

Edgar Aguadé Nadal

Quadern de laboratori Estructura de Computadors

Emilio Castillo
José María Cela
Montse Fernández
David López
Joan Manuel Parcerisa
Angel Toribio
Rubèn Tous
Jordi Tubella
Gladys Utrera

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Quadrimestre de Primavera - Curs 2014/15



Aquest document es troba sota una llicència Creative Commons

Licencia Creative Commons

Esta obra está bajo una licencia Reconocimiento-No comercial-Compartir bajo la misma licencia 2.5 España de Creative Commons. Para ver una copia de esta licencia, visite

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/>

o envíe una carta a

Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Usted es libre de:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

- **Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).
- **No comercial.** No puede utilizar esta obra para fines comerciales.
- **Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.
- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Advertencia: Este resumen no es una licencia. Es simplemente una referencia práctica para entender el Texto Legal (la licencia completa).

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.

Sessió 2: Traducció de Programes

Objectiu: Es pretén que l'alumne sigui capaç de fer ús dels desplaçaments i de les operacions lògiques bit a bit, així com de traduir sentències condicionals, bucles i subrutines. L'alumne ha de saber aplicar les regles de codificació de subrutines que formen part del ABI de MIPS (en concret el subconjunt de regles que s'estudien en EC).

Lectura prèvia

Desplaçaments de bits

Per desplaçar els bits d'un número una posició a la dreta o a l'esquerra es copia cada bit a la posició adjacent. En un desplaçament de múltiples posicions, el resultat és el mateix que s'obtindria de desplaçar un lloc múltiples vegades.

Per desplaçar els bits d'un número una posició a la dreta es pot fer de dues formes diferents: desplaçament lògic i desplaçament aritmètic. Mentre que en un desplaçament lògic a la dreta, el bit de major pes passa a valdre 0, en un desplaçament aritmètic a la dreta, aquest bit no es modifica. Els desplaçaments d'una posició a l'esquerra només són lògics, i el bit de menor pes sempre passa a valdre 0.

El repertori MIPS disposa de 3 instruccions per fer desplaçaments lògics de shamt posicions, essent shamt un immediat sense signe de 5 bits: sll per als desplaçaments lògics a l'esquerra, srl per als desplaçaments lògics a la dreta, i sra per als desplaçaments aritmètics a la dreta.

Exemples:

sll \$s1, \$s2, 3	# \$s1 <- \$s2 despl. lògic esquerra 3 bits
srl \$s1, \$s2, 3	# \$s1 <- \$s2 despl. lògic dreta 3 bits
sra \$s1, \$s2, 3	# \$s1 <- \$s2 despl. aritmètic dreta 3 bits

Un dels usos freqüents dels desplaçaments és el de multiplicar i dividir per potències de 2. El desplaçament d'un lloc a l'esquerra d'un nombre natural o enter equival a multiplicar-lo per 2. Tant el desplaçament lògic d'un lloc a la dreta d'un nombre natural com el desplaçament aritmètic d'un lloc a la dreta d'un nombre enter parell o positiu equival a dividir el nombre per 2. En canvi, per als enters senars negatius l'operació de desplaçament a la dreta no equival a la divisió per 2, com la que fa la instrucció div, on el reste té sempre el signe del dividend, sinó que obté un quocient i reste diferents, on el reste és sempre positiu.

Operacions lògiques bit a bit

Les operacions lògiques and i or bit a bit s'usen freqüentment per modificar bits individuals dins d'una dada. Un operand conté la dada a modificar i l'altre operand conté un patró de bits, anomenat màscara, que indica quins bits es modificaran i quins no. L'operació and serveix per posar bits a zero, i la màscara corresponent ha de tenir a zero solament aquells bits a modificar. Aquesta operació s'usa sovint per consultar determinats bits (posant a zero la resta). A la inversa, l'operació or serveix per posar bits a 1, i la màscara corresponent ha de tenir a 1 solament aquells bits a modificar. Anàlogament, l'operació xor serveix per complementar bits (zeros passen a ser uns i viceversa), i la màscara ha de tenir uns solament als bits que es volen complementar. El repertori d'instruccions MIPS disposa de les instruccions and, or i xor bit a bit entre registres. També disposa de les intruccions andi, ori i xori, que fan el mateix entre un registre i un immediat de 16 bits al qual la instrucció convertirà previament en números de 32 bits aplicant una extensió de zeros. El llenguatge C disposa de les mateixes operacions bit a bit, usant els operadors & (and), | (or) i ^ (xor).

Per exemple:

```
li $s2, 0xFFFF3      # $s2 = 1111 1111 1111 1111 1111 1111 1111 0011
and $s1, $s1, $s2    # posa a 0 els bits 2 i 3 de $s1
li $s2, 0x000C       # $s2 = 0000 0000 0000 0000 0000 0000 0000 1100
or $s3, $s3, $s2     # posa a 1 els bits 2 i 3 de $s3
li $s4, 0x0f0f       # $s4 = 0000 0000 0000 0000 0000 1111 0000 1111
xori $s4, $s4, 0x00ff # complementa els bits 0 al 7 de $s4 = 0x00000ff0
```

També està disponible la instrucció nor bit a bit (fa la or dels operands i complementa el resultat). La nor s'usa freqüentment per a realitzar l'operació lògica not bit a bit:

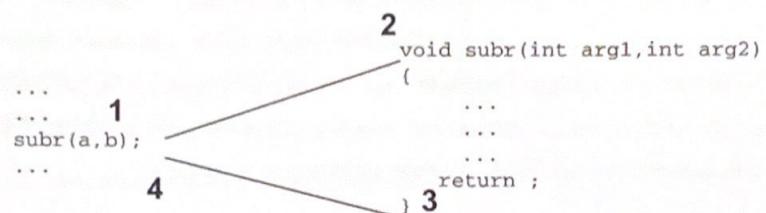
```
li $s2, 0xFFFF3      # $s2 = 1111 1111 1111 1111 1111 1111 1111 0011
nor $s2, $s2, $zero  # $s2 = 0000 0000 0000 0000 0000 0000 0000 1100
```

Subrutines

A continuació trobareu un resum amb els conceptes més importants que s'han introduït en aquest tema de subrutines: definicions, fases de la seva gestió, i traducció de codi en alt nivell que utilitza funcions i/o procediments.

Definicions

Subrutina: Conjunt d'instruccions de llenguatge màquina (o assemblador) que realitza una tasca parametrizable. Serveix per implementar les funcions i accions que existeixen en els llenguatges d'alt nivell. En llenguatge C una crida a una subrutina té la següent estructura



- | | |
|--|---------------------------------------|
| 1 Preparació i execució de la crida | 2 Inici de la subrutina |
| 4 Recuperació del estat i resultat | 3 Finalització de la subrutina |

En el moment de traduir un codi amb subrutines a MIPS, hem de mantenir la mateixa estructura assegurant-nos de mantenir, a la vegada, la coherència en els registres i la memòria, es a dir, el context del programa.

Context del programa

Hem de recordar que en una subrutina es poden modificar els valors dels registres, per aquesta raó s'hauran de preservar els valors anteriors utilitzant la pila. L'encarregat de guardar aquests valors depèn del registre de què es tracti. Suposant que A crida B:

1) Rutina A

- Els registres \$t0-\$t9, \$a0-\$a3, \$v0-\$v1, són registres temporals, de manera que la rutina B no ens assegura que els preservi. Per tant és la rutina A, que fa la crida, qui ha d'assegurar els valors d'aquests registres si és que els necessita posteriorment, salvant-los en registres \$s.

2) Rutina B

- Els registres `$s0-$s7` poden formar part del context de la rutina A que ha fet la crida, i s'ha de preservar el seu valor. Per tant, la subrutina B ha d'encarregar-se de salvar el seu valor a la pila en cas que els modifiqui.
- El registre `$ra` també s'ha de salvar a la pila en el cas que B sigui una subrutina multinivell (és a dir, si B fa crides a altres routines) ja que, en cas de no fer-ho, es perdria l'adreça de retorn.

La reserva d'espai en la pila es farà una sola vegada al principi de cada subrutina, decrementant el punter de la pila (*stack pointer*), que es troba al registre `$sp`, tantes posicions de memòria com siguin necessàries per fer lloc a tots els registres que s'hagin de salvar durant la subrutina, i també per guardar variables locals que no es puguin ubicar en registres. Al final de la subrutina, just abans de retornar, es restaurarà l'estat inicial de la pila, incrementant el valor de `$sp` en la mateixa quantitat decrementada al principi. Per exemple, suposant que la subrutina sub ha de guardar espai per a 2 registres:

```
sub:
    addiu $sp, $sp, -8
    ...
    addiu $sp, $sp, 8
    jr $ra
```

Fases de la gestió d'una subrutina

Una vegada explicats els punts anteriors, pasem a resumir quines són les coses a fer en MIPS, en cadascun dels 4 passos de les subroutines. Com hem vist anteriorment, cada part s'encarrega de mantenir coherent el seu àmbit d'execució, i preparar el següent pas inicialitzant, si s'escau, els registres (com per exemple, els paràmetres en els registres `$a0-$a3`, el valor de retorn `$v0`, etc.).

Utilitzant l'esquema donat anteriorment, passem a definir les regles per a una correcta interfície entre el programa que crida a una subrutina i el codi d'aquesta subrutina. Aquestes regles corresponen a la interfície de programació MIPS adoptada en l'assignatura d'EC.

- 1) Preparació i execució de la crida (codi de A que fa la crida): Abans de cridar a una subrutina cal *salvar alguns registres, passar els paràmetres, i executar la instrucció de salt*.

- Salvar registres: Només cal salvar els registres \$a0-\$a3,\$t0-\$t9, si aquests guarden part del context actual del programa (els seus valors es necessiten més enllà de la crida). Per preservar-los utilitzarem registres \$s.
 - Pas de paràmetres: Utilitzarem els registres \$a0-\$a3 per passar els paràmetres de la subrutina (com a màxim seran quatre).
 - Executar la instrucció de salt: executem la instrucció jal etiqueta la qual escriurà l'adreça de retorn en \$ra.
- 2) Inici de la subrutina (codi de B, subrutina cridada):
- Reservar memòria a la pila per al bloc d'activació restant la distància adequada al \$sp (pels registres a salvar i per les variables locals que no poden anar en registres).
 - Salvar registres: si és una subrutina multinivell, salvar \$ra. Si modifica algun dels registres \$s0-\$s7, salvar-los.
- 3) Finalització de la subrutina (codi de B, subrutina cridada):
- Copiar el valor de retorn en \$v0.
 - Restaurar tots els registres salvats a l'inici (\$s0-\$s7, \$ra).
 - Alliberar tot l'espai de la pila reservat inicialment, sumant-li la distància abans restada.
 - Retornar, executant la instrucció jr \$ra.
- 4) Codi posterior a la crida (codi de A, que fa la crida). Just després de la instrucció jal cal:
- Recollir el resultat que es troba al registre \$v0, si A és una funció.

Enunciats de la sessió

Activitat 2.A: Operacions lògiques i desplaçaments

Les operacions lògiques i els desplaçaments ens permeten fer algunes operacions aritmètiques de forma més eficient donat que podem evitar, en alguns casos, l'ús d'operacions de divisió i multiplicació per potències de 2, o bucles iteratius.

A part de les intruccions d'operacions bit a bit explicades a la lectura prèvia, MARS disposa de les instruccions sllv, srlv, srv per fer desplaçaments d'un nombre variable de bits, especificant aquest número en un registre.

```
sllv $s1,$s2,$s3    # $s1<-$s2 despl. lògic esq. tants bits com indica $s3
srlv $s1,$s2,$s3    # $s1<-$s2 despl. lògic dreta tants bits com indica $s3
srv $s1,$s2,$s3     # $s1<-$s2 despl. aritm. dreta tants bits com indica $s3
```

Nota: aquestes 3 instruccions sols usen els 5 bits de menor pes del tercer operand (\$s3).

La resta de bits s'ignoren (poden no ser zero).

Exercici 2.1: Escriviu un programa en MIPS que, donats dos enters X i Y que es troben als registres $\$s0$ i $\$s1$ respectivament, inverteixi (complementi) els X bits de menys pes de Y . Per fer-ho, podeu usar operacions bit a bit, però no podeu fer servir instruccions de salt ni de comparació, i us ha de sortir un programa amb menys de 5 instruccions.

Nota: podeu fer servir la següent sentència en C (on l'operador \wedge expressa la xor bit a bit, i l'operació $1 \ll X$ significa "el valor 0x01 desplaçat X bits a l'esquerra"):

$$Y = Y \wedge ((1 \ll X) - 1);$$

```
la $t0, 0x01
sllv $t0, $t0, $s0 # 1 << X
addiu $t0, $t0, -1 # (1 << X) - 1
xor $s1, $s1, $t0 # Y ^ ((1 << X) - 1)
```

Copieu el codi de l'exercici anterior al fitxer s2a.s. Comproveu que el codi és correcte mirant els valors inicial i final de $\$s1$, per a diferents valors de X i Y .

Activitat 2.B: Sentències if-then-else

El codi següent comprova si el codi ASCII que conté la variable *num* correspon a un símbol alfabètic, a un dígit decimal, o a cap dels dos (pot ser de control, un símbol de puntuació, etc). Si és un símbol alfabètic, escriurem a la variable *result* el valor de *num*. Si és un dígit decimal, hi escriurem el seu valor en format d'enter. Altrament, hi escriurem un -1:

```
int result = 0 ;
char num = '7' ;
main()
{
    if (((num >= 'a') && (num <= 'z')) || ((num >= 'A') && (num <= 'Z')))
        result = num;
    else
        if ((num >= '0') && (num <= '9'))
            result = num - '0';
        else
            result = -1;
}
```

Figura 2.1: Programa que classifica un caràcter

Completa l'exercici 2.2 abans de continuar.

Exercici 2.2: Tradueix a assemblador MIPS el programa de la Figura 2.1

```
.data
result: .word 0
num: .byte '7'
.text
main:
    la $t0, result
    la $t1, num      #&num
    lb $t2, 0($t1) # num
    #inici cond_if
    li $t2, 'a'
    blt $t1, $t2, or #salta si fals
    li $t2, 'z'
    ble $t1, $t2, if #salta si cert
    org
    li $t2, 'A'
    blt $t1, $t2, else_if #sol si fals
    li $t2, 'Z'
    blt $t1, $t2, else_if # 'Z' < num
    if:
        sw $t1, 0($t0)
        b end_if
    else_if:
        li $t3, '0'
        blt $t1, $t3, else # salta si fals
        li $t2, '9'
        blt $t1, $t2, else # salta si fals
        # instruccions if
        subc $t1, $t1, $t3
        sw $t1, 0($t0) # (result = num-'0')
        b end_if
    else:
        li $t2, -1
        sw $t2, 0($t0)
end_if:
```

Copia el programa de l'exercici anterior al fitxer s2b.s i comprova que al final de la seva execució el valor de *result* és 7. Fés la prova per a diferents valors de *num*: 'a', 'z', 'Z', '0' i ';' , per exemple. Els seus codis ASCII són 0x61, 0x7a, 0x5A, 0x30 i 0x3B, respectivament.

Activitat 2.C: Calcular el caràcter més freqüent d'un string

El següent programa (veure Figura 2.2) declara el vector global *w* del tipus string, format per 32 caràcters. Els 31 primers representen díigits numèrics (del '0' al '9', codificats amb valors del 48 al 57), i l'últim és el sentinel·la (caràcter null = '\0', codificat amb el valor 0).

La funció *moda* es crida una vegada des del *main* per tal que calculi quin és el caràcter més freqüent de la cadena *w*. Aquesta funció construeix, al vector local *histo* de 10 enters, un histograma que emmagatzema, per cada possible caràcter numèric, la seva freqüència d'aparició. A més a més, a cada pas, el caràcter més freqüent es guarda a la variable local *max*. La funció consta de dos bucles, un per inicialitzar les freqüències a zero, i l'altre per recórrer la cadena de caràcters, fent una crida a la funció *update* per cada caràcter visitat.

La funció *update* actualitza la freqüència del caràcter visitat en l'histograma, la compara amb la del caràcter *max*, i retorna el nou caràcter més freqüent. Dins d'aquesta funció hi ha una crida a l'acció *nofares*, que no fa res d'útil, i que està posada per facilitar la verificació de

```

char w[32] = "8754830094826456674949263746929";
char resultat; /* dígit ascii més frequent */

main()
{
    resultat = moda(w, 31);
    print_character(resultat); /* consultar lectura prèvia sessió 1 */
}

char moda(char *vec, int num)
{
    int k; histo[10];
    char max;

    for (k=0; k<10; k++) { Inicialització Histograma.
        histo[k] = 0;
    }

    max = '0';
    for (k=0; k<num; k++)
        max = '0' + update(histo, vec[k]-'0', max-'0');

    return max;
}

void nofares();
char update(int *h, char i, char imax)
{
    nofares();
    h[i]++;
    if (h[i] > h[imax])
        return i;
    else
        return imax;
}

```

Bloc d'Act
0(\$sp)| \$s0| 4
4(\$sp)| \$s1| 4
8(\$sp)| \$s2| 4
12(\$sp)| \$ra| 4
16(\$sp)| histo| 40
Tot: 56 bytes.

Bloc Act.

0(\$sp)| \$s0| 4
4(\$sp)| \$s1| 4
8(\$sp)| \$s2| 4
12(\$sp)| \$ra| 4
total 16 bytes

Figura 2.2: Càlcul de la moda.

la correctesa del codi que genereu.

Completa l'exercici 2.3 abans de continuar.

Exercici 2.3: Tradueix a assemblador MIPS el codi de les funcions *moda* i *update* de la

Figura 2.2. No oblidis posar dins d'*update* la crida a la subrutina *nofares*.

```

moda :
    addiu $sp,$sp,-56 #BaA
    sw $s0,0($sp)
    sw $s1,4($sp)
    sw $s2,8($sp)
    sw $ra,12($sp)
    move $s1,$a0 #&vector
    move $s2,$a1 # num
    addiu $t0,$sp,16 #&histo[0]
    li $s0,0 # k=0
    -for_1:
        slti $t1,$s0,10 # k<10
        beq $t1,$zero,-end_for_1 #salta si fals
        sw $zero,0($t0) #histo[k]=0
        addiu $t0,$t0,4 #&histo[k+1]
        addiu $s0,$s0,1 # k++
        b_for_1
    -end_for_1
    li $v0,10 # max=0
    li $s0,0 # k=0
    -for_2:
        slt $t1,$s0,$s2 # k < num
        beq $t1,$zero,-end_for_2 #salta si fals
        li $t2,10
        addiu $s0,$sp,16 #&histo[0]
        lb $a1,0($s1) # vec[k]
        subu $a1,$a1,$t2 # vec[k]-10
        subu $a2,$v0,$t2 # max-10
        jal update
        addiu $v0,$v0,10 # max=10+update
        addiu $s1,$s1,1 #&vec[k+1]
        addiu $s0,$s0,1 # k++
        b_for_2
    -end_for_2
    lw $s0,0($sp)
    lw $s1,4($sp)
    lw $s2,8($sp)
    lw $ra,12($sp)
    addiu $sp,$sp,56 #Alliberem BaA
    jr $ra #return

update:
    addiu $sp,$sp,-16 #Reserva BaA
    sw $s0,0($sp)
    sw $s1,4($sp)
    sw $s2,8($sp)
    sw $ra,12($sp)
    move $s0,$a0 #&h[0]
    move $s1,$a1 #i
    move $s2,$a2 #i_max
    jal nofares

```

Copia el codi anterior al fitxer s2c.s. Comprova amb el simulador que al final de l'execució, el valor de la variable *resultat* és '4'.

Activitat 2.D: (Opcional) Depuració de codi erroni en assemblador.

En aquest apartat estudiarem el programa de la Figura 2.3. El vector *alfabet* és un string que conté la llista ordenada de les lletres majúscules i, com és costum en els strings, acaba amb un byte sentinella que val 0. La funció *codifica* fa el següent: donat un string d'entrada, genera un string de sortida on cada lletra ha estat intercanviada, de la següent manera: una 'A' es convertirà en una 'Z' o viceversa; una 'B' en una 'Y' o viceversa, etc. El programa principal fa dues crides a *codifica*. La primera vegada li passa un string d'entrada *w1* = "ARQUITECTURA", i retorna un string de sortida *w2*. En la segona crida li passa com a entrada *w2*, i retorna un string de sortida *w3*:

```
char alfabet[27] = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
char w1[16] = "ARQUITECTURA";
char w2[16];
char w3[16];
int count=0;

main()
{
    count = codifica(w1, w2);
    count += codifica(w2, w3);
    print_string(w2); /* consultar lectura prèvia sessió 1 */
    print_string(w3); /* consultar lectura prèvia sessió 1 */
}

char g(char alfa[], char *pfrase)
{
    return alfa[25 - (*pfrase - 'A')];
}

int codifica(char *pfrasein, char *pfraseout)
{
    int i; $S2
    i = 0;
    while (*pfrasein != 0)
    {
        *pfraseout = g(alfabet, pfrasein);
        pfrasein++;
        pfraseout++;
        i++;
    }

    *pfraseout = 0;
    return i;
}
```

Figura 2.3: Programa que codifica i decodifica un string

Nosaltres hem fet una traducció a MIPS d'aquest programa (Figura 2.4), però conté tres errors. Els errors es troben localitzats en el codi de la subrutina *codifica*.

```

g:
    lb      $t0, 0($a1)
    li      $t1, 'A'
    addiu   $t1, $t1, 25
    subu   $t1, $t1, $t0
    addu   $t0, $a0, $t1
    lb      $v0, 0($t0)
    jr      $ra

```

Figura 2.4: Traducció del programa, amb errors (continuació)

Completa l'exercici 2.4 abans de continuar.

Exercici 2.4: Explica breument quins són els **3 errors** del programa, i com s'haurien de corregir:

- ① $lb\ $t0, 0($t0)$ sobra, ja que $\$t0$ ja és $*pfrasein$ i per tant $lb\ $t0, 0($t0)$ ja es $*pfrasein$.
- ② $move\ \$a1, \$s1$ passa com a paràmetre a g $pfraseout$, quan hauria de ser $pfrasein$
 ↳ Correcció: $move\ \$a1, \$s0$.
- ③ $sb\ \$v0, 0($t0)$ fa $*pfrasein = g(alfabet, pfrasein)$; quan l'assignació hauria de ser a $*pfraseout$.
 ↳ Correcció: $sb\ \$v0, 0($t0)$

Per comprovar-ho, carrega el fitxer s2d.s, que conté el programa de la Figura 2.4, i corregeix els errors. Verifica que després d'executar-se, el programa imprimeix els strings "ZIJ-FRGVXGFIZ" i "ARQUITECTURA", i que la variable global *count* guardada a la memòria val 24.

Si no és així, depura el programa: executa'l pas a pas verificant a cada pas si fa el que s'espera que faci. La depuració requereix paciència i concentració!

```

        .data
alfabet:    .asciiz "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
w1:         .asciiz "ARQUITECTURA"
w2:         .space 17
w3:         .space 17
count:      .word 0

        .text
        .globl main
main:
        addiu   $sp, $sp, -4
        sw      $ra, 0($sp)
        la      $s0, count
        la      $a0, w1
        la      $a1, w2
        jal     codifica
        sw      $v0, 0($s0)
        la      $a0, w2
        la      $a1, w3
        jal     codifica
        lw      $s1, 0($s0)
        addu   $s1, $s1, $v0
        sw      $s1, 0($s0)
        li      $v0, 4
        la      $a0, w2
        syscall          # print_string(w2)
        la      $a0, w3
        syscall          # print_string(w3)
        lw      $ra, 0($sp)
        addiu   $sp, $sp, 4
        jr      $ra

codifica:
        addiu   $sp, $sp, -16
        sw      $ra, 0($sp)
        sw      $s0, 4($sp)
        sw      $s1, 8($sp)
        sw      $s2, 12($sp)
        move   $s2, $zero  # l = 0
        move   $s0, $a0  # frasein
        move   $s1, $a1  # fraseout

while:
        lb      $t0, 0($s0) # frasein
        ①    lb      $t0, 0($t0) # malament (sobra)
        beq   $t0, $zero, fi_while
        la      $a0, alfabet
        ②    move   $s0, $a0 # malament $s1 és fraseout
        jal     g
        ③    sb      $v0, 0($s0) # malament $s0 és frasein
        addiu   $s0, $s0, 1
        addiu   $s1, $s1, 1
        addiu   $s2, $s2, 1
        b       while

fi_while:
        sb      $zero, 0($s1)
        move   $v0, $s2
        lw      $ra, 0($sp)
        lw      $s0, 4($sp)
        lw      $s1, 8($sp)
        lw      $s2, 12($sp)
        addiu   $sp, $sp, 16
        jr      $ra

```

Figura 2.4: Traducció del programa, amb errors (continua a la pàg. següent)