (**c**ompteur de **p**rogramme). Mais avec le saut j Loop, la prochaine instruction à exécuter est celle portant l'étiquette symbolique Loop. C'est donc l'adresse où elle se trouve en mémoire qui doit être chargée dans le CP. Cette adresse A est fournie (en grande partie) par l'instruction j Loop elle-même et c'est le compilateur qui l'y a mise.

Précisément, elle en occupe les 26 derniers bits (le code opératoire du *jump* occupe les 6 premiers) : ils constituent en fait les bits A_{27} à A_2, sachant que A_1=A_0=0 puisque l'adresse d'une instruction est un multiple de 4 octets. Les 4 derniers bits , A_{31} à A_{28}, sont quant à eux pris sur l'instruction courante : ils demeurent donc inchangés, imposant au saut de rester dans un même bloc d'instruction de 256 Mo (1/4 Go), soit 64 M·instructions.

Pour sauter plus loin, il faut recourir à l'instruction *jr* (jump register), principalement utilisée pour les retours de sous programmes (cf. exercice suivant).

2d) Le segment de code ci-dessus comporte également une instruction *bne*. De quoi s'agit-il ? Quelles particularités de fonctionnement du processeur exige-t-elle ? Quelle est sa « portée » ? Comment l'adresse correspondant à l'étiquette Exit est-elle calculée ?

bne (**b**ranch if **n**ot **e**qual) est une instruction de branchement**.** Lorsque « bne $t0,$s3,Exit » est exécutée, la prochaine instruction dépend du résultat de la comparaison entre $t0 et $s3, comparaison réalisée par l'ALU (par soustraction et détection de résultat nul). En cas d'échec du test (ici $t0=$s3) la prochaine instruction exécutée sera la suivante dans la séquence. En cas de succès (ici $t0≠$s3), ce sera celle étiquetée par Exit. Bref, le résultat du test, fourni sous forme binaire par l'ALU, détermine si le CP est incrémenté de 4 ou chargé avec l'adresse correspondant à l'étiquette Exit.

Parmi les 32 bits qui codent l'instruction, 6 sont pris pour désigner *bne* et 2x5 pour désigner *t0* et *s3*. Il en reste seulement 16 pour exprimer Exit. Cela ne permet d'adresser qu'un petit espace mémoire de 2^{16} = 64 k·instructions (256 ko), ridicule comparé aux capacités mémoire actuelles. Est-ce viable ? Oui car, presque toujours, un branchement pointe sur une instruction proche de lui en mémoire. Ici, l'instruction étiquetée par Exit se trouve 3 instructions (12 octets) plus loin. Fréquemment, il faut revenir un peu en arrière. Il y a donc tout intérêt à utiliser un repère relatif à l'instruction courante plutôt qu'un repère absolu sur la mémoire d'instructions. Au lieu de charger l'adresse correspondant à Exit dans le CP, il suffit d'ajouter le nombre 3, en instructions (soit 12 octets), à la valeur actuelle du CP. Ceci nécessite un additionneur et c'est pour cela que le CP vu en cours n'est pas équipé d'un simple compteur. Finalement, les 16 bits permettent de se brancher jusqu'à $\pm 2^{15}$ = 32 k·instructions en arrière ou en avant de l'instruction de branchement, c'est qui est très confortable. Dans les cas très rares où il faut aller plus loin, on peut se brancher sur un *jump*…

2e) Un compilateur plus performant produit le nouveau code ci-dessous. Expliquer et apprécier.

```
          j Begin
Loop :    add $s1,$s1,$s2
Begin :   sll $t1,$s1,2
          add $t1,$t1,$s4
          lw  $t0,$t1+0
          beq $t0,$s3,Loop
```

Il y a toujours 6 instructions, mais la longueur de la boucle est passée de 6 à 5, soit un gain de 17% en vitesse. En fait, il n'y a plus de *jump* dans la boucle. Comment est-ce possible ?

La boucle *while* ci-dessus met en jeu un *test*, l'égalité entre data[i] et k, et une *action* : i←i+j. Le premier code assembleur était organisé comme suit :

```
Boucle :    préparation test
            si test négatif, aller à Sortie
            action
            aller à Boucle
Sortie :
```

Tandis que le nouveau est organisé ainsi :

```
            aller à Début
Boucle :    action
Début :     préparation test
            si test positif, aller à Boucle
```

On a fait « tourner » le corps de la boucle pour amener le branchement en dernière position. Dans ces conditions, plus besoin d'aller à la Sortie, on y est déjà. On peut donc inverser le test et s'en servir pour retourner au début de la boucle en cas de test réussi. Plus besoin de saut dans ces conditions, si ce n'est à l'extérieur, lors de la première exécution de la boucle, pour la commencer au Début, qui se retrouve quelque part au milieu du code désormais.

3) Fonctions, passage de paramètres, appels imbriqués

Parmi les 32 registres disponibles sur processeur MIPS I, il y en a 6 servant spécifiquement en tant qu'**a**rguments et **v**aleurs de fonctions : *a0*, *a1*, *a2*, *a3*, *v0* et *v1*. Le code suivant, qui utilise *a0*, *a1* et *v0*, est celui d'une fonction min(), qui retourne le minimum de ses deux arguments.

```
Min2 : sub $t0,$a0,$a1
       bgtz $t0,Arg1    bgtz : branch if greater than zero
       mv $v0,$a0
       j Fin
Arg1 : mv $v0,$a1
 Fin : jr $ra
```

3a) Hormis la dernière instruction, quelle structure de contrôle reconnaît-on ?

Une structure « if then else », qui copie effectivement l'argument le plus petit dans v0.

L'appel de la fonction min() apparaîtra en un ou plusieurs autres endroits du programme, chargeant d'abord les registres *a0* et *a1* avec les deux arguments souhaités, puis exécutant un *jump and link*, comme suit :

```
mv $a0,...
mv $a1,...
jal Min2
```

3b) Quels sont les rôles des instructions `jal` et `jr` ci-dessus ? Que signifie la notation *`ra`* ?

`jal Min2` provoque le saut vers le code implantant la fonction min(). Une fois min() exécutée, il faut revenir juste derrière l'instruction qui l'a appelée (l'instruction appelante). C'est l'instruction `jr $ra` qui assure ce retour. Donc *ra* est un registre qui doit contenir l'adresse de l'instruction qui suit `jal Min2`. C'est `jal Min2` elle-même qui charge *ra*. C'est tout le sens du « link » de *jump and link*. La notation *ra* signifie simplement **r**eturn **a**ddress. Le registre *ra* pointe donc vers l'instruction où il faudra retourner après achèvement du sous-programme.

3c) Le code MIPS ci-dessous semble être celui d'une fonction min3() à 3 arguments. Expliquer en quoi. Toutefois, ce code présente un grave dysfonctionnement. Lequel ?

```
Min3 : jal Min2
       mv $a0,$v0
       mv $a1,$a2
       jal Min2
       jr $ra
```

Lorsque min3() est appelée – par une instruction `jal Min3` – elle reçoit ses 3 arguments dans *a0*, *a1* et *a2*. Un appel de min() fournit alors min($a0,$a1) dans *v0*. Il suffit ensuite d'appeler de nouveau min(), mais avec $v0 et $a2 comme arguments, pour récupérer le minimum des 3. Essayons …

Tiens, mon ordinateur est planté : l'instruction `jr $ra` de min3() est en train de boucler sur elle-même ! Pourquoi ? Le registre *ra* aurait dû contenir l'adresse de l'instruction se trouvant après `jal Min3`, l'appel de min3(). Mais, lors de l'exécution de min3(), le contenu de *ra* a été écrasé successivement par les deux `jal Min2`, afin de revenir correctement de min(). Du coup, lorsque s'exécute l'instruction `jr $ra` de min3(), c'est sa propre adresse qui se trouve dans *ra* : horreur ! Quand on imbrique des appels de fonctions, il faut sauvegarder quelque part l'adresse de retour courante avant de faire un nouvel appel. Comme le nombre d'imbrications possibles n'est pas limité, il est exclus de mobiliser des registres pour cela. On utilise plutôt une pile logicielle, comme présenté ci-dessous.

Dans le cadre d'un appel imbriqué de fonction, tout registre utilisé à la fois par la fonction appelante et la fonction appelée doit être sauvegardé en mémoire juste avant ou juste après l'appel, pour être restauré ensuite. Une pile LIFO (Last In, First Out) est une structure de

données efficace pour réaliser de telles sauvegardes et restaurations, puisque tout appel postérieur à un autre doit effectuer son retour avant lui. En pratique, on utilise la queue de la mémoire de données pour réaliser une pile LIFO logicielle : on la remplit en partant de l'adresse la plus élevée. Le sommet de la pile (correspondant à l'adresse la plus petite) est pointé par la valeur du registre spécifique nommé *sp* (stack pointer).

3d) Doter les codes des fonctions min() et min3() des bonnes sauvegardes dans la pile.

min() exploite *t0*, qui est peut-être déjà utilisé par la fonction appelante, d'où :

```
Min2 : addi $sp,$sp,-4      on ajoute un niveau sur la pile (vers le bas)
       sw  $t0,$sp+0        on y sauvegarde $t0
       sub $t0,$a0,$a1
       bgtz $t0,Arg1
       mv $v0,$a0
       j Fin
Arg1 : mv $v0,$a1
 Fin : lw $t0,$sp+0         on restaure t0 à partir de la pile
       addi $sp,$sp,4       on réduit la pile (vers le haut)
       jr $ra
```

min3() est à la fois appelée et appelante, d'où :

```
Min3 : addi $sp,$sp,-4      on ajoute un niveau sur la pile
       sw  $ra,$sp+0        on y sauvegarde le contenu de ra
       jal Min2
       mv $a0,$v0           transfert de la valeur de retour dans a0
       mv $a1,$a2           a1 ← $a2
       jal Min2
       lw  $ra,$sp+0        on restaure ra à partir de la pile
       addi $sp,$sp,4       on réduit la pile
       jr $ra               v0 a déjà la bonne valeur
```

Ci-dessus, on a sauvegardé *t0* après l'appel et *ra* avant. Indispensable pour *ra*, c'est un choix pour *t0* : la sauvegarde aurait pu aussi être faite avant l'appel, dans la fonction appelante.

Pour information, il existe en MIPS une *convention* dite *d'appel* selon laquelle la sauvegarde des registres *t0* à *t7* est de la responsabilité de l'appelant, et celle des registres *s0* à *s7* de la responsabilité de l'appelé. Ainsi, la sauvegarde de *t0* ci-dessus dans l'implantation de min() était inutile, car censée être déjà réalisé par l'appelant si nécessaire.

3e) Comment passe-t-on des paramètres à une fonction qui en compte plus que 4 ?

Puisque les registres *a0*, *a1*, *a2* et *a3* ne suffisent pas, on peut passer les paramètres excédentaires en les empilant sur la pile.

3f) Comment passe-t-on des paramètres à une fonction qui peut en compter un nombre variable, telle que `printf()` ?

A condition de passer en paramètre le *nombre* de paramètres, on peut empiler la liste variable de paramètres sur la pile et la fonction appelée saura combien en dépiler.