

TEMA 2

EL LENGUAJE DEL COMPUTADOR



Conjunto de instrucciones

- Repertorio de instrucciones del computador
- Computadores distintos tienen diferentes conjuntos de instrucciones
 - (Pero con muchos aspectos en común)
- Los primeros computadores tenían un conjunto de instrucciones muy simple
 - Simplificación de la implementación
- Muchos computadores modernos también tienen un sencillo conjunto de instrucciones

El conjunto de instrucciones de MIPS

- Se usará como ejemplo en la asignatura
- Comercializado actualmente por *Wave Computing*:
<https://www.mips.com/>
- Muy utilizado como computador empotrado
 - Electrónica de consumo, redes, almacenamiento, cámaras, impresoras, etc.
- Similar a otras arquitecturas de repertorio de instrucciones (*ISA = instruction set architecture*)

Operaciones Aritméticas

- Suma y resta: tres operandos
 - Dos operandos fuente
 - Un operando destino (modificado por la instrucción)
 $\text{add } a, b, c \quad \# a \leftarrow b + c$
- Todas las operaciones aritméticas tienen esta misma forma

1er. principio de diseño:

La simplicidad favorece la regularidad

- La regularidad simplifica la implementación.
- La simplicidad mejora el rendimiento a menor coste

Ejemplo aritmético

- Código de alto nivel (C):

$$f = (g + h) - (i + j);$$

- Código compilado para MIPS:

```
add t0, g, h    # temp t0 = g + h  
add t1, i, j    # temp t1 = i + j  
sub f, t0, t1  # f = t0 - t1
```

- Las variables f, g, h, i y j se alojarán en registros de la arquitectura MIPS como veremos a continuación.

Operandos en registros

- Las instrucciones aritméticas sólo usan operandos en registros
- MIPS tiene 32 registros de 32 bits cada uno (32×32 bits)
 - Se utilizan para datos de uso frecuente
 - Se numeran de 0 a 31
 - A un dato de 32 bits se le llama “palabra”
- Nombres de los registros en ensamblador:
 - `$t0, $t1, ..., $t9` para datos temporales
 - `$s0, $s1, ..., $s7` para variables salvadas en las llamadas a funciones

2º principio de diseño:

Cuanto más pequeño, más rápido

- Los registros tienen muy poca capacidad pero son mucho más rápidos que la memoria principal.

Ejemplo de operandos en registros

- Código de alto nivel (C):

$$f = (g + h) - (i + j);$$

○ Se supone que las variables f , g , h , i y j están en los registros $\$s0$, $\$s1$, $\$s2$, $\$s3$ y $\$s4$, respectivamente.

- Código real compilado para MIPS:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Acceso a memoria: *load*

Ejemplo (leer, cargar):

- Coger bola de cajón o y dejarla en la mano:
 - ▣ Mano \leftarrow Cajonera[Cajón 0]
 - Coger dato de memoria de la posición o y dejarlo en un registro:
 - ▣ Registro \leftarrow Memoria[0]



Diferencia:

AI final:

- La bola está solo en la mano
 - El dato está en el registro y en la memoria



Acceso a memoria: *store*

Ejemplo (almacenar/escribir/guardar):

- Dejar la bola en el último cajón:
 - ✖ Cajonera[Cajón $n-1$] ← Mano
- Guardar el dato del registro en la última posición de memoria:
 - ✖ Memoria[$n-1$] ← Registro



Registro CPU

Diferencia:

Al final:

- La bola está solo en el cajón
- El dato está en el registro y en la memoria



Operandos en memoria

- La memoria principal se usa para datos múltiples:
 - Vectores, matrices, estructuras, datos dinámicos, etc.
- Para realizar operaciones aritméticas con estos datos:
 - Se cargan los valores de la memoria a los registros (*load*)
 - Se efectúa la operación sobre los registros
 - Se almacena el registro de resultado en memoria (*store*)
- La mínima información de memoria accesible es un byte
 - Cada dirección identifica un dato de 8 bits (byte)
- Las palabras (4 bytes) están alineadas en memoria
 - Sus direcciones tienen que ser múltiplos de 4

Instrucciones de acceso a memoria en MIPS

- Cargar una palabra de memoria en un registro (*load*)
 - **lw rt, d(rs)**
 - *rs, rt*: dos registros, *rs* se suele llamar registro base
 - *d*: desplazamiento
 - Carga la palabra contenida en la dirección de memoria *rs + d* en el registro *rt*
- Almacenar el contenido de un registro en memoria (*store*)
 - **sw rt, d(rs)**
 - *rs, rt*: dos registros, *rs* se suele llamar registro base
 - *d*: desplazamiento
 - Almacena el contenido del registro *rt* en la dirección de memoria *rs + d*

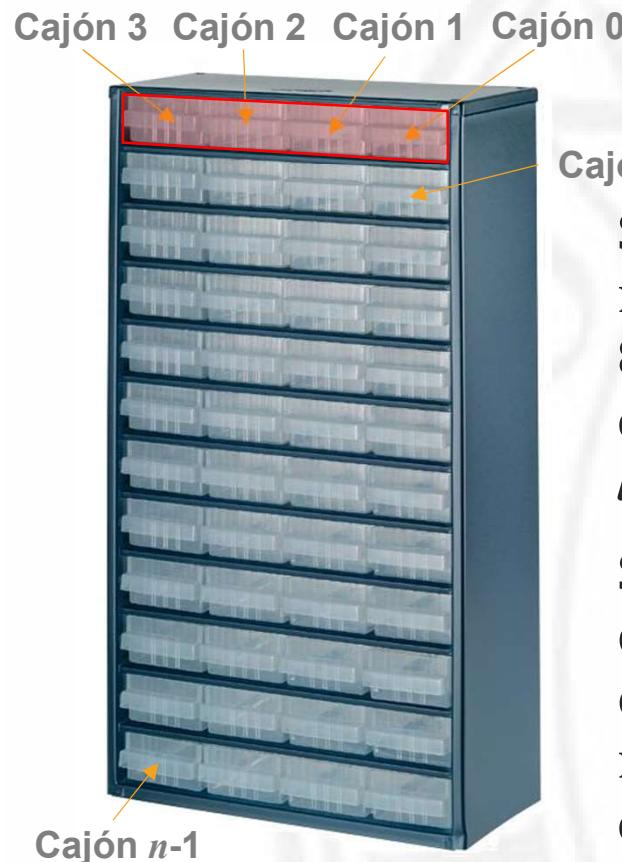
Little-endian y big-endian (I)

Cuando se lee de memoria un objeto de varios bytes, por ejemplo un entero (palabra, que ocupa 4 bytes) hay que especificar el orden en que se toman esos bytes:

- ***Little-endian***: el byte menos significativo se almacena en la dirección más baja.
- ***Big-endian***: el byte menos significativo se almacena en la dirección más alta.
- MIPS puede trabajar de las dos maneras (***bi-endian***) aunque por defecto es ***big-endian***.
- Si solo trabajamos con palabras no notaremos la diferencia.
- Los objetos que solo ocupan un byte no están afectados por el ***endianness***.

Little-endian y big-endian (II)

Little-endian



Si en la dirección 0 de memoria se lee un entero, estará formado por los bytes de las direcciones 0, 1, 2 y 3.

Si el byte de la dirección 0 (la más baja de las 4) contiene los 8 bits de orden más bajo del entero, se sigue el convenio ***little-endian***.

Si, por el contrario, el byte de la dirección 3 (la más alta) contiene los 8 bits de orden más bajo del entero, se seguiría el convenio ***big-endian***.

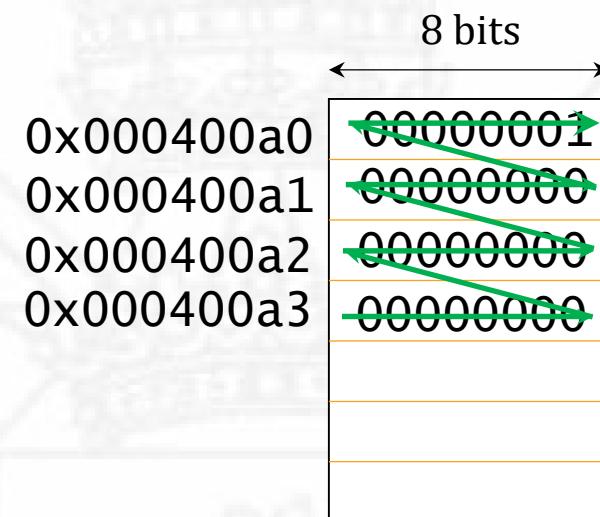
Big-endian



Ejemplo *little-endian*

- Supongamos que a partir de la dirección de memoria 0x000400a0 se encuentran los bytes 0x01, 0x00, 0x00 y 0x00, que \$s1 contiene 0x000400a0 y que ejecutamos la instrucción `lw $t2, 0($s1)`

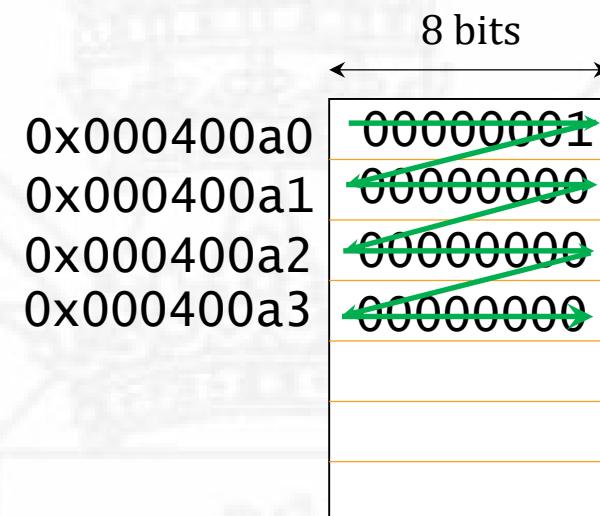
Con el convenio *little-endian* el **byte menos significativo** está en la **dirección más baja** (en nuestro caso 0x000400a0) por lo que, después de la ejecución de la instrucción, \$t2 contendrá $0x00000001 = 1$



Ejemplo *big-endian*

- Supongamos que a partir de la dirección de memoria 0x000400a0 se encuentran los bytes 0x01, 0x00, 0x00 y 0x00, que \$s1 contiene 0x000400a0 y que ejecutamos la instrucción lw \$t2, 0(\$s1)

Con el convenio *big-endian* el **byte menos significativo** está en la **dirección más alta** (en nuestro caso 0x000400a3) por lo que, después de la ejecución de la instrucción, \$t2 contendrá $0x1000000 = 2^{24}$

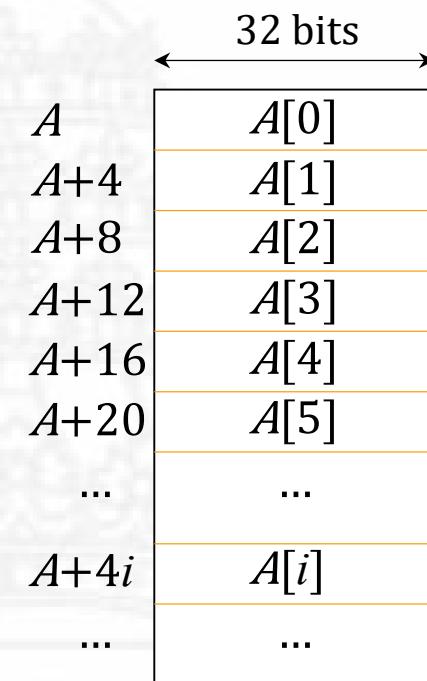


Almacenamiento de vectores en memoria

- Las componentes de un vector se almacenan en posiciones de memoria consecutivas

- Supongamos un vector de enteros de 32 bits que comienza en la dirección A

- 4 bytes por componente, ya que $8 * 4 = 32$
- La primera componente estará en la dirección A
- La segunda estará en la dirección $A + 4$
- La componente i estará en la dirección $A + 4i$



Ejemplo 1 de operando en memoria

- Código de alto nivel (C):

$g = h + A[8];$

○ g está $\$s1$, h en $\$s2$

○ Dirección de comienzo del vector A en $\$s3$

- Código compilado para MIPS:

○ El índice 8 necesita un desplazamiento de 32

(4 bytes por palabra, $8 * 4 = 32$)

```
lw $t0, 32($s3)      # lw: carga palabra  
add $s1, $s2, $t0
```

Desplazamiento

Registro base

Ejemplo 2 de operando en memoria

- Código de alto nivel (C): $A[12] = h + A[8];$
 - h en $\$s2$
 - Dirección de comienzo del vector A en $\$s3$
- Código compilado para MIPS:
 - El índice 8 necesita un desplazamiento de 32 ($8 * 4 = 32$)
 - El índice 12 necesita un desplazamiento de 48 ($12 * 4 = 48$)

```
lw $t0, 32($s3) # lw: carga palabra
                  # $t0 ← A[8]
add $t0, $s2, $t0 # $t0 ← h + A[8]
sw $t0, 48($s3) # sw: almacena palabra
                  # A[12] ← $t0 (= h + A[8])
```

Registros frente a memoria

- El acceso a los registros es más rápido que el acceso a memoria
- Operar con datos en memoria necesita cargas (*load*) y almacenamientos (*store*)
 - Deben ejecutarse más instrucciones y más lentas
- El compilador debe usar los registros siempre que pueda
 - Dejar en memoria sólo las variables menos frecuentemente usadas y las estructuras de datos (vectores, matrices, etc.)
 - Es importante optimizar el uso de registros con el fin de dejarlos disponibles para nuevas variables

Operandos inmediatos

- Son datos constantes especificados en la misma instrucción

```
addi $s3, $s3, 4
```

- MIPS no tiene resta de constantes (resta inmediata)
o Basta usar una constante negativa:

```
addi $s2, $s1, -1
```

La constante 0

- El registro 0 (\$zero) de MIPS es la constante 0
 - No se puede escribir en ese registro
- Útil para operaciones muy frecuentes
 - Ejemplo: copiar el contenido de un registro a otro (**move**)
`add $t2, $s1, $zero`

3^{er} principio de diseño:

Mejorar en lo posible los casos más frecuentes

- Las constantes pequeñas son muy comunes en los programas
- Los operandos inmediatos evitan una instrucción de carga
- El 0 es un operando muy frecuente

Codificación de las instrucciones

- Las instrucciones se codifican en binario
 - Es el llamado código maquina
- Instrucciones de MIPS
 - Se codifican en palabras de 32 bits
 - Pocos formatos para codificar las operaciones, números de registros, etc.
 - Mucha regularidad
- Numeración de los registros de MIPS
 - \$zero es el registro 0 (que siempre contiene 0)
 - \$t0 – \$t7 son los registros 8 al 15
 - \$t8 – \$t9 son los registros 24 al 25
 - \$s0 – \$s7 son los registros 16 al 23

Formato de instrucción R en MIPS

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>rd</i>	<i>shamt</i>	<i>funct</i>
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Campos de la instrucción:

- *op*: código de operación (*opcode*)
- *rs*: número del primer registro fuente
- *rt*: número del segundo registro fuente
- *rd*: número del registro destino
- *shamt*: número de desplazamientos (ahora no usado: 00000)
- *funct*: código de función (código de operación extendido)

Ejemplo de formato R

add $rd, rs, rt \quad (rd \leftarrow rs + rt)$

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>rd</i>	<i>shamt</i>	<i>funct</i>
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add \$t0, \$s1, \$s2 ($\$t0 \leftarrow \$s1 + \$s2$)

especial	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

0000 0010 0011 0010 0100 0000 0010 0000₍₂₎ = 0x02324020

Formato de instrucción I en MIPS

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>constante/desplazamiento</i>
6 bits	5 bits	5 bits	16 bits

- Instrucciones aritmético-lógicas inmediatas y de carga y almacenamiento (*load/store*)
 - *rt*: número de registro fuente o destino
 - *constante*: desde -2^{15} a $+2^{15} - 1$
 - *desplazamiento*: cantidad que se suma a la dirección base situada en *rs*

4º principio de diseño:

Un buen diseño requiere buenas soluciones de compromiso

- Diferentes formatos complican la decodificación, pero permiten una palabra de instrucción uniforme de 32 bits
- Se deben dejar formatos lo más similares posible

Ejemplo de formato I (I)

addi *rt*, *rs*, *cte.* (*rt* ← *rs* + *cte.*)

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>constante/desplazamiento</i>
6 bits	5 bits	5 bits	16 bits

addi \$t0, \$s1, 12 (\$t0 ← \$s1 + 12)

addi	\$s1	\$t0	<i>constante</i>
8	17	8	12
001000	10001	01000	0000000000001100

0010 0010 0010 1000 0000 0000 0000 1100₍₂₎ = 0x2228000c

Ejemplo de formato I (II)

`lw rt, despl.(rs) (rt ← Mem[rs + despl.])`

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>constante/desplazamiento</i>
6 bits	5 bits	5 bits	16 bits

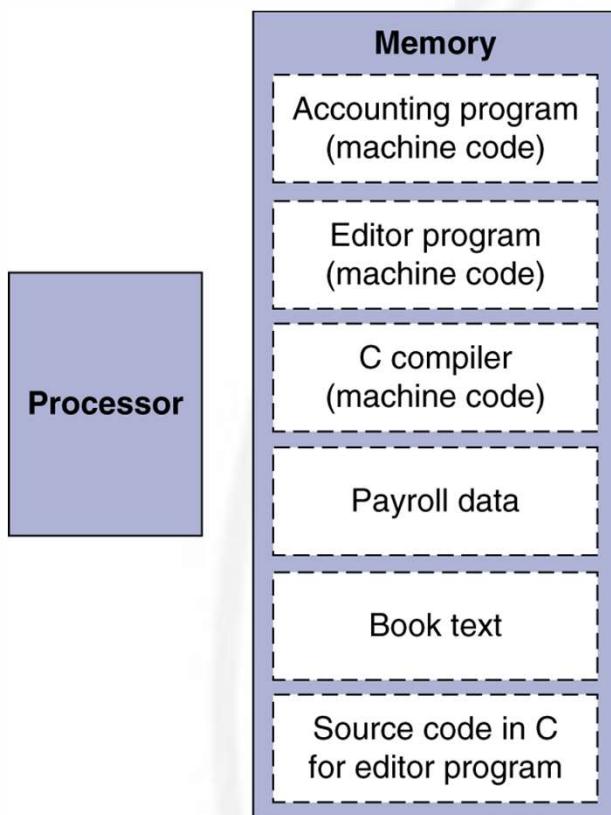
`lw $t0,12($s1) ($t0 ← Mem[$s1 + 12])`

<i>lw</i>	<i>\$s1</i>	<i>\$t0</i>	<i>desplazamiento</i>
35	17	8	12
100011	10001	01000	0000000000001100

`1000 1110 0010 1000 0000 0000 0000 1100(2) = 0x8e28000c`

Computadores de programa almacenado

Recuadro importante



- Las instrucciones se representan en binario, igual que los datos
- Ambos (instrucciones y datos) se almacenan en memoria
- Los programas pueden trabajar con otros programas (como datos)
 - ej.: compiladores, cargadores, etc.
- La compatibilidad binaria permite a los programas compilados operar en computadores diferentes
 - ISAs estandarizadas

Operaciones lógicas

Instrucciones para la manipulación de bits

Operación	C	Java	MIPS
Desplazamiento a la izquierda	<<	<<	sll
Desplazamiento a la derecha	>>	>>>	srl
AND de bits	&	&	and, andi
OR de bits			or, ori
NOT de bits	~	~	nor

Sirven para extraer e insertar grupos de bits en una palabra

Operaciones de desplazamiento

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>rd</i>	<i>shamt</i>	<i>funct</i>
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *shamt*: número de lugares a desplazar
- Desplazamiento lógico a la izquierda (**sll**)
 - Desplaza a la izquierda y rellena con 0's
 - **sll** desplazando i lugares multiplica por 2^i
- Desplazamiento lógico a la derecha (**srl**)
 - Desplaza a la derecha y rellena con 0's
 - **srl** desplazando i lugares divide por 2^i
(sólo para números representados en binario natural)

Operación AND

Efectúa la operación lógica AND bit a bit entre los operandos
Seleccionando ciertos bits, no los cambia y pone los demás a 0

and \$t0, \$t1, \$t2

	\$t2: dato	\$t1: máscara	
\$t2	0000 0000 0000 0000 0000	1101 1100 0000	
\$t1	0000 0000 0000 0000 0011	1100 0000 0000	
\$t0	0000 0000 0000 0000 0000	1100 0000 0000	

Operación OR

Efectúa la operación lógica OR bit a bit entre los operandos
Seleccionando ciertos bits, los pone a 1, sin cambiar los demás

or \$t0, \$t1, \$t2

\$t2: dato

\$t2	0000 0000 0000 0000 0000	1101 1100 0000
------	--------------------------	----------------

\$t1: máscara

\$t1	0000 0000 0000 0000 0011	1100 0000 0000
------	--------------------------	----------------

\$t0	0000 0000 0000 0000 0011	1101 1100 0000
------	--------------------------	----------------

Operación XOR

Efectúa la operación lógica OR exclusivo bit a bit entre los operandos

Seleccionando ciertos bits, los complementa, sin cambiar los demás

`xor $t0, $t1, $t2`

\$t2: dato

\$t2	0000 0000 0000 0000 0000	1101 1100 0000
------	--------------------------	----------------

\$t1: máscara

\$t1	0000 0000 0000 0000 0011	1100 0000 0000
------	--------------------------	----------------

\$t0	0000 0000 0000 0000 0011	0011 1100 0000
------	--------------------------	----------------

Propiedades de la operación XOR

La operación XOR tiene interesantes propiedades, algunas de ellas pueden deducirse de su tabla de verdad:

- Mediante la operación XOR se pueden complementar bits:
 - $x \oplus 0 = x$ $x \oplus 1 = \bar{x}$
- XOR es el complemento de la igualdad, por lo que se puede emplear para conocer los bits diferentes entre dos registros:
 - $x \oplus y = !(x == y)$ o $x \oplus y = (x != y)$
- Cuando se opera dos veces con la misma variable, regenera el valor original:
 - $(x \oplus y) \oplus y = x$

Operación NOT

Sirve para invertir bits en una palabra

- o Cambia 0 por 1 y 1 por 0
- MIPS posee la instrucción NOR de 3 operandos

o $a \text{ NOR } b == \text{NOT}(a \text{ OR } b)$

`nor $t0, $t1, $zero`

Registro 0: siempre contiene cero

`$t1` 0000 0000 0000 0000 0011 1100 0000 0000

`$t0` 1111 1111 1111 1111 1100 0011 1111 1111

Utilidades de las operaciones lógicas (I)

Incluir bits de una palabra

Supongamos que queremos poner los bits 13 a 10 de **\$t2** con el valor almacenado en **\$t3**, que ocupa solo 4 bits, sin cambiar el resto de **\$t2**

\$t3	0000 0000 0000 0000 0000 0000 0000 1110
\$t2	0000 0100 1011 0111 1110 1101 1101 0101

Utilidades de las operaciones lógicas (I)

Incluir bits de una palabra

Supongamos que queremos poner los bits 13 a 10 de **\$t2** con el valor almacenado en **\$t3**, que ocupa solo 4 bits, sin cambiar el resto de **\$t2**

Pasos:

- Poner a 0 los bits implicados de **\$t2**, para lo que colocamos en **\$t1** la máscara 1111 1111 1111 1111 1100 0011 1111 1111 y efectuamos la operación AND con ella
- Desplazar **\$t3** la diferencia de lugares a la izquierda
- Efectuar la operación OR entre los resultados de las operaciones anteriores

and \$t2, \$t1, \$t2

\$t3 0000 0000 0000 0000 0011 1000 0000 0000

sll \$t3, \$t3, 10

\$t2 0000 0100 1011 0111 1111 1001 1101 0101

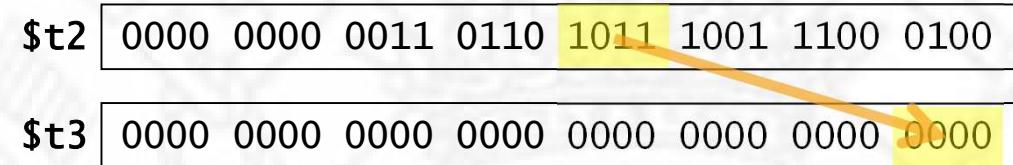
or \$t2, \$t3, \$t2

\$t1 1111 1111 1111 1111 1100 0011 1111 1111

Utilidades de las operaciones lógicas (II)

Extraer bits de una palabra

Supongamos que queremos extraer los bits 15 a 12 de \$t2 y depositarlos en \$t3



Utilidades de las operaciones lógicas (II)

Extraer bits de una palabra

Supongamos que queremos extraer los bits 15 a 12 de **\$t2** y depositarlos en **\$t3**

Pasos:

- Enmascarar el resto de bits de **\$t2**, para ello cargamos **\$t1** con la máscara 0000 0000 0000 0000 1111 0000 0000 0000 y efectuamos la operación AND con ella dejando el resultado en **\$t3**
- Desplazar **\$t3** a la derecha la diferencia de lugares

and \$t3, \$t1, \$t2

srl \$t3, \$t3, 12

\$t2 0000 0000 0011 0110 1011 1001 1100 0100

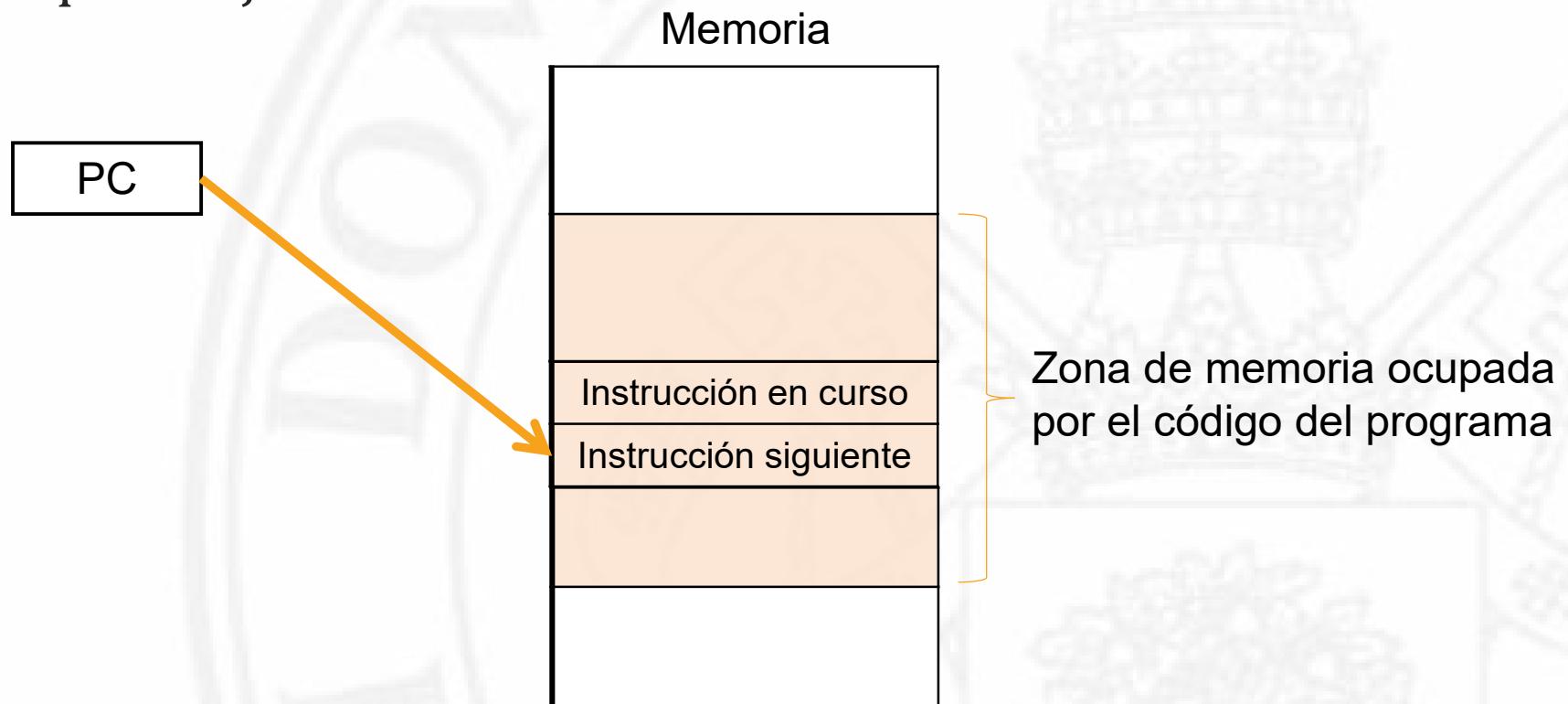
\$t3 0000 0000 0000 0000 0000 0000 0000 1011

\$t1 0000 0000 0000 0000 1111 0000 0000 0000

El contador de programa (PC)

Es un registro de control del computador

- Contiene la dirección de la siguiente instrucción del programa que se ejecutará



Bifurcaciones condicionales y saltos

Bifurcaciones condicionales: saltan si la condición se cumple

o Si no, continúan con la instrucción siguiente

■ **beq rs, rt, L1**

Bifurca a la instrucción con la etiqueta *L1* si (*rs* == *rt*);

■ **bne rs, rt, L1**

Bifurca a la instrucción con la etiqueta *L1* si (*rs* != *rt*)

Salto incondicional

■ **j L1**

Salta incondicionalmente a la instrucción con la etiqueta *L1*

Compilación de la instrucción if

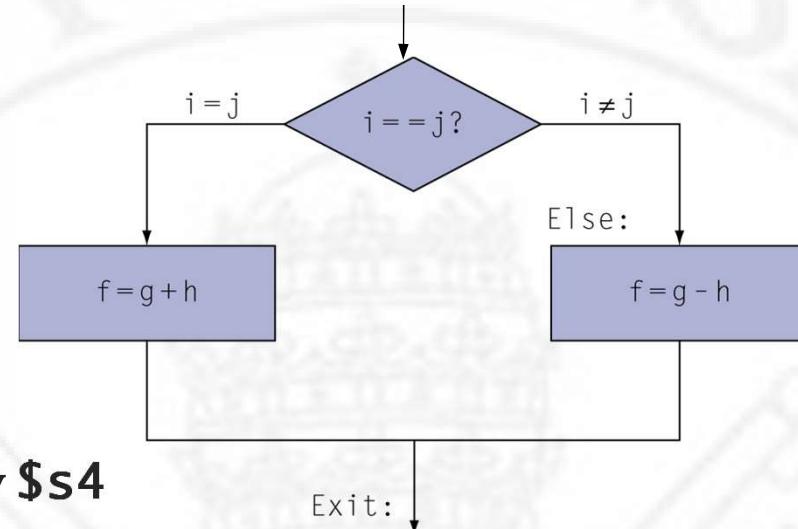
- Código de alto nivel (C):

```
if (i==j)
    f = g+h;
else
    f = g-h;
```

o f, g, h, i y j en $\$s0, \$s1, \$s2, \$s3$ y $\$s4$

- Código ensamblador MIPS:

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else: sub $s0, $s1, $s2
Exit: ...
```



El ensamblador
calcula las direcciones

Compilación de bucles

- Código de alto nivel (C):

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

o i en \$s3, k en \$s5, dirección inicial de save en \$s6

- Código ensamblador MIPS:

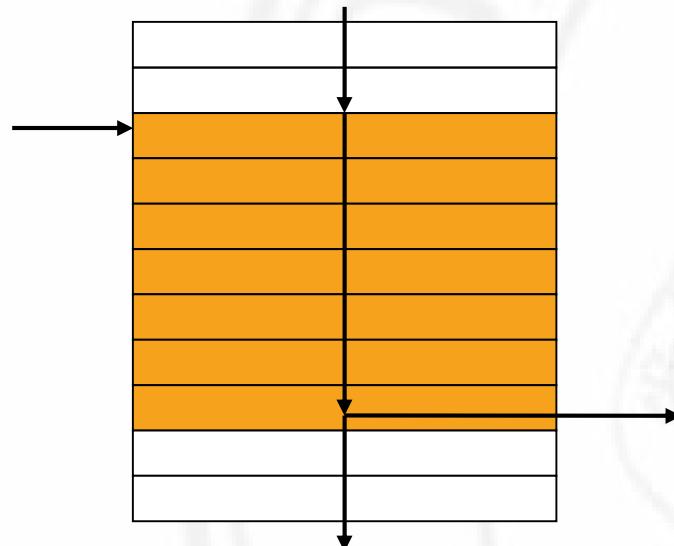
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
```

Exit: ...

Bloques básicos

- Un bloque básico es una secuencia de instrucciones con las siguientes condiciones:

- No tiene bifurcaciones (salvo al final)
- No es destino de bifurcaciones (salvo al principio)



- Los compiladores identifican los bloques básicos para optimizar el código
- Un procesador avanzado puede acelerar la ejecución de bloques básicos

Más instrucciones condicionales

Ponen el resultado a 1 si la condición se cumple

o si no, lo ponen a 0

■ **slt rd, rs, rt**

o if ($rs < rt$) $rd = 1$; else $rd = 0$;

■ **slti rt, rs, constante**

o if ($rs < \text{constante}$) $rt = 1$; else $rt = 0$;

■ Se usan en combinación con **beq** y **bne**

s1t \$t0, \$s1, \$s2 #	si $(\$s1 < \$s2)$
bne \$t0, \$zero, L #	bifurca a L

Diseño de las instrucciones de bifurcación

¿Por qué no hay `blt`, `bge`, etc en MIPS?

- El hardware para detectar $<$, \geq , ... es más lento que para detectar $=$ o \neq
 - Requería más trabajo por instrucción, lo que nos llevaría a un reloj más lento
 - ¡Todas las instrucciones resultarían penalizadas!
- `beq` y `bne` son los casos más frecuentes
- Es una buena solución de compromiso
(4º principio de diseño)

Comparaciones con signo y sin signo

- Comparaciones con signo: `slt`, `slti`
- Comparaciones sin signo: `sltu`, `sltui`
- Ejemplo

`$s0 = 1111 1111 1111 1111 1111 1111 1111 1111 1111`

`$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`

oslt \$t0, \$s0, \$s1 # con signo

- $-1 < +1 \Rightarrow \$t0 = 1$

osltu \$t0, \$s0, \$s1 # sin signo

- $+4.294.967.295 > +1 \Rightarrow \$t0 = 0$

Operaciones sobre bytes y semipalabras

- Podrían usarse las operaciones de bits
- MIPS dispone de instrucciones de carga y almacenamiento de byte y semipalabra

El proceso de cadenas es un caso frecuente:

$l\text{b } rt, \text{ desp1}(rs) \quad l\text{h } rt, \text{ desp1}(rs)$

Extiende el signo a 32 bits en rt

$l\text{bu } rt, \text{ desp1}(rs) \quad l\text{hu } rt, \text{ desp1}(rs)$

Extiende 0's a 32 bits en rt

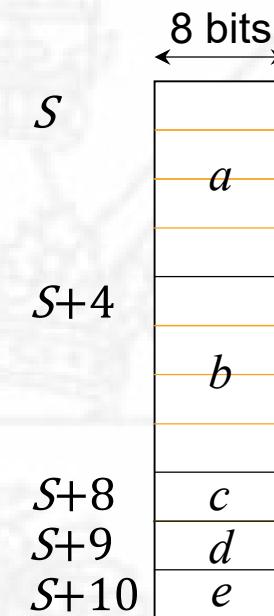
$sb \ rt, \text{ desp1}(rs) \quad sh \ rt, \text{ desp1}(rs)$

Almacena sólo el byte/semipalabra de menos peso de rt

Almacenamiento de estructuras de datos en memoria (I)

- Las estructuras de datos (*struct, class*) son similares a los vectores, pero difieren de estos en que las componentes (campos) pueden tener tamaños diferentes.
 - Cada campo se identifica por un desplazamiento respecto a la dirección inicial
- Ejemplo:

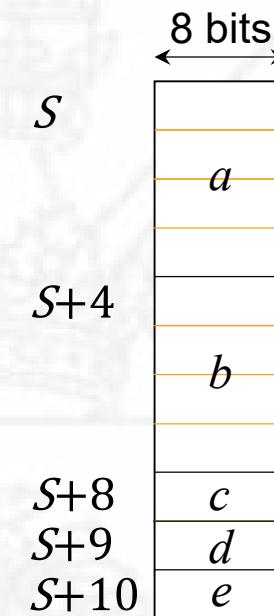
```
struct S {  
    int a;    // Desplazamiento: 0  
    int b;    // Desplazamiento: 4  
    char c;   // Desplazamiento: 8  
    char d;   // Desplazamiento: 9  
    char e;} // Desplazamiento: 10
```



Almacenamiento de estructuras de datos en memoria (II)

- En el ejemplo anterior, supongamos que en el registro \$s0 tenemos la dirección de comienzo de la estructura, S. Si se quisiera cargar el valor de los diferentes campos a registros MIPS, lo haríamos de la siguiente forma:

lw \$t0, 0(\$s0)	# a -> \$t0
lw \$t1, 4(\$s0)	# b -> \$t1
lb \$t2, 8(\$s0)	# c -> \$t2
lb \$t3, 9(\$s0)	# d -> \$t3
lb \$t4, 10(\$s0)	# e -> \$t4



Llamadas a funciones

Pasos necesarios:

1. Depositar los parámetros en los registros
2. Guardar la dirección de retorno
3. Transferir el control a la función
4. Reservar espacio para las variables de la función
5. Efectuar las tareas asignadas a la función
6. Depositar los resultados en los registros de retorno
7. Recuperar la dirección de retorno
8. Devolver el control al programa que efectuó la llamada.

Uso de los registros para las funciones

- \$a0 - \$a3: parámetros (registros 4 al 7)
- \$v0 , \$v1: valores de retorno (registros 2 y 3)
- \$t0 - \$t9: temporales (regs. 8 al 15, 24 y 25)
 - Pueden ser sobreescritos por la función llamada
- \$s0 - \$s7: salvados (regs. 16 al 23)
 - Deben ser salvados y restaurados por la función llamada si los emplea.
- \$gp: apuntador global para datos estáticos (reg. 28)
- \$sp: apuntador de pila (reg. 29)
- \$fp: apuntador de trama (reg. 30)
- \$ra: dirección de retorno (reg. 31)

Instrucciones de llamada a funciones

- Llamada a función: *jump and link*

`jal Etiqueta_función`

- Guarda la dirección de la siguiente instrucción en \$ra

- Transfiere el control a la dirección de la función

- Retorno de la función: *jump register*

`jr $ra`

- Copia \$ra en el contador de programa

- Esta instrucción puede usarse para saltos ordinarios

- Por ejemplo, case/switch, etc.

Ejemplo de función “hoja” (I)

- Función hoja o función terminal es la que no llama a otras funciones
- Código de alto nivel (C):

```
int ejemplo_hoja (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Parámetros **g, h, i** y **j** en \$a0, \$a1, \$a2 y \$a3
- **f** en \$s0 (por tanto, debe salvarse \$s0 en la pila)
- Resultado retorna en \$v0

Ejemplo de función “hoja” (II)

- Código ensamblador MIPS:

ejemplo_hoja:

```
addi $sp, $sp, -4
sw    $s0, 0($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero
lw    $s0, 0($sp)
addi $sp, $sp, 4
jr  $ra
```

Salvar \$s0 en la pila

Cuerpo de la función

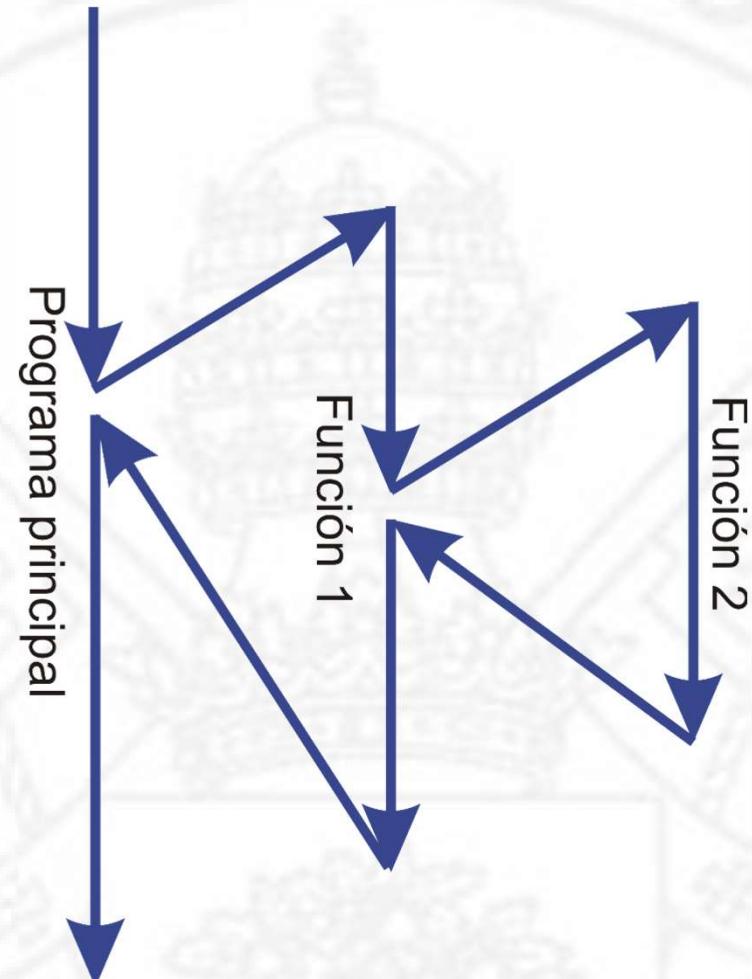
Depósito del resultado

Recuperación de \$s0

Retorno

Funciones anidadas (I)

- Función anidada: la que llama a su vez a otras funciones
- Ejemplo:
 - El programa principal llama a la función 1 que a su vez llama a la función 2



Funciones anidadas (II)

- Una función no debe modificar registros \$s ya que el programa principal los puede emplear para sus variables.
- Sin embargo, si llama a otra función (en el ejemplo, la función 2) esta puede modificar los registros \$t.
- Por tanto, si queremos que haya variables de la función 1 que conserven su valor después de la llamada a la función 2 deberán almacenarse en registros \$s.
- Como la función 1 no debe modificar los registros \$s, la solución es emplearlos, pero guardándolos en la pila al comenzar la función y restaurarlos al final para que conserven su valor después de la llamada.

Funciones recursivas

- Función recursiva: la que se llama a sí misma
- Para las llamadas anidadas o recursivas la función debe salvar en la pila:
 - La dirección de retorno
 - Todos los parámetros y variables que se necesiten después de la llamada
- Recuperar de la pila estas informaciones después de la llamada
- Las funciones recursivas deben tener una condición de salida para no entrar en un bucle infinito

Ejemplo de función recursiva (I)

- Código de alto nivel (C):

```
int fact (int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

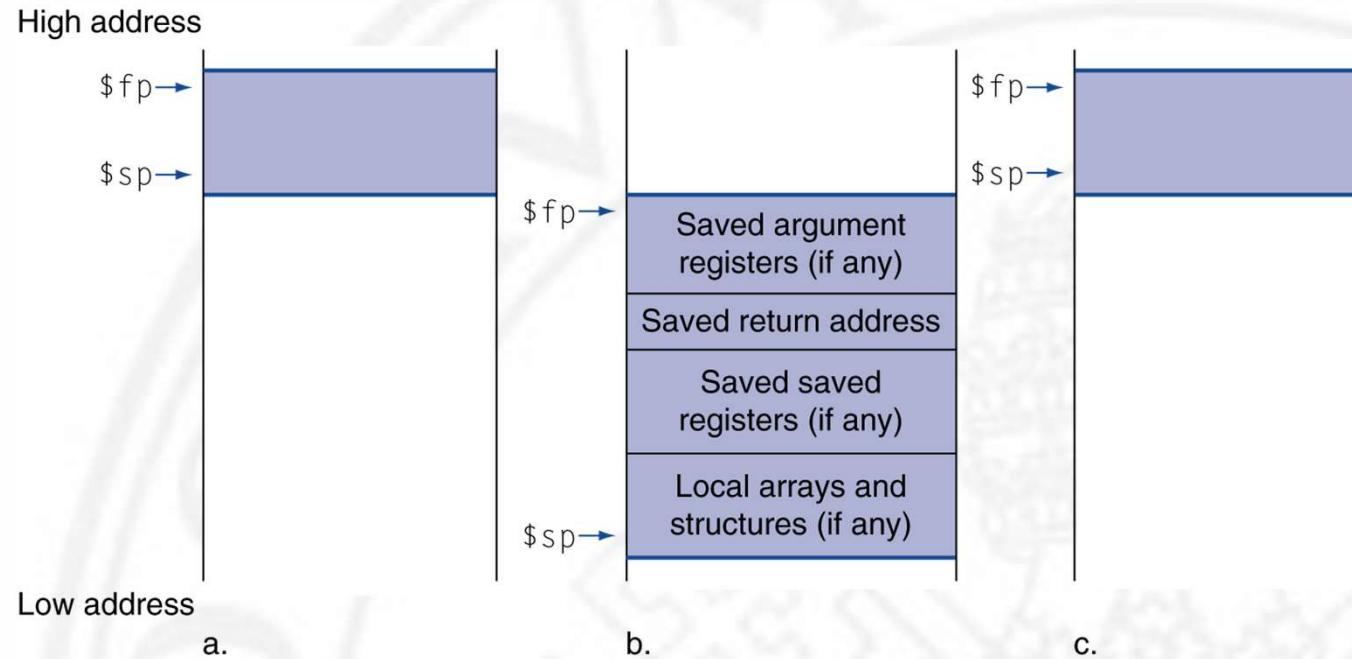
- Parámetro **n** en \$a0
- Resultado en \$v0
- Variable **n** dentro de la función en \$s0

Ejemplo de función recursiva (II)

Código ensamblador MIPS:

```
fact: addi $sp, $sp, -8      # reserva 2 lugares en la pila
      sw   $ra, 4($sp)       # guarda dirección de retorno
      sw   $s0, 0($sp)       # guarda registro $s utilizado
      slti $t0, $a0, 2        # n <= 1?
      beq  $t0, $zero, L1     # Si n > 1 va a L1
      addi $v0, $zero, 1       # si n <= 1 devuelve 1,
      addi $sp, $sp, 8         # libera 2 lugares de la pila
      jr   $ra                 # y retorna
L1:   add  $s0, $a0, $zero    # si no, guarda n en $s0 y
      addi $a0, $a0, -1        # decrementa n para hacer
      jal   fact               # llamada recursiva
      mul  $v0, $s0, $v0       # n!=n*(n-1)!, devuelve resultado
      lw    $s0, 0($sp)        # recupera valor original de $s0
      lw    $ra, 4($sp)        # y la dirección de retorno,
      addi $sp, $sp, 8         # libera 2 lugares de la pila
      jr   $ra                 # y retorna
```

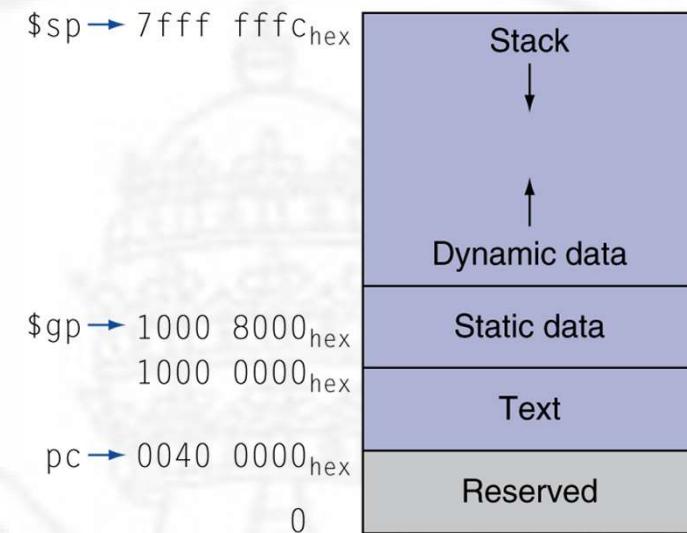
Datos locales en la pila



- La función llamada reserva el espacio para las variables locales
 - Por ejemplo, las variables automáticas de C
- Trama de pila de la función (registro de activación)
 - Los compiladores la usan para gestionar el almacenamiento en la pila

Esquema de la memoria

- ***Text:*** código del programa
- ***Static data:*** variables globales
 - Ej., variables estáticas, vectores y cadenas
 - \$gp se carga con la dirección base que permita direccionar las variables globales mediante desplazamientos
- ***Dynamic data: heap***
 - Ej.: malloc en C, new en Pascal
- ***Stack:*** variables automáticas



Ejemplo: copia de cadenas (`strcpy`) (I)

- Código de alto nivel (C):

- El terminador de cadena es el carácter nulo ('\0')

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i]) != '\0')
    i += 1;
}
```

- Direcciones de x e y en \$a0 y \$a1

- i en \$s0

Ejemplo: copia de cadenas (strcpy) (II)

- Código ensamblador MIPS:

strcpy:

```
addi $sp, $sp, -4      # reserva un lugar en la pila
sw   $s0, 0($sp)       # guarda $s0
add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1  # dirección de y[i] en $t1
    lw   $t2, 0($t1)     # $t2 = y[i]
    add  $t3, $s0, $a0  # dirección de x[i] en $t3
    sb   $t2, 0($t3)     # x[i] = y[i]
    beq $t2, $zero, L2   # Sale del bucle si y[i] == 0
    addi $s0, $s0, 1      # i = i + 1
    j    L1               # Siguiente iteración
L2: lw   $s0, 0($sp)     # recupera valor de $s0,
    addi $sp, $sp, 4      # libera el lugar de la pila,
    jr  $ra               # y retorna
```

Constantes de 32 bits

- La mayoría de las constantes son pequeñas
 - 16 bits para constante inmediata son suficientes
- Para casos ocasionales de constantes de 32 bits
 - lui *rt*, *constante*
 - Copia la constante de 16 bits en los 16 bits de más peso de *rt*
 - Pone a 0 los 16 bits de menos peso de *rt*

lui \$s0, 61

0000 0000 0011 1101	0000 0000 0000 0000
---------------------	---------------------

ori \$s0, \$s0, 2304

0000 0000 0011 1101	0000 1001 0000 0000
---------------------	---------------------

Direccionamiento en bifurcaciones

- Las instrucciones de bifurcación especifican:
 - Código de operación, 2 registros y la dirección de destino
- La mayoría de los destinos de bifurcación son cercanos. Se emplea el formato I
 - Las bifurcaciones pueden ser hacia adelante o hacia atrás

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>constante/desplazamiento</i>
6 bits	5 bits	5 bits	16 bits

- Se emplea direccionamiento relativo al PC
 - Dirección de destino = $PC + desplazamiento \times 4$
 - El PC ya está incrementado antes de la operación

Direccionamiento para saltos

- Los destinos de los saltos (`j` y `jal`) pueden estar en cualquier parte del segmento de código
 - En la instrucción se codifican los bits 27:2 de la dirección
 - Formato específico para estas instrucciones: formato J

<i>op</i>	<i>pseudoaddress</i>
6 bits	26 bits

- Direccionamiento (pseudo)directo en saltos:
Dirección de destino = $\text{PC}_{31 \dots 28} : (\text{pseudoaddress} \times 4)$
 $= \text{PC}_{31 \dots 28} : \text{pseudoaddress} : 00$

Ejemplo de dirección de destino

Código del bucle (ejemplo anterior, diapositiva 2.48)

O Supongamos que Loop corresponde a la dirección 80000

Loop:	sll	\$t1, \$s3, 2	80000	0	0	19	9	4	0
	add	\$t1, \$t1, \$s6	80004	0	9	22	9	0	32
	lw	\$t0, 0(\$t1)	80008	35	9	8		0	
	bne	\$t0, \$s5, Exit	80012	5	8	21		2	
	addi	\$s3, \$s3, 1	80016	8	19	19		1	
	j	Loop	80020	2			20000		
Exit:	...		80024						

Bifurcaciones lejanas

- Cuando el destino de una bifurcación sea demasiado lejano como para codificarlo con un desplazamiento de 16 bits, hay que reescribir el código
- Ejemplo

```
beq $s0,$s1, L1
```

↓

```
bne $s0,$s1, L2  
j L1
```

L2:

...

Resumen de los modos de direccionamiento

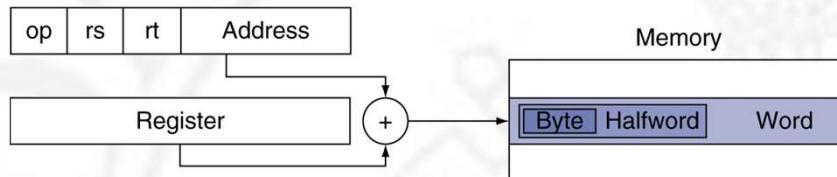
1. Immediate addressing



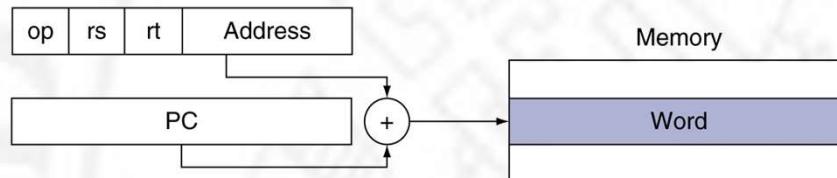
2. Register addressing



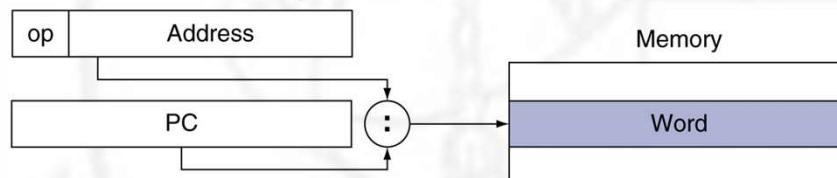
3. Base addressing



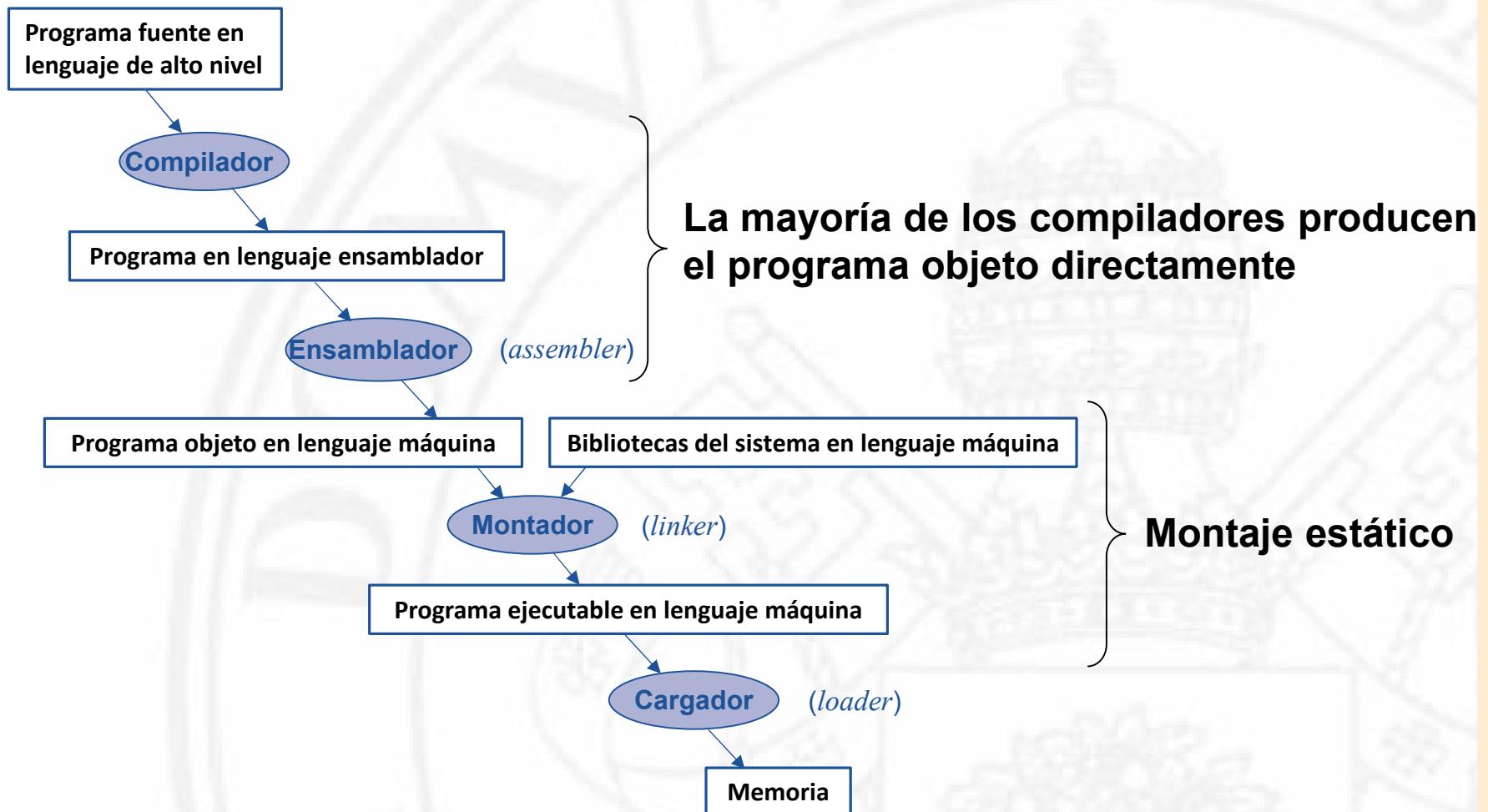
4. PC-relative addressing



5. Pseudodirect addressing



Traducción y arranque de programas



Instrucciones sintéticas (Pseudoinstrucciones)

- La mayor parte de las instrucciones del ensamblador representan sólo una instrucción en lenguaje máquina
- Instrucciones sintéticas: nuevas instrucciones inventadas por el ensamblador

move \$t0, \$t1 → add \$t0, \$zero, \$t1

blt \$t0, \$t1, L → slt \$at, \$t0, \$t1
bne \$at, \$zero, L

○ **\$at** (registro 1): almacenamiento temporal para el ensamblador

Producción del programa objeto

- El ensamblador (o compilador) traduce el programa a código máquina
- Suministra toda la información para construir el programa ejecutable completo a partir de sus partes:
 - Cabecera: describe el contenido del programa objeto
 - Segmento de código (*text*): instrucciones traducidas
 - Segmento de datos estáticos: espacio de datos reservado durante todo el tiempo de ejecución del programa
 - Información para relocalización: para contenidos que dependen de la localización final del programa cargado
 - Tabla de símbolos: contiene las definiciones de las referencias externas
 - Información para depuración (si fuera necesaria)

Montaje de los módulos objeto

- Produce una imagen ejecutable
 1. Combina todos los segmentos
 2. Determina las direcciones representadas por las etiquetas
 3. Fija las referencias relocalizables y externas

Carga del programa

Carga del fichero ejecutable en memoria

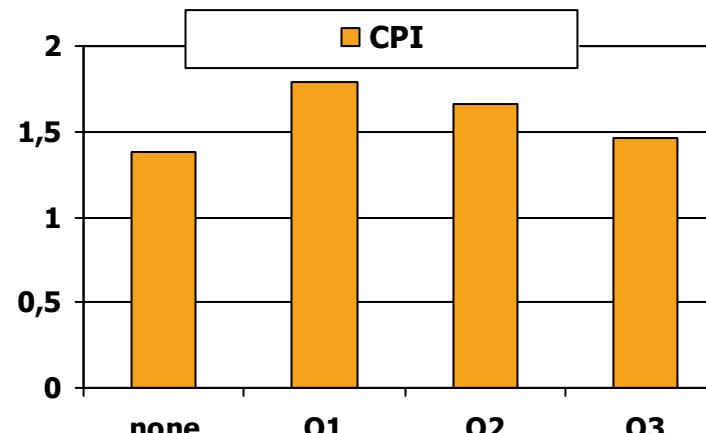
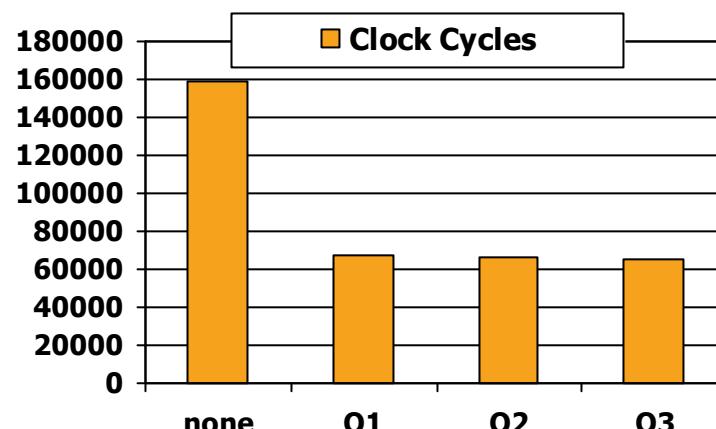
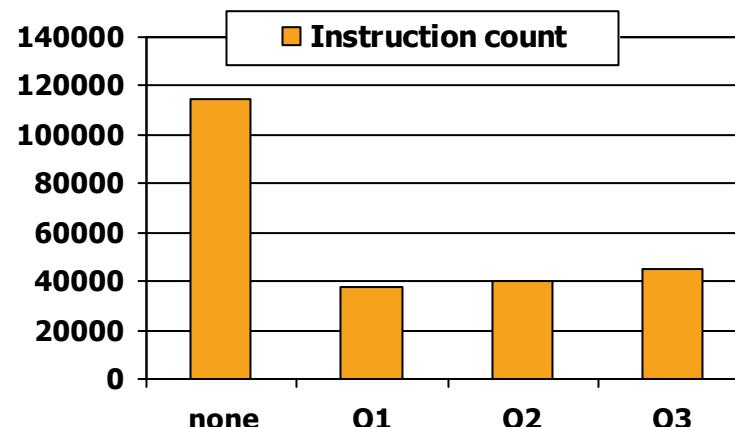
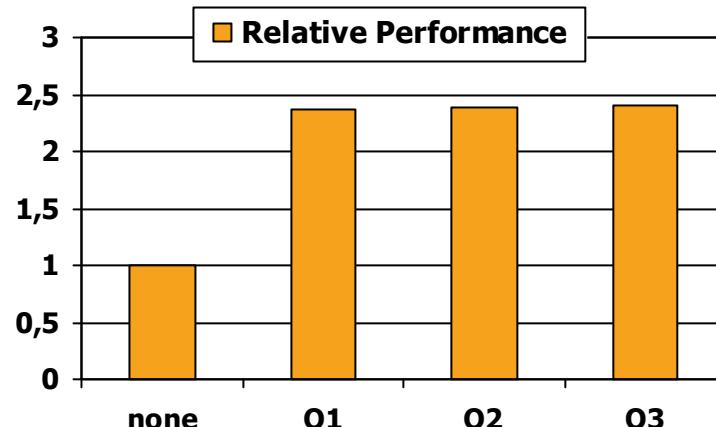
1. Lee la cabecera para determinar el tamaño de cada segmento
2. Crea un espacio de direcciones suficiente para el código y los datos del programa
3. Copia el código y los datos inicializados en la memoria
4. Copia los parámetros del programa principal (si los hay) en la pila
5. Inicializa registros (incluyendo \$sp, \$fp, \$gp)
6. Salta a la rutina de inicio que:
 - Copia los parámetros en \$a0, ... y llama al programa principal
 - Cuando el programa principal retorna, llama a la función del sistema `exit` que devuelve el control al Sistema Operativo

Montaje dinámico

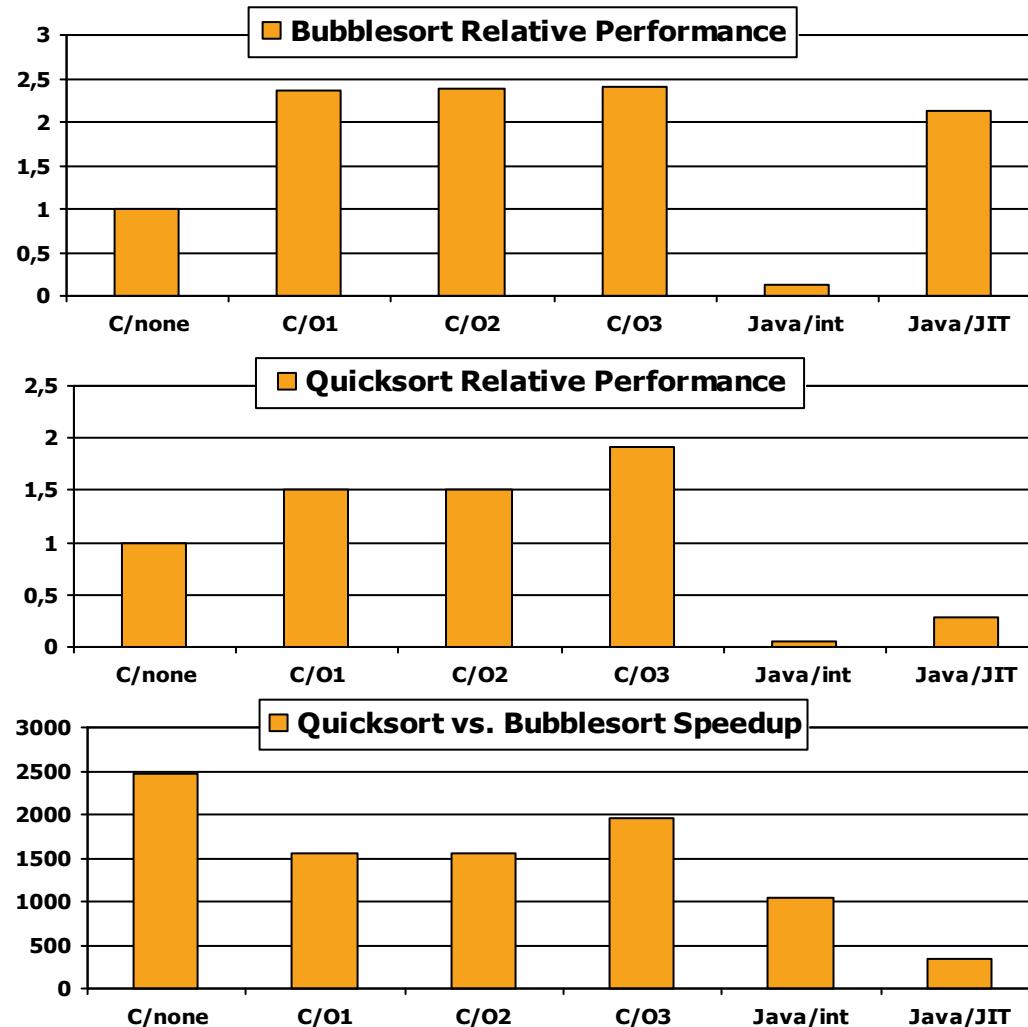
- Sólo se cargan y se montan las bibliotecas de funciones si son invocadas
 - Es necesario que el código de la función sea relocalizable
 - Evita el aumento de tamaño del programa ejecutable producido por el montaje estático, que carga todas las bibliotecas independientemente de que se usen o no
 - Automáticamente recoge las nuevas versiones de las bibliotecas

Efecto de la compilación optimizada

Compilado con gcc para Pentium 4 bajo Linux



Efecto del lenguaje y del algoritmo



Enseñanzas de estas estadísticas

- El número de instrucciones y el CPI, de forma aislada, no son buenos indicadores del rendimiento
- Las optimizaciones del compilador dependen del algoritmo utilizado
- Los lenguajes compilados son mucho más rápidos que los interpretados
- No hay compilador que arregle un mal algoritmo

Vectores y apuntadores

- La indexación de vectores en código máquina implica:
 - Multiplicar el índice por el tamaño del elemento
 - Sumarlo a la dirección base del vector
- Los apuntadores corresponden directamente a las direcciones de memoria
 - Pueden evitar la complejidad de la indexación

Ejemplo: Puesta a 0 de un vector

```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        array[i] = 0;
}
```

```
clear1:
    move $t0,$zero    # i = 0
L1:   sll $t1,$t0,2    # $t1 = i * 4
      add $t2,$a0,$t1  # $t2 = &array[i]
      sw $zero, 0($t2) # array[i] = 0
      addi $t0,$t0,1    # i++
     slt $t3,$t0,$a1    # $t3 = (i<size)
      bne $t3,$zero,L1  # if (i<size)
                      #     goto L1
```

```
clear2(int *array, int size)
{
    int *p;
    for (p = array; p < &array[size]; p++)
        *p = 0;
}
```

```
clear2:
    move $t0,$a0        # p = & array[0]
    sll $t1,$a1,2        # $t1 = size * 4
    add $t2,$a0,$t1        # $t2 = &array[size]
L2:   sw $zero,0($t0)    # Memory[p] = 0
      addi $t0,$t0,4        # p = p + 4 (p++)
      slt $t3,$t0,$t2        # $t3 = (p<&array[size])
      bne $t3,$zero,L2      # if (p<&array[size])
                      #     goto L2
```

Comparación entre vectores y apuntadores

- En ambos casos la multiplicación se reduce a un desplazamiento
- La versión con vector tiene el desplazamiento dentro del bucle
 - Para el cálculo del desplazamiento correspondiente al índice *i*
 - Por contra es más simple incrementar un apuntador
- Un buen compilador puede conseguir, a partir de vectores, un código de eficiencia similar al obtenido mediante el uso de apuntadores
 - El compilador puede eliminar el índice
 - El código basado en vectores es más claro y seguro

Conclusiones (I)

- Principios de diseño:
 1. La simplicidad favorece la regularidad
 2. Cuanto más pequeño, más rápido
 3. Mejorar en lo posible los casos más frecuentes
 4. Un buen diseño requiere buenas soluciones de compromiso
- Capas de software y hardware
 - Compilador, ensamblador, hardware
- MIPS: caso típico de ISA RISC
 - En contraposición a la arquitectura x86

Conclusiones(II)

- Medida de las frecuencias de las instrucciones de MIPS en programas de prueba (*benchmarks*)
 - Debe procurarse mejorar los casos más frecuentes
 - Hay que adoptar soluciones de compromiso

Clase de instrucciones	Ejemplos en MIPS	SPEC2006 Int	SPEC2006 FP
Aritméticas	add, sub, addi	16%	48%
Transferencia de datos	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Lógicas	and, or, nor, andi, ori, sll, sr1	12%	4%
Bifurcación condicional	beq, bne,slt, slti, sltiu	34%	8%
Saltos	j, jr, jal	2%	0%