Architecture des ordinateurs et systèmes d'exploitation

Corrigé du TP 6: DLXview

Arnaud Giersch Benoît Meister Nicolas Passat

Analyse d'un programme DLX

Récupérer le programme *example.s*. Celui-ci doit être lancé après avoir initialisé le registre r1 à la valeur 0 : soit en tapant préalablement dans le terminal de contrôle **put r1 0** soit en chargeant un fichier d'initialisation *example.i*. Il faut, de même, initialiser r4 à zéro.

Questions :

– Sans l'exécuter, dites ce que fait ce programme.

Correction : *Ce programme contient un tableau d'entiers placé à partir de l'adresse 0 (soit* tab *ce tableau). Pendant l'exécution, il lit* $\text{tab}[0]$ *entiers à partir de* $\text{tab}[1]$*, les somme et enregistre le résultat dans* $\text{tab}[\text{tab}[0]+1]$ *(à la fin du tableau).*

– Chargez ensuite le programme dans dlxview. Dans le terminal de contrôle, affichez le contenu de la mémoire pour vérifier que celle-ci contient bien les données ainsi que les instructions.

Correction : *On a défini 4 entiers à partir de l'adresse 0 :*

```
(dlxview) get 0 4d
0x0:    3
0x4:    9
0x8:    -14
0xc:    11
```

Le code est placé à l'adresse 256, il y a 9 instructions :

```
(dlxview) get 256 9i
_main:  lw r2,0x0(r1)
loop:   addi r1,r1,0x4
loop+0x4:       lw r3,0x0(r1)
loop+0x8:       add r4,r4,r3
loop+0xc:       subi r2,r2,0x1
loop+0x10:      bnez r2,loop
loop+0x14:      addi r1,r1,0x4
loop+0x18:      sw 0x0(r1),r4
loop+0x1c:      trap 0x0
```

– Exécutez ce programme avec dlxview. On s'aperçoit qu'il ne produit pas le résultat escompté. Cherchez la cause de cet échec en réalisant une exécution pas à pas et en consultant régulièrement le contenu des registres. Que signifient les « `stall` » qui apparaissent dans le pipeline ? Modifiez ensuite le programme pour rendre son exécution correcte.

Correction : *On remarque que l'instruction* `addi r1,r1,0x4` *à l'adresse* `loop+0x14` *est exécutée avant de faire le* `bnez r2,loop`*. Il y a donc un* delay slot *pour les branchements (comme en assembleur* SPARC*).*
Les « `stall` » signifient que l'exécution d'une instruction est retardée jusqu'à ce que la précédente soit complètement exécutée. On peut observer que cela se produit quand une instruction suit un chargement en mémoire et qu'elle utilise le même registre, où quand un branchement conditionnel utilise le résultat de l'instruction le précédent.
On corrige le programme en ajoutant un `nop` *après l'instruction* `bnez` *:*

```
(dlxview) get 256 9i
_main:  lw r2,0x0(r1)
loop:   addi r1,r1,0x4
loop+0x4:       lw r3,0x0(r1)
loop+0x8:       add r4,r4,r3
loop+0xc:       subi r2,r2,0x1
loop+0x10:      bnez r2,loop
loop+0x14:      nop
loop+0x18:      addi r1,r1,0x4
loop+0x1c:      sw 0x0(r1),r4
loop+0x20:      trap 0x0
```

Exercice de programmation

L'objectif est de réaliser un programme qui trouve le plus petit et le plus grand entier d'un tableau. Dans le programme, ce tableau sera donné dans la zone `data` qui contiendra le nombre d'éléments du tableau puis les éléments. Les deux résultats doivent être inscrits dans la zone mémoire à la suite des éléments du tableau.

Correction : *Une solution s'exécutant en 93 cycles est :*

```
;       Données placées en mémoire à l'adresse 0
; .word introduit des entiers signés, pour les autres types, on
; utilise .float, .double, .byte

.data 0
.word 6, 1, 9, 6, -14, 11, 2
.space 4
.space 4

;       Instructions placées en mémoire à l'adresse 256 (par défaut)

.text
_main:  xor r1, r1, r1          ; r1: pointeur <- 0
        lw r2, 0(r1)            ; r2: compteur <- *r1
        addi r1, r1, 4          ; r1 ++
        lw r3, 0(r1)            ; r3: min <- *r1
        lw r4, 0(r1)            ; r4: max <- *r1

loop:
        subi r2, r2, 1          ; while (-- r2 != 0) {
        beqz r2, end_loop
        nop

        addi r1, r1, 4          ;       r1 ++
        lw r5, 0(r1)            ;       r5 <- *r1
        slt r6, r5, r3          ;       if (r5 < min) {
        beqz r6, notmin
        nop
        ori r3, r5, 0           ;               min <- r5
        j loop                  ;               continue
        nop
notmin:
        sgt r6, r5, r4          ;       } else if (! (r5 > max)) {
        beqz r6, loop           ;               continue
        nop                     ;       } else {
        ori r4, r5, 0           ;               max <- r5
        j loop                  ;               continue
        nop                     ;       }
                                ; }
end_loop:
        addi r1, r1, 4          ; r1 ++
        sw 0(r1), r3            ; *r1 <- min
        addi r1, r1, 4          ; r1 ++
        sw 0(r1), r4            ; *r1 <- max
        trap 0

; trap 0 provoque la sortie du programme, sans cela, le programme
;       exécuterait à l'infini des instructions nop
```

Dans un deuxième temps, vous ordonnerez au mieux vos instructions pour minimiser le nombre de `stall`.

Correction : *En éliminant tous les* `stall` *et en essayant de minimiser le nombre de* `nop`*, on obtient une solution s'exécutant en 65 cycles :*

```
;       Données placées en mémoire à l'adresse 0
; .word introduit des entiers signés, pour les autres types, on utilise
; .float, .double, .byte

.data 0
.word 6, 1, 9, 6, -14, 11, 2
.space 4
.space 4

;       Instructions placées en mémoire à l'adresse 256 (par défaut)

.text
_main:  xor r1, r1, r1          ; r1: pointeur <- 0
        lw r2, 0(r1)            ; r2: compteur <- *r1
        addi r1, r1, 4          ; r1 ++
        subi r2, r2, 1          ; r2 --
        lw r3, 0(r1)            ; r3: min <- *r1
        lw r4, 0(r1)            ; r4: max <- *r1

loop:
        beqz r2, end_loop       ; while (r2 != 0)
        addi r1, r1, 4          ;       r1 ++ (delay slot)
        lw r5, 0(r1)            ;       r5 <- *r1
        subi r2, r2, 1          ;       r2 --
        slt r6, r5, r3          ;       r6 <- (r5 < min)
        sgt r7, r5, r4          ;       r7 <- (r5 > max)
        beqz r6, notmin         ;       if (r6) {
        nop
        j loop                  ;               continue
        ori r3, r5, 0           ;               min <- r5 (delay slot)
notmin:                         ;       } else if (! r7) {
        beqz r7, loop           ;               continue
        nop                     ;       } else {
        j loop                  ;               continue
        ori r4, r5, 0           ;               max <- val (delay slot)
                                ;       }
                                ; }
end_loop:                       ; }
        sw 0(r1), r3            ; *r1 <- min
        addi r1, r1, 4          ; r1 ++
        sw 0(r1), r4            ; *r1 <- max
        trap 0

; trap 0 provoque la sortie du programme, sans cela, le programme
; exécuterait à l'infini des instructions nop
```