ARCHITECTURE DES ORDINATEURS

2ème Année Informatique

Langage d'assemblage du MIPS R3000

Centre universitaire Mila

1

Plan

- Introduction
- Programmation en langage d'assemblage
- Format d'un programme assembleur MIPS
- Déclaration de données
- Les instructions de l'assembleur MIPS

2

Introduction

- Ecrire un programme directement en langage machine (0 et 1) est une tâche très difficile.
- Déchiffrer la signification d'instructions codées numériquement est fatigant pour les humains.
- Au lieu d'utiliser le langage machine, on peut programmer avec le langage d'assemblage.
- Le langage d'assemblage est la représentation symbolique du code binaire des instructions.
- Il est proche du langage machine et lisible par les humains.

3

Programmation en langage d'assemblage

Programme en langage de haut niveau (en C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Programme en langage d'assemblage (en Mips)

```
swap:
        mul  $2, $5,4
        add  $2, $4,$2
        lw   $15, 0($2)
        lw   $16, 4($2)
        sw   $16, 0($2)
        sw   $15, 4($2)
        jr   $31
```

Assembler

Programme en langage machine pour le processeur Mips

```
00000000101000010000000000011000
00000000000011000000110000010000
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011110000000000000000001000
```

4

Programmation en langage d'assemblage

- Par exemple, l'instruction qui additionne le contenu des registres $t3 et $t4 et de stocker le résultat dans $t6 est codée par :

  00000001011011000111000000100000

- Et pour ne pas utiliser beaucoup de place, on peut les réduire en hexadécimal:

  016c7020

- Elle est représentée en langage assembleur comme suit :

  add $t6, $t3, $t4

- Ici, la signification de l'instruction est beaucoup plus claire qu'en code machine.

5

Programmation en langage d'assemblage

Les noms de fichiers :

- Les noms des fichiers contenant un programme source en langage d'assemblage doivent être suffixé par « .asm » ou « .s ».
  Exemple : premierprog.asm

- Pour écrire un programme en langage assembleur de Mips R3000 il faut suivre les règles suivantes:

6

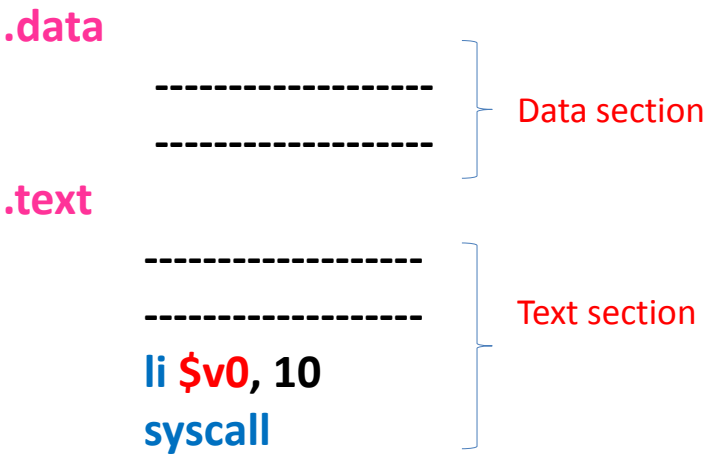
Format d'un programme assembleur:

Un programme assembleur est composé de deux parties:

- Data section : contient la déclaration de données.
- Text section: contient le code du programme.

7

Format d'un programme assembleur:



8

Les commentaires:

- Les commentaires permettent de donner plus d'explication sur le code
- Ils commencent par un # ou un ; et s'achèvent à la fin de la ligne courante.

Exemple :

```
# Ceci est un commentaire
; Ceci est un commentaire
```

9

Déclaration de données:

- Les données (constantes et variables) doivent être déclarées dans « .Data » section.
- Les données doivent commencer par une lettre suivie des lettres, chiffres ou caractères spéciaux.
- Le format général de la déclaration d'une donnée est:

« nom de variable »: .« type de donnée » « valeur initiale »

Exemple:

```
        .data
c       .byte   'a'     ; octet
n1      .half    26     ; 2 octets
n2      .word    353    ; 4 octets
tab     .space   40
```

10

Déclaration de données:

- Les données en assembleur sont de différents types:

Declaration	
`.byte`	8-bit variable(s)
`.half`	16-bit variable(s)
`.word`	32-bit variable(s)
`.ascii`	ASCII string
`.asciiz`	NULL terminated ASCII string
`.float`	32 bit IEEE floating point number
`.double`	64 bit IEEE floating point number
`.space <n>`	<n> bytes of uninitialized memory

11

Déclaration de données:

- Déclaration des entiers :

–Une valeur entière décimale est notée 253.

–Une valeur entière hexadécimale est notée 0xFA (préfixée par zéro suivi de x).

–En hexadécimal, les lettres de A à F peuvent être écrites en majuscule ou en minuscule.

–Les entiers sont déclarés par :

- **.word, .half, .byte**
- Exemple :

```
var1:  .word   100000
var2:  .byte    20
```

12

Déclaration de données:

➢ Les chaînes de caractères :

Les chaînes de caractères sont déclarées par:

.ascii, **.asciiz**

Remarque:

.asciiz termine la chaîne de caractères par NULL (0). Le but est de faire savoir au compilateur la fin de la chaîne.

Exemple : la déclaration suivante défini une chaîne de caractères « message » de type asciiz et qui a comme contenu « hello world ».

- « \n »: est un retour à la ligne

```
message:   .asciiz    "Hello World\n"
```

13

Déclaration de données:

➢ Déclaration des nombres réels :

•Les nombres réels sont déclarés par :

.float, **.double**

Exemple :

•Les déclarations suivantes sont utilisées pour définir la variable « pi » sur 32 bits par le type .float et l'initialiser à 3,14159.

•Et la variable « Var » qui prend le type .double et sera enregistrée sur 64 bits. Elle est initialisée à 7,28318.

```
Pi:    .float   3,14159
Var:  .double   7,28318
```

14

Les instructions de l'assembleur Mips:

- Pour pouvoir faire des programmes exécutables sur la machine MIPS R3000, on dispose d'un certain nombre d'instructions qui forment le langage de la machine.
- Le jeu d'instructions est "orienté registres". Cela signifie que les instructions utilisent les registres pour le transfert et le rangement des données.
- Le processeur possède plusieurs instructions qui se répartissent en 4 classes :

• Instructions de lecture/écriture mémoire (Load and Store)

•Instructions arithmétiques/logiques entre registres (calcul)

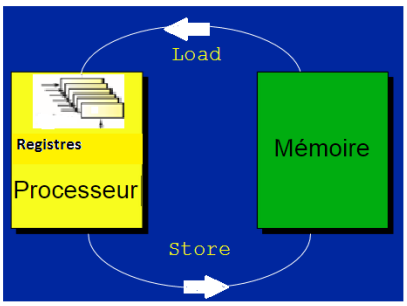
• Instructions de branchement (Saut)

• Appels système (Syscall)

15

Les instructions de lecture/écriture

- Ces instructions transfèrent les données entre la mémoire et les registres.
- L'opération de chargement (Load) permet de lire une donnée de la mémoire et la charger dans un registre.
- L'opération de sauvegarde (Store) permet d'écrire le contenu d'un registre dans la mémoire.



16

Les instructions de lecture/écriture

1. Les instruction de chargement: (Load)

Lw - Load Word -

Lw Rdest, adresse

load word	lw
load byte	lb
load byte unsigned	lbu
load half	lh
load half unsigned	lhu

load immediate	li
load address	la

- Charge le contenu de l'adresse mémoire dans le registre Rdest.

Exemple:

Lw $t0, var1

17

Les instructions de lecture/écriture

1. Les instruction de chargement: (Load)

- Li - Load immediate -

Li Rdest, Imm

Charge la valeur immédiate « imm » (constante) dans le registre Rdest.

Exemple:

Li $t0, 23

Après l'exécution $t0=23

- La - Load address-

La Rdest, adresse

Charge l'adresse de la donnée dans le registre Rdest.

18

Les instructions de lecture/écriture

➢ 2. Les instruction de sauvegarde: (Store)

•Sw - Store Word -

Sw Rsrc, adresse

Sauvegarde le contenu du registre source Rsrc dans la mémoire.

- Exemple:

  Li $t0, 23

  sw $t0, var

Les instructions de lecture/écriture

➢ Copie de registres

Move Rdest, Rsrc

•Copie le contenu du registre Rsrc dans le registre Rdest.

Exemple:

•Cette instruction donne au registre $t0 la valeur 42 ensuite copie le contenu de $t0 dans $t1.

$t1 =42

```
li    $t0, 42
move  $t1, $t0
```

Remarque:

Il n'y a pas d'instruction move permettant de faire copier une donnée d'une cellule mémoire vers une autre.

Les appels système syscall

- Pour exécuter certaines fonctions système, principalement les entrées/sorties (lire ou écrire un nombre, ou un caractère), il faut utiliser des appels système.
- Pour exécuter un appel système, il faut chercher:
  - –L'argument dans le registre $a0 ou $a1.
  - –Le numéro de l'appel système est contenu dans le registre $v0
  - –syscall

$v0	commande	argument	résultat
1	print_int	$a0 = entier	
4	print_string	$a0 = adresse de chaîne	
5	read_int		$v0 = entier lu
8	read_string	$a0 = adresse de chaîne, $a1 = longueur max	
10	exit		

21

Les appels système syscall

- Ecrire un entier: (Print_int)

Il faut mettre l'entier à écrire dans le registre $a0 et exécuter l'appel système numéro 1.

- Exemple:

```
li  $a0, 10    # load argument $a0=10
li  $v0, 1     # call code to print integer
syscall        # print $a0
```

22

Les appels système
Syscall

- Lire un entier : (Read_int)

Consiste à exécuter l'appel système numéro 5 et récupérer le résultat dans le registre $v0.

- Exemple:

```
li        $v0, 5          # system call code for Read Integer
syscall                   # reads the value into $v0
```

23

Les appels système
Syscall

- Lire une chaîne de caractères : (Read_string)

Il faut:

• Adresse du début de la chaîne et la passer dans $a0

• La longueur maximum et la mettre dans $a1

• Exécuter l'appel système numéro 8.

• Le résultat sera mis dans l'espace pointé.

• Exemple:

```
li        $v0, 8          # system call code for Read a String
la        $a0, buffer     # load address of  input buffer into $a0
li        $a1, 60         # Length of buffer
syscall
```

24

Les appels système
Syscall

- Ecrire une chaîne de caractères: (Print_string)

Une chaîne de caractères étant identifiée par une adresse de début de la chaîne.

Il faut passer cette adresse dans $a0 et exécuter l'appel système numéro 4 pour l'afficher.

Exemple:

```
li      $v0, 4          # system call code for Print a String
la      $a0, buffer     # Load address of output buffer into $a0
syscall
```

- Quitter le programme:

L'appel système numéro 10 effectue la sortie du programme

```
li      $v0, 10         # terminate program run and
syscall                 # return control to  system
```

25

Les instructions arithmétiques et logiques

➢ Les instructions arithmétiques et logiques permettent de faire les 4 opérations de calcul (addition, multiplication, soustraction, division) ainsi que les opérations logiques (Or, And, or, xor..) et les opérations de décalage…

1.add

add Rdest, Rsrc1, Rsrc2

•Description

- Les contenus des registres **Rsrc1** et **Rsrc2** sont ajoutés pour former un résultat qui est placé dans le registre **Rdest**.

- On peut aussi faire une addition avec les constante au lieu du registre Rsrc2

addi Rdest, Rsrc1, imm

26

Les instructions arithmétiques et logiques

2.sub

sub Rdest, Rsrc1, Rsrc2

Description

Le contenu du registre **Rsrc2** est soustrait du contenu du registre **Rsrc1** pour former un résultat qui est placé dans le registre **Rdest**.

3.mul

mul Rdest, Rsrc1, Rsrc2

Description

Les contenus des registres **Rsrc1** et **Rsrc2** sont multipliés pour former un résultat qui est placé dans le registre **Rdest**.

27

Les instructions arithmétiques et logiques

4.mult

mult Rsrc1, Rsrc2

Description

mult permet de multiplier les contenus des registres **Rsrc1** et **Rsrc2**. Le résultat est placé dans les registres spéciaux : **hi** et **lo**.

Les 32 bits de poids fort du résultat sont placés dans le registre **hi**, et les 32 bits de poids faible dans **lo**.

5.div

div Rsrc1, Rsrc2

Description

Le contenu du registre **Rsrc1** est divisé par le contenu du registre **Rsrc2**.Le résultat de la division est placé dans le registre **lo**, et le reste dans le registre **hi**.

28

Les instructions arithmétiques et logiques

6. mfhi - Move from $hi -

mfhi Rdest

Description

Le contenu du registre spécialisé **hi** — qui est mis à jour par l'opération de multiplication ou de division — est recopié dans le registre général **Rdest**.

7. mflo - Move from $ho -

mflo Rdest

Description

Le contenu du registre spécialisé **ho**— qui est mis à jour par l'opération de multiplication ou de division — est recopié dans le registre général **Rdest**.

29

Les instructions arithmétiques et logiques

8. and - Et logique -

and Rdest, Rsrc1, Rsrc2

Description

Un **et** bit-à-bit est effectué entre les contenus des registres **Rsrc1** et **Rsrc2**. Le résultat est placé dans le registre **Rdest**.

9. or - ou logique -

or Rdest, Rsrc1, Rsrc2

Description

Un **ou** bit-à-bit est effectué entre les contenus des registres **Rsrc1** et **Rsrc2**. Le résultat est placé dans le registre **Rdest**.

30

Les instructions arithmétiques et logiques

8. and - Et logique -

and Rdest, Rsrc1, Rsrc2

Description

Un **et** bit-à-bit est effectué entre les contenus des registres **Rsrc1** et **Rsrc2**. Le résultat est placé dans le registre **Rdest**.

9. or - ou logique -

or Rdest, Rsrc1, Rsrc2

Description

Un **ou** bit-à-bit est effectué entre les contenus des registres **Rsrc1** et **Rsrc2**. Le résultat est placé dans le registre **Rdest**.

31

Les instructions arithmétiques et logiques

10. neg - negatif -

neg Rdest, Rsrc1

Description

Donne le contraire de **Rsrc1**.Le résultat est placé dans le registre **Rdest**.

Rdest= -(Rsrc1)

11. sll - Shift Left Logical -

sll Rdest, Rsrc1, shamt

Description

Décalage à gauche immédiat

- Le registre Rsrc1 est décalé à gauche de la valeur immédiate shamt.
- Des zéros étant introduits dans les bits de poids faibles.
- Le résultat est placé dans le registre Rdest.

32

Les opérations de Saut (Jump) et de Branchement (Branch)

• Les structures de contrôles comme le IF..THEN..ELSE, WHILE et FOR n'existent pas en langage d'assemblage.

• Pour les programmer, il faut utiliser les instructions de sauts et de branchements.

• Ces instructions sont de type conditionnel ou inconditionnel.

33

Les instructions de branchement

beq $s0, $s1, label	if $s0==$s1 goto label
bne $s0, $s1, label	if $s0!=$s1 goto label
bge $s0, $s1, label	if $s0>=$s1 goto label
bgt $s0, $s1, label	if $s0> $s1 goto label
ble $s0, $s1, label	if $s0<=$s1 goto label
blt $s0, $s1, label	if $s0<$s1 goto label
bgez $s0, label	if $s0>=0 goto label
bgtz $s0, label	if $s0> 0 goto label
blez $s0, label	if $s0<=0 goto label
bltz $s0, label	if $s0< 0 goto label
bnez $s0, label	if $s0!=0 goto label
beqz $s0, label	if $s0==0 goto label

34

Les instructions de saut

• J - Jump - / • b - Branch

j Label / b Label

Description

Le programme saute inconditionnellement à Label.

Exemple:

```
blockA: … instructions
        j exit
blockB: … instructions
exit:   …
```

35

Les instructions de saut

• Jr - Jump register –

Jr $rs

Description

Le programme saute à l'adresse contenue dans le registre $rs.

• Jal - Jump and link –

Jal Label

Description

Utilisé dans les appels des procédures et fonctions. L'adresse de l'instruction suivant le jal est sauvée dans le registre ($ra).

36

Les tableaux et les piles

• Les principaux types de données manipulés par le MIPS R3000 sont: les entiers, les réels et les caractères.

• Les structures de données comme les tableaux et les piles sont utilisées en MIPS pour stocker des données les unes à la suite des autres.

37

Les tableaux

- Un tableau est une zone mémoire dans laquelle les données sont rangées les unes à la suite des autres;
- Il est repéré par l'adresse du premier élément;
- On accède aux différentes cases par déplacement par rapport à la première;
- Un tableau est déclaré comme suit:

```
.data
T: .word  10,-1,5,0,-9
```

- Cette déclaration permet l'allocation en mémoire d'un tableau de 5 éléments (mots de 32 bits) qui sont: {10,-1,5,0,-9}

38

Les tableaux

- Le programme suivant permet de faire la multiplication des éléments d'un tableau par 10:

```
.data
tab: .word 5,2,-10,4,0,6,7,3,9,8
.text
li $t0,0
li $t1,10
li $t2,0
saut: beq $t0,10,fin
lw $t3,tab($t2)
mul $t3,$t3,$t1
sw $t3,tab($t2)
li $v0,1
move $a0,$t3
syscall
add $t0,$t0,1
add $t2,$t2,4
j saut
fin: li $v0,10
syscall
```

39

Les tableaux

- Pour récupérer l'adresse du premier élément dans le tableau, on utilise l'instruction suivante:

  la $a2,tab

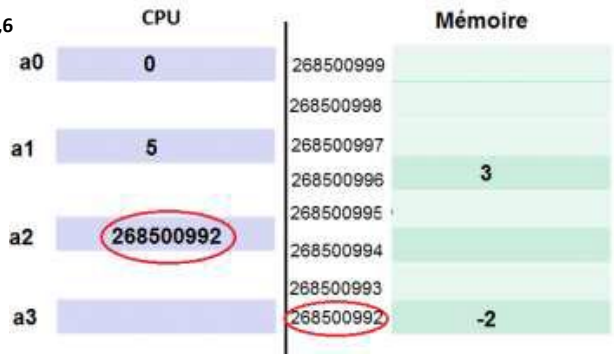
- Après l'exécution de cette instruction le registre $a2 contient l'adresse du premier élément du tableau

40

Les tableaux

- Le contenu des registres et de la mémoire après l'exécution de la partie suivante est:

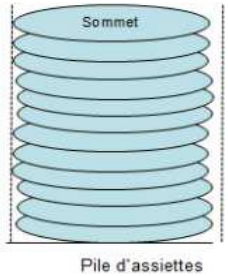
```
.data
tab: .word  -2,3,4,0,6
.text
li $a0,0
li $a1,5
la $a2,tab
```

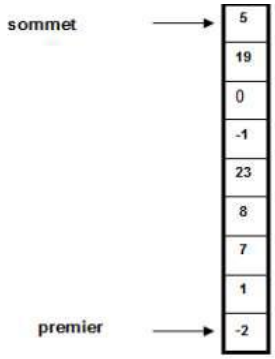


41

Les piles

- La pile est une structure de données, qui permet de stocker les données dans l'ordre LIFO (Last In First Out) - Dernier Entré Premier Sorti).
- La récupération des données sera faite dans l'ordre inverse de leur insertion.





42

Les piles

- On peut comparer l'organisation de la pile à une pile d'assiettes: on peut parfaitement rajouter une assiette au sommet de la pile, ou enlever celle qui est au sommet, mais on ne peut pas toucher aux autres.
- La pile est utilisée dans la programmation pour stocker les informations durant les appels des procédures et des fonctions.
- Pour ajouter un élément à la pile on utilise l'opération Push (Empiler).
- Pour retirer un élément de la pile on utilise l'opération Pop (Dépiler).
- Le sommet de la pile est pointé par le registre $sp.
- Tout les éléments empilés et dépilés de la pile ont 32 bits.

43

Les piles

- Push (empiler): Empiler ou ajouter un élément à la pile signifie soustraire 4 du registre $sp et stocker le résultat dans cette adresse.

• Exemple :

l'instruction 'Push $t0' effectue les opérations suivantes:

```
subu $sp, $sp, 4     # Mettre à jour $sp
sw $t0, ($sp)        # ranger $t0 au sommet de la pile
```

44

Les piles

- Pop (dépiler):Retirer un élément de la pile signifie ajouter 4 au registre $sp.

Dépiler un élément signifie qu'il est copié à un autre endroit (un registre).

Exemple :

L'instruction 'Pop $t0' effectue les opérations suivantes:

```
lw $t0, ($sp)      # copie le sommet dans $t0
addu $sp, $sp,4    # Mettre à jour $sp
```

45

Les piles

- Exemple: Ce programme permet d'empiler les éléments d'un tableau dans la pile puis dépiler et afficher.

```
.data
tab: .word 5,2,-10,4,0,6,7,3,9,8
.text
la $t0,tab
li $t1,0
li $t2,10
push:
lw $t3,($t0)
move $a0,$t3
li $v0,1
syscall
subu $sp,$sp,4
sw $t3,($sp)
add $t1,$t1,1
add $t0,$t0,4
blt $t1,$t2,push
pop:beq $t4,10,end
lw $t0, ($sp)
addu $sp, $sp, 4
move $a0,$t0
li $v0,1
syscall
add $t4,$t4,1
j pop
end: li $v0,10
syscall
```

46

Les procédures et les fonctions

- Les fonctions et les procédures permettent de décomposer un programme complexe en une série de sous-programmes plus simples.
- Le but est d'éviter de réécrire le même code plusieurs fois.
- Une fonction est une suite ordonnée d'instructions qui retourne une valeur .
- Par contre, une procédure est un sous programme qui ne retourne pas de valeur.

47

Les procédures et les fonctions

- Pour déclarer une procédure/fonction en Mips on utilise:

```
.globl  procedureName
procedureName:
# code goes here
.end  procedureName
```

Exemple:
.text
main:
.....
Jal calcul
…
.end main
calcul:
#code de la fonction
Jr $ra
.end calcul

48

Les procédures et les fonctions

La fonction appelante utilise les registres suivant:

- Les registres $a0-$a3 pour passer les arguments
- Les registres $t0-$t9
- Les registres $v0, $v1 pour retourner les résultats de la fonction.

Les arguments sont passés comme suit:

- Le premier argument est passé dans le registre $a0.
- Le deuxième argument est passé dans le registre $a1.
- Le troisième argument est passé dans $a2.
- Les autres arguments sont passés par la pile.

49

Les procédures et les fonctions

Pour appeler une fonction on utilise l'instruction suivante:

Jal <fonct>

- Cette instruction de saut permet de faire un saut avec lien
- C'est-à-dire que le programme appelant fait un saut pour

exécuter la fonction ensuite un retour pour continuer l'exécution du programme.

- Les principales instruction de saut avec lien sont:

Jal et Jr

50

Les procédures et les fonctions

- Les deux instructions utilisent le registre $ra.
- Ce registre est utilisé pour sauvegarder l'adresse de retour.
- L'instruction Jal copie $pc dans $ra et saute à la fonction appelée.
- Quand la fonction appelée se termine on utilise : Jr $ra pour aller à l'instruction suivante dans le programme appelant.

51

Les procédures et les fonctions

```
.data
        x: .word 3
        y: .word 5
        answer: .word 0

.text                          Arguments pour passer x et y

main:
        lw $a0, x
        lw $a1, y
        jal power
        sw $v0, answer
        li $v0, 10
        syscall # terminer le programme
.end main

power:
        li $v0, 1
        li $t0, 0
        powLoop:
        mul $v0,$v0, $a0
        add $t0, $t0, 1
        blt $t0, $a1, powLoop
        jr $ra
.end power
```

52

Les procédures et les fonctions

- L'exemple précédent permet de calculer la puissance du nombre X par Y. (X^Y)
- X et Y sont passées par valeur.
- La valeur du premier argument est passé dans le registre $a0.
- La valeur du deuxième argument est passée dans le registre $a1.
- Le résultat est mis dans la variable answer.
- La fonction power sauvegarde le résultat dans $v0.
- Le programme principale copie le contenue de $v0 dans la variable answer.

53

Les procédures et les fonctions

Le passage des paramètres pour les fonctions et les procédures se fait par valeur ou par adresse.

- Le passage par valeur: les variables transmises ne changent pas de valeur lors du retour au programme principal.
- Le passage par adresse:

• Le passage de la variable se fait par l'adresse et n'ont pas par la valeur.

• Après l'exécution de la fonction appelée les variables changent de valeur.

• La passage par adresse est utilisé pour faire passer des tableaux par exemple.

54