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**
  - Simplicación 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 *Imagination Technologies*:** https://imgtec.com/mips/
- **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)

```
add a, b, c   # a ← b + c
```

- **Todas las operaciones aritméticas tienen esta misma forma**
- ***1er. principio de diseño:***
  La simplicidad favorece la regularidad
  - La regularidad simplifica la implementacion
  - 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
- ***2º Principio de diseño:***

  Cuanto más pequeño, más rápido

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, respectivamente, en `$s0`, `$s1`, `$s2`, `$s3` y `$s4`

- **Código real compilado para MIPS:**

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

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
- **MIPS es *Big Endian*:**
  - El byte más significativo se almacena en la dirección más baja
  - *Little Endian*: El byte menos significativo se almacena en la dirección más baja

Ejemplo 1 de operando en memoria

- **Código de alto nivel (C):**

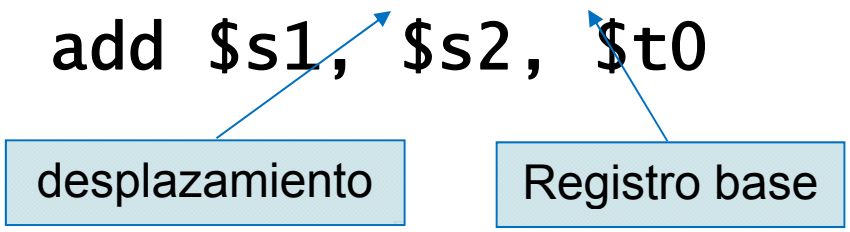
```
g = h + A[8];
```

  - 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
```



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 se 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 instrucción**

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

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

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

- ***3er 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

La constante 0 (Zero)

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

```
add $t2, $s1, $zero
```

Enteros binarios sin signo
(binario natural)

Un número x se representa mediante la secuencia de bits $x_{n-1}x_{n-2}\ldots x_1x_0$ para la que

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0 = \sum_{i=0}^{n-1} x_i\, 2^i$$

- Ejemplo

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2 =$
  $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 =$
  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- Rango representable: de 0 a $+2^n - 1$
- Con 32 bits

  Rango: de 0 a +4.294.967.295

Enteros binarios con signo en binario signado o signo-magnitud

El signo se representa mediante un bit separado. El resto representa el módulo. Un número x se representa mediante la secuencia de bits $x_{n-1}x_{n-2}...x_1x_0$ para la que:

$$x = (-1)^{x_{n-1}} \times (x_{n-2}2^{n-2} + ... + x_1 2^1 + x_0 2^0)$$

- Ejemplo: $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_{(2} =$
  $= (-1)^1 x(0+ \ldots + 1\times2^3 + 0\times2^2 +1\times2^1 +1\times2^0) =$
  $= -(0 + \ldots + 8 + 0 + 2 + 1) = -11_{(10}$
- Rango representable: desde $-(2^{n-1} - 1)$ a $+(2^{n-1} - 1)$
- Inconvenientes:
  - Hay dos representaciones para el 0 (+0 y -0)
  - Hay que procesar el signo antes y después de cada operación.
- No se emplea para representar números enteros en computadores

Enteros con signo en complemento a 2 (I)

Un número x se representa mediante la secuencia de bits $x_{n-1}x_{n-2}...x_1x_0$ para la que

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + ... + x_1 2^1 + x_0 2^0$$

- Ejemplo

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
  $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
  $= -2.147.483.648 + 2.147.483.644 = -4_{10}$

- Rango representable: de -2^{n-1} a $+2^{n-1} - 1$
- Con 32 bits

  Rango: de –2.147.483.648 a +2.147.483.647

Enteros con signo en complemento a 2 (II)

- **El bit de más peso es el bit de signo**
  - 1 para números negativos
  - 0 para números no negativos
- **$-(-2^{n-1})$ no se puede representar**
- **Los números no negativos se representan de la misma forma en binario natural que en complemento a 2**
- **Algunos números concretos:**
  - 0: 0000 0000. . .0000
  - –1: 1111 1111. . .1111
  - Número más negativo: 1000 0000. . .0000
  - Número más positivo: 0111 1111. . .1111

Cálculo del opuesto en complemento a 2

- **Complementar y sumar 1**
  - Complementar significa $1 \rightarrow 0,\ 0 \rightarrow 1$
- **Demostración:**

$$x + \overline{x} = 1111...111_2 = -1 \Rightarrow \overline{x} + 1 = -x$$

- Ejemplo: calcular el opuesto de +2
  - +2 = $0000\ \ 0000\ \ .\ .\ .\ \ 0010_2$
  - –2 = $1111\ \ 1111\ \ .\ .\ .\ \ 1101_2$ + 1
    = $1111\ \ 1111\ \ .\ .\ .\ \ 1110_2$

Extensión de signo

- **Representación de un número con más bits**
  - Debe preservar el valor numérico con su signo
- **En el conjunto de instrucciones de MIPS:**
  - `addi`: extiende el signo a la constante
  - `lb, lh`: extienden el signo al valor cargado: byte o semipalabra (*halfword*)
  - `beq, bne`: extienden el signo del desplazamiento
- **Se rellena por la izquierda con el bit de signo**
  - De la misma forma que en binario natural se rellena con 0's por la izquierda
- **Ejemplos: extensión de 8 a 16 bits:**
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Hexadecimal

- **Base 16**
  - Es una representación compacta de cadenas de bits
  - 4 bits por cada dígito hexadecimal:

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

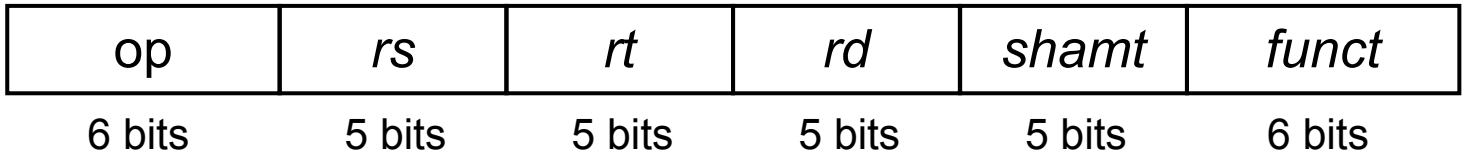
- Ejemplo:

```
eca8  6420
  e     c     a     8     6     4     2     0
1110  1100  1010  1000  0110  0100  0010  0000
```

Codificación de las intrucciones

- **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**
  - `$t0` – `$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



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* (*rd* ← *rs* + *rt*)

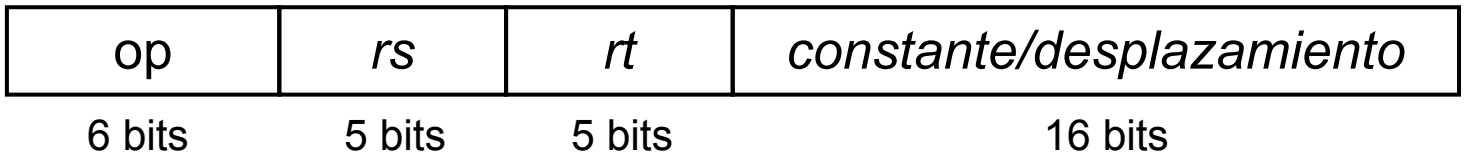
op	*rs*	*rt*	*rd*	*shamt*	*funct*
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

`add $t0, $s1, $s2` (**`$t0 ← $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

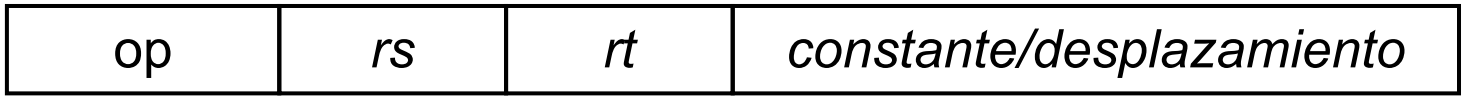


- **Instrucciones aritméticas inmediatas e instrucciones de carga y almacenamiento (`load/store`)**
  - *rt*: número de registro fuente o destino
  - *constante*: desde -2^{15} a $+2^{15} - 1$
  - *desplazamiento*: cantidad que se suma a la dirección base situada en *rs*
- ***4º principio de diseño:***

  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 los formatos lo más similares posible

Ejemplo de formato I (I)

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



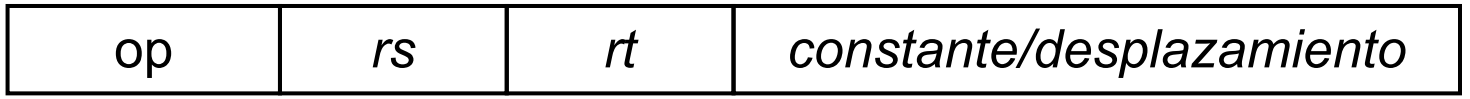
addi $t0, $s1, 12 ($t0 ← $s1 + 12)

addi	$s1	$t0	*constante*
8	17	8	12
001000	10001	01000	0000000000001100

$0010\ 0010\ 0010\ 1000\ 0000\ 0000\ 0000\ 1100_{(2} = 0x2228000c$

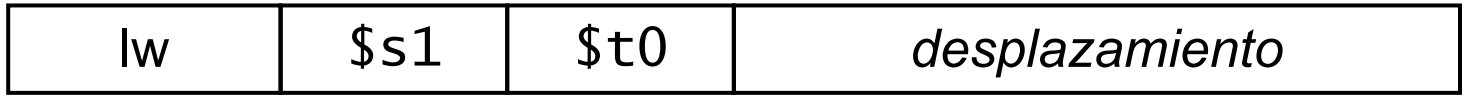
Ejemplo de formato I (II)

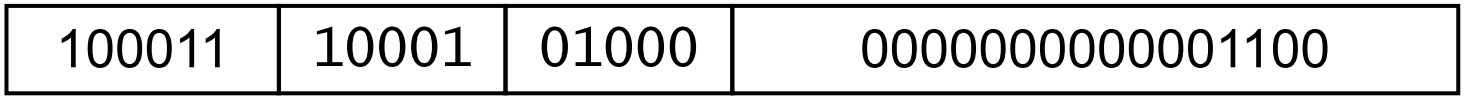
`lw` *rt, despl.*(*rs*) (*rt* ← Mem(*rs* + *despl.*))



6 bits 5 bits 5 bits 16 bits

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

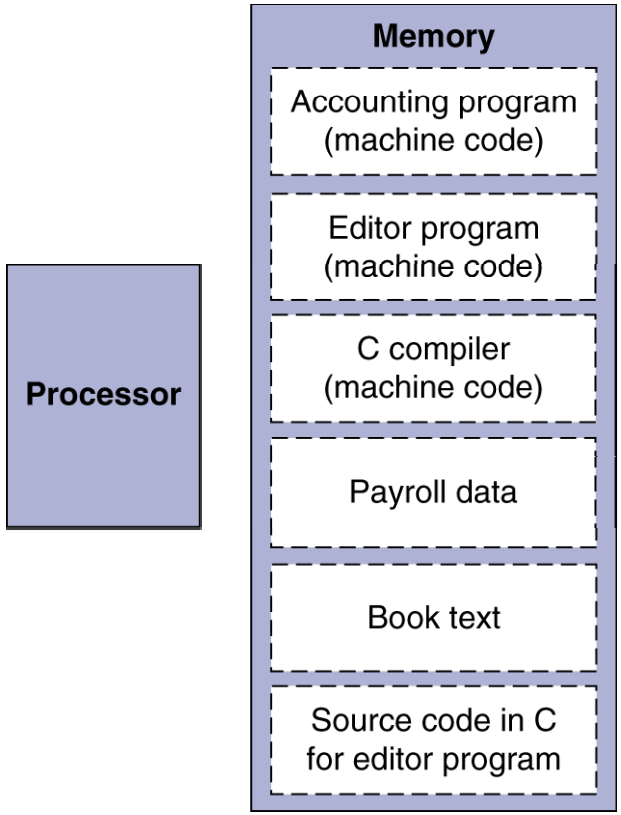




$1000\ 1110\ 0010\ 1000\ 0000\ 0000\ 0000\ 1100_{(2} = 0x8e28000c$

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**
  - *ISA*s estandarizadas

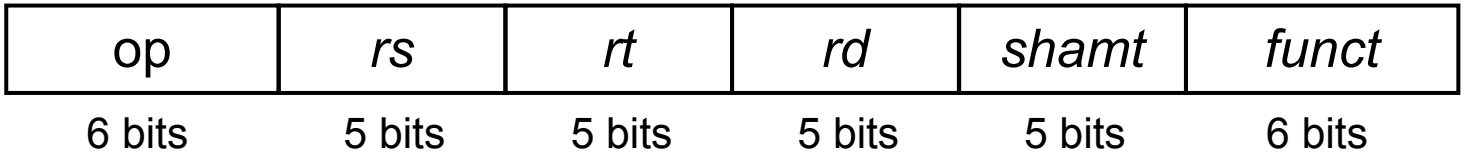
Operaciones lógicas

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



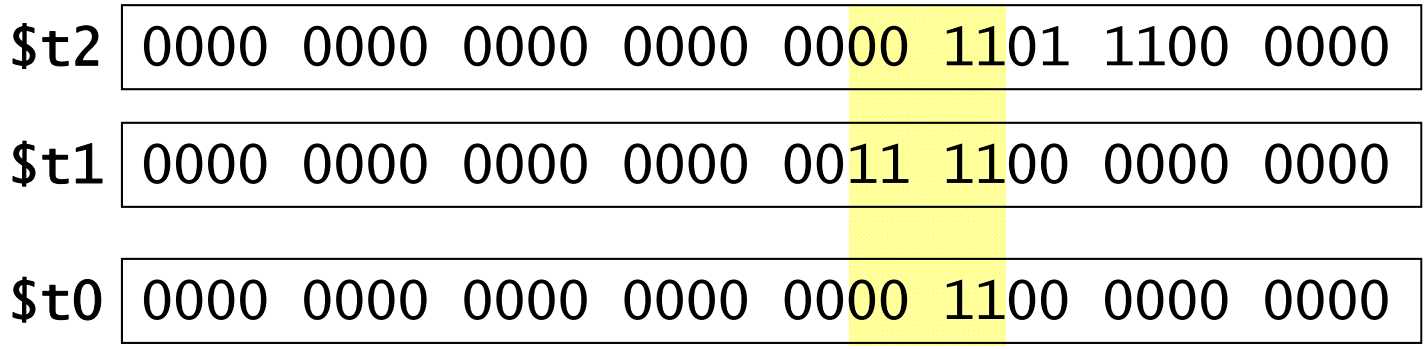
- *shamt*: **número de lugares a desplazar**
- **Desplazamiento lógico a la izquierda (`sll`)**
  - Desplaza a la izquierda y rellena con 0's
  - `sll` desplazando *i* lugares multiplica por 2^i
- **Desplazamiento lógico a la derecha (`srl`)**
  - Desplaza a la derecha y rellena con 0's
  - `srl` desplazando *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
```



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              $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 0011 1101 1100 0000

Operación NOT

Sirve para invertir bits en una palabra

- Cambia 0 por 1 y 1 por 0

- **MIPS posee la instrucción NOR de 3 operandos**
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

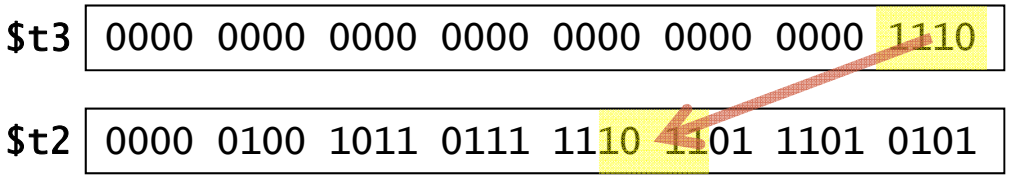
Registro 0: siempre contiene cero

$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`



Utilidades de las operaciones lógicas (I)

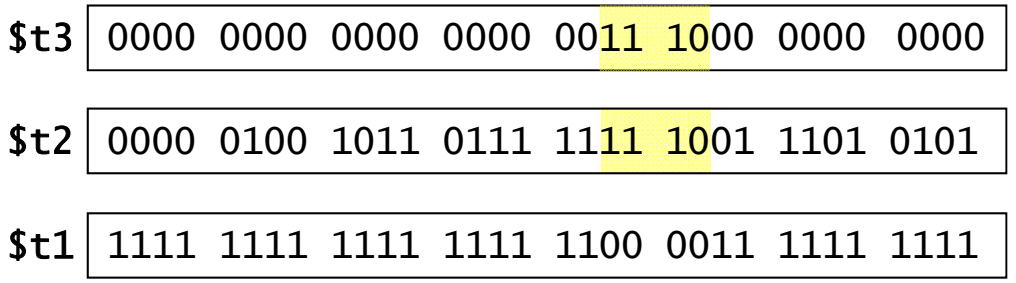
Incluir bits de una palabra

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

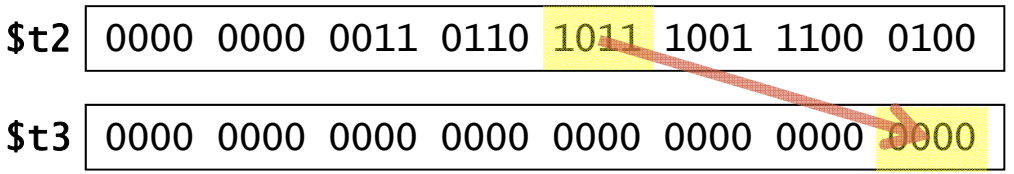
```
and $t2, $t1, $t2
sll $t3, $t3, 10
or  $t2, $t3, $t2
```



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)

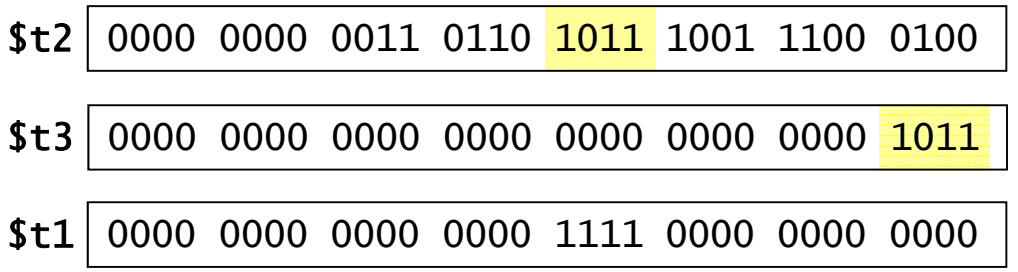
Extraer bits de una palabra

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
```



Bifurcaciones condicionales

Bifurcan si la condición se cumple

- 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*)

- `j` *L1*

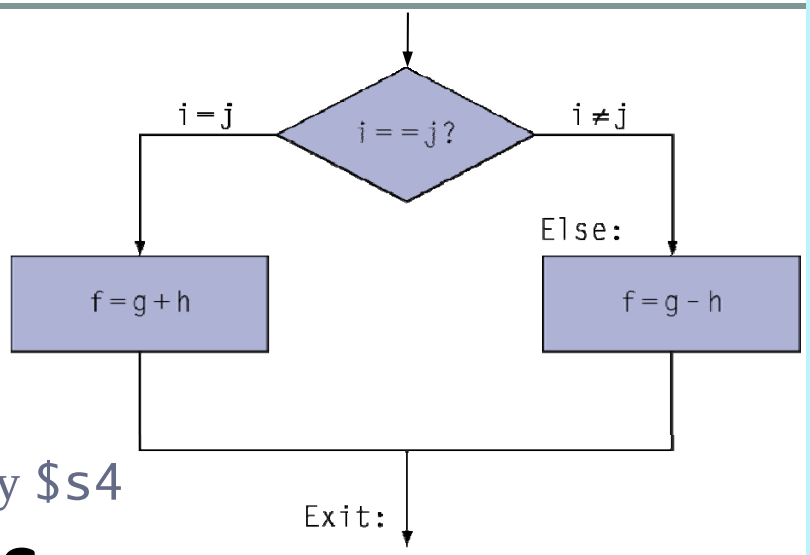
  Salta incondicionalmente a la instrucción con la etiqueta *L1*

Compilación de la instrucción If

- **Código de alto nivel (C):**

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

  - 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;
```

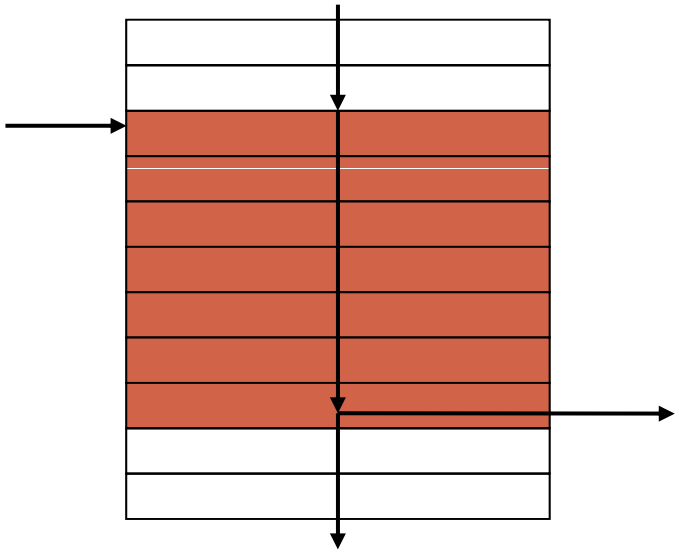
  - i en $s3, k en $s5, dirección inicial de save en $s6

- **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 instruciones 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

- si no, lo ponen a 0

- `slt` *`rd, rs, rt`*
  - if (*rs* < *rt*) *rd* = 1; else *rd* = 0;
- `slti` *`rt, rs, constante`*
  - if (*rs* < constante) *rt* = 1; else *rt* = 0;
- **Se usan en combinación con beq y bne**

```
slt $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 <, ≥, … es más lento que para detectar = o ≠**
  - 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**

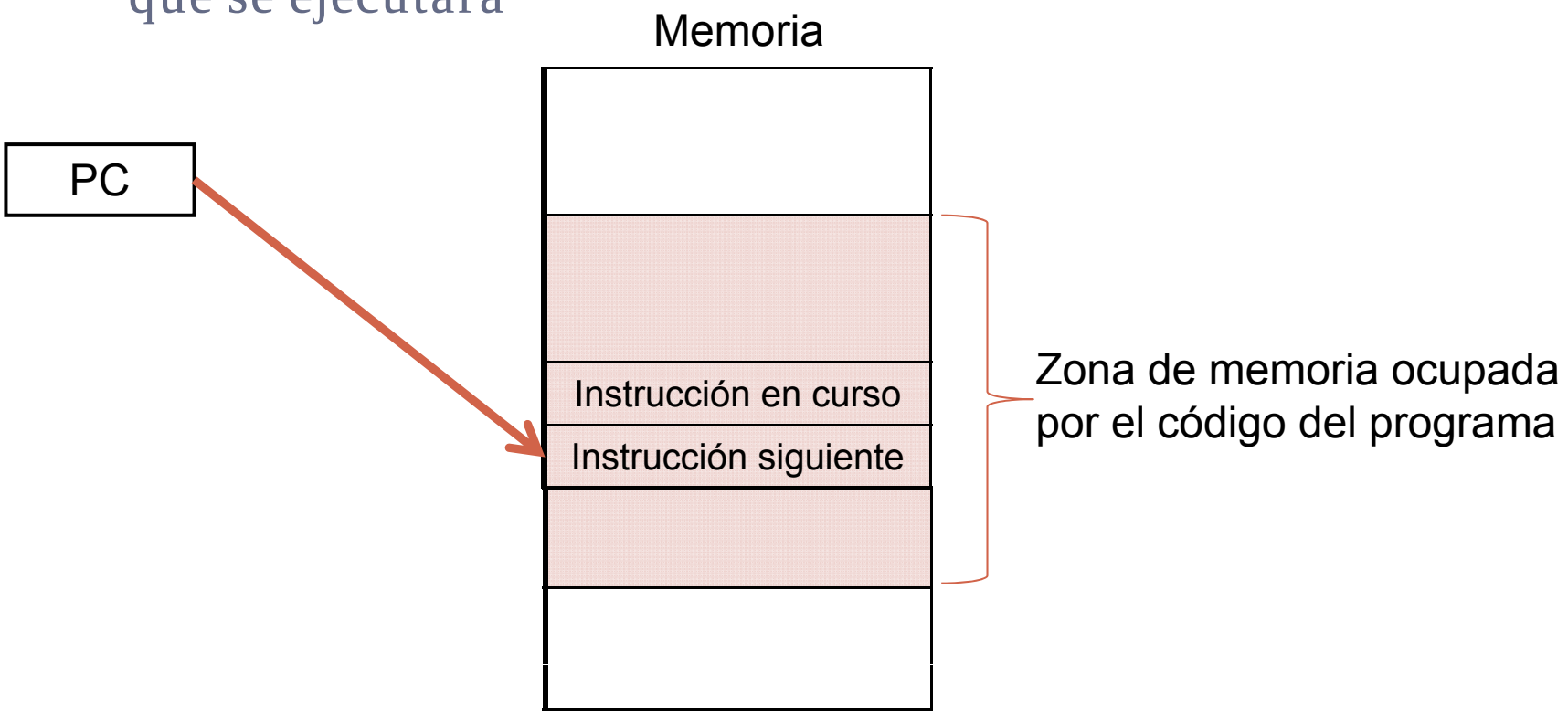
```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

  - `slt  $t0, $s0, $s1  # con signo`
    - $-1 < +1 \Rightarrow$ `$t0` $= 1$
  - `sltu $t0, $s0, $s1  # sin signo`
    - $+4.294.967.295 > +1 \Rightarrow$ `$t0` $= 0$

El contador de programa (PC)

Es un registro de control del computador

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



Llamadas a procedimientos

Pasos necesarios:

1. Depositar los parámetros en los registros
2. Guardar la dirección de retorno
3. Transferir el control al procedimiento
4. Reservar espacio para las variables del procedimiento
5. Efectuar las tareas asignadas al procedimiento
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 los procedimientos

- **`$a0` – `$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 el procedimiento llamado
- **`$s0` – `$s7`: salvados (regs. 16 al 23)**
  - Deben ser salvados y restaurados por el procedimiento llamado
- **`$gp`: apuntador global para datos estáticos (reg. 28)**
- **`$sp`: apuntador de pila (reg. 29)**
- **`$fp`: apuntador de trama (reg. 30)**
- **`$ra`: dirección de retorno (reg. 31)**

Instrucciones de llamada a procedimientos

- **Llamada a procedimiento: *jump and link***

  `jal` *`Etiqueta_procedimiento`*

  - Guarda la dirección de la siguiente instrucción en `$ra`
  - Transfiere el control a la dirección del procedimiento

- **Retorno de procedimiento: *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 procedimiento “hoja” (I)

- **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 procedimiento “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 del procedimiento

Depósito del resultado

Recuperación de `$s0`

Retorno

Procedimientos anidados y recursivos

- **Procedimiento anidado: el que llama a otros procedimientos**
- **Procedimientos recursivo: el que se llama a sí mismo**
- **Para las llamadas anidadas o recursivas el procedimiento debe salvar en la pila:**
  - La dirección de retorno
  - Todos los parámetros y variables temporales que se necesiten después de la llamada
- **Recuperar de la pila estas informaciones después de la llamada**
- **Los procedimientos recursivos deben tener una condición de salida para no entrar en un bucle infinito**

Ejemplo de procedimiento recursivo (I)

- **Código de alto nivel (C):**

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

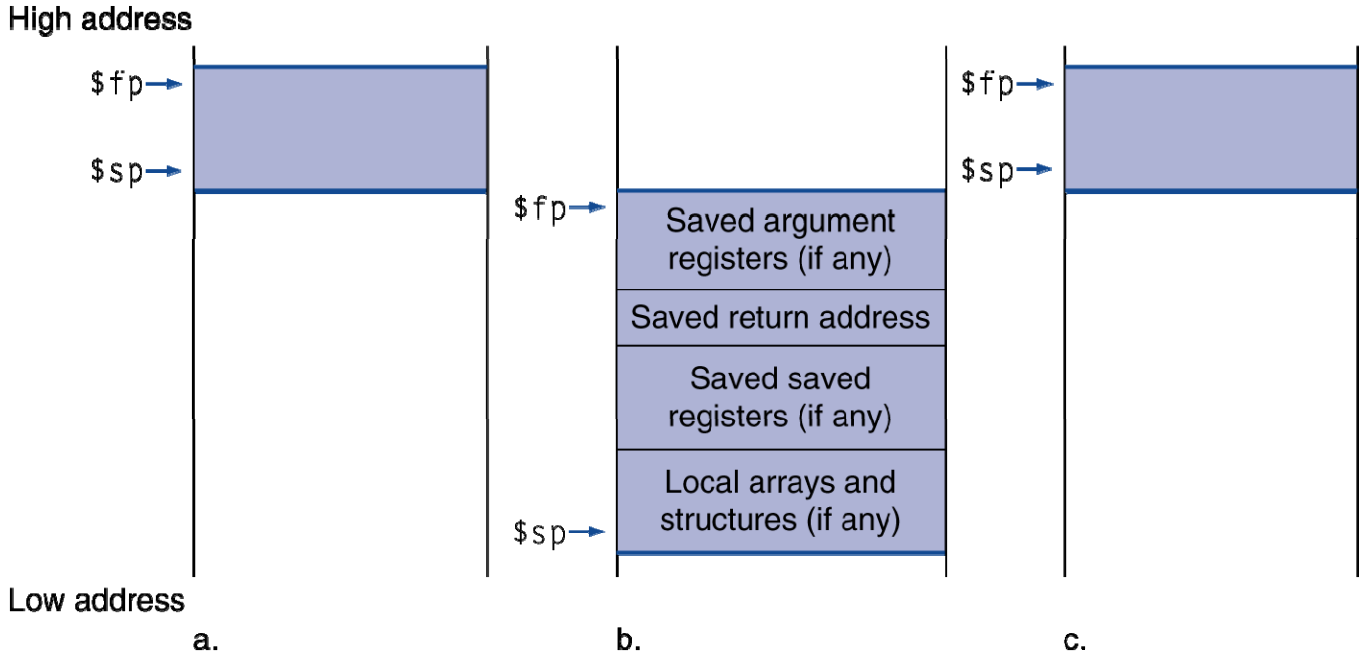
  - Parámetro `n` en `$a0`
  - Resultado en `$v0`

Ejemplo de procedimiento recursivo (II)

- **Código ensamblador MIPS:**

```
fact:
     addi $sp, $sp, -8    # reserva 2 lugares en la pila
     sw   $ra, 4($sp)     # guarda dirección de retorno
     sw   $a0, 0($sp)     # guarda parámetro
     slti $t0, $a0, 1     # 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 los 2 lugares de la pila
     jr   $ra             # y retorna
L1:  addi $a0, $a0, -1    # si no, decrementa n
     jal  fact            # llamada recursiva
     lw   $a0, 0($sp)     # recupera valor original de n
     lw   $ra, 4($sp)     # y dirección de retorno,
     addi $sp, $sp, 8     # libera los 2 lugares de la pila,
     mul  $v0, $a0, $v0   # multiplica, devuelve resultado
     jr   $ra             # y retorna
```

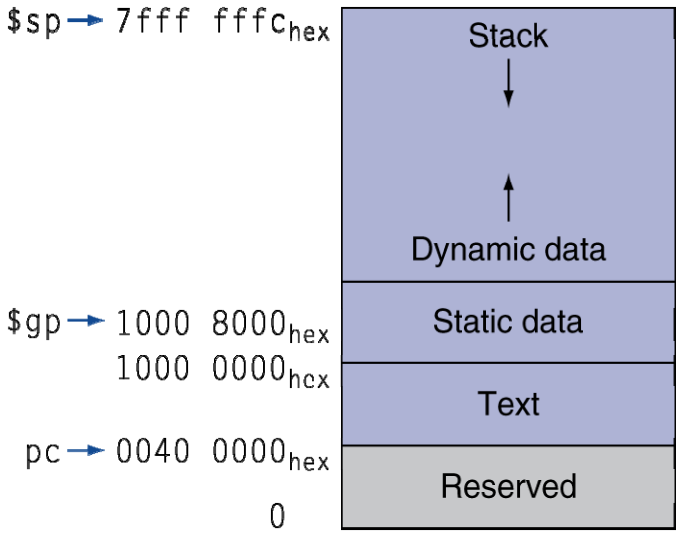
Datos locales en la pila



- **El procedimiento llamado reserva el espacio para las variables locales**
  - Por ejemplo, las variables automáticas de C
- **Trama de pila del procedimiento (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**



Datos de tipo carácter

- **Juegos de caracteres codificados en bytes**
  - ASCII: 128 caracteres
    - 95 visibles, 33 de control
  - Latin-1: 256 caracteres
    - ASCII y 96 caracteres visibles más
- **Unicode: Juego de caracteres de 32 bits**
  - Se usa en Java, C++, etc.
  - Incluye la mayoría de los alfabetos del mundo y símbolos
  - UTF-8 y UTF-16: son implementaciones del Unicode con codificaciones de longitud variable

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:

  `lb` *rt, despl*(*rs*) `lh` *rt, despl*(*rs*)

  Extiende el signo a 32 en *rt*

  `lbu` *rt, despl*(*rs*) `lhu` *rt, despl*(*rs*)

  Extiende 0's a 32 bits en *rt*

  `sb` *rt, despl*(*rs*) `sh` *rt, despl*(*rs*)

  Almacena sólo el byte/semipalabra de menos peso de *rt*

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)

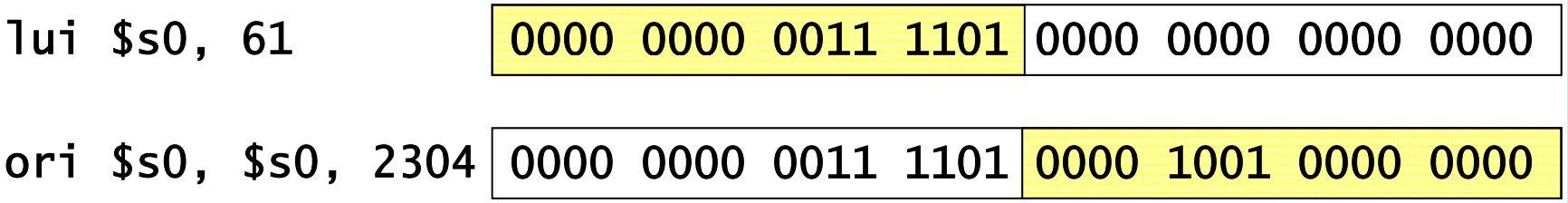
- **Código ensamblador MIPS:**

```
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
    lbu  $t2, 0($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

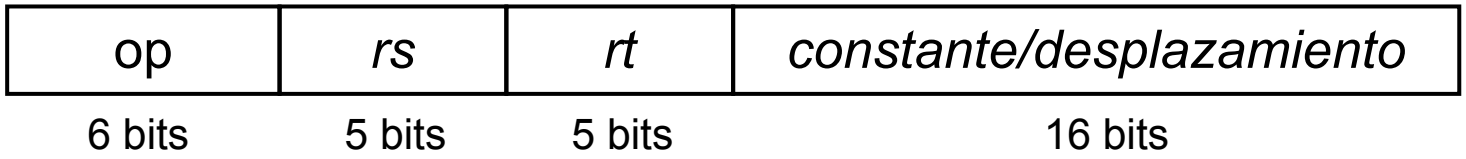
- **La mayoría de las constantes son pequeñas**
  - 16 bits para constante inmediata son suficientes
- **Para casos ocasionales de constantes de 32 bits**

  `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`*



Direccionamiento en bifurcaciones

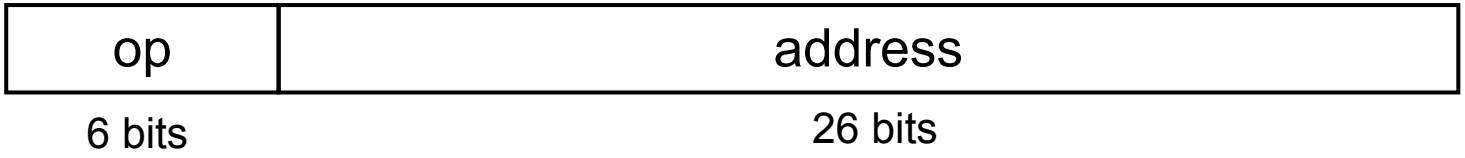
- **Las instrucciones de bifurcación especifican:**
  - Código de operación, 2 registros y la dirección de destino
- **La mayoría de los destinos de bifurcación son cercanos. Se emplea el formato I**
  - Las bifurcaciones pueden ser hacia adelante o hacia atrás



- Se emplea direccionamiento relativo al PC
  - Dirección de destino = PC + *desplazamiento* × 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 (*text*)**
  - En la instrucción se codifican los bits 27:2 de la dirección
  - Formato específico para estas instrucciones: **formato J**



- Direccionamiento (pseudo)directo en saltos:

  Dirección de destino = $PC_{31...28}$: (address × 4)

Ejemplo de dirección de destino

- **Código del bucle (ejemplo anterior, transp. 2.39)**
  - Supongamos que Loop corresponde a la dirección 80000

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

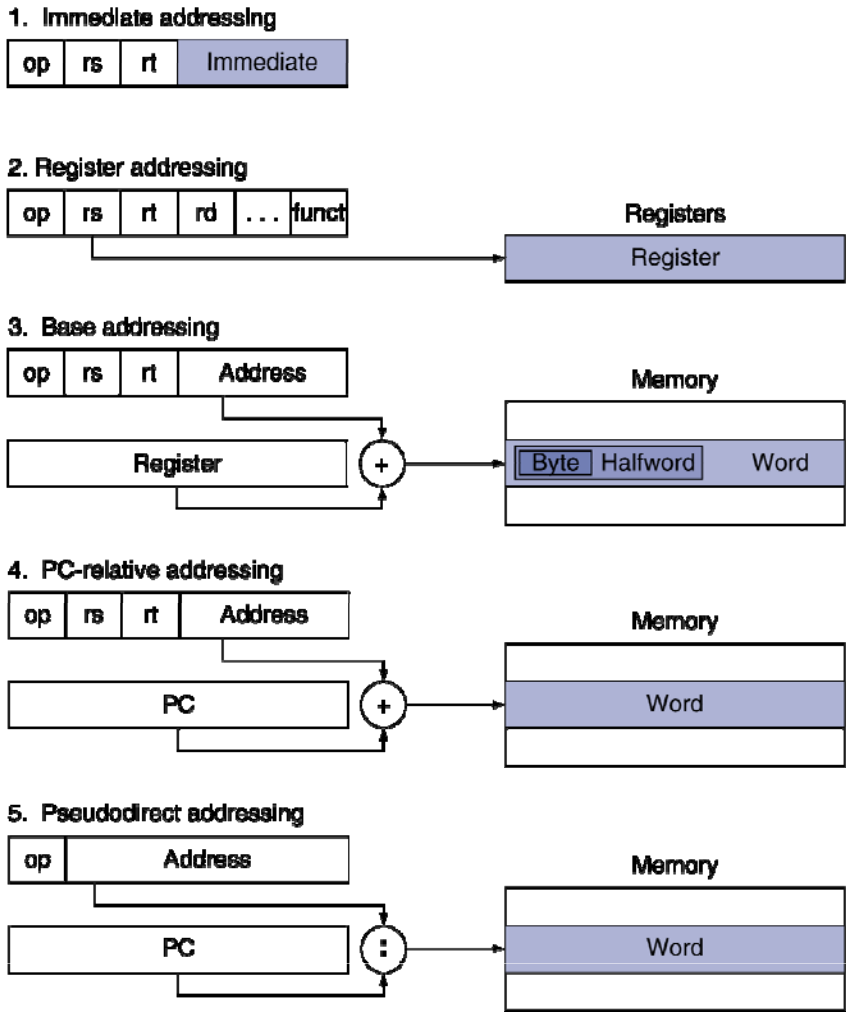
0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

Bifurcaciones lejanas

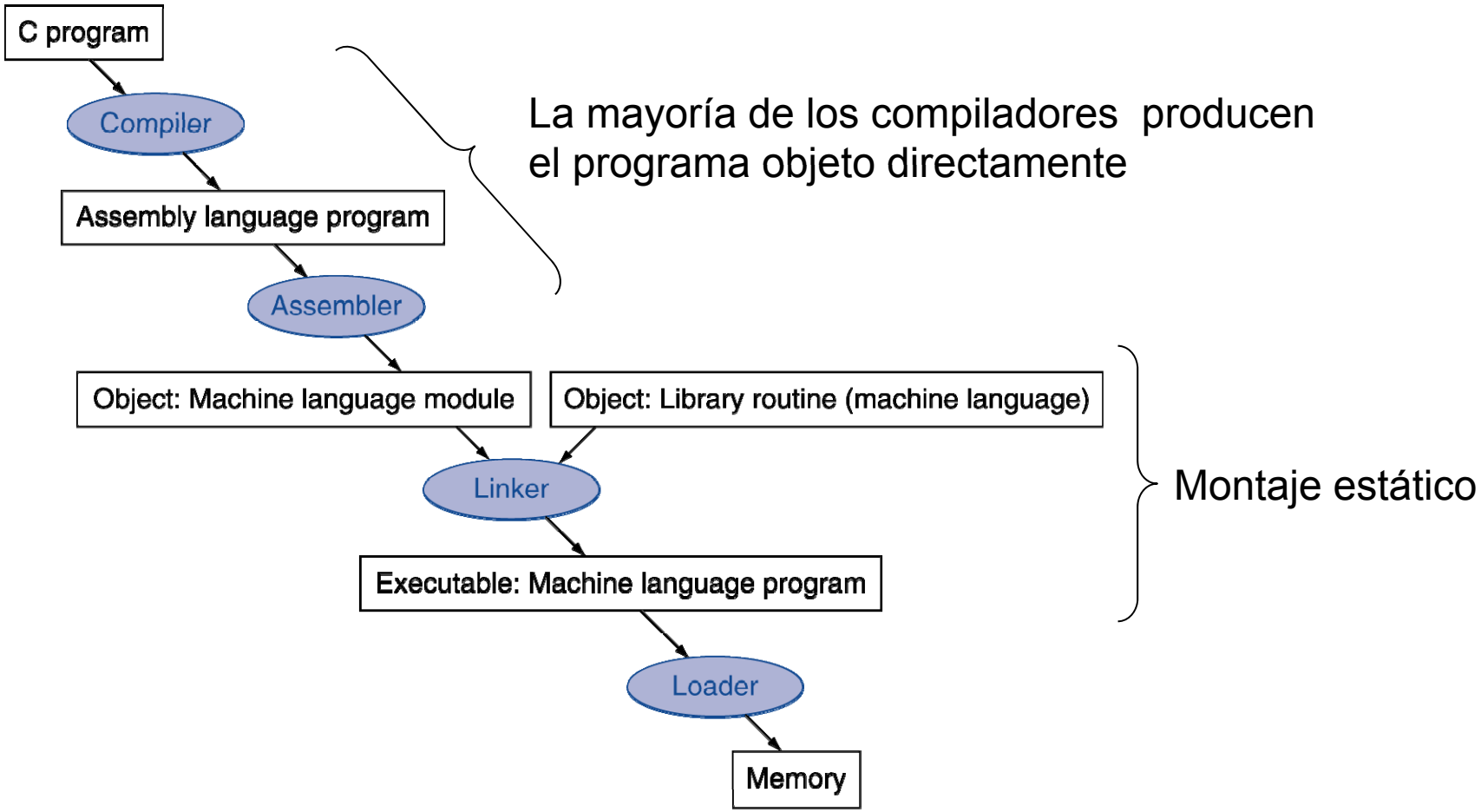
- **Cuando el destino de una bifurcación sea demasiado lejano como para codificarlo con un desplazamiento de 16 bits, hay que reescribir el código**
- **Ejemplo**

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

Resumen de los modos de direccionamiento



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

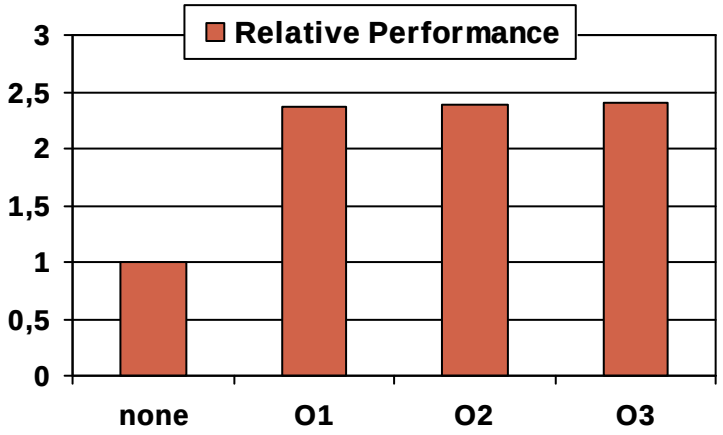
1. Lee la cabecera para determinar el tamaño de cada segmento
2. Crea un espacio de direcciones suficiente para el código y los datos del programa
3. Copia el código y los datos inicializados en la memoria
4. Copia los parámetros del programa principal (si los hay) en la pila
5. Inicializa registros (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 al procedimiento del sistema **`exit`** que devuelve el control al Sistema Operativo

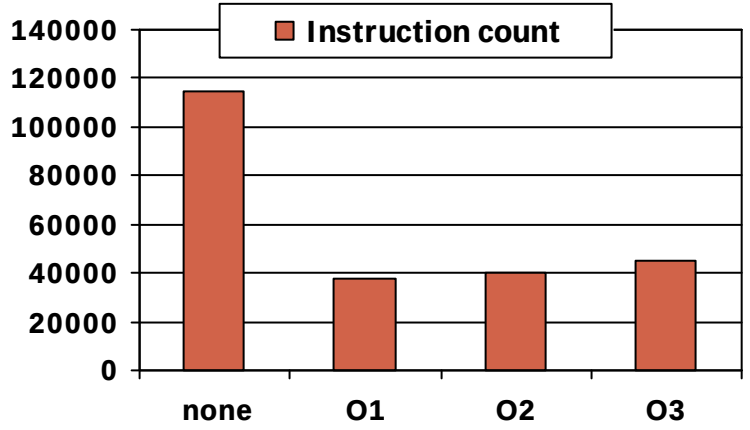
Montaje dinámico

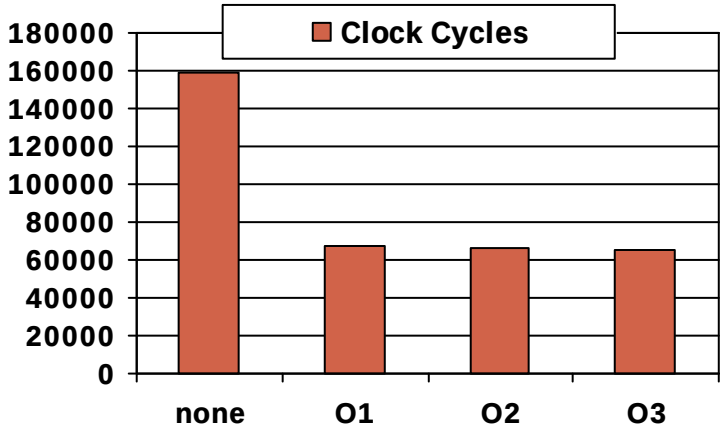
- **Sólo se cargan y se montan las bibliotecas de procedimiento si son invocadas**
  - Es necesario que el código del procedimiento sea relocalizable
  - Evita el aumento de tamaño del programa ejecutable producido por el montaje estático, que carga todas las bibliotecas independientemente de que se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

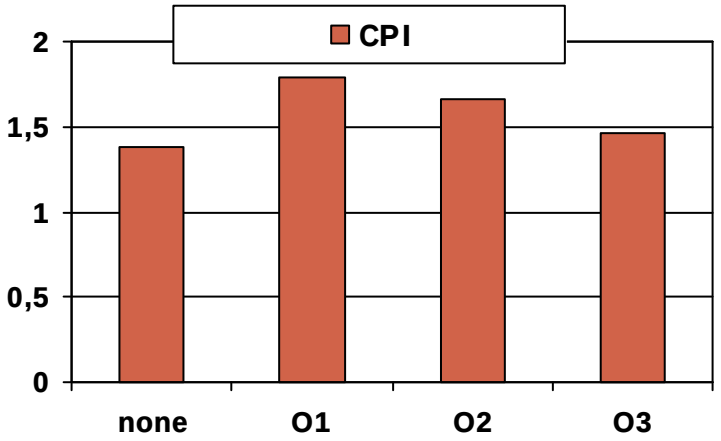
Efecto de la compilación optimizada

Compilado con gcc para Pentium 4 bajo Linux

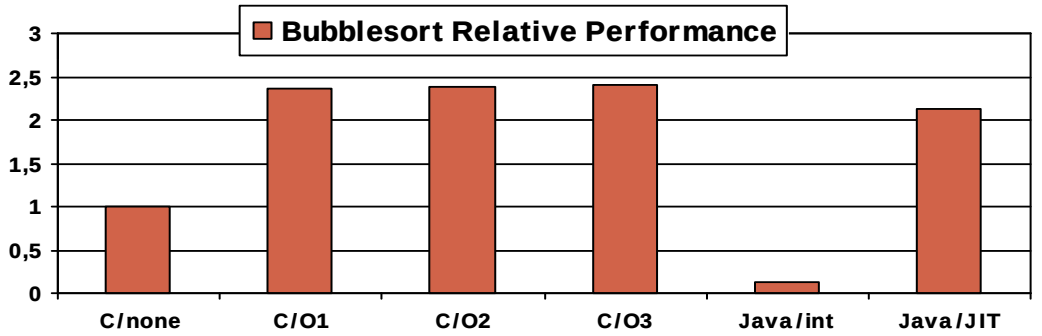


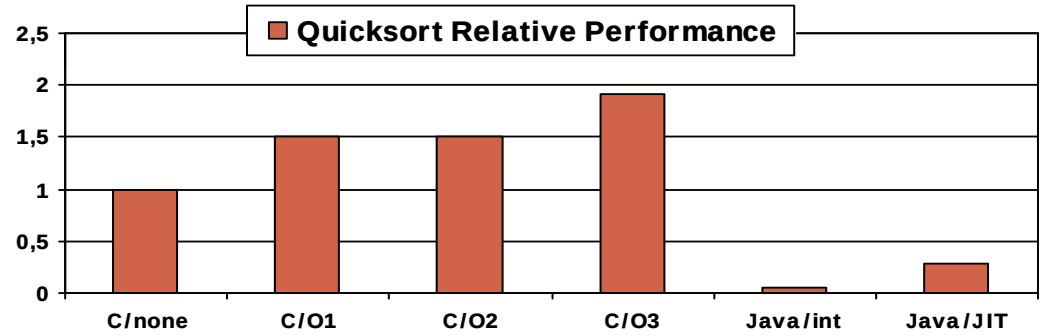


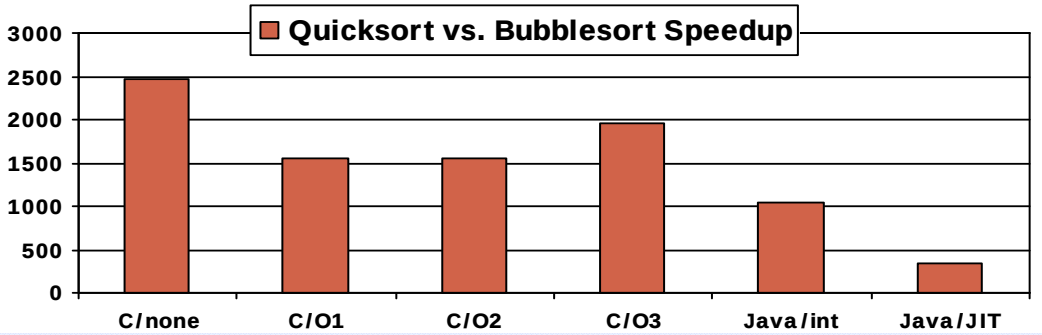




Efecto del lenguaje y del algoritmo







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;
}
```

```
        move $t0,$zero      # i = 0
loop1:  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,loop1 # if (…)
                            # goto loop1
```

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

```
        move $t0,$a0     # p = & array[0]
        sll $t1,$a1,2    # $t1 = size * 4
        add $t2,$a0,$t1  # $t2 =
                         #   &array[size]
loop2:  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,loop2 # if (…)
                            # goto loop2
```

Comparación entre vectores y apuntadores

- **En ambos casos las 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 enMIPS	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, srl`	12%	4%
Bifurcación condicional	`beq, bne, slt, slti, sltiu`	34%	8%
Saltos	`j, jr, jal`	2%	0%