

Architecture des ordinateurs

Cours 5

Responsable de l'UE : Karine Heydemann

Contact : prenom.nom@lip6.fr

Informations

- Le partiel aura lieu le lundi 8 novembre de 8h30 à 10h (dans 3 amphis, répartition donnée sur le site de la Licence)
- Le TME solo aura lieu le lundi 13 décembre de 8h30 à 10h30
Convocation de 50% des étudiants à 8h30, 50% à 9h30.

Plan du cours 5

- 1 Introduction
- 2 Instructions de saut MIPS
- 3 Calculer en fonction d'une condition
- 4 Itérer un calcul n fois
- 5 Encodage des instructions de saut et détermination de l'adresse de saut
- 6 Réalisation des structures de contrôle des langages de haut niveau à l'aide de sauts

Programme avec flot d'exécution variable

On sait désormais écrire des programmes avec des instructions utilisant l'ALU et avec des variables globales. Mais...

- Comment exprimer des calculs différents en fonction d'une condition ?
Par exemple, comment coder le calcul de la valeur absolue d'une variable stockée en mémoire ?
- Comment itérer n fois un même calcul ?
Par exemple, comment parcourir n éléments implantés consécutivement en mémoire (tableau) pour leur ajouter 1 ?

⇒ Utilisation d'instructions de saut qui permettent de casser l'exécution séquentielle par défaut du code.

Deux type d'instructions de sauts

Sauts inconditionnels ou "jump"

- Ils ont toujours lieu : le PC est modifié avec une valeur donnée par un opérande de l'instruction
 - `j etiquette_inst, jr $31`
- L'exécution continue (affectation de PC) à l'adresse spécifiée par l'étiquette ou contenue dans un registre.

Sauts conditionnels ou branchements

- Ils sont réalisés ou **pris**, si et seulement si, une condition spécifiée dans le codop de l'instruction est vérifiée. Cette condition peut être :
 - égalité ou différence de deux registres :
`beq $0, $4, etiquette_inst, bne $0, $4, etiquette_inst,`
 - comparaison d'un registre à 0 ($< 0, \leq 0, > 0, \geq 0$) :
`bgez $4, etiquette_inst`
- L'exécution continue (affectation de PC) à l'adresse spécifiée par l'étiquette si la condition est vérifiée et en séquence sinon (incrément normal de PC).

Les instructions de sauts inconditionnnels

Syntaxe des sauts inconditionnels directs

Saut inconditionnel de la forme `j etiq` avec `etiq` une étiquette positionnée dans le code avant ou après le saut

```
.data
.text
ori $4, $0, 10
ori $3, $0, 8
j suite
add $4, $4, $3
suite:

# instructions quelconques ici

ori $2, $0, 10
syscall
```

```
.data
.text
ori $4, $0, 10
ori $3, $0, 8
suite:
add $4, $4, $1
j suite

# instructions quelconques ici

ori $2, $0, 10
syscall
```

Les instructions de sauts inconditionnnels : exemple

```
.data
.text
ori $4, $0, 10
ori $3, $0, 8
j suite
add $4, $4, $3
suite:

# instructions quelconques ici

ori $2, $0, 10
syscall
```

```
.data
.text
ori $4, $0, 10
ori $3, $0, 8
suite:
add $4, $4, $1
j suite

# instructions quelconques ici

ori $2, $0, 10
syscall
```

Quel est l'effet de ces instructions sur ces 2 codes ?

- À gauche : c'est un saut en avant, l'instruction `add $4, $4, $3` qui suit le saut n'est pas exécutée
 - À droite : c'est un saut en arrière, permet de répéter l'exécution de l'instruction entre l'étiquette cible du saut et le saut... mais engendre une exécution sans fin
- ⇒ Solution : utilisation d'un saut conditionnel (à la place du saut / entre l'étiquette et le saut conditionnel)

Saut conditionnels (1)

Condition d'égalité du contenu de deux registres

```
beq $10, $8, label # beq = branch if equal
```

- Si $\$10 = \8 alors branchement à l'étiquette label, sinon exécution séquentielle

Condition d'inégalité du contenu de deux registres

```
bne $10, $8, label # bne = branch if not equal
```

- Si $\$10 \neq \8 alors branchement à l'étiquette label, sinon exécution séquentielle

Sauts conditionnels (2)

4 instructions de saut conditionnel avec une comparaison à 0 du contenu d'un registre

`bgez $8, label # branch if greater or equal than zero`

- Si $\$8 \geq 0$ alors branchement à l'étiquette label, sinon exécution séquentielle

`bgtz $8, label # branch if greater than zero`

- Si $\$8 > 0$ alors branchement à l'étiquette label, sinon exécution séquentielle

`blez $8, label # branch if less or equal than zero`

- Si $\$8 \leq 0$ alors branchement à l'étiquette label, sinon exécution séquentielle

`bltz $8, label # branch if less than zero`

- Si $\$8 < 0$ alors branchement à l'étiquette label, sinon exécution séquentielle

Calculer en fonction d'une condition

Calculer la valeur absolue d'un entier rangé en mémoire et l'afficher

- Lecture d'un entier en mémoire : déjà vu
- Calcul de valeur absolue : ???
- Affichage du résultat (un entier) : déjà vu

Calculer en fonction d'une condition

Calculer la valeur absolue d'un entier rangé en mémoire et l'afficher

- Lecture d'un entier en mémoire : déjà vu
- Calcul de valeur absolue : ???
- Affichage du résultat (un entier) : déjà vu

```
.data
n:    .word -1          # allocation d'un entier initialisé à une valeur (-1)

.text
    lui $3, 0x1001      # chargement de l'adresse n dans $3
    lw  $4, 0($3)       # lecture de la valeur (en mémoire) dans $4

    # ici il faut calculer la valeur absolue

    ori $2, $0, 1       # affichage de l'entier contenu dans $4
    syscall

    ori $2, $0, 10      # fin de programme
    syscall
```

Calculer en fonction d'une condition

Calculer la valeur absolue d'un entier contenu dans un registre

- Si l'entier est positif : sa valeur absolue est lui-même, sinon il faut calculer l'opposé (0 - valeur)
- L'instruction `bgez OpReg, etiquette` spécifie que l'exécution continue à `etiquette` si `OpReg` positif ou nul \Rightarrow c'est le cas où il n'y a rien à faire, on saute à l'affichage

```
.data
n: .word -1      # allocation d'un entier initialisé à une valeur (-1)

.text
lui $3, 0x1001 # chargement de l'adresse n dans $3
lw $4, 0($3)   # lecture de la valeur (en mémoire) dans $4

# test de positivité et saut si positif
bgez $4, affiche
sub $4, $0, $4 # calcul de l'opposé

affiche:
ori $2, $0, 1  # affichage de l'entier contenu dans $4
syscall

ori $2, $0, 10 # fin de programme
syscall
```

Calculer en fonction d'une condition

- Comment réaliser un calcul en fonction d'une condition qui compare deux valeurs ?

Exemple : calculer l'opposé d'un entier n rangé en mémoire s'il est supérieur à un autre entier p rangé en mémoire ou à une constante ?

- Les instructions de branchement permettent uniquement de tester l'égalité ou la différence entre deux registres ou de comparer à 0...

Sauts conditionnels (3)

Comment réaliser un saut conditionnel avec comparaison de deux valeurs ?

- Comparaison des deux valeurs avec une **instruction de comparaison**
- Le résultat de la comparaison vaut 1 (vrai) ou 0 (faux)
- Branchement conditionnel avec le résultat

Comparaison registre-registre

- `slt $10, $8, $9 # set if less than`
- `$10 = 1 si $8 < $9, $10 = 0 sinon`

Saut conditionnels (4)

Comparaison registre-immédiat

- `slti $10, $8, 10` # set if less than immediate
- `$10 = 1` si `$8 < 10`, `$10 = 0` sinon
- Immédiat étendu de manière signée
- `sltiu` : comparaison non signée, mais immédiat étendu de manière signée (ex : -1)

Branchement conditionnel avec le résultat d'une comparaison

- `beq $10, $0, label`
 - Saut à `label` si la comparaison est fausse
- `bne $10, $0, label`
 - Saut à `label` si la comparaison est vraie

Autre cas de comparaison de deux valeurs

Cas et suites d'instructions correspondantes

- Saut à l'adresse label si $\$10 < \8
 slt \$9, \$10, \$8 # \$9 vaut 1 si $\$10 < \8
 bne \$9, \$0, label
- Saut à l'adresse label si $\$10 > \8
 slt \$9, \$8, \$10 # \$9 vaut 1 si $\$8 < \10
 bne \$9, \$0, label
- Saut à l'adresse label si $\$10 \leq \8
 slt \$9, \$8, \$10 # \$9 vaut 1 si $\$8 < \10
 beq \$9, \$0, label
- Saut à l'adresse label si $\$10 \geq \8
 slt \$9, \$10, \$8 # \$9 vaut 1 si $\$10 < \8
 beq \$9, \$0, label

Calculer en fonction d'une condition

- Calculer l'opposé d'un entier n rangé en mémoire s'il est supérieur à une constante \Rightarrow tester si n est supérieur à la constante ($n > \text{constante}$?)
 - Il faut utiliser une instruction `slt OpReg1, OpReg2, OpReg3` qui effectue la comparaison $\text{OpReg2} < \text{OpReg3}$
 \Rightarrow `OpReg2` doit contenir la constante et `OpReg3` la valeur de n .
 - Si `OpReg1` vaut 1, calcul de l'opposé ; s'il vaut 0, il faut aller à la suite : \Rightarrow utilisation d'un saut comparant à 0 : `beq OpReg1, $0, suite`

```
.data
n: .word -1          # allocation d'un entier initialisé à une valeur (-1)
.text
lui $3, 0x1001      # chargement de l'adresse n dans $3
lw $4, 0($3)        # lecture de la valeur (en mémoire) dans $4

# test de la condition
ori $5, $0, 0xA     # constante dans $5
slt $6, $5, $4       # $6 vaut 1 si $5 < $4 (soit 0xA < n), 0 sinon
beq $6, $0, suite    # saut à la suite si condition fausse

# calcul de l'opposé
sub $4, $0, $4       # calcul de l'opposé sinon

suite:
...
```

Calculer en fonction d'une condition

- Correspond à la structure de contrôle de haut niveau `if-then`

```
.text
# code avant

# test de la condition
ori $5, $0, 0xA
slt $6, $5, $4      # $6 vaut 1 si 10 < n, 0 sinon
beq $6, $0, suite  # saut si condition fausse

# calcul de l'opposé
sub $4, $0, $4      # calcul de l'opposé sinon

suite:
# code apres
```

- À quoi ressemblerait un code avec des calculs de type `if-then-else` ?
- Ajouter 10 à \$4 dans le cas où `n` est inférieur ou égal à 10 dans le code ci-dessus ?

Calculer en fonction d'une condition

- Ajouter 10 à \$4 dans le cas où n est inférieur ou égal à 10
- Ajouter une suite d'instructions correspondant au cas "else" + adapter le saut pour aller exécuter cette séquence d'instructions

```
.text
# code avant

# test de la condition
ori $5, $0, 0xA
slt $6, $5, $4      # $6 vaut 1 si 10 < n
bne $6, $0, else # saut si condition fausse

# cas then : calcul de l'opposé
sub $4, $0, $4      # calcul de l'opposé sinon

else: # cas else : ajouter 10
add $4, $4, 10

suite:
# code apres
```

- Est-ce correct ?

Calculer en fonction d'une condition

Erreur classique à éviter

- On ne doit pas exécuter les 2 cas ("then" puis "else") : il faut un saut inconditionnel à la fin du cas "then"!

```
.text
# code avant

# test de la condition
ori $5, $0, 0xA
slt $6, $5, $4      # $6 vaut 1 si 10 < n
beq $6, $0, else # saut si condition fausse

# cas then : calcul de l'opposé
sub $4, $0, $4      # calcul de l'opposé sinon
j suite

else: # cas else : ajouter 10
add $4, $4, 10

suite:
# code apres
```

On peut inverser la position des séquences "then" et "else" en inversant la condition dans l'instruction de saut conditionnel

Calculer en fonction d'une condition

Inversion du placement des séquences `then - else`

On peut inverser la position des séquences en inversant la condition dans l'instruction de saut conditionnel

```
.text
# code avant

# test de la condition
ori $5, $0, 0xA
slt $6, $5, $4      # si 10 < n $6 vaut 1
bne $6, $0, then # saut si condition vraie

# cas else : ajouter 10
add $4, $4, 10
j suite

then: cas then : calcul de l'opposé
    sub $4, $0, $4      # calcul de l'opposé sinon

suite:
# code apres
```

Itérer un calcul n fois

- Mettre n dans un registre
- Tester si n est supérieur à 0 : soit finir l'exécution, soit exécuter le calcul
⇒ L'instruction `beq OpReg, $0, etiquette` spécifie que l'exécution continue à `etiquette` si `OpReg` est égal à 0
- Il faut organiser le code et positionner des étiquettes pour réaliser ce calcul conditionnel (1 exécution)

```
.data
n:    .word 0xA    # allocation d'un entier initialisé à 10

.text
    lui $3, 0x1001 # chargement de l'adresse de n dans $3
    lw  $4, 0($3)  # mettre n dans un registre
                        # ici n est dans $4

    # tester si n vaut 0, effectuer le calcul ou non
    beq $4, $0, non_calcul

    # instructions du calcul

non_calcul: # instructions après les itérations du calcul
    ori $2, $0, 10 # fin de programme
    syscall
```

Itérer un calcul n fois

Quand la valeur de n est supérieure à 0 : on exécute le calcul 1 fois puis il faut décrémenter la valeur de n et recommencer le test de positivité de n

- ⇒ positionner une étiquette avant l'évaluation de la condition (debut)
- ⇒ revenir à l'évaluation de la condition : saut inconditionnel (j debut)

```
.data
n: .word 0x0A    # allocation d'un entier initialisé à 10

.text
lui $3, 0x1001   # chargement de l'adresse de n dans $3
lw $4, 0($3)     # mettre n dans un registre
                  # ici n est dans $4

debut:
# tester si n vaut 0, effectuer le calcul ou non
beq $4, $0, non_calcul

#ici instructions du calcul

#ici décrémenter $4 et retourner à l'évaluation de la condition

    addi $4, $4, -1
    j  debut

non_calcul: # ici instructions après les itérations du calcul
    ori $2, $0, 10 # fin de programme
    syscall
```

Iterer n fois un calcul : exemple complet

- Soit n une variable globale entière strictement positive (par exemple 10)
- Effectuer le calcul de la somme des entiers de 1 à n et afficher le resultat

```
.data
n:    .word 0x0A    # allocation d'un entier initialisé à 10

.text
lui $3, 0x1001    # chargement de l'adresse de n dans $3
lw $4, 0($3)      # mettre n dans un registre
                    # ici n est dans $4
xor $5, $5, $5    # registre contenant la somme
debut:
    # tester si n vaut 0, effectuer le calcul ou non
    beq $4, $0, non_calcul

    # instructions du calcul : ajouter n à la somme
    add $5, $5, $4

    # décrémenter $4 et retourner à l'évaluation de la condition
    addi $4, $4, -1
    j debut

non_calcul: # instructions après les itérations du calcul
    ori $4, $5, 0 # affichage de la somme
    ori $2, $0, 1
    syscall
    ori $2, $0, 10 # fin de programme
    syscall
```


Détermination de l'adresse de saut : deux formes d'adressage

Adressage absolu

- Concerne uniquement les instructions de format J
- Le label (sur 26 bits) est une partie de l'adresse à laquelle il faut se brancher
- Ex : `j label` donne $PC := PC[31:28] \mid I * 4$
- I = adresse de la cible du saut privée des bits 31:28 et avec 1:0 nuls

000010	Immédiat sur 26 bits
--------	----------------------

Détermination de l'adresse de saut : deux formes d'adressage

Adressage relatif

- Concerne les instructions B_{xx}
- Le déplacement est relatif à la valeur actuelle de PC
- Ex : `bne $9, $8, label` donne $PC := PC + 4 + (I * 4)$
- I = nombre d'instructions entre celle suivant le saut et la destination du saut

000101	R _x	R _y	Immédiat sur 16 bits
--------	----------------	----------------	----------------------

Détermination de l'adresse de saut

Conséquences

- Nécessité de connaître les instructions cibles des sauts et leurs adresses d'implantation pour déterminer la valeur du champ Immédiat dans les formats \mathcal{J} et \mathcal{I} pour les sauts
- \Rightarrow L'assemblage nécessite deux passes

Passé 1 : implantation de toutes les instructions

- Assignation des adresses des instructions
- Codage des instructions (sur 32 bits) en laissant les champs \mathcal{I} des instructions de saut à une valeur qui n'est pas définitive (nulle)

Passé 2 : détermination des champs laissés vides

- Les adressages absolus sont résolus : il suffit de placer la partie de l'adresse de l'instruction vers laquelle on saute dans le champ \mathcal{I} de l'instruction de saut
- Les adressages relatifs sont calculés en fonction de l'adresse de l'instruction de saut et de l'adresse de l'instruction cible du saut : nombre d'instructions entre les deux

Détermination de l'adresse de saut

Première passe

Langage d'assemblage	Adresse d'implantation	Binaire
.data		
n: .word 0x0A		
.text		
lui \$3, 0x1001	0x00400000	0x3c031001
lw \$4, 0(\$3)	0x00400004	0x8c640000
xor \$5, \$5, \$5	0x00400008	0x00a52826
debut:		
beq \$4, \$0, non_calcul	0x0040000c	0x1080 XXXX
add \$5, \$5, \$4	0x00400010	0x00a42820
addi \$4, \$4, -1	0x00400014	0x2084ffff
j debut	0x00400018	0b000010 BB...BB
non_calcul:		
ori \$4, \$5, 0	0x0040001c	0x34a40000
ori \$2, \$0, 1	0x00400020	0x34020001
syscall	0x00400024	0x0000000c
ori \$2, \$0, 10	0x00400028	0x3402000a
syscall	0x0040002c	0x0000000c

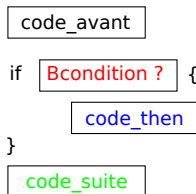
Détermination de l'adresse de saut

Première passe

Langage d'assemblage	Adresse d'implantation	Binaire
.data		
n: .word 0x0A		
.text		
lui \$3, 0x1001	0x00400000	0x3c031001
lw \$4, 0(\$3)	0x00400004	0x8c640000
xor \$5, \$5, \$5	0x00400008	0x00a52826
debut:		
beq \$4, \$0, non_calcul	0x0040000c	0x1080 0003
add \$5, \$5, \$4	0x00400010	0x00a42820
addi \$4, \$4, -1	0x00400014	0x2084ffff
j debut	0x00400018	0x08100003
non_calcul:		
ori \$4, \$5, 0	0x0040001c	0x34a40000
ori \$2, \$0, 1	0x00400020	0x34020001
syscall	0x00400024	0x0000000c
ori \$2, \$0, 10	0x00400028	0x3402000a
syscall	0x0040002c	0x0000000c

- 1 Introduction
- 2 Instructions de saut MIPS
- 3 Calculer en fonction d'une condition
- 4 Itérer un calcul n fois
- 5 Encodage des instructions de saut et détermination de l'adresse de saut
- 6 Réalisation des structures de contrôle des langages de haut niveau à l'aide de sauts**

If-Then



code_avant

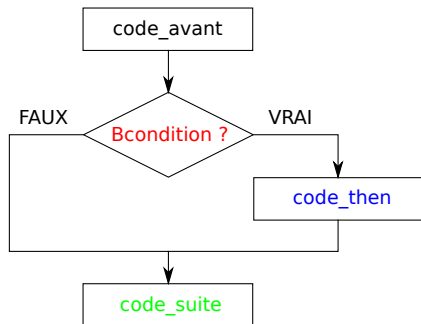
Elaboration de Bcondition

Bcondition_vraie LV

J LF

LV: code_then

LF: code_suite



code_avant

Elaboration de Bcondition

Bcondition_fausse LF

code_then

LF: code_suite

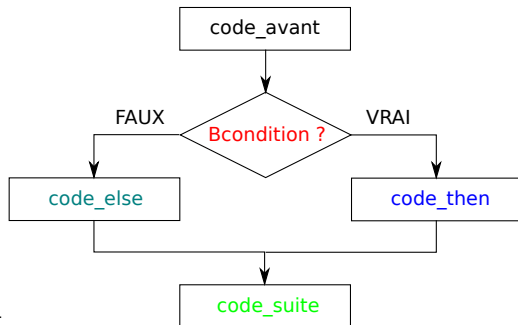
If-Then-Else

code_avant

```
if Bcondition ? {  
    code_then  
}
```

```
else {  
    code_else  
}
```

code_suite



code_avant

Elaboration de Bcondition

Bcondition_fausse label_else

code_then

] label_fin

label_else: code_else

label_fin: code_suite

Boucle While

code_avant

```
while condition {  
    code_while  
}
```

code_suite

code_avant

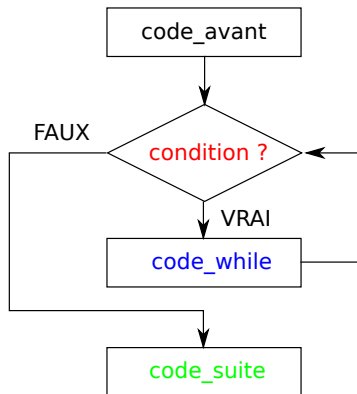
label_deb: Elaboration de condition

Bcondition_fausse label_fin

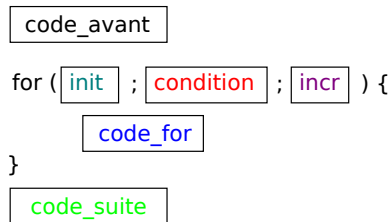
code_while

J label_deb

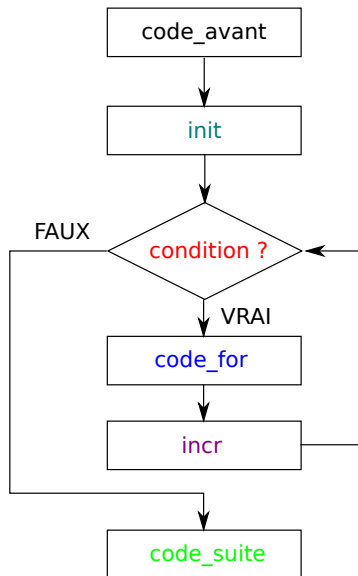
label_fin: code_suite



Boucle For



```
code_avant
init
label_deb: Elaboration de condition
Bcondition_fausse label_fin
code_for
incr
J label_deb
label_fin: code_suite
```



Boucle Do-While

code_avant

do {

code_do_while

}

while (condition);

code_suite

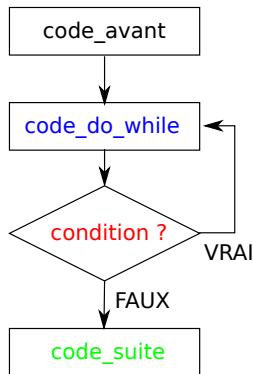
code_avant

label_deb: code_do_while

Elaboration de condition

Bcondition_vraie label_deb

code_suite



Exemple : code C

```
int i = 0;
char str[] = "une chaine quelconque\n";

int main() {
    printf("%s",str); /* affichage chaine de caracteres */
    while (str[i] != 0) {
        printf("%d", str[i]); /*affichage d'un caractere */
        i += 1;
    }
    return 0;
}
```

Exemple : code assembleur

```
.data
i: .word 0x0
str: .asciiz "une chaine quelconque\n"

.text
lui $16, 0x1001 # $16 = 0x10010000 = @i
lui $17, 0x1001
ori $17, $17, 4 # $9 = 0x10010004 = @str

ori $4, $17, 0 # $4 = @str
ori $2, $0, 4 # $2 = num affichage chaine
syscall

# code de la boucle

# fin du programme
ori $2, $0, 10 # Numéro appel système exit
syscall
```

Exemple : code assembleur

```
# code de la boucle
debut:
# Calcul de la condition et branchement
lw  $10, 0($16)    # lecture i
addu $11, $17, $10 # @str[i] = @str + i*1
lb  $12, 0($11)    # lecture de str[i]
beq $12, $0, fin

# Corps de la boucle
ori $2, $0, 1      # Appel système affichage d'un entier
lb  $4, 0($11)     # Paramètre de l'appel système
syscall
lw  $10, 0($16)    # lecture i
addiu $10, $10, 1 # i + 1
sw  $10, 0($16)    # écriture i

# Saut au début de la boucle
j   debut
fin:
```

Exemple : code assembleur

```
.data
i: .word 0x0
str: .ascii "une chaine quelconque\n"
.text
lui $16, 0x1001      # $16 = 0x10010000 = @i
lui $17, 0x1001
ori $17, $17, 4      # $9 = 0x10010004 = @str
ori $4, $17, 0       # $4 = @str
ori $2, $0, 4        # $2 = num affichage chaine
syscall
debut: # code de la boucle
# Calcul de la condition et branchement
lw $10, 0($16)      # lecture i
addu $11, $17, $10 # @str[i] = @str + i*1
lb $12, 0($11)      # lecture de str[i]
beq $12, $0, fin
# Corps de la boucle
ori $2, $0, 1       # Appel système affichage d'un entier
lb $4, 0($11)       # Paramètre de l'appel système
syscall
lw $10, 0($16)      # lecture i
addiu $10, $10, 1   # i + 1
sw $10, 0($16)      # écriture i
# Saut au début de la boucle
j debut
fin:
# fin du programme
ori $2, $0, 10      # Numéro appel système exit
syscall
```

conclusion

On a vu

- Les instructions de saut conditionnel (commencent par `b`) et inconditionnel (commencent pas `j`), ainsi que leur codage binaire
- L'écriture de code comportant des parties exécutées conditionnellement (`if-then` ou `if-then-else`)
- L'écriture de code comportant des itération (boucles)
- Le principe de traduction de structure de contrôle de haut niveau en assembleur.

Vous devez

- Connaître les instructions de saut et savoir les coder en binaire
- Être capable d'écrire des codes comportant des parties exécutées conditionnellement (`if-then` ou `if-then-else`) et/ou des boucles
- Savoir traduire des codes C (sans variable locale) avec des structures de contrôle en assembleur

Prochain cours : pile d'exécution et variables locales !