UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tema 3
Traducció de programes

Estructura de Computadors (EC)
2023 - 2024 Q2
Adrià Armejach (adria.armejach@upc.edu)

Exercici

Donada la següent sentència escrita en alt nivell en C:

```
if (((a<=b)&&(b!=0)) || (((b%8)^0x0005)>0))
        z=5;
else
        z=a-b;
```

Completa el següent fragment de codi MIPS, que tradueix l'anterior sentència, escrivint en cada calaix un mnemònic d'instrucció o macro, etiqueta, registre o immediat. Les variables `a`, `b` i `z` són de tipus `int` i estan inicialitzades i guardades als registres `$t0`, `$t1` i `$t2`, respectivament.

```
        bgt     $t0, $t1,       etq2
etq1:   bne     $t1, $zero,     etq5
etq2:   andi    $t3, $t1,       7
etq3:   xori    $t5, $t3,       5
etq4:   ble     $t5, $zero,     etq7
etq5:   li      $t2, 5
etq6:   b       etq8
etq7:   subu    $t2, $t0, $t1
etq8:
```

Sentències iteratives: *while*, *for*, i *do-while*

Sentència *while*

Sigui el cas general en C:

```
while (condicio)
      sentencia_cos_while
```

El patró en MIPS serà:

```
while:
        avaluar condicio
        salta si és falsa a fiwhile
        traducció de sentencia_cos_while
        salta a while
fiwhile:
```

Exemple sentència *while*

- Tradueix a MIPS el codi en C suposant que `dd`, `dr` i `q` son enters emmagatzemats a `$t1`, `$t2` i `$t3`:

```
q = 0;
while (dd >= dr) {
     dd = dd - dr;
     q++;
}
```

```
     move $t3, $zero        # q = 0
while:
blt $t1, $t2, fiwhile  # salta si dd<dr
     subu $t1, $t1, $t2     # dd = dd - dr
     addiu $t3, $t3, 1 # q++
     b while
fiwhile:
```

Optimització: Avaluar condició al final del bucle

- Tradueix a MIPS el codi en C suposant que `dd`, `dr` i `q` son enters emmagatzemats a `$t0`, `$t1`, `$t2` i `$t3`:

```
q = 0;
while (dd >= dr) {
     dd = dd - dr;
     q++;
}
```

```
          move $t3, $zero        # q = 0
          blt $t1, $t2, fiwhile  # salta si dd<dr
while:
          subu $t1, $t1, $t2
          addiu $t3, $t3, 1
          bge  $t1, $t2, while   # salta si dd>=dr
fiwhile:
```

Sentència *for*

```
for (s1; condicio; s2)
    s3;
```

- És equivalent a un while:

```
s1;
while (condicio) {
    s3;
    s2;                // al final de la iteració
}
```

Sentència *do-while*

- Executa una o més iteracions mentre es compleix la condició
  - La primera iteració sempre s'executa

Sigui el cas general en C:

```
do
        sentencia_cos_do
while (condicio);
```

El patró en MIPS serà:

```
do:
        traducció de sentencia_cos_do
        avaluar condicio
        salta si és certa a do
```

Subrutines

Subrutines

- Estructura de programació - acció o funció
  - Programació modular
  - Reutilització de codi
    - Invocada des de múltiples punts del programa

Exemples de subrutines

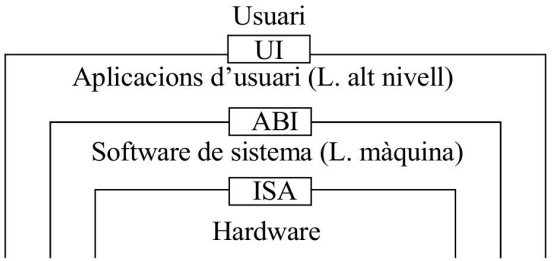
- Amb retorn de resultat

```
int max(int a, int b) {
     if(a > b) return a;
     else return b;
}
```

- Sense retorn de resultat

```
void init(int v[], int a) {
     int i;
     for (i=0; i<10; i++) v[i] = a;
}
```

Traducció a MIPS



- Traducció a MIPS de subrutines
  - Crida i retorn
  - Pas de paràmetres
  - Devolució de resultat
  - Variables locals

- Respectar regles (ABI) per garantir interoperabilitat

Crida i retorn

- Podem utilitzar la macro `b`?

```
max:
    …
    b ret


main:
    …
    b max
ret:
```

```
max:
    …
    b ret1 o ret2?


main:
    …
    b max
ret1:
    …
    b max
ret2:
```

Crida i retorn

- S'ha de guardar l'adreça de retorn a un registre
  - jal guarda l'adreça de retorn i salta a una etiqueta
    - L'adreça de retorn es guarda al registra $ra
  - jr permet saltar a l'adreça guardada a un registre

```
max:
    …
    jr $ra          # salta a @ emmagatzemada a $ra

main:
    …
    jal max         # salta a max i $ra = PC + 4
    …
    jal max         # salta a max i $ra = PC + 4
```

Pas de paràmetres i retorn de resultat

- Els paràmetres es passen en els registres `$a0 - $a3`
- El resultat es passa en el registre `$v0`

```
void main() {
     int x,y,z; /*en $t0,$t1,$t2 */
     ...
     z = suma2(x, y);
     ...
}
```

```
int suma2(int a, int b) {
     return a+b;
}
```

```
main:
     ...
     move   $a0, $t0   # passem x
     move   $a1, $t1   # passem y
     jal    suma2
     move   $t2, $v0   # z=resultat
     ...
```

```
suma2:
     addu   $v0, $a0, $a1
     jr     $ra
```

Pas de paràmetres i retorn de resultat

- Els paràmetres de menys de 32 bits (`char` o `short`) s'extenen a 32 bits
  - Extensió de zeros (`unsigned`) o de signe (`signed`)
- Vectors: es passa l'adreça base

```
short vec[3] = {5, 7, 9};
short sumv(short v[]);

void main() {
  short res;
  res = sumv(vec);
}
```

```
      .data
vec:  .half 5, 7, 9

sumv:
  ...

main:
  la $a0, vec
  jal sumv
  move $t0, $v0
```

Pas de paràmetres per valor i per referència

- Per valor: còpia del paràmetre real
- Per referència: les modificacions afecten al paràmetre real
- C només admet pas de paràmetres per valor
  - Es pot passar per valor un punter

```
int a, b;

void main() {
   int *p = &a;
   sub(p);
   sub(&b);
}
```

```
void sub(int *p) {
   *p = *p + 10;
}
```

Exemple

- Traduir a MIPS les sentències visibles de `funcA` i `funcB`

```
short x[10], y, z;
void funcA(){
  int k; /* suposem que k es guarda en $t0 */
  ...
  z = funcB(x, y, k);
  ...
}

short funcB(short *vec, short n, int i){
  return vec[i] - n;
}
```

Variables locals

- Es creen cada vegada que s'invoca la funció
  - Valor indeterminat si no s'inicialitzen de manera explícita
  - Només son visibles dintre de la funció
  - Es guarden en registres temporals o a la pila

```
int funcA( int x, int y) {
    int a, b = 0;              // a registres temporals $t0 - $t9
    short vecs[8];             // a la pila
    …
}
```

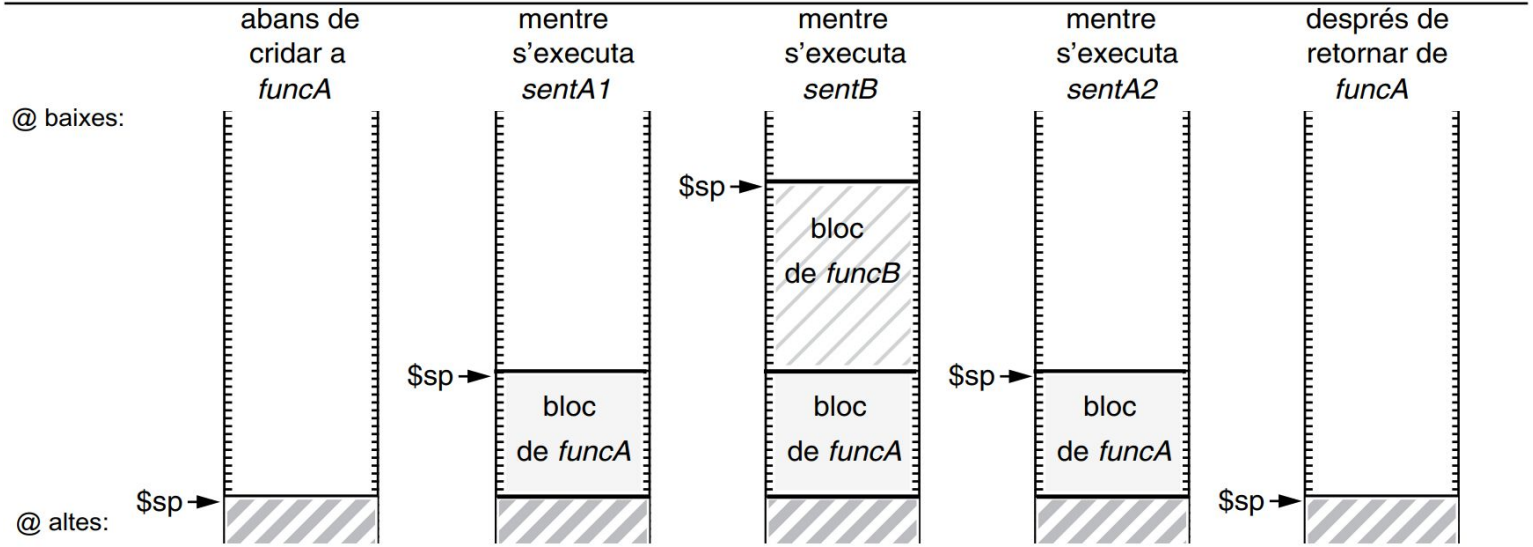
La pila i els blocs d'activació

- Algunes funcions requereixen guardar variables locals a memòria
- Aquestes variables locals s'emmagatzemen a la pila
  - La pila està inicialment situada a la direcció `0x7FFFFFFC`
  - Creix des de una adreça alta cap a adreces més baixes
  - El registre `$sp` (Stack Pointer) apunta al cim de la pila
- Cada funció manté el seu bloc d'activació a la pila
  - Al inici decrementa `$sp` per reservar memòria
  - Al final incrementa `$sp` per alliberar memòria
  - `$sp` sempre ha de ser múltiple de 4
- La pila pot tenir més d'un bloc d'activació

La pila i els blocs d'activació

```
funcA()
{       sentA1;
        funcB();
        sentA2;
}
```

```
funcB()
{
        sentB;
}
```



Regles de l'ABI per variables locals

- Les variables escalars es guarden en els registres `$t0 - $t9`, `$s0 - $s7` o `$v0 - $v1`
  - Excepte si en el cos de la funció apareix una variable local precedida del operador unari `&`
  - Si no hi ha suficients registres, les que no càpiguen es guarden a la pila
- Les variables estructurades (vectors, matrius, …) es guarden a la pila

Regles de l'ABI per variables locals

- El bloc d'activació ha de respectar les següents normes
  - Les variables locals es col·loquen a la pila seguint l'ordre en què apareixen declarades al codi, començant per l'adreça més baixa (cim de la pila)
  - S'han de respectar les normes d'alineament (com amb les variables globals)
  - L'adreça inicial i el tamany del bloc d'activació han de ser múltiples de 4

Exemple

- Tradueix la següent subrutina a MIPS i dibuixa el bloc d'activació

```
char func(int i) {
     char v[10];
     int w[10], k;          // guardem k en $t0
     …                      // inicialitzacions
     return v[w[i]+k];
}
```

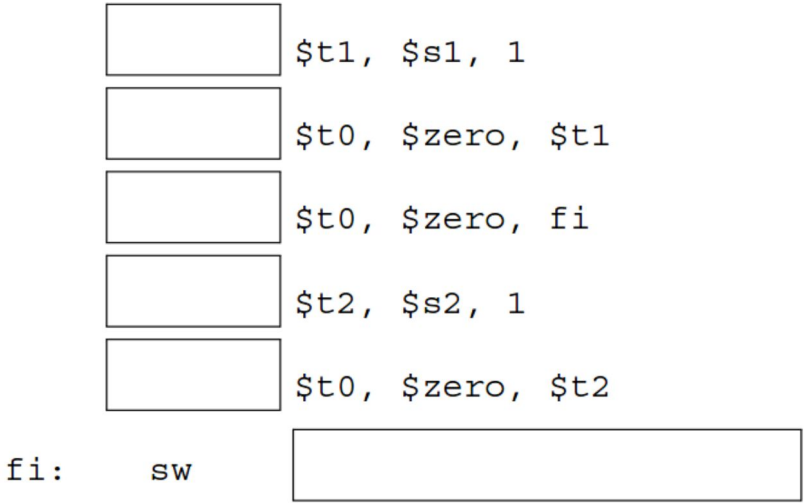
Per fer a casa....

Per fer a casa...

- Els farem a la sessió de teoria del Dijous 7 de Març

Exercici 1

- Una funció té les següents variables locals guardades a `$s0, $s1 i $s2`
  - `int *a, b, c;`
- Tradueix la següent sentencia de C a assemblador MIPS
  - `*(a+1) = b & 1 || c >> 1;`



Exercici 2

```
func:           ble     $a0, $a2, etiq1
                ble     $a1, $a2, etiq2
etiq1:          beq     $a2, $zero, etiq3
etiq2:          li      $v0, 0
                b       etiq4
etiq3:          li      $v0,1
etiq4:
                jr      $ra
```

```
int func(int x, int y, int z)
{
        int res;

        if ( ((          )&&(          )) || (          ) )

        {
                res = 0;
        } else
        {
                res = 1;
        }
        return res;
}
```

Exercici 3 - a,b,c,d a $t0, $t1, $t2, $t3

```
unsigned int a, b, c;
int d;

if (((a<=b) && (a>c)) || (d!=0))
        d = -1;
else
        d = 0;
```

```
et1:    [    ]  $t0, $t1, [    ]
et2:    [    ]  $t0, $t2, [    ]
et3:    [    ]  $t3, $zero, [    ]
et4:    li      $t3, -1
et5:    b       et7
et6:    move    $t3, $zero
et7:
```

Exercici 4

- Escriu el valor final en hexadecimal del registre `$t0`

```
li   $t0, 0x0020A040
sw   $t0, 0($sp)
lb   $t0, 1($sp)
srl  $t0, $t0, 4
ori  $t0, $t0, 0x0099
```

Exercici 5

- Escriu el valor final en hexadecimal del registre `$t0`

```
addiu    $t0, $zero, -1
sltu     $t0, $zero, $t0
addiu    $t0, $t0, -1
```

Exercici 6

- Donades les següents declaracions en C

```
char a;
int b;
long long int c;
main() {
  char *p;           /* punter guardat en $t0 */
  int *q;            /* punter guardat en $t1 */
  long long int *h; /* punter guardat en $t2 */
}
```

- Tradueix a MIPS les següents sentències en C que formen part de la funció `main`:

1. `q = q + 1;`
2. `a = *p;`
3. `h = &c;`
4. `b = *(q + b);`
5. `p[*q + 10] = a;`
6. `h = &h[*p];`

Exercici 7

- Tradueix la següent subrutina a assemblador MIPS

```
int fib(int n) {
    int tmp;
    if (n<2)
        tmp = n;
    else
        tmp = fib(n-1) + fib(n-2);
    return tmp;
}
```

Exercici 8

- Quins elements de `subr1` s'han de guardar en registres de tipus segur `$s`
- Dibuixa el bloc d'activacio de `subr1` i tradueix-la a MIPS

```
void subr2(short *x, int *y, char *z);
void subr1(int a[], int b) {
  char k;
  short v[7];
  int i;
        for (i = 0; i < b; i++)
              subr2(v, &a[3], &k);
}
```

Exercici 9

- Tradueix la següent subrutina a assemblador MIPS

```
short s3(unsigned long long *p1) {
    short v1;
    if (*p1 != 0)
        v1 = 1 + s3(p1+1);
    else
        v1 = 0;
    return v1;
}
```