ESTRUCTURA DE COMPUTADORES

Tema 4: Programación en ensamblador de MIPS

GRADO EN INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DE COMPUTADORES

Luis Rincón

Los computadores ejecutan programas escritos en lenguaje máquina. Como el código máquina es poco inteligible para los programadores, es más sencillo estudiar el lenguaje ensamblador, que es un lenguaje simbólico que facilita a los programadores la comprensión y la escritura de los programas en bajo nivel.

Este tema presenta los aspectos prácticos de la programación en ensamblador, centrándose en la familia de procesadores MIPS en su versión de 32 bits. No se perderá de vista la relación entre el lenguaje ensamblador y los lenguajes de alto nivel, en concreto el lenguaje C. Así, resultará más sencillo diseñar algoritmos en C para después traducirlos a ensamblador, con objeto de facilitar la aproximación a éste último lenguaje.

Índice

Universidad Rey Juan Carlos

Comenzaremos estudiando cómo se utilizan los números enteros con y sin signo. Tras ello, analizaremos cómo se pueden implementar en ensamblador de MIPS algunas de las sentencias de control propias de C. Continuaremos con ejemplos que ilustrarán el tratamiento en ensamblador de datos y estructuras de datos de diferentes tipos. Se presentará el concepto de subrutina, y se estudiará cómo se escriben subrutinas en ensamblador. Finalmente se hablará del concepto de optimización de código, y se presentarán diferentes técnicas y ejemplos de optimizaciones.

Bibliografía

[PAT] D.A. PATTERSON, J.L. HENNESSY. *Computer Organization and Design*, 5th ed. Morgan Kaufmann, 2014 (o la traducción española más reciente: *Estructura y Diseño de Computadores*, 4ª ed. Reverté, 2011).

[SWE] D. SWEETMAN. *See MIPS Run.* Morgan Kaufmann, 2002.

[PAR] B. PARHAMI. *Arquitectura de Computadores.* McGraw-Hill, 2007.

[CAR] F. GARCÍA CARBALLEIRA y otros. *Problemas resueltos de Estructura de Computadores*. Paraninfo, 2009.

[BER] J.A. ÁLVAREZ BERMEJO. *Estructura de Computadores: procesadores MIPS y su ensamblador*. Ra-Ma, 2008.

[MIPS32-I] *MIPS32 Architecture For Programmers – Volume I: Introduction to the MIPS32 Architecture*. MIPS Technologies Inc., 2003.

[MIPS32-II] *MIPS32 Architecture For Programmers – Volume II: The MIPS32 Instruction Set*. MIPS Technologies Inc., 2003.

[MIPS32-III] *MIPS32 Architecture For Programmers – Volume III: The MIPS32 Privileged Resource Architecture*. MIPS Technologies Inc., 2003.

Universidad Rey Juan Carlos

La bibliografía de este tema es la misma que la del tema anterior.

Se recomienda leer el capitulo 2 y el apéndice A de [PAT], que es el texto base seleccionado y está muy bien explicado.

[PAR] es un texto (con traducción latinoamericana) que tiene una estructura parecida a [PAT] en cuanto a los conceptos que trata, aunque la distribución de temas es diferente. En [PAR] se trata el lenguaje ensamblador de MIPS en la parte 2, que incluye los capítulos del 5 al 7.

El texto [BER] puede ser adecuado, ya que está íntegramente dedicado a la programación en ensamblador de MIPS.

[CAR] puede servir como referencia rápida, ya que en los capítulos 3 y sobre todo el 4 incluye un resumen sobre MIPS y su ensamblador, junto con algunos ejercicios resueltos. El capítulo 9 ofrece una guía de referencia al ensamblador de MIPS32.

[SWE] es un texto con un elevado nivel de complejidad, y debe ser consultado sólo cuando se quiera obtener información muy avanzada sobre MIPS.

Las tres últimas referencias corresponden con sendos manuales oficiales de MIPS. El primero de ellos es una introducción general a la arquitectura; el segundo es una referencia muy amplia del repertorio de instrucciones completo de MIPS32, que deberá ser consultada sólo puntualmente para conocer detalles acerca de instrucciones concretas; y el tercero, incluido aquí para mantener la referencia completa a los tres manuales oficiales, describe la arquitectura de recursos privilegiados implementada en MIPS. Existen otros tres manuales similares que describen la arquitectura MIPS de 64 bits.

Índice

Universidad Rey Juan Carlos

Anteriormente se presentaron diversos ejemplos con pequeños programas en ensamblador de MIPS que utilizaban datos de tipo entero. Ahora analizaremos ejemplos concretos que detallan las operaciones básicas que pueden realizarse con enteros con signo (en complemento a 2) y sin signo (en binario puro).

En la familia MIPS32 los datos de tipo entero tienen un ancho de una palabra, es decir, 32 bits.

MIPS y los enteros con signo

Creación de espacio en memoria / carga y almacenamiento / constantes / sumas y restas

C: `int A;` → Ensamblador:

```
A:   .space 4
```

C: `int A=5;` → Ensamblador:

```
A:   .word 5
```

Carga de una variable en un registro → `lw $s0,A`

`A = 65536;` →

```
li      $s0,65536
sw      $s0,A
```

```
A = B + C;
    lw      $s0,B
    lw      $s1,C
    add     $s2,$s0,$s1
    sw      $s2,A

A = B + 2;
    lw      $s0,B
    addi    $s1,$s0,2
    sw      $s1,A
```

```
A = B - C;
    lw      $s0,B
    lw      $s1,C
    sub     $s2,$s0,$s1
    sw      $s2,A

A = B - 2;
    lw      $s0,B
    addi    $s1,$s0,-2
    sw      $s1,A
```

Universidad Rey Juan Carlos

Los enteros con signo se representan en MIPS32 como datos de 32 bits en complemento a 2. Como un dato de tipo entero ocupa una palabra, se puede reservar espacio para el mismo con **.space 4** , y si queremos darle un valor inicial, lo haremos con la directiva **.word** .

Si en mitad del programa queremos copiar una constante en una variable de tipo entero, podremos utilizar la pseudoinstrucción **li** para cargarla previamente en un registro, y después la almacenaríamos en la variable de memoria mediante **sw**.

Para sumar o restar variable de tipo entero con signo, usaremos **add** y **sub**, que generan una excepción si se produce desbordamiento en la operación. Si no queremos que se genere tal excepción, podríamos usar **addu** o **subu**, que proporcionan el mismo resultado binario que **add** y **sub** pero no chequean un posible desbordamiento.

Si queremos sumar o restar una variable con una constante, usaremos **addi**, que admite una constante positiva o negativa. También valdría **addiu**. La diferencia entre ambas es la misma que entre **add** y **addu** o **sub** y **subu**: la generación o no de una posible excepción por desbordamiento. En los dos casos, las constantes deben ser representables con 16 bits. Si no es el caso, habría que cargar previamente la constante en un registro con **li**, y después hacer la suma o la resta entre registros, según proceda.

MIPS y los enteros con signo

Cambio de signo / valor absoluto / producto / división y cálculo del resto

```
A = -B;  →   lw    $s0,B
             neg   $s1,$s0
             sw    $s1,A
```

```
A = abs(B); →  lw    $s0,B
               abs   $s1,$s0
               sw    $s1,A
```

```
A = B * C;
     lw      $s0,B
     lw      $s1,C
     mul     $s2,$s0,$s1
     sw      $s2,A

A = B / C;
     lw      $s0,B
     lw      $s1,C
     div     $s2,$s0,$s1
     sw      $s2,A

A = B % C;
     lw      $s0,B
     lw      $s1,C
     rem     $s2,$s0,$s1
     sw      $s2,A
```

```
A = B * C;
     lw      $s0,B
     lw      $s1,C
     mult    $s0,$s1
     mflo    $s2
     sw      $s2,A
A = B / C;
     lw      $s0,B
     lw      $s1,C
     div     $s0,$s1
     mflo    $s2
     sw      $s2,A
A = B % C;
     lw      $s0,B
     lw      $s1,C
     div     $s0,$s1
     mfhi    $s2
     sw      $s2,A
```

Universidad Rey Juan Carlos

Existen pseudoinstrucciones para obtener el valor absoluto de un dato (**abs**) y para cambiar el signo de un número (**neg** si queremos que se genere una excepción si hay desbordamiento, y **negu** si no queremos que se genere).

El producto de números con signo se realiza mediante **mult**. Esta instrucción toma dos operandos de entrada residentes en registros de propósito general, los multiplica y guarda el resultado (que tendrá un ancho de 64 bits) en la pareja de registros **Hi-Lo**. Si luego queremos copiar este resultado en un registro de propósito general, habrá que confiar que el registro **Hi** no tiene un contenido significativo y que el resultado ha cabido completamente en **Lo**. Para copiar el contenido de **Lo** en un registro de propósito general, usaremos **mflo** (*move from Lo*).

En las últimas versiones de MIPS32 se incorporó la instrucción **mul** con tres operandos en registros de propósito general. El primero de ellos es el registro destino, como es usual, y en él se guarda el resultado del producto de los otros dos. Tras la operación, los registros **Hi-Lo** quedan con un contenido impredecible. No se genera excepción por desbordamiento con esta instrucción.

La división con signo se realiza mediante la instrucción **div** con dos operandos. Esta instrucción pone en **Lo** el cociente de la división, y en **Hi** el resto. Después de realizar la división, podemos recuperar el cociente con **mflo** y el resto con **mfhi** (*move from Hi*). No se genera excepción por desbordamiento. Si el divisor es 0, el resultado de la operación es impredecible.

Si usamos **div** con tres operandos, estaremos utilizando una pseudoinstrucción que pone el cociente en el registro destino, que aparece en primer lugar. Aquí sí se genera excepción por desbordamiento. La pseudoinstrucción **rem** pone el resto de la división en el registro destino.

MIPS y los enteros con signo

Evaluación de expresiones aritméticas

```
w=(x*y+z*z)/(x-y);
```

```
lw    $s0,x         # $s0 ← x
lw    $s1,y         # $s1 ← y
mul   $t0,$s0,$s1   # $t0 ← x*y
lw    $s2,z         # $s2 ← z
mul   $t1,$s2,$s2   # $t1 ← z*z
add   $t0,$t0,$t1   # $t0 ← x*y+z*z
sub   $t1,$s0,$s1   # $t1 ← x-y
div   $s3,$t0,$t1   # $s3 ← (x*y+z*z) / (x-y)
sw    $s3,w         # w ← $s3
```

Las expresiones aritméticas se pueden evaluar mediante el modelo de notación polaca inversa con la ayuda de una pila. Pero como en MIPS hay muchos registros, se sustituirá la pila total o parcialmente por registros de propósito general. Por convenio, los más adecuados para ello son los registros temporales **$t0...$t9**, que se suelen usar para mantener los datos intermedios generados precisamente en la evaluación de expresiones. También por convenio, las variables de memoria se cargan en registros seguros (*saved*) **$s0...$s7**, pensados para mantener copia de las variables de larga duración. No debe olvidarse de que el resultado final de la evaluación de la expresión debe ser grabado en la variable de memoria mediante **sw**.

MIPS y los enteros sin signo

Creación de espacio en memoria / carga y almacenamiento / constantes / sumas y restas

C: `unsigned int A;` → Ensamblador:

```
A:   .space   4
```

Carga de una variable en un registro → `lw $s0,A`

`A = 65536;` →

```
li      $s0,65536
sw      $s0,A
```

```
A = B + C;
    lw      $s0,B
    lw      $s1,C
    addu    $s2,$s0,$s1
    sw      $s2,A
```

```
A = B - C;
    lw      $s0,B
    lw      $s1,C
    subu    $s2,$s0,$s1
    sw      $s2,A
```

```
A = B + 2;
    lw      $s0,B
    addiu   $s1,$s0,2
    sw      $s1,A
```

```
A = B - 2;
    lw      $s0,B
    addiu   $s1,$s0,-2
    sw      $s1,A
```

Universidad Rey Juan Carlos

Los enteros sin signo se representan en MIPS32 como datos de 32 bits en binario puro. Como un dato de tipo entero ocupa una palabra, se puede reservar espacio para el mismo con **.space 4** , y si queremos darle un valor inicial, lo haremos con la directiva **.word** .

Si en mitad del programa queremos copiar una constante en una variable de tipo entero, podremos utilizar la pseudoinstrucción **li** para cargarla previamente en un registro, y después la almacenaríamos en la variable de memoria mediante **sw**.

Para sumar o restar variable de tipo entero sin signo, usaremos **addu** y **subu**. Si queremos sumar o restar una variable con una constante, usaremos **addiu**, que admite una constante positiva o negativa que debe ser representable con 16 bits. Si no es el caso, habría que cargar previamente la constante en un registro con **li**, y después hacer la suma o la resta entre registros, según proceda.

Muy a menudo la aritmética de los enteros sin signo se utiliza para calcular direcciones. Los resultados de estas operaciones se guardarían en registros que actuarían como punteros a datos o instrucciones.

MIPS y los enteros sin signo

Cambio de signo / valor absoluto / producto / división y cálculo del resto

```
A = B * C;
     lw         $s0,B
     lw         $s1,C
     mulou      $s2,$s0,$s1
     sw         $s2,A
```

```
A = B * C;
     lw         $s0,B
     lw         $s1,C
     multu      $s0,$s1
     mflo       $s2
     sw         $s2,A
```

```
A = B / C;
     lw         $s0,B
     lw         $s1,C
     divu       $s2,$s0,$s1
     sw         $s2,A
```

```
A = B / C;
     lw         $s0,B
     lw         $s1,C
     divu       $s0,$s1
     mflo       $s2
     sw         $s2,A
```

```
A = B % C;
     lw         $s0,B
     lw         $s1,C
     remu       $s2,$s0,$s1
     sw         $s2,A
```

```
A = B % C;
     lw         $s0,B
     lw         $s1,C
     divu       $s0,$s1
     mfhi       $s2
     sw         $s2,A
```

Universidad Rey Juan Carlos

El producto de números sin signo se realiza mediante **mult**. Esta instrucción toma dos operandos de entrada residentes en registros de propósito general, los multiplica y guarda el resultado (que tendrá un ancho de 64 bits) en la pareja de registros **Hi-Lo**. Si luego queremos copiar este resultado en un registro de propósito general, habrá que confiar que el registro **Hi** no tiene un contenido significativo y que el resultado ha cabido completamente en **Lo**. Para copiar el contenido de **Lo** en un registro de propósito general, usaremos **mflo** (*move from Lo*).

La división sin signo se realiza mediante la instrucción **divu** con dos operandos. Esta instrucción pone en **Lo** el cociente de la división, y en **Hi** el resto. Después de realizar la división, podemos recuperar el cociente con **mflo** y el resto con **mfhi** (*move from Hi*). No se genera excepción por desbordamiento. Si el divisor es 0, el resultado de la operación es impredecible.

Si usamos **divu** con tres operandos, estaremos utilizando una pseudoinstrucción que pone el cociente en el registro destino, que aparece en primer lugar. Aquí sí se genera excepción por desbordamiento. La pseudoinstrucción **remu** pone el resto de la división en el registro destino.

Índice

Universidad Rey Juan Carlos

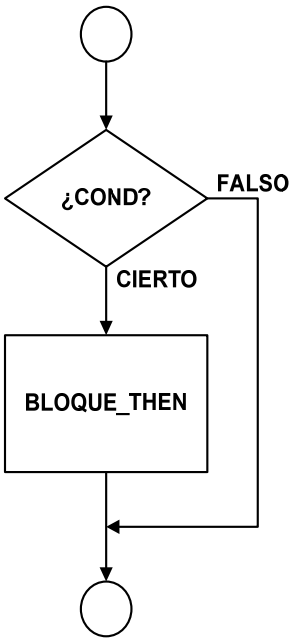
Las estructuras de control son necesarias en los lenguajes de programación de alto nivel para poder realizar algoritmos en los que se puedan tomar decisiones y ejecutar bucles. Para soportar estas opciones, el lenguaje ensamblador cuenta con las instrucciones de ramificación y salto, que permiten romper la secuencia de ejecución de instrucciones, y que ya han sido estudiadas con anterioridad.

Ahora presentaremos algunas de las estructuras de control más habituales del lenguaje C, y una traducción de las mismas a lenguaje ensamblador de MIPS. Es preciso tener en cuenta que las traducciones aquí presentadas no son únicas, es decir, existen implementaciones diferentes e igualmente válidas de estas estructuras, algunas incluso más optimizadas.

Se analizarán las siguientes sentencias de control:

- Selección **if**.
- Selección **if-else**.
- Bucle **do-while**.
- Bucle **while**.
- Bucle **for**.

MIPS y las sentencias de control: IF



C (variables enteras):

```
if (x >= y) {
   x = x+2;
   y = y-2;
};
```

Ensamblador MIPS:

```
# if (x >= y)
if:
        lw      $s0,x
        lw      $s1,y
        blt     $s0,$s1,end
# bloque then
then:
# x = x+2;
        lw      $s0,x
        addi    $s0,$s0,2
        sw      $s0,x
# y = y-2;
        lw      $s1,y
        addi    $s1,$s1,-2
        sw      $s1,y
end:
```

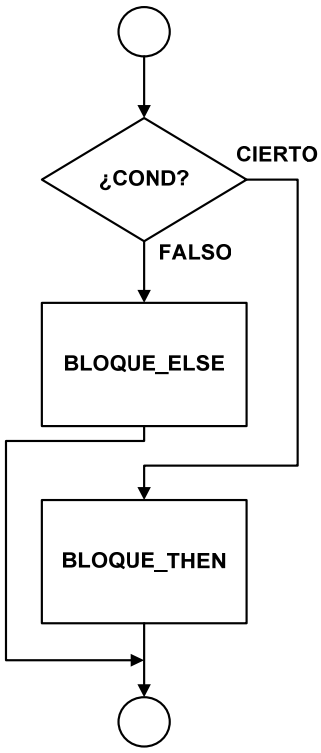
La estructura **if** es la más sencilla de todas, y su implementación en ensamblador es muy simple. Basta con comprobar si una condición es cierta o no, y en caso de serlo, ejecutar un bloque de sentencias (**then**). Por tanto, es necesario evaluar una condición y saltar mediante una operación de bifurcación condicional. El problema es que este tipo de operaciones saltan cuando la condición es cierta, y lo que hay que hacer en realidad es saltarse el bloque de sentencias **then** cuando la condición es falsa.

Existen diferentes opciones de implementación de esta estructura. Lo más sencillo es evaluar la condición contraria a la expresada en alto nivel, y cuando la contraria es cierta, entonces saltarse el bloque **then**. Por tanto, el bloque **then** se ejecutará cuando la condición contraria sea falsa, o sea, cuando la condición directa de la estructura de alto nivel sea cierta.

Así, en el ejemplo se ejecutará el bloque **then** cuando **x>=y**, y por tanto la operación de bifurcación condicional deberá saltar al final de la estructura de control (etiqueta **end**) cuando **x<y**: por eso se utiliza la pseudoinstrucción **blt**.

Como MIPS es una máquina de carga-almacenamiento, cada vez que se utiliza una variable de memoria se ejecuta una operación de carga **lw** sobre un registro, y cada vez que se modifica el valor de una variable, se realiza un almacenamiento con **sw**.

MIPS y las sentencias de control: IF - ELSE



C (variables enteras):

```
if (x >= y) {
   x = x+2;
}
else {
   x = x-2;
}
```

Ensamblador MIPS:

```
# if (x>=y)
if:
        lw      $s0,x
        lw      $s1,y
        bge     $s0,$s1,then
# bloque else
else:
# x = x-2;
        lw      $s0,x
        addi    $s0,$s0,-2
        sw      $s0,x
# bloque then
        b       end
then:
# x = x+2;
        lw      $s0,x
        addi    $s0,$s0,2
        sw      $s0,x
end:
```

Universidad Rey Juan Carlos

La estructura **if-else** es algo más compleja, ya que ahora elegiremos entre ejecutar dos bloques de sentencias: el bloque **then** cuando la condición es cierta, y el bloque **else** cuando es falsa.

Si en ensamblador queremos preguntar la misma condición que en alto nivel, entonces lo óptimo es colocar el bloque **else** antes del **then**. Por supuesto, tendremos que separar ambos con una operación de salto incondicional al final de la estructura (etiqueta **end**), para no ejecutar el bloque **then** después del bloque **else**.

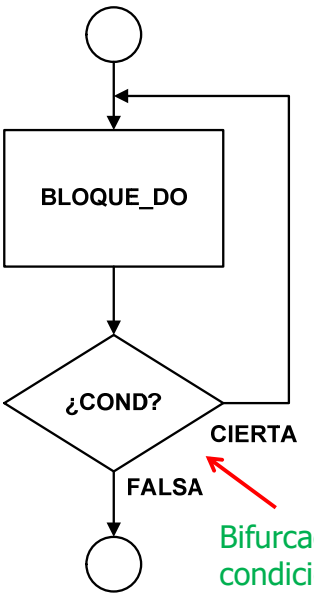
En C y otros lenguajes existe la estructura de selección **switch**, que permite seleccionar múltiples alternativas dependiendo de un único valor. Esta estructura puede implementarse mediante una secuencia de estructuras **if-else**, lo cual no suele ser lo más óptimo, pero sí lo más sencillo conceptualmente.

Para implementar una estructura **switch** a menudo resulta eficiente contar con una **tabla de direcciones de salto** que almacene las direcciones de los puntos de entrada de la secuencia de instrucciones que implementa cada una de las opciones. La primera instrucción de cada secuencia llevará una etiqueta (*valor1*, *valor2*, etc), y la tabla se rellenará con una secuencia de palabras cuyos valores se corresponden precisamente con dichas etiquetas. Para saltar a la secuencia de instrucciones adecuada, se pondrá un puntero apuntando al comienzo de la tabla, y se le sumará la distancia a la que se encuentra la entrada elegida respecto de la dirección base de la propia tabla; después se cargará en un registro **$reg** el contenido de dicha entrada mediante una operación **lw**, y después se saltará a la opción elegida mediante **jr $reg**.

MIPS y las sentencias de control: DO - WHILE

C (variables enteras):

```
a = 0;
z = 1;
do {
    a = a+z;
    z = z+1;
} while (z <> 10);
```



Bifurcación condicional

Ensamblador MIPS:

```
# a = 0;
          sw        $zero,a
# z = 1;
          li        $s1,1
          sw        $s1,z
# do {
do:
# a = a+z;
          lw        $s0,a
          lw        $s1,z
          add       $s0,$s0,$s1
          sw        $s0,a
# z = z+1;
          lw        $s1,z
          addi      $s1,$s1,1
          sw        $s1,z
# } while (z <> 10);
while:    lw        $s1,z
          li        $t0,10
          bne       $s1,$t0,do
```

Optimización: asignación de registros a variables

```
# Variable a: copia en $s0
# Variable z: copia en $s1
# a = 0;
          li        $s0,0
# z = 1;
          li        $s1,1
# do {
do:
# a = a+z;
          add       $s0,$s0,$s1
# z = z+1;
          addi      $s1,$s1,1
# } while (z <> 10);
while:    li        $t0,10
          bne       $s1,$t0,do
# Sincronizar registros
# con variables en memoria
          sw        $s0,a
          sw        $s1,z
```

Universidad Rey Juan Carlos

El bucle **do-while** es el más fácil de implementar. Siempre se ejecuta al menos una vez, y la condición para continuar en el bucle se evalúa por primera vez al terminar la primera iteración. Si la condición es cierta, se continúa iterando, para lo cual hay que realizar un salto condicional hacia atrás. Como el salto será efectivo si la condición sea cierta, se efectúa mediante una instrucción de bifurcación condicional que evalúe la misma condición que aparece en el programa en alto nivel.

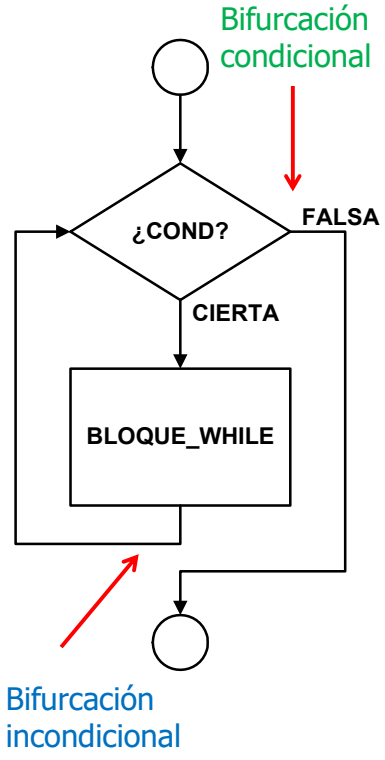
La naturaleza de carga-almacenamiento del repertorio de instrucciones de MIPS hace que el código resulte un tanto farragoso. Puede observarse que en la secuencia de instrucciones resultante (mostrada en el centro de la diapositiva) hay múltiples instrucciones **lw** y **sw** que incrementan notablemente su tiempo de ejecución.

A la derecha se presenta una versión mejorada del bucle **do-while**, en la que se ha realizado una optimización fundamental: la **asignación de registros a variables**. Su objetivo es reducir el trasiego de información entre memoria y registros, manteniendo temporalmente copia de las variables que estamos utilizando en ciertos registros seleccionados.

En este ejemplo, hay dos variables: **a** y **z**. Si mantenemos copia de la primera en **$s0** y de la segunda en **$s1**, evitamos las cargas dentro del bucle cuando las utilizamos, y los almacenamientos cuando las modificamos. Pero si escribimos sobre **$s0** y/o **$s1**, la variable en memoria y su copia temporal en registro dejan de estar sincronizadas. Todo está bien si finalmente se sincronizan ambas mediante operaciones de almacenamiento **sw**, dado que lo realmente importante es que al final las variables residentes en memoria queden con su valor esperado.

Aún queda una optimización más. En la evaluación de la condición del bucle, se carga el valor 10 en el registro **$t0** en todas las iteraciones, y este registro no se modifica de una iteración a la siguiente: esta instrucción constituye un **invariante del bucle**. Por tanto, podríamos mover la operación de carga y ponerla antes de entrar en el bucle, de forma que así se ejecutaría una única vez, y no una vez por iteración. Esta optimización se denomina **movimiento de código**.

MIPS y las sentencias de control: WHILE



C (enteros):

```
a = 0;
z = 1;
while (z <> 10) {
    a = a+z;
    z = z+1;
};
```

Ensamblador MIPS:

```
# Variable a: copia en $s0
# Variable z: copia en $s1
# a = 0;
          li        $s0,0
# z = 1;
          li        $s1,1
# while (z <> 10) {
while:    li        $t0,10
          beq       $s1,$t0,end
cuerpo_bucle:
# a = a+z;
          add       $s0,$s0,$s1
# z = z+1;
          addi      $s1,$s1,1
# };
          b         while
end:
# Almacenar a y z en memoria
          sw        $s0,a
          sw        $s1,z
```

El bucle for es como el while.

Universidad Rey Juan Carlos

El bucle **while** se ejecuta mientras se cumple una cierta condición. Como dicha condición se evalúa al principio del bucle, habrá que realizar una bifurcación condicional preguntando por la condición contraria a la de la permanencia. Como última operación del bucle es preciso incluir un salto incondicional al punto en que se evalúa la condición de permanencia en el bucle.

Este ejemplo incorpora la optimización de ubicación de variables en registros.

El bucle **for** se implementa igual que el **while**, teniendo en cuenta algunos detalles:

- Las sentencias de inicio van justo antes de comenzar el bucle, y no forman parte del cuerpo del mismo.
- La evaluación de la condición es justo donde comienza el bucle.
- Las sentencias de actualización van justo al final del cuerpo del bucle, antes del salto incondicional hacia atrás para evaluar de nuevo la condición.

Índice

Una **estructura de datos** (ED) es una agrupación de datos simples bajo una denominación única, en la que los datos simples pueden ser todos del mismo tipo o de tipos diferentes. Si todos los datos de la estructura son del mismo tipo y el acceso a los mismos se realiza mediante índices, estaremos hablando de **vectores** (unidimensionales) o **matrices** (multidimensionales). Un caso especial de vectores lo constituyen las **cadenas o tiras de caracteres**, en las que todos los elementos son de tipo carácter.

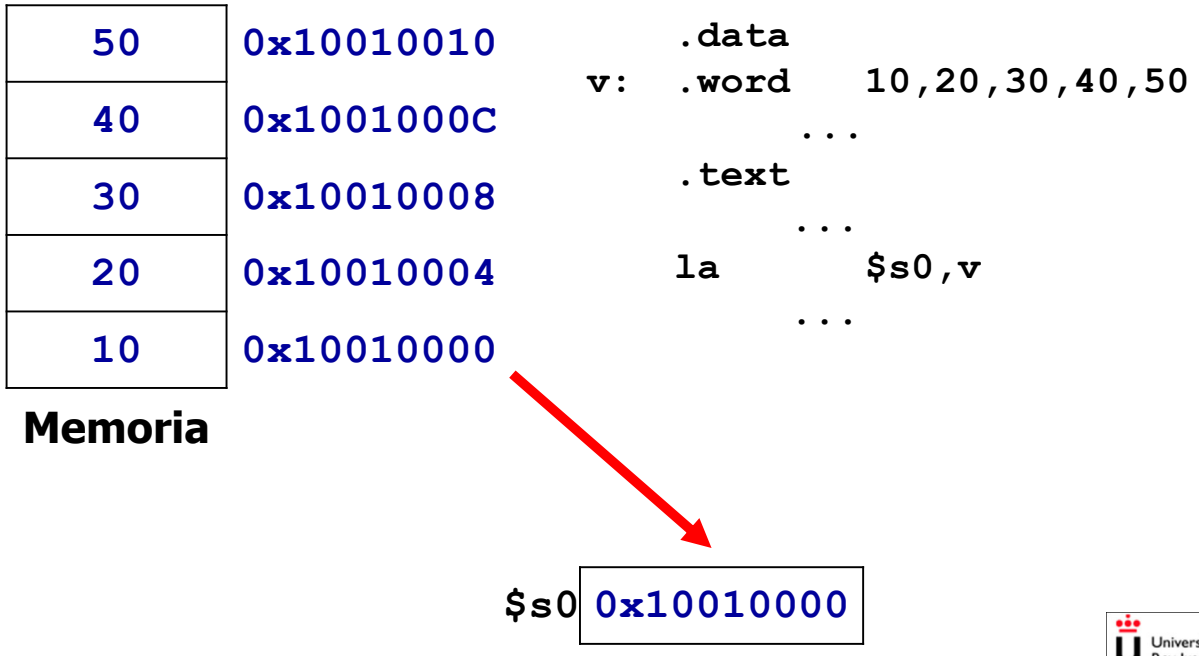
Los datos elementales que componen una ED se almacenan en memoria de forma contigua. En total, una ED ocupa muchos *bytes* seguidos en memoria.

En ensamblador de MIPS no se pueden manejar estructuras de datos completas de un golpe, sino que hay que tratar de forma individual a los datos simples que las componen. Por supuesto, una ED completa no cabe en un registro: a lo sumo, en un registro cabrá un dato individual entre todos los que forman la estructura. Entonces, para realizar el tratamiento de la ED será preciso manipular uno por uno sus campos o datos elementales, para lo cual tendremos que cargarlos uno por uno en registros.

En este apartado vamos a estudiar cómo se definen y manipulan los vectores de datos de tipo entero, y lo haremos a través de varios ejemplos.

Carga de una dirección en un registro

```
la  registro_destino,direccion_de_variable
```



Todos los datos dentro de una ED inicialmente se encuentran en memoria. Por tanto, para acceder a ellos es preciso conocer o calcular su dirección.

La dirección base de una ED es la dirección en la que se encuentra el primer dato básico de la misma. Por ejemplo, en un vector **v** de **N** elementos, su dirección base es la posición de memoria en la que se encuentra el elemento **v[0]** (si los índices van de 0 a **N-1**, que es lo que sucede en C). Si esta dirección es conocida, es fácil calcular la dirección de los demás elementos de la ED, porque sabemos a qué distancia del primer elemento están almacenados. Entonces, lo usual es cargar la dirección base de la estructura en un registro que actúa como puntero a la misma, y al que se suele llamar **registro base** o **puntero base**.

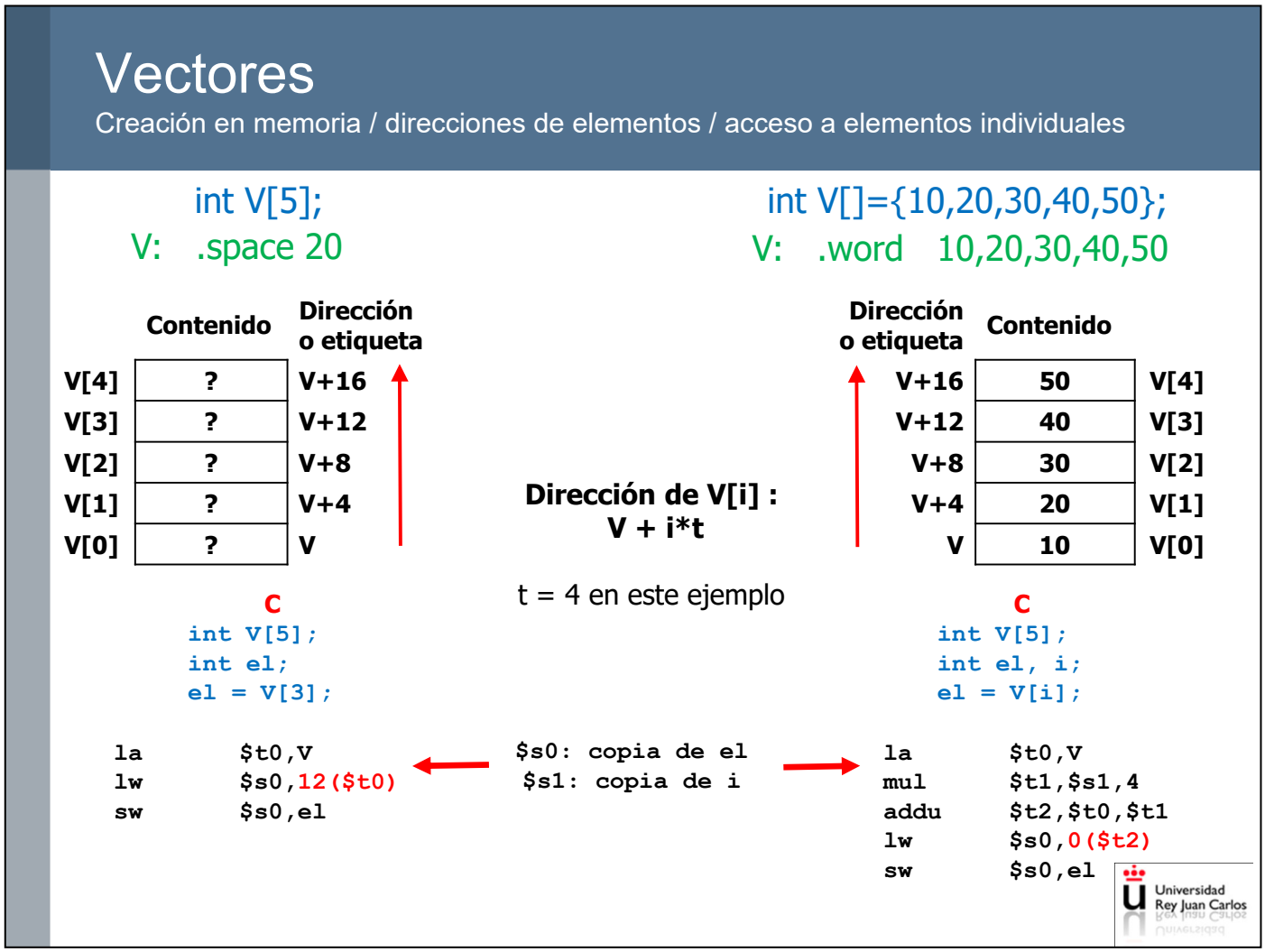
La pseudoinstrucción **la** de MIPS permite fácilmente cargar en un registro la dirección de cualquier objeto residente en memoria, ya sea una variable simple, la dirección base de una ED o la dirección de cualquiera de sus componentes. Esta pseudoinstrucción tienen dos operandos: el primero es el registro base que actúa como destino, y el segundo es la dirección del objeto, que puede ser dada mediante direccionamiento indirecto a registro con desplazamiento, o bien mediante una etiqueta.

Así, el acceso a los componentes de la ED puede realizarse de diferentes formas:

- Sumando al puntero base un desplazamiento equivalente a la distancia existente en memoria entre el componente accedido y la posición apuntada por el puntero.
- Incrementando o decrementando el puntero. Esto es típico cuando los componentes son todos del mismo tipo, como es el caso de los vectores, las matrices o las tiras de caracteres, y el proceso implica realizar un recorrido por la ED a través de un bucle.

Vectores

Creación en memoria / direcciones de elementos / acceso a elementos individuales



Los vectores son secuencias unidimensionales de longitud definida que contienen datos simples del mismo tipo y tamaño. Los elementos de un vector se encuentran almacenados consecutivamente en memoria. Se utilizará el convenio de C, que establece que, para un vector de **N** elementos, los índices de los mismos se encuentran en el rango **0..N-1**.

En general, si el vector comienza en una cierta dirección etiquetada como **V**, la dirección del elemento i-ésimo es **V+i*t**, siendo **t** el tamaño de cada elemento.

Para acceder a los elementos de un vector, lo primero es cargar en un registro la dirección base del mismo, lo cual podemos conseguir mediante la pseudoinstrucción **la**. Una vez hecho esto, podemos encontrarnos con dos casos, dependiendo de si el índice del elemento es conocido en tiempo de compilación o no lo es.

Si el acceso es a un elemento de índice constante conocido en tiempo de compilación, se calcula la distancia del elemento al comienzo del vector, multiplicando manualmente el índice por el tamaño del elemento: **i * t**. Evidentemente, el resultado de este producto es conocido en tiempo de compilación, y puede ponerse como una constante que actuará como desplazamiento que se aplicará al registro base en el direccionamiento.

Si el acceso es a un elemento de índice desconocido en tiempo de compilación, el índice viene dado por una variable o una expresión con variables. Bastaría con multiplicarlo por el tamaño del elemento, pero habría que hacerlo a base de instrucciones, en particular usando **mul**. Después, habría que sumar el resultado del producto con el registro base, y cargar el resultado en algún registro, que quedaría apuntando directamente al elemento buscado. Por tanto, para acceder a dicho elemento usaríamos direccionamiento indirecto a registro con desplazamiento, siendo éste último igual a 0.

Vectores

Recorrido de un vector

```
        .data
# Tamaño de un entero: 4
v:      .space  20
suma:   .space  4
# suma: registro $s0
# i:    registro $t0
# temporales: $t0 ... $t5
        .text
        ...
        li      $s0,0
for:    li      $t0,0
cond_for:
        li      $t1,5
        beq     $t0,$t1,end
cuerpo_for:
        la      $t2,V
        mul     $t3,$t0,4
        addu    $t4,$t2,$t3
        lw      $t5,0($t4)
        addu    $s0,$s0,$t5
inc_for:
        addiu   $t0,$t0,1
        b       cond_for
end:
        sw      $s0,suma
```

C

```
int V[5], i, suma;

suma = 0;
for (i=0; i<5; i++)
  suma = suma+V[i];
```

C

```
int V[5], *p, *f;
int suma;

suma = 0;
for (p=V, f=p+5;
     p != f; p++)
  suma = suma+*p;
```

```
        .data
# Tamaño de un entero: 4
v:      .space  20
suma:   .space  4
# suma: registro $s0
# p:    registro $t0
# f:    registro $t1
# temporales: $t0 ... $t2
        .text
        ...
        li      $s0,0
for:    la      $t0,V
        addi    $t1,$t0,20
cond_for:
        beq     $t0,$t1,end
cuerpo_for:
        lw      $t2,0($t0)
        addu    $s0,$s0,$t2
inc_for:
        addiu   $t0,$t0,4
        b       cond_for
end:
        sw      $s0,suma
```

Universidad Rey Juan Carlos

Para procesar un vector en ensamblador con un bucle podemos adoptar dos estrategias: recorrido con índice y recorrido con puntero.

En el recorrido con índice, se usa un registro para el índice (**$t0** en el ejemplo). En un recorrido hacia arriba, el índice se inicia a 0 y se incrementa al final de cada iteración. El bucle termina cuando el índice se sale del rango de índices del vector. En cada iteración, se calcula la dirección del elemento accedido cargando un puntero base al comienzo del bucle con **la**, para después multiplicar el índice por el tamaño de cada elemento y sumar el resultado del producto al puntero base. El acceso al elemento se realiza con direccionamiento indirecto a registro con desplazamiento 0. Al índice se le denomina **variable de inducción**.

En el recorrido con puntero, se pone un puntero base al comienzo del vector con **la** (en el ejemplo es **$t0**), y se pone otro puntero marcando el final del vector (**$t1** en el ejemplo) sin más que sumar el tamaño del vector completo al puntero base. Después, el puntero base se va incrementando, de modo que pasa a apuntar en cada iteración al elemento al que queremos acceder. El acceso propiamente dicho se vuelve a producir mediante direccionamiento indirecto a registro con desplazamiento 0. Para incrementar el puntero se le suma el tamaño de un elemento (en el ejemplo, el tamaño es 4). El bucle termina cuando el puntero que recorre el vector se iguala al puntero que señala el final del mismo.

El recorrido mediante punteros es más rápido, y se puede considerar como una optimización del recorrido mediante índices. Esta optimización se denomina **eliminación de la variable de inducción**, ya que al usar punteros eliminamos el índice **i**.

Existe una solución intermedia que usa índice y hace recorrido con puntero. En esta solución, controlamos el número de iteraciones con el índice, pero usamos el puntero para recorrer el vector y acceder a sus elementos. Entonces, en cada iteración se incrementan índice y puntero, y el bucle termina cuando el índice se sale del vector. Esta solución queda como ejercicio propuesto.

Índice

Universidad Rey Juan Carlos

Los datos en coma flotante requieren un tratamiento específico, y por tanto existen instrucciones específicas en el repertorio de MIPS para ello.

Vamos a estudiar cómo definir y operar con datos en el estándar IEEE 754 de precisión simple (32 bits) y doble (64 bits). En el caso de la precisión simple, las instrucciones llevarán un sufijo **.s**, mientras que para precisión doble el sufijo es **.d**.

Al igual que para los registros de propósito general, existe un convenio que indica el uso que se le da a cada uno:

- **$f0-$f1**: se usan para que las subrutinas graben en ellos el valor de retorno.
- **$f2-$f11**, **$f16-$f19**: se usan para mantener datos temporales.
- **$f12-$f15**: se emplean para introducir argumentos de las subrutinas que sean operandos en coma flotante y se encuentren entre los cuatro primeros.
- **$f20-$f31**: se utilizan para mantener copias de las variables de memoria. Son el equivalente a los registros seguros del banco de registros de propósito general, y también deben ser preservados en las llamadas a subrutina.

MIPS y la coma flotante

Precisión simple

C: float A; → Ensamblador: `A: .space 4`

C: float A=5.1; → Ensamblador: `A: .float 5.1`

Carga de una variable en un registro → `l.s $f20,A`

```
A = B + C;
     l.s      $f20,B
     l.s      $f22,C
     add.s    $f24,$f20,$f22
     s.s      $f24,A
```

```
A = B - C;
     l.s      $f20,B
     l.s      $f22,C
     sub.s    $f24,$f20,$f22
     s.s      $f24,A
```

```
A = B * C;
     l.s      $f20,B
     l.s      $f22,C
     mul.s    $f24,$f20,$f22
     s.s      $f24,A
```

```
A = B / C;
     l.s      $f20,B
     l.s      $f22,C
     div.s    $f24,$f20,$f22
     s.s      $f24,A
```

```
A = fabs(B);
l.s     $f20,B
abs.s   $f22,$f20
s.s     $f22,A
```

```
A = -B;
l.s     $f20,B
neg.s   $f22,$f20
s.s     $f22,A
```

```
A = sqrt(B);
l.s     $f20,B
sqrt.s  $f22,$f20
s.s     $f22,A
```

Universidad Rey Juan Carlos

Los datos en coma flotante de precisión simple se representan en MIPS32 según el estándar IEEE 754, y por tanto ocupan 32 bits. Entonces se puede reservar espacio para una variable de este tipo con **.space 4** , y si queremos darle un valor inicial, lo haremos con la directiva **.float**.

Para cargar una variable de 32 bits memoria sobre un registro de coma flotante, podremos usar la pseudoinstrucción **l.s**, que en realidad se traduce por la pareja de instrucciones siguientes:

```
lui $rbase,cte16          # Cargar la parte alta de la dirección en un registro base
lwc1 $fdest,desp($rbase)  # Carga en un registro el contenido de la variable
```

Por tanto, la carga de una variable en coma flotante es parecida a la carga de un entero, pero la instrucción de carga propiamente dicha es **lwc1**, que viene de ***load word to coprocessor 1***, dado que el coprocesador 1 es el que contiene el banco de registros de coma flotante.

La suma, la resta, el producto y la división se efectúan mediante **add.s**, **sub.s**, **mul.s** y **div.s** respectivamente. En este caso, el resultado siempre será un número en precisión simple, igual que los operandos fuente. La instrucción **abs.s** calcula el valor absoluto de un número, **neg.s** sirve para cambiar un dato de signo y **sqrt.s** calcula la raíz cuadrada de un dato. En todo caso, para almacenar un dato en precisión simple usaremos **s.s**, que es una pseudoinstrucción parecida a **l.s**, pero que se traduce por **lui + swc1** , siendo esta última una instrucción cuyo nemotécnico viene de ***store word from coprocessor 1***.

Para cantidades en IEEE 754 de precisión doble, reservaríamos espacio para una variable sin valor inicial con **.space 8**, y si queremos darle valor inicial usaremos **.double**. En las operaciones de carga, almacenamiento, suma, resta, etc., sustituiríamos el sufijo **.s** por **.d**. Además, tendríamos que usar operandos en registros pares, ya que un dato en precisión doble se almacena en una pareja de registros contiguos, empezando por uno par (**$f0-$f1**, **$f2-$f3**, etc). La traducción de **l.d** se realiza mediante la pareja de instrucciones **lui** + **ldc1** (***load double to coprocessor 1***), y la de **s.d** por **lui** + **sdc1** (***store double from coprocessor 1***).

MIPS y la coma flotante

Copia de registros / carga de constantes en registros de coma flotante

Copia de registros

FP a FP:	`$f4 ← f2`	`mov.s`	`$f4,$f2`
FP a GPR:	`$t0 ← f2`	`mfc1`	`$t0,$f2`
GPR a FP:	`$f2 ← t0`	`mtc1`	`$t0,$f2`

Carga de constantes en registros de coma flotante

A través de un registro de propósito general

```
$f4 ← 65.3
     li      $t0,0x4282999A
     mtc1    $t0,$f4
```

A través de una variable con valor inicial

```
$f4 ← 65.3
     .data
cte: .float  65.3
        ...
     .text
        ...
     l.s     $f4,cte
```

Universidad Rey Juan Carlos

Existen múltiples instrucciones que permiten copiar el contenido de un registro de coma flotante en otro, todas ellas con un nemotécnico que comienza por **mov**. Las más sencillas son **mov.s** , que copia un registro en otro, y **mov.d** , que copia una pareja de registros en otra. Ambas tienen dos operandos. Otras instrucciones copian condicionalmente el contenido de un registro de coma flotante en otro, dependiendo del valor de un tercer registro, o dependiendo del valor de un *flag* del coprocesador. No las estudiaremos aquí.

También es posible copiar registros entre bancos diferentes. Para copiar el contenido de un registro de coma flotante en un registro de propósito general contamos con la instrucción **mfc1** (***move from coprocessor 1***), y si lo que queremos es copiar el contenido de un registro de propósito general en un registro de coma flotante usaremos **mfc1** (***move from coprocessor 1***). En ambos casos, el operando que aparece en primer lugar es el registro de propósito general, y el que aparece en segundo lugar es el registro de coma flotante.

No hay instrucciones de máquina para datos en coma flotante que admitan operandos constantes. Entonces, para cargar un valor constante en un registro de coma flotante habrá que recurrir a otras opciones.

Si el ensamblador no tiene pseudoinstrucciones que permitan operandos en coma flotante, tendremos que averiguar la representación de la constante en IEEE 754 en notación compacta hexadecimal y cargarla en un registro de propósito general con **li**, tras lo cual si es necesario se podrá copiar en un registro de coma flotante con **mtc1**. Para datos en precisión doble, habrá que cargar las dos mitades de la constante en sendos registros de propósito general y ejecutar **mtc1** dos veces para cargar la pareja de registros de coma flotante.

Otra posibilidad es definir una variable en memoria con valor inicial mediante **.float** (para precisión simple) o **.double** (para precisión doble), lo que haría entonces disponible la constante para leerla con **l.s** o **l.d** y utilizarla en instrucciones posteriores.

MIPS y la coma flotante

Precisión simple: conversiones

```
A_real = A_entero;
     lw        $s0,A_entero
     mtc1      $s0,$f4
     cvt.s.w   $f20,$f4
     s.s       $f20,A_real
```

```
A_entero = A_real;
     l.s       $f20,A_real
     cvt.w.s   $f4,$f20
     mfc1      $s0,$f4
     sw        $s0,A_entero
```

```
A_entero = (int) trunc(A_real);
     l.s       $f20,A_real
     trunc.w.s $f4,$f20
     mfc1      $s0,$f4
     sw        $s0,A_entero
```

```
A_entero = (int) round(A_real);
     l.s       $f10,A_real
     round.w.s $f0,$f10
     mfc1      $s0,$f0
     sw        $s0,A_entero
```

```
A_entero = (int) floor(A_real);
     l.s       $f20,A_real
     floor.w.s $f4,$f20
     mfc1      $s0,$f4
     sw        $s0,A_entero
```

```
A_entero = (int) ceil(A_real);
     l.s       $f20,A_real
     ceil.w.s  $f4,$f20
     mfc1      $s0,$f4
     sw        $s0,A_entero
```

Universidad Rey Juan Carlos

MIPS permite convertir un dato entero a coma flotante mediante las instrucciones **cvt.s.w** (destino en precisión simple) y **cvt.d.w** (destino en precisión doble). Estas instrucciones llevan dos sufijos: el primero indica el formato del operando destino, y segundo indica el formato del operando fuente. El registro fuente contiene un patrón de bits que corresponde con un número entero en complemento a 2, y **cvt** lo convierte a un patrón de bits que representa el mismo número, pero codificado en coma flotante según el estándar IEEE 754. Los dos operandos de **cvt** residen en registros de coma flotante. Como el entero inicialmente estará en un registro de propósito general, usaremos **mtc1** para copiar su contenido en el registro de coma flotante.

También es posible convertir un dato desde coma flotante a entero. Aquí hay varias posibilidades (**.w.s**: el destino es un entero y el fuente está en coma flotante con precisión simple; **.w.d**: el destino es un entero y el fuente está en coma flotante con precisión doble):

- Convertir con redondeo según el criterio establecido por defecto en MIPS: **cvt.w.s** / **cvt.w.d**
- Convertir con redondeo al entero más próximo: **round.w.s** / **round.w.d**
- Convertir con redondeo por truncamiento: **trunc.w.s** / **trunc.w.d**
- Convertir con redondeo por exceso (al próximo entero mayor): **ceil.w.s** / **ceil.w.d**
- Convertir con redondeo por defecto (al próximo entero menor): **floor.w.s** / **floor.w.d**

Las instrucciones que convierten a entero también tienen sus dos operandos en registros de coma flotante, si bien el registro destino contendrá un patrón de bits que realmente está representando un entero en complemento a 2. Para copiar este patrón de bits a un registro de propósito general usaremos la instrucción **mfc1**.

MIPS y la coma flotante

Precisión simple: evaluación de expresiones aritméticas

`w = (x*y+z*z)/(x-y);`

```
l.s   $f20,x              # $f20 ← x
l.s   $f22,y              # $f22 ← y
mul.s $f4,$f20,$f22       # $f4 ← x*y
l.s   $f24,z              # $f24 ← z
mul.s $f6,$f24,$f24       # $f6 ← z*z
add.s $f4,$f4,$f6         # $f4 ← x*y+z*z
sub.s $f6,$f10,$f12       # $f6 ← x-y
div.s $f26,$f4,$f6        # $f26 ← (x*y+z*z) / (x-y)
s.s   $f26,w              # w ← $f26
```

Universidad Rey Juan Carlos

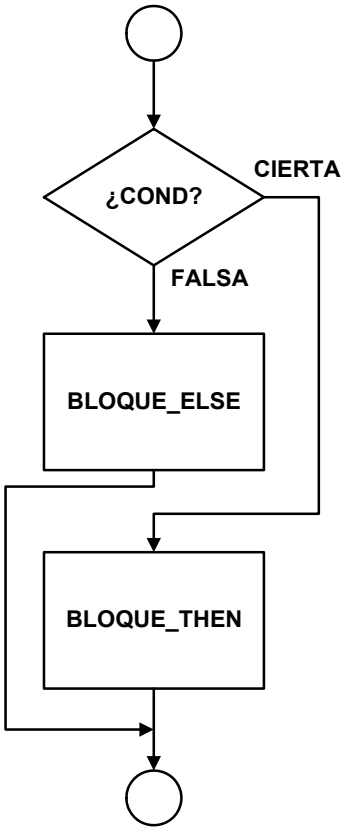
Las expresiones aritméticas en coma flotante se pueden evaluar mediante el modelo de notación polaca inversa con la ayuda de una pila, igual que para los enteros. Pero como en MIPS hay muchos registros, se sustituirá la pila total o parcialmente por registros de propósito general. Por convenio, los más adecuados para ello son los registros temporales **$f4…$f11** y **$f16…$f19**, que se suelen usar para mantener los datos intermedios generados precisamente en la evaluación de expresiones. También por convenio, las variables de memoria se cargan en registros seguros (*saved*) **$f20…$f31**, pensados para mantener copia de las variables de larga duración. No debe olvidarse de que el resultado final de la evaluación de la expresión debe ser grabado en la variable de memoria mediante **s.s**.

Para precisión doble, sólo cambiaría el sufijo de las operaciones, que sería **.d**, y además sería preciso tener en cuenta que los operandos utilizados tienen que ser siempre registros con índice par.

Las instrucciones de coma flotante generan excepciones si hay desbordamiento u otras situaciones no deseadas. En ciertos casos se pueden producir **NaN**, que luego se propagarán a través de las instrucciones que los utilicen indicando que el resultado obtenido es incorrecto.

MIPS y la coma flotante

Precisión simple: estructura de control if-else



C (variables reales en precisión simple):

```
if (x >= y) {
   x = x+z;
}
else {
   x = x-z;
}
```

Ensamblador MIPS:

```
# if (x >=y)
if:     l.s     $f20,x
        l.s     $f22,y
        c.le.s  1,$f22,$f20
        bc1t    1,then
# parte else: x = x-z;
else:   l.s     $f24,z
        sub.s   $f20,$f20,$f24
        s.s     $f20,x
        b       end
# parte then: x = x+z;
then:   l.s     $f24,z
        add.s   $f20,$f20,$f24
        s.s     $f20,x
end:
```

Universidad Rey Juan Carlos

Si queremos tomar decisiones tras comparar datos en coma flotante, no podremos utilizar las instrucciones de ramificación condicional que hemos estudiado hasta ahora (**beq**, **bne**, etc.), ya que éstas comparan datos de tipo entero con o sin signo y residentes en registros de propósito general. Para establecer condiciones que impliquen operandos en coma flotante y ramificar en consecuencia tendremos que recurrir a una instrucción de comparación seguida de una instrucción de ramificación, ambas específicas para el coprocesador 1.

La operación de comparación es **c.cond.fmt cc,$fs, $ft**, donde *cond* se refiere a la condición, y *fmt* se refiere al formato de los datos. Entre las condiciones tendremos **eq** (igual), **le** (menor o igual) y **lt** (menor que), y entre los formatos están la precisión simple (**.s**) y la doble (**.d**). Si comparamos dos registros fuente $fs y $ft y la condición resulta cierta, entonces un cierto *flag* del coprocesador 1 pasa a valer 1, y si no pasa a valer 0. El *flag* seleccionado es el operando **cc**, que aparece en primer lugar y está indicado mediante una constante entre 0 y 7, lo cual indica que la instrucción puede activar o desactivar uno entre 8 *flags* posibles. Existe la posibilidad de utilizar la instrucción de comparación con dos operandos: **c.cond.fmt $fs, $ft** , pero en este caso el *flag* involucrado es siempre el 0.

Para ramificar utilizaremos la instrucción **bc1f cc,etiqueta** , en la que ramificaremos cuando el *flag* **cc** seleccionado sea 0 (*false*) o la instrucción **bc1t cc,etiqueta** , en la que ramificaremos cuando el *flag* **cc** seleccionado sea 1 (*true*). Existen opciones que no indican el *flag* y toman entonces el flag 0 por defecto: son **bc1f etiqueta** y **bc1t etiqueta** .

De esta forma, seremos capaces de realizar estructuras de selección e incluso bucles en los que las condiciones dependan de los valores de ciertos datos que estén codificados en coma flotante.

MIPS y la coma flotante

Precisión simple: recorrido de un vector

```
        .data
# Tamaño dato FP simple: 4
V:      .space  20
suma:   .space  4
# suma: registro $f20
# i:    registro $t0
# temporales: $t0-$t4,$f2
        .text
        ...
        li      $t4,0
        mtc1    $t4,$f20
for:    li      $t0,0
cond_for:
        li      $t1,5
        beq     $t0,$t1,end
cuerpo_for:
        la      $t2,V
        mul     $t3,$t0,4
        addu    $t4,$t2,$t3
        l.s     $f2,0($t4)
        add.s   $f20,$f20,$f2
inc_for:
        addiu   $t0,$t0,1
        b       cond_for
end:
        s.s     $f20,suma
```

C

```
float V[5], suma;
int i;

suma = 0;
for (i=0; i<5; i++)
  suma = suma+V[i];
```

C

```
float V[5], suma;
float *p, *f;

suma = 0;
for (p=V, f=p+5;
     p != f; p++)
  suma = suma+*p;
```

```
        .data
# Tamaño dato FP simple: 4
V:      .space  20
suma:   .space  4
# suma: registro $f20
# p:    registro $t0
# f:    registro $t1
# temporales: $t0,$t1,$f2
        .text
        ...
        li      $t1,0
        mtc1    $t1,$f20
for:    la      $t0,V
        addi    $t1,$t0,20
cond_for:
        beq     $t0,$t1,end
cuerpo_for:
        l.s     $f2,0($t0)
        add.s   $f20,$f20,$f2
inc_for:
        addiu   $t0,$t0,4
        b       cond_for
end:
        s.s     $f20,suma
```

Los vectores de datos en coma flotante se recorren igual que los vectores de números enteros. Aquí se muestra un ejemplo de recorrigo con índices y otro de recorrido con puntero.

En el primer caso, la comparación para determinar la permanencia en el bucle es una comparación de enteros. Las operaciones en coma flotante se circunscriben a la lectura del dato, la actualización del registro utilizado para la copia de la variable suma y su grabación en memoria.

En el segundo caso sucede algo similar: las comparaciones de punteros siguen siendo comparaciones de tipo entero, y las operaciones de coma flotante son las mismas que en el caso anterior.

Índice

Universidad Rey Juan Carlos

Como es sabido, los caracteres constituyen un tipo enumerado que se representa internamente en binario, de acuerdo con un cierto código. El código más habitual es el **ASCII extendido**, en el que cada carácter se representa mediante un octeto (*byte*). Así, en C un carácter es en realidad un entero de 8 bits, que puede ser equiparado a un entero pequeño sin signo (**unsigned char**) o con signo (**signed char**, **char**).

En ensamblador de MIPS (como en C) los caracteres se pueden utilizar en operaciones aritméticas como números enteros de 8 bits con o sin signo extendidos a 32 bits.

En ensamblador, una constante de tipo carácter puede especificarse de forma numérica (con el ordinal correspondiente al carácter) o como un carácter entrecomillado con comillas simples (‘a’, ‘2’, etc). Entonces, **poner un carácter entrecomillado equivale a poner su ordinal**.

Aunque es posible, no es habitual que los programas manipulen caracteres individuales. En su lugar, los caracteres suelen ir en secuencia formando **cadenas** o **tiras de caracteres**. Una cadena de caracteres es un vector cuyos elementos son de tipo carácter. En lenguaje C, las cadenas de caracteres están delimitadas por el final mediante un carácter nulo ó ‘\0’.

A través de una serie de ejemplos, en este apartado vamos a estudiar cómo se definen y manipulan en ensamblador de MIPS los caracteres individuales y las cadenas de caracteres utilizando la codificación ASCII extendido Latin-1 (ISO 8859-1).

MIPS y los caracteres

Creación de espacio en memoria / carga / sumas y restas

C: `char A;` → Ensamblador:

```
A:   .space 1
```

C: `char A='k';` → Ensamblador:

```
A:  .byte  'k'
```

Carga de una variable en un registro → `lbu $s0,A`

`C = 'm';` →

```
li   $s0,'m'
sb   $s0,C
```

Conversión de una variable que contiene una letra minúscula a mayúscula:

- 'A' = 65; 'a' = 97; 'A' - 'a' =-32

`B = C - 'a' + 'A';` →

```
lbu     $s0,C
li      $t0,'a'
subu    $t0,$s0,$t0
addiu   $s1,$t0,'A'
sb      $s1,B
```

Universidad Rey Juan Carlos

Como un carácter ASCII extendido ocupa 8 bits, al crear espacio para una variable sin valor inicial haremos **.space 1**, y si tiene valor inicial, usaremos **.byte** y pondremos el carácter entre comillas simples o bien lo expresaremos mediante su ordinal en cualquier base.

La carga de una variable carácter sobre un registro se puede hacer con **lb** o **lbu**, dependiendo de que consideremos que el carácter tiene signo o no. Recuérdese que **lb** realiza extensión de signo en el registro destino, mientras que **lbu** realiza extensión con ceros.

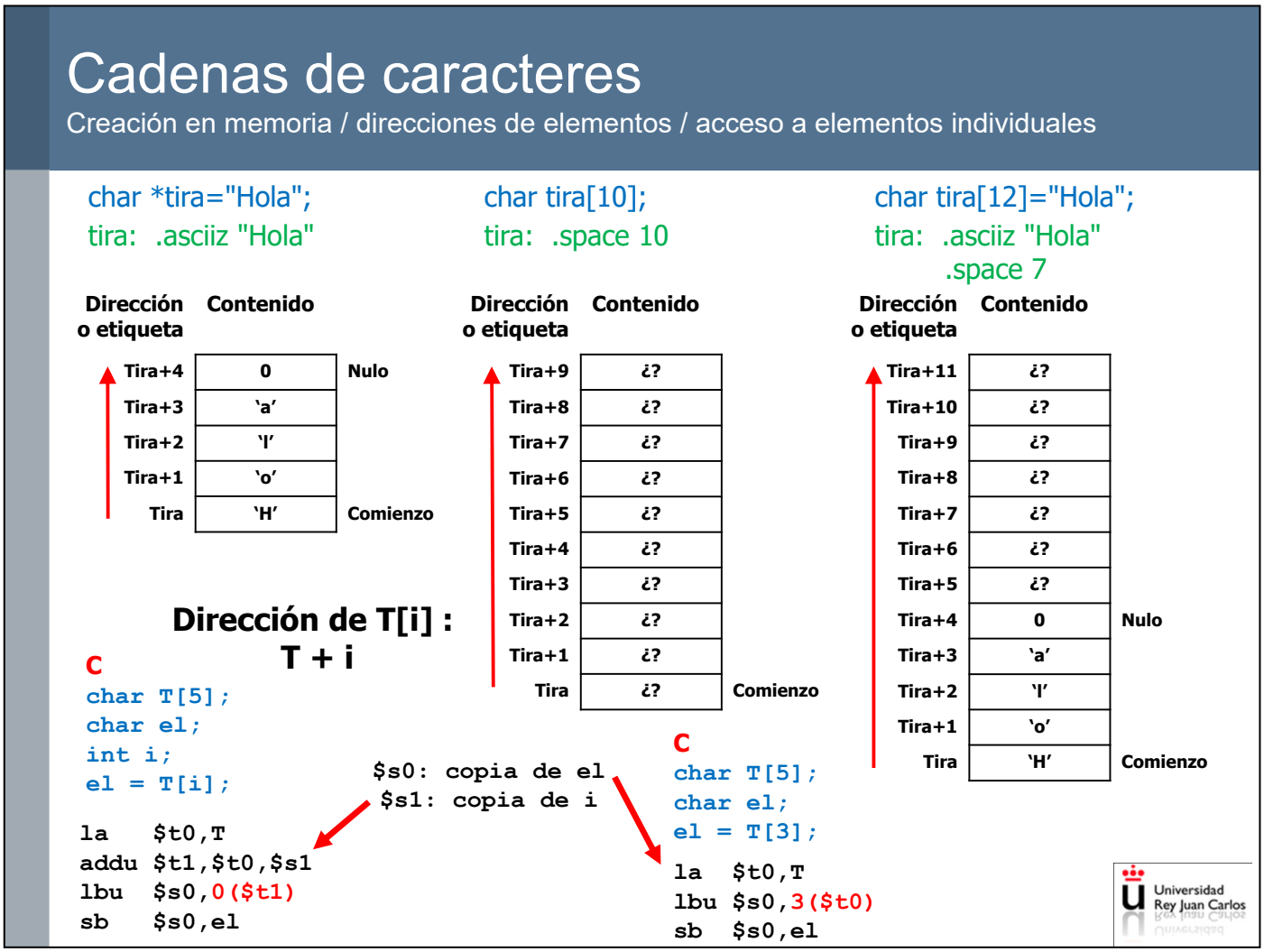
Las constantes de tipo carácter pueden cargarse en registros mediante la pseudoinstrucción **li**.

Igual que en C, los caracteres pueden actuar como operandos en expresiones aritméticas. En este caso, los valores representados por los caracteres son sus ordinales.

A menudo los caracteres no aparecen solos, sino que forman parte de tiras o cadenas de caracteres (***strings***). Un *string* está entonces formado por una secuencia de caracteres almacenados en memoria uno detrás de otro, y terminando por el carácter nulo '\0' (cuyo ordinal es precisamente el 0).

Cadenas de caracteres

Creación en memoria / direcciones de elementos / acceso a elementos individuales



Las tiras o cadenas de caracteres son secuencias unidimensionales de longitud definida que contienen datos simples de tipo carácter. En realidad, son vectores cuyo tipo básico de elemento es un carácter. Los elementos de una tira de caracteres se encuentran almacenados consecutivamente en memoria, y su final está delimitado por la aparición de un carácter nulo (cuyo ordinal es 0). Se utilizará el convenio de C, que establece que, para una cadena de caracteres de **N** elementos, sus índices se encuentran en el rango **0..N-1**.

Hay diferentes formas de crear una tira de caracteres en memoria:

- Creación de una tira con un contenido inicial conocido: lo haremos cuando la tira de caracteres no se modifica a lo largo de la ejecución del programa. En estos casos, la crearemos en memoria en tiempo de ensamblaje con **.asciiz**, que ya pone al final el carácter nulo.
- Creación de una tira vacía: al comenzar el programa no tiene un contenido inicial conocido, y será a lo largo de la ejecución del mismo cuando se escriba algo sobre ella. En este caso crearemos espacio para la misma con **.space**.
- Creación de una tira con un contenido inicial conocido pero que puede crecer en tamaño. En este caso crearemos la tira en memoria en tiempo de ensamblaje mediante **.asciiz**, pero a continuación reservaremos espacio para que pueda crecer en longitud con **.space**.

En general, si la tira de caracteres comienza en una cierta dirección etiquetada como **T**, la dirección del elemento i-ésimo es **T+i**, porque cada carácter ocupa un único *byte*.

Para acceder a los elementos de una tira de caracteres se actúa igual que en el caso de los vectores, con la diferencia de que aquí el tamaño de un elemento es de un *byte*, con lo cual no hace falta multiplicar por el tamaño del elemento, tanto si el índice es constante como si es variable.

Cadenas de caracteres

Recorrido de una cadena

C

```
char t[10];
int lon;

lon = 0;
while ( t[lon] != '\0' )
   lon = lon+1;
```

```
        .data
t:      .space  10
lon:    .space  4
        .text
        ...
        li      $s0,0
while:
        la      $t0,t
        addu    $t1,$t0,$s0
        lbu     $t2,0($t1)
        beqz    $t2,fin
cuerpo_while:
        addiu   $s0,$s0,1
        b       while
fin:
        sw      $s0,lon
```

C

```
char t[10], *p;
int lon;

p = t;
while ( *p != '\0' )
   p++;
lon = p - t;
```

```
        .data
t:      .space  10
lon:    .space  4
# p:    registro $s0
        .text
        ...
        la      $s0,t
while:
        lbu     $t0,0($s0)
        beqz    $t0,fin
cuerpo_while:
        addiu   $s0,$s0,1
        b       while
fin:
        la      $t1,t
        sub     $s1,$s0,$t1
        sw      $s1,lon
```

Universidad Rey Juan Carlos

Para procesar una cadena de caracteres en ensamblador con un bucle podemos adoptar las mismas estrategias que en el caso de los vectores: recorrido con índice y recorrido con puntero.

Si el recorrido es ascendente, en general no controlaremos el final del bucle mediante un índice, sino que el bucle terminará cuando lleguemos al carácter nulo que marca el final de la tira. En los dos ejemplos, esta comprobación se realiza mediante la pseudoinstrucción **beqz**.

Índice

Además de las estructuras de datos con elementos de tipo homogéneo como los vectores, las matrices o las cadenas de caracteres, existen otras estructuras cuyos datos elementales son de tipo heterogéneo, como los datos de tipo **struct** en **C** (**RECORD** en Pascal). Vamos a ver cómo se manejan estos últimos en ensamblador de MIPS.

Las tablas son estructuras de datos unidimensionales cuyos elementos son de tipo **struct** y están almacenados en memoria en posiciones contiguas. Así, para manejar una tabla necesitaremos combinar los conceptos de acceso a elementos de un vector y acceso a uno de los campos de un **struct**.

Las listas encadenadas son estructuras de datos unidimensionales cuyos elementos son de tipo **struct**, pero los elementos de una lista no tienen por qué situarse uno tras otro en memoria: uno de los campos de cada elemento de la misma es un puntero al siguiente, y gracias a esos punteros podremos realizar un recorrido por la lista.

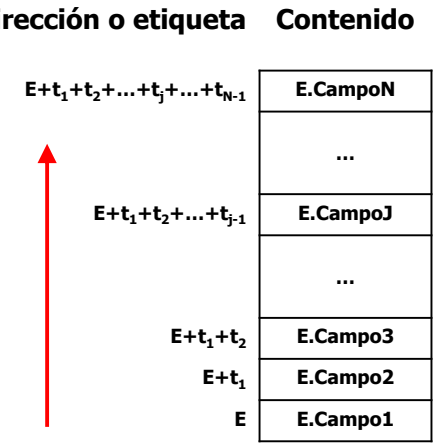
En este apartado sólo estudiaremos cómo se define y manipula un **struct** individual, y no se abordará el manejo de tablas, listas encadenadas y otras estructuras más complejas.

Estructuras de datos (*struct*)

Creación en memoria / direcciones de elementos / acceso a elementos individuales

C

```
struct {
  Tipo1 Campo1; /* t1 octetos */
        ...
  TipoN CampoN; /* tN octetos */
} TipoEstructura t;
```



Dirección de E.campoJ :
E + t1+t2+...+t_{j-1}

C

```
struct {
   char Nombre[32];        /* 32 octetos */
   int DNI;                /* 4 octetos  */
   char LetraDNI;          /* 1 octeto   */
   char Departamento;      /* 1 octeto   */
/*  Hueco para alineamiento: 2 octetos  */
   int Sueldo;             /* 4 octetos  */
} TipoEmpleado;    /* TOTAL: 44 octetos  */

TipoEmpleado empl;
         ...
empl.Sueldo = empl.Sueldo + 100;
```

```
          .data
empl:     .space   44

          .text
          ...
          la       $s0,empl
          lw       $t0,40($s0)
          addi     $t0,$t0,100
          sw       $t0,40($s0)
```

Universidad Rey Juan Carlos

Un ***struct*** es una ED formada por un número fijo de campos de igual o distinto tipo, cada uno con su propio identificador. La información contenida en los campos forma en conjunto un todo que permite representar objetos o conceptos de una cierta complejidad. En Pascal a este tipo de estructura se le llama **registro** o **estructura de tipo RECORD**.

Un campo de un **struct** puede ser un dato simple o a su vez una estructura de datos tipo **struct**, un vector, una cadena de caracteres, etc. Entonces, para acceder a un campo de un **struct** necesitaremos conocer la dirección base de la estructura, y la distancia del campo concreto al comienzo de la misma.

Como los campos de un **struct** se almacenan de forma consecutiva en memoria, la distancia de un campo al origen de la estructura es igual a la suma de los tamaños de todos los campos precedentes. Si suponemos que cada campo **CampoI** tiene un tamaño $\mathbf{t_i}$, entonces la distancia de **CampoJ** al comienzo del **struct** es $\mathbf{d_j=t_1+t_2+...+t_{j-1}}$. Dado que $\mathbf{d_j}$ es conocido en tiempo de compilación, si cargamos la dirección base del **struct** con **la $reg_base,empl** podremos acceder a cualquiera de sus campos mediante el direccionamiento $\mathbf{d_j}$**($reg_base)**.

Una variable **struct TipoEmpleado** de la diapositiva en principio ocuparía 32 (**Nombre**) + 4 (**DNI**) + 1 (**LetraDNI**) + 1 (**Departamento**) + 4 (**Sueldo**) = 42 bytes. El campo **Sueldo** es de tipo entero, y si sumamos los tamaños de los campos previos sale que está a 32+4+1+1=38 bytes del comienzo del **struct**. Si la dirección base de **TipoEmpleado** está alineada a múltiplo de 4, entonces el campo **Sueldo** no estará correctamente alineado, ya que su dirección no será múltiplo de 4. Por tanto, habrá que recurrir a una de estas dos opciones:

- Dejar un hueco de 2 *bytes* detrás del campo **Departamento**, y el **struct** ocupará 44 bytes.
- Poner los campos de tipo carácter al final para no estropear el alineamiento de los demás.

En el ejemplo se muestra cómo subir el sueldo del empleado: primero se lee el campo, después se le suma 100 y finalmente se almacena el nuevo sueldo en el propio campo en memoria.

Índice

Universidad Rey Juan Carlos

En ciertos lenguajes ensambladores existen instrucciones que permiten manipular bits individuales en los datos, no así en MIPS, en el que el dato accesible de menor tamaño ocupa un octeto. Por tanto, aunque para almacenar una variable booleana sería suficiente con un bit, en MIPS utilizaremos al menos 8 bits para ello.

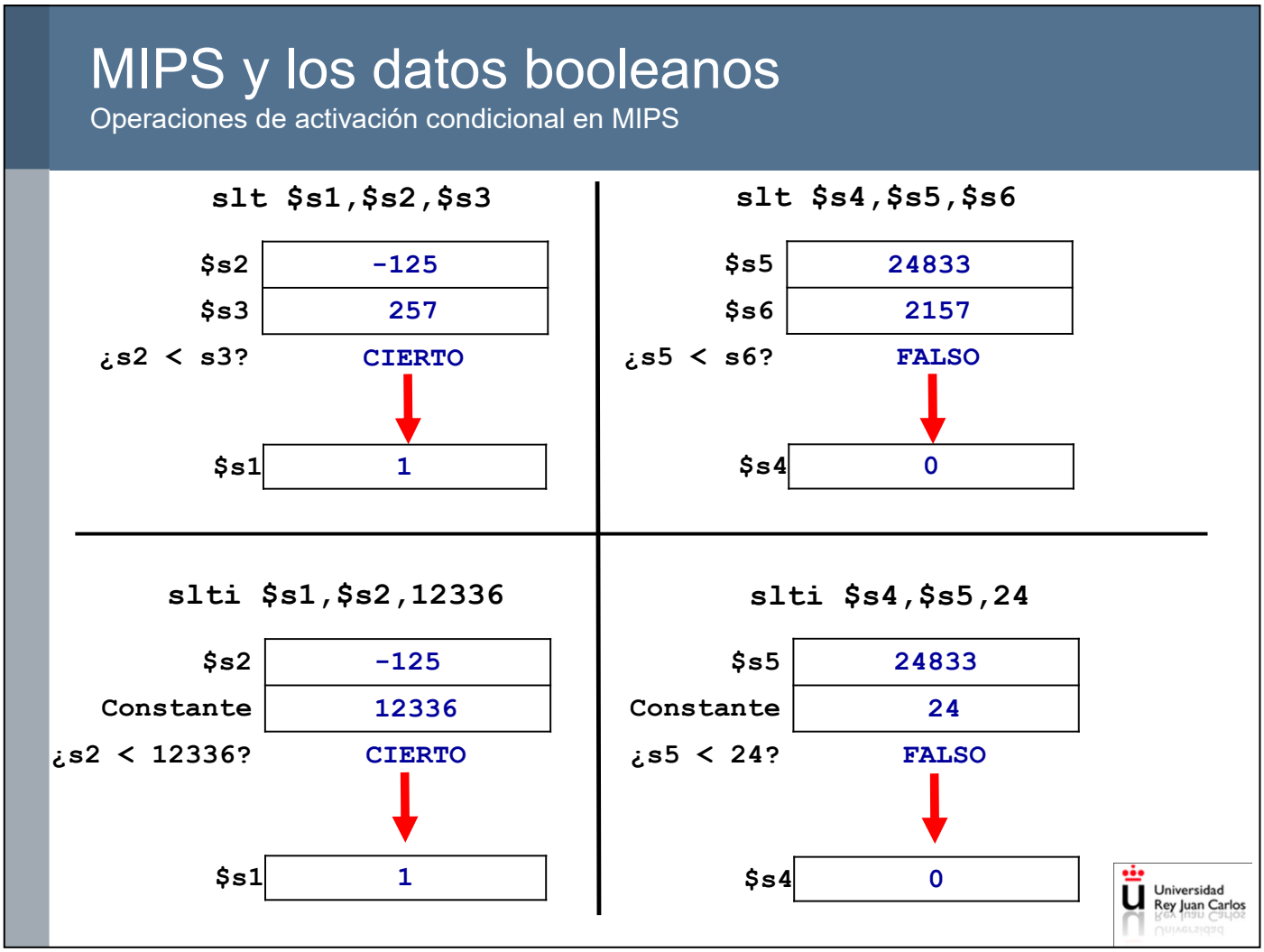
En lenguaje C no existen las variables booleanas, aunque sí se pueden escribir expresiones booleanas cuyo resultado puede ser "falso" (igual a 0) o "cierto" (cualquier valor distinto de 0). Pero esta falta de estandarización para el valor "cierto" puede causar problemas. Pensemos por ejemplo en una instrucción **and**, que realiza el producto lógico de dos operandos bit a bit. Supongamos que el primer operando tiene a 1 los bits 0, 1, 2 y 3, y que el segundo tiene a 1 los bits 4, 5, 6 y 7. Según el criterio anterior, ambos datos son ciertos, pero el **and** lógico de los mismos nos daría un resultado con todos sus bits a 0, es decir, daría un valor falso.

Para evitar problemas, estandarizaremos los valores booleanos, de modo que un dato será "**cierto**" (***true***) cuando contenga el valor entero 1 (todos los bits a 0 menos el último), y será "**falso**" (***false***) cuando valga 0 (todos sus bits a 0).

Las instrucciones aritméticas y lógicas manejan registros de 32 bits.

MIPS y los datos booleanos

Operaciones de activación condicional en MIPS



Las operaciones de activación condicional están relacionadas con las comparaciones de datos numéricos que devuelven valor cierto o falso.

El nemotécnico de estas operaciones comienza por **s** (de ***set***) y va seguido de varias letras que indican el tipo de comparación. Por ejemplo, **slt** significa **poner a 1 si menor que** (***set on less than***), y tiene tres operandos; esta instrucción compara el primer operando fuente **op1** con el segundo **op2**, y pone a 1 el operando destino (que va en primer lugar en la instrucción, como casi siempre) si **op1 < op2**. Si la comparación resulta falsa, en el destino se almacena un 0.

Existen múltiples operaciones de este tipo (realizar una comparación y poner a 1 si es cierto). Todas tienen tres operandos, siendo el destino el primero de ellos y los fuentes los otros dos. Los tres operandos residen en registros, salvo cuando el nemotécnico lleva el sufijo **i**, en cuyo caso el segundo operando fuente es un inmediato. Las comparaciones consideran operandos con signo en complemento a 2, salvo que lleven el sufijo **u**, en cuyo caso se consideran operandos en binario puro. En total tenemos las siguientes:

- **slt**, **slti**, **sltu**, **sltiu**: ponen a 1 el destino si **op1 < op2**. Son las únicas instrucciones, el resto son pseudoinstrucciones.
- **seq**: pone a 1 el destino si **op1 = op2**. Vale para números con y sin signo.
- **sne**: pone a 1 el destino si **op1 <> op2**. Vale para números con y sin signo.
- **sgt**, **sgtu**: ponen a 1 el destino si **op1 > op2**.
- **sge**, **sgu**: ponen a 1 el destino si **op1 >= op2**.
- **sle**, **sleu**: ponen a 1 el destino si **op1 <= op2**.

MIPS y los datos booleanos

Datos booleanos

C: `char A;` → Ensamblador: `A: .space 1`

Carga de una variable en un registro →

```
lbu $s0,A
```

Almacenamiento de una constante →

```
li  $s0,1
sb  $s0,A
```

```
A = (B < C);
{A "booleano";
B y C enteros con signo}
```

→

```
lw  $s0,B
lw  $s1,C
slt $s2,$s0,$s1
sb  $s2,A
```

Universidad Rey Juan Carlos

Para mantener una variable booleana en memoria bastaría con un octeto, y por tanto crearemos el espacio para la misma mediante la directiva **.space 1**. Entonces usaremos instrucciones de *byte* (**lbu**, **lb**, **sb**) para cargar y almacenar variables booleanas.

MIPS y los datos booleanos

Evaluación de expresiones booleanas

zb = (x > y) && (!xb || yb);

```
# Evaluación completa
        lw      $s0,x           # $s0 ← x
        lw      $s1,y           # $s1 ← y
        sgt     $t0,$s0,$s1     # $t0 ← x > y
        lbu     $s2,xb          # $s2 ← xb
        seq     $t1,$s2,$zero   # $t1 ← !xb
        lbu     $s3,yb          # $s3 ← yb
        or      $t1,$t1,$s3     # $t1 ← !xb || yb
        and     $s4,$t0,$t1     # $s4 ← (x > y) && (!xb || yb)
        sb      $s4,zb          # zb ← $s4
```

```
# Evaluación por cortocircuito (perezosa)
        lw      $s0,x           # $s0 ← x
        lw      $s1,y           # $s1 ← y
        sgt     $t0,$s0,$s1     # $t0 ← x > y
        beqz    $t0,fin         # Cortocircuito si falso (AND)
        lbu     $s2,xb          # $s2 ← xb
        seq     $t0,$s2,$zero   # $t0 ← NOT xb
        bnez    $t0,fin         # Cortocircuito si cierto (OR)
        lbu     $t0,yb          # $t0 ← yb
        sb      $t0,zb          # zb ← $t0
```

Universidad Rey Juan Carlos

La evaluación de expresiones booleanas, como la de las expresiones aritméticas, puede hacerse mediante el modelo de notación polaca inversa con ayuda de una pila. Pero como MIPS tiene muchos registros, los utilizaremos para ello.

Aquí se muestran dos ejemplos de evaluación de la expresión **zb = (x > y) && (!xb || yb)**. La primera es una evaluación completa, mientras que la segunda es una evaluación perezosa o por cortocircuito. Esta última es más eficiente, ya que termina la evaluación en cuanto se conoce cuál va a ser el resultado de la misma. En realidad, en la evaluación perezosa de la expresión del ejemplo se ejecuta el siguiente algoritmo:

IF **(x > y)** es FALSO
THEN la expresión es FALSA (porque es un AND);
ELSE IF **NOT xb** es CIERTO
THEN la expresión es CIERTA (porque es un OR);
ELSE **yb** tiene el valor booleano final de la expresión.

Índice

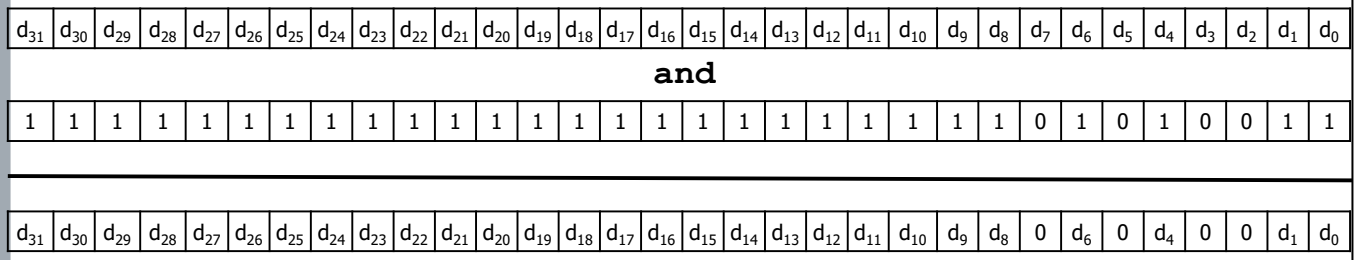
Universidad Rey Juan Carlos

Gracias a las operaciones lógicas y a las **máscaras de bits**, en MIPS es posible alterar algunos bits (no necesariamente contiguos) de un dato, dejando intactos los restantes.
Todos los datos de un computador están representados en binario. Lo que da sentido a las máscaras de bits es que están diseñadas *ex profeso* para actuar selectivamente sobre ciertos bits de otro dato a través de una operación lógica AND, OR o XOR.
Las máscaras de bits habitualmente tienen valor constante, y se suelen crear en binario, pero en ensamblador se escriben en hexadecimal.
Podemos distinguir entre tres tipos de máscaras:

- **Máscaras AND**: sirven para poner a 0 ciertos bits de un dato y dejar inalterados los demás mediante una operación de producto lógico.
- **Máscaras OR**: sirven para poner a 1 ciertos bits de un dato y dejar inalterados los demás mediante una operación de suma lógica.
- **Máscaras XOR**: sirven para invertir ciertos bits de un dato y dejar inalterados los demás mediante una operación de suma lógica exclusiva.

MIPS y las máscaras de bits: AND

- Máscara AND: pone a 0 los bits seleccionados.
  - Ejemplo: borrar los bits 2, 3, 5 y 7 de $v0, sin tocar los demás.



Máscara binaria: 1111 1111 1111 1111 1111 1111 0101 0011

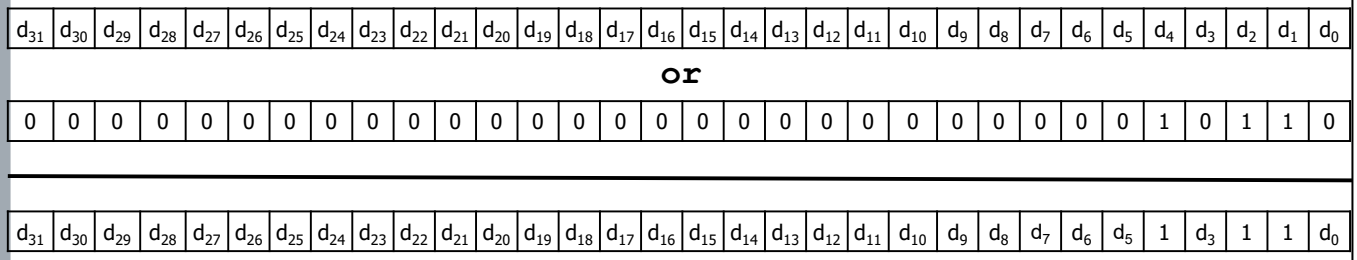
```
li   $t0,0xffffff53
and  $v0,$v0,$t0
```

Universidad Rey Juan Carlos

Para las máscaras AND lo lógico sería usar la instrucción **andi**. Pero hay que tener en cuenta que, como el inmediato es de 16 bits y se realiza extensión con ceros, aplicar una máscara de este tipo con **andi** pone a 0 los 16 bits superiores del resultado. Si no deseamos esto último, será preciso cargar la máscara de 32 bits en un registro mediante **li** y después usar una instrucción **and**.

MIPS y las máscaras de bits: OR

- Máscara OR: pone a 1 bits seleccionados.
  - Ejemplo: poner a 1 los bits 1, 2 y 4 de $v0, sin tocar los demás.



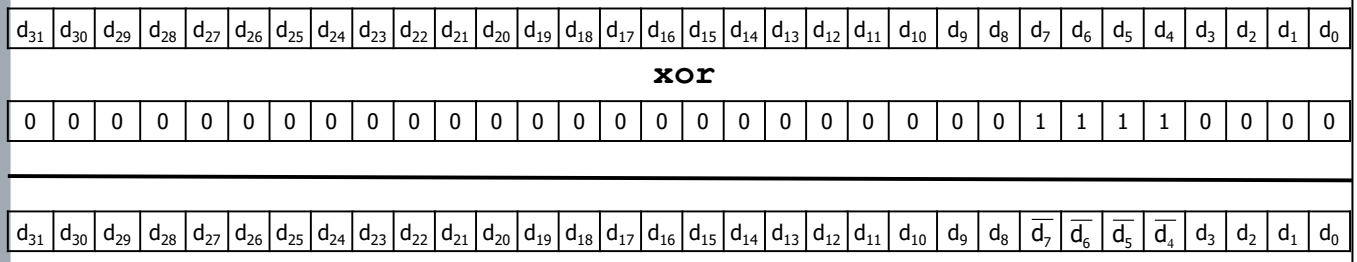
Máscara binaria: 0000 0000 0000 0000 0000 0000 0001 0110

```
ori $v0,$v0,0x0016
```

Si aplicamos una máscara OR mediante una instrucción **ori**, por la razón anterior no podremos actuar sobre los 16 bits más significativos, y tendríamos que cargar la máscara de 32 bits con **li** y después utilizar la instrucción **or**.

MIPS y las máscaras de bits: XOR

- Máscara XOR: invierte bits seleccionados.
  - Ejemplo: invertir los bits 4, 5, 6 y 7 de $v0, sin tocar los demás.



Máscara binaria: 0000 0000 0000 0000 0000 0000 1111 0000

```
xori $v0,$v0,0x00f0
```

Universidad Rey Juan Carlos

Con las máscaras XOR sucede algo parecido: con **xori** no podremos invertir ninguno de los 16 bits más significativos del dato. Para poder hacerlo cargaremos la máscara de 32 bits en un registro con **li** y después usaremos **xor**.

Índice

Universidad Rey Juan Carlos

Las **subrutinas** o **subprogramas** son fragmentos de código diseñados para realizar determinadas tareas y que pueden ser invocados desde diferentes puntos del programa principal o desde otras subrutinas. Los lenguajes de alto nivel tienen mecanismos para ejecutar subprogramas, y por tanto el lenguaje ensamblador tiene que dar soporte a las mismas.

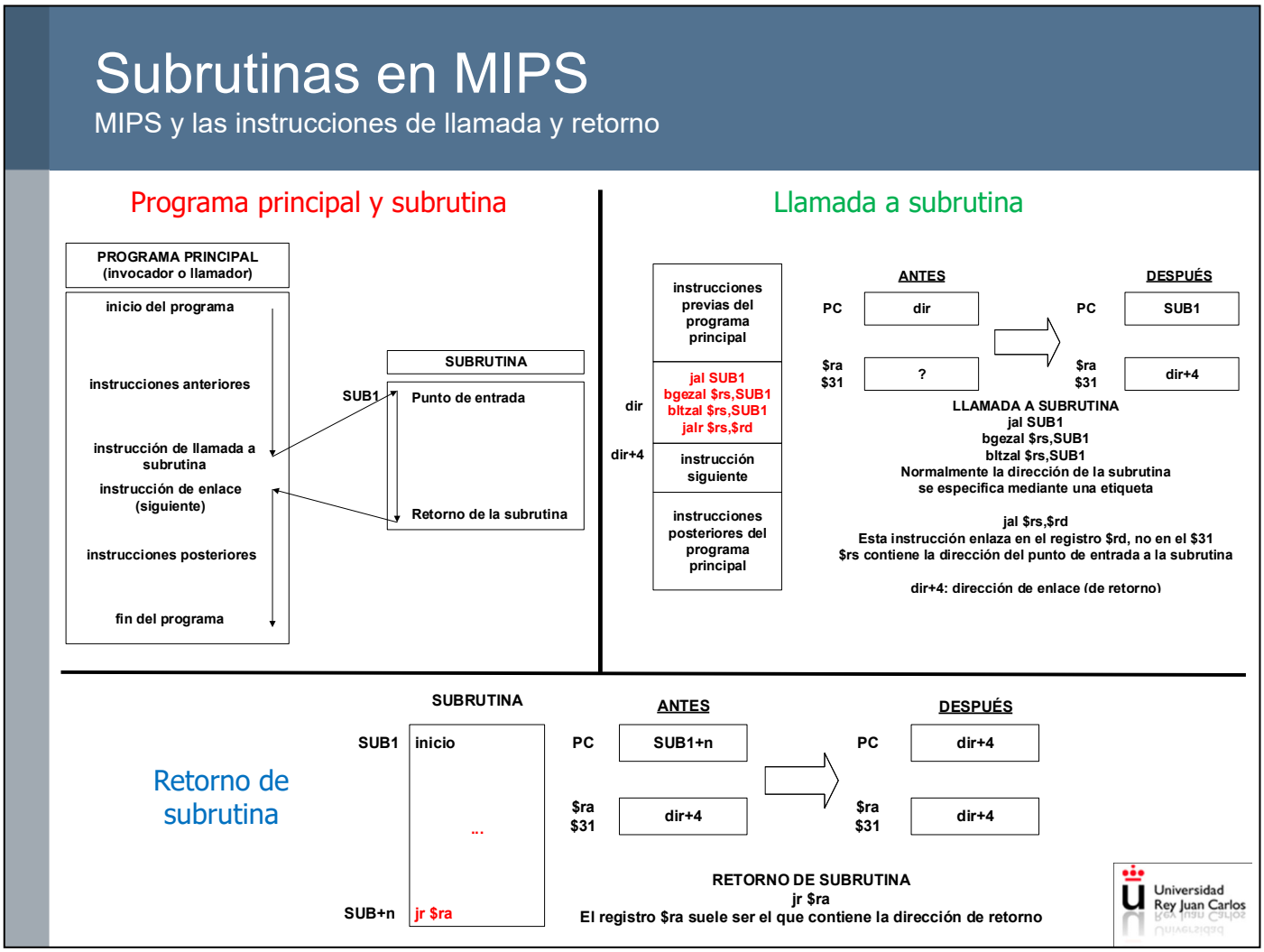
En lenguaje C, las subrutinas reciben el nombre de **funciones**. Una función en C recibe una serie de **argumentos** y produce un **valor de retorno**. Las funciones reciben todos sus **argumentos por copia**, es decir, reciben una copia de sus valores, pero no pueden acceder a las variables que se pasan como argumentos, y por tanto no pueden modificarlas. Si una función quiere acceder y modificar una variable pasada por argumento, entonces debe recibir un puntero a la variable (**argumento por referencia**) en vez de una copia de la misma. A través del puntero, el acceso a la variable verdadera pasada como argumento está garantizado.

En lenguaje Pascal hay dos tipos de subprogramas: **funciones** (que reciben todos sus argumentos por copia y que devuelven un valor simple) y **procedimientos** (pueden recibir argumentos pasados por copia o por referencia, y no devuelven ningún valor).

Tanto en C como en Pascal u otros lenguajes los subprogramas tienen **variables locales**, que usan para sus propios cálculos.

El **convenio de llamada a subrutina**, dependiente del compilador utilizado, establece cómo deben gestionarse los argumentos, los valores de retorno y las variables locales, y también indica cómo manejar los registros de propósito general y de coma flotante, facilitando el desarrollo de programas y garantizando que la conexión entre invocadores y subrutinas se realiza correctamente.

Cuando una subrutina no invoca a ninguna otra, se la denomina subrutina hoja (***leaf subroutine***). Una subrutina que contenga una o varias llamadas anidadas a otras subrutinas recibe el nombre de subrutina tallo (***stem subroutine***).



Como es sabido, la ejecución de un programa comporta la ejecución secuencial de las instrucciones que lo componen. En ocasiones hay rupturas de la secuencia de ejecución, que hasta ahora nos han servido para implementar sentencias de selección y bucles. Al invocar una subrutina, también se rompe la secuencia de ejecución, de modo que se deja de ejecutar las instrucciones del **invocador** (el programa principal en la figura), y se pasa a ejecutar las instrucciones de la subrutina (el **invocado**) a partir de su **punto de entrada** (que es la dirección de la primera instrucción de la subrutina). La última instrucción ejecutada en la subrutina es una **instrucción de retorno**, que devuelve el flujo de ejecución al **punto de retorno** (que es la dirección de la **instrucción de enlace**, que es la que va detrás de la llamada).

Como es posible invocar una subrutina desde diferentes puntos, la subrutina deberá ser capaz de volver a diferentes puntos de retorno. Para ello, invocaremos la subrutina con una **bifurcación con enlace**, que antes de saltar a la subrutina se encargará de guardar la **dirección de enlace** en un lugar conocido por la subrutina, que así podrá utilizarlo para devolver el flujo de ejecución al lugar desde el que fue invocada.

En **MIPS**, la instrucción de llamada a subrutina más habitual es **jal**, que tiene como operando una etiqueta que indica el punto de entrada de la subrutina. Lo primero que hace **jal** es guardar la dirección de enlace (que está en el **PC**, que ya ha sido incrementado) en el registro **$31** o **$ra** (de ***return address***), y después escribe en el **PC** la dirección del punto de entrada de la subrutina.

Otras instrucciones de bifurcación con enlace son **bgezal** y **bltzal**, que comparan un registro con 0, y si la evaluación que realizan es correcta saltan a la subrutina enlazando sobre **$ra**. También está **jalr**, que lleva dos registros como operandos: el primero contiene la dirección del punto de entrada a la subrutina, y el segundo es un registro en el que se guardará la dirección de enlace.

Para retornar de la subrutina ejecutaremos la instrucción **jr**, que salta a la dirección indicada en un registro. Normalmente haremos **jr $ra**, ya que **$ra** será el registro de enlace más habitual.

Subrutinas en MIPS

Argumentos, valor de retorno y variables locales

Cabecera: `int funcion (int arg0, int arg1, char arg2, int *arg3);`

Llamada: `z = funcion(x,5,car,&y);`

Programa principal (invocador)

```
            ...
# Preparación de argumentos
      lw    $a0,x
      li    $a1,5
      lbu   $a2,car
      la    $a3,y
# Llamada a subrutina
      jal   funcion
# Punto de retorno
      sw    $v0,z
            ...
```

Subrutina

```
# Punto de entrada
funcion:
            ...
# Uso de un parámetro por copia
            addi   $t0,$a0,2
            ...
# Lectura de un parámetro por referencia
            lw     $t1,0($a3)
            ...
# Escritura de un parámetro por referencia
            sw     $t2,0($a3)
            ...
# Escritura del valor de retorno
            add    $v0,$t1,$t0
            ...
# Retorno de la subrutina
            jr     $ra
```

Universidad Rey Juan Carlos

En los lenguajes de alto nivel, los **argumentos** se incluyen en la llamada a la función, y van entre paréntesis separados por comas. Pero los operandos de las instrucciones de llamada a subrutina sólo indican la dirección del punto de entrada y a veces la del punto de retorno. Antes de efectuar la llamada a la subrutina, el invocador debe colocar los argumentos en un lugar concreto donde la subrutina espera encontrarlos. Por convenio, los compiladores de C para **MIPS** reservan 4 registros de propósito general para los argumentos de las funciones: **$a0** (**$4**), **$a1** (**$5**), **$a2** (**$6**) y **$a3** (**$7**), y el invocador copia los argumentos en dichos registros antes de invocar a la subrutina.

La diapositiva presenta un ejemplo de una función con cuatro parámetros, tres pasados por copia y uno por referencia. Los argumentos por copia se cargan en **$a0-$a3** desde memoria con **lw**, **lbu**, etc., o se copian desde otro registro con **move** u otra operación con el registro de argumento como destino, o si es una constante, la copiaremos sobre el registro de argumento con **li**. Para pasar un argumento por referencia, se carga su dirección de memoria en el registro de argumento con **la**, indicando como segundo operando la etiqueta de la variable, aunque hay otras opciones posibles (a veces tendremos que calcular la dirección mediante aritmética de punteros).

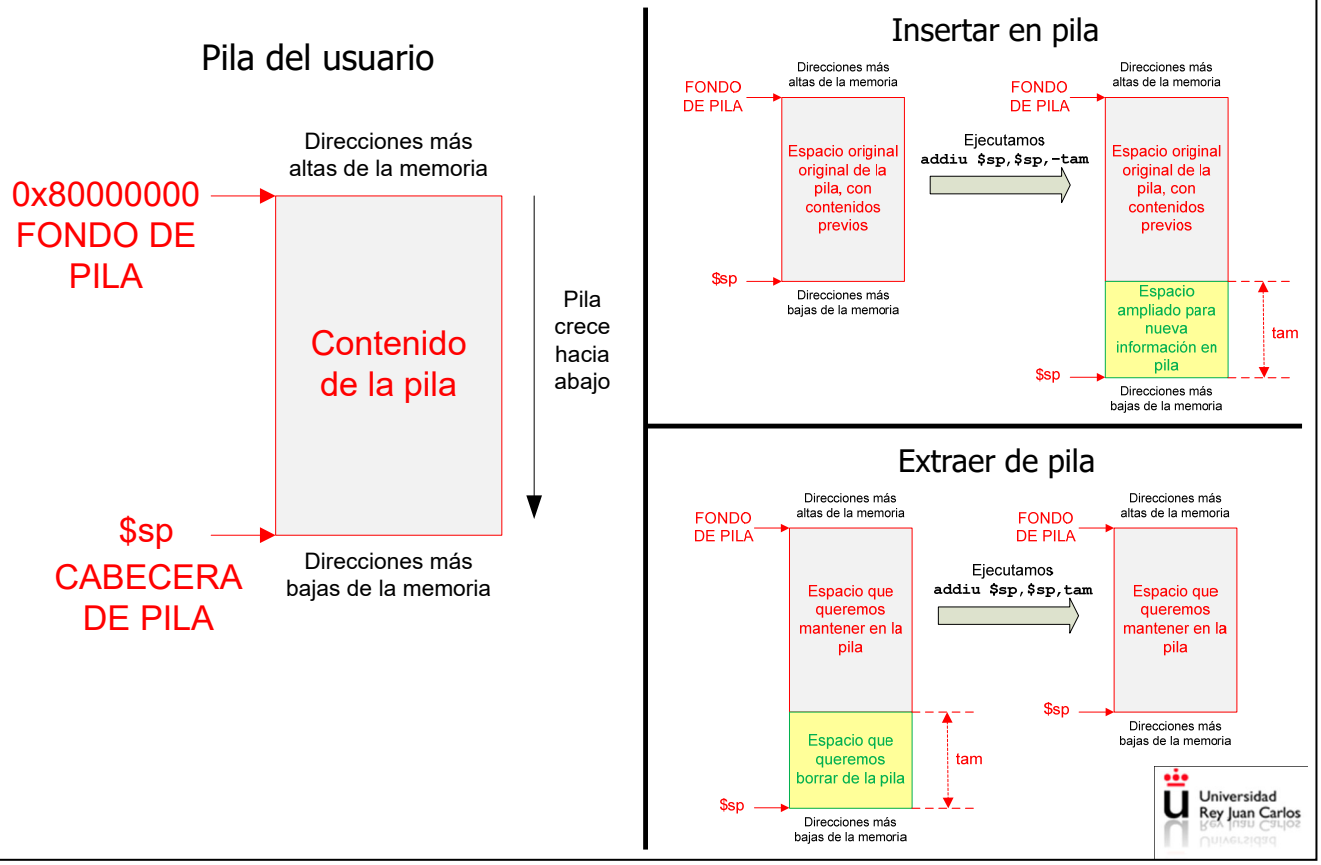
Las subrutinas ponen su **valor de retorno** en los registros **$v0** (**$2**) y **$v1** (**$3**). El segundo registro se usa cuando el valor de retorno no cabe en el primero. Si el valor de retorno ocupa un espacio mayor y no cabe en los dos registros, entonces la subrutina copia el valor de retorno en memoria, y utiliza **$a0** como un puntero al lugar donde lo ha copiado, con lo que sólo quedan tres registros para los argumentos.

Los argumentos de tipo **float** se pasan a través de los registros **$f12**, **$f13**, **$f14** y **f15**, y si la subrutina devuelve un dato en coma flotante, lo pone en los registros **$f0** y **f1**.

Como veremos, lo más cómodo es utilizar los registros temporales **$t0-$t9** para albergar las **variables locales** de las subrutinas. En cuanto a la coma flotante, los registros **$f2-f11** y **$f16-$f19** también se usan como temporales y son aptos para las variables locales de las subrutinas.

Subrutinas en MIPS

MIPS y la pila de usuario



MIPS cuenta con varios registros para los parámetros, el valor de retorno y las variables locales. Sin embargo, a veces estos registros no son suficientes para albergar la información requerida. Por tanto, necesitamos un espacio adicional al que recurrir si es necesario. Este espacio es la **pila**.

La **pila** (*stack*) es una zona de memoria de suma importancia cuando los programas cuentan con subprogramas, ya que en ella se pueden almacenar múltiples informaciones como los argumentos de la subrutina, la dirección de retorno o las variables locales, o incluso se pueden almacenar copias de registros cuyo contenido queramos salvaguardar ante llamadas a subrutina.

Los programas de usuario tienen una pila en memoria que comienza en una dirección fija, denominada **fondo de pila**, y termina en una dirección variable llamada **cabecera de pila**. El contenido de la pila es precisamente todo lo que se encuentra entre la cabecera de pila y el fondo de pila. Por tanto, la pila es de tamaño variable, ya que crece cuando se produce una llamada a subrutina, y decrece cuando se retorna. En **MIPS** y otras muchas máquinas existe un registro llamado **$sp** (***stack pointer***) que contiene en todo momento la dirección de la cabecera de la pila. Para asegurar el alineamiento de la información almacenada en la pila, la dirección de la cabecera de la pila tiene que ser siempre múltiplo de 8.

La pila varía durante la ejecución del programa. En una llamada a subrutina, se inserta en pila un conjunto de informaciones denominado **marco de pila**, **registro de activación** o **bloque de activación**. En **MIPS**, la reserva de espacio para el marco de pila se hace sumando un valor negativo al puntero de pila, así que **la pila crece hacia direcciones decrecientes de memoria**. Se inserta información en la pila (**apilar** o ***push***) mediante instrucciones de almacenamiento (**sw**, etc), con direccionamiento indirecto al puntero de pila con desplazamiento. Utilizando este direccionamiento siempre es posible acceder al contenido de la pila con instrucciones de carga (**lw**, etc). Para "eliminar" información de la pila (**desapilar** o ***pop***), se suma al puntero de pila un valor positivo igual en valor absoluto al que se restó antes, dejando así la cabecera de pila donde estaba.

Subrutinas en MIPS

Convenio de salvaguarda de registros de propósito general en pila ante llamadas a subrutina

Nombre	Función	Guardado en pila por el invocador si los quiere preservar	Preservados por el invocado si modifica su contenido
$zero ($0)	Valor constante 0	No	No
$at ($1)	Uso propio del traductor en pseudoinstrucciones	No	No
$v0 y **$v1** ($2 y $3)	Valor de retorno de subrutina	No	No
$a0 ... $a3 ($4 ... $7)	Argumentos de la subrutina	Sí	No
$t0 ... $t9 ($8 ... $15, $24 y $25)	Variables temporales o auxiliares de corta duración	Sí	No
$s0 ... $s7 ($16 ... $23)	Variables de larga duración	No	Sí
$k0 y **$k1** ($26 y $27)	Uso propio del núcleo (*kernel*) del sistema operativo	No	No
$gp ($28)	Puntero a zona de variables estáticas (*global pointer*)	No	Sí
$sp ($29)	Puntero de pila (*stack pointer*)	No	Sí (no lo almacena en pila)
$fp ($30)	Puntero de marco (*frame pointer*)	No	Sí
$ra ($31)	Dirección de retorno (*return address*)	No	Sí

`$s0` ... `$s7`: *saved registers* (registros "seguros" o salvados)

`$t0` ... `$t9`: *temporary registers* (registros temporales)

Universidad Rey Juan Carlos

Los compiladores para **MIPS** tienen establecido un convenio para gestionar el uso de los registros de propósito general, con objeto de que no haya problemas con sus contenidos en las llamadas a subrutina. Pensemos en una subrutina **SUB1** que usa **$s0-$s1** y **$t0-$t3** para variables locales y cálculo de expresiones. Pensemos ahora en un programa principal que **$s0-$s2** para mantener variables de larga duración y **$t0-$t2** para cálculos temporales. Si el programa principal invoca a **SUB1**, al retornar de ella **$s0**, **$s1**, **$t0**, **$t1** y **$t2** podrían tener un valor distinto al de antes, con lo cual el programa principal funcionará mal con toda probabilidad.

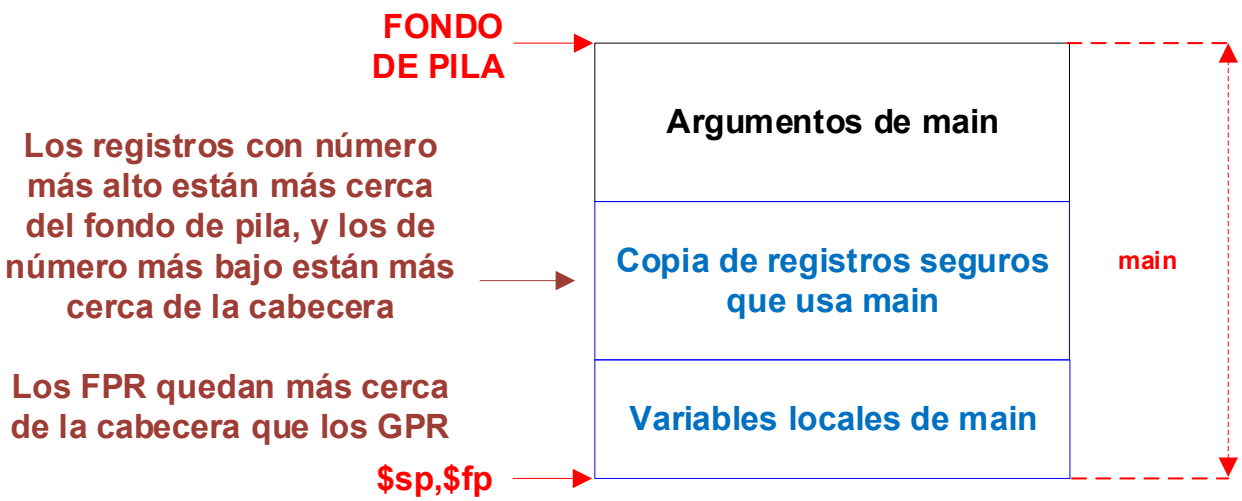
Para evitar esto se puede utilizar la pila. Si copiamos el contenido de los registros que usa el programa principal en la pila antes de que la subrutina los modifique, podremos restaurarlos con su valor anterior más adelante, y así el invocador funcionará correctamente.

En **MIPS** la tarea de salvaguarda y restauración se reparte entre invocador e invocado. Así, la subrutina deberá devolver intacto el contenido de los registros seguros **$s0-s7**, el registro de enlace **$ra**, el puntero global **$gp** y el puntero de marco **$fp** (que se usa para apuntar a la pila y acceder a sus contenidos a través de él). Por contra, la subrutina podrá modificar libremente los contenidos de los registros temporales **$t0-$t9** y de los registros de argumento **$a0-$a3**. Por tanto:

- Si el invocador quiere preservar el contenido de algunos registros temporales y/o registros de argumento, debe salvarlos en pila antes de invocar a la subrutina, para poder restaurarlos después de que la subrutina retorne.
- Si el invocado quiere alterar el contenido de los registros **$s0-$s7**, **$ra**, **$gp** o **$fp**, copiará su contenido en la pila antes de modificarlos y lo restaurará antes de retornar.
- Los registros de coma flotante **$f2-$f11** y **$f16-$f19** se usan para mantener datos temporales, y los registros **$f12-$f15** sirven para pasar argumentos, con lo que las subrutinas disponen de ellos con libertad, pero los registros **$f20-$f31** son para variables de larga duración y la subrutina debe preservar su contenido.

Subrutinas en MIPS

Marco de pila: subrutina hoja



Azul: parte de la pila creada por **main**

El **marco de pila** (***stack frame***) está formado por la información que se inserta en la pila cuando se invoca una subrutina. Para acceder a esta información puede utilizarse direccionamiento indirecto con desplazamiento al **puntero de pila** (***stack pointer***, **$sp**), que apunta permanentemente a la cabecera de la pila, o al **puntero de marco** (***frame pointer***, **$fp**), que apunta a algún punto determinado del marco de pila. El convenio utilizado por el compilador **GCC** estipula que el puntero de marco **$fp** apunta a la cabecera de la pila, igual que **$sp**. El marco de pila lo crea la propia subrutina nada más comenzar y lo destruye justo antes de retornar. En él están las variables locales, las copias de los registros que es necesario preservar y el espacio para sus argumentos. El tamaño del marco de pila, medido en bytes, debe ser múltiplo de 8, para garantizar que se cumplen las restricciones de alineamiento.

Si una función tiene más de cuatro argumentos, del quinto en adelante los recibe a través de la pila. El convenio establece que **siempre se reservan 16 bytes en la cabecera de la pila para los cuatro primeros argumentos** (aunque haya menos de cuatro). El invocador no copia los argumentos en este hueco, que queda para uso de la subrutina invocada. El espacio para el primer argumento queda en la cabecera de la pila, y los demás van a continuación. El quinto y demás argumentos son copiados en orden por el invocador en la pila detrás del hueco para el cuarto.

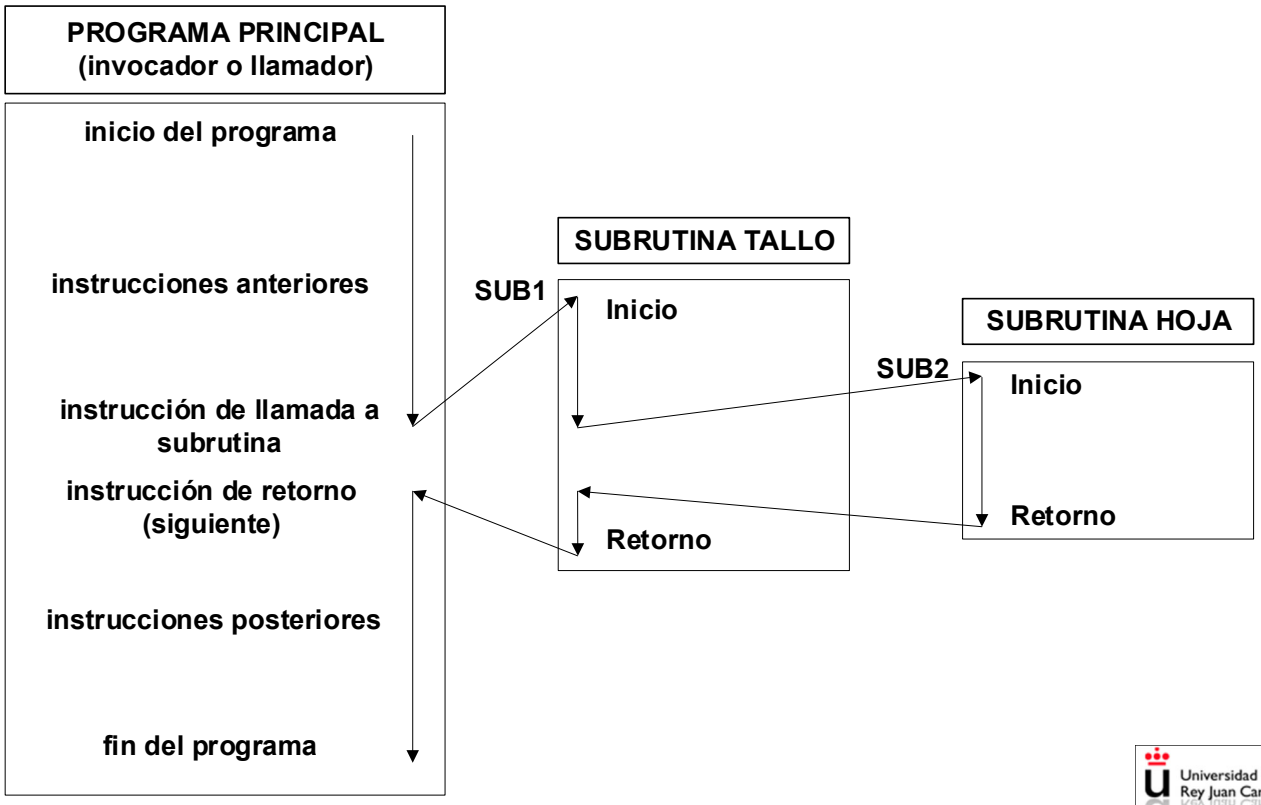
Según el convenio, es práctico usar los registros **$t0-$t9** para las variables locales. Si no bastan, la subrutina tendrá que reservar espacio adicional para variables locales en la pila. El acceso a las mismas se efectuará con direccionamiento indirecto a **$sp** o **$fp** con desplazamiento.

En un caso real, la ejecución comienza por un código de arranque del programa, insertado por el montador, que inicia registros y pone los argumentos de **main** en la pila, y después salta a la propia función **main**. Entonces lo primero que hace **main** es reservar espacio en pila para registros y variables locales, y salvaguarda en ella los registros seguros que usa.

Si **main** no contiene llamadas a subrutina, es una **subrutina hoja**.

Subrutinas en MIPS

Subrutinas anidadas



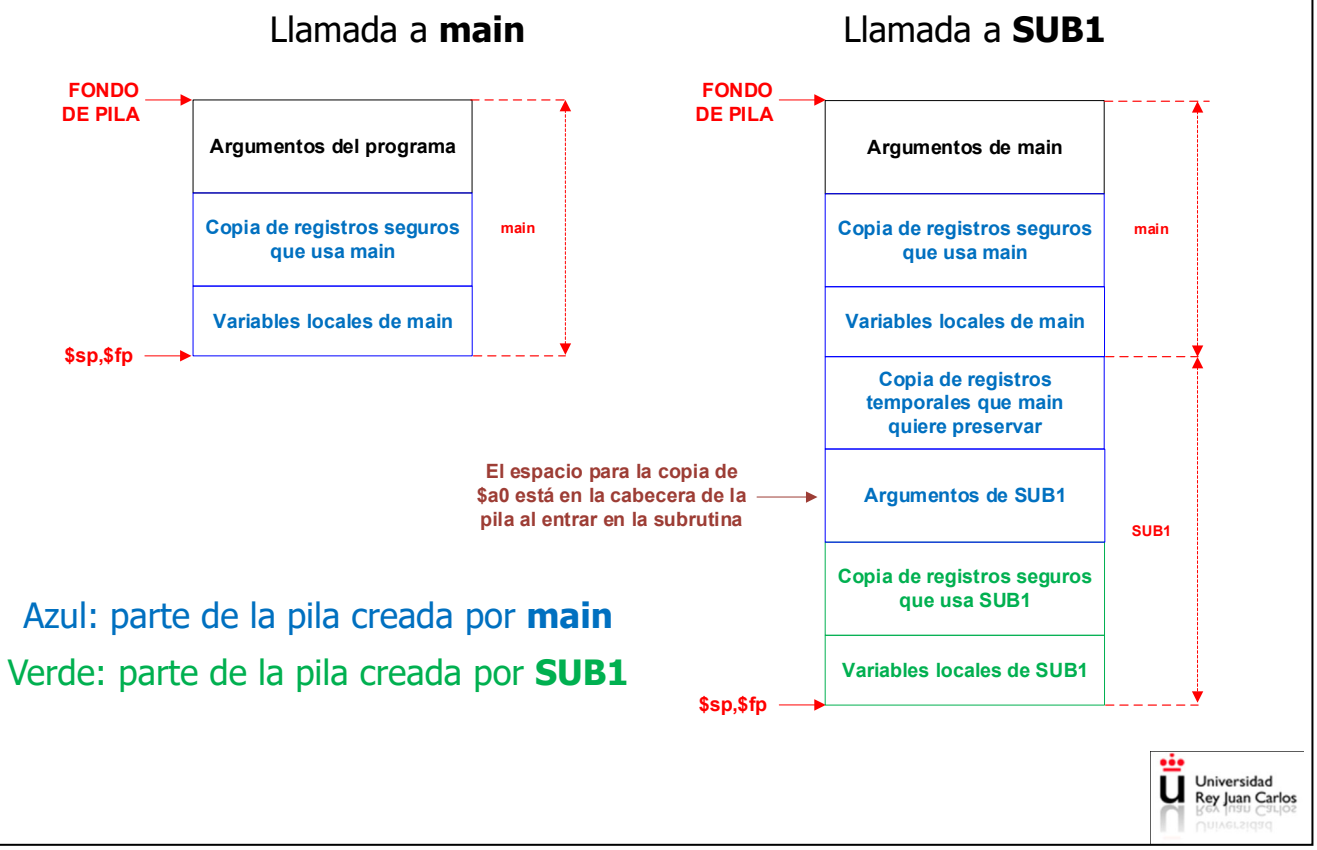
Una subrutina puede contener llamadas anidadas a otras subrutinas. En la figura, **SUB1** es una **subrutina tallo**, ya que invoca a su vez a **SUB2**, que en este caso es una **subrutina hoja**.

En **MIPS**, si el programa principal ejecuta **jal SUB1**, la dirección de retorno al mismo se guarda en **$ra**. Si luego **SUB1** ejecuta **jal SUB2**, se sobreescribirá el valor de **$ra** con la dirección de la instrucción a la que debe retornar **SUB2**, con lo que **SUB1** habrá perdido su propia dirección de enlace y no podrá retornar al programa principal.

Para resolver este problema, **SUB1** debe salvaguardar en pila el contenido de **$ra** como un registro seguro más antes de invocar a **SUB2**, y restaurarlo justo antes de ejecutar **jr $ra** para poder volver al punto de retorno en el programa principal. Esto está recogido en el convenio de llamada a subrutina: si el invocado modifica **$ra**, debe salvaguardarlo en pila para restaurarlo antes de retornar.

Subrutinas en MIPS

Marco de pila en llamadas anidadas



Si **main** llama a la subrutina **SUB1**, debe copiar en la pila el contenido de los registros temporales, de argumento y de retorno que quiere preservar ante la llamada. Una vez hecho esto, copiará en los registros de argumento (y en la pila los que no quepan) los parámetros de la llamada a **SUB1**, y pasará el control a la misma mediante **jal SUB1**.

La subrutina **SUB1** creará espacio en la pila para los registros seguros y las variables locales, y después copiará en pila los registros seguros que utilice. A continuación, realizará los cálculos que le han sido encomendados, y, antes de retornar, restaurará los registros seguros desde la pila, copiará el valor de retorno en **$v0** (o donde sea necesario) y retornará a **main** mediante la instrucción **jr $ra**.

Cuando se retorna a **main**, ésta restaura la copia de los registros temporales que ha guardado en pila antes de llamar a **SUB1**, recoge el valor de retorno y después continúa con la ejecución.

La parte de la pila que está en azul es creada por **main**, y comprende la copia de los registros seguros que modifica, sus propias variables locales, la copia de los registros temporales que quiere presevar ante la llamada a **SUB1** y los argumentos que le pasa a **SUB1**.

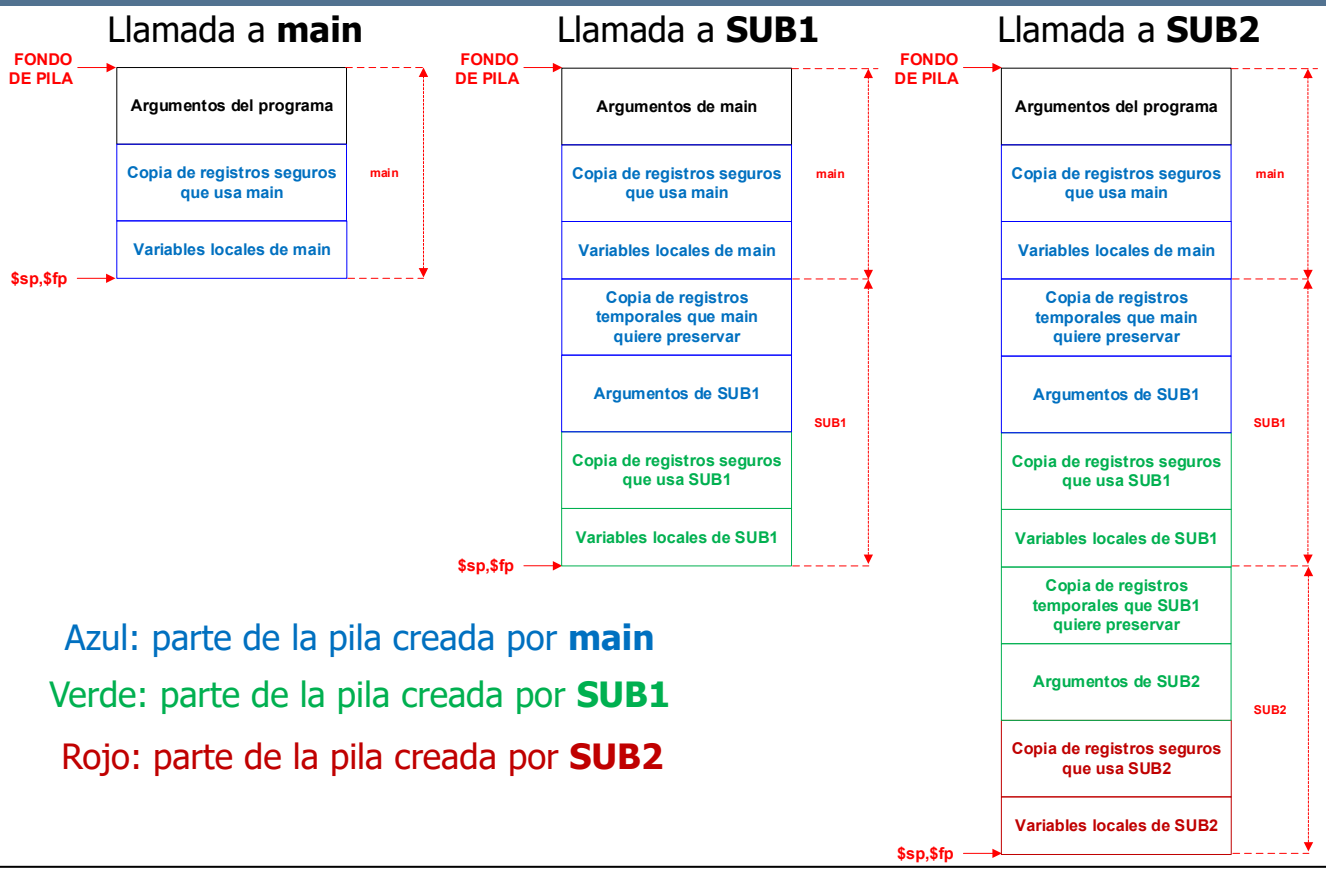
Por su parte, la parte verde es la creada por **SUB2**, e incluye la copia de los registros seguros que utiliza y sus propias variables locales.

En la figura de la izquierda se muestran los contenidos insertados en la pila a causa de la llamada a **main**: sus argumentos están ahí antes de entrar en **main** (los inserta el código previo), mientras que es **main** quien pone en pila la copia de sus registros seguros y las variables locales.

En la parte inferior de la figura de la derecha aparecen los contenidos insertados en pila a causa de la llamada a **SUB1**: los registros temporales que quiere preservar **main** y los argumentos de **SUB1** son insertados en pila por **main**, y la copia de los registros seguros que usa **SUB1** y sus variables locales son insertados por **SUB1**.

Subrutinas en MIPS

Marco de pila en llamadas anidadas



En la parte inferior de la figura de la derecha se añade el efecto que produce en pila la llamada a **SUB2**. Antes de invocar a **SUB2**, la subrutina **SUB1** copia en pila los registros temporales que quiere preservar, y después pone en la misma los argumentos de llamada a **SUB2**. A continuación, realiza la llamada a **SUB2**, que pone en pila la copia de los registros seguros que utiliza, y también pone sus variables locales.

La parte verde de la figura corresponde con contenidos puestos en la pila por **SUB1**, y la parte roja es la que **SUB2** pone en la pila.

MIPS y el convenio de llamada a subrutina

Marco de pila: ejemplo

```
int SUB1 (int arg1, int *arg2, int arg3, int arg4,
          int *arg5, int arg6) {
     int x, y;
     char c;
// Su traducción a ensamblador usa $s0 y $s1
       ...
// Su traducción a ensamblador necesita preservar $t3
     x  = SUB2(p0,p1,p2,p3,p4,p5);
     ...
}
int SUB2 (int arg1, int arg2, int *arg3, int arg4,
          int arg5, int *arg6) {
     ...
}
```

Información insertada en pila por **SUB1**

8 bytes para `$ra` y `$fp`.

8 bytes para `$s0` y `$s1`.

9 bytes para variables locales.

4 bytes para `$t3`.

24 bytes para argumentos de SUB2.

TOTAL: 53 bytes ⇒ 56 bytes para que sea múltiplo de 8.

Universidad Rey Juan Carlos

Se presenta a continuación un ejemplo de llamada anidada a subrutina.

La función **SUB1** tiene 6 parámetros, cuatro de ellos por valor y dos por referencia.

SUB1 tiene dos variables locales de tipo entero y una de tipo carácter que van a ubicarse en pila. Además, utiliza **$s0** y **$s1** para otras dos variables locales.

La función **SUB1** es de tipo tallo, porque invoca a la subrutina **SUB2**, que a su vez tiene otros 6 parámetros. Por ello, SUB1 tiene que guardar en pila el contenido de **$ra** para poder recuperarlo al final y volver así al punto de retorno en el invocador.

Para que no haya errores en la ejecución de **SUB1**, ésta tiene que preservar el contenido del registro **$t3** ante la llamada a **SUB2**.

Como **SUB1** introduce información en la pila, modifica el puntero de marco **$fp**, por lo cual tiene que copiar su contenido en pila al principio y restaurarlo al final.

- El espacio en pila creado por SUB1 es el siguiente:
- Espacio para copiar la dirección de enlace ($ra): 4 *bytes*.
- Espacio para copiar el puntero de marco ($fp): 4 *bytes*.
- Espacio para copiar los registros seguros ($s0 y $s1): 8 *bytes*.
- Espacio para las variables locales: 9 *bytes*.
- Espacio para registros temporales que hay que preservar ($t3): 4 *bytes*.
- Espacio para los argumentos de llamada a **SUB2**: 24 *bytes*.

Sumando todo, **SUB1** debe insertar en la pila informaciones que en total ocupan 53 *bytes*. Como el puntero de pila tiene que estar alineado a múltiplo de 8, **SUB1** reservará en pila 56 *bytes*.

MIPS y el convenio de llamada a subrutina

Ejemplo: crear y destruir marco de pila / salvar y restaurar registros seguros

Dirección	Direcciones más altas de memoria	Tamaño
	Resto del marco de pila creado por el invocador	
$fp+76	Argumento 6 de SUB1	4 bytes
$fp+72	Argumento 5 de SUB1	4 bytes
$fp+68	Hueco para el argumento 4 de SUB1	4 bytes
$fp+64	Hueco para el argumento 3 de SUB1	4 bytes
$fp+60	Hueco para el argumento 2 de SUB1	4 bytes
$fp+56	Hueco para el argumento 1 de SUB1	4 bytes
$fp+52	Copia de $ra	4 bytes
$fp+48	Copia de $fp	4 bytes
$fp+44	Copia de $s1	4 bytes
$fp+40	Copia de $s0	4 bytes
	Hueco de 3 bytes	3 bytes
$fp+36	Variable local c	1 byte
$fp+32	Variable local y	4 bytes
$fp+28	Variable local x	4 bytes
$fp+24	Copia de $t3	4 bytes
$fp+20	Argumento 6 de SUB2	4 bytes
$fp+16	Argumento 5 de SUB2	4 bytes
$fp+12	Hueco para el argumento 4 de SUB2	4 bytes
$fp+8	Hueco para el argumento 3 de SUB2	4 bytes
$fp+4	Hueco para el argumento 2 de SUB2	4 bytes
$fp+0	Hueco para el argumento 1 de SUB2	4 bytes
	Direcciones más bajas de memoria	

```
SUB1:
# Crear marco de pila
        addiu   $sp,$sp,-56
# Salvar registros $ra, $fp y seguros
        sw      $ra,52($sp)
        sw      $fp,48($sp)
        sw      $s1,44($sp)
        sw      $s0,40($sp)
# Asignar puntero de marco
        move    $fp,$sp
# Realizar operaciones
        ...
# Apilar 4 primeros argumentos de SUB1
# Acceder a variables locales
# Acceder a los argumentos
# Invocar a SUB2
        ...
# Restaurar registros $ra, $fp y seguros
        lw      $ra,52($sp)
        lw      $fp,48($sp)
        lw      $s1,44($sp)
        lw      $s0,40($sp)
# Destruir marco de pila
        addiu   $sp,$sp,56
# Retornar de SUB1
        jr      $ra
```

Universidad Rey Juan Carlos

Ante todo, es necesario hacer un esquema de los contenidos de la pila que de uno u otro modo afectan a la ejecución de **SUB1**. Teniendo en cuenta que en el convenio del compilador **GCC** el puntero de marco apunta también a la cabecera de la pila, hay que anotar los desplazamientos relativos a **$fp** correspondientes a todos los ítems almacenados en la misma, con objeto de acceder a ellos correctamente durante la ejecución de **SUB1**.

Dado que el marco de pila que debe crear **SUB1** ocupa 56 bytes, al comenzar la subrutina crearemos el espacio necesario en la pila con **addiu $sp,$sp,-56**.

A continuación copiaremos los contenidos de los registros **$ra**, **$fp**, **$s0** y **$s1** en la pila con instrucciones **sw**, utilizando direccionamiento indirecto al puntero de pila **$sp** con desplazamiento.

Lo siguiente será ajustar el puntero de marco para que apunte a la cabecera de la pila. Esto lo haremos con **move $fp,$sp**.

A partir de este momento, **SUB1** realizará los cálculos que le han sido encomendados, y accederá a la información contenida en el marco de pila mediante operaciones de carga y/o almacenamiento con direccionamiento indirecto al puntero de marco **$fp** con desplazamiento.

Antes de retornar, es preciso restaurar los valores de los registros **$ra**, **$fp**, **$s0** y **$s1** mediante instrucciones **lw** con direccionamiento indirecto al puntero de pila **$sp** con desplazamiento.

Destruiremos el marco de pila con **addiu $sp,$sp,56**, y después saltaremos al punto de retorno en el invocador mediante la instrucción **jr $ra**.

MIPS y el convenio de llamada a subrutina

Ejemplo: copiar argumentos propios en pila / acceder a variables locales

Dirección	Direcciones más altas de memoria	Tamaño
	Resto del marco de pila creado por el invocador	
$fp+76	Argumento 6 de SUB1	4 bytes
$fp+72	Argumento 5 de SUB1	4 bytes
$fp+68	Hueco para el argumento 4 de SUB1	4 bytes
$fp+64	Hueco para el argumento 3 de SUB1	4 bytes
$fp+60	Hueco para el argumento 2 de SUB1	4 bytes
$fp+56	Hueco para el argumento 1 de SUB1	4 bytes
$fp+52	Copia de $ra	4 bytes
$fp+48	Copia de $fp	4 bytes
$fp+44	Copia de $s1	4 bytes
$fp+40	Copia de $s0	4 bytes
	Hueco de 3 bytes	3 bytes
$fp+36	Variable local c	1 byte
$fp+32	Variable local y	4 bytes
$fp+28	Variable local x	4 bytes
$fp+24	Copia de $t3	4 bytes
$fp+20	Argumento 6 de SUB2	4 bytes
$fp+16	Argumento 5 de SUB2	4 bytes
$fp+12	Hueco para el argumento 4 de SUB2	4 bytes
$fp+8	Hueco para el argumento 3 de SUB2	4 bytes
$fp+4	Hueco para el argumento 2 de SUB2	4 bytes
$fp+0	Hueco para el argumento 1 de SUB2	4 bytes
	Direcciones más bajas de memoria	

```
SUB1:
# Crear marco de pila
        addiu   $sp,$sp,-56
# Salvar registros seguros
# Asignar puntero de marco
# Realizar operaciones
# Copiar 4 primeros args. de SUB1 en pila
        sw      $a0,56($fp)
        sw      $a1,60($fp)
        sw      $a2,64($fp)
        sw      $a3,68($fp)
# Realizar operaciones
        ...
# Acceso a variables locales
        lw      $s0,28($fp) # Acceso a x
        sw      $s1,32($fp) # Acceso a y
        lbu     $s2,36($fp) # Acceso a z
# Realizar operaciones
        ...
# Restaurar registros $ra, $fp y seguros
        ...
# Destruir marco de pila
        addiu   $sp,$sp,56
# Retornar de S1
        jr      $ra
```

Universidad Rey Juan Carlos

El espacio que queda justo antes del marco de pila creado por **SUB1** corresponde con sus argumentos de llamada, y fue creado y rellenado por el código que la invocó. El invocador copió en los huecos correspondientes los argumentos 5 y 6, pero para los cuatro primeros simplemente reservó 16 *bytes* que no rellenó con ningún contenido.

Si **SUB1** lo precisa, será ella quien copie en dicho espacio los argumentos del primero al cuarto mediante cuatro instrucciones **sw** que tendrán como operando fuente los registros **$a0**, **$a1**, **$a2** y **$a3** respectivamente. Esta operación puede ser necesaria si, por ejemplo, hay que pasar por referencia alguno de estos cuatro argumentos a una subrutina anidada. Como es sabido, los argumentos por referencia deben residir forzosamente en memoria, y el espacio que acabamos de rellenar sirve de soporte en memoria para ello.

Para acceder a las variables locales ubicadas en la pila, utilizaremos instrucciones de carga o almacenamiento con direccionamiento indirecto a **$fp** con desplazamiento. Es necesario establecer un desplazamiento fijo para cada variable, que nos permitirá acceder a ellas a lo largo de toda la ejecución de **SUB1**.

MIPS y el convenio de llamada a subrutina

Ejemplo: acceder a argumentos pasados por valor y por referencia

Dirección	Direcciones más altas de memoria	Tamaño
	Resto del marco de pila creado por el invocador	
$fp+76	Argumento 6 de SUB1	4 bytes
$fp+72	Argumento 5 de SUB1	4 bytes
$fp+68	Hueco para el argumento 4 de SUB1	4 bytes
$fp+64	Hueco para el argumento 3 de SUB1	4 bytes
$fp+60	Hueco para el argumento 2 de SUB1	4 bytes
$fp+56	Hueco para el argumento 1 de SUB1	4 bytes
$fp+52	Copia de $ra	4 bytes
$fp+48	Copia de $fp	4 bytes
$fp+44	Copia de $s1	4 bytes
$fp+40	Copia de $s0	4 bytes
	Hueco de 3 bytes	3 bytes
$fp+36	Variable local c	1 byte
$fp+32	Variable local y	4 bytes
$fp+28	Variable local x	4 bytes
$fp+24	Copia de $t3	4 bytes
$fp+20	Argumento 6 de SUB2	4 bytes
$fp+16	Argumento 5 de SUB2	4 bytes
$fp+12	Hueco para el argumento 4 de SUB2	4 bytes
$fp+8	Hueco para el argumento 3 de SUB2	4 bytes
$fp+4	Hueco para el argumento 2 de SUB2	4 bytes
$fp+0	Hueco para el argumento 1 de SUB2	4 bytes
	Direcciones más bajas de memoria	

```
SUB1:
# Crear marco de pila
        addiu   $sp,$sp,-56
# Salvar registros seguros
# Asignar puntero de marco
# Copiar 4 primeros args. de SUB1 en pila
# Realizar operaciones
# Leer argumentos por valor
        move    $s0,$a0     # Argumento 1
        lw      $s6,76($fp) # Argumento 6
# Leer argumentos por referencia
        lw      $s1,0($a1)  # Argumento 2
        lw      $t5,72($fp) # Argumento 5
        lw      $s5,0($t5)
# Escribir argumentos por referencia
        sw      $t0,0($a1)  # Argumento 2
        lw      $t5,72($fp) # Argumento 5
        sw      $t1,0($t5)
# Realizar operaciones
# Destruir marco de pila
        addiu   $sp,$sp,56
# Retornar de S1
        jr      $ra
```

Universidad Rey Juan Carlos

La diapositiva muestra ejemplos de cómo **SUB1** accede a sus propios argumentos.

El argumento 1 está pasado por valor, y reside en $a0. Para leerlo, simplemente nombraremos **$a0** como un operando fuente en cualquier instrucción. Si queremos modificar la copia local, usaremos **$a0** entonces como operando destino.

El argumento 6 también está pasado por valor, pero reside en memoria en la posición **$fp+76**. Para leerlo, usaremos una instrucción de carga **lw** con **76($fp)** como operando fuente, y para modificar la copia local, usaremos **sw** con **76($fp)** como operando destino.

El argumento 2 está pasado por referencia. Por tanto, reside en memoria, y está apuntado por el registro **$a1**. Entonces, para leerlo usaremos **lw** con **0($a1)** como operando fuente, y para modificarlo recurriremos a **sw** con **0($a1)** como operando destino.

El argumento 5 también está pasado por referencia. En este caso, el puntero también reside en memoria, y está en **$fp+72**. Por tanto, antes de acceder a la posición de memoria donde se encuentra la variable, es preciso cargar el puntero en un registro disponible. Supondremos que **$t5** lo está. Entonces, con **lw $t5,72($fp)** ponemos el puntero a la variable en **$t5**, y leeremos el argumento y lo copiaremos en otro registro libre, por ejemplo **$s5**, con **lw $s5,0($t5)**. Si mantenemos el puntero al argumento en **$t5**, lo modificaremos con una instrucción **sw** que lleve como operando destino precisamente **0($t5)**.

MIPS y el convenio de llamada a subrutina

Ejemplo: invocar subrutina hoja

Dirección	Direcciones más altas de memoria	Tamaño
	Resto del marco de pila creado por el invocador	
$fp+76	Argumento 6 de SUB1	4 bytes
$fp+72	Argumento 5 de SUB1	4 bytes
$fp+68	Hueco para el argumento 4 de SUB1	4 bytes
$fp+64	Hueco para el argumento 3 de SUB1	4 bytes
$fp+60	Hueco para el argumento 2 de SUB1	4 bytes
$fp+56	Hueco para el argumento 1 de SUB1	4 bytes
$fp+52	Copia de $ra	4 bytes
$fp+48	Copia de $fp	4 bytes
$fp+44	Copia de $s1	4 bytes
$fp+40	Copia de $s0	4 bytes
	Hueco de 3 bytes	3 bytes
$fp+36	Variable local c	1 byte
$fp+32	Variable local y	4 bytes
$fp+28	Variable local x	4 bytes
$fp+24	Copia de $t3	4 bytes
$fp+20	Argumento 6 de SUB2	4 bytes
$fp+16	Argumento 5 de SUB2	4 bytes
$fp+12	Hueco para el argumento 4 de SUB2	4 bytes
$fp+8	Hueco para el argumento 3 de SUB2	4 bytes
$fp+4	Hueco para el argumento 2 de SUB2	4 bytes
$fp+0	Hueco para el argumento 1 de SUB2	4 bytes
	Direcciones más bajas de memoria	

```
SUB1:
# Crear marco de pila
        addiu   $sp,$sp,-56
# Realizar operaciones
        ...
# Antes de llamar a SUB2, salvar $t3
        sw      $t3,24($fp)
# Poner parámetros de SUB2
        move    $a0,$t7     # Argumento 1
        lw      $a1,28($fp) # Argumento 2
        addiu   $a2,$fp,32  # Argumento 3
        li      $a3,25      # Argumento 4
        sw      $t3,16($fp) # Argumento 5
        addiu   $t8,$fp,68  # Argumento 6
        sw      $t8,20($fp)
# Invocar a SUB2
        jal     SUB2
# Recoger el valor de retorno
        sw      $v0,28($fp)
# Tras volver de SUB2 se restaura $t3
        lw      $t3,24($fp)
# Realizar operaciones
        ...
# Destruir marco de pila
        addiu   $sp,$sp,56
# Retornar de S1
        jr      $ra
```

Universidad Rey Juan Carlos

Esta diapositiva muestra un ejemplo de las acciones realizadas por **SUB1** cuando tiene que invocar a **SUB2**.

En primer lugar, **SUB1** copiará en pila los registros temporales cuyo valor quiere preservar ante la llamada. En esta situación sólo se encuentra **$t3**. La posición que le toca ocupar a la copia de **$t3** en pila es **$fp+24**, así que salvamos su valor en la misma con **sw $t3,24($fp)**.

Los argumentos primero, segundo y cuarto se pasan por copia. Si queremos poner en ellos el contenido de **$t7**, la variable local **x** (que está en la dirección **$fp+28**) y la constante **25** respectivamente, haremos **move $a0,$t7**, **lw $a1,28($fp)** y **li $a3,25**.

El tercer argumento se pasa por referencia. Si suponemos que queremos meter como argumento la variable local **y**, residente en memoria en la posición **$fp+32**, ejecutaremos **addiu $a2,$fp,32**.

El quinto argumento es por copia, pero hay que ponerlo en la pila, en la dirección **$fp+16**. Si queremos pasar como argumento el valor contenido en el registro **$t3**, haremos **sw $t3,16($fp)**.

El sexto y último argumento se pasa por referencia, pero hay que poner el puntero en la pila, en la dirección **$fp+20**. Supongamos que queremos pasarle como argumento un puntero al cuarto argumento que recibió **SUB1**, que se encuentra copiado en memoria en la dirección **$fp+68**. Supongamos también que el registro **$t8** está disponible. Entonces copiaremos dicha dirección en **$t8**, con **addiu $t8,$fp,68**, y después guardaremos el puntero en la pila con **sw $t8,20($fp)**.

Lo siguiente es saltar a la subrutina con enlace. Lo haremos con **jal SUB2**. Tras retornar, recogeremos el valor de retorno desde **$v0**. En el ejemplo, dicho valor se copia en la variable local **x**, ubicada en la dirección de memoria **$fp+28**, así que tendremos que ejecutar **sw $v0,28($fp)**.

Ahora hay que restaurar el contenido de **$t3** desde la pila para continuar con la ejecución normal de **SUB1**. Para ello haremos **lw $t3,36($fp)**.

Índice

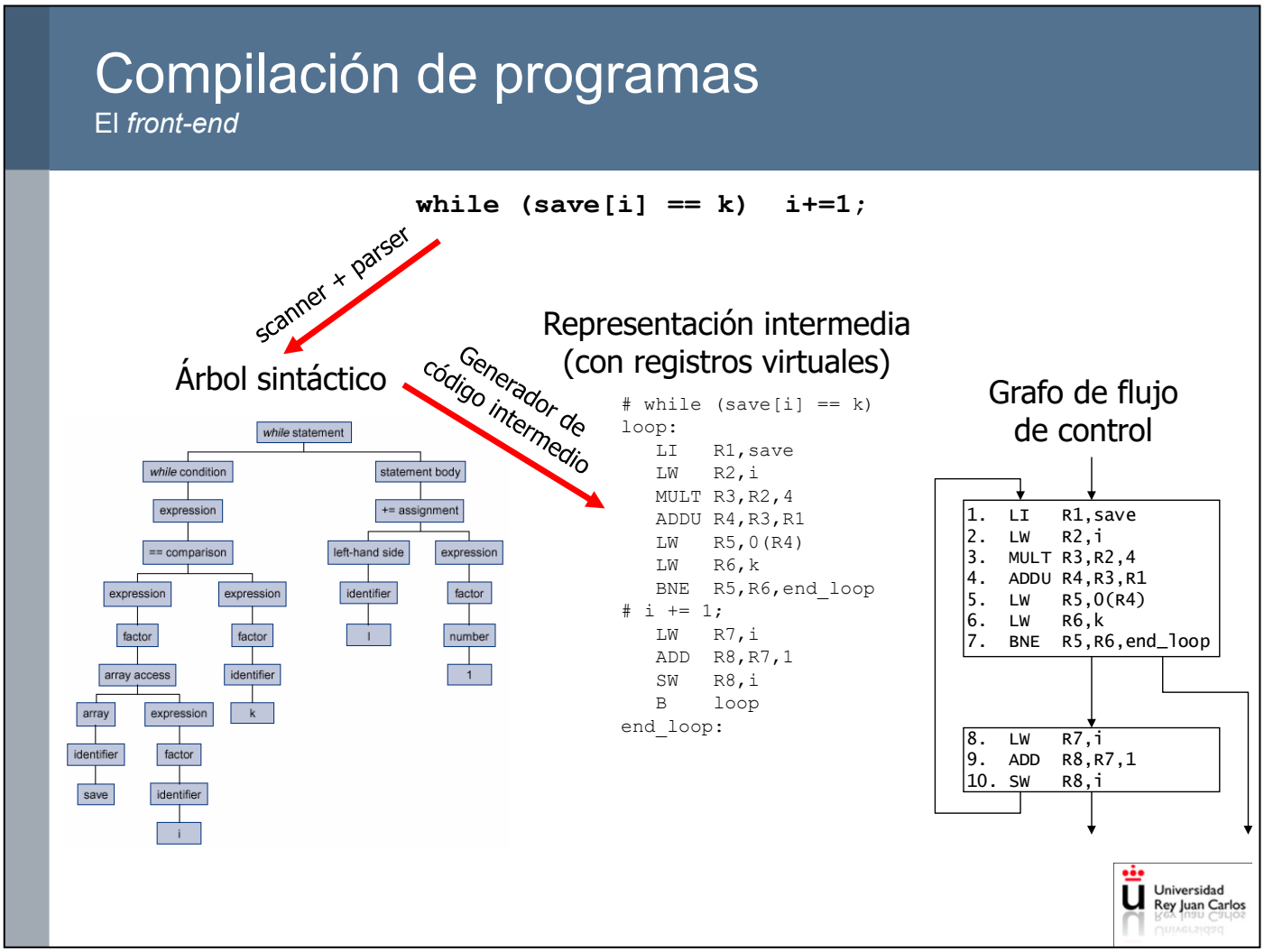
Universidad Rey Juan Carlos

Hasta ahora hemos estudiado cómo se pueden escribir programas en ensamblador, tomando como punto de partida un fragmento de código escrito en lenguaje C. En realidad, estamos realizando la función de un compilador. Las técnicas presentadas hasta ahora conducen a soluciones no óptimas: con esta mecánica escribiremos código correcto, pero no siempre eficiente. Para conseguir código eficiente, los compiladores aplican un amplio abanico de técnicas de **optimización de código**.

Un compilador tiene varios componentes: el ***front-end***, el **optimizador global** y el **generador de código**. El ***front-end*** lee el código fuente, chequea su sintaxis y su semántica y lo traduce a una representación intermedia, que puede ser más o menos próxima al lenguaje ensamblador. Sus funciones son realizadas por los siguientes componentes:

- El **analizador lexicográfico** o ***scanner***, que lee el código fuente y lo divide en ***tokens***, que son entidades básicas con significado propio (palabras reservadas, constantes numéricas o alfanuméricas, nombres definidos por el programador, operadores, etc).
- El **analizador sintáctico** o ***parser***, que toma los tokens, comprueba que son sintácticamente correctos y produce un **árbol sintáctico** (**AS**) que representa la estructura del programa.
- El **analizador semántico**, que toma el **AS**, comprueba la semántica del código por comprobación de tipos en declaraciones y operaciones, y crea la **tabla de símbolos** (**TS**) con los símbolos definidos por el programador, sus valores y sus tipos.
- El **generador de representación intermedia** (**RI**), que toma la **TS** y el **AS** y genera una representación del programa formada por un conjunto reducido de tipos de datos (carácter, entero, real) y de primitivas sencillas sobre ellos (copiar, sumar, comparar, etc). La **RI** es similar al código ensamblador de **MIPS**, pero usa **registros virtuales**, no físicos, ya que es independiente de la máquina objetivo.

(CONTINÚA EN LA PÁGINA SIGUIENTE)



(VIENE DE LA PÁGINA ANTERIOR)

El *front-end* es muy dependiente del lenguaje utilizado para el escribir código fuente, y la **RI** generada representa de forma abstracta la funcionalidad del programa de forma independiente del lenguaje original empleado y de la máquina objetivo.

El código generado se estructura en una sucesión de **bloques básicos** que se ejecutan en secuencia o en estructuras alternativas o de bucle. Un **bloque básico** es un conjunto de sentencias que **siempre** se ejecutan secuencialmente. Un bloque básico no puede contener ninguna etiqueta a la que se salte desde ningún otro lugar del programa, salvo quizá una al principio del todo. Todo bloque básico terminará en una instrucción de salto o bifurcación, o bien justo antes de una instrucción que pueda ser destino de un salto.

Con los bloques básicos se realiza el **grafo de flujo de control**, que tiene una serie de nodos que representan los **bloques básicos** presentes en el código, y de arcos que representan el flujo de control entre los bloques básicos. El grafo de flujo de control es una herramienta fundamental para realizar el análisis del flujo de datos del programa y realizar optimizaciones, y es muy fácil construirlo a partir de la **RI**.

En la diapositiva se muestra un ejemplo de traducción de un pequeño bucle **while** de C a su representación intermedia. En primer lugar, la acción del analizador lexicográfico y el analizador sintáctico genera el árbol sintáctico correspondiente al bucle. Junto con la tabla de símbolos, el generador de código intermedio construye la representación intermedia, en la que aparecen referencias a registros virtuales. La asignación final a registros reales se efectúa en las últimas etapas de la optimización. También se muestra el grafo de flujo de control.

A la hora de realizar la traducción, un compilador debe ser **conservador**, ya que debe proporcionar código que funcione tal como lo espera el programador.

Optimización de código

Optimización de alto nivel

- Fases de la optimización:
  - Optimizaciones de alto nivel.
  - Optimizaciones de código intermedio.
- El programador decide qué clases de optimizaciones realizará el compilador.
- **gcc**:
  - Permite activar cada tipo de optimización individualmente.
  - Agrupa las optimizaciones por niveles: **O0** (sin optimización), **O1** (nivel medio), **O2** (nivel completo), **O3** (nivel completo con integración de procedimientos pequeños).

OPTIMIZACIONES DE ALTO NIVEL		
Nombre	**Explicación**	**Nivel en gcc**
Transformaciones de bucle	Cambios en los bucles para aumentar su velocidad de ejecución.	O1
Desenrollamiento de bucles	Replicar el cuerpo del bucle varias veces para reducir el número de iteraciones y la sobrecarga en su control.	O3
Integración de procedimientos	Reemplazar llamada a procedimiento por el cuerpo del mismo.	O3

Universidad Rey Juan Carlos

Los compiladores aplican diferentes tipos de optimizaciones en diferentes fases de la compilación:

- **Optimizaciones de alto nivel**: se realizan sobre el código fuente en alto nivel, mientras se está generando la representación intermedia del código.
- **Optimizaciones de código intermedio**: se realizan sobre el código intermedio, una vez ha sido totalmente generado por el *front-end*.

Al invocar al compilador, los programadores pueden elegir cuáles son las optimizaciones que deseen. El compilador **gcc** de **GNU** implementa múltiples técnicas de optimización, que el programador puede seleccionar una por una. Como esto resulta algo tedioso, gcc agrupa las optimizaciones por niveles, de modo que activar un nivel implica la activación de múltiples técnicas. Los niveles son:

- **O0**: ninguna optimización.
- **O1**: nivel medio (*medium*)
- **O2**: nivel completo (*full*)
- **O3**: nivel completo con integración de procedimientos pequeños (*full with integration of small procedures*).

Al seleccionar un nivel, **gcc** aplica todas las optimizaciones incluidas en el mismo. Además, **gcc** ofrece flexibilidad total, ya que permite activar individualmente cada tipo de optimización.

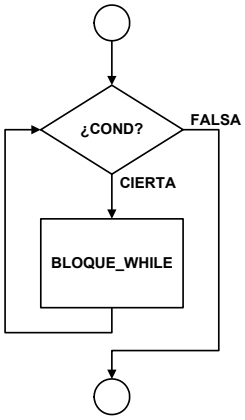
Las optimizaciones de alto nivel son dependientes del lenguaje fuente utilizado al escribir el programa, y son las primeras que se realizan. La más utilizada es la **integración de procedimientos** (*procedure inlining*), que consiste en sustituir la llamada a una función por el cuerpo de la misma, reemplazando los parámetros de llamada por los argumentos que se le hayan pasado. Otras técnicas utilizadas a menudo son las **transformaciones de bucle**, cuyo objetivo es reducir la sobrecarga de los mismos, mejorar los accesos a memoria o aprovechar las características del *hardware*.

Transformaciones de bucle

Ejemplos: inversión de bucle / desenrollamiento de bucle

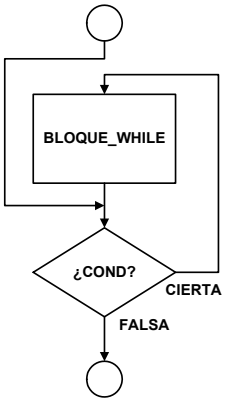
Inversión de bucle: `while (save[i]==k) i+=1;`

Bucle original




```
while_loop:
cond_loop:
   LI   R1,save
   LW   R2,i
   MULT R3,R2,4
   ADDU R4,R3,R1
   LW   R5,0(R4)
   LW   R6,k
   BNE  R5,R6,end_loop
body_loop:
   LW   R7,i
   ADD  R8,R7,1
   SW   R8,i
   B    cond_loop
end_loop:
```

Bucle optimizado 1




```
while_loop:
   B    cond_loop
body_loop:
   LW   R7,i
   ADD  R8,R7,1
   SW   R8,i
cond_loop:
   LI   R1,save
   LW   R2,i
   MULT R3,R2,4
   ADDU R4,R3,R1
   LW   R5,0(R4)
   LW   R6,k
   BEQ  R5,R6,body_loop
end_loop:
```

Desenrollamiento de bucle

```
#define N 12
int v[N], i, suma;

suma = 0;
for (i = 0; i < N; i++)
   suma = suma+v[i];
```

```
#define N 12
int v[N], i, suma;

suma = 0;
for (i = 0; i < N/2; i++) {
   suma = suma+v[i*2];
   suma = suma+v[i*2+1];
}
```

Universidad Rey Juan Carlos

La diapositiva muestra dos ejemplos de optimizaciones por transformación de bucle.

La primera es la **inversión de bucle**, consistente en modificar su estructura para ejecutar menos instrucciones. En el ejemplo, la primera versión del bucle **while** comienza comprobando la condición de permanencia, para después ejecutar el cuerpo del bucle y terminar la iteración con un salto incondicional hacia atrás para volver a comprobar la condición. En cada iteración se ejecutan dos instrucciones de ramificación o salto: una condicional, dependiente del resultado de la evaluación de la condición de permanencia, para eventualmente abandonar el bucle; y otra incondicional hacia atrás para saltar al código que comprueba dicha condición. En la segunda versión, la evaluación de la condición se ha movido al final, de modo que no es preciso incluir en todas las iteraciones un salto incondicional hacia atrás. Eso sí, comprobar la condición es lo primero que hay que hacer al entrar en el bucle, y por ello se incluye un salto incondicional (que se ejecuta una única vez) delante del cuerpo del bucle para evaluar dicha condición antes de empezar a iterar.

La segunda optimización presentada es el **desenrollamiento de bucle**, que consiste en incluir varias veces el cuerpo del bucle en cada iteración, adaptando el código, para así reducir el número de iteraciones, y aligerar la sobrecarga debida al control del bucle. En el ejemplo, el bucle se ha desenrollado una vez, reduciendo el número de iteraciones a la mitad. Esto puede hacerse si el número de iteraciones es conocido, lo que sucede en los bucles de tipo **for**. Además, esta optimización normalmente aumenta la longitud del bloque o bloques básicos interiores al cuerpo del bucle, lo que proporciona oportunidades a otras muchas optimizaciones que puedan aplicarse en las etapas posteriores de la compilación.

Hay otras transformaciones de bucle, como la fisión (parte un bucle en varios), la fusión (une varios bucles en uno único), la reversión de bucle (se cambia de lugar la asignación del índice para evitar dependencias en el código), etc. También hay transformaciones de bucle relacionadas con el aprovechamiento de la caché, la paralelización de código, etc.

Optimización de código intermedio

Algunas técnicas habituales

Nombre	Tipo y nivel en gcc	Explicación
Eliminación de subexpresiones comunes	Local (O1) / Global (O2)	Eliminar repeticiones del mismo cálculo.
Propagación de copia	Local (O1) / Global (O2)	Sustituye por ***X*** todas las apariciones de una variable ***Z*** a la cual se ha asignado ***X*** con anterioridad (ej: *Z=X*), eliminando la necesidad de cargar datos varias veces y facilitando otras optimizaciones.
Propagación de constantes	Local (O1) / Global (O2)	Reemplazar apariciones de una variable a la que se ha asignado una constante por la propia constante.
Plegamiento de constantes	Local (O1) / Global (O2)	Evalúa en compilación expresiones con constantes.
Reducción de la altura de la pila	Local (O1)	Reorganización de expresiones con objeto de minimizar los recursos necesarios para su evaluación.
Eliminación de código muerto	Local (O1) / Global (O2)	Elimina código inalcanzable, redundante o que no afecta al resultado final del programa.
Eliminación de almacenamientos muertos	Local (O1) / Global (O2)	Elimina almacenamientos de variables que no vuelven a utilizarse.
Movimiento de código	Global (O2)	Sacar fuera de un bucle un cálculo que siempre da el mismo valor y se repite en todas las iteraciones.
Eliminación de variables de inducción	Global (O2)	Simplificar o eliminar cálculos de direcciones en vectores o matrices en bucles (a veces usando punteros que se incrementan o decrementan).
Factorización de código	Global (O2)	Sustituir esquemas de cálculo repetidos por llamadas a subrutina.
Eliminación de la comprobación de límites	Local (O1)	Eliminar la comprobación de los índices en accesos a *arrays*.
Reducción de potencia	Dependiente del procesador (O1)	Cambiar operaciones complejas por otras más simples.

Universidad Rey Juan Carlos

Una vez obtenida la representación intermedia, se pueden realizar optimizaciones de tres tipos:

- **Optimizaciones locales**: se realizan dentro de cada bloque básico.
- **Optimizaciones globales**: involucran a varios bloques básicos a la vez, y se basan en el análisis del flujo de datos. Son más complejas que las optimizaciones locales.
- **Optimizaciones dependientes del procesador**: se aplican en función del conocimiento que se tenga sobre las características del procesador, su arquitectura y su repertorio de instrucciones. La más importante es la **asignación de registros**.

Hay técnicas que pueden aplicarse localmente, globalmente o de ambas formas.

La **propagación de copia** es aplicable cuando realizamos asignaciones de variables de tipo **Z=X**, y consiste en sustituir las apariciones de **Z** por **X**. La **propagación de constantes** es útil cuando asignamos una constante a una variable, y consiste en sustituir las referencias a la variable por la propia constante. El **plegamiento de constantes** consiste en que sea el compilador evalúe las expresiones con constantes, y después reemplace las expresiones por el valor obtenido.

A veces las expresiones se evalúan a través de la pila. La **reducción de la altura de la pila** consiste en reorganizar el código para minimizar el espacio de pila utilizado en la evaluación.

La **eliminación de código muerto** elimina el código inalcanzable, repetido o que no afecta al resultado final producido por el programa. Esto incluye la **eliminación de almacenamientos muertos**, que evita almacenar variables que ya no vuelven a utilizarse.

Ciertos lenguajes exigen que en los accesos a *array* se compruebe si el índice está dentro de límites. La **eliminación de la comprobación de límites** se efectúa cuando esté garantizado que los accesos son dentro de límites (por ejemplo en ciertos bucles).

La **factorización de código** sustituye fragmentos de código similares parametrizables por llamadas a subrutina. Esta optimización reduce el tamaño del código, pero no lo hace más rápido.

Optimización de código intermedio

Ejemplo: eliminación de subexpresiones comunes + reducción de potencia

```
x[i] = x[i] + 4;
```

Punto de partida

```
# x[i] + 4
 LI    R100,x
 LW    R101,i
 MULT  R102,R101,4
 ADDU  R103,R100,R102
 LW    R104,0(R103)
# valor de x[i]: en R104
 ADD   R105,R104,4
# x[i] =
 LI    R106,x
 LW    R107,i
 MULT  R108,R107,4
 ADD   R109,R106,R107
 SW    R105,0(R109)
```

Código optimizado 1

```
# x[i] + 4
 LI    R100,x
 LW    R101,i
 MULT  R102,R101,4
 ADDU  R103,R100,R102
 LW    R104,0(R103)
# valor de x[i]: en R104
 ADD   R105,R104,4
# x[i] =
 SW    R105,0(R103)
```

Optimización realizada

Eliminación de subexpresiones comunes

Se ha eliminado el segundo cálculo de la dirección del elemento del vector.

Código optimizado 2

```
# x[i] + 4
 LI    R100,x
 LW    R101,i
 SLL   R102,R101,2
 ADDU  R103,R100,R102
 LW    R104,0(R103)
# valor de x[i]: en R104
 ADD   R105,R104,4
# x[i] =
 SW    R105,0(R103)
```

Optimización realizada

Reducción de potencia

Cambiar la operación **mult** por un desplazamiento.

Universidad Rey Juan Carlos

La diapositiva muestra la traducción de una asignación que incluye un acceso a un elemento de un vector. La primera versión (a la izquierda) comienza con un acceso de lectura al elemento **x[i]**, para después sumarle 4 y realizar un nuevo acceso a **x[i]**, esta vez de escritura. Este código sin optimizar evalúa la expresión **dir(x)+i*4** dos veces. La técnica de **eliminación de subexpresiones comunes** quitaría del código la segunda evaluación, que es a todas luces innecesaria, y dejaría el código tal como aparece en el centro de la diapositiva.

La optimización por **reducción de potencia** es otra técnica habitual, que sustituye operaciones costosas por otras más sencillas. En este caso, tenemos una multiplicación por 4 para calcular la distancia del elemento **v[i]** al origen del vector. Esta multiplicación puede ser reemplazada por un desplazamiento dos lugares hacia la izquierda, obteniéndose el mismo resultado. Así, la operación **MULT** de la **RI** se cambia por **SLL**.

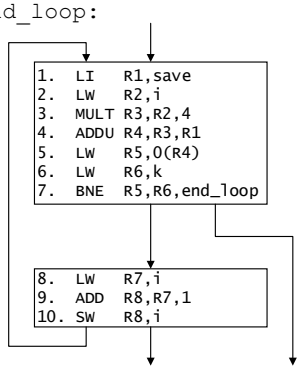
Optimización de código intermedio

Ejemplo: eliminación de variables de inducción / movimiento de código / reducción de potencia

```
while (save[i]==k) i+=1;
```

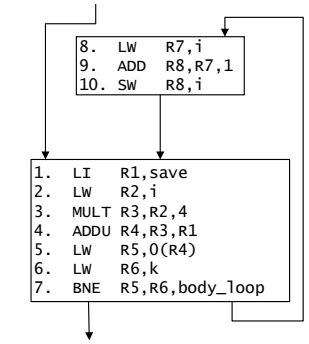
Bucle original

```
# while (save[i] == k)
while_loop:
    LI    R1,save
    LW    R2,i
    MULT  R3,R2,4
    ADDU  R4,R3,R1
    LW    R5,0(R4)
    LW    R6,k
    BNE   R5,R6,end_loop
# i += 1;
    LW    R7,i
    ADD   R8,R7,1
    SW    R8,i
    B     while_loop
end_loop:
```



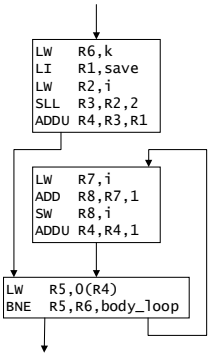
Bucle optimizado 1

```
while_loop:
    B     cond_loop
body_loop:
    LW    R7,i
    ADD   R8,R7,1
    SW    R8,i
cond_loop:
    LI    R1,save
    LW    R2,i
    MULT  R3,R2,4
    ADDU  R4,R3,R1
    LW    R5,0(R4)
    LW    R6,k
    BEQ   R5,R6,body_loop
end_loop:
```



Bucle optimizado 2

```
while_loop:
      LW    R6,k
      LI    R1,save
      LW    R2,i
      SLL   R3,R2,2
      ADDU  R4,R3,R1
      B     cond_loop
body_loop:
      LW    R7,i
      ADD   R8,R7,1
      SW    R8,i
      ADDU  R4,R4,4
cond_loop:
      LW    R5,0(R4)
      BEQ   R5,R6,body_loop
end_loop:
```



Universidad Rey Juan Carlos

Volvamos al ejemplo del bucle sobre el que se había realizado la inversión. La versión original está a la izquierda, y la versión optimizada por inversión de bucle queda en el centro. Sobre esta versión aún pueden aplicarse varias optimizaciones.

La primera es la **eliminación de variables de inducción**, que es una combinación de transformaciones que reduce la sobrecarga a la hora de indexar *arrays* en bucles, y que consiste habitualmente en reemplazar la indexación por accesos mediante puntero. En este bucle se recorre un vector a partir de una variable **i**. En cada iteración se calcula la posición del elemento i-ésimo poniendo un puntero al vector, multiplicando **i** por 4 y sumándoselo al puntero para acceder así al elemento buscado. El recorrido por el vector se puede simplificar, sin más que poner un puntero al primer elemento que se consulta (**R4**) fuera del bucle, e ir actualizándolo en cada pasada mediante una suma (en recorridos ascendentes) o una resta (en recorridos descendentes). En ocasiones, esta técnica conduce a que la manipulación del índice del bucle (la **variable de inducción**) se convierta en código muerto que puede ser eliminado. No es así, en este caso, ya que la **i** es una variable cuyo valor nos interesa al salir del bucle.

Es evidente que aquí podemos aplicar de nuevo **reducción de potencia**, sustituyendo el producto de **i** por 4 por un desplazamiento a la izquierda de longitud 2.

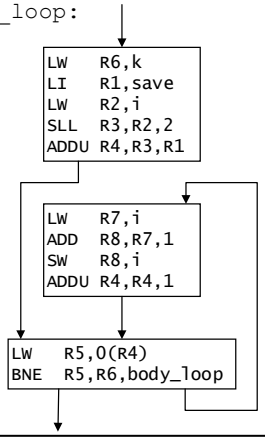
El **movimiento de código** es otra optimización relacionada con los bucles, y que consiste en sacar de un bucle sus **invariantes**, que son cálculos o asignaciones repetidos que se encuentran dentro del bucle y que siempre proporcionan el mismo resultado. Por tanto, es mejor sacarlos delante del bucle y ejecutarlos una única vez. Si hacemos antes la eliminación de variables de inducción, la única operación candidata para ser movida es la carga de la variable **k** sobre **R6**. Si la hubiéramos realizado antes, otro invariante del bucle habrá sido la carga de la dirección inicial del vector en **R1**.

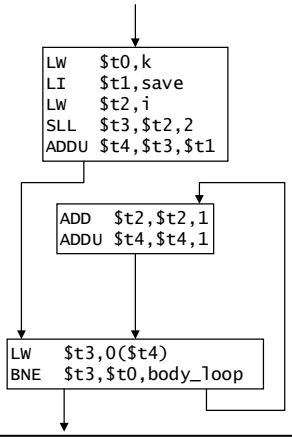
Asignación de registros y generación de código

```
while_loop:
  LW   R6,k
  LI   R1,save
  LW   R2,i
  SLL  R3,R2,2
  ADDU R4,R3,R1
  B    cond_loop
body_loop:
  LW   R7,i
  ADD  R8,R7,1
  SW   R8,i
  ADDU R4,R4,4
cond_loop:
  LW   R5,0(R4)
  BEQ  R5,R6,body_loop
end_loop:
```

```
while_loop:
  LW   $t0,k
  LI   $t1,save
  LW   $t2,i
  SLL  $t3,$t2,2
  ADD  $t4,$t3,$t1
  B    cond_loop
body_loop:
  ADD  $t2,$t2,1
  ADDU $t4,$t4,4
cond_loop:
  LW   $t3,0($t4)
  BEQ  $t3,$t0,body_loop
end_loop:
```

```
while_loop:
  lw   $t0,k
  la   $t1,save
  lw   $t2,i
  sll  $t3,$t2,2
  add  $t4,$t3,$t1
  b    cond_loop
body_loop:
  addi $t2,$t2,1
  addu $t4,$t4,4
cond_loop:
  lw   $t3,0($t4)
  beq  $t3,$t0,body_loop
end_loop:
```





Asignación de registros

- R1: **$t1**
- R2, R7 y R8 (variable **i**): **$t2**
- R3 y R5: **$t3** (**reutilizado**)
- R4: **$t4**
- R6: **$t0**

Se descarta **SW R8,i** (posible almacenamiento muerto).

Universidad Rey Juan Carlos

En las arquitecturas de carga / almacenamiento, la **asignación de registros** (***register allocation***) es la optimización más importante. Su objetivo es sustituir los registros virtuales generados en la **RI** por registros reales del procesador, minimizando el número de instrucciones de carga y almacenamiento del programa. Puede realizarse a nivel local o global, siempre dentro de una misma función.

La asignación de registros se basa en el **análisis de los rangos vivos** de la variables. El **rango vivo de una variable** es la región de código en la que la variable está en uso. Si la intersección de los rangos vivos de dos variables es vacía, ambas pueden compartir un único registro, ya que no estarán nunca en uso a la vez. Sin embargo, si la intersección no es vacía, ambas no pueden compartir el mismo registro, pues en la zona de la intersección las dos variables están en uso al mismo tiempo. Con los rangos vivos de las variables se crea un **grafo de interferencia**, en el que cada nodo representa una variable. Dos variables con intersección de rango vivo no vacía se conectan mediante un arco. Después de trazar todos los arcos se ejecuta un algoritmo de **coloración de grafos**, técnica que asigna etiquetas (colores) distintos a nodos adyacentes. Al final, cada color representa un registro del procesador, y nodos con igual color referencian variables que pueden compartir un registro. Si el número de colores es menor o igual que el número de registros disponibles, todas las variables se pueden ubicar en registros. Cuando el número de colores es mayor que el número de registros disponibles, será preciso dividir en varios nodos el rango vivo de una o más variables, rehacer el grafo de interferencia y colorearlo de nuevo.

Tras realizar las optimizaciones viene la fase **de generación de código**, en la cual se realiza la selección de instrucciones y se aplican aún las optimizaciones dependientes de la máquina (plegamiento de constantes, planificación de instrucciones, etc). Esta fase puede generar directamente código objeto, o bien puede generar código en lenguaje de ensamble y después lanzar la ejecución de un programa ensamblador.