# Estructura de Computadores

# Tema 3. Traducción de Programas

# Sentencia *while*

- Bucle while en C

```
while (condicion)
    sentencia_cuerpo_while
```

# Sentencia *while*

- Bucle while en C

```
while (condicion)
    sentencia_cuerpo_while
```

- Bucle while en MIPS

```
while:
    evaluar condición
    salta si es falsa a fiwhile
    traducción de sentencia_cuerpo_while
    salta a while
fiwhile:
```

# Sentencia *while*

- Traduce a MIPS el código en C, suponiendo que `dd`, `dr` y `q` son enteros almacenados en $t1, $t2 y $t3:

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

# Sentencia *while*

- Traduce a MIPS el código en C, suponiendo que `dd`, `dr` y `q` son enteros almacenados en $t1, $t2 y $t3:

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

```
        move $t3, $zero
while:
        blt $t1, $t2, fiwhile
        subu $t1, $t1, $t2
        addiu $t3, $t3, 1
        b while
fiwhile:
```

# Sentencia *while* - Evaluar la condición al final del bucle

- Traduce a MIPS el código en C, suponiendo que `dd`, `dr` y `q` son enteros almacenados en $t1, $t2 y $t3:

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

# Sentencia *while* - Evaluar la condición al final del bucle

- Traduce a MIPS el código en C, suponiendo que `dd`, `dr` y `q` son enteros almacenados en $t1, $t2 y $t3:

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

```
        move $t3, $zero
        blt $t1, $t2, fiwhile
while:
        subu $t1, $t1, $t2
        addiu $t3, $t3, 1
        bge $t1, $t2, while
fiwhile:
```

# Sentencia *for*

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

- Es equivalente a un while:

```
s1;
while (condicion) {
    s3;
    s2;
}
```

# Sentencia *do-while*

- Ejecuta una o más iteraciones mientras se cumpla la condición (la primera iteración siempre se ejecuta)

```
do
    sentencia_cuerpo_do
while (condicion);
```

# Sentencia *do-while*

- Ejecuta una o más iteraciones mientras se cumpla la condición (la primera iteración siempre se ejecuta)

```
do
   sentencia_cuerpo_do
while (condicion);
```

- Bucle *do-while* en MIPS

do:

traducción de sentencia_cuerpo_do

evaluar condición

salta si es cierta a do

# Subrutinas

```
void main() {
   int x, y, z, m;
   ...
   if (x > y) m = x;
   else       m = y;
   ...
   if (y > z) m = y;
   else       m = z;
   ...
}
```

# Subrutinas

```
void main() {
  int x, y, z, m;
  ...
  if (x > y) m = x;
  else       m = y;
  ...
  if (y > z) m = y;
  else       m = z;
  ...
}
```

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

void main() {
  int x, y, z, m;
  ...
  m = max(x, y);
  ...
  m = max(y, z);
  ...
}
```

# Subrutinas

- Programación modular
- Reutilización de código
- ¿Traducción a MIPS?
  - Llamada y retorno
  - Paso de parámetros
  - Devolución del resultado
  - Variables locales
  - Respetar reglas (ABI) para garantizar interoperabilidad

# Llamada a subrutina y retorno

- ¿Podemos utilizar la macro `b`?

```
max:
   ...
   b ret

main:
   ...
   b max
ret:
```

# Llamada a subrutina y retorno

- ¿Podemos utilizar la macro b?

```
max:
   ...
   b ret

main:
   ...
   b max
ret:
```

```
max:
   ...
   b ret1 o ret2?

main:
   ...
   b max
ret1:
   ...
   b max
ret2:
```

# Llamada a subrutina y retorno

- Hay que guardar la dirección de retorno en un registro
  - `jal` guarda la dirección de retorno y salta a una etiqueta
  - La dirección de retorno se guarda en `$ra`
  - `jr` permite saltar a la dirección guardada en un registro

# Llamada a subrutina y retorno

- Hay que guardar la dirección de retorno en un registro
  - `jal` guarda la dirección de retorno y salta a una etiqueta
  - La dirección de retorno se guarda en `$ra`
  - `jr` permite saltar a la dirección guardada en un registro

```
max:
   ...
   jr $ra

main:
   ...
   jal max
   ...
   jal max
```

# Parámetros y resultados

- Los parámetros se pasan en los registros `$a0-$a3`
- El resultado se pasa en el registro `$v0`

# Parámetros y resultados

- Los parámetros se pasan en los registros $a0-$a3
- El resultado se pasa en el registro $v0

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

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

# Parámetros y resultados

- Los parámetros se pasan en los registros $a0-$a3
- El resultado se pasa en el registro $v0

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

```
main:
  move $a0, $t0
  move $a1, $t1
  jal suma2
  move $t2, $v0
```

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

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

# Parámetros y resultados

- Los parámetros de menos de 32 bits (`char` o `short`) se extienden a 32 bits
  - Extensión de ceros (unsigned) o de signo (signed)

# Parámetros y resultados

- Los parámetros de menos de 32 bits (`char` o `short`) se extienden a 32 bits
  - Extensión de ceros (unsigned) o de signo (signed)
- Vectores: se pasa la dirección 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
```

# Parámetros y resultados

- Por valor vs por referencia
- C solo admite paso de parámetros por valor
- Se puede pasar por valor un puntero

```
int a, b;

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

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

# Ejemplo

- Traducir a MIPS las sentencias visibles de `funcA` y `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 locales

- Se crean cada vez que se invoca la función
  - Valor indeterminado si no se inicializan de forma explícita
  - Solo son visibles dentro de la función
  - Se guardan en registros temporales o en la pila

```
int funcA(int x, int y) {
  int a, b = 0;
  short vecs[8];
  ...
}
```

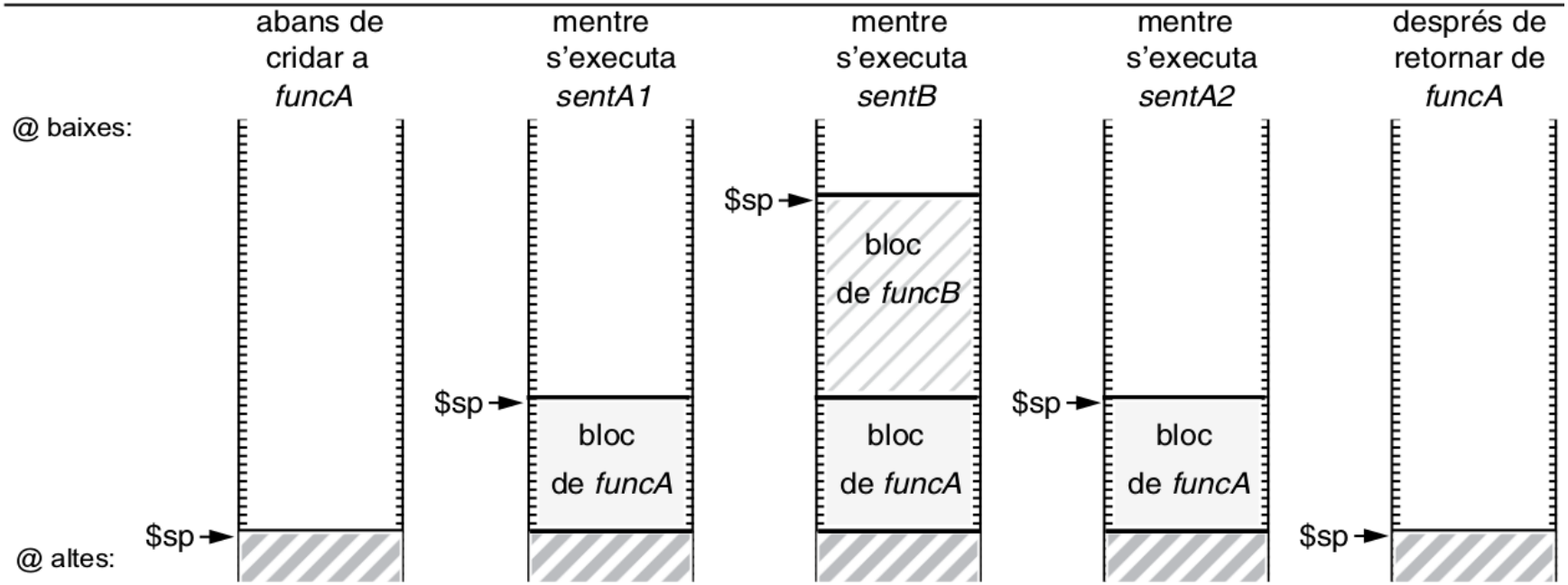
# Bloque de activación y la pila

- Algunas funciones requieren guardar variables locales en memoria
- Estas variables locales se almacenan en la pila del programa
  - La pila está inicialmente situada en la dirección 0x7FFFFFFC
  - Crece desde una dirección alta hacia direcciones más bajas
  - El registro `$sp` (Stack Pointer) apunta a la cabeza de la pila
- Cada función mantiene su bloque de activación en la pila
  - Al inicio se decrementa `$sp` para reservar memoria en la pila
  - Al final se incrementa `$sp` para liberar memoria en la pila
  - `$sp` siempre debe ser múltiplo de 4

# Bloque de activación y la pila

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

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



# Reglas de la ABI para variables locales

- Las variables escalares se guardan en los registros `$t0-$t9`, `$s0-$s7` o `$v0-$v1`
  - Excepto si en el cuerpo de la función aparece la variable local precedida del operador unario `&`
  - Si no hay suficientes registros, las que no caben se guardan en la pila
- Las variables estructuradas (vectores, matrices...) se guardan en la pila

# Reglas de la ABI para variables locales

- El bloque de activación ha de respetar las siguientes normas:
  - Las variables locales se colocan en la pila siguiendo el orden en que aparecen en la declaración, empezando desde la cabeza de la pila
  - Se han de respetar las normas de alineación (padding)
  - La dirección inicial y el tamaño del bloque de activación han de ser múltiplos de 4

# Ejemplo

- Traduce la siguiente subrutina a MIPS y dibuja el bloque de activación

```
char func(int i) {
  char v[10];
  int w[10], k;
  ...
  return v[w[i]+k];
}
```