T3: Traducció De Programes

Shift Left "<<" ←00 ≡ Multiplicar || Shift Right 00→ ">>" ≡ Dividir || x = $\log_2$(divisor)

li $t0, 0x99999999	li $t0, 0xFFFF FF02
sra $t0, $t0, 2 # $t0 = 0xE6666666	li $t1, 0x99999999
	srlv $t2, $t1, $t0
	# $t2 = $t1 <<< (2)

Pq. 'srlv' agafa desplaçant de variable i agafa 4:0 bits. 1111,2002,0010 — F F F F F 0 2

a = ~a ≡ nor $t0, $t0, $zero # Fer not

& AND „ | OR „ ^ XOR „ ~ NOT — ! Totes bit a bit

#AND: Comproven bits o "esborrar-los"

#OR: Ficar '1's al número

#XOR: Complementar bits #Donat que 1^1 = 0 „ 1^0 = 0^1 = 1

//Això para a=0 si a≠0 o a=1 si a==0 // Aquí considerem que a ∈ ℕ i l'únic menor que 1 és el 0.

!a ≡ sltiu $t1, $t0, 1 — ! Impo que sigui sltiu pq. el negatiu donaria error.

Per normalitzar ≡ sltu $t0, $zero, $t0 Pq. tot valor (major 0) és més gran). Unsigned

c = (a <= b) ≡ slt $t4, $t1, $t0 # b<a
sltiu $t2, $t4, 1 # !(b<a)

! Ara els salts, el num. es refereix a nº d'instruccions. És en Ca2 per anar ↶ / ↷

blt rs, rt, etiq. ≡ slt $at, rs, rt
if (rs<rt) bne $at, $zero, etiq.

bge rs, rt, etiq ≡ slt $at, rt, rs
if (rs ≥ rt) beq $at, $zero, etiq

Subrutina

- jal memoritza mem. de retorn en $ra
- jr salta a adreça de la variable

! $a0 - $a3 són els reservats per paràmetres. ! $f12 i $f14 per coma flotant
#Si és char o half és fa SE → 0's si unsigned / → signe si és Ca2

! $v0 és el registre de return $f0 si és coma flotant

Reservem l'espai per a vectors (o mat.) de la funció o variables segures o variables internes

(Variables de la funció que tinguin & , i això vol dir diferent de la BA)

1) Fem els "move $a0, $t0" donat que els passem com a par.

2) Fem el salt a l'etiqueta de la subrutina "jal func"

3) Ara hem de guardar el return "move $t1, $v0"

Sense fer move
Recorda fer "la a $tY"

#Si el valor a passar està en variable global podem fer → lb / lh / lw $a.x, 0($tY)

! Local té més preferència que global. # "$sp" stack pointer

El fons de pila "0x7FFF FFFC" i la 'C' és pq. pugui ser múltiple de 4.

"addiu $sp, $sp, -4" SEMPRE per guardar $ra

@w[i] (En una subrutina) $sp + X + i * T   X = Espai entre inici de SP fins W[0]   (Tamany)

→ Temporals: $t0 - $t9, $v0 - $v1, $a0 - $a3, $f0 - $f19.

→ Segurs: $s0 - $s7, $sp, $ra, $f20 - $f31.   El fill pot (o no) tocar els segurs: Al inici farà la còpia dels que tocarà.

"El fill s'encarrega que els segurs estiguin iguals al acabar per no molestar al pare"

En C, els valors a funcions es passen per valor i no per referència (amb punter).

No pots fer com en C++ (int &a) i et modifica 'a'. Per fer-ho en C s'ha de fer *

Si "a" és local i surt "&a" en el codi → "a" s'ha de guardar en BA.

$ra només es guarda quan hi ha crida a funció. // Si main fa crida, main necessita pila per guardar-se $ra.

Estructura de la memòria

Heap: Regió de la memòria (prèviament reservada pel S.O.) per l'assignació dinàmica de mem. durant l'execució del programa. Si es necessita més espai, pot ampliar-se fins límit físic. Aquí se sol guardar estructures de dades que canvien de tamany o no se sap l'exacte.

* Recorda que variables globals van al segment de dades i no aquí.

Pila: Temps de vida més curt i està vinculat a l'àmbit de funcions. Es reserva un bloc cada cop que es crida i on es guarden dades locals i al return s'allibera.

L'espai total de la Pila és inferior i si no es gestiona hi ha Stack Overflow.

// Totes dues són gestionades pel S.O. i s'ubiquen a la R.A.M.

Compilació, assemblatge, enllaçat i càrrega

1. Compilació: El programa compilador agafa fitxer i el tradueix a ensamblador.
2. Assemblatge: El programa assemblador agafa fitxer assembly i tradueix a màquina en objecte.
3. Linker: Combina multiples fitxers obj. i fitxers de la biblio i fica en 1 executable.
4. Loader: Programa del S.O. que llegeix executable i el carrega en memòria.

Donat que hi ha etiquetes, es necessita saber l'ubicació final d'aquestes:

- Llista de reubicació: Pos. Instrucció + Tipus ins. + @provisional. // Ajustar direccions dins codi comp
- Llista referència no resoltes: Llista funcions, variables o recursos que no estan mateix fitxer.
- Taula de símbols globals: Conté identificadors que tenen àmbit global.

// Linker agafa Ref. No Res + Símb. Globals i emparella. Si algun no pot → ERROR

Pau Bru Ribes.

(3.5). a) Shift esquerre 1 posició

```
slti $t0,$t1,0  # Aquí no é unsigned pq. m'interessa saber si MSB és 1 → negatiu
sll  $t1,$t1,1
sll  $t2,$t2,1
addu $t2,$t2,$t0  # Sumo el resultat del check i pot sumar → 0x00000000 / → 0x00000001 (Era neg)
```

(3.9). $t4 ≅ ABCD EFGH IJKL MNOP abcd efgh ijkl mnop (A = bit 32, p = bit 0; bit 4 marked above "l")

a) MSB (A) sigui LSB. Rotar circularment $t4.

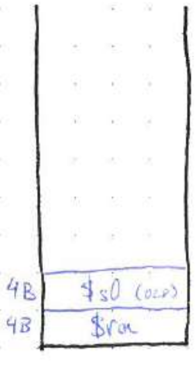
```
srl  $t5,$t4,8       # 0000 0000 0000 0000 ABCD EFGH IJKL MNOP
andi $t5,$t5,0x8000  # 0000 0000 0000 0000 A000 0000 0000 0000
srl  $t5,$t5,7       # 0000 0000 0000 0000 0000 0000 0000 000A
sll  $t4,$t4,1       # BCDE FGHI JKLM NOPa bcde fghi jklm nop0
addu $t4,$t4,$t5     # BCDE FGHI JKLM NOPa bcde fghi jklm nopA
```

(*). Quina declaració correspon la crida

Crida	Declaració	Variables
1) a = f(V1); (&V1[0])	int f(short *pi);	short V1[10];
2) a = f(p);	int f(int vi[]); vi[] = punter	char V2[20];
3) a = f(&V2[10]);	int f(char *pi);	int a;
4) a = f(V2[10]);	int f(char c);	int *p;
5) a = f(*p);	int f(int i);	

(3.28)

```
int s1(int x, long long v[], int *p){     ($a2 = p)
    long long *k;                          ($ra)
    k = &v[x];
    x = s2(*p, k, v);
    return *p + x;
}
```




```
s1:  addiu $sp,$sp,-8
     sw    $s0,0($sp)     # El del meu pare
     sw    $ra,4($sp)     # pq faig crida s2
     move  $s0,$a2        # *p
     sll   $t0,$a0,3      # x*8
     addu  $t0,$a1,$t0    # &v[x]
     lw    $a0,0($a2)     # $a1 = *p      (L'ordre IMPO)
     move  $a2,$a1        # $a2 = v[0]
     move  $a1,$t0        # $a1 = k
     jal   s2
     lw    $t0,0($s0)     # Apuntar a la mateixa dir que m'han passat en $a2.
     addu  $v0,$t0,$v0
     lw    $s0,0($sp)
     lw    $ra,4($sp)
     addiu $sp,$sp,8
     jr    $ra
```

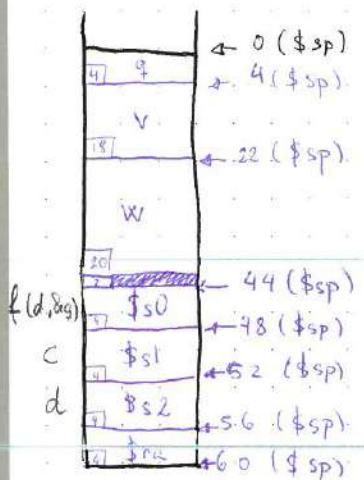
(*) Fixe't que 'q' és local però demana '&' així que s'ho de guardar en pila.

```
int f(int m, int *n);

int g(char *y, char *z);

char exemple(int a, int b, int c) {
                $t0 $t1 $t2
    int d, e, q;
    char v[18], w[20];
    d = a + b;
    e = f(d, &q) + g(v, w);

    return v[e+q] + w[d+c];
}
```



(3.34).

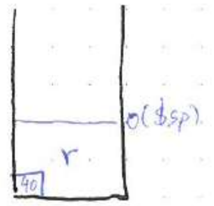
a) Tradueix la sentència: x=A(x, z, w); c) Tradueix A

Repetir ben fet.

```
la $t0, x
lb $t0, 0($t0)  x
la $t1, z
lw $t1, 0($t1)  z
la $t2, w       x w[0]
move $a0, $t0
move $a1, $t1
move $a2, $t2
jal A
move $t0, $v0   (ratllat)
la $t0, x
sb $v0, 0($t0)
```

Recorda que és global. Hi ho millores eficiència no fent tants 'move'.

b) Dibuixa BA.



TOTAL = 40B

```
addiu $sp, $sp, -40
li $t3, 12
addu $a2, $a2, $t3  # v[3]
lb $t0, 3($a2)  # c = v[3]
sll $a1, $a0, 2  # v*4
addu $t1, $sp, $a1
sw $a0, 0($t1)  # v[x] =
move $v0, $t0
addiu $sp, $sp, +40
```

Fixet que és vect. de char i simplement fes suma amb offset.

(3.26). Tradueix la subrutina.

```
int sub3(int param) {
    int ret, tam = 7;
    ret = s3(v, tam, param, 'D');
    return ret + tam + param;
}
```

$ra · tam: s0 · param: s1

TOTAL = 12 B

```
addiu $sp, $sp, -12
sw    $ra, 0($sp)
sw    $s1, 4($sp)
sw    $s0, 8($sp)
li    $t0, 7     # ret = 7
li    $1, 7      # tam = 7
move  $s0, $a0
la    $a0, v
move  $a1, $s1
move  $a2, $s0
li    $a3, 'D'
jal   s3
move  $t0, $v0
addu  $t0, $t0, $s1   ret = ret + tam
addu  $t0, $t0, $s0   ret = ret + param
move  $v0, $t0        retu. sum
lw    $s0, 8($sp)
lw    $s1, 4($sp)
lw    $ra, 0($sp)
addiu $sp, $sp, 12
jr    $ra
```

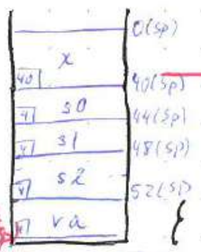
Aquí si haguessin operat amb $v0 ens estalviem 2 ins.

Cuidado que "ret" és int

```
addiu $sp, $sp, -56
sw    $s0, 40($sp)
sw    $s1, 44($sp)
sw    $s2, 48($sp)
sw    $ra, 52($sp)
move  $s1, $a1
move  $s2, $a2
sll   $s0, $s0, 2
addu  $t0, $a0, $s0
lw    $t0, 0($t0)   # p1[x2]
sll   $t1, $s1, 2
addu  $t1, $sp, $t1   # &x1[p2]
addu  $t2, $t0, $a2
sw    $t2, 0($t1)   x1[p2] = sum
li    $t3, 12
addu  $t3, $a0, $t3
lb    $a2, 0($t3)   # $a2 = p1[3]
move  $a1, $s1      # $a1 = p2
move  $a0, $sp      # $a0 = x1[0]
jal   sub
addu  $s0, $s1, $s2
```

x2 no és global així que no fa falta que el guardem.

Recorda fer lw $sX, Y($sp)



TOTAL = 56B

No sé si haig de mostrar recuperació de la BA i tal

(3.36) Tradueix.

```
void sub(int p1[10], int p2, int p3) {
    int x1[10], x2;
    ...
    x1[p2] = p1[x2] + p3;
    ...
    sub(x1, p2, p1[3]);
    ...
    x2 = p2 + p3;
    ...
}
```

$ra · x2: S0 · p2: S1 · p3: S2

no que assegurar $a0

(3.34) Traduix. A

```
char x;
int  z;
char w[20];

char A(char i, int k, char *k){

    int r[10];
    char c; ---> #$t0
    c = w[3];
    r[k] = i;
    return c;

}
```

```
addiu $sp, $sp, -40
```

∇ Recorda que '*k' és un punter i guarda una direcció de memòria. Així que no s'ha de fer un "la k"!

```
lb  $t0, 3($a2)

sll $t2, $a1, 2
addiu $t2, $t2, $sp
sw  $a0, 0($t2)

move $v0, $t0

addiu $sp, $sp, 40
```

∇ Recorda SEMPRE

Pau Bru Ribes

3.5.a , 3.9

EC Examen de Problemes

Exercici 1 (problema 2.29 de la col.lecció)

Donades les següents declaracions de variables globals, en llenguatge C:

```
char *punterc;
short *punterh;
int *punteri;
long long *punterd;
```

Tradueix a assemblador MIPS les següents sentències:

a) `punterc++;`

b) `punteri++;`

c) `punterh++;`

d) `punterd++;`

e) `*punteri = *punteri + 5;`

f) `*punterh = *punterh + 10;`

Exercici 2 (problema 2.28 de la col.lecció)

Donades les següents declaracions de variables globals, en C:

```
int vec[6] = {2, 4, 0, 6, 8, 0};
int *punter;
```

Tradueix a una única sentència en C el conjunt d'iinstruccions de cada apartat:

a)
```
la $t0, punter   &punter
la $t1, vec + 8  &vec[2]
sw $t1, 0($t0)   punter = &
```

b)
```
la $t0, punter   &puter
lw $t1, 0($t0)   puter
addiu $t1, $t1, 4  puter++
sw $t1, 0($t0)
```

c)
```
la $t1, vec      &vec[0]
lw $t3, 0($t1)   vec[0]
lw $t4, 4($t1)   vec[1]
addiu $t3, $t3, $t4
sw $t3, 4($t1)
```

d)
```
la $t1, vec      &vec[0]
la $t0, punter   &puter
lw $t2, 0($t0)   puter
lw $t3, 0($t2)   *puter
addiu $t3, $t3, 1  *puter+1
sw $t3, 8($t1)   vec[2]=
```

e)
```
la $t0, punter   &puter
lw $t2, 0($t0)   puter
lw $t3, 0($t2)   *puter
addiu $t3, $t3, 1  *puter+1
sw $t3, 8($t2)   *(puter+2)=*puter+1
```

Ian Isern Ribes

①.

a) punterc++;

```
la $t0, punterc
lw $t1, 0($t0)
addiu $t1, $t1, 1
sw $t1,0($t0)
```

b) punteri++;

```
la $t0, punteri
lw $t1, 0($t0)
addiu $t1, $t1, 4
sw $t1, 0($t0)
```

c) punterh++;

```
la $t0, punterh
lw $t1,0($t0)
addiu $t1, $t1,2
sw $t1, 0($t0)
```

d) punterd++;

```
la $t0, punterd
lw $t1, 0($t0)
addiu $t1, $t1, 8
sw $t1, 0($t0)
```

e) *punteri = *punteri + 5;

```
la $t0, punteri
lw $t0, 0($t0) #punteri
lw $t1, 0($t0) # *punteri
addiu $t2, $t1, 5
sw $t2, 0($t1)
```

f) *punterd = *punterd + 10;

long long int ≡ 64 bits i MIPS és de 32. Necessitem 2 registres i operar amb els 2.

```
la $t0, punterh
lw $t1, 0($t0) #Part baixa
lw $t2, 4($t0) #Part alta
addiu $t1, $t1, 10 #Sumem part baixa
[...]
sw $t1, 0($t0)
sw $t2, 4($t0)
```

[...] =

```
sltiu $t3, $t1, 10
addu $t2, $t2, $t3
```

#Comprobar si hi ha carry i sumar-lo (si no hi ha se sumarà 0 així que np)

- "sltiu" pq. necessitem veure si hi ha carry. La u és important pq. -7<10 pero -7 és vàlid.
- addu del $t3 és correcte pq està normalitzat

$t3 = → 0x00000000
→ 0x00000001 ← i seria el carry

②.

a) punter = &vec[2];

b) punter++;

c) vec[1] = vec[0] + vec[1];

d) vec[2] = *punter + 1;

e) *(punter + 2) = *punter + 1;