

Grado en Ingeniería Informática  
2019-2020

*Apuntes*  
**Estructura de Computadores**

---

Jorge Rodríguez Fraile<sup>1</sup>



Esta obra se encuentra sujeta a la licencia Creative Commons  
**Reconocimiento - No Comercial - Sin Obra Derivada**

---

<sup>1</sup>Universidad: 100405951@alumnos.uc3m.es | Personal: jrf1616@gmail.com



## **ÍNDICE GENERAL**

<b>I   Información</b>	<b>3</b>
<b>II   Tema 1. Introducción a los computadores</b>	<b>47</b>
<b>III   Tema 2. Representación de la información</b>	<b>183</b>
<b>IV   Tema 3. Programación en ensamblador</b>	<b>339</b>
<b>V   Tema 4. El procesador</b>	<b>795</b>
<b>VI   Tema 5. Jerarquía de memoria</b>	<b>1005</b>
<b>VII   Tema 6. Sistemas de Entrada Salida</b>	<b>1185</b>



# **Parte I**

## **Información**



Grupo ARCOS

**uc3m | Universidad Carlos III de Madrid**

## Objetivos y presentación del curso

Estructura de Computadores  
Grado en Ingeniería Informática



# Estructura de Computadores en la UC3M

- ▶ Asignatura obligatoria de segundo curso de:
  - ▶ Grado en Ingeniería Informática
  - ▶ Doble Grado en Ingeniería Informática y Administración de Empresas

# Estructura de Computadores en la UC3M

---

2º

Estructura de Computadores

# Estructura de Computadores en la UC3M

1º

Programación

Tecnología de Computadores

Estructuras de Datos y Algoritmos

---

2º

Estructura de Computadores

# Estructura de Computadores en la UC3M

1º

Programación

Tecnología de Computadores

Estructuras de Datos y Algoritmos

---

2º

Estructura de Computadores

Sistemas Operativos

---

# Estructura de Computadores en la UC3M

1º

Programación

Tecnología de Computadores

Estructuras de Datos y Algoritmos

---

2º

Estructura de Computadores

Sistemas Operativos

---

3º

Arquitectura de Computadores

Redes de Ordenadores

Organización de Computadores

Sistemas Distribuidos

# Estructura de Computadores

1º

Programación

Tecnología de Computadores

Estructuras de Datos y Algoritmos

---

2º

Estructura de Computadores

Sistemas Operativos

---

3º

Arquitectura de Computadores

Redes de Ordenadores

Organización de Computadores

Sistemas Distribuidos

---

4º

Sistemas de Tiempo Real

Desarrollo de SW de sistemas

Panorámica de las Com. digitales

# Página Web de la asignatura

<http://www.arcos.inf.uc3m.es/~infoec/>

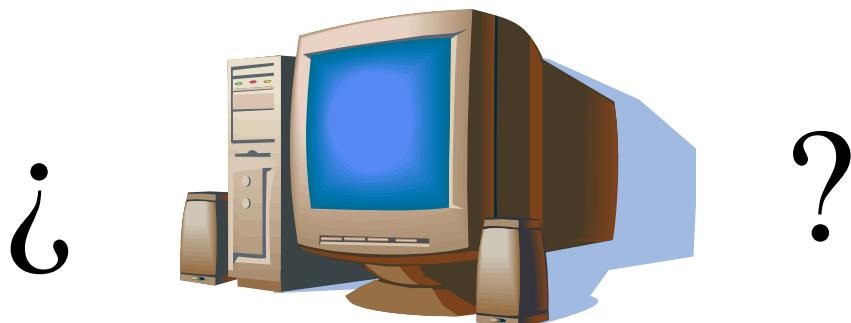
- Toda la información sobre la asignatura estará en esta página
- También en [Aula Global](#)

Profesor coordinador:

- Félix García Carballeira  
[felix.garcia@uc3m.es](mailto:felix.garcia@uc3m.es)



# Objetivos del curso



Conocer y entender los componentes y el funcionamiento básico de un computador

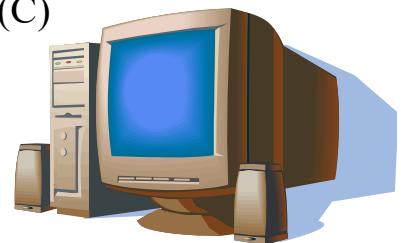
# Cualquier tipo de computador



# Entender cómo se ejecuta un programa

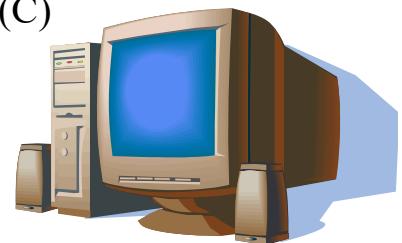
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Lenguaje de alto nivel (C)



# Entender cómo se ejecuta un programa

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;      Lenguaje de alto nivel (C)
```



```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```

Instrucciones máquina

# Entender cómo se ejecuta un programa

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

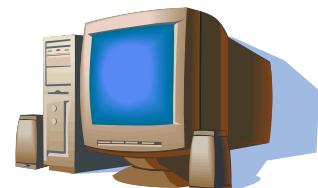
Lenguaje de alto nivel (C)

```
lw    $t0, 0($2)  
lw    $t1, 4($2)  
sw    $t1, 0($2)  
sw    $t0, 4($2)
```

Lenguaje ensamblador

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```

Instrucciones máquina



# Entender cómo se ejecuta un programa

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw    $t0, 0($2)  
lw    $t1, 4($2)  
sw    $t1, 0($2)  
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



Compilador

Ensamblador

# Entender cómo se ejecuta un programa

## Ejemplo 1

```
int n;  
n = 40000;  
printf( "%d \n", n *n );  
  
n = 50000;  
printf( "%d \n", n *n );
```

- ▶ ¿Es correcta esta salida?

1600000000

-1794967296

## Entender cómo se ejecuta un programa Ejemplo 2

```
float x, y , z;  
  
x = 1.0e20;  
y = -1.0e20;  
z = 3.14;  
  
printf( "%f\n" , (x + y) + z );  
printf( "%f\n" , x + (y + z) );
```

- ▶ ¿Se cumple  $(x+y) + z == x + (y+z)$ ?

# Entender cómo se ejecuta un programa

## Ejemplo 3

### ▶ Código 1

```
int a[N][N]
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        sum = sum + a[i][j];
```

### ▶ Código 2

```
int a[N][N]
for (j=0; j < N; j++)
    for (i=0; i < N; i++)
        sum = sum + a[i][j];
```

- ▶ ¿Hacen lo mismo?
- ▶ ¿Tardan lo mismo en ejecutarse?

## Entender cómo se ejecuta un programa Ejemplo 4

```
#include <stdio.h>

#define BLOCK_SIZE    512

void main(int argc, char **argv)
{
    int fde, fds;
    char buffer[BLOCK_SIZE];
    int n;

    fde = open(argv[1], 0);
    fds = creat(argv[2], 0666);

    while((n = read(fde, buffer, BLOCK_SIZE)) > 0)
        write(fds, buffer, n);

    close(fde);
    close(fds);

    return;
}
```

¿Qué ocurre si BLOCK\_SIZE = 8192?

## Entender cómo se ejecuta un programa

### Ejemplo 5

¿Dado el siguiente fragmento?

```
if (i == (int)((float) i))  
{  
    printf("true");  
}
```

¿Se ejecuta siempre la función printf( )?

## Entender cómo se ejecuta un programa Ejemplo 6

- ▶ ¿Se puede intercambiar el valor de dos variables sin usar una variable intermedia?
  
- ▶ ¿Cómo se puede saber si el número de bits igual a 1 de una variable long de Java es par (de forma eficiente)?

## Entender cómo se ejecuta un programa Ejemplo 7

### ► ¿Son correctas las siguientes afirmaciones?

Un programa escrito en lenguaje máquina/ensamblador es más eficiente que un programa escrito en un lenguaje de alto nivel como C

Un programa siempre ejecutará más rápido cuanto más cores/núcleos tenga el procesador

## Ejemplo 8

¿Funciona correctamente este programa?

```
public class Stack {
    private             Object[] elements;
    private             int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

22

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

## Ejemplo 8

¿Funciona correctamente este programa?

```
public class Stack {  
    private Object[] elements;  
    private int size = 0;  
    private static final int DEFAULT_INITIAL_CAPACITY = 16;  
  
    public Stack() {  
        elements = new Object[DEFAULT_INITIAL_CAPACITY];  
    }  
  
    public void push(Object e) {  
        ensureCapacity();  
        elements[size++] = e;  
    }  
                                            Memory Leaks  
  
    public Object pop() {  
        if (size == 0)  
            throw new EmptyStackException();  
        return elements[--size];  
    }  
  
    private void ensureCapacity() {  
        if (elements.length == size)  
            elements = Arrays.copyOf(elements, 2 * size + 1);  
    }  
}
```

# Ejemplo 8

## ¿Funciona correctamente este programa?

```
public class Stack {
    private             Object[] elements;
    private             int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
        elements[size] = null; // Eliminate obsolete reference
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

## Ejemplo 9

- ▶ ¿Qué procesador es más rápido?



**Intel Atom x7-Z8700**

<b>1.60 GHz</b>	Frequency	<b>1.50 GHz</b>
<b>2.40 GHz</b>	Turbo (1 Core)	
<b>2.40 GHz</b>	Turbo (All Cores)	
<b>4</b>	Cores	<b>2</b>
<b>No</b>	Hyperthreading	<b>Yes</b>
<b>No</b>	Overclocking ?	<b>No</b>
<b>2 MB</b>	Cache	<b>3 MB</b>

**Intel Core i3-4020Y**

# Temario

**Tema 1. Introducción a los computadores**

**Tema 2. Representación de la información**

**Tema 3. Fundamentos de la programación en ensamblador**

**Tema 4. El procesador**

**Tema 5. Sistemas de memoria**

**Tema 6. Sistemas de Entrada/salida**

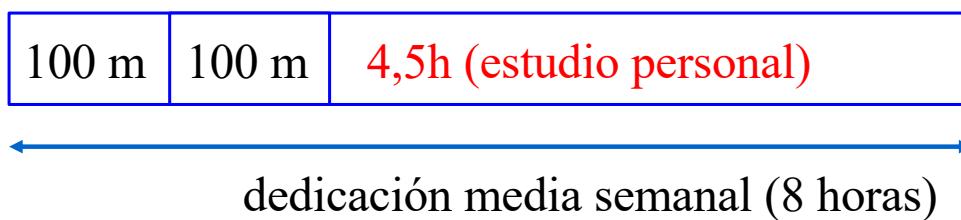
# Desarrollo del curso

- ▶ **14 semanas**
  - ▶ **I sesión por semana de 100 minutos en grupo magistral**
  - ▶ **I sesión por semana de 100 minutos en grupo reducido**
    - ▶ **4 de estas sesiones son de laboratorio**
  - ▶ **I sesión adicional de 100 minutos en la última semana**

# Desarrollo del curso

## ► 14 semanas

- ▶ 1 sesión por semana de 100 minutos en grupo magistral
- ▶ 1 sesión por semana de 100 minutos en grupo reducido
  - ▶ 4 de estas sesiones son de laboratorio
- ▶ 1 sesión adicional de 100 minutos en la última semana



# Evaluación

**60 % mediante evaluación continua**

**30% Prácticas**

**30% Pequeños exámenes**

**40 % mediante un examen final**

# Evaluación continua

## ► Prácticas: 30%

- ▶ Se realizarán DOS prácticas obligatorias
  - ▶ Nota mínima de cada práctica: 2
  - ▶ Nota mínima media de todas las prácticas: 4
- ▶ Pesos de cada práctica:
  - ▶ Práctica 1: 15%
  - ▶ Práctica 2: 15%
- ▶ Se realizarán en grupos de dos alumnos
- ▶ En caso de que se detecte copia de prácticas, a ambas partes implicadas (copiados y copiadores) se les calificará con un 0

# Evaluación continua

## ► Pequeños exámenes en cada grupo reducido: 30%

- Se realizarán **4 exámenes pequeños** (15-20 minutos)
- En el que se evaluarán **todos** los conocimientos adquiridos por el alumno hasta ese momento
- Se hará la media de las **3 mejores notas**
  
- No se repetirá ningún examen.
- Salvo causa médica justificada con suficiente antelación, no se admitirá que un alumno realice el examen en un grupo distinto al que está matriculado.

## Evaluación continua

- ▶ Un alumno **sigue el proceso de evaluación continua** cuando:
  - ▶ Entrega **todas** las prácticas
    - ▶ Nota mínima de cada práctica: **2**
    - ▶ Nota mínima de todas las prácticas: **4**

# Examen final de la convocatoria ordinaria

- ▶ El examen final incluirá todo el contenido de la asignatura (tanto teórico como práctico).
  - ▶ Nota mínima: 4
- ▶ Para aquellos alumnos que hayan seguido el proceso de evaluación continua, el valor de este examen supondrá el 40% de la nota final. El otro 60% corresponderá a la nota obtenida en la evaluación continua.
- ▶ Para aquellos alumnos que no hayan completado el proceso de evaluación continua, el examen final tendrá un valor del 60% de la calificación total de la asignatura. Por tanto, para poder aprobar, el alumno deberá obtener una calificación superior a 8.33 sobre 10 en este examen

# Convocatoria extraordinaria

El **examen final** incluirá **todo** el contenido de la asignatura (tanto teórico como práctico).

**Nota mínima: 4**

Si el estudiante siguió el proceso de evaluación continua, el examen final de esta convocatoria tendrá un peso del 40% y la calificación final tendrá en cuenta el otro 60% obtenido en el proceso de evaluación continua.

Si el estudiante no completó el proceso de evaluación continua el examen de esta convocatoria tendrá un valor del 100% de la calificación final de la asignatura.

Aunque el estudiante hubiera seguido el proceso de evaluación continua, tendrá derecho a ser calificado en la forma indicada en el apartado anterior cuando le resulte más favorable.

## Calificación como NO PRESENTADO

- ▶ Un estudiante aparecerá como **no presentado** cuando no realice el correspondiente examen final.
  - ▶ Aunque ya se encuentre aprobado por evaluación continua

## Nota final

- ▶ La nota final se incrementará en **I punto** a aquellos alumnos que realicen:
  - ▶ **Todas** las pruebas de evaluación continua.
    - ▶ Todos los exámenes pequeños (**los 4**)
    - ▶ Todas las prácticas
  - ▶ Obtengan más de un 7 sobre 10 de calificación en la evaluación continua y al menos 4 puntos en el examen final.

# Importancia de la evaluación continua

	2014-2015	2015-2016	2016-2017	2017-2018	2018-2019
Alumnos que siguen la evaluación continua	78%	74%	74%	71%	78%
Alumnos que aprueban la evaluación continua	74%	64%	67%	66%	76%
Alumnos que aprueban la evaluación continua respecto de los que la siguen	87%	87%	90%	89%	89%
Alumnos que aprobaron al final la asignatura aunque abandonaron la evaluación continua	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %
Alumnos que aprobaron la evaluación continua y han aprobado la asignatura al final	92%	89%	85%	86%	94%
Alumnos que completaron y suspendieron la evaluación continua han aprobado la asignatura al final	6%	7%	4%	6%	2%
<b>Alumnos aprobados</b>	<b>67%</b>	<b>66%</b>	<b>65%</b>	<b>62%</b>	<b>78%</b>
<b>Alumnos no presentados</b>	<b>23%</b>	<b>20%</b>	<b>22%</b>	<b>26%</b>	<b>15%</b>
<b>Alumnos suspensos</b>	<b>10%</b>	<b>14%</b>	<b>13%</b>	<b>12%</b>	<b>7%</b>

## Bibliografía

- ▶ Problemas resueltos de Estructura de Computadores. 2<sup>a</sup> edición  
F. García, J. Carretero, J. D. García  
D. Expósito  
2015



- ▶ Computer Organization and Design  
The Hardware/Software Interface  
D.A. Patterson, J. Hennessy  
Quinta edición  
2014

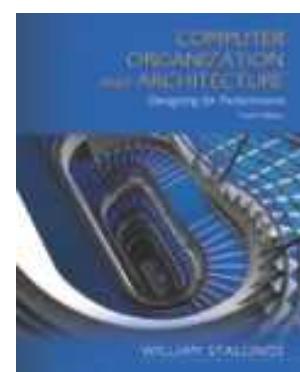


# Bibliografía

- ▶ **Fundamentos de Sistemas Digitales**  
Thomas L. Floyd  
Pearson 2016



- ▶ **Computer Organization and Architecture**  
Décima edición  
William Stallings  
Pearson 2016



## Materiales complementarios

- ▶ [Computer History Museum](#)
- ▶ [Museo histórico de la Informática, Universidad Politécnica de Madrid](#)
- ▶ [Museo virtual de la Informática. Universidad de Castilla-la Mancha](#)
- ▶ <https://www.computer.org/cms/Computer.org/Publications/timeline.pdf>
- ▶ [The EDSAC Simulator](#)
- ▶ [IBM Archives](#)
- ▶ [Charles Babbage Institute](#)

# ¿Por qué estudiar Estructura de Computadores? W. Stallings

El «IEEE/ACM Computer Curricula 2001» [JTF01], preparado por la *Joint Task Force* de currículo de computadores de la Sociedad de Computadores IEEE (*Institute of Electrical and Electronics Engineers*) y la ACM (*Association for Computing Machinery*), citan la arquitectura de computadores como uno de los temas troncales que debe estar en todos los currículos de todos los estudiantes de licenciatura e ingeniería informática. El informe dice lo siguiente:

«El computador está en el corazón de la informática. Sin él la mayoría de las asignaturas de informática serían hoy una rama de la matemática teórica. Para ser hoy un profesional en cualquier campo de la informática uno no debe ver al computador como una caja negra que ejecuta programas mágicamente. Todos los estudiantes de informática deben, en cierta medida, comprender y valorar los componentes funcionales de un computador, sus características, su funcionamiento y sus interacciones. También sus implicaciones prácticas. Los estudiantes necesitan comprender la arquitectura del computador para estructurar un programa de forma que este sea más eficiente en una máquina real. Seleccionando el sistema que se va a usar, debe ser capaz de comprender el compromiso entre varios componentes, como la velocidad del reloj de la CPU frente al tamaño de la memoria».

# ¿Por qué estudiar Estructura de Computadores? W. Stallings

En [CLEM00] se dan los siguientes ejemplos como razones para estudiar arquitectura de computadores:

1. Supóngase que un licenciado trabaja en la industria y se le pide seleccionar el computador con la mejor relación calidad precio para utilizarlo en una gran empresa. Comprender las implicaciones de gastar más en distintas alternativas, como una caché grande o una velocidad de reloj mayor, es esencial para tomar esta decisión.
2. Hay muchos procesadores que no forman parte de equipos PC o servidores, pero sí en sistemas embebidos. Un diseñador debe ser capaz de programar un procesador en C que esté embebido en algún sistema en tiempo real o sistema complejo, como un controlador electrónico de un coche inteligente. Depurar el sistema puede requerir utilizar un analizador lógico que muestre la relación entre las peticiones de interrupción de los sensores del sistema y el código máquina.
3. Los conceptos utilizados en arquitectura de computadores tienen aplicación en otros cursos. En particular, la forma en la que el computador ofrece un soporte arquitectural a los lenguajes de programación y funciones en principio propias del sistema operativo, refuerza los conceptos de estas áreas.

## **Parte II**

### **Tema 1. Introducción a los computadores**



Grupo ARCOS

**uc3m | Universidad Carlos III de Madrid**

# Tema 1

## Introducción a los computadores

Estructura de Computadores  
Grado en Ingeniería Informática



# Contenidos

1. **¿Qué es un computador?**
2. **Concepto de estructura y arquitectura**
3. **Elementos constructivos de un computador**
4. **Computador Von Neumann**
5. **Instrucciones máquina y programación**
6. **Fases de ejecución de una instrucción**
7. **Parámetros característicos de un computador**
8. **Tipos de computadores**
9. **Evolución histórica**

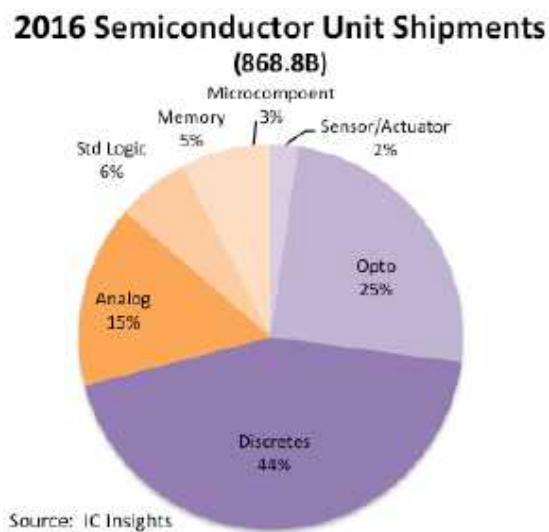
# ¿Qué aspecto tiene un computador?



# ¿Qué aspecto tiene un computador?

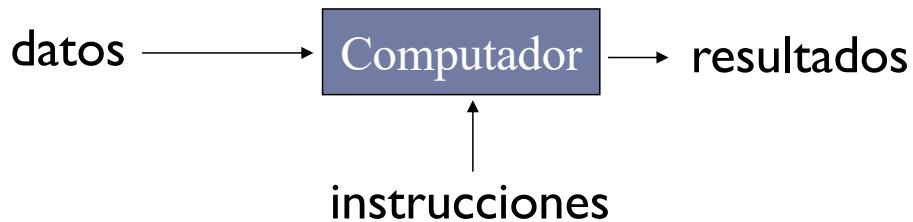


# Industria de los semiconductores



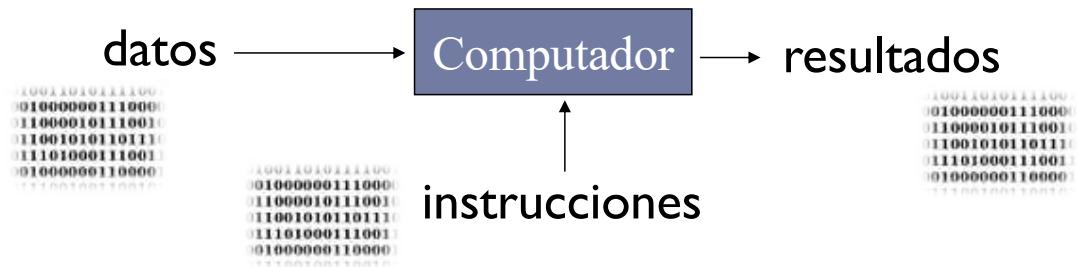
Procesadores:  
**3% de la industria**

# ¿Qué es un computador?



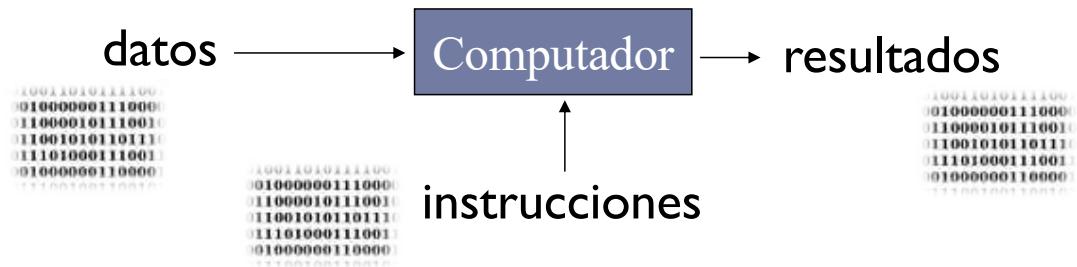
- ▶ **Computador: máquina destinada a procesar datos.**
  - ▶ Sobre ellos se aplican unas instrucciones obteniendo después unos resultados (datos/información)

# ¿Qué es un computador?



- ▶ **Computador: máquina destinada a procesar datos.**
  - ▶ Computador digital: datos e instrucciones en formato binario.

# ¿Qué es un computador?



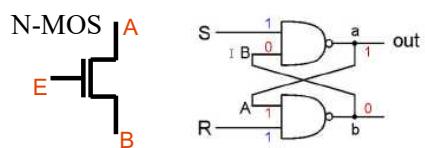
- ▶ **Computador: máquina destinada a procesar datos.**
  - ▶ Computador digital: datos e instrucciones en formato binario.
  - ▶ Matemáticamente se puede representar como:

$$f : \{0,1\}^n \rightarrow \{0,1\}^m$$

# Contenidos

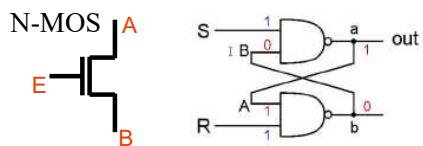
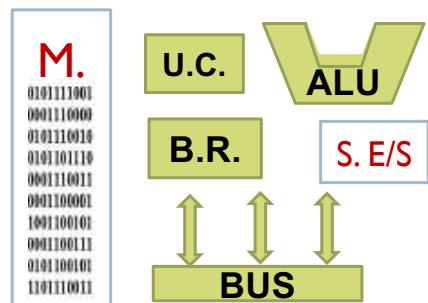
1. **¿Qué es un computador?**
2. **Concepto de estructura y arquitectura**
3. Elementos constructivos de un computador
4. Computador Von Neumann
5. Instrucciones máquina y programación
6. Fases de ejecución de una instrucción
7. Parámetros característicos de un computador
8. Tipos de computadores
9. Evolución histórica

¿Qué aspectos hay que conocer en un computador?



▶ **Tecnología:**  
▶ Cómo se construyen los componentes

¿Qué aspectos hay que conocer en un computador?



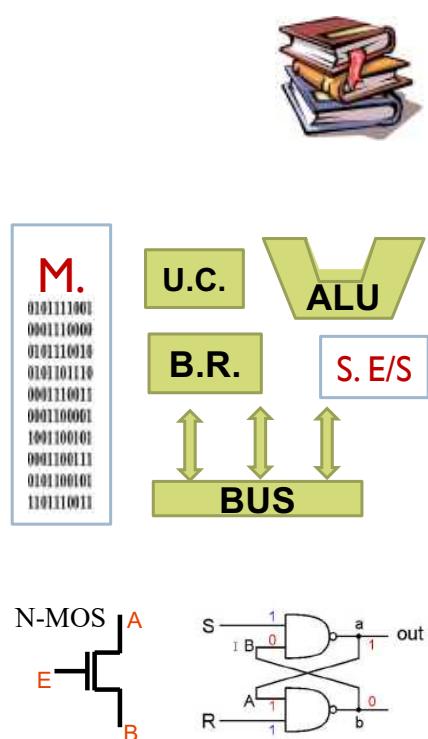
## ► Estructura:

- Componentes y su organización

## ► Tecnología:

- Cómo se construyen los componentes

¿Qué aspectos hay que conocer en un computador?



### ► Arquitectura:

- Atributos visibles para un programador

### ► Estructura:

- Componentes y su organización

### ► Tecnología:

- Cómo se construyen los componentes

# Arquitectura de un computador

## ▶ Atributos visibles para un programador

- ▶ Juego de instrucciones que ofrece la máquina (ISA, Instruction Set Architecture)
- ▶ Tipo y formato de datos que es capaz de utilizar el computador
- ▶ Número y tamaño de los registros
- ▶ Técnicas y mecanismos de E/S
- ▶ Técnicas de direccionamiento de la memoria

# Ejercicio

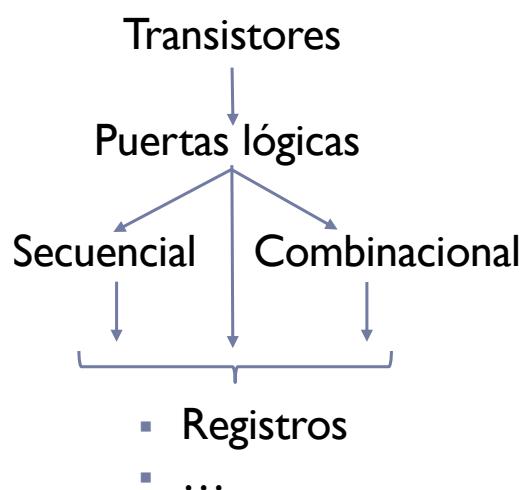
- ▶ **¿Qué es un computador?**
- ▶ **¿Qué aspecto tiene un computador?**
- ▶ **¿Qué aspectos de un computador se han de conocer?**

# Contenidos

1. **¿Qué es un computador?**
2. Concepto de estructura y arquitectura
3. **Elementos constructivos de un computador**
4. Computador Von Neumann
5. Instrucciones máquina y programación
6. Fases de ejecución de una instrucción
7. Parámetros característicos de un computador
8. Tipos de computadores
9. Evolución histórica

# Repasso

- ▶ Sistema digital basado en: 0 y 1
- ▶ Elementos constructivos: transistores, puertas lógicas, ....



# Sistema binario

## ▶ Binario

$$X = \begin{matrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ \dots & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{matrix}$$

dígito binario  $d_i$   
Peso  $p_i$

$$\text{▶ Valor} = d_{31} \times 2^{31} + d_{30} \times 2^{30} + \dots + d_1 \times 2^1 + d_0 \times 2^0$$

# Sistema binario

## ▶ Binario

$$X = \begin{array}{cccccccccc} 1 & 0 & 1 & 0 & 0 & 1 & 0 & \text{dígito binario } d_i \\ \dots & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \text{Peso } p_i \end{array}$$

$$\text{▶ Valor} = d_{31} \times 2^{31} + d_{30} \times 2^{30} + \dots + d_1 \times 2^1 + d_0 \times 2^0$$

- ▶ ¿Cuántos valores se pueden representar con **n bits**?
- ▶ ¿Cuántos bits se necesitan para representar **m ‘valores’**?
- ▶ Con **n bits**, si los valores a representar son números y comienzo en el 0, ¿Cuál es el **máximo valor representable**?

# Sistema binario

## ▶ Binario

$$X = \begin{array}{cccccccccc} 1 & 0 & 1 & 0 & 0 & 1 & 0 & \text{dígito binario } d_i \\ \dots & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \text{Peso } p_i \end{array}$$

$$\text{▶ Valor} = d_{31} \times 2^{31} + d_{30} \times 2^{30} + \dots + d_1 \times 2^1 + d_0 \times 2^0$$

- ▶ ¿Cuántos valores se pueden representar con **n bits**?  **$2^n$**
- ▶ ¿Cuántos bits se necesitan para representar **m ‘valores’**?  **$\log_2(m)$  por exceso**
- ▶ Con **n bits**, si los valores a representar son números y comienzo en el 0, ¿Cuál es el **máximo valor representable**?  **$2^n - 1$**

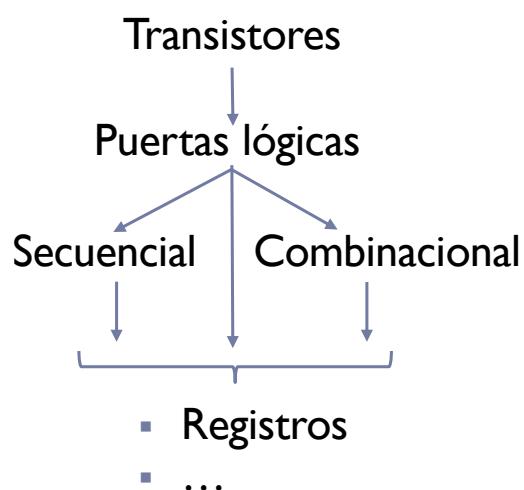
## Ejercicio

- ▶ ¿Cuántos códigos distintos se pueden codificar con 8 bits?
- ▶ ¿Cuántos bits hacen falta para representar 512 códigos?

# Repasso

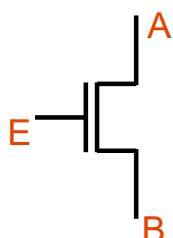
- ▶ Sistema digital basado en: 0 y 1

- ▶ Elementos constructivos: transistores, puertas lógicas, ....

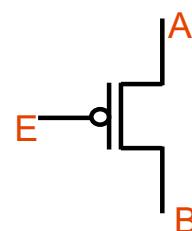


# Transistor

N-MOS



P-MOS

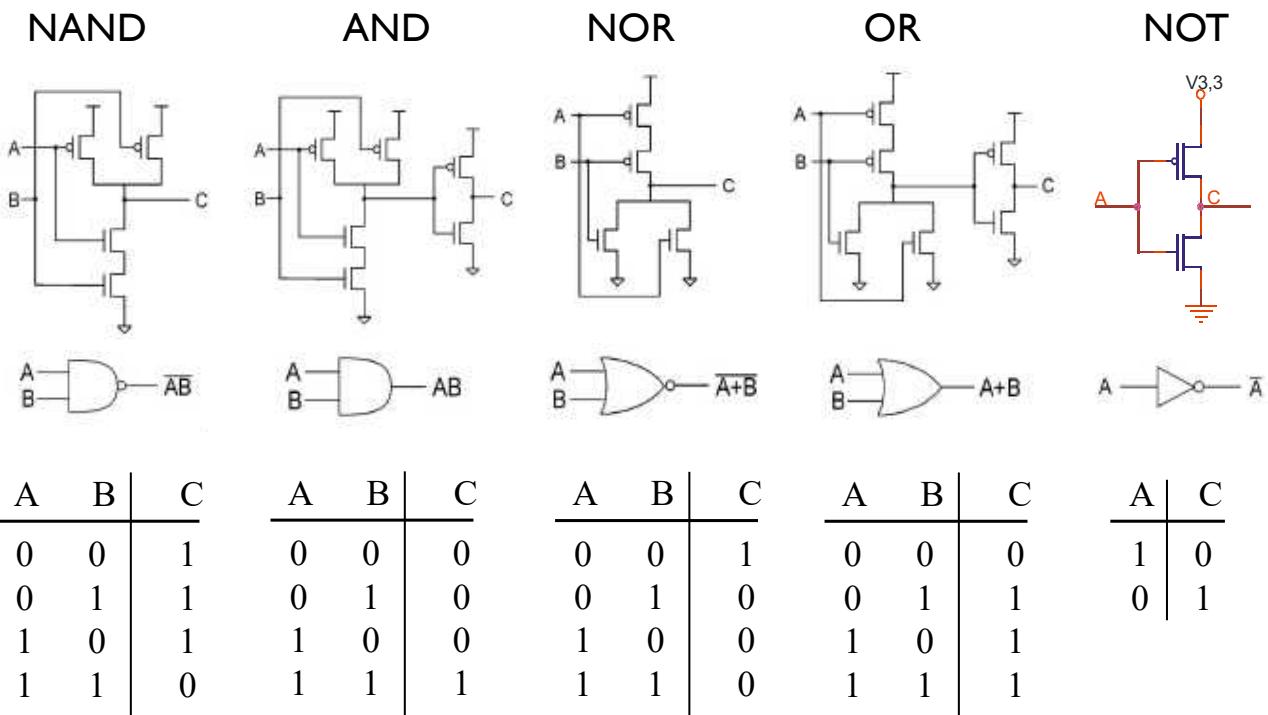


E	Funcionamiento
1	Conecta A con B (circuito abierto)
0	No conecta A con B (circuito cerrado)

E	Funcionamiento
0	Conecta A con B (circuito abierto)
1	No conecta A con B (circuito cerrado)

- ▶ Un transistor actúa como un interruptor
- ▶ Los transistores tipo p y n son transistores de tipo MOSFET (Metal-Oxide-Semiconductor-Field-Effect Transistor)
- ▶ La combinación de transistores tipo p y n dan lugar a la familia CMOS

# Puertas lógicas



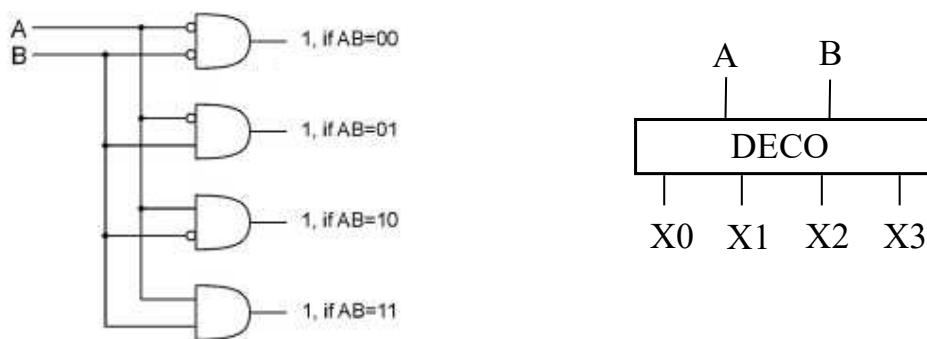
# Circuitos combinacionales

- ▶ La salida depende solo de los valores de entrada
- ▶ Ejemplos:
  - ▶ Decodificadores
  - ▶ Multiplexores
  - ▶ Operadores aritméticos y lógicos

# Decodificadores

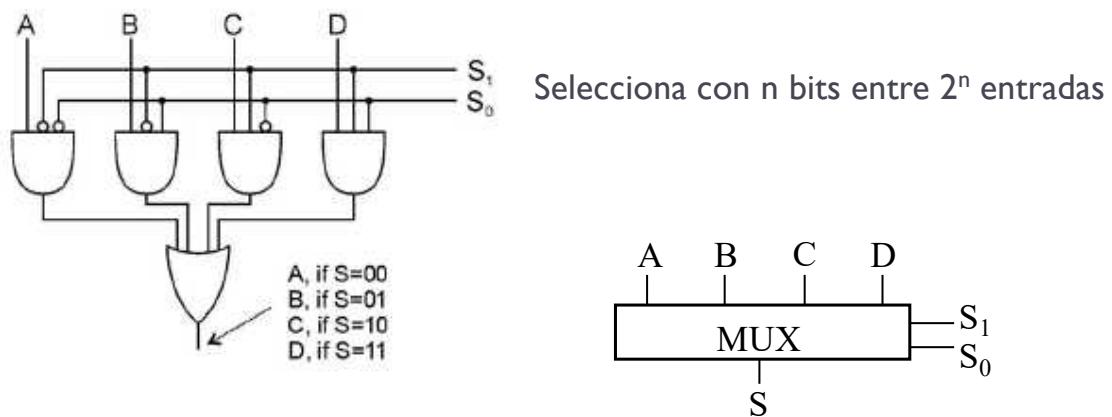
- ▶ Transforma un valor codificado en la activación de una señal de salida
  - ▶ Los codificadores realizan el proceso inverso

$n$  entradas,  $2^n$  salidas



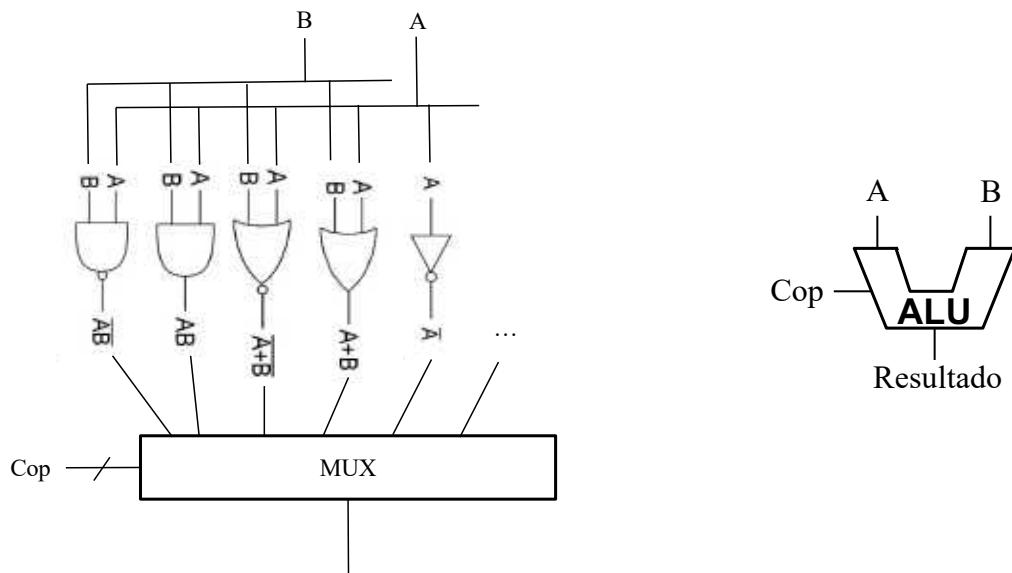
# Multiplexores

- ▶ Selecciona una de las entradas y copia su valor a la salida
  - ▶ Los demultiplexores realizan el proceso inverso
- ▶ Con N entradas se necesitan  $\log_2 N$  señales de control



# ALU. Unidades aritmético-lógicas

- ▶ Realiza una operación aritmético-lógica

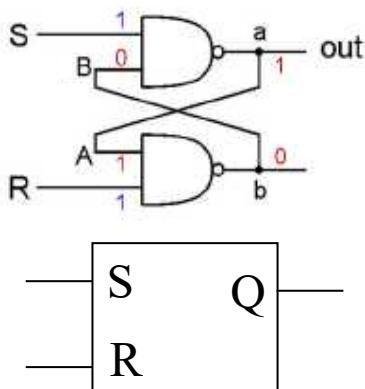


# Circuitos secuenciales

- ▶ La salida depende de los valores de entrada y del estado actual
  - ▶ Necesitan almacenar estado

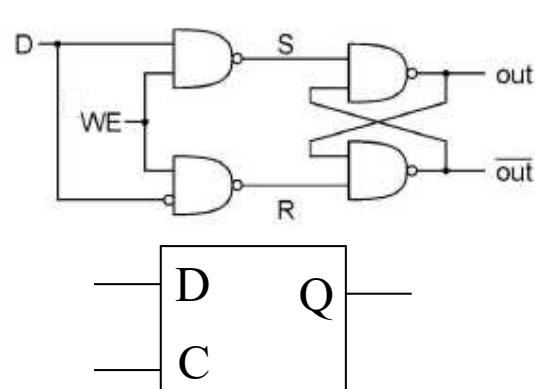
BIESTABLE R-S

Almacena un bit



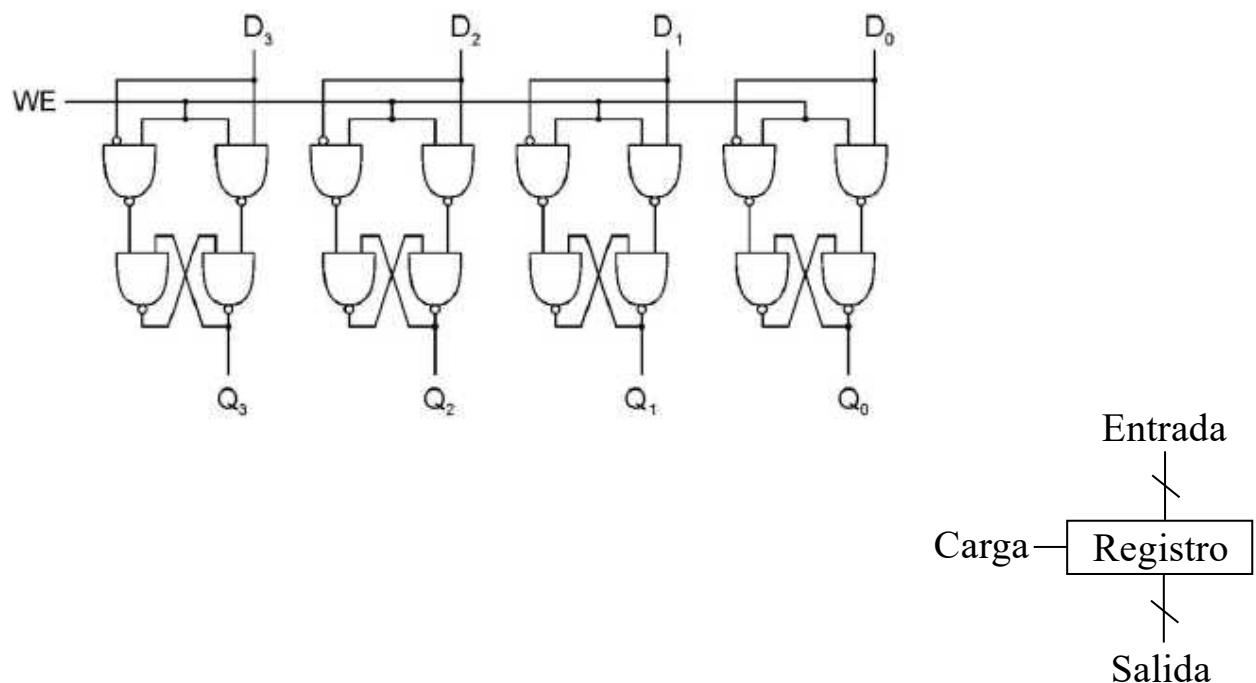
BIESTABLE D

Almacena un bit



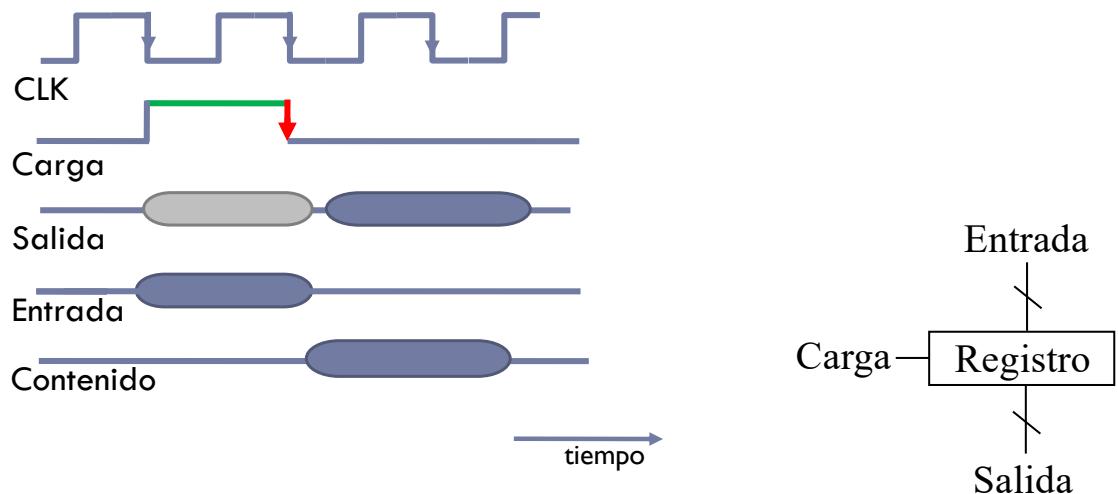
# Registro

- ▶ Elemento que almacena n bits (a la vez)



# Registro

- ▶ Elemento que almacena n bits (a la vez)
  - ▶ Durante el **nivel de Carga** el registro tiene el valor antiguo
  - ▶ En el **flanco de Carga** se almacena el valor en la entrada



# Contenidos

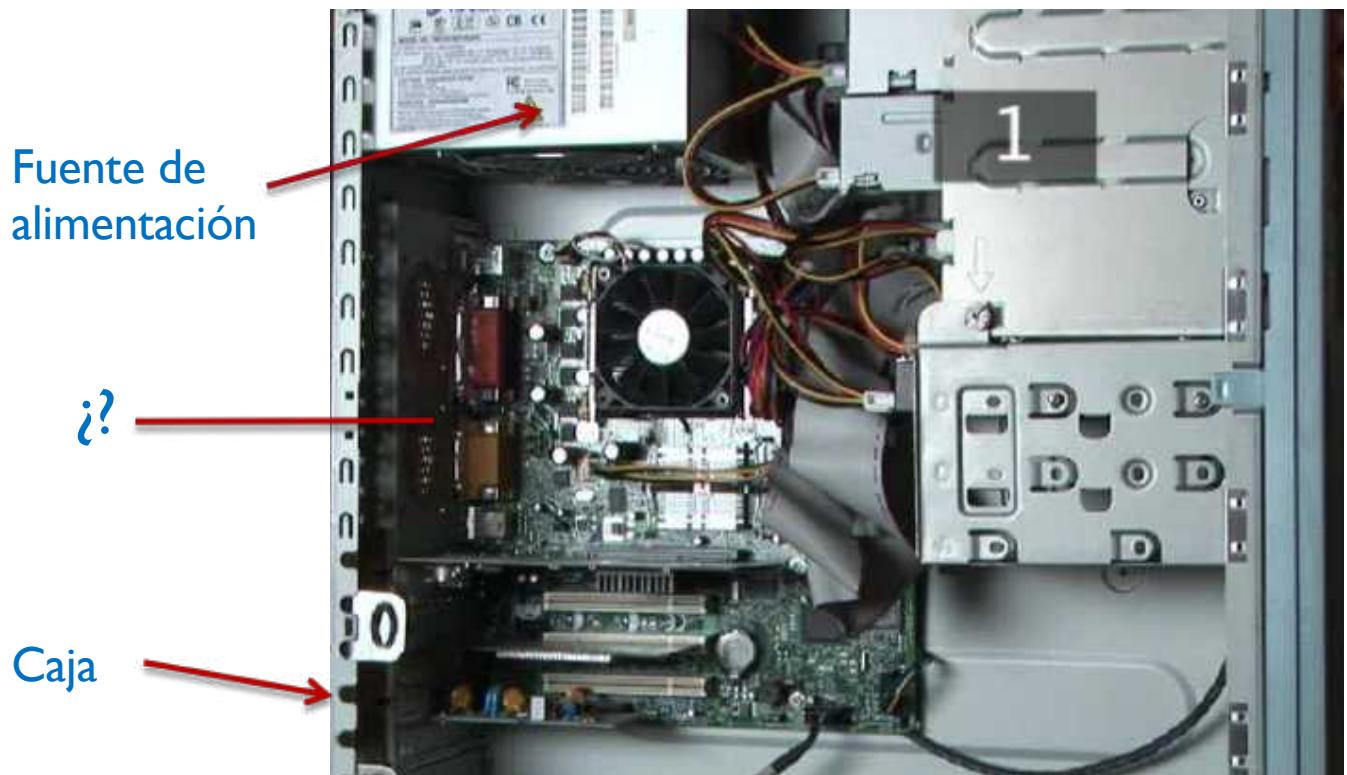
1. **¿Qué es un computador?**
2. **Concepto de estructura y arquitectura**
3. **Elementos constructivos de un computador**
4. **Computador Von Neumann**
5. **Instrucciones máquina y programación**
6. **Fases de ejecución de una instrucción**
7. **Parámetros característicos de un computador**
8. **Tipos de computadores**
9. **Evolución histórica**

¿Podemos distinguir los componentes internos al abrir un ordenador personal?



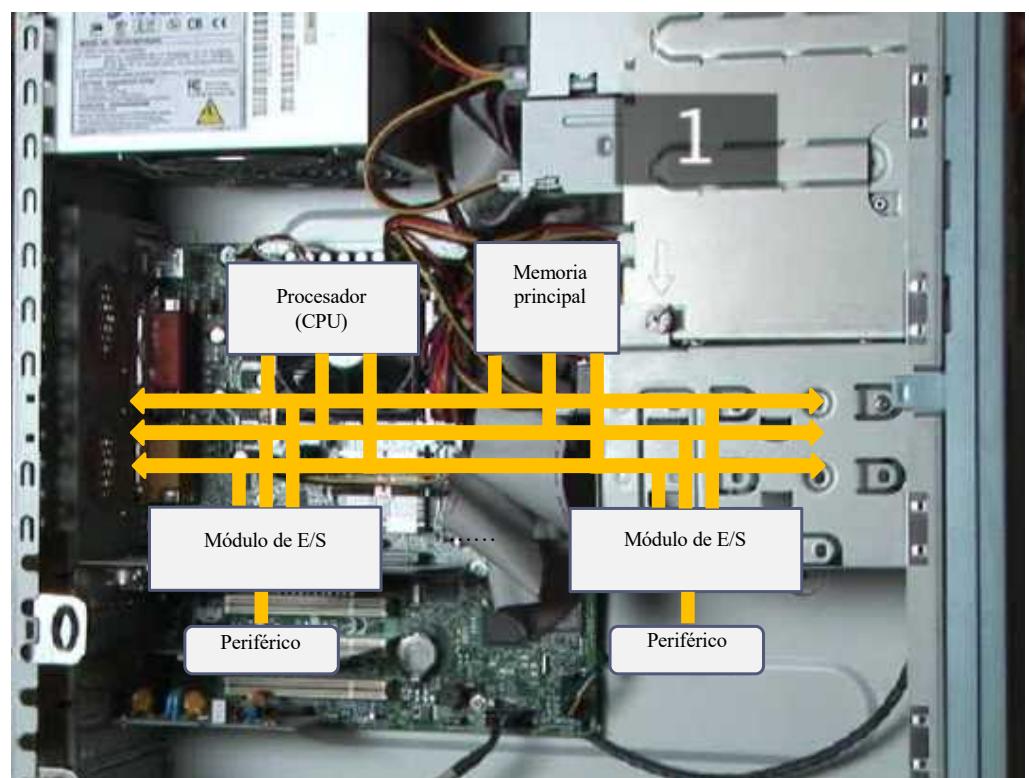
<http://www.videojug.com/film/what-components-are-inside-my-computer>

¿Podemos distinguir los componentes internos al abrir un ordenador personal?



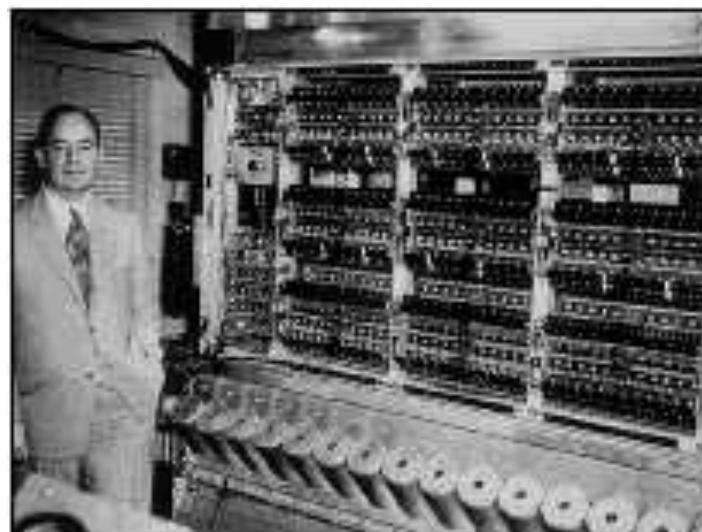
<http://www.videojug.com/film/what-components-are-inside-my-computer>

# Modelo usado como base



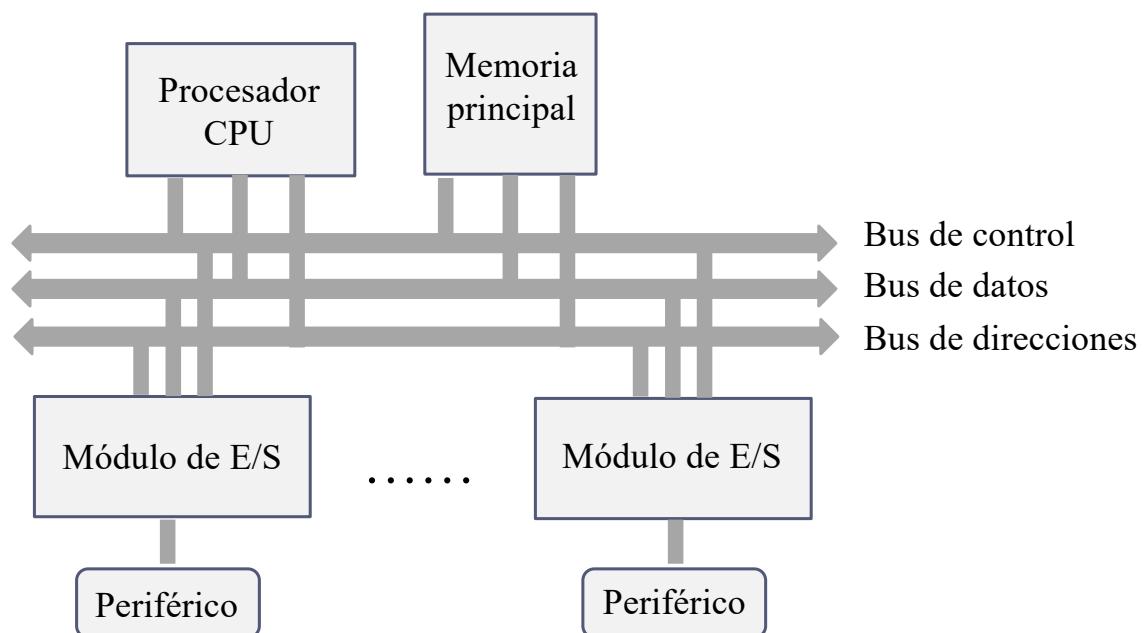
<http://www.videojug.com/film/what-components-are-inside-my-computer>

# Computador Von Neumann

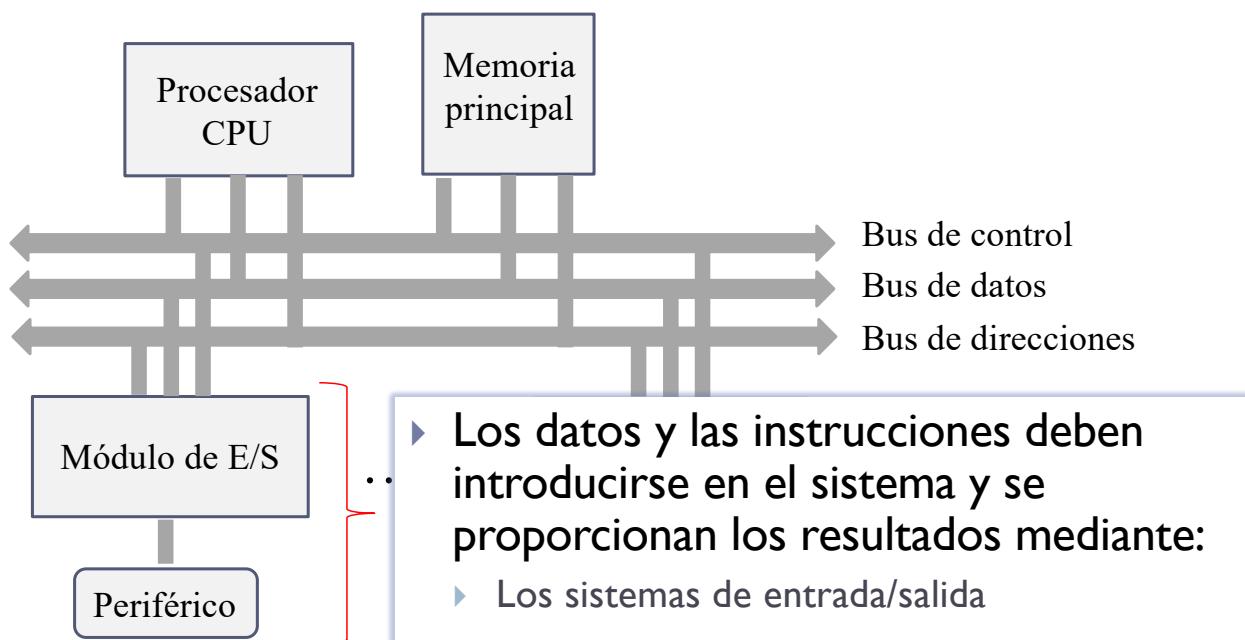


Máquina capaz de ejecutar una serie de instrucciones elementales (instrucciones máquina) que están almacenadas en memoria (son leídas y ejecutadas)

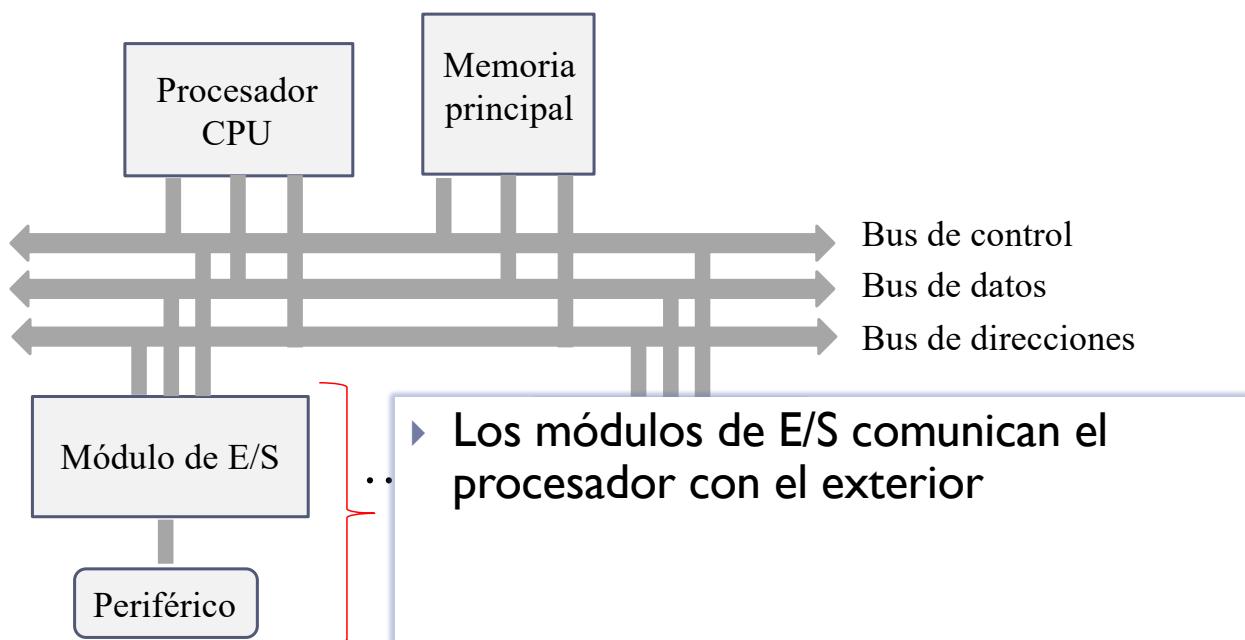
# Arquitectura Von Neumann



# Arquitectura Von Neumann (1 / 4)



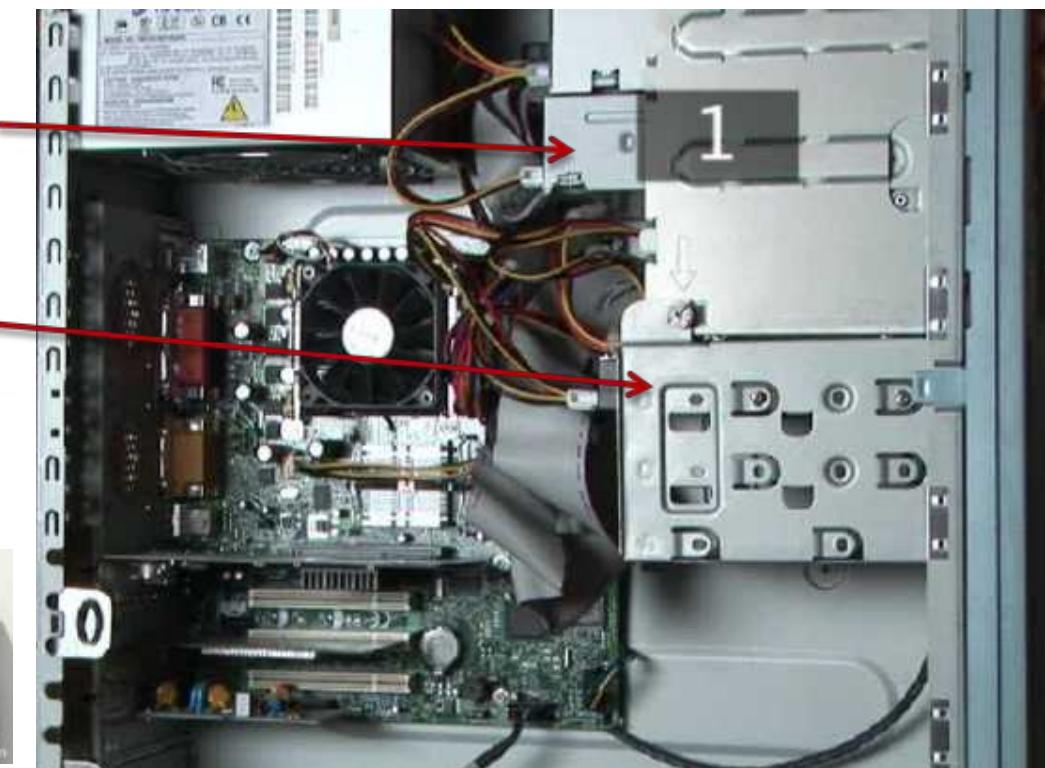
# Arquitectura Von Neumann (1 / 4)



## Ejemplo de módulos + periféricos almacenamiento

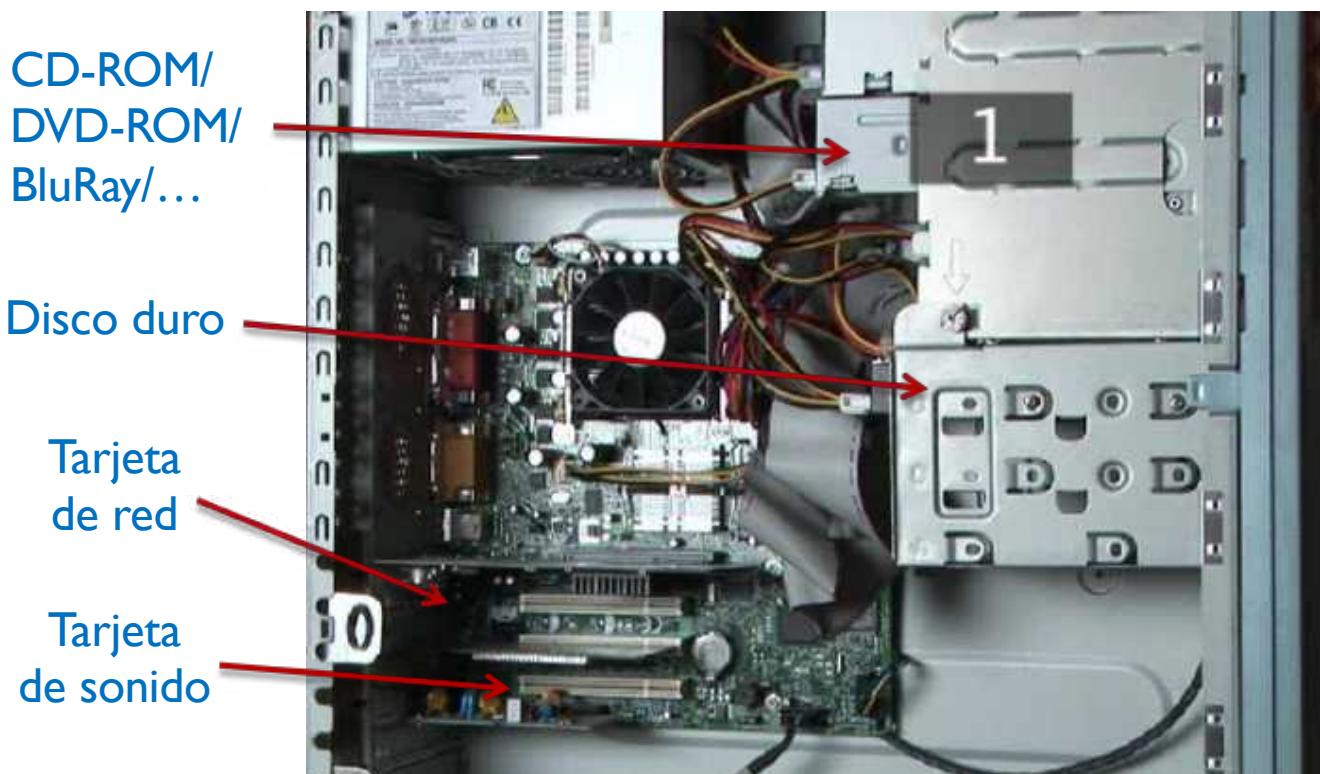
CD-ROM/  
DVD-ROM/  
BluRay/...

Disco duro

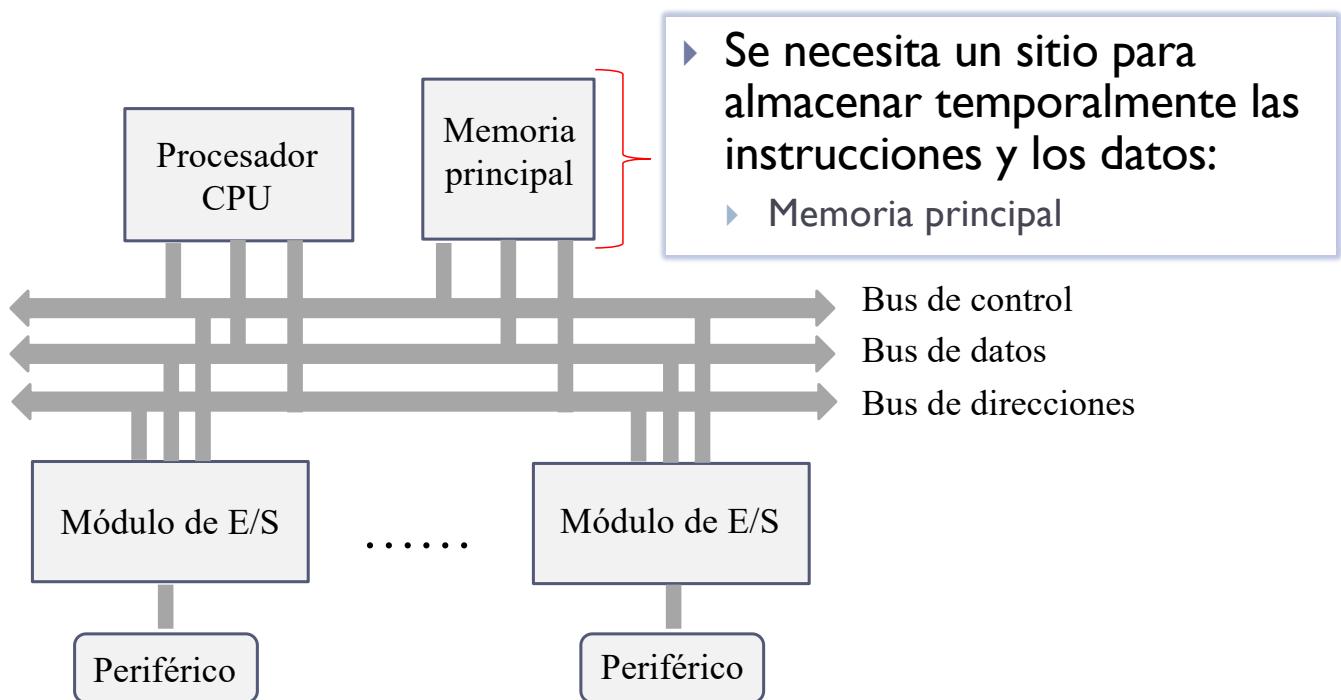


<http://www.videojug.com/film/what-components-are-inside-my-computer>

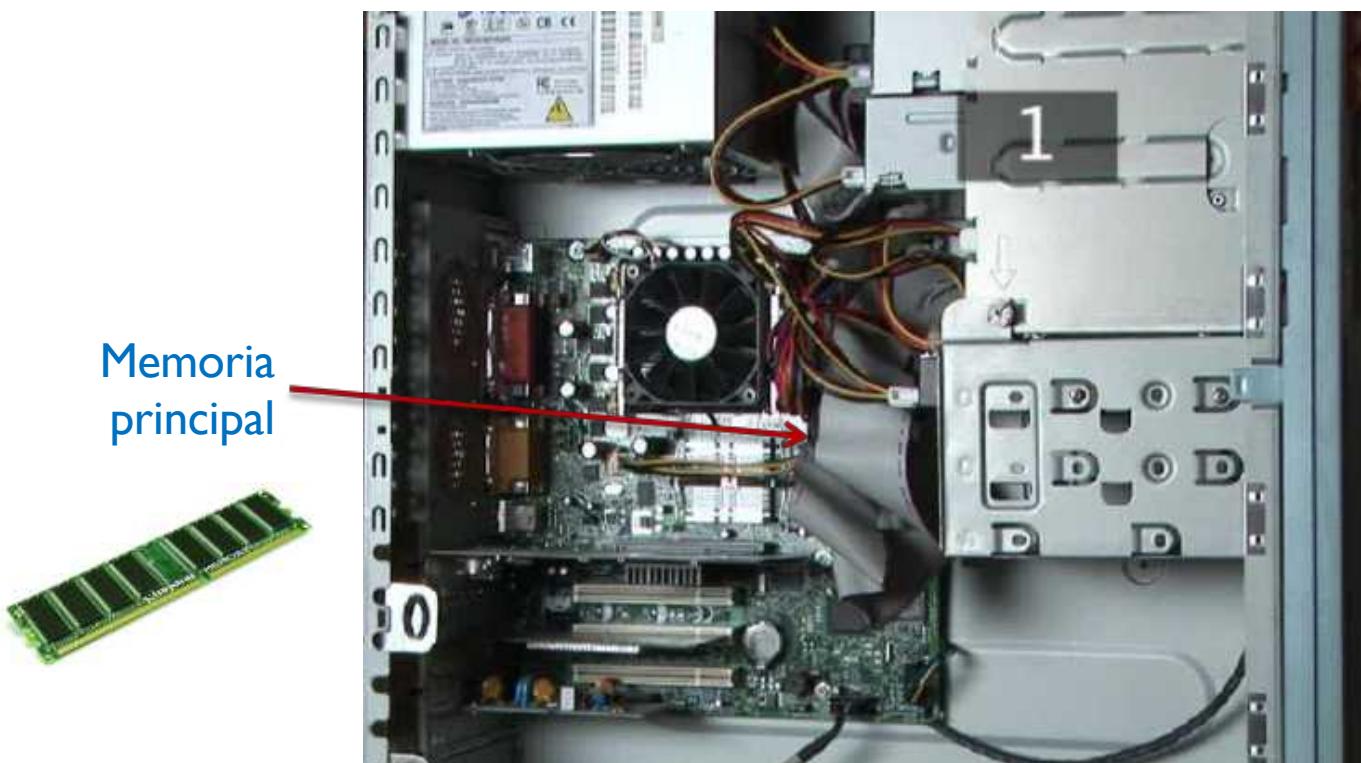
## Ejemplo de módulos + periféricos comunicación



## Arquitectura Von Neumann (2 / 4)

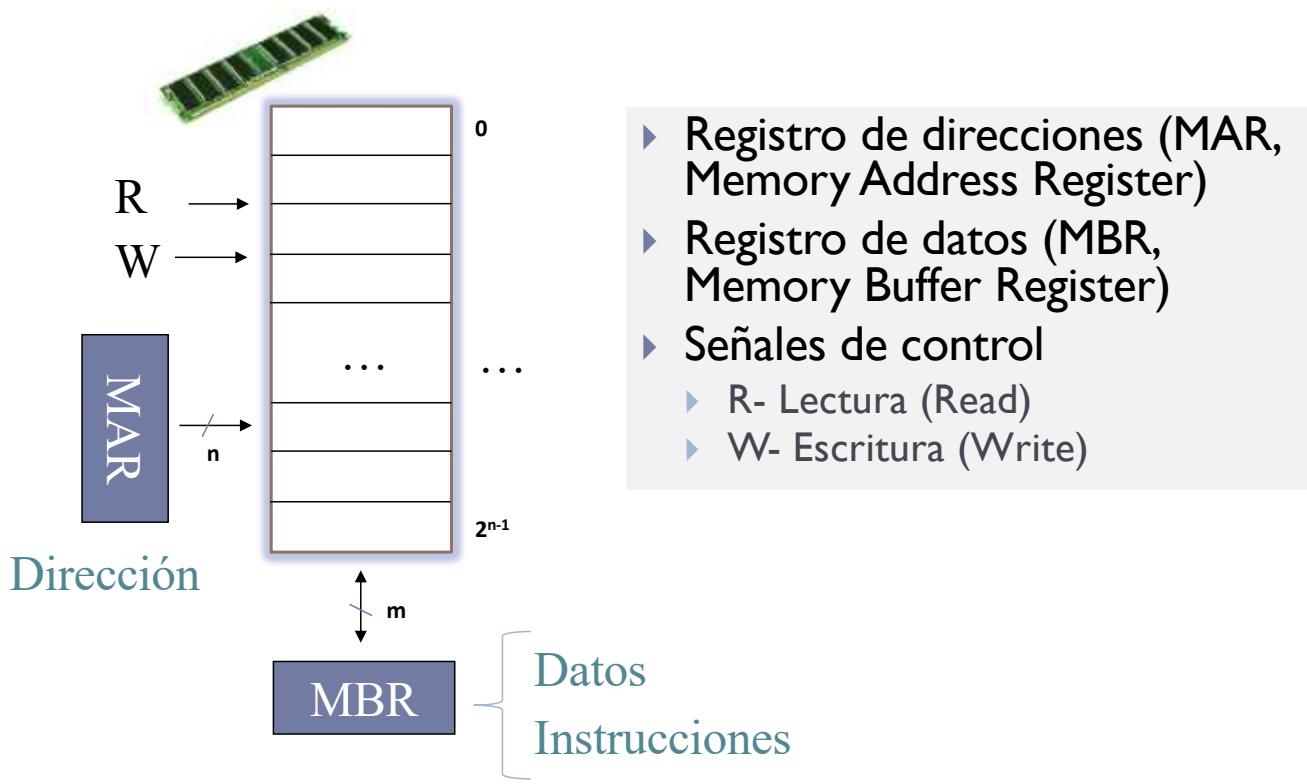


# Ejemplo de memoria principal



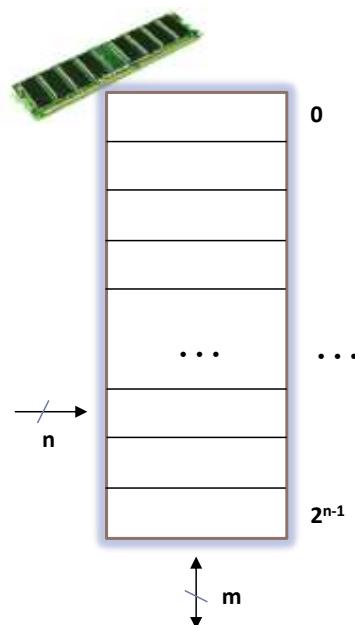
<http://www.videojug.com/film/what-components-are-inside-my-computer>

# Elementos de la memoria principal



# Espacio de direcciones vs. tamaño de palabra

Espacio de direcciones:  
Número de posiciones

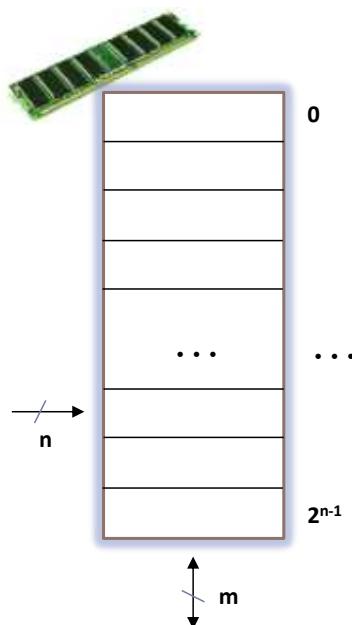


Tamaño de cada posición:  
Número de bits por posición

## Espacio de direcciones vs. tamaño de palabra

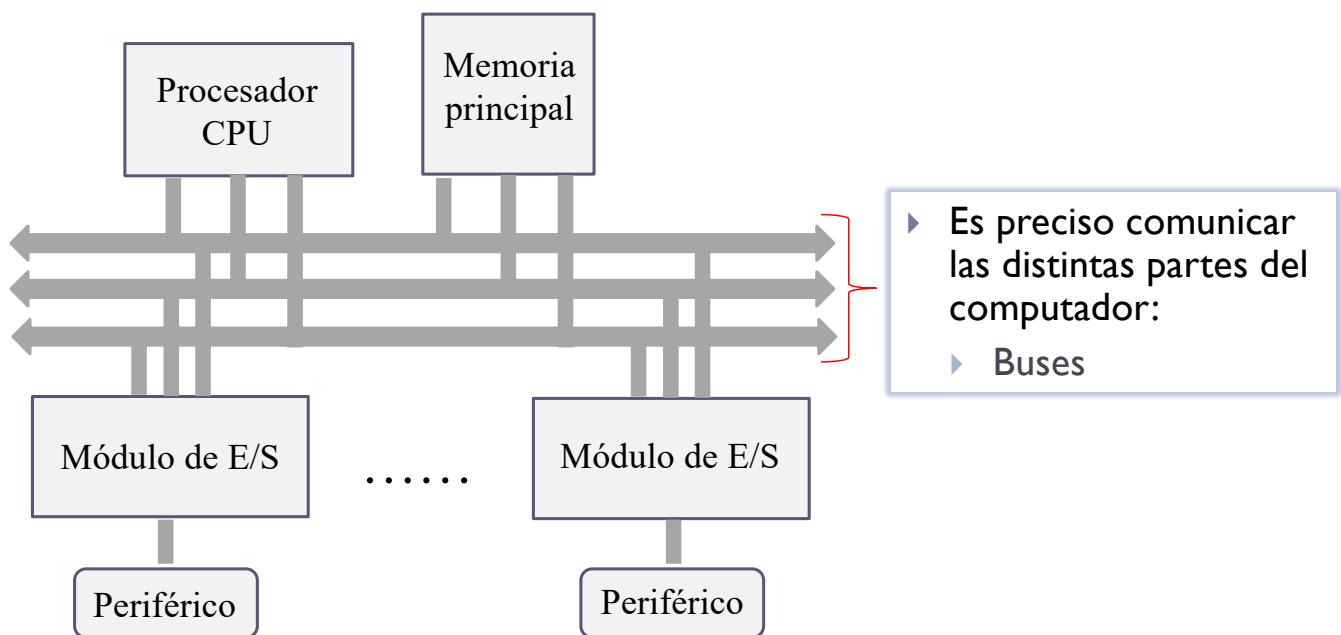
Espacio de direcciones:  
Número de posiciones

$2^n$  posiciones



Tamaño de cada posición:  
Número de bits por posición

## Arquitectura Von Neumann (3 / 4)



# Ejemplo de buses

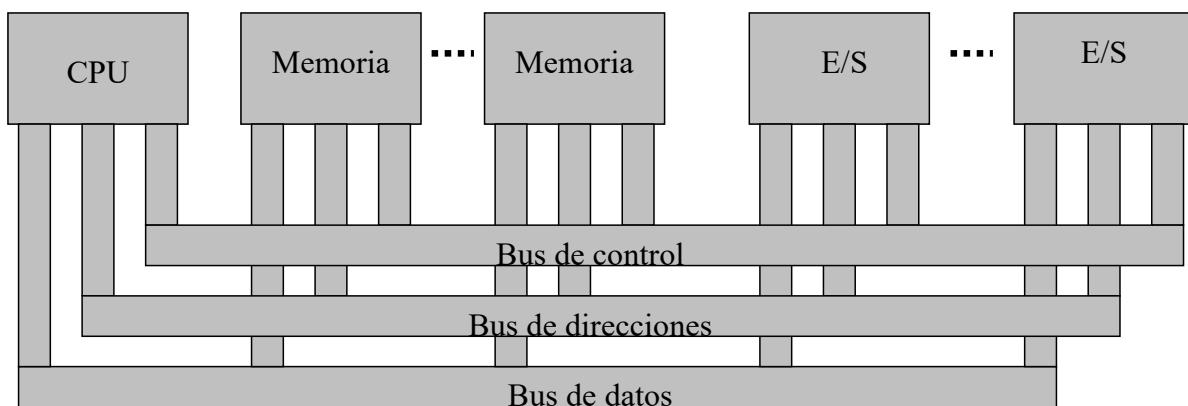


<http://www.videojug.com/film/what-components-are-inside-my-computer>

# Buses

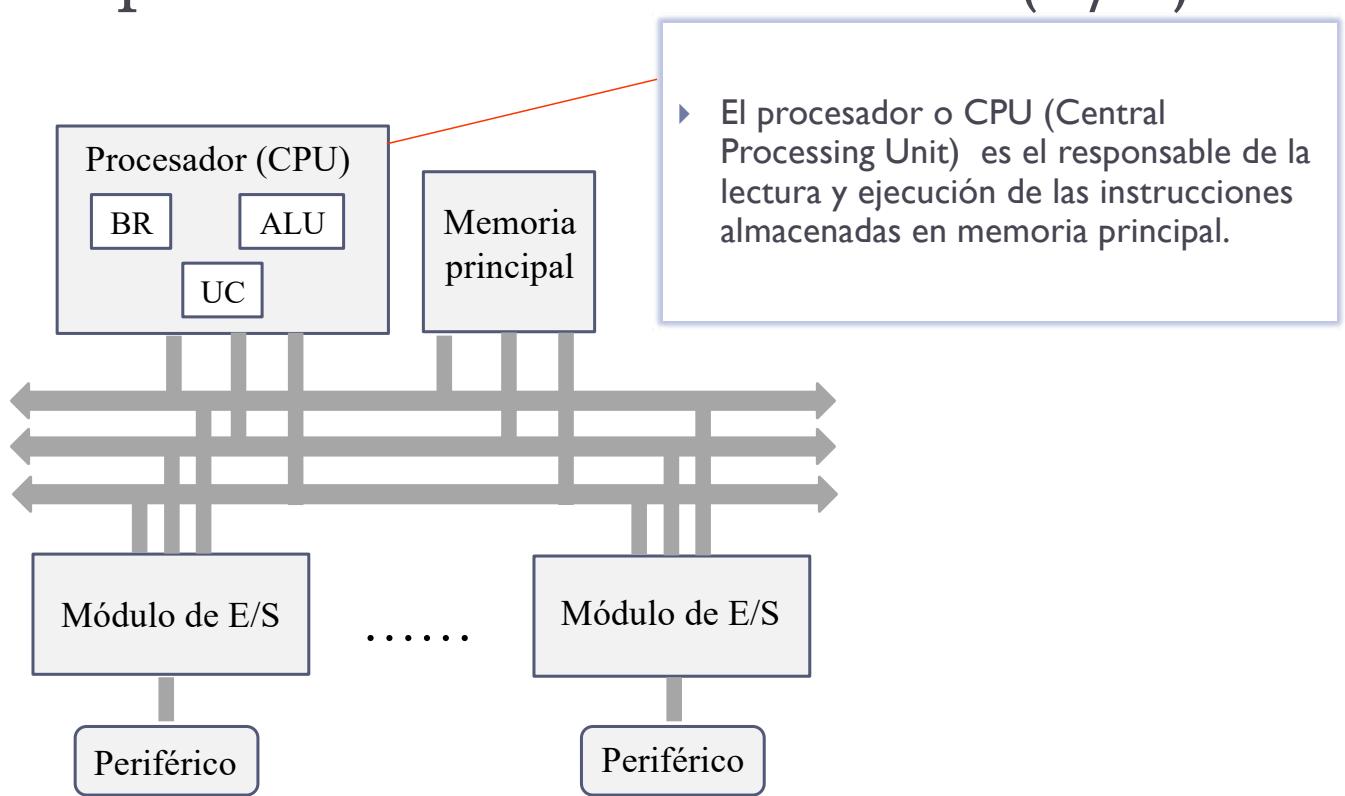
- ▶ Un **bus** es un **camino de comunicación** entre dos o más elementos (procesador, memoria, ...) **para la transmisión de información** entre ellos.
- ▶ Un bus suele formarse por varias líneas de comunicación, cada una transmite un bit.
  - ▶ El ancho del bus representa el tamaño con el que trabaja el computador (ejemplo: bus de 32 bits)
- ▶ Tres tipos principales: **datos, direcciones y control.**

# Esquema de interconexión de bus

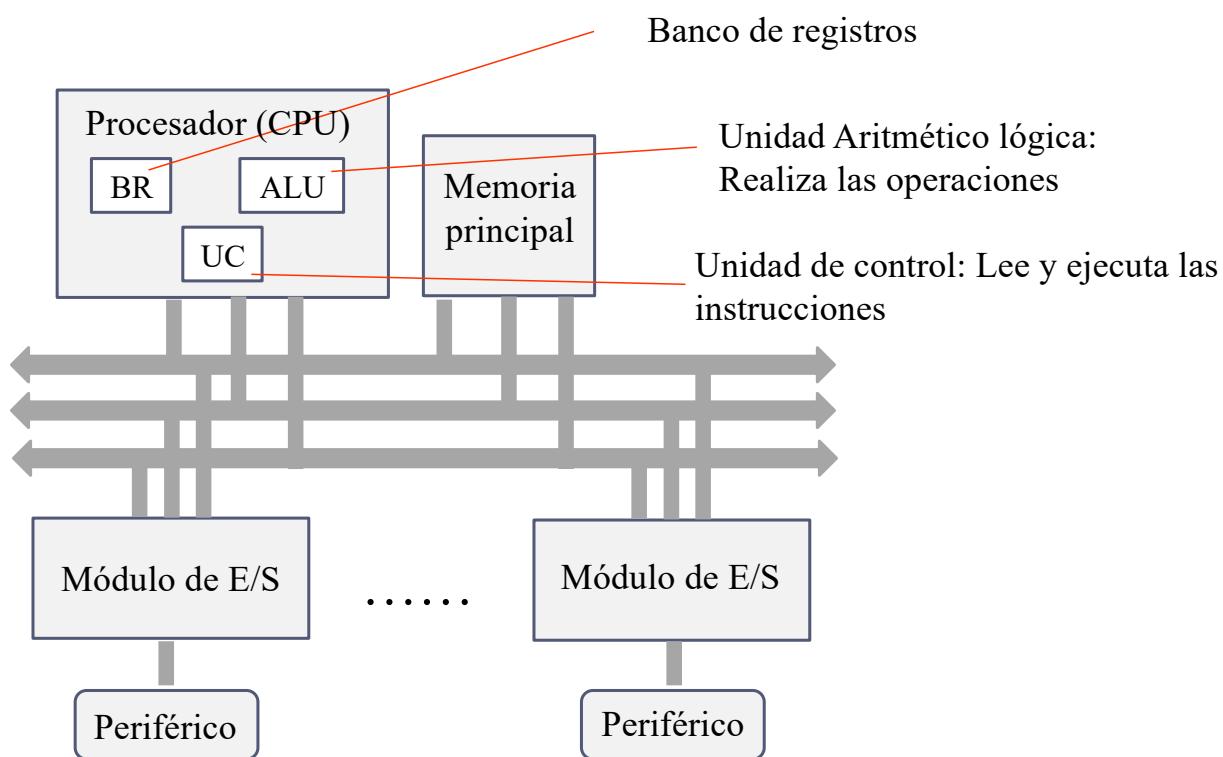


- ▶ **Bus de control:** señales de control y temporización
- ▶ **Bus de direcciones:** designa la fuente o destino de un dato
  - ▶ Su anchura determina la máxima capacidad de memoria del sistema
- ▶ **Bus de datos:** movimiento de datos entre componentes

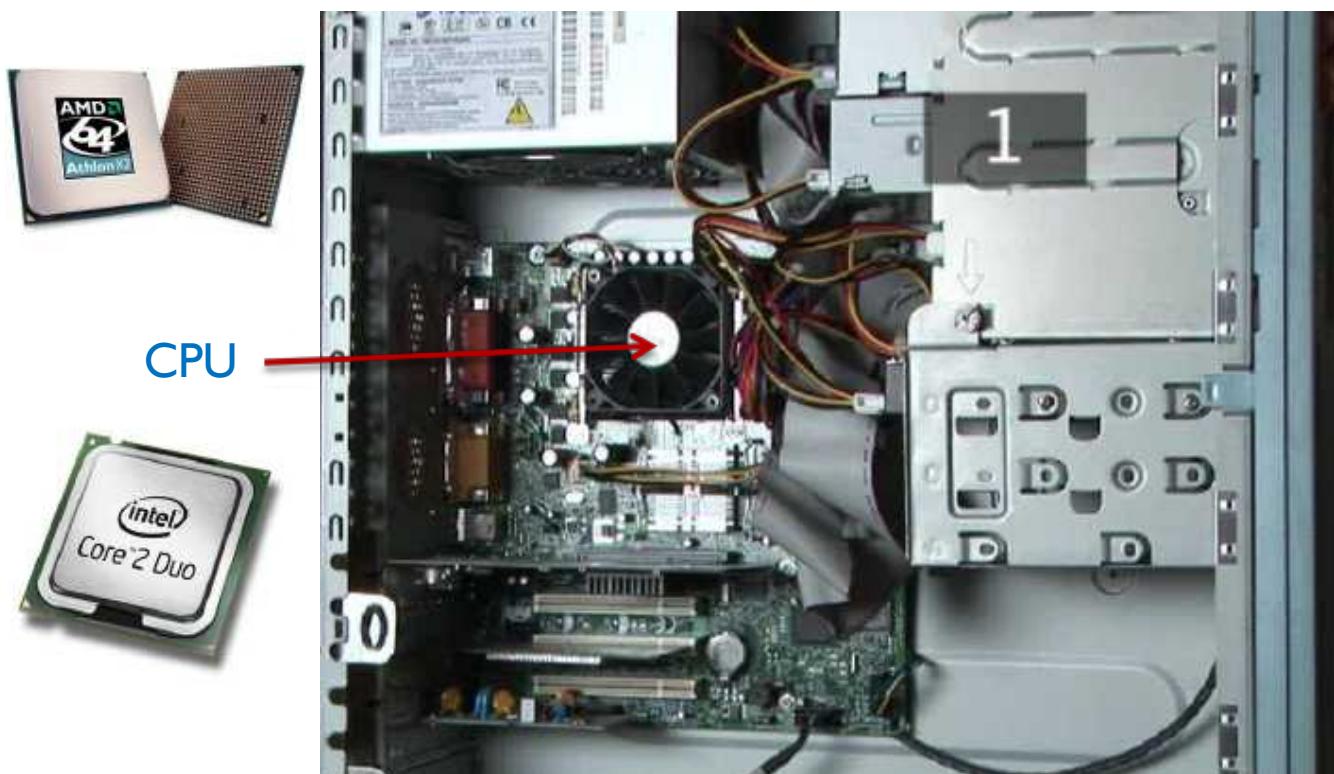
## Arquitectura Von Neumann (4 / 4)



## Arquitectura Von Neumann (4 / 4)

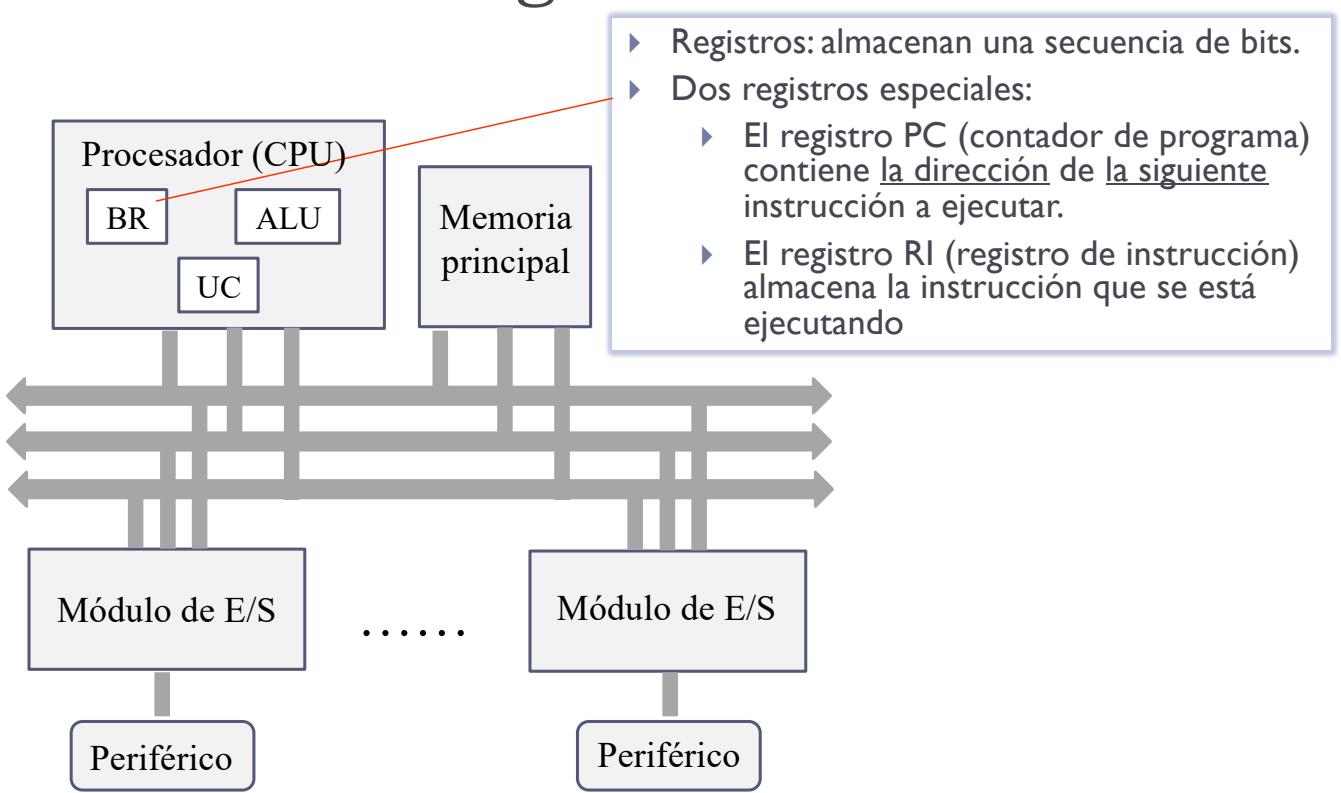


# Ejemplo de CPU

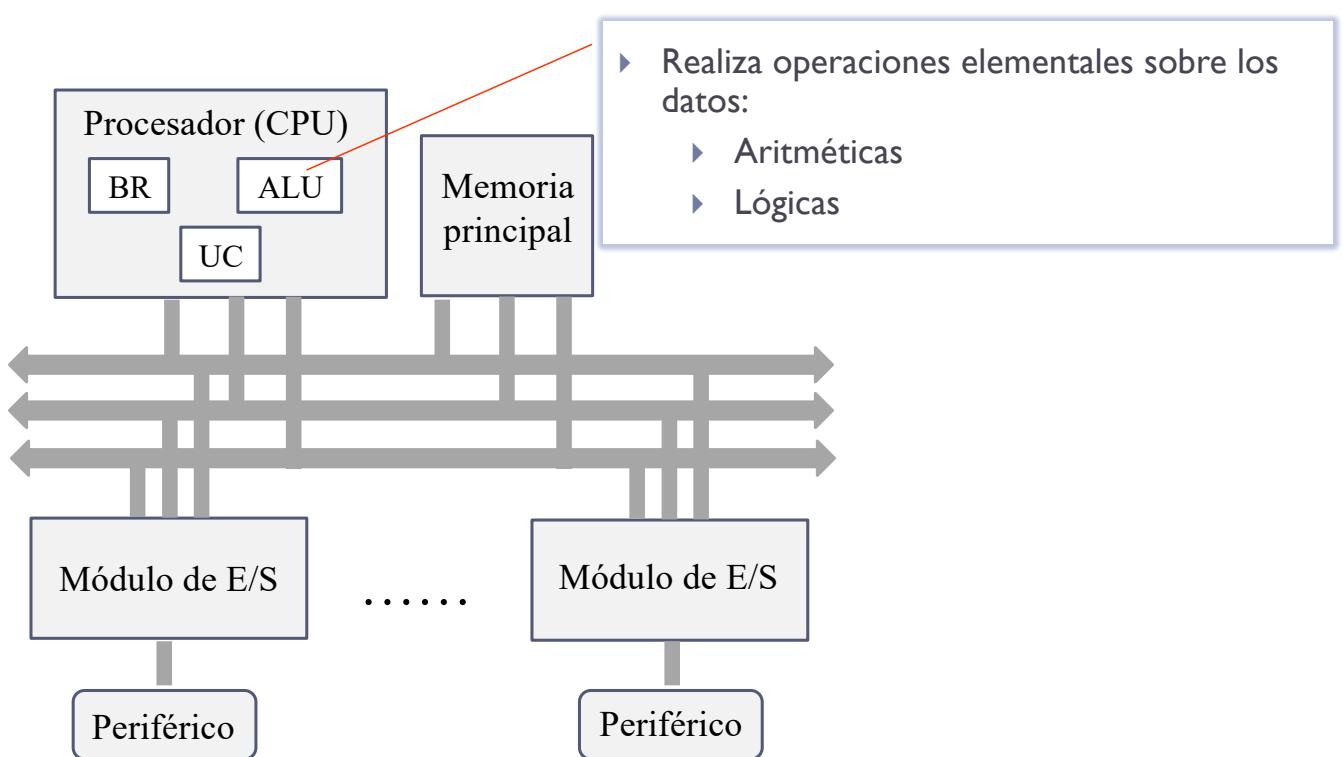


<http://www.videojug.com/film/what-components-are-inside-my-computer>

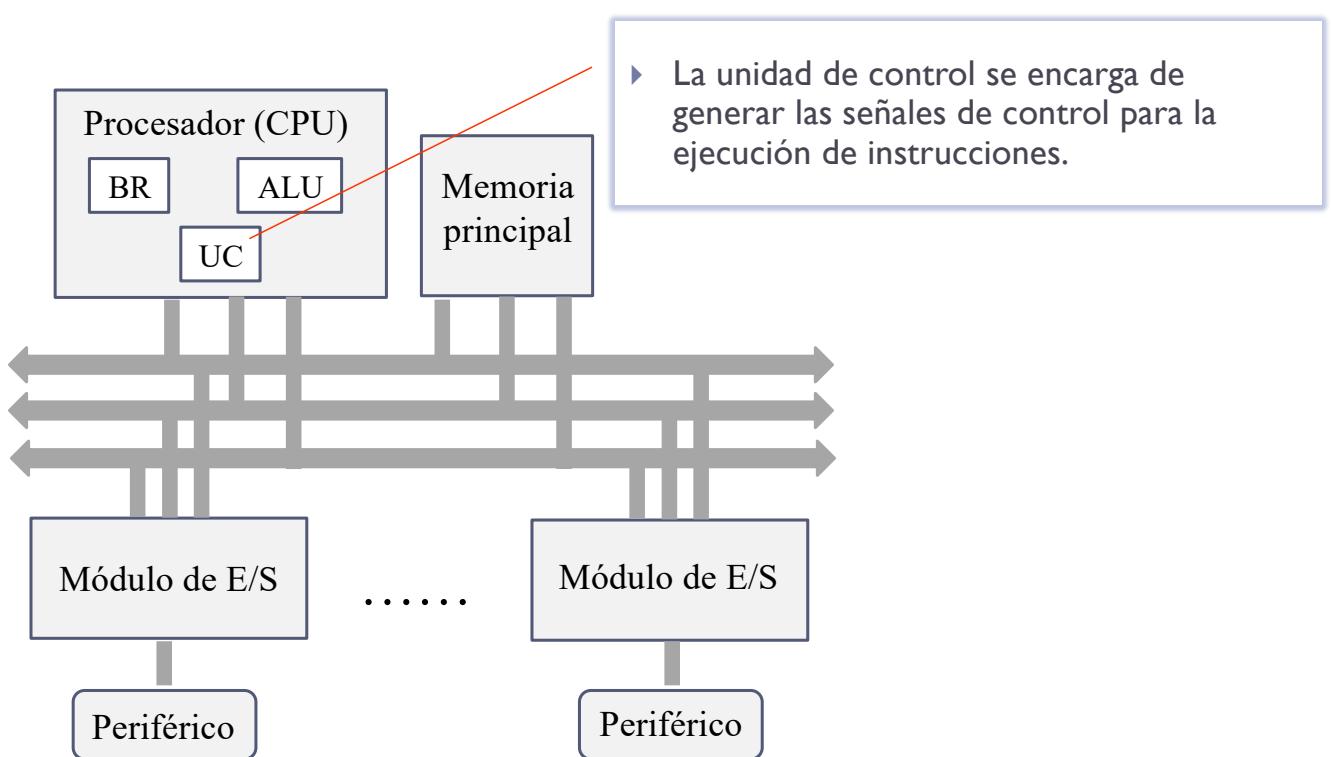
# Procesador: registros



# Procesador: Unidad aritmético lógica ALU



# Procesador: Unidad de control, UC



# Contenidos

1. **¿Qué es un computador?**
2. Concepto de estructura y arquitectura
3. Elementos constructivos de un computador
4. Computador Von Neumann
5. **Instrucciones máquina y programación**
6. Fases de ejecución de una instrucción
7. Parámetros característicos de un computador
8. Tipos de computadores
9. Evolución histórica

# Programa

## ▶ Secuencia consecutiva de instrucciones máquina

```
00001001110001101010111101011000  
1010111101011000000100111000110  
1100011010101111010110000001001  
01011000000010011100011010101111
```

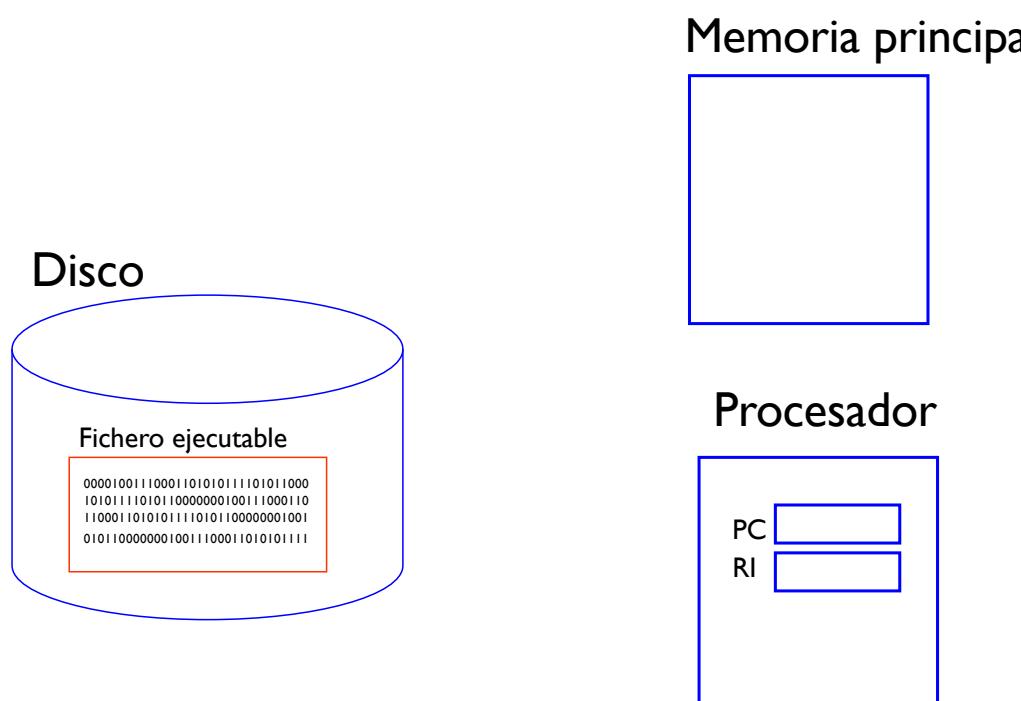


# Programa

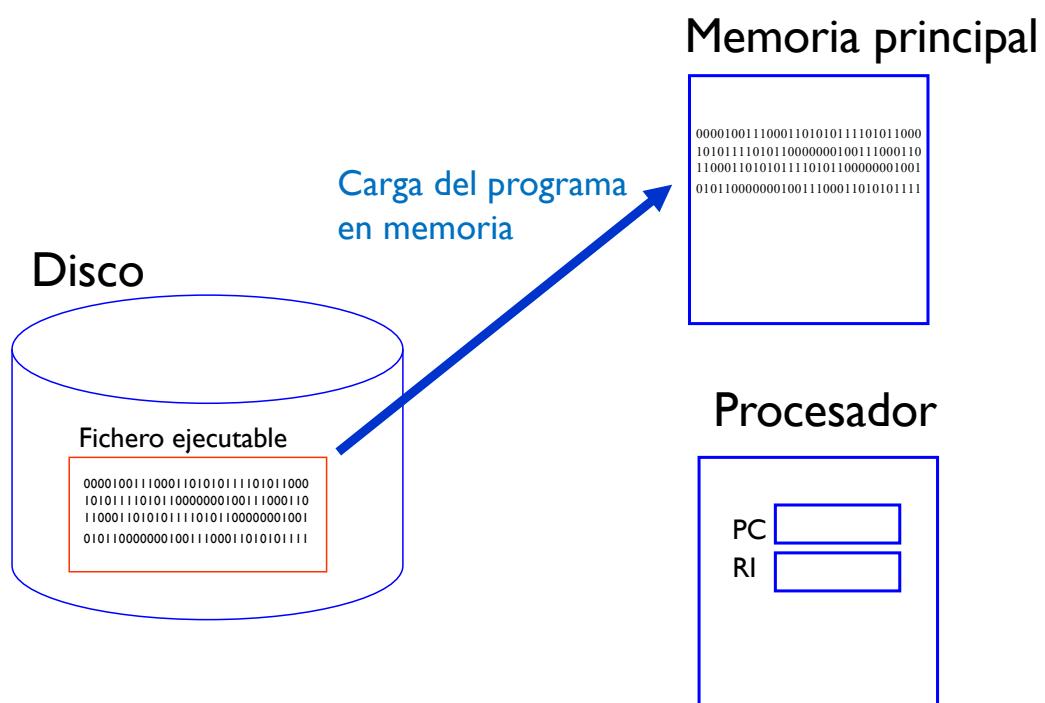
- ▶ Secuencia consecutiva de instrucciones máquina
- ▶ **Instrucción máquina:** operación elemental que puede ejecutar directamente un procesador
  - ▶ Codificación en binario

```
00001001110001101010111101011000      temp = v[k];  
10101111010110000000100111000110      v[k] = v[k+1];  
1100011010101111010110000001001      v[k+1] = temp;  
01011000000010011100011010101111  
.  
..
```

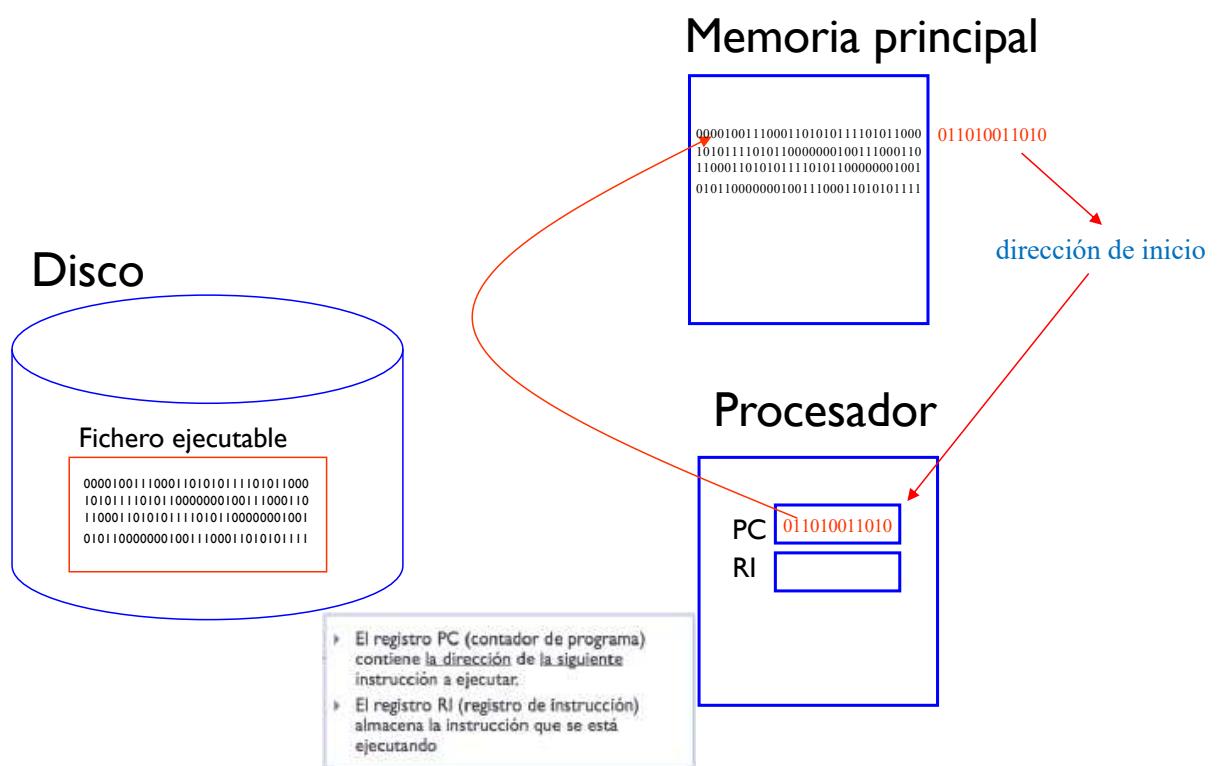
# Ejecución de un programa



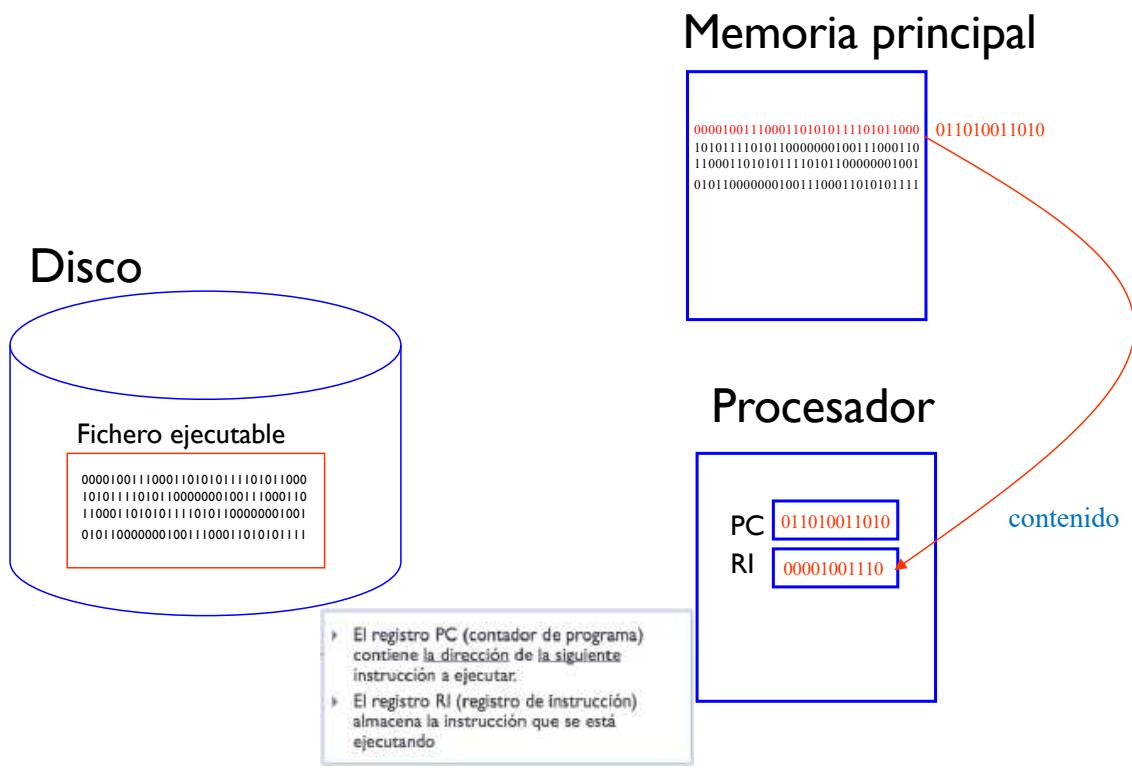
# Ejecución de un programa



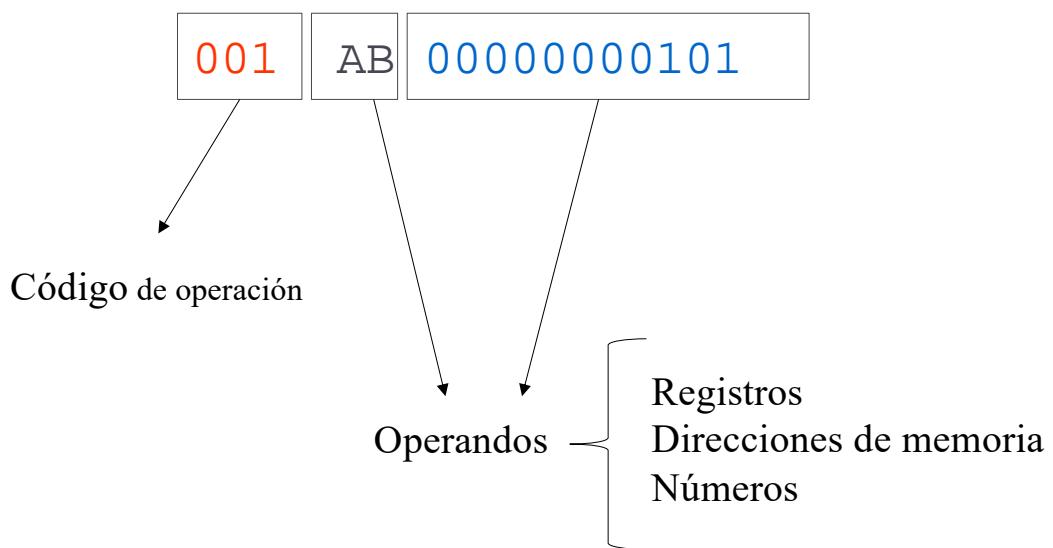
# Ejecución de un programa



# Ejecución de un programa

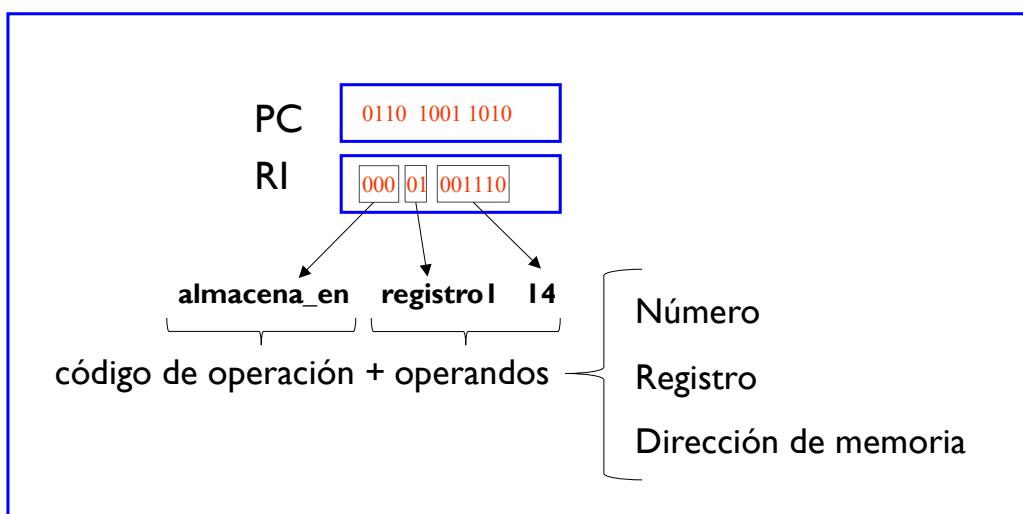


# Formato de una instrucción máquina



# Formato de instrucción

## Procesador



# Ejemplo de juego de instrucciones

- ▶ Conjunto de instrucciones con las siguientes características:
  - ▶ Tamaño de una posición de memoria: 16 bits
  - ▶ Tamaño de la instrucción: 16 bits
  - ▶ Código de operación: 3 bits
    - ▶ ¿Cuántas instrucciones diferentes puede tener este computador?
    - ▶ Número de registros de propósito general: 4
    - ▶ Identificadores simbólicos:
      - R0
      - R1
      - R2
      - R3
    - ▶ ¿Cuántos bits se necesitan para representar estos 4 registros?

## Ejemplo de juego de instrucciones

- ▶ Conjunto de instrucciones con las siguientes características:
  - ▶ Tamaño de una posición de memoria: 16 bits
  - ▶ Tamaño de la instrucción: 16 bits
  - ▶ Código de operación: 3 bits
    - ▶ ¿Cuántas instrucciones diferentes puede tener este computador? **8**
    - ▶ Número de registros de propósito general: 4 (**2 bits**)
    - ▶ Identificadores simbólicos:
      - R0 (**00**)
      - R1 (**01**)
      - R2 (**10**)
      - R3 (**11**)
    - ▶ ¿Cuántos bits se necesitan para representar estos 4 registros? **2**

# Ejemplo de juego de instrucciones

Instrucción	Descripción
000CC <sub>A</sub> BBXXXXXXXX	Suma el registro AA con el BB y deja el resultado en CC
001AA00000000101	Almacena en el registro AA el valor 00000000101
010AA00000001001	Almacena en el registro AA el valor almacenado en la posición de memoria 00000001001
011AA00000001001	Almacena en la posición de memoria 00000001001 el contenido del registro AA
100000000001001	Se salta a ejecutar la instrucción almacenada en la posición de memoria 0000000001001
101AA <sub>B</sub> B000001001	Si el contenido del registro AA es igual al del registro BB se salta a ejecutar la instrucción almacenada en 000001001

Siendo A,B,C,D,E,F = 0 ó 1

# Ejemplos

- ▶ Instrucción que almacena un 5 en el registro 00
- ▶ Instrucción que almacena un 7 en el registro 01
- ▶ Instrucción que suma el contenido del registro 00 y el registro 01 y deja el resultado en el registro 10
- ▶ Instrucción que almacena el resultado anterior en la posición de memoria 1027 (en decimal)

# Ejemplos

Instrucción	Descripción
000CCAABXXXXXX	Suma el registro AA con el BB y deja el resultado en CC
001AA00000000101	Almacena en el registro AA el valor 00000000101
010AA00000001001	Almacena en el registro AA el valor almacenado en la posición de memoria 00000001001
011AA000000001001	Almacena en la posición de memoria 00000001001 el contenido del registro AA
100000000001001	Se salta a ejecutar la instrucción almacenada en la posición de memoria 000000001001
101AA BB000001001	Si el contenido del registro AA es igual al del registro BB se salta a ejecutar la instrucción almacenada en 000001001

Siendo A,B,C,D,E,F = 0 ó 1

- ▶ Instrucción que almacena un 5 en el registro 00
- ▶ Instrucción que almacena un 7 en el registro 01
- ▶ Instrucción que suma el contenido del registro 00 y el registro 01 y deja el resultado en el registro 10
- ▶ Instrucción que almacena el resultado anterior en la posición de memoria 1027 (en decimal)

# Ejemplos

Instrucción	Descripción
000CCAABXXXXXX	Suma el registro AA con el BB y deja el resultado en CC
001AA00000000101	Almacena en el registro AA el valor 00000000101
010AA00000001001	Almacena en el registro AA el valor almacenado en la posición de memoria 00000001001
011AA000000001001	Almacena en la posición de memoria 00000001001 el contenido del registro AA
100000000001001	Se salta a ejecutar la instrucción almacenada en la posición de memoria 000000001001
101AABB000001001	Si el contenido del registro AA es igual al del registro BB se salta a ejecutar la instrucción almacenada en 000001001

Siendo A,B,C,D,E,F = 0 ó 1

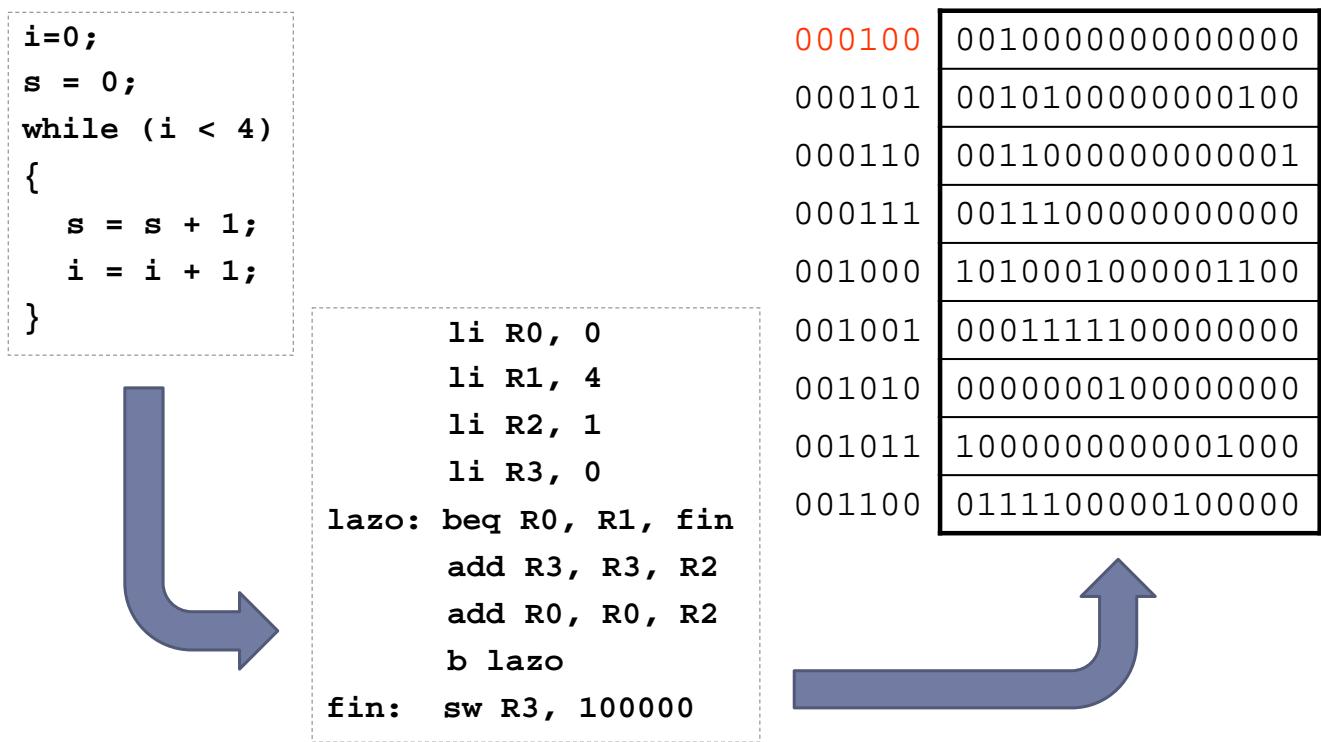
- ▶ Instrucción que almacena un 5 en el registro 00  
**001000000000101**
- ▶ Instrucción que almacena un 7 en el registro 01  
**001010000000111**
- ▶ Instrucción que suma el contenido del registro 00 y el registro 01 y deja el resultado en el registro 10  
**000100001XXXXXXX**
- ▶ Instrucción que almacena el resultado anterior en la posición de memoria 1027 (en decimal)  
**0111010000000011**

# Ejemplo de programa cargado en memoria

Memoria principal

Dirección	Contenido
000100	0010000000000000
000101	001010000000100
000110	001100000000001
000111	001110000000000
001000	1010001000001100
001001	000111100000000
001010	0000000100000000
001011	100000000001000
001100	0111100000100000

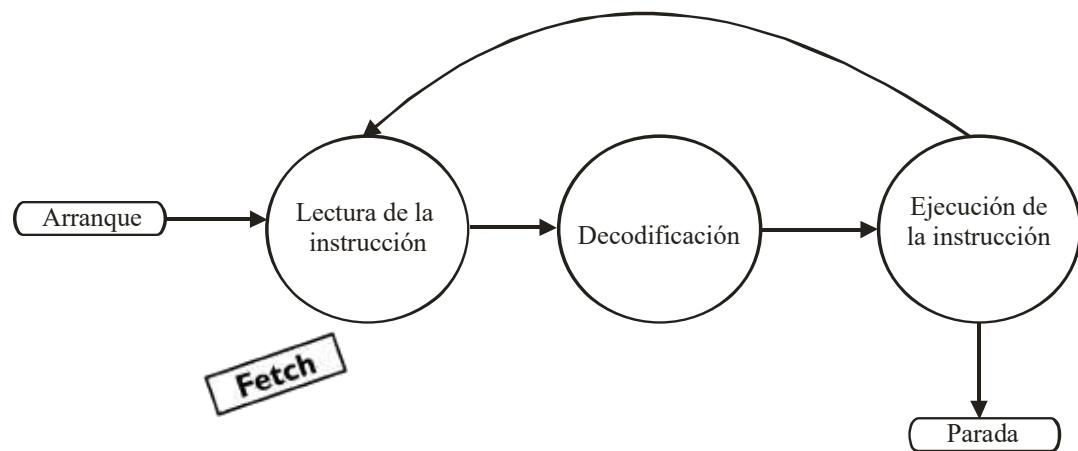
# Generación y carga de un programa



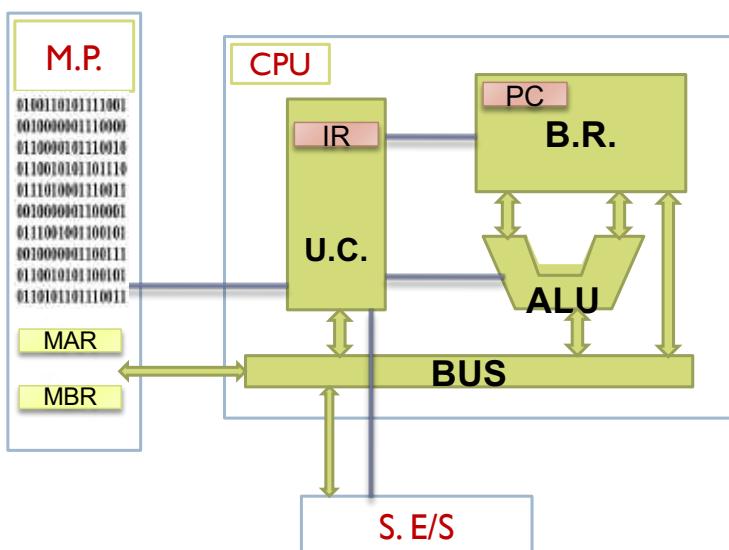
# Contenidos

1. **¿Qué es un computador?**
2. **Concepto de estructura y arquitectura**
3. **Elementos constructivos de un computador**
4. **Computador Von Neumann**
5. **Instrucciones máquina y programación**
6. **Fases de ejecución de una instrucción**
7. **Parámetros característicos de un computador**
8. **Tipos de computadores**
9. **Evolución histórica**

## Fases de ejecución de una instrucción

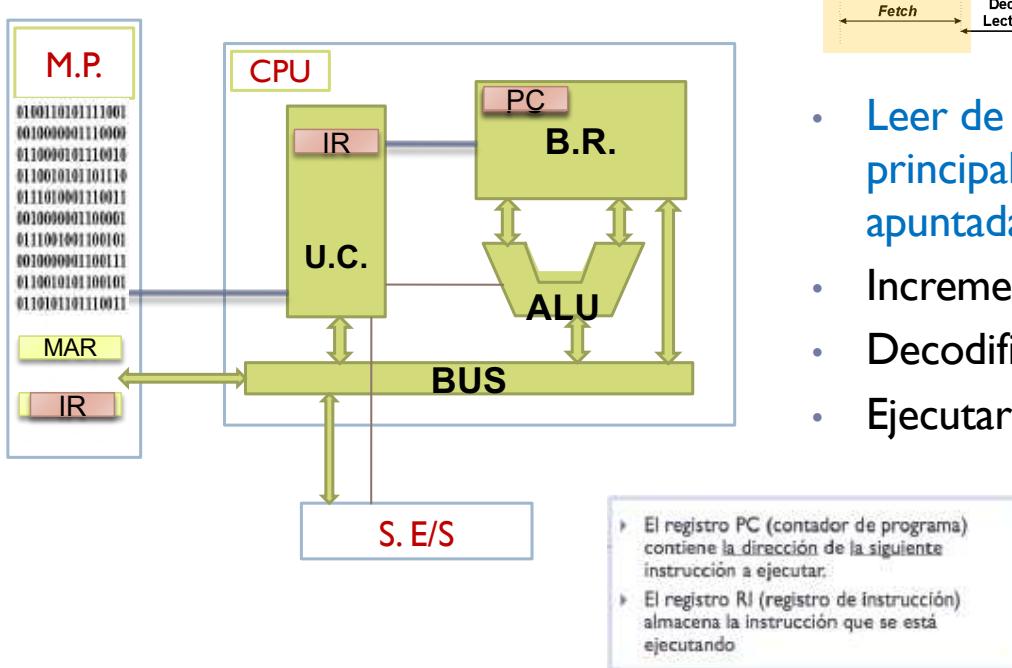


# Fases de ejecución (1)



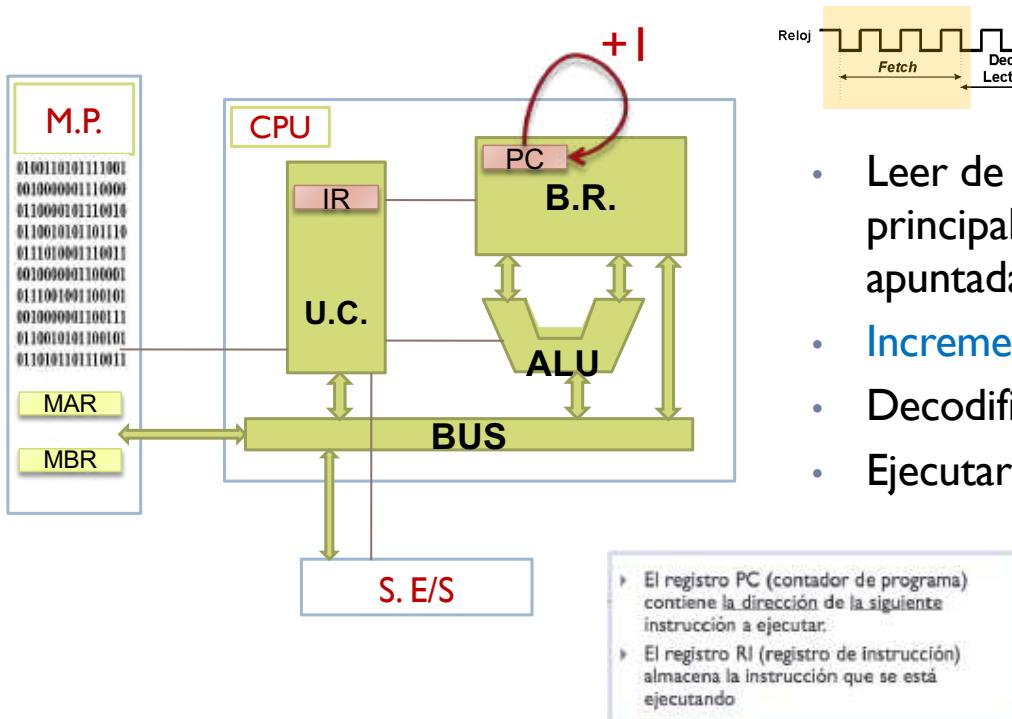
- Leer de memoria principal la instrucción apuntada por el PC
- Incrementar PC
- Decodificar instrucción
- Ejecutar la instrucción

## Fases de ejecución (2)



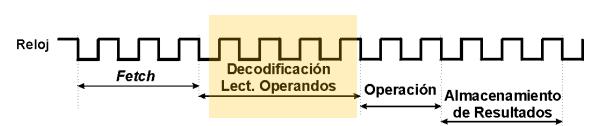
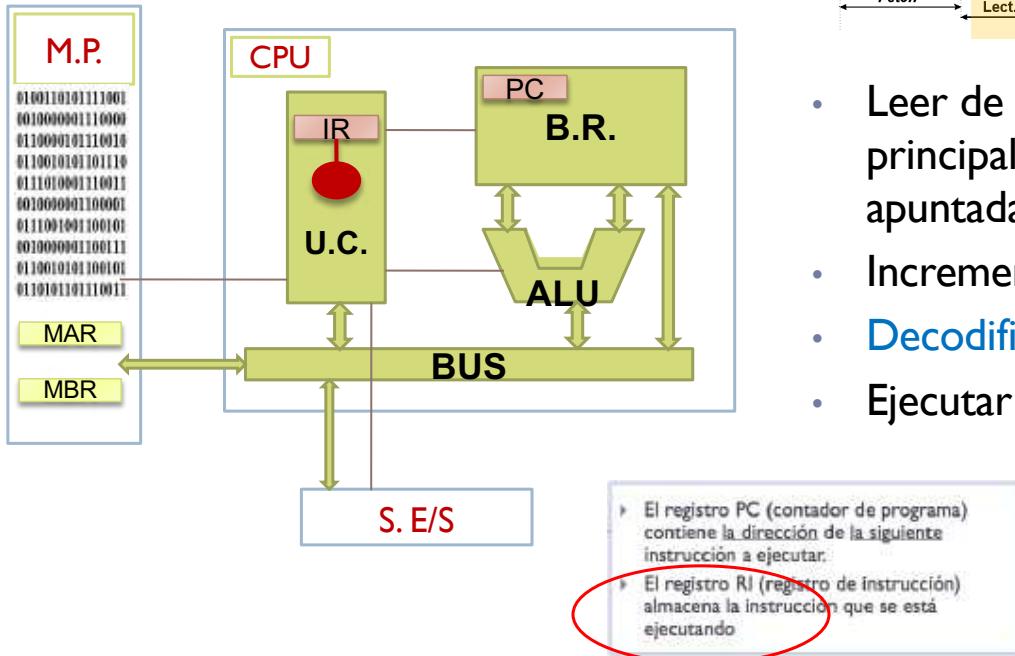
- Leer de memoria principal la instrucción apuntada por el PC
- Incrementar PC
- Decodificar instrucción
- Ejecutar la instrucción

## Fases de ejecución (3)



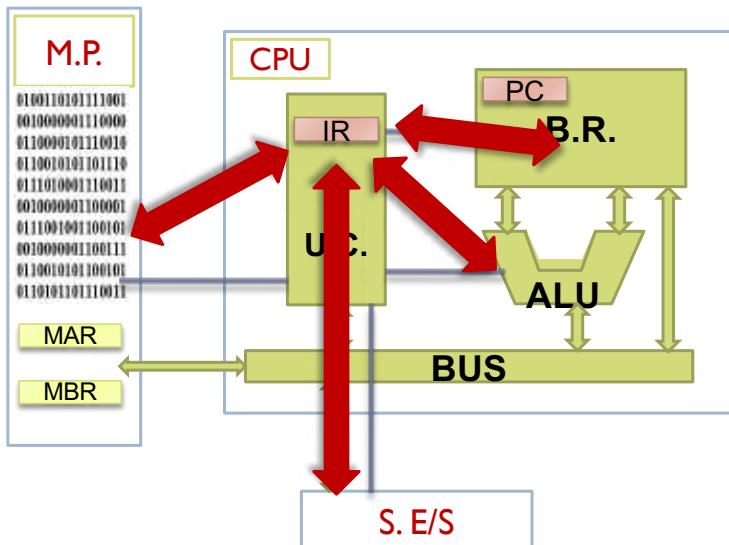
- Leer de memoria principal la instrucción apuntada por el PC
- Incrementar PC
- Decodificar instrucción
- Ejecutar la instrucción

## Fases de ejecución (4)



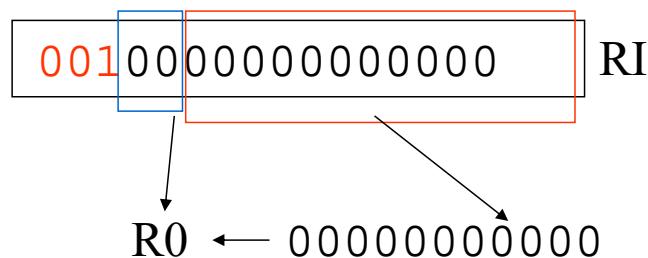
- Leer de memoria principal la instrucción apuntada por el PC
- Incrementar PC
- Decodificar instrucción
- Ejecutar la instrucción

## Fases de ejecución (5)



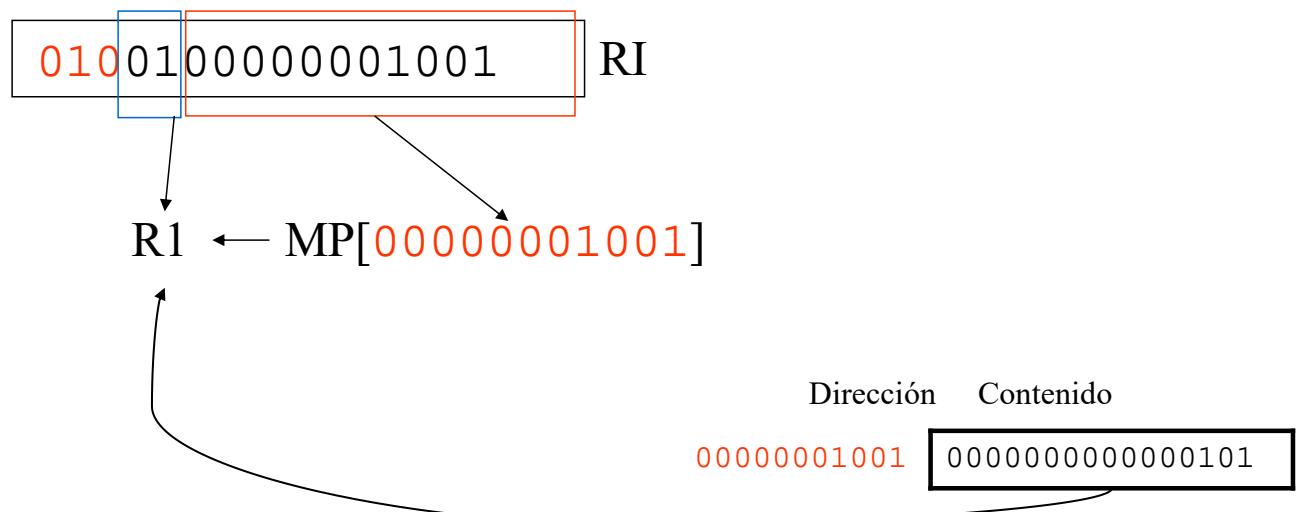
- Leer de memoria principal la instrucción apuntada por el PC
- Incrementar PC
- Decodificar instrucción
- Ejecutar la instrucción

# Ejemplo ejecución de instrucciones



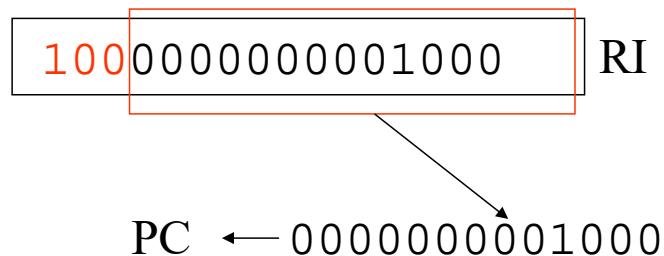
Se carga en R0 el valor 0

# Ejemplo ejecución de instrucciones



Se carga en R1 el contenido de la posición de memoria  
00000001001

# Ejemplo ejecución de instrucciones



Se modifica el PC con la dirección 000000001000  
de forma que la siguiente instrucción a ejecutar es la que se  
encuentra en 000000001000

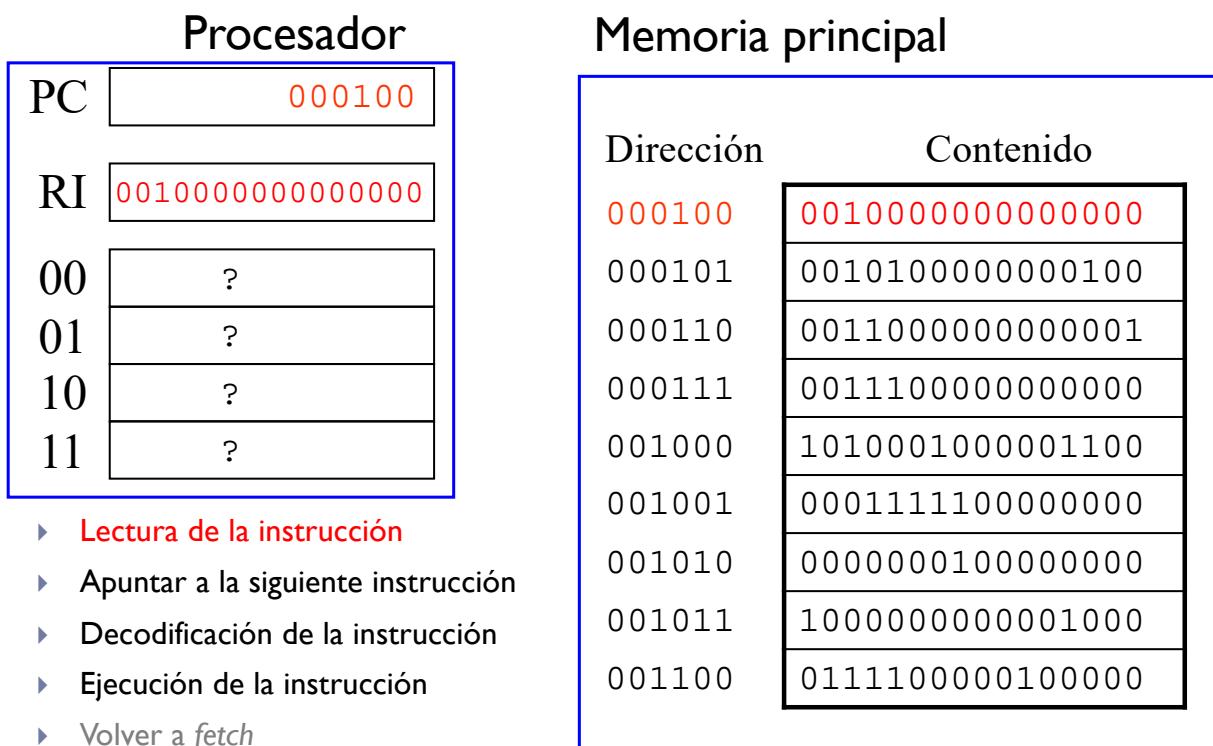
# Ejemplo de ejecución de un programa

Procesador	
PC	000100
RI	?
00	?
01	?
10	?
11	?

- ▶ Lectura de la instrucción
- ▶ Apuntar a la siguiente instrucción
- ▶ Decodificación de la instrucción
- ▶ Ejecución de la instrucción
- ▶ Volver a *fetch*

Dirección	Contenido
000100	0010000000000000
000101	001010000000100
000110	0011000000000001
000111	0011100000000000
001000	1010001000001100
001001	0001111000000000
001010	0000000100000000
001011	100000000001000
001100	0111100000100000

# Ejemplo de ejecución de un programa



# Ejemplo de ejecución de un programa

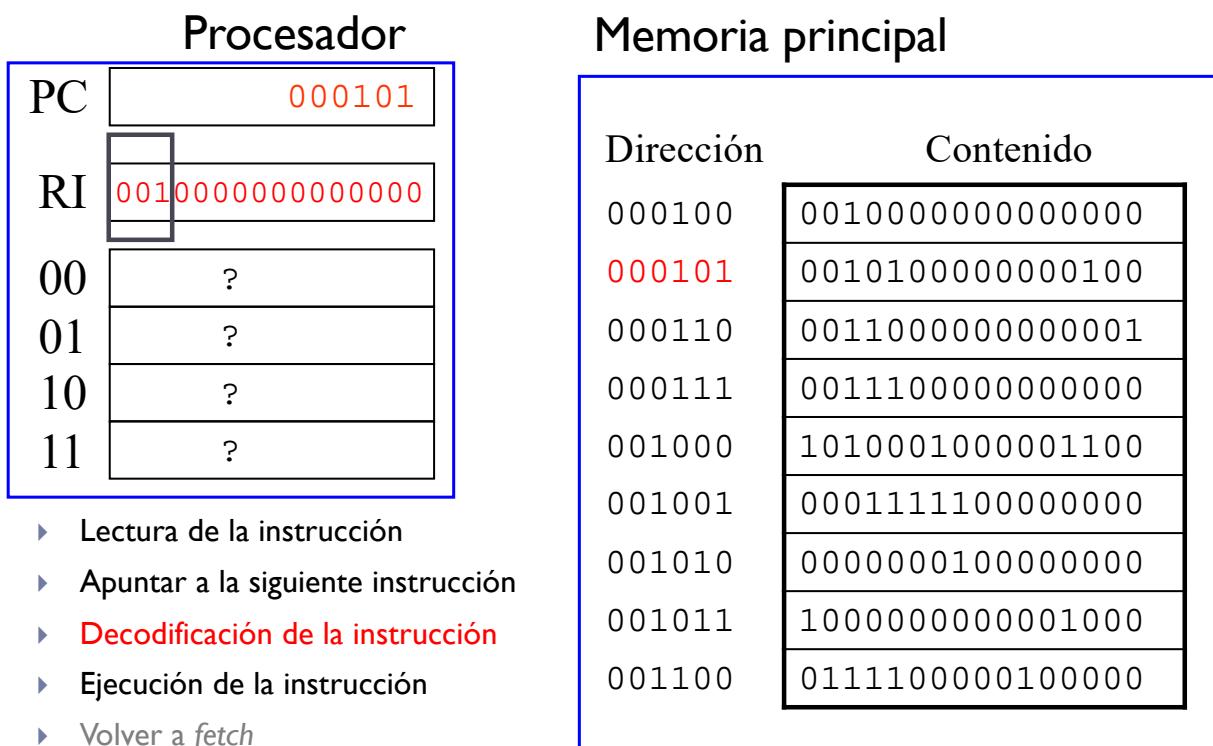
Procesador	
PC	000101
RI	0010000000000000
00	?
01	?
10	?
11	?

- ▶ Lectura de la instrucción
- ▶ Apuntar a la siguiente instrucción
  - ▶  $PC \leftarrow PC + 1$
- ▶ Decodificación de la instrucción
- ▶ Ejecución de la instrucción
- ▶ Volver a fetch

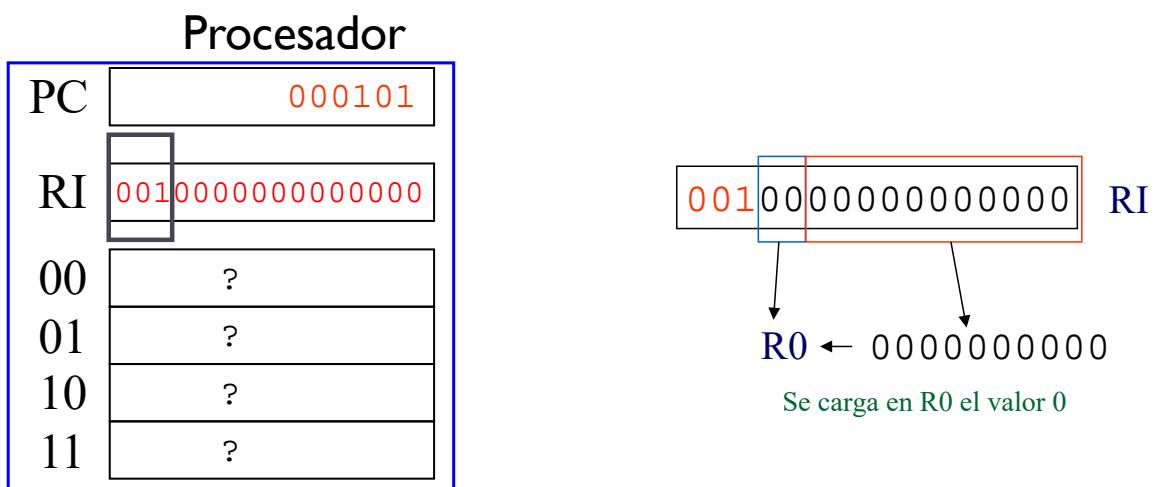
86

Memoria principal	
Dirección	Contenido
000100	0010000000000000
000101	0010100000001000
000110	0011000000000001
000111	0011100000000000
001000	1010001000001100
001001	0001111000000000
001010	0000000100000000
001011	1000000000010000
001100	0111100000100000

# Ejemplo de ejecución de un programa

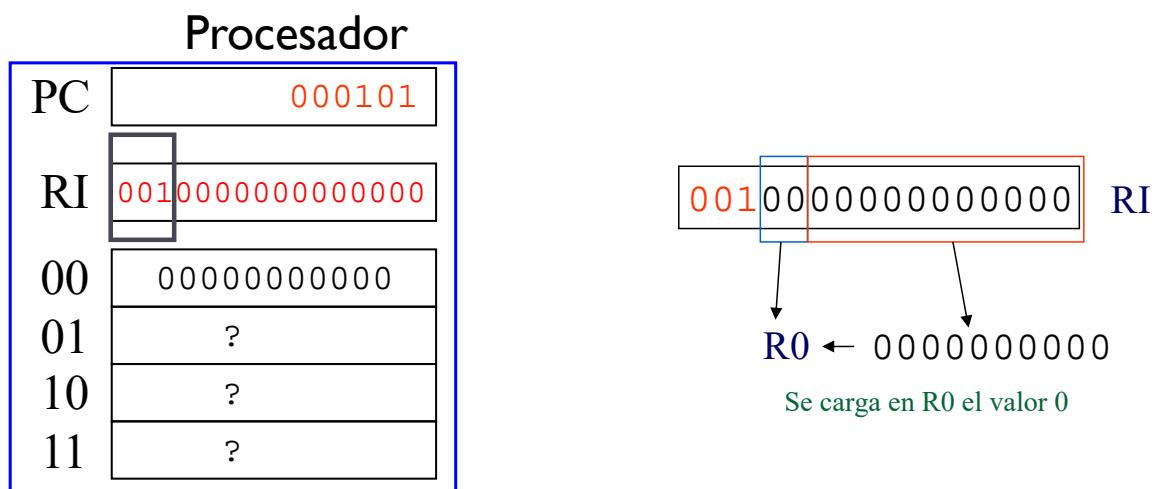


# Ejemplo de ejecución de un programa



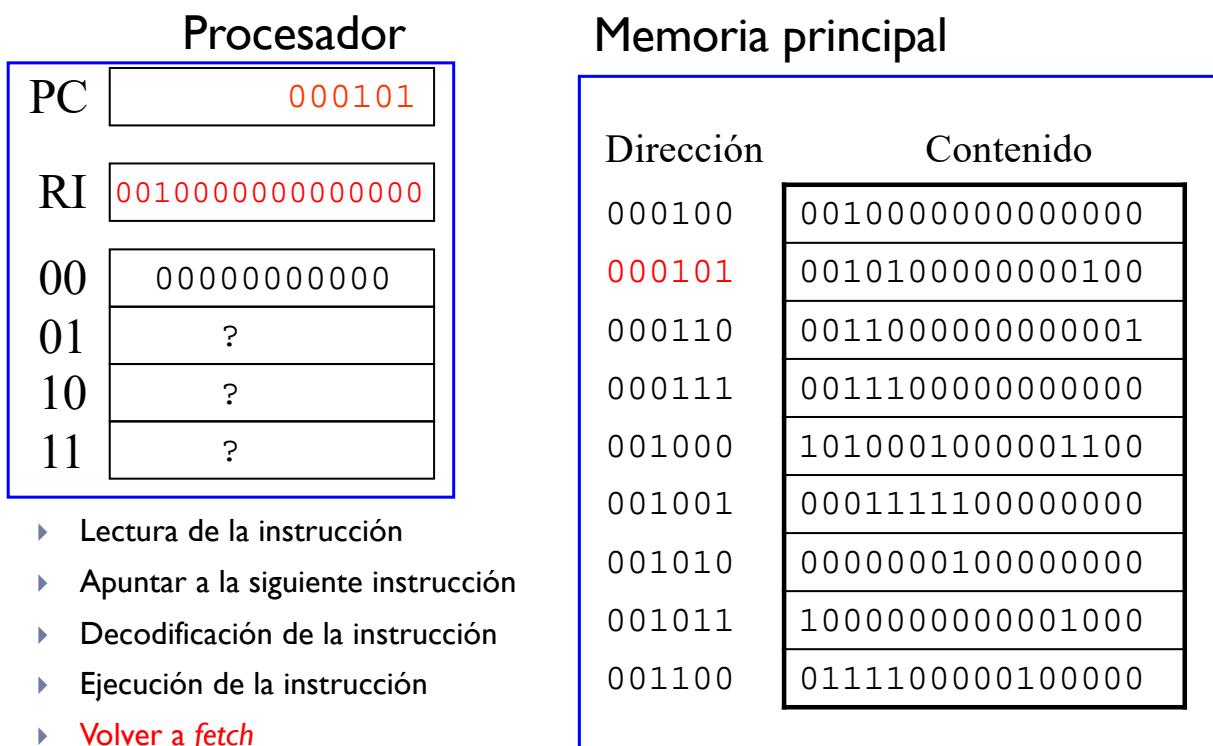
- ▶ Lectura de la instrucción
- ▶ Apuntar a la siguiente instrucción
- ▶ Decodificación de la instrucción
- ▶ Ejecución de la instrucción
- ▶ Volver a *fetch*

# Ejemplo de ejecución de un programa

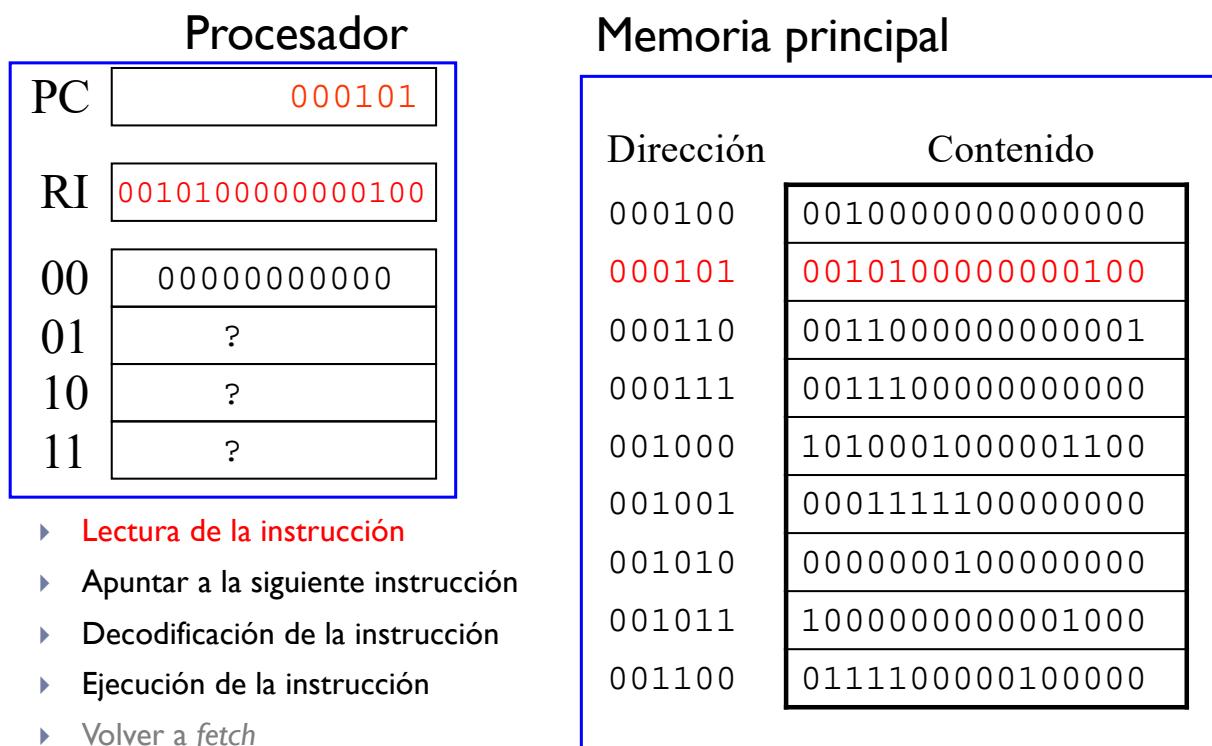


- ▶ Lectura de la instrucción
- ▶ Apuntar a la siguiente instrucción
- ▶ Decodificación de la instrucción
- ▶ Ejecución de la instrucción
- ▶ Volver a fetch

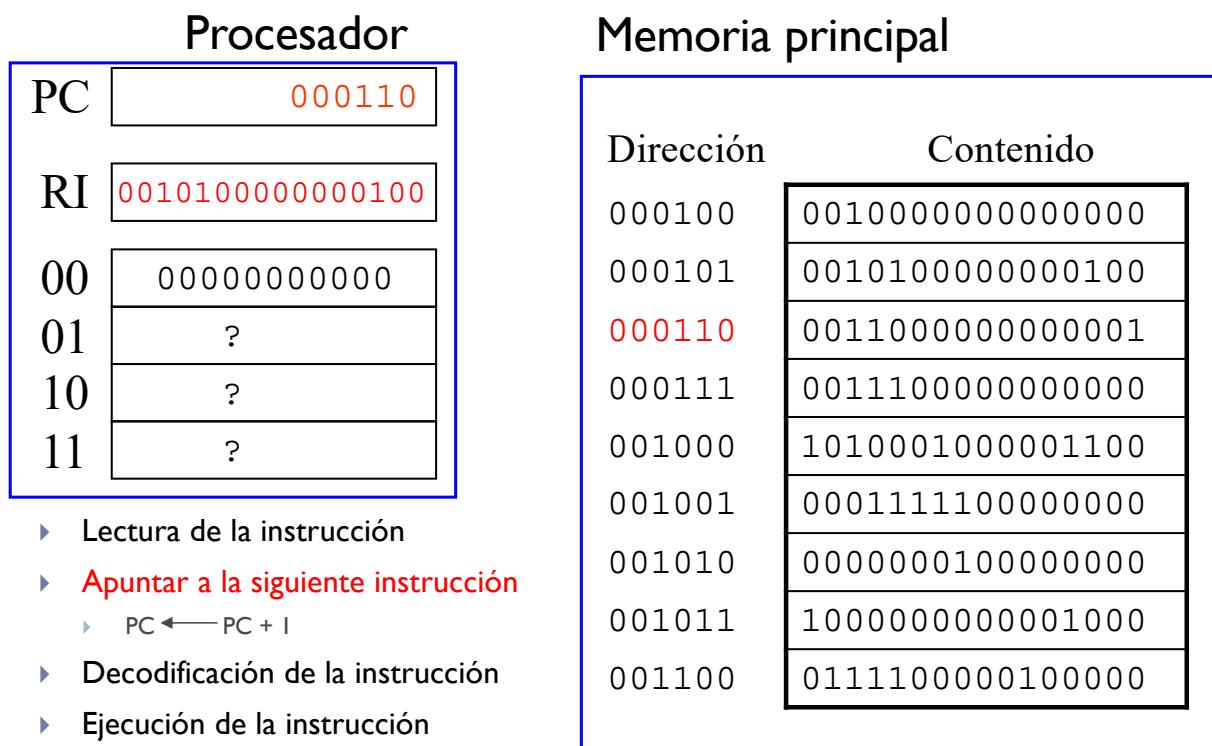
# Ejemplo de ejecución de un programa



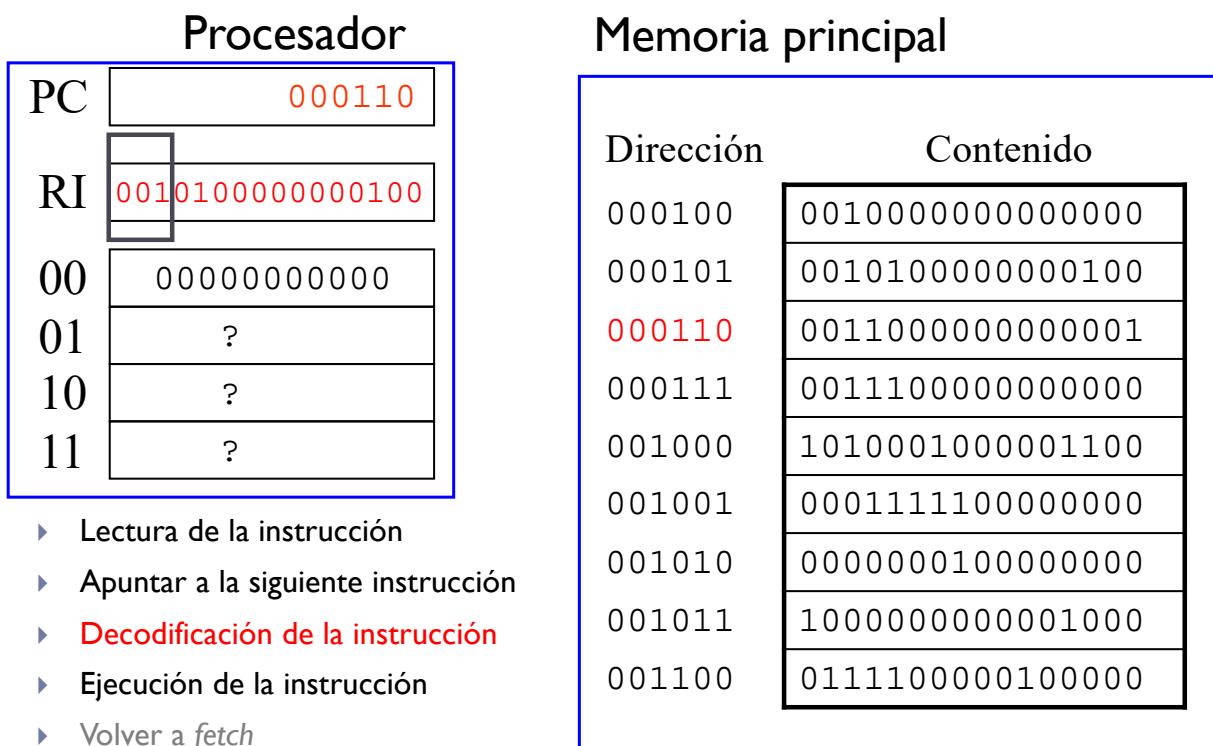
# Ejemplo de ejecución de un programa



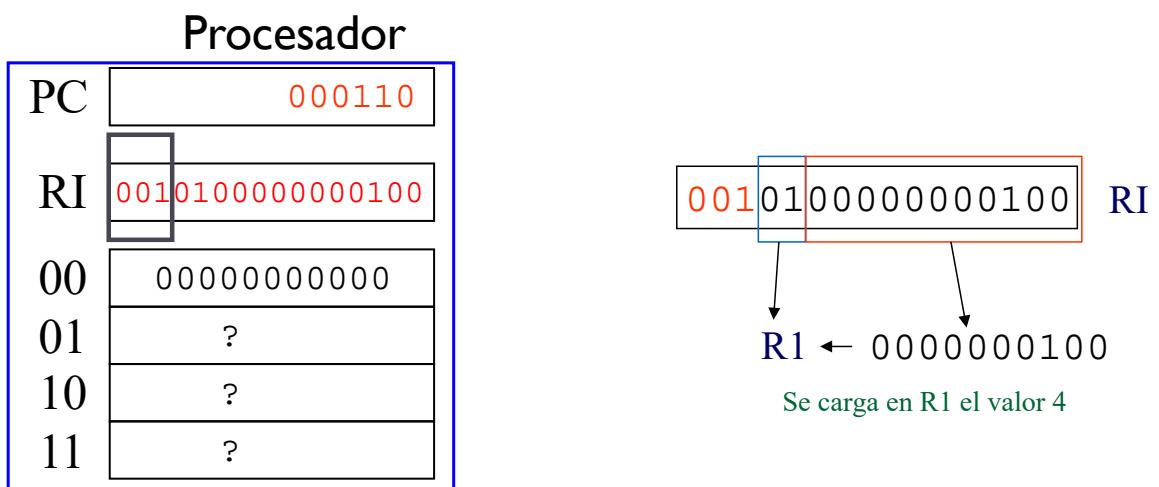
# Ejemplo de ejecución de un programa



# Ejemplo de ejecución de un programa

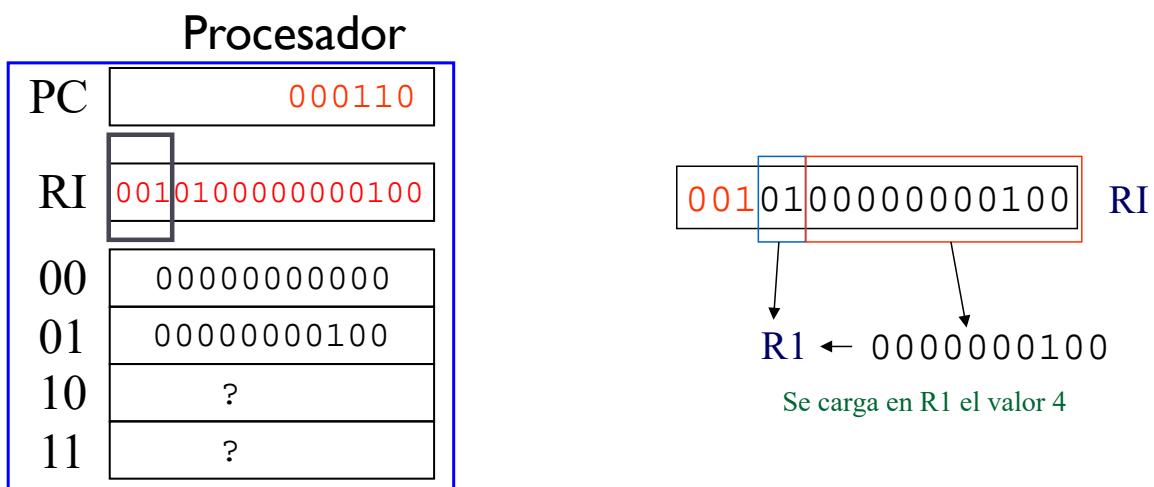


# Ejemplo de ejecución de un programa



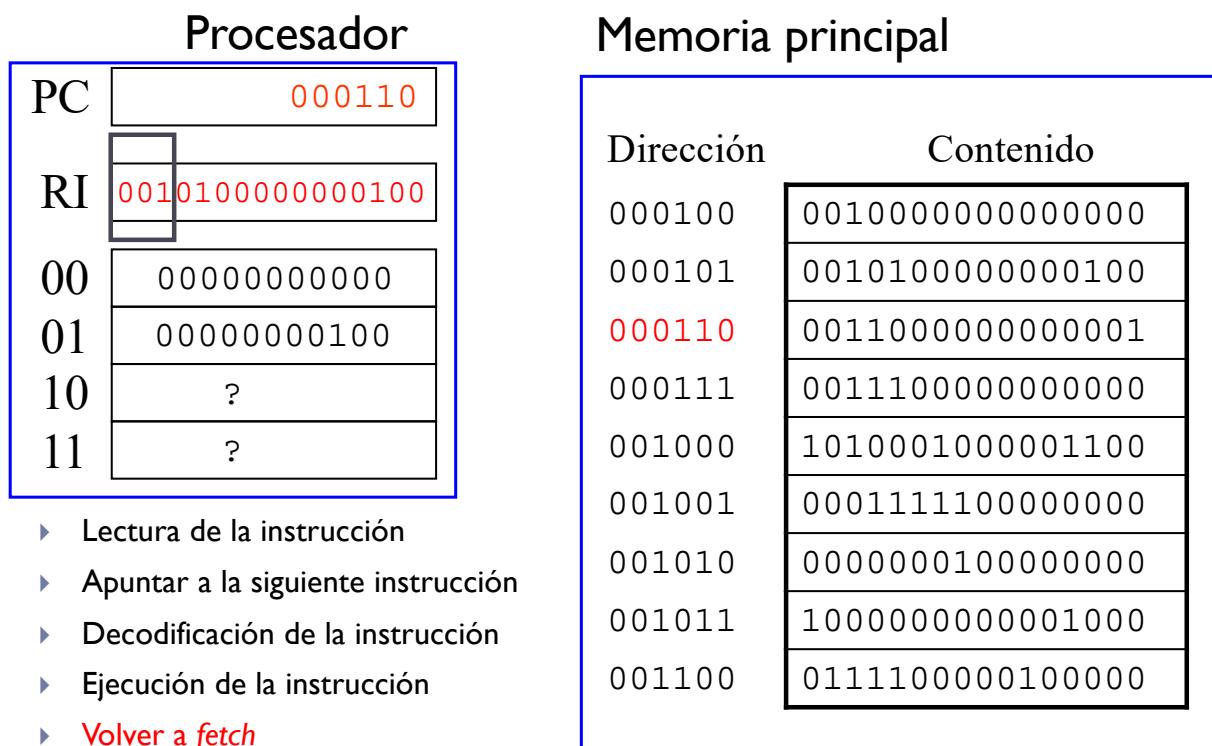
- ▶ Lectura de la instrucción
- ▶ Apuntar a la siguiente instrucción
- ▶ Decodificación de la instrucción
- ▶ Ejecución de la instrucción
- ▶ Volver a *fetch*

# Ejemplo de ejecución de un programa



- ▶ Lectura de la instrucción
- ▶ Apuntar a la siguiente instrucción
- ▶ Decodificación de la instrucción
- ▶ Ejecución de la instrucción
- ▶ Volver a fetch

# Ejemplo de ejecución de un programa



# Ejemplo de ejecución de un programa

Procesador	
PC	000110
RI	001010000000100
00	0000000000
01	00000000100
10	?
11	?

- ▶ Continúa la ejecución

Dirección	Contenido
000100	0010000000000000
000101	001010000000100
000110	0001100000000001
000111	0011100000000000
001000	1010001000001100
001001	0001111000000000
001010	0000000100000000
001011	100000000001000
001100	0111100000100000

## Algoritmo del programa anterior

```
i=0;  
s = 0;  
while ( i < 4 )  
{  
    s = s + 1;  
    i = i + 1;  
}
```

El programa almacena en la posición de memoria 00000100000  
el valor:  $1 + 1 + 1 + 1$

# Lenguaje ensamblador

- ▶ Utiliza códigos simbólicos y nemáticos para representar las instrucciones máquina que ejecuta un computador

	Instrucción en ensamblador	Instrucción máquina
	li R0, 0	0010000000000000
	li R1, 4	001010000000100
	li R2, 1	001100000000001
	li R3, 0	0011100000000000
bucle:	beq R0, R1, fin	1010001000001100
	add R3, R3, R2	0001111000000000
	add R0, R0, R2	0000000100000000
	b bucle	100000000001000
fin:	sw R3, 100000	0111100000100000

# Contenidos

1. **¿Qué es un computador?**
2. **Concepto de estructura y arquitectura**
3. **Elementos constructivos de un computador**
4. **Computador Von Neumann**
5. **Instrucciones máquina y programación**
6. **Fases de ejecución de una instrucción**
7. **Parámetros característicos de un computador**
8. **Tipos de computador**
9. **Evolución histórica**

# Parámetros característicos de un computador

- ▶ Respecto a su arquitectura
  - ▶ Ancho de palabra
- ▶ Almacenamiento
  - ▶ Tamaño
  - ▶ Unidades de almacenamiento
- ▶ Comunicaciones
  - ▶ Ancho de banda
  - ▶ Latencia
- ▶ Potencia del computador
  - ▶ MIPS
  - ▶ MFLOPS

# Ancho de Palabra

- ▶ Número de bits manejados en paralelo en el interior del computador.
  - ▶ Influye en el tamaño de los registros (BR)
  - ▶ Por tanto, también en la ALU
    - ▶ No es lo mismo dos sumas de 32 bits que una sola de 64
  - ▶ Por tanto, también en el ancho de los buses
    - ▶ Un bus de direcciones de 32 bits ‘solo’ direcciona 4 GB
- ▶ Tamaños típicos → 32 bits, 64 bits

# Tamaños privilegiados

- ▶ **Palabra**
  - ▶ Información manejada en paralelo en el interior del procesador
  - ▶ Típicamente 32/64 bits
- ▶ **Media palabra**
- ▶ **Doble palabra**
- ▶ **Octeto, carácter o byte**
  - ▶ Representación de un carácter
  - ▶ Típicamente 8 bits

## Ejercicio

- ▶ Considere un hipotético computador con un ancho de palabra de 20 bits con 60 registros que direcciona la memoria por bytes. Responda a las siguientes preguntas:
- ¿Cuántos bits se emplean para las direcciones de memoria?
  - ¿Cuál es el tamaño de los registros?
  - ¿Cuántos bits se almacenan en cada posición de memoria?
  - ¿Cuántas posiciones de memoria se pueden direccionar? Exprese el resultado en KB.
  - ¿Cuántos bits se necesitan para identificar a los registros?

# Tamaño de la Memoria

- ▶ **Tamaño de la memoria principal (RAM)**
  - ▶ Capacidad habitual: 512MB – 4 GB
  - ▶ Se expresa en octetos o bytes
- ▶ **Tamaño de la memoria auxiliar (Capacidad de almacenamiento de dispositivo de memoria secundaria)**
  - ▶ Papel: pocos bytes
  - ▶ Diskette: 1,44 KB
  - ▶ CD-ROM: 600 MB
  - ▶ DVD: 4.7GB
  - ▶ Blu-ray: 50 GB
  - ▶ Disco duro: 10 GB – 2 TB

# Unidades para tamaño

- ▶ Normalmente se expresa en octetos o bytes:

Nombre	Abr	Factor	SI
Kilo	K	$2^{10} = 1,024$	$10^3 = 1,000$
Mega	M	$2^{20} = 1,048,576$	$10^6 = 1,000,000$
Giga	G	$2^{30} = 1,073,741,824$	$10^9 = 1,000,000,000$
Tera	T	$2^{40} = 1,099,511,627,776$	$10^{12} = 1,000,000,000,000$
Peta	P	$2^{50} = 1,125,899,906,842,624$	$10^{15} = 1,000,000,000,000,000$
Exa	E	$2^{60} = 1,152,921,504,606,846,976$	$10^{18} = 1,000,000,000,000,000,000$
Zetta	Z	$2^{70} = 1,180,591,620,717,411,303,424$	$10^{21} = 1,000,000,000,000,000,000,000$
Yotta	Y	$2^{80} = 1,208,925,819,614,629,174,706,176$	$10^{24} = 1,000,000,000,000,000,000,000,000$

## Unidades para tamaño

- ▶ En **comunicación** se utilizan potencias de 10
  - ▶ 1 Kb = 1000 bits
  - ▶ 1 KB = 1000 bytes
- ▶ En **almacenamiento** algunos fabricantes no utilizan potencias de dos, sino potencias de 10:
  - ▶ kilobyte 1 KB = 1.000 bytes  $10^3$  bytes
  - ▶ megabyte 1 MB = 1.000 KB  $10^6$  bytes
  - ▶ gigabyte 1 GB = 1.000 MB  $10^9$  bytes
  - ▶ terabyte 1 TB = 1.000 GB  $10^{12}$  bytes
  - ▶ .....

## Ejercicio

- ▶ ¿Cuántos bytes tiene un disco duro de 200 GB?
  
- ▶ ¿Cuántos bytes por segundo transmite mi ADSL de 20 Mb?

## Ejercicio (solución)

- ▶ ¿Cuántos bytes tiene un disco duro de 200 GB?
  - ▶  $200 \text{ GB} = 200 * 10^9 \text{bytes} = 186.26 \text{ Gigabytes}$
- ▶ ¿Cuántos bytes por segundo transmite mi ADSL de 20 Mb?
  - ▶ B → Byte
  - ▶ b → bit.
  - ▶  $20 \text{ Mb} = 20 * 10^6 \text{bits} = 20 * 10^6 / 8 \text{ bytes} = 2.38 \text{ Megabytes por segundo}$

# Ancho de banda

- ▶ **Varias interpretaciones:**
  - ▶ Caudal de información que transmite un bus.
  - ▶ Caudal de información que transmite una unidad de E/S.
  - ▶ Caudal de información que puede procesar una unidad.
  - ▶ **Número de bits transferidos por unidad de tiempo.**
- ▶ **Unidades:**
  - ▶ Kb/s (Kilobits por segundo, no confundir con KB/s)
  - ▶ Mb/s (Megabits por segundo, no megabytes por segundo)

# Latencia

- ▶ **Varias interpretaciones:**
  - ▶ Tiempo transcurrido en la emisión de una petición en un sistema de mensajería fiable.
  - ▶ Tiempo transcurrido entre la emisión de una petición y la realización de la acción asociada.
  - ▶ **Tiempo transcurrido entre la emisión de una petición y la recepción de la respuesta.**
- ▶ **Unidades:**
  - ▶ s (segundos)

# Potencia de cómputo

- ▶ Medición de la potencia de cómputo.
- ▶ Factores que intervienen:
  - ▶ Juego de instrucciones
  - ▶ Reloj de la CPU (1 GHz vs 2 GHz vs 4 GHz...)
  - ▶ Número de 'cores' (quadcore vs dualcore vs...)
  - ▶ Ancho de palabra (32 bits vs 64 bits vs...)
- ▶ Formas típicas de expresar potencia de cómputo:
  - ▶ MIPS
  - ▶ MFLOPS
  - ▶ ...

# MIPS

- ▶ Millones de Instrucciones Por Segundo.
- ▶ Rango típico: 10-100 MIPS
- ▶ No todas las instrucciones tardan lo mismo en ejecutar
  - ➔ Depende de qué instrucciones se ejecutan.
- ▶ No es fiable 100% como medida de rendimiento.

# MFLOPS

- ▶ Millones de Operaciones en coma Flotante por Segundo.
- ▶ Potencia de cálculo científico.
- ▶ MFLOPS < MIPS (operación flotante más compleja que operación normal).
- ▶ Computadores vectoriales: MFLOPS > MIPS
- ▶ Ejemplo: Itanium 2 → 3,5 GFLOPS

## Vectores por segundo

- ▶ Potencia de cálculo en la generación de gráficos.
- ▶ Aplicable a procesadores gráficos.
- ▶ Se pueden medir en:
  - ▶ Vectores 2D.
  - ▶ Vectores 3D.
- ▶ Ejemplo: ATI Radeon 8500 → 3 Millones.

## Tests sintéticos

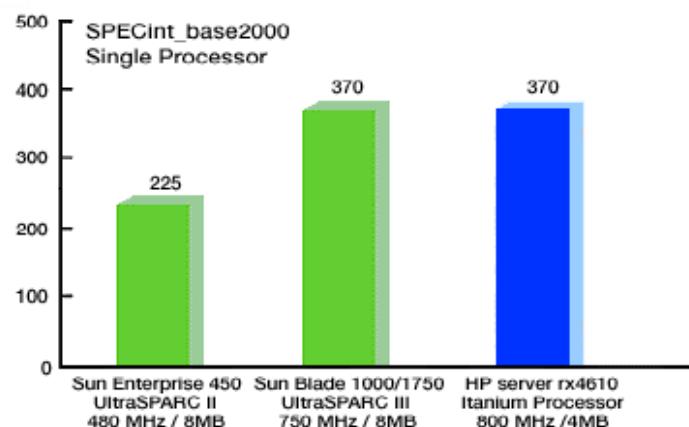
- ▶ MIPS y MFLOPS no válidos para comparar distintas máquinas.
  - ▶ Tests basados en ejecutar un mismo programa en distintas máquinas para compararlas.
  - ▶ Miden efectividad Compilador + CPU
- 
- ▶ Los test sintéticos estandarizados (“oficiales”) buscan comparar la potencia de dos computadores.
  - ▶ Es posible usar test sintéticos “no oficiales” para hacerse a la idea de la mejora con la carga de trabajo diaria

# Tests sintéticos “oficiales”

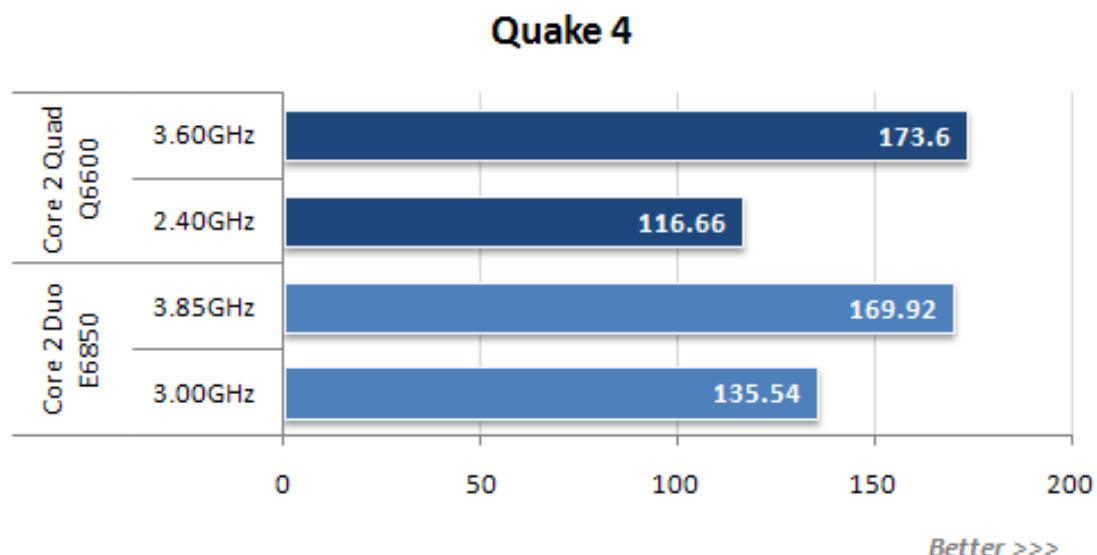
## ▶ Tests más usados:

- ▶ Linpack.
- ▶ SPEC.

**SPEC CPU2000 Performance – SPECint2000**  
*Itanium™ Processor delivers best of class floating point performance and competitive integer performance*

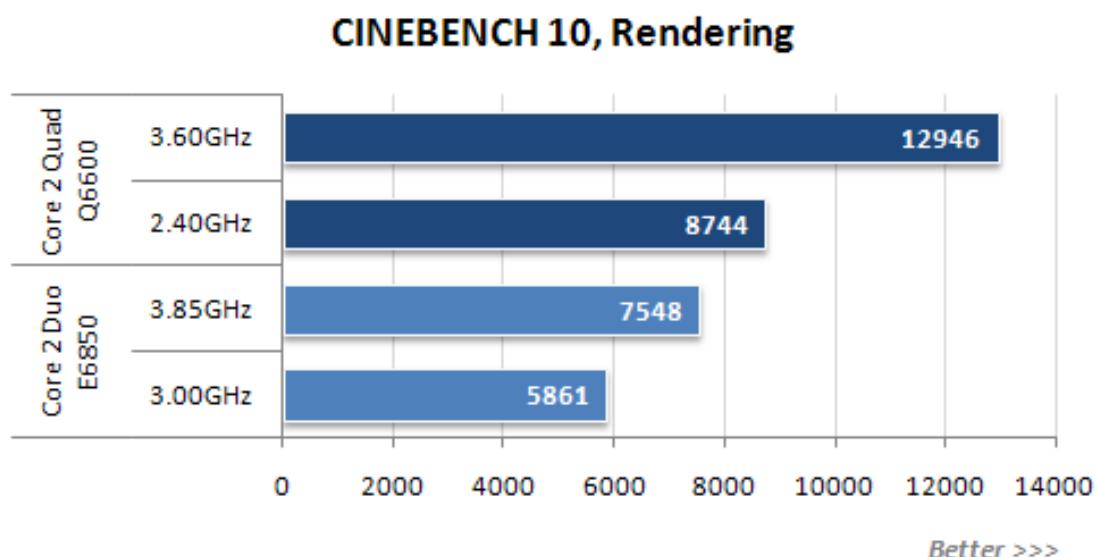


## Tests sintéticos “no oficiales”



[http://www.xbitlabs.com/articles/cpu/display/core2quad-q6600\\_11.html](http://www.xbitlabs.com/articles/cpu/display/core2quad-q6600_11.html)

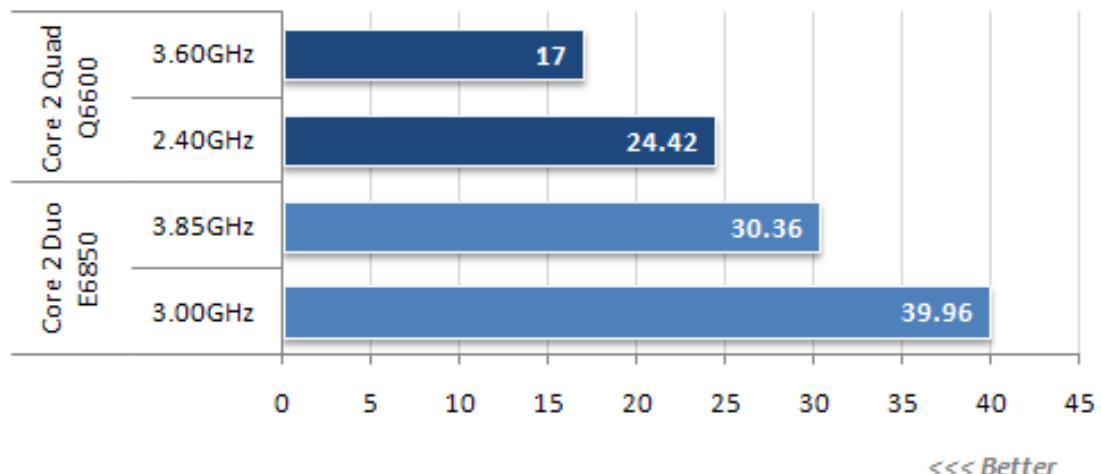
## Tests sintéticos “no oficiales”



[http://www.xbitlabs.com/articles/cpu/display/core2quad-q6600\\_11.html](http://www.xbitlabs.com/articles/cpu/display/core2quad-q6600_11.html)

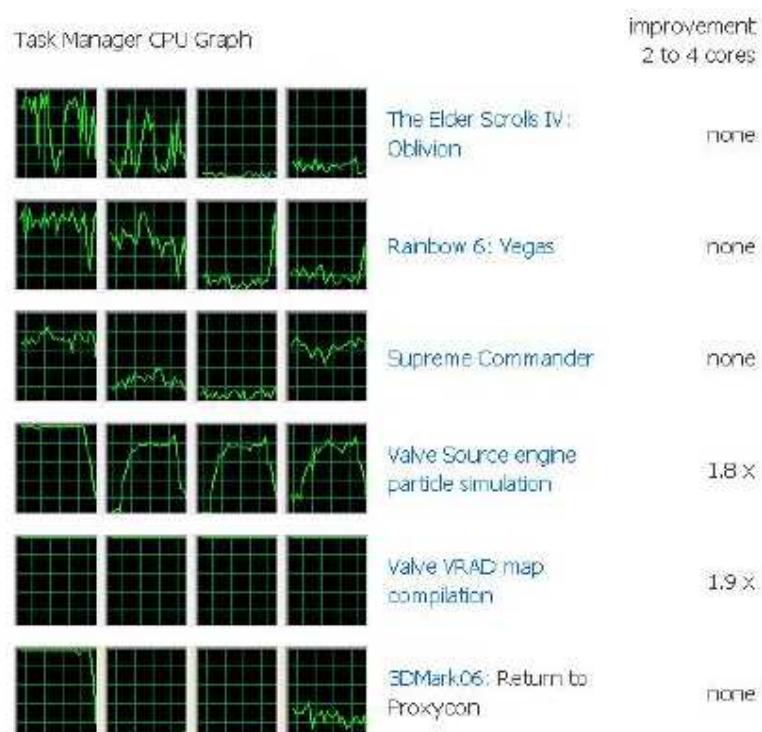
## Tests sintéticos “no oficiales”

**Excel 2007, sec**



[http://www.xbitlabs.com/articles/cpu/display/core2quad-q6600\\_11.html](http://www.xbitlabs.com/articles/cpu/display/core2quad-q6600_11.html)

# Tests sintéticos “no oficiales”



<http://www.codinghorror.com/blog/archives/000942.html>

# Contenidos

1. **¿Qué es un computador?**
2. **Concepto de estructura y arquitectura**
3. **Elementos constructivos de un computador**
4. **Computador Von Neumann**
5. **Instrucciones máquina y programación**
6. **Fases de ejecución de una instrucción**
7. **Parámetros característicos de un computador**
8. **Tipos de computadores**
9. **Evolución histórica**

# Tipos de computadores

- ▶ Dispositivos móviles personales
- ▶ Desktop
- ▶ Servidores
- ▶ Clusters
- ▶ Empotrados

# Tipos de computadores

## ▶ Desktop

- ▶ Diseñados para ofrecer un buen rendimiento a los usuarios
- ▶ Actualmente, la mayor parte son portátiles
- ▶ Aspectos de diseño:
  - ▶ Relación precio-rendimiento
  - ▶ Energía
  - ▶ Rendimiento de los gráficos

# Tipos de computadores

## ► Dispositivos móviles personales

- ▶ Dispositivos sin cables con interfaz de usuario multimedia
- ▶ Móviles, tablets,...
- ▶ Aspectos de diseño:
  - ▶ Precio
  - ▶ Energía
  - ▶ Rendimiento
  - ▶ Tiempo de respuesta

# Tipos de computadores

## ▶ Servidores

- ▶ Usados para ejecutar aplicaciones de alto rendimiento o escala
- ▶ Dan servicio a múltiples usuarios de forma simultánea
- ▶ Aspectos de diseño:
  - ▶ *Throughput* (Tasa de procesamiento)
  - ▶ Disponibilidad
  - ▶ Fiabilidad
  - ▶ Energía
  - ▶ Escalabilidad

# Tipos de computadores

## ▶ Clusters

- ▶ Conjunto de computadores conectados mediante una red que actúa como un único computador de más prestaciones
- ▶ Utilizando en supercomputadores y grandes centros de datos
- ▶ Aspectos de diseño:
  - ▶ Precio-rendimiento
  - ▶ *Throughput* (Tasa de procesamiento)
  - ▶ Disponibilidad
  - ▶ Fiabilidad
  - ▶ Energía
  - ▶ Escalabilidad

# Tipos de computadores

## ▶ **Empotrados**

- ▶ Computador que se encuentra dentro de otro sistema para controlar su funcionamiento
  - ▶ Lavadoras, TV, MP3, consolas de videojuegos, etc.
- ▶ Aspectos de diseño:
  - ▶ Precio
  - ▶ Energía
  - ▶ Rendimiento de la aplicación específica

# Contenidos

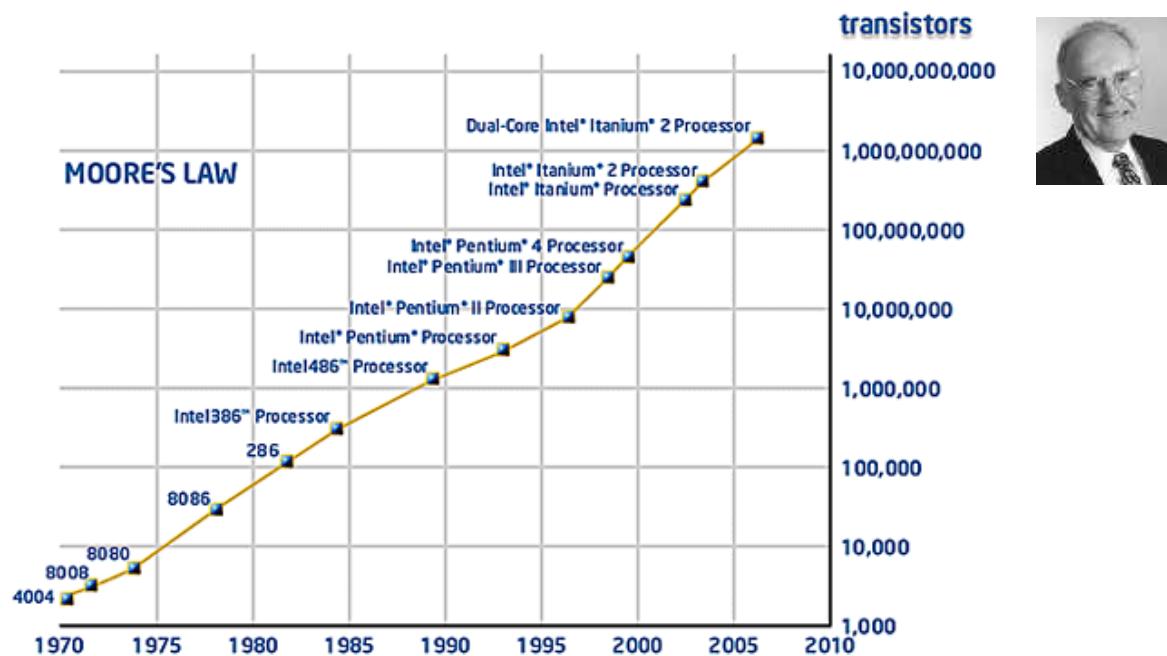
1. **¿Qué es un computador?**
2. **Concepto de estructura y arquitectura**
3. **Elementos constructivos de un computador**
4. **Computador Von Neumann**
5. **Instrucciones máquina y programación**
6. **Fases de ejecución de una instrucción**
7. **Parámetros característicos de un computador**
8. **Tipos de computadores**
9. **Evolución histórica**

# Microprocesador

- ▶ Incorpora las funciones de la CPU de un computador en un único circuito integrado

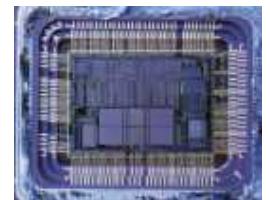


# Ley de Moore



# Ley de Moore

- ▶ Doblar la densidad implica reducir las dimensiones de sus elementos en un 30%
- ▶ En 1971 el Intel 4004 tenía 2.300 transistores con tamaños de 10 micrometros
- ▶ Hoy en día se consiguen chips con distancias de 14 nanometros
- ▶ Para cumplir la ley de Moore se necesita tecnología cuyo precio se dobla cada 4,4 años



# Mejoras en la tecnología

## ▶ Memoria

- ▶ Capacidad de DRAM: 2x / 2 años (desde 1996);  
**64x en la última década.**

## ▶ Procesador

- ▶ Velocidad: 2x / 1.5 años (desde 1985);  
**100X en la última década.**

## ▶ Discos

- ▶ Capacidad: 2x / 1 año (desde 1997)  
**250X en la última década.**

# Evolución histórica: bibliografía

- ▶ <http://history.sandiego.edu/GEN/recording/computer1.html>
- ▶ <http://www.computerhope.com/history/>
- ▶ <http://www.computerhistory.org/>
- ▶ <http://www.computersciencelab.com/ComputerHistory/History.htm>
- ▶ Museos de informática
- ▶ Buscar en google: “Computer history”



## **Parte III**

### **Tema 2. Representación de la información**



Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 2 Coma flotante

Estructura de Computadores  
Grado en Ingeniería Informática



# Contenidos

## 1. Introducción

1. Objetivo
2. Motivación
3. Sistemas posicionales

## 2. Representaciones

1. Alfanuméricas: caracteres y cadenas
2. Numéricas: naturales y enteras
3. **Numéricas: coma fija**
4. Numéricas: coma flotante: estándar IEEE 754

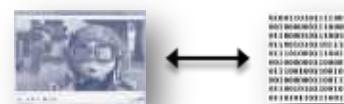
## Recordatorio: necesitaremos...

- ▶ Conocer **posibles representaciones**:



- ▶ Conocer las **características** de las mismas:

- ▶ Limitaciones



- ▶ Conocer **cómo operar** con la representación:



## Otras necesidades de representación

- ▶ ¿Cómo representar?
  - ▶ Números muy grandes:  $30.556.926.000_{(10)}$
  - ▶ Números muy pequeños:  $0.000000000529177_{(10)}$
  - ▶ Números con decimales:  $1,58567$

## Ejemplo de fallo...

- ▶ **Explosión del Ariane 5** (primer viaje)
  - ▶ Enviado por ESA en junio de 1996
  - ▶ Coste del desarrollo:  
**10 años y 7000 millones de dólares**
  - ▶ Explotó 40 segundos después de despegar,  
a 3700 metros de altura.
  - ▶ Fallo debido a la pérdida total de la información de altitud:
    - ▶ El software del sistema de referencia inercial realizó la  
conversión de un valor real en coma flotante de 64 bits a un  
valor entero de 16 bits. El número a almacenar era mayor de  
32767 (el mayor entero con signo de 16 bits) y se produjo un  
fallo de conversión y una excepción.



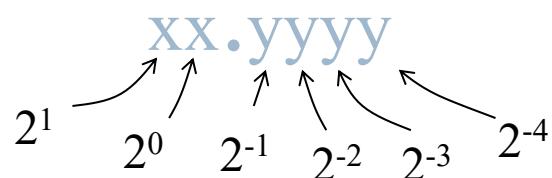
## Coma fija [racionales]

- ▶ Se fija la posición de la coma binaria y se utilizan los pesos asociados a las posiciones decimales
- ▶ Ejemplo:

$$1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9,625$$

## Representación de fracciones con representación binaria en coma fija

- ▶ Ejemplo de representación con 6 bits:



$$10,1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.62510$$

Asumiendo esta coma fija, el rango sería:

- 0 a 3.9375 (casi 4)

# Potencias negativas

i	$2^{-i}$	
0	1.0	1
1	0.5	$1/2$
2	0.25	$1/4$
3	0.125	$1/8$
4	0.0625	$1/16$
5	0.03125	$1/32$
6	0.015625	
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	

# Contenidos

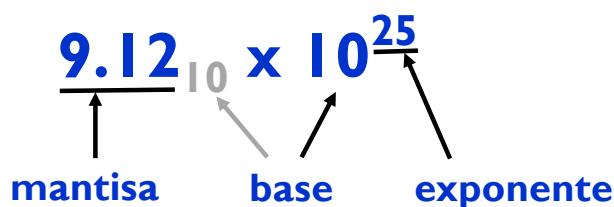
## 1. Introducción

1. Objetivo
2. Motivación
3. Sistemas posicionales

## 2. Representaciones

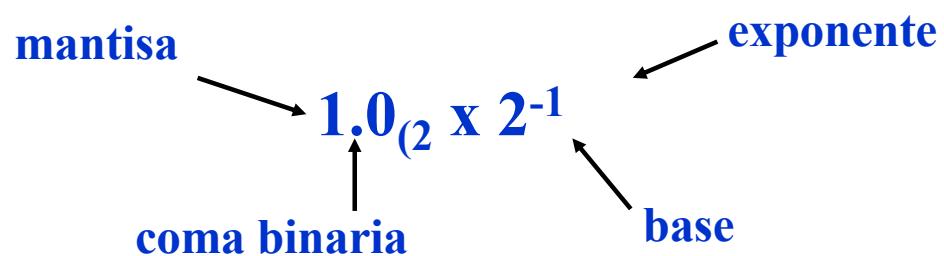
1. Alfanuméricas: caracteres y cadenas
2. Numéricas: naturales y enteras
3. Numéricas: coma fija
4. **Numéricas: coma flotante: estándar IEEE 754**

## Notación científica decimal



- ▶ Cada número lleva asociado una **mantisa** y un **exponente**
- ▶ Notación científica decimal usada: notación normalizada
  - ▶ Solo un dígito distinto de 0 a la izquierda del punto
- ▶ Se adapta el número al **orden de magnitud** del valor a representar, trasladando la *coma decimal* mediante el exponente

## Notación científica en binario



- ▶ Forma normalizada: Un 1 (solo un dígito) a la izquierda de la coma
  - ▶ Normalizada:  $1.0001 \times 2^{-9}$
  - ▶ No normalizada:  $0.0011 \times 2^{-8}$ ,  $10.0 \times 2^{-10}$

## Estándar IEEE 754 [racionales]



- ▶ Estándar para coma flotante usado en la mayoría de los ordenadores.
- ▶ **Características** (salvo casos especiales):
  - ▶ Exponente: en exceso con sesgo  $2^{\text{num\_bits\_exponente} - 1} - 1$
  - ▶ Mantisa: signo-magnitud, normalizada, con bit implícito
- ▶ Diferentes **formatos**:
  - ▶ **Precisión simple:** 32 bits (signo: 1, exponente: 8 y mantisa: 23)
  - ▶ **Doble precisión:** 64 bits (signo: 1, exponente: 11 y mantisa: 52)
  - ▶ **Cuádruple precisión:** 128 bits (signo: 1, exponente: 15 y mantisa: 112)

## Números normalizados

- ▶ En este estándar los números a representar tienen que estar **normalizados**. Un número normalizado es de la forma:
  - ▶ **1,bbbbbb × 2<sup>e</sup>**
  - ▶ mantisa: 1,bbbbbb (siendo b = 0, 1)
  - ▶ 2 es la base del exponente
  - ▶ e es el exponente

# Normalización y bit implícito

## ▶ Normalización

Para normalizar la mantisa se ajusta el exponente para que el bit más significativo de la mantisa sea 1

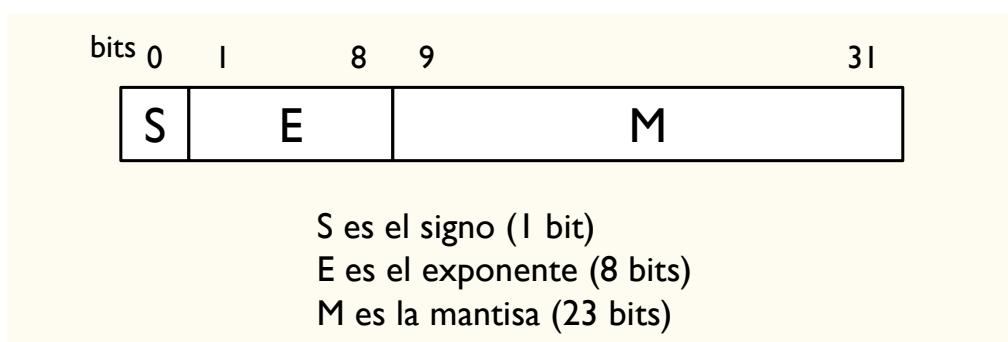
- ▶ Ejemplo: 10010000000000000000000000000000  $\times 2^3$  (ya está normalizado)
- ▶ Ejemplo: 00010000000010101  $\times 2^3$  (no lo está)  
100000000010101000  $\times 2^0$  (ahora sí)

## ▶ Bit implícito

Una vez normalizado, dado que el bit más significativo es 1, **no** se almacena para dejar espacio para un bit más (aumenta la precisión)

- ▶ Así se puede representar mantisas con un bit más

## Estándar IEEE 754 de precisión simple [racionales]



- El valor se calcula con la siguiente expresión (salvo casos especiales):

$$N = (-1)^S \times 2^{E-127} \times 1.M$$

donde:

$S = 0$  indica número positivo,  $S = 1$  indica número negativo

$0 < E < 255$  ( $E=0$  y  $E=255$  indican casos especiales)

$000000000000000000000000 \leq M \leq 11111111111111111111111$

## Estándar IEEE 754 de precisión simple [racionales]

- ▶ Existencia de casos especiales:

Exponente	Mantisa	Valor especial
0 (0000 0000)	0	+/- 0 (según signo)
0 (0000 0000)	No cero	Número NO normalizado
255 (1111 1111)	No cero	NaN (0/0,...)
255 (1111 1111)	0	+/-infinito (según signo)
-127	Cualquiera	Número normalizado (no especial)

$(-1)^s * 0.\text{mantisa} * 2^{-126}$

$(-1)^s * 1.\text{mantisa} * 2^{\text{exponente}-127}$

## Ejemplos (incluyen casos especiales)

S	E	M	N
1	00000000	0000000000000000000000000000	-0 (Excepción 0) E=0 y M=0.
1	01111111	0000000000000000000000000000	$-2^0 \times 1.0_2 = -1$
0	10000001	1110000000000000000000000000	$+2^2 \times 1.111_2 = +2^2 \times (2^0 + 2^{-1} + 2^{-2} + 2^{-3}) = +7.5$
0	11111111	0000000000000000000000000000	$\infty$ (Excepción $\infty$ ) E=255 y M=0
0	11111111	1000000000000000000000000001	NaN (Not a Number, como la raíz cuadrada de un número negativo) E=255 y M $\neq$ 0.

# Ejercicio

- a) Calcular el valor correspondiente al número  
0 10000011 1100000000000000000000000000  
dato en coma flotante según norma 754 de simple precisión

## Ejercicio (solución)

- a) Calcular el valor correspondiente al número  
0 10000011 1100000000000000000000000000  
dados en coma flotante según norma 754 de simple precisión

- a) Bit de signo:  $0 \Rightarrow (-1)^0 = +1$
- b) Exponente:  $10000011_2 = 131_{10} \Rightarrow E - 127 = 131 - 127 = 4$
- c) Mantisa:  $1100000000000000000000000000 \Rightarrow 1 \times 2^{-1} + 1 \times 2^{-2} = 0,75$

Por tanto el valor decimal del nº es  $+1 \times 2^4 \times 1,75 = +28$

# Ejercicio

- b) Expresar según norma IEEE 754 de simple precisión el n° -9

## Ejercicio (solución)

- b) Expresar según norma IEEE 754 de simple precisión el n° -9

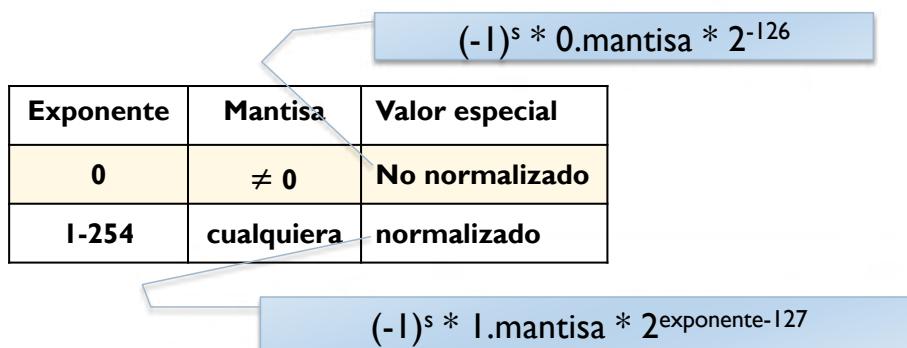
$$-9_{10} = -1001_2 = -1001_2 \times 2^0 = -1,001_2 \times 2^3 \text{ (mantisa normalizada)}$$

- a) Bit de signo: negativo  $\rightarrow S=1$
- b) Exponente:  $3+127$  (exceso) = 130  $\rightarrow 10000010$
- c) Mantisa: 1,001 (bit impl.)  $\rightarrow 00100000000000000000000000000000$

Por tanto  $-9 = 1\ 1000010\ 00100000000000000000000000000000$

## Estándar IEEE 754 de precisión simple [racionales]

- ▶ Rango de magnitudes representables (sin considerar el signo):
  - ▶ Menor normalizado:  
 $2^{-1-127} \times 1.00000000000000000000000000_2$
  - ▶ Mayor normalizado:  
 $2^{254-127} \times 1.1111111111111111111111_2$
  - ▶ Menor no normalizado:  
 $2^{-126} \times 0.000000000000000000000000001_2$
  - ▶ Mayor no normalizado:  
 $2^{-126} \times 0.1111111111111111111111_2$



## Estándar IEEE 754 de precisión simple [racionales]

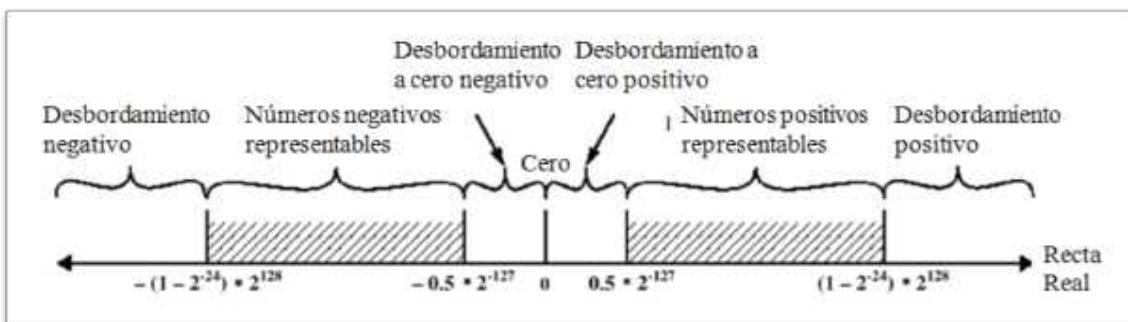
- ▶ Rango de magnitudes representables (sin considerar el signo):
  - ▶ Menor normalizado:  
 $2^{-1-127} \times 1.00000000000000000000000000_2 = 2^{-126}$
  - ▶ Mayor normalizado:  
 $2^{254-127} \times 1.11111111111111111111111111_2 = 2^{127} \times (2 - 2^{-23}) = 2^{128} \times (1 - 2^{-24})$
  - ▶ Menor no normalizado:  
 $2^{-1-126} \times 0.000000000000000000000000001_2 = 2^{-149}$
  - ▶ Mayor no normalizado:  
 $2^{-1-126} \times 0.11111111111111111111111111_2 = 2^{-126} \times (1 - 2^{-23})$

Truco:

$$\begin{array}{rcl} 1.11111111111111111111111111_2 & = & X \\ + 0.000000000000000000000000001_2 & = & 2^{-23} \\ \hline 10.00000000000000000000000000_2 & = & 2 \end{array}$$
$$X = 2 - 2^{-23}$$

## Estándar IEEE 754 de precisión simple [racionales]

- ▶ Rango de magnitudes representables (sin considerar el signo):
  - ▶ Menor normalizado:  
 $2^{-127} \times 1.0000000000000000000000000_2 = 2^{-126} = 2^{-127} \times 0.5$
  - ▶ Mayor normalizado:  
 $2^{254-127} \times 1.11111111111111111111111_2 = 2^{127} \times (2 - 2^{-23}) = 2^{128} \times (1 - 2^{-24})$
  - ▶ Menor no normalizado:  
 $2^{-126} \times 0.0000000000000000000000000_2 = 2^{-149}$
  - ▶ Mayor no normalizado:  
 $2^{-126} \times 0.11111111111111111111111_2 = 2^{-126} \times (1 - 2^{-23})$



## Ejercicio

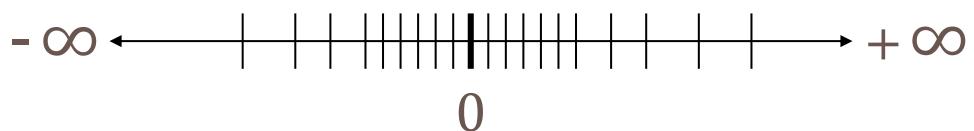
- ▶ ¿Cuántos números de *floats* (coma flotante de simple precisión) hay entre el 1 y el 2 (no incluido)?
  
- ▶ ¿Cuántos números de *floats* (coma flotante de simple precisión) hay entre el 2 y el 3 (no incluido)?

## Ejercicio (solución)

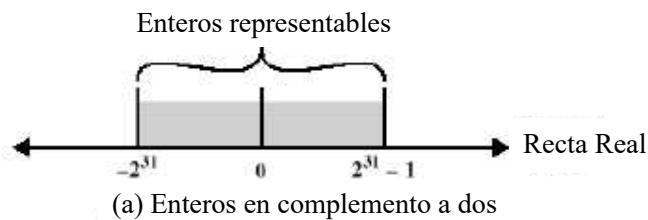
- ▶ ¿Cuántos números de *floats* (coma flotante de simple precisión) hay entre el 1 y el 2 (no incluido)?
  - ▶  $1 = 1,000000000000000000000000000 \times 2^0$
  - ▶  $2 = 1,000000000000000000000000000 \times 2^1$
  - ▶ Entre 1 y 2 hay  $2^{23}$  números
- ▶ ¿Cuántos números de *floats* (coma flotante de simple precisión) hay entre el 2 y el 3 (no incluido)?
  - ▶  $2 = 1,000000000000000000000000000 \times 2^1$
  - ▶  $3 = 1,100000000000000000000000000 \times 2^1$
  - ▶ Entre 2 y 3 hay  $2^{22}$  números

# Números representables

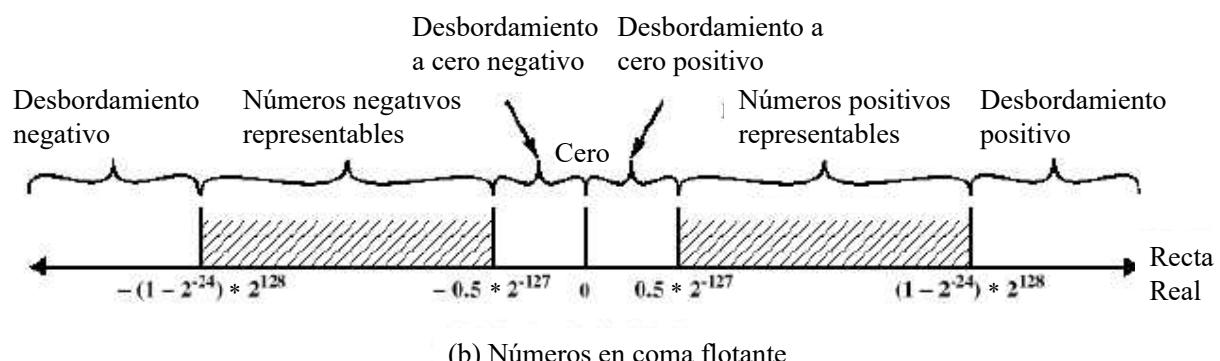
- ▶ Resolución variable:  
Más denso cerca de cero, menos hacia el infinito



# Números representables



(a) Enteros en complemento a dos



(b) Números en coma flotante

## Ejemplo 1 imprecisión

**0,4 →**

0	0111101	1001100110011001101
---	---------	---------------------

  
  
**3.999998 × 10<sup>-1</sup>**

**0,1 →**

0	01111011	1001100110011001100
---	----------	---------------------

  
  
**9.999994 × 10<sup>-2</sup>**

## Ejemplo 2 imprecisión

- ▶ ¿Cómo realiza C una división?

t2.c

```
#include <stdio.h>

int main ( )
{
    float a ;

    a = 3.0/7.0 ;
    if (a == 3.0/7.0)
        printf("Igual\n") ;
    else printf("No Igual\n") ;
    return (0) ;
}
```

## Ejemplo 2 imprecisión

- ▶ ¿Cómo realiza C una división?

t2.c

```
#include <stdio.h>

int main ()
{
    float a ;

    a = 3.0/7.0 ;
    if (a == 3.0/7.0)
        printf("Igual\n") ;
    else printf("No Igual\n") ;
    return (0) ;
}
```

\$ gcc -o t2 t2.c

\$ ./t2

No Igual

## Ejemplo 2 imprecisión

- ▶ ¿Cómo realiza C una división?

```
t2.c
#include <stdio.h>

int main ()
{
    float a ;
    a = 3.0/7.0 ;
    if (a == 3.0/7.0)
        printf("Igual\n") ;
    else printf("No Igual\n") ;
    return (0) ;
}
```

A diagram showing the flow from the source code file `t2.c` to its execution. A red box encloses the code. A red arrow points from the top right of the code box to a white box labeled `t2.c`. From there, another red arrow points down to a terminal window.

```
$ gcc -o t2 t2.c
$ ./t2
No Igual
```

## Ejemplo 3 imprecisión

- ▶ La propiedad asociativa no siempre se cumple  
 $a + (b + c) = (a + b) + c$  ?

t1.c

```
#include <stdio.h>

int main ( )
{
    float x, y, z ;

    x = 10e30;  y = -10e30;  z = 1;
    printf("(x+y)+z = %f\n", (x+y)+z) ;
    printf("x+(y+z) = %f\n", x+(y+z)) ;

    return (0) ;
}
```

## Ejemplo 3 imprecisión

- ▶ La propiedad asociativa no siempre se cumple  
 $a + (b + c) = (a + b) + c$  ?

t1.c

```
#include <stdio.h>

int main ()
{
    float x, y, z ;

    x = 10e30;  y = -10e30;  z = 1;
    printf("(x+y)+z = %f\n", (x+y)+z) ;
    printf("x+(y+z) = %f\n", x+(y+z)) ;

    return (0) ;
}
```

```
$ gcc -o t1 t1.c
$ ./t1
(x+y)+z = 1.000000
x+(y+z) = 0.000000
```

# Redondeo

- ▶ El redondeo elimina cifras menos significativas de un número para obtener un valor aproximado.
- ▶ **Tipos de redondeo:**
  - ▶ Redondeo **hacia  $+\infty$** 
    - ▶ Redondeo “hacia arriba”:  $2.001 \rightarrow 3, -2.001 \rightarrow -2$
  - ▶ Redondeo **hacia  $-\infty$** 
    - ▶ Redondea “hacia abajo”:  $1.999 \rightarrow 1, -1.999 \rightarrow -2$
  - ▶ **Truncar**
    - ▶ Descarta los últimos bits:  $1.299 \rightarrow 1.2$
  - ▶ **Redondeo al más cercano**
    - ▶  $2.4 \rightarrow 2, 2.6 \rightarrow 3, -1.4 \rightarrow -1$

# Redondeo

- ▶ El redondeo **supone ir perdiendo precisión.**
- ▶ El redondeo **ocurre:**
  - ▶ Al pasar a una representación con menos representables:
    - ▶ Ej.: Un valor de doble a simple precisión
    - ▶ Ej.: Un valor en coma flotante a entero
  - ▶ Al realizar operaciones aritméticas:
    - ▶ Ej.: Después de sumar dos números en coma flotante  
(al usar dígitos de guarda)

## Dígitos de guarda

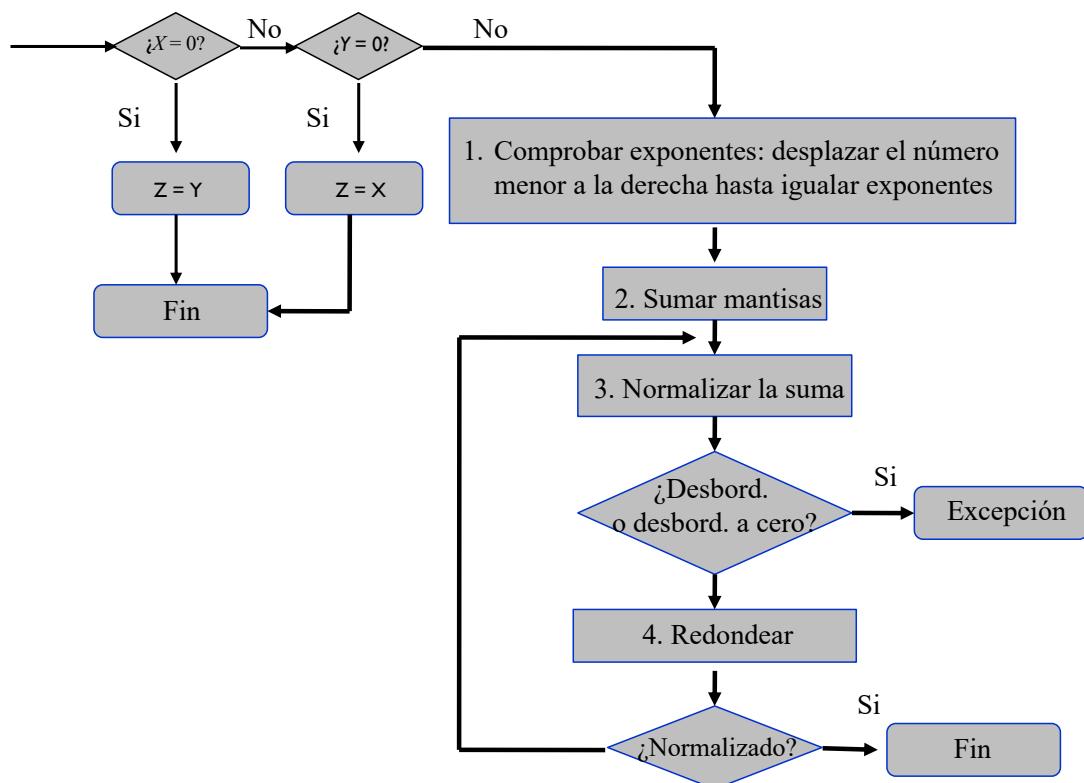
- ▶ Se utilizan **dígitos de guarda** para mejorar la precisión: internamente se usan dígitos adicionales para operar.
- ▶ Ejemplo:  $2,65 \times 10^0 + 2.34 \times 10^2$

	SIN dígitos de guarda	CON dígitos de guarda
1.- igualar exponentes	$0,02 \times 10^2$ $+ 2,34 \times 10^2$	$0,02\textcolor{blue}{65} \times 10^2$ $+ 2,\textcolor{blue}{34}\textcolor{blue}{00} \times 10^2$
2.- sumar	$2,36 \times 10^2$	$2,36\textcolor{blue}{65} \times 10^2$
3.- redondear	$2,3\textcolor{red}{6} \times 10^2$	$2,3\textcolor{red}{7} \times 10^2$

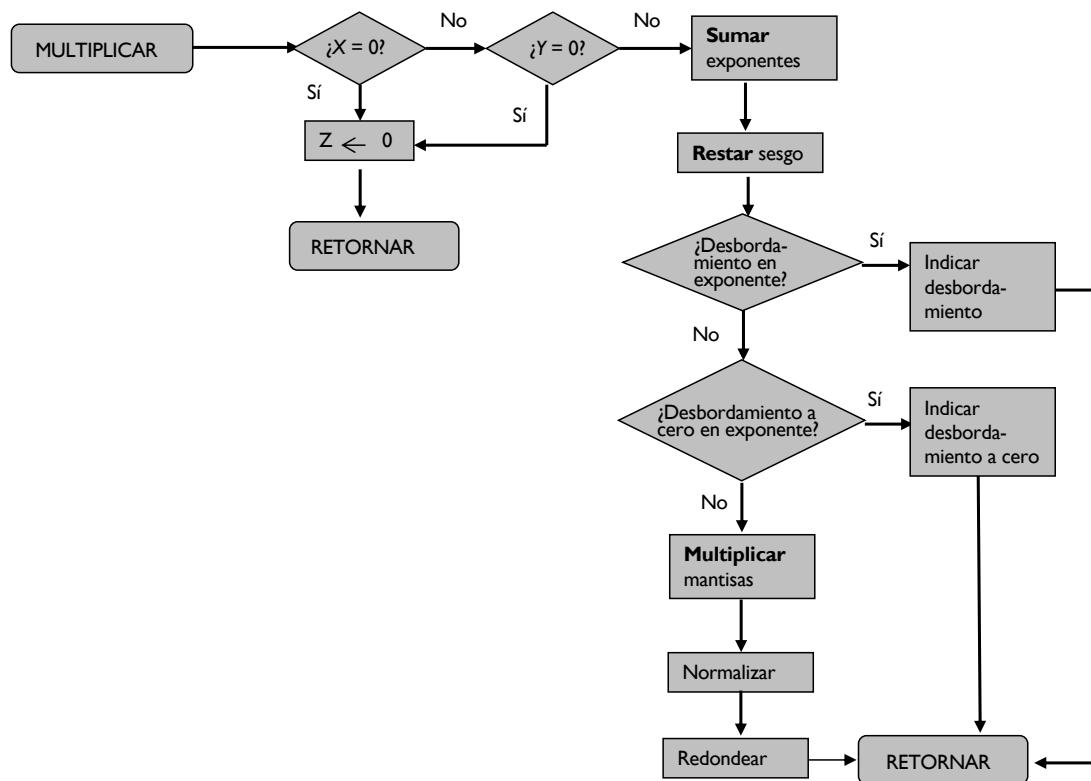
# Operaciones en coma flotante

- ▶ **Sumar**
- ▶ **Restar**
  1. Comprobar valores cero.
  2. Igualar exponentes (desplazar número menor a la derecha).
  3. Sumar/restar las mantisas.
  4. Normalizar el resultado.
- ▶ **Multiplicar**
- ▶ **Dividir**
  1. Comprobar valores cero.
  2. Sumar/restar exponentes.
  3. Multiplicar/dividir mantisas (teniendo en cuenta el signo).
  4. Normalizar el resultado.
  5. Redondear el resultado.

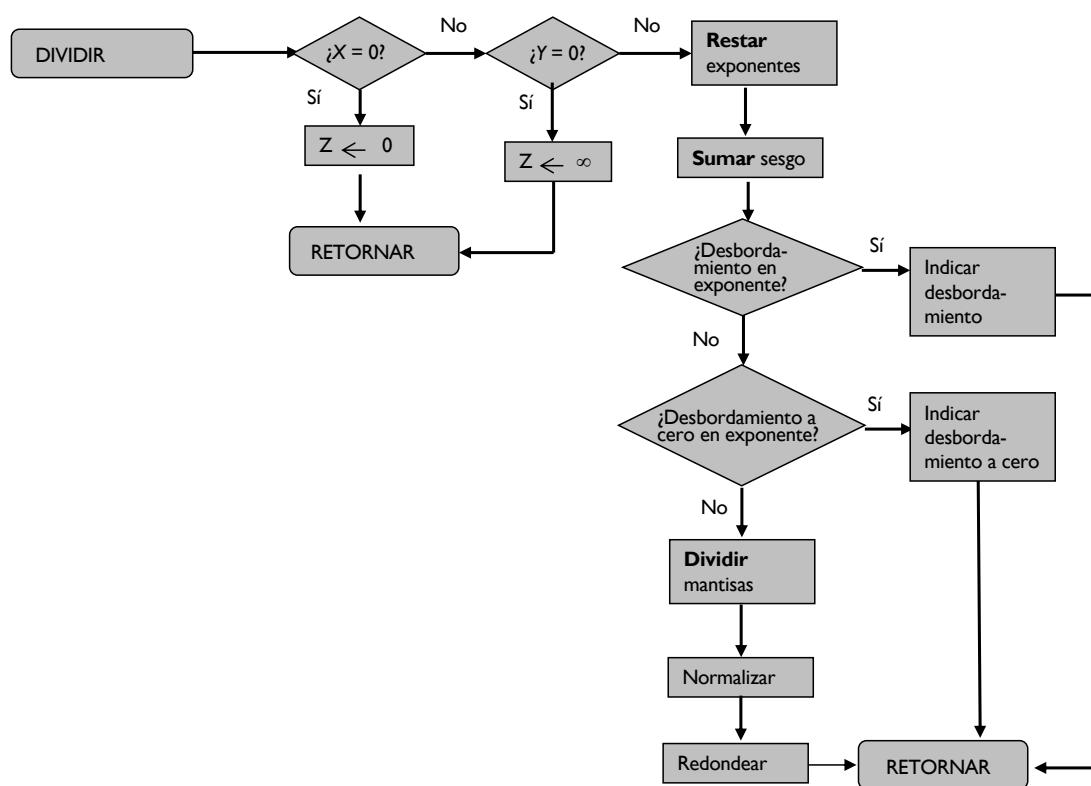
## Suma y resta: $Z=X+Y$ y $Z=X-Y$



## Multiplicación: $Z=X \cdot Y$



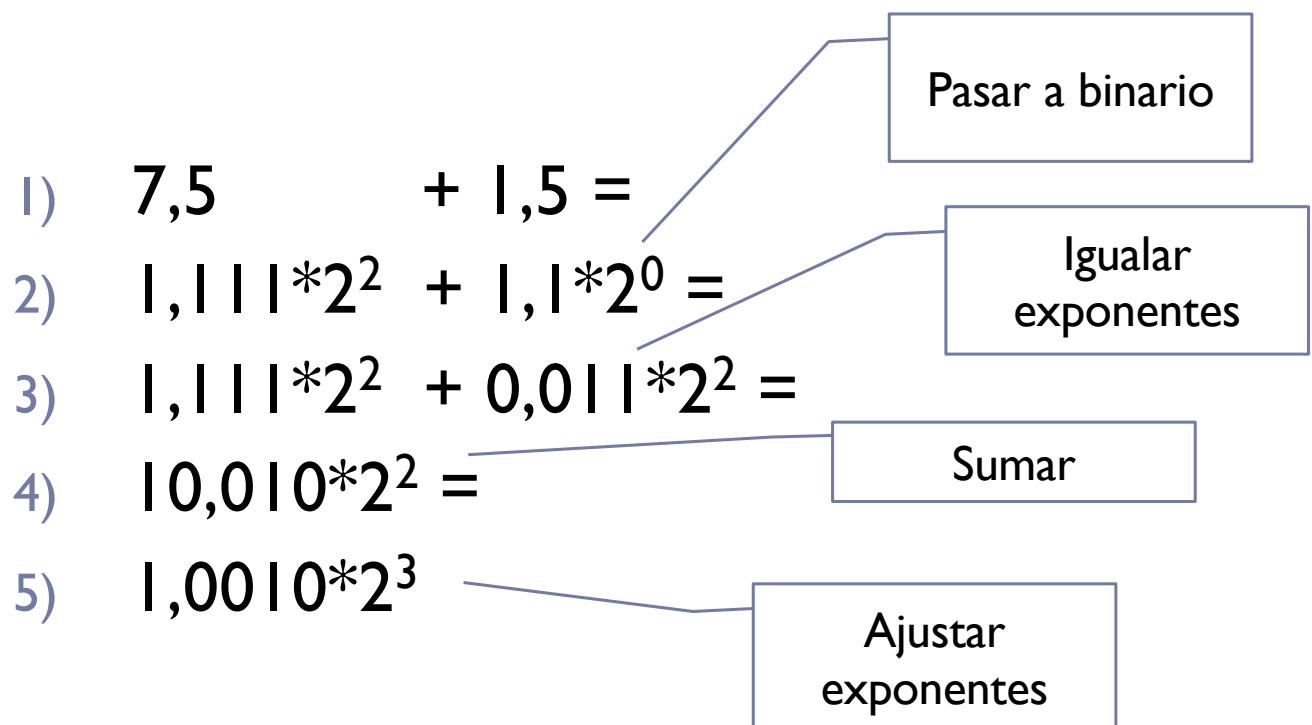
## División: $Z=X/Y$



# Ejercicio

- ▶ Usando el formato IEEE 754, sumar 7,5 y 1,5 paso a paso

## Solución



## Solución

### ▶ Representación de los números

**7,5 → 0 10000001 1100000000000000000000000000**

+ **1,5 → 0 0111111 1000000000000000000000000000**

## Solución

- ▶ Se separa exponentes y mantisas y se añade el bit implícito

$7,5 \rightarrow$	0	10000001	I.11100000000000000000000000000000
$+ 1,5 \rightarrow$	0	01111111	I.10000000000000000000000000000000
<hr/>			

# Solución

## ▶ Igualar exponentes

$$\begin{array}{r} 7,5 \rightarrow 0 \boxed{10000001} \boxed{1.11100000000000000000000000000000} \\ + \quad 1,5 \rightarrow 0 \boxed{01111111} \boxed{1.10000000000000000000000000000000} \\ \hline \end{array}$$

Above the first binary number, there is a red arrow pointing to the leftmost '1' in the significand, and a red vertical bar is positioned between the two binary numbers, extending from the decimal point to the right.

# Solución

## ▶ Igualar exponentes

$7,5 \rightarrow$	0	10000001	I.11100000000000000000000000000000
$+ 1,5 \rightarrow$	0	01111111	I.10000000000000000000000000000000
			<hr/>

# Solución

## ▶ Igualar exponentes

$$\begin{array}{r} 7,5 \rightarrow 0 \boxed{10000001} \boxed{1.11100000000000000000000000000000} \\ + 1,5 \rightarrow 0 \boxed{10000000} \boxed{0.11100000000000000000000000000000} \\ \hline \end{array}$$

The diagram shows two floating-point numbers being added. The first number is 7.5, represented as 1.11100000000000000000000000000000 × 10<sup>1</sup>. The second number is 1.5, represented as 0.11100000000000000000000000000000 × 10<sup>0</sup>. A red bracket indicates that the exponent of the second number needs to be increased by 1 to match the first number's exponent. A double-headed arrow above the numbers shows the value "1" with a "+1" above it, indicating the shift of the decimal point.

# Solución

## ▶ Igualar exponentes

$$\begin{array}{r} 7,5 \rightarrow 0 \boxed{10000001} | 1.11100000000000000000000000 \\ + 1,5 \rightarrow 0 \boxed{10000001} | 0.01100000000000000000000000 \\ \hline \end{array}$$

The diagram shows two floating-point numbers being added. The first number is 7.5, represented in binary as 1.11100000000000000000000000, with an exponent of 7,5. The second number is 1.5, represented in binary as 0.01100000000000000000000000, with an exponent of 1,5. To add them, the exponents must be equal. A red bracket labeled '+1' indicates that the exponent of the second number is increased by 1 to match the first. The resulting sum is 1.11100000000000000000000000.

# Solución

## ▶ Sumar mantisas

$$\begin{array}{r} 7,5 \rightarrow 0 \boxed{10000001} \boxed{1.11100000000000000000000000000000} \\ + 1,5 \rightarrow 0 \boxed{10000001} \boxed{0.01100000000000000000000000000000} \\ \hline \end{array}$$

A red vertical line is drawn through the binary digits of the fractions in both numbers, starting from the first digit after the decimal point.

# Solución

## ▶ Normalizar el resultado

$7,5 \rightarrow$	0   10000001   1.1110000000000000000000000000000
$+ 1,5 \rightarrow$	0   10000001   0.0110000000000000000000000000000
	<hr/>
$9 \rightarrow$	0   10000001   10.0100000000000000000000000000000

Se produce un acarreo, mantisa no  
normalizada

# Solución

## ▶ Normalizar el resultado

$$\begin{array}{r} 7,5 \rightarrow 0 | 10000001 | 1.11100000000000000000000000 \\ + 1,5 \rightarrow 0 | 10000001 | 0.01100000000000000000000000 \\ \hline 9 \rightarrow 0 | 10000001 | 10.01000000000000000000000000 \end{array}$$

The diagram shows the addition of two floating-point numbers. The first number is 7.5 (binary 1.11100000000000000000000000) and the second is 1.5 (binary 0.01100000000000000000000000). The result is 9 (binary 10.01000000000000000000000000). A red vertical line indicates the decimal point. A double-headed arrow between the 1 and the 0 in the result's fraction part is labeled '+1' above it, indicating a rightward shift of one bit position.

## Solución

$$\begin{array}{r} 7,5 \rightarrow 0 \boxed{10000001} \boxed{1.11100000000000000000000000000000} \\ + \quad 1,5 \rightarrow 0 \boxed{10000001} \boxed{0.01100000000000000000000000000000} \\ \hline 9 \rightarrow 0 \boxed{10000010} \boxed{\color{blue}{1.00100000000000000000000000000000}} \end{array}$$

## Solución

- ▶ Se almacena el resultado eliminando el bit implícito

**9 → 0 10000010 00100000000000000000000000**

# Ejercicio

- ▶ Usando el formato IEEE 754, calcular  $9 - 7.5$  paso a paso

## Solución

### ▶ Representación de los números

**9 → 0 10000010 00100000000000000000000000000000**

- **7,5 → 1 10000001 11100000000000000000000000000000**

## Solución

- ▶ Se separan exponentes y mantisas y se añade bit implícito

$9 \rightarrow$	0	10000010	1.0010000000000000000000000000000
- 7,5 $\rightarrow$	I	10000001	1.1110000000000000000000000000000
<hr/>			
Se añade el bit implícito para operar			

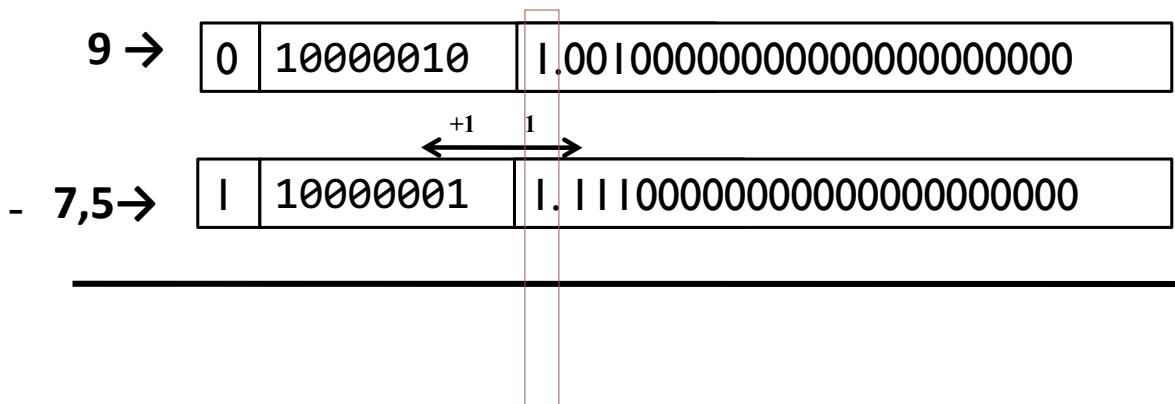
# Solución

## ▶ Igualar exponentes

$9 \rightarrow$	0	10000010	1.0010000000000000000000000000000
- $7,5 \rightarrow$	1	10000001	1.1110000000000000000000000000000
<hr/>			

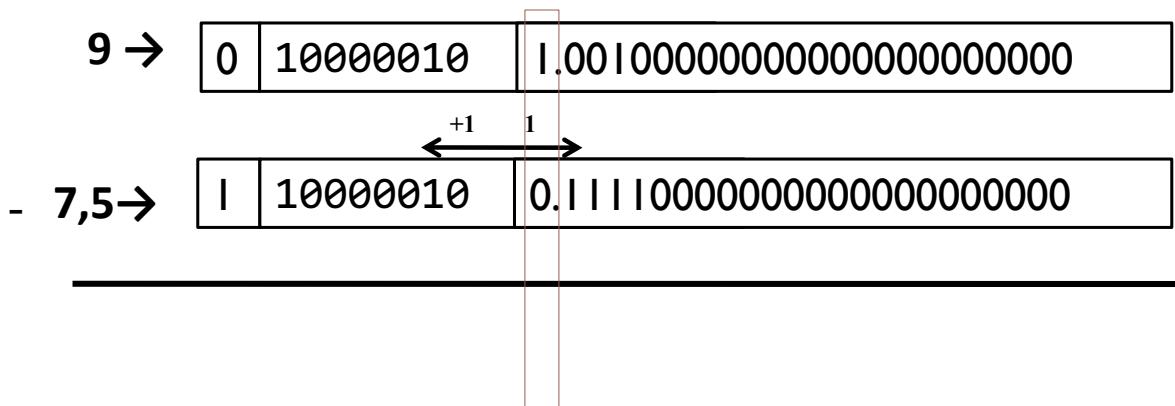
# Solución

## ► Igualar exponentes



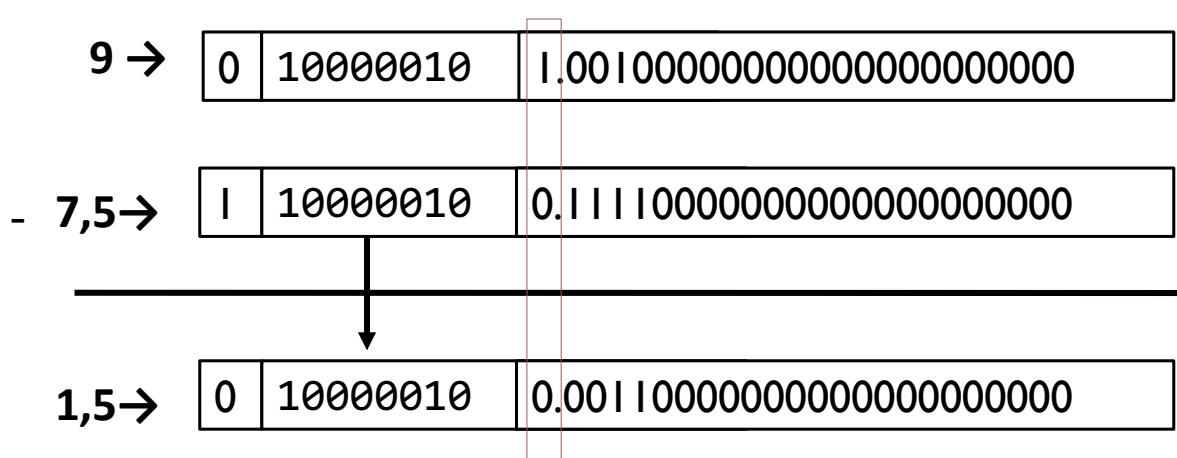
# Solución

## ► Igualar exponentes



# Solución

## ▶ Resta



# Solución

## ▶ Normalizar el resultado

$9 \rightarrow$	0   10000010   1.00100000000000000000000000000000
- $7,5 \rightarrow$	I   10000010   0.11110000000000000000000000000000
<hr/>	
$1,5 \rightarrow$	0   10000010   0.00110000000000000000000000000000

# Solución

## ▶ Normalizar el resultado

$9 \rightarrow$	0	10000010	1.0010000000000000000000000000000
- $7,5 \rightarrow$	1	10000010	0.1111000000000000000000000000000
<hr/>			
$1,5 \rightarrow$	0	10000001	0.0110000000000000000000000000000

# Solución

## ▶ Normalizar el resultado

# Solución

## ▶ Normalizar el resultado

**9 →**

0	10000010	1.0010000000000000000000000000000
---	----------	-----------------------------------

- **7,5 →**

I	10000010	0.1111000000000000000000000000000
---	----------	-----------------------------------

---

**1,5 →**

0	01111111	1.1000000000000000000000000000000
---	----------	-----------------------------------

mantisa ya normalizada

## Solución

- ▶ Se almacena el resultado definitivo eliminando el bit implícito

**1,5→ 0 01111111 100000000000000000000000000000**

# Ejercicio

- ▶ Usando el formato IEEE 754, multiplicar 7,5 y 1,5 paso a paso

## Solución

### ▶ Representación de los números

**9 → 0 10000010 00100000000000000000000000000000**

- **7,5 → 1 10000001 11100000000000000000000000000000**

## Solución

- ▶ Se separan exponentes y mantisas y se añade bit implícito

$7,5 \rightarrow$	0	10000001	1.111000000000000000000000000000
X $1,5 \rightarrow$	0	01111111	1.100000000000000000000000000000
<hr/>			

Se añade el bit implícito para operar

## Solución

- ▶ Multiplicar: sumar exponentes y multiplicar mantisas

$7,5 \rightarrow$	0   10000001	1.1110000000000000000000000000000
$\times$	$1,5 \rightarrow$	0   01111111   1.1000000000000000000000000000000
		<hr/> $+ \quad \quad \quad \times$ <hr/>
	0   <b>1</b> 00000000	10.1101000000000000000000000000000

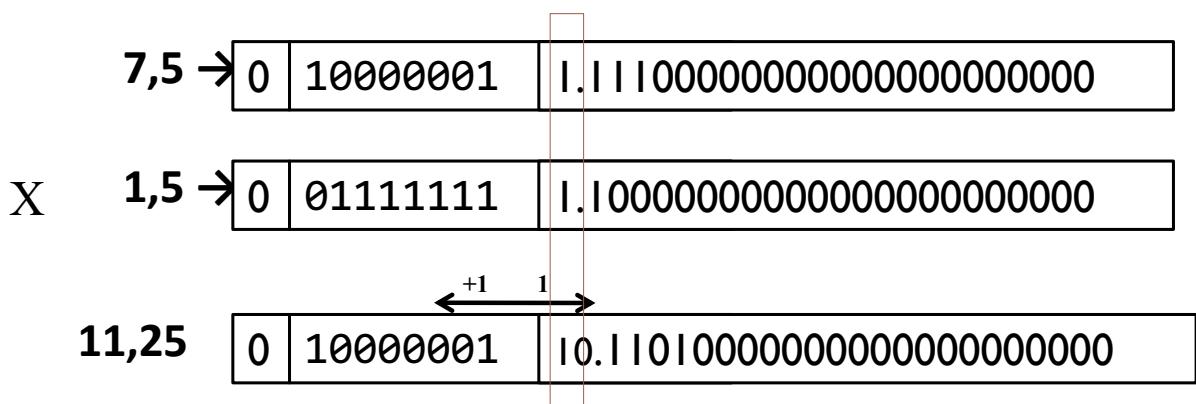
## Solución

- ▶ Multiplicar: quitar el sesgo al exponente (hay dos)

$7,5 \rightarrow$	0   10000001	1.1110000000000000000000000000000
X $1,5 \rightarrow$	0   01111111	1.1000000000000000000000000000000
<hr/>		
	0   1000000000	10.1101000000000000000000000000000
$- 01111111$		

## Solución

- ▶ Multiplicar: normalizar el resultado



## Solución

- ▶ Resultado normalizado...

$7,5 \rightarrow$	0   10000001	1.1110000000000000000000000000000
X $1,5 \rightarrow$	0   01111111	1.1000000000000000000000000000000
$11,25$	0   10000010	1.0110100000000000000000000000000

## Solución

- ▶ Se almacena el resultado eliminando el bit implícito

**11,25    0 10000010 01101000000000000000000000000000**

# Evolución de IEEE 754

- ▶ 1985 – IEEE 754
- ▶ 2008 – IEEE 754-2008 (754+854)
- ▶ 2011 – ISO/IEC/IEEE 60559:2011 (754-2008)

Name	Common name	Base	Digits	E min	E max	Notes	Decimal digits	Decimal E max
binary16	Half precision	2	10+1	-14	+15	storage, not basic	3.31	4.51
binary32	Single precision	2	23+1	-126	+127		7.22	38.23
binary64	Double precision	2	52+1	-1022	+1023		15.95	307.95
binary128	Quadruple precision	2	112+1	-16382	+16383		34.02	4931.77
decimal32		10	7	-95	+96	storage, not basic	7	96
decimal64		10	16	-383	+384		16	384
decimal128		10	34	-6143	+6144		34	6144

## Asociatividad

- ▶ **La coma flotante no es asociativa**
  - ▶  $x = -1.5 \times 10^{38}, y = 1.5 \times 10^{38}, y z = 1.0$
  - ▶ 
$$\begin{aligned}x + (y + z) &= -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) \\&= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = 0.0\end{aligned}$$
  - ▶ 
$$\begin{aligned}(x + y) + z &= (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 \\&= (0.0) + 1.0 = 1.0\end{aligned}$$
- ▶ **Las operaciones coma flotante no son asociativas**
  - ▶ Los resultados son aproximados
  - ▶  $1.5 \times 10^{38}$  es mucho más grande que 1.0
  - ▶  $1.5 \times 10^{38} + 1.0$  en la representación en coma flotante sigue siendo  $1.5 \times 10^{38}$

## Conversión int → float → int

```
if (i == (int)((float) i)) {  
    printf("true");  
}
```

- ▶ **No siempre es cierto**
- ▶ Muchos valores enteros grandes no tienen una representación exacta en coma flotante
- ▶ ¿Qué ocurre con double?

## Ejemplo

- ▶ El número 133000405 en binario es:
  - ▶ 11111011010110110011010101 (27 bits)
- ▶  $11111011010110110011010101 \times 2^0$
- ▶ Se normaliza
  - ▶  $1,1111011010110110011010101 \times 2^{26}$
  - ▶  $S = 0$  (positivo)
  - ▶  $e = 26 \rightarrow E = 26 + 127 = 153$
  - ▶  $M = 1111011010110110011010$  (se pierden los 3 últimos bits)
- ▶ El número realmente almacenado es
  - ▶  $1,1111011010110110011010 \times 2^{26} =$
  - ▶  $11111011010110110011010 \times 2^3 = 133000400$

## Conversión float → int → float

```
if (f == (float)((int) f)) {  
    printf("true");  
}
```

- ▶ **No siempre es cierto**
- ▶ **Los números con decimales no tienen representación entera**

Grupo ARCOS

**uc3m | Universidad Carlos III de Madrid**

## Tema 2 (I) Representación de la información

Estructura de Computadores  
Grado en Ingeniería Informática



# Contenidos

## 1. Introducción

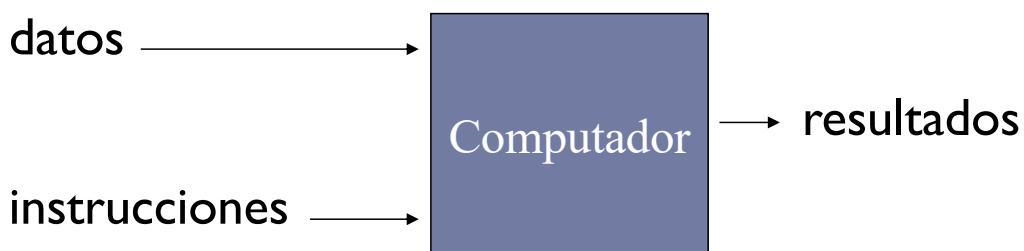
1. Motivación y objetivos
2. Sistemas posicionales

## 2. Representaciones

1. Alfanuméricas: caracteres y cadenas
2. Numéricas: naturales y enteras
3. Numéricas: coma fija
4. Numéricas: coma flotante: estándar IEEE 754

## Introducción: computador

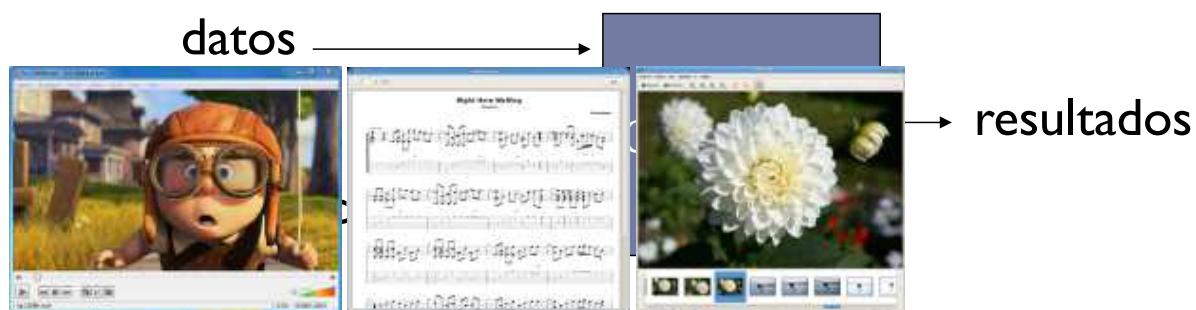
- ▶ Un computador es una máquina destinada a procesar datos.



- ▶ Se aplican unas instrucciones y se obtiene unos resultados

# Introducción: computador

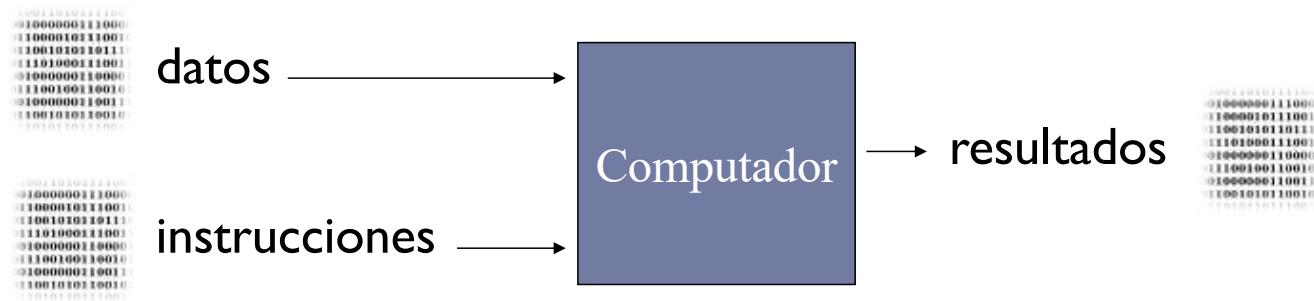
- ▶ Un computador es una máquina destinada a procesar datos.



- ▶ Se aplican unas instrucciones y se obtiene unos resultados
- ▶ Los datos/información pueden ser de **distintos tipo**

# Introducción: computador

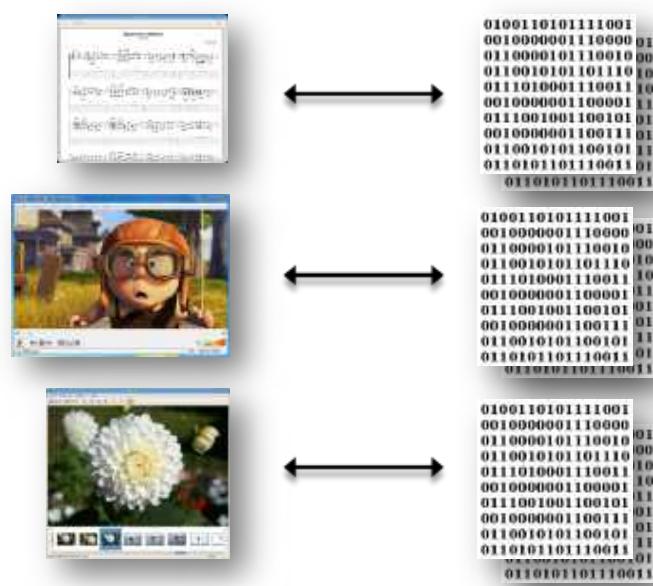
- ▶ Un computador es una máquina destinada a procesar datos.



- ▶ Se aplican unas instrucciones y se obtiene unos resultados
  - ▶ Los datos/información pueden ser de **distintos tipo**
  - ▶ Un computador solo usa una representación: **binario.**

# Introducción: representación de la información

- ▶ El uso de una **representación** permite transformar los distintos tipos de información en binario (y viceversa)



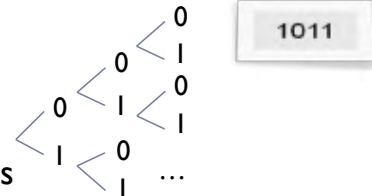
Necesitaremos...

- ▶ Conocer **posibles representaciones**:



# Introducción: características de la información

- ▶ Un computador maneja un conjunto finito de valores
  - ▶ Tipo binario (dos estados)
  - ▶ Finito (representación acotada)
    - ▶ N° de bits de palabra del computador
    - ▶ Con  $n$  bits se pueden codificar  $2^n$  valores distintos
- ▶ Hay algunos tipos de información que son infinitos
  - ▶ Imposible representar todos los valores de los números naturales, reales, etc.
- ▶ La representación elegida tiene limitaciones



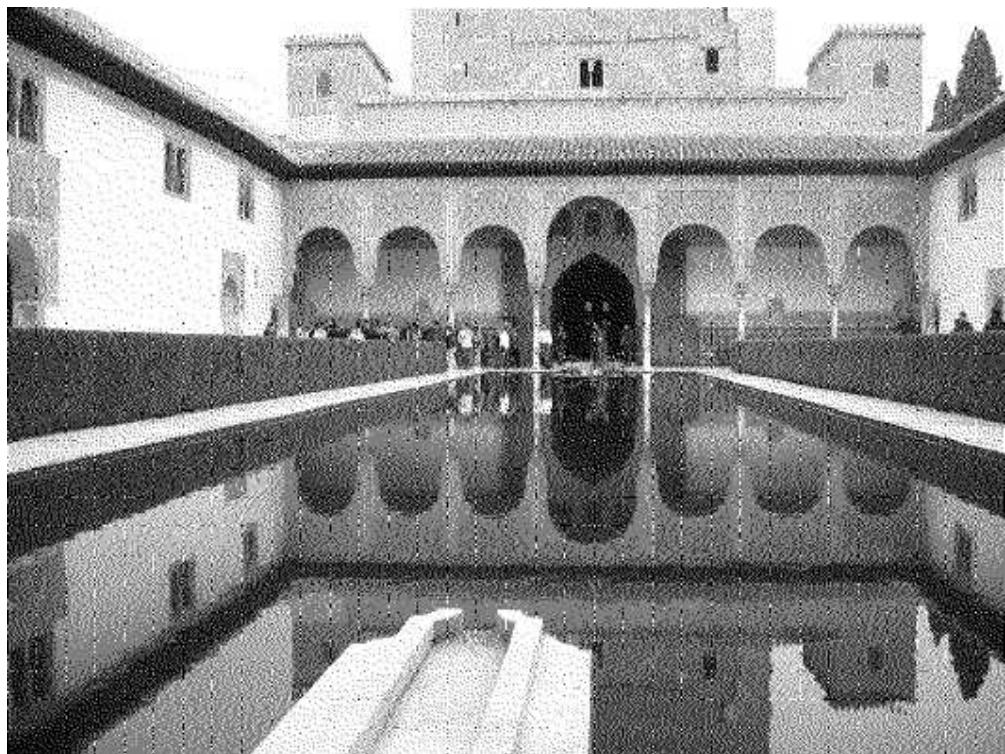
## Ejemplo 1: la calculadora de Google con 15 dígitos...

The screenshot shows a Google search results page. At the top, there is a navigation bar with links for 'La Web', 'Imágenes', 'Maps', 'Noticias', 'Video', 'Gmail', 'Más', and 'Acceder'. Below the navigation bar is the Google logo and a search bar containing the query '3999999999999 - 3999999999998'. To the right of the search bar are buttons for 'Buscar' (Search) and 'Búsqueda avanzada' (Advanced search). Below the search bar, there is a link for 'Preferencias' (Preferences) and a section for 'Buscar en:' with options for 'la Web', 'páginas en español', and 'páginas de España'. The main content area is titled 'La Web' and displays a result from a calculator. The result is shown in a large black font: '399 999 999 999 999 - 399 999 999 999 998 = 0'. Below this result is a link 'Más sobre la calculadora.' (More about the calculator.). Further down, there is a link 'Buscar documentos que contengan los términos [399999999999999](#) - [399999999999998](#)' (Search for documents containing the terms 399999999999999 - 399999999999998). At the bottom of the page, there is a footer with links for 'Google Página principal de Google - Programas de publicidad - Soluciones Empresariales - Privacidad - Todo acerca de Google'.

<http://www.20minutos.es/noticia/415383/0/google/restar/error/>

## Ejemplo 2: la profundidad de color...

1 bit	2 colores
4 bits	16 colores
8 bits	256 colores



<http://platea.pntic.mec.es/~lgonzale/tic/imagen/conceptos.html>

## Ejemplo 2: la profundidad de color...

1 bit	2 colores
4 bits	16 colores
8 bits	256 colores



<http://platea.pntic.mec.es/~lgonzale/tic/imagen/conceptos.html>

## Ejemplo 2: la profundidad de color...

1 bit	2 colores
4 bits	16 colores
8 bits	256 colores



<http://platea.pntic.mec.es/~lgonzale/tic/imagen/conceptos.html>

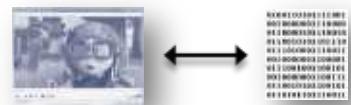
## Necesitaremos...

- ▶ Conocer **posibles representaciones**:



- ▶ Conocer las **características** de las mismas:

- ▶ Limitaciones



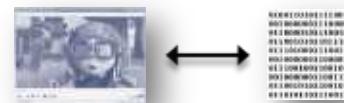
## Necesitaremos...

- ▶ Conocer **posibles representaciones**:



- ▶ Conocer las **características** de las mismas:

- ▶ Limitaciones



- ▶ Conocer **cómo operar** con la representación:



# Contenidos

## 1. Introducción

1. Motivación y objetivos
2. Sistemas posicionales

## 2. Representaciones

1. Alfanuméricas: caracteres y cadenas
2. Numéricas: naturales y enteras
3. Numéricas: coma fija
4. Numéricas: coma flotante: estándar IEEE 754

# Sistemas de representación posicionales

- ▶ Un número se define por una **cadena de dígitos**, estando **afectado** cada uno de ellos por un **factor de escala** que **depende de la posición** que ocupa en la cadena.
- ▶ Dada una base de numeración  $b$ , un número  $X$  se define como la cadena de dígitos:  
$$X = (\dots x_2 \ x_1 \ x_0, \ x_{-1} \ x_{-2} \ \dots)_b \quad \text{Con } 0 \leq x_i < b$$
con una lista de pesos asociados:  
$$P = (\dots b^2 \ b^1 \ b^0 \ b^{-1} \ b^{-2} \ \dots)_b$$

# Sistemas de representación posicionales

- ▶ Un número se define por una **cadena de dígitos**, estando **afectado** cada uno de ellos por un **factor de escala** que **depende de la posición** que ocupa en la cadena.
- ▶ Dada una base de numeración  $b$ , un número  $X$  se define como la cadena de dígitos:  
$$X = (\dots x_2 \ x_1 \ x_0, \ x_{-1} \ x_{-2} \ \dots)_b \quad \text{Con } 0 \leq x_i < b$$
 con una lista de pesos asociados:  
$$P = (\dots b^2 \ b^1 \ b^0 \ b^{-1} \ b^{-2} \ \dots)_b$$
- ▶ Su valor es:

$$V(X) = \sum_{i=-\infty}^{+\infty} b^i \cdot x_i = \dots b^2 \cdot x_2 + b^1 \cdot x_1 + b^0 \cdot x_0 + b^{-1} \cdot x_{-1} + b^{-2} \cdot x_{-2} \dots$$


# Sistemas de representación posicionales

## ► Decimal

$$X = \begin{array}{cccc|c} & 9 & 7 & 3 & 1 \\ & \dots & 10^3 & 10^2 & 10^1 & 10^0 \end{array}$$

## ► Binario

$$X = \begin{array}{cccc|c} & 0 & 1 & 0 & 1 \\ & \dots & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

## ► Hexadecimal

$$X = \begin{array}{cccc|c} & 1 & F & A & 8 \\ & \dots & 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$

# Sistemas de representación posicionales

## ► Decimal

$$X = \begin{array}{cccc} 9 & 7 & 3 & 1 \\ \dots & 10^3 & 10^2 & 10^1 & 10^0 \end{array}$$

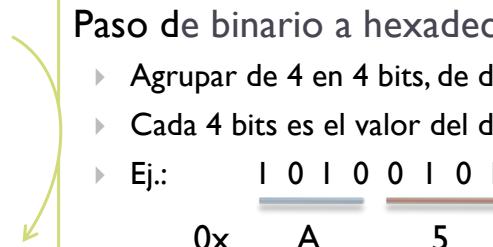
## ► Binario

$$X = \begin{array}{cccc} 0 & 1 & 0 & 1 \\ \dots & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

## ► Hexadecimal

$$X = \begin{array}{cccc} 1 & F & A & 8 \\ \dots & 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$

### Paso de binario a hexadecimal:

- Agrupar de 4 en 4 bits, de derecha a izquierda
- Cada 4 bits es el valor del dígito hexadecimal
- Ej.: 

# Sistemas de representación posicionales

## ► Decimal

$$X = \begin{array}{cccc} 9 & 7 & 3 & 1 \\ \dots & 10^3 & 10^2 & 10^1 & 10^0 \end{array}$$


¿?

## ► Binario

$$X = \begin{array}{cccc} 0 & 1 & 0 & 1 \\ \dots & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

## ► Hexadecimal

$$X = \begin{array}{ccccc} 1 & F & A & 8 \\ \dots & 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$

## Ejercicio

- ▶ Representar 342 en binario:

256	128	64	32	16	8	4	2	1
?	?	?	?	?	?	?	?	?

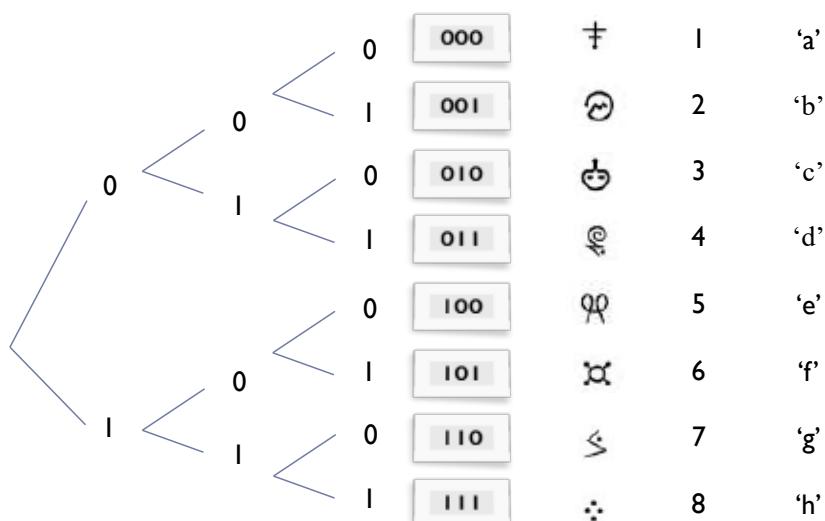
## Ejercicio (solución)

- ▶ Representar 342 en binario:

256	128	64	32	16	8	4	2	1
	0		0		0			0

## Ejemplo

- Con 3 dígitos binarios, se pueden representar 8 símbolos:



# Sistemas de representación posicionales

- ▶ **¿Cuántos valores se pueden representar con  $n$  bits?**
- ▶ **¿Cuántos bits se necesitan para representar  $m$  ‘valores’?**
- ▶ **Con  $n$  bits,**  
si el valor mínimo representable corresponde al número 0,  
**¿Cuál es el máximo valor numérico representable?**

# Sistemas de representación posicionales

- ▶ ¿Cuántos valores se pueden representar con  $n$  bits?
  - ▶  $2^n$
  - ▶ Ej.: con 4 bits se pueden representar 16 valores
- ▶ ¿Cuántos bits se necesitan para representar  $m$  ‘valores’?
  - ▶  $\log_2(n)$  por exceso
  - ▶ Ej.: para representar 35 valores se necesitan 6 bits
- ▶ Con  $n$  bits,  
si el valor mínimo representable corresponde al número 0,  
¿Cuál es el máximo valor numérico representable?
  - ▶  $2^{n-1}$

## Ejercicio

- ▶ Calcular el valor de (23 unos):

||||| | | | | | | | | | | | | | | | | | | |<sub>2</sub>

## Ejercicio (solución)

- ▶ Calcular el valor de (23 unos):

$$X = 2^{23} \cdot 1$$

Truco:

10000000000000000000, =  $2^{23}$

$$X = 2^{23} - 1$$

## Ejemplo: operaciones

► **Sumar en binario:**

$$\begin{array}{r} 1 & 1 & 1 \\ & 1 & 0 & 1 & 0 & 0 \\ + & 1 & 1 & 1 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 & 1 & 0 \end{array}$$

## Ejemplo: operaciones

► Sumar en binario:

$$\begin{array}{r} 1 & 1 & 1 \\ & 1 & 0 & 1 & 0 & 0 \\ + & 1 & 1 & 1 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 & 1 & 0 \end{array}$$

► Restar en binario:

$$\begin{array}{r} 1 \rightarrow 1 \rightarrow \\ 0 & 1 & 1 & 0 & 0 \\ - & 0 & 1 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 \end{array}$$

# Contenidos

## 1. Introducción

- 1. Motivación y objetivos
- 2. Sistemas posicionales

## 2. Representaciones

- 1. Alfanuméricas: caracteres y cadenas
- 2. Numéricas: naturales y enteras
- 3. Numéricas: coma fija
- 4. Numéricas: coma flotante: estándar IEEE 754

# Representación alfanumérica

- ▶ Cada carácter se codifica con un byte.
- ▶ Para  $n$  bits  $\Rightarrow 2^n$  caracteres representables:

# bits	# caracteres	Incluye...	Ejemplo
6	64	<ul style="list-style-type: none"><li>• 26 letras: a...z</li><li>• 10 números: 0...9</li><li>• Puntuación: .,;:....</li><li>• Especiales: + - [ ...</li></ul>	<b>BCDIC</b>
7	128	<ul style="list-style-type: none"><li>• añade mayúsculas y caracteres de control</li></ul>	<b>ASCII</b>
8	256	<ul style="list-style-type: none"><li>• añade letras acentuadas, ñ, caracteres semigráficos</li></ul>	<b>EBCDIC</b> <b>ASCII extendido</b>
16	34.168	<ul style="list-style-type: none"><li>• Añade distintos idiomas (chino, árabe,...)</li></ul>	<b>UNICODE</b>

# Ejemplo: tabla ASCII (7 bits)

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	À	097	à
002	●	STX	034	"	066	À	098	à
003	♥	ETX	035	#	067	Ç	099	ç
004	♦	EOT	036	\$	068	Ð	100	ð
005	♣	ENQ	037	%	069	È	101	è
006	♠	ACK	038	&	070	Ð	102	ñ
007	(beep)	BEL	039	,	071	Ð	103	g
008	■	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	:	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	▬	DLE	048	0	080	P	112	p
017	▬	DC1	049	1	081	Q	113	q
018	▬	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	π	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↑	ETB	055	7	087	W	119	w
024	↖	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	^	092	\	124	,
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	∨	094	^	126	~
031	(cursor down)	US	063	?	095	-	127	▷

Copyright 1998, JimPrice.Com Copyright 1992, Loading Edge Computer Products, Inc.

# Ejemplo: tabla ASCII (7 bits) caracteres de control

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	À	097	à
002	●	STX	034	"	066	À	098	à
003	♥	ETX	035	#	067	À	099	à
004	♦	EOT	036	\$	068	À	100	à
005	♣	ENQ	037	%	069	À	101	à
006	♠	ACK	038	&	070	À	102	à
007	(beep)	BEL	039	,	071	À	103	à
008	■	BS	040	(	072	À	104	à
009	(tab)	HT	041	)	073	À	105	à
010	(line feed)	LF	042	*	074	À	106	à
011	(home)	VT	043	+	075	À	107	à
012	(form feed)	FF	044	,	076	À	108	à
013	(carriage return)	CR	045	-	077	À	109	à
014	♫	SO	046	.	078	À	110	à
015	☼	SI	047	/	079	À	111	à
016	▬	DLE	048	0	080	À	112	à
017	▬	DC1	049	1	081	À	113	à
018	▬	DC2	050	2	082	À	114	à
019	▬	DC3	051	3	083	À	115	à
020	▬	DC4	052	4	084	À	116	à
021	▬	NAK	053	5	085	À	117	à
022	▬	SYN	054	6	086	À	118	à
023	↑	ETB	055	7	087	À	119	à
024	↖	CAN	056	8	088	À	120	à
025	↓	EM	057	9	089	À	121	à
026	→	SUB	058	:	090	À	122	à
027	←	ESC	059	;	091	À	123	à
028	(cursor right)	FS	060	^	092	À	124	à
029	(cursor left)	GS	061	=	093	À	125	à
030	(cursor up)	RS	062	∨	094	À	126	à
031	(cursor down)	US	063	?	095	À	127	à

Copyright 1998, JimPrice.Com Copyright 1992, Leading Edge Computer Products, Inc.

< 32

## Ejemplo: tabla ASCII (7 bits) distancia mayúsculas-minúsculas

97-65=32

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	⌚	SOH	033	!	065	À	097	à
002	●	STX	034	"	066	À	098	à
003	♥	ETX	035	#	067	Ç	099	ç
004	♦	EOT	036	\$	068	Ð	100	ð
005	♣	ENQ	037	%	069	È	101	è
006	♠	ACK	038	&	070	Ð	102	ñ
007	(beep)	BEL	039	,	071	Ð	103	g
008	■	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	:	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	▬	DLE	048	0	080	P	112	p
017	▬	DC1	049	1	081	Q	113	q
018	▬	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	π	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↑	ETB	055	7	087	W	119	w
024	◀	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	^	092	\	124	,
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	∨	094	^	126	~
031	(cursor down)	US	063	?	095	-	127	▷

Copyright 1998, JimPrice.Com Copyright 1992, Leading Edge Computer Products, Inc.

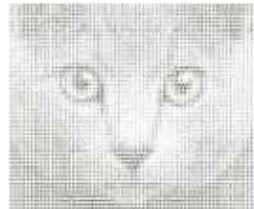
## Ejemplo: tabla ASCII (7 bits) conversión de un número a carácter

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	⌚	SOH	033	!	065	À	097	à
002	⌚	STX	034	"	066	À	098	à
003	♥	ETX	035	#	067	À	099	à
004	♦	EOT	036	\$	068	À	100	à
005	♣	ENQ	037	%	069	À	101	à
006	♠	ACK	038	&	070	À	102	à
007	(beep)	BEL	039	'	071	À	103	à
008	■	BS	040	(	072	À	104	à
009	(tab)	HT	041	)	073	À	105	à
010	(line feed)	LF	042	*	074	À	106	à
011	(home)	VT	043	+	075	À	107	à
012	(form feed)	FF	044	,	076	À	108	à
013	(carriage return)	CR	045	-	077	À	109	à
014	♫	SO	046	.	078	À	110	à
015	☼	SI	047	/	079	À	111	à
016	▬	DLE	048	0	080	À	112	à
017	▬	DC1	049	1	081	À	113	à
018	▬	DC2	050	2	082	À	114	à
019	!!	DC3	051	3	083	À	115	à
020	π	DC4	052	4	084	À	116	à
021	§	NAK	053	5	085	À	117	à
022	▬	SYN	054	6	086	À	118	à
023	↑	ETB	055	7	087	À	119	à
024	↖	CAN	056	8	088	À	120	à
025	↓	EM	057	9	089	À	121	à
026	→	SUB	058	:	090	À	122	à
027	←	ESC	059	:	091	À	123	à
028	(cursor right)	FS	060	^	092	À	124	à
029	(cursor left)	GS	061	=	093	À	125	à
030	(cursor up)	RS	062	∨	094	À	126	à
031	(cursor down)	US	063	?	095	À	127	à

Copyright 1998, JimPrice.Com Copyright 1992, Leading Edge Computer Products, Inc.

$$6+48=54$$

# Curiosidad: Visualización ‘gráfica’ con caracteres



<http://www.typorganism.com/asciiomatic/>

# Cadenas de caracteres

1. Cadenas de longitud fija:



2. Cadenas de longitud variable con separador:



3. Cadenas de longitud variable con longitud en cabecera:



# Contenidos

## 1. Introducción

1. Objetivo
2. Motivación
3. Sistemas posicionales

## 2. Representaciones

1. Alfanuméricas: caracteres y cadenas
2. **Numéricas: naturales y enteras**
3. Numéricas: coma fija
4. Numéricas: coma flotante: estándar IEEE 754

# Representación numérica

- ▶ Clasificación de números reales:
  - ▶ Naturales: 0, 1, 2, 3, ...
  - ▶ Enteros: ... -3, -2, -1, 0, 1, 2, 3, ....
  - ▶ Racionales: fracciones ( $5/2 = 2,5$ )
  - ▶ Irracionales:  $2^{1/2}, \pi, e, \dots$
- ▶ Conjuntos infinitos y espacio de representación finito:
  - ▶ Imposible representar todos
- ▶ Características de la representación usada:
  - ▶ Elemento representado:  
Natural, entero, ...
  - ▶ Rango de representación:  
Intervalo entre el menor y mayor nº representable
  - ▶ Resolución de representación:  
Diferencia entre un nº representable y el siguiente.  
Representa el máximo error cometido. Puede ser cte. o variable.

## Sistemas de representación binarios más usados

- A. Coma fija sin signo o binario puro naturales

---

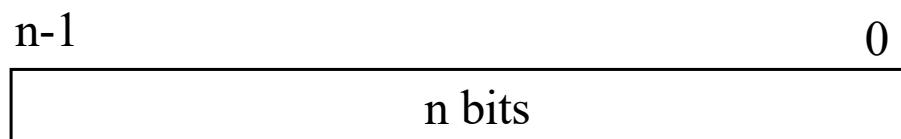
- B. Signo magnitud
- C. Complemento a uno (Ca 1) enteros
- D. Complemento a dos (Ca 2)
- E. Exceso  $2^{n-1}-1$

---

- F. Coma flotante: Estándar IEEE 754 reales

## Coma fija sin signo o binario puro [naturales]

- ▶ Sistema posicional con base 2 y sin parte fraccionaria.



$$V(X) = \sum_{i=0}^{n-1} 2^i \cdot x_i$$

- Rango de representación:  $[0, 2^n - 1]$
- Resolución: 1 unidad

## Ejemplo comparativo (3 bits)

Decimal	Binario Puro
+7	111
+6	110
+5	101
+4	100
+3	011
+2	010
+1	001
+0	000
-0	N.D.
-1	N.D.
-2	N.D.
-3	N.D.
-4	N.D.
-5	N.D.
-6	N.D.
-7	N.D.

## Coma fija con signo o signo magnitud [enteros]

- ▶ Se reserva un bit (S) para el signo ( $0 \Rightarrow +$ ;  $1 \Rightarrow -$ )

$n-1$	$n-2$	0
S	Magnitud (n-1 bits)	

$$\begin{array}{ll} \text{Si } x_{n-1} = 0 & V(X) = \sum_{i=0}^{n-2} 2^i \cdot x_i \\ \text{Si } x_{n-1} = 1 & V(X) = -\sum_{i=0}^{n-2} 2^i \cdot x_i \end{array} \quad \left| \Rightarrow V(X) = (1 - 2 \cdot x_{n-1}) \cdot \sum_{i=0}^{n-2} 2^i \cdot x_i \right.$$

- Rango de representación:  $[-2^{n-1} + 1, 2^{n-1} - 1]$
- Resolución: 1 unidad
- Ambigüedad del 0

## Ejemplo comparativo (3 bits)

Decimal	Binario Puro	Signo magnitud
+7	111	N.D.
+6	110	N.D.
+5	101	N.D.
+4	100	N.D.
+3	011	011
+2	010	010
+1	001	001
+0	000	000
-0	N.D.	100
-1	N.D.	101
-2	N.D.	110
-3	N.D.	111
-4	N.D.	N.D.
-5	N.D.	N.D.
-6	N.D.	N.D.
-7	N.D.	N.D.

## Complemento a uno (a la base menos uno) [enteros] (1 / 3)

- ▶ **Número positivo:**  
se representa en binario puro con  $n-1$  bits

			0
0	Magnitud ( $n-1$ bits)		

$$V(X) = \sum_{i=0}^{n-1} 2^i \cdot x_i = \sum_{i=0}^{n-2} 2^i \cdot x_i$$

- Rango de representación (+):  $[0, 2^{n-1} - 1]$
- Resolución: 1 unidad

## Complemento a uno (a la base menos uno) [enteros] (2/3)

### ► Número negativo:

- Se complementa a la base menos uno
- El número  $X < 0$  se representa como  $2^n - X - 1$  con n bits

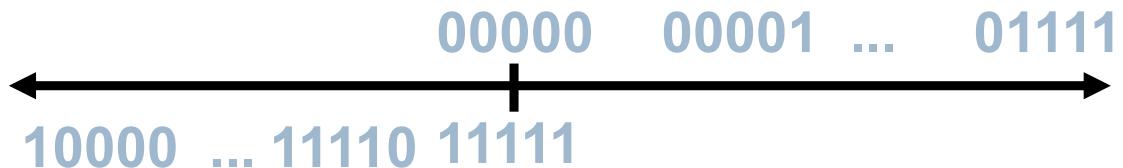
			0
n-1	n-2		
1	C a 1 de la Magnitud (n-1 bits)		

$$V(X) = -2^n + \sum_{i=0}^{n-1} 2^i \cdot y_i + 1$$

- Rango de representación (-):  $[-(2^{n-1}-1), -0]$
- Resolución: 1 unidad

## Complemento a uno

- ▶ Los números positivos tienen un 0 en el bit más significativo



- ▶ Los números negativos tienen un 1 en el bit más significativo

## Complemento a uno (a la base menos uno) [enteros] (3/3)

Truco: C a 1 (X) = X

C a 1 (-X) = cambiar los 1 por 0 y los 0 por 1

- ▶ Ejemplo: Para  $n=4 \Rightarrow$  el valor  $+3_{10} = 0011_2$
- ▶ Ejemplo: Para  $n=4 \Rightarrow$  el valor  $-3_{10} = 1100_2$ 
  - ▶ -  $\Rightarrow$  1 (bit signo y también parte de magnitud)
  - ▶ C a 1(3)  $\Rightarrow 2^4 - 0011_2 - 1 = 2^4 - 3 - 1 = 12 \Rightarrow 1100_2$

- Rango de representación:  $[-2^{n-1} + 1, 2^{n-1} - 1]$
- Resolución: 1 unidad
- El 0 tiene doble representación (+0 y -0)
- Rango simétrico

## Ejemplo comparativo (3 bits)

Decimal	Binario Puro	Signo magnitud	Complemento a uno
+7	111	N.D.	N.D.
+6	110	N.D.	N.D.
+5	101	N.D.	N.D.
+4	100	N.D.	N.D.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
-0	N.D.	100	111
-1	N.D.	101	110
-2	N.D.	110	101
-3	N.D.	111	100
-4	N.D.	N.D.	N.D.
-5	N.D.	N.D.	N.D.
-6	N.D.	N.D.	N.D.
-7	N.D.	N.D.	N.D.

## Ejemplo

- ▶ Para  $n = 5$  bits
- ▶ ¿Cómo se representa  $X = 5$ ?
  - ▶ Como es positivo, en binario puro
    - ▶ 00101
- ▶ ¿Cómo se representa  $X = -5$ ?
  - ▶ Como es negativo, se complementa el valor 5 (00101)
    - ▶ 11010
- ▶ ¿Cuál es el valor de 00111 en complemento a 2?
  - ▶ Como es positivo, su valor es directamente 7
- ▶ ¿Cuál es el valor de 11000 en complemento a 2?
  - ▶ Como es negativo, se complementa y se obtiene 00111 (7)
    - ▶ El valor es -7

## Aritmética en complemento a uno

- ▶ Sumas y restas se realizan de igual forma

- ▶ Para  $n = 5$  bits

- ▶ Sea  $X = 5$

- ▶ En complemento a uno = 00101

- ▶ Sea  $Y = 7$

- ▶ En complemento a uno = 00111

- ▶  $X + Y?$

$$\begin{array}{rcl} X & = & 00101 \\ Y & = & \underline{00111} \\ X+Y & = & 01100 \end{array}$$

- ▶ El valor de 01100 en complemento a uno es 12

## Aritmética en complemento a uno

- ▶ Para  $n = 5$  bits
- ▶ Sea  $X = -5$ 
  - ▶ En complemento a uno = complemento de 00101: 11010
- ▶ Sea  $Y = -7$ 
  - ▶ En complemento a uno = complemento de 00111: 11000
- ▶  $X + Y?$

$$-X = 11010$$

$$-Y = \underline{11000+}$$

$-(X+Y) = 110010$       Se produce un acarreo, se suma y se desprecia

$$\begin{array}{r} & & | \\ -X & & \\ -Y & & \\ \hline 10011 \end{array}$$

- ▶ El valor de 10011 en complemento a uno es el valor negativo de su complemento -01100 = - 12

¿Porqué se desprecia el acarreo y se suma al resultado?

- ▶ -X se representa como  $2^n - X - I$
- ▶ -Y se representa como  $2^n - Y - I$
- ▶ -(X + Y) se representa como  $2^n - (X+Y) - I$
  
- ▶ Cuando sumamos directamente  $-X - Y$  se obtiene
  - $-X = 2^n - X - I$
  - $-Y = \underline{2^n - Y - I}$
  - $-(X+Y) = 2^n + 2^n - (X + Y) - 2$

Se corrige el resultado sumando el acarreo ( $2^n$ ) y despreciándolo  
 $\Rightarrow 2^n - (X + Y) - I$

## Complemento a dos (complemento a la base) [enteros] (1 / 3)

- ▶ **Número positivo:**  
se representa en binario puro con  $n-1$  bits

			0
$n-1$	$n-2$		
0	Magnitud ( $n-1$ bits)		

$$V(X) = \sum_{i=0}^{n-1} 2^i \cdot x_i = \sum_{i=0}^{n-2} 2^i \cdot x_i$$

- Rango de representación (+):  $[0, 2^{n-1} - 1]$
- Resolución: 1 unidad

## Complemento a dos (complemento a la base) [enteros] (2/3)

### ► Número negativo:

- ▶ Se complementa a la base
- ▶ El número  $X < 0$  se representa como  $2^n - X$  con n bits

			0
n-1	n-2		
1	C a 2 de la Magnitud (n-1 bits)		

$$V(X) = -2^n + \sum_{i=0}^{n-1} 2^i \cdot y_i$$

- Rango de representación (-):  $[-2^{n-1}, -1]$
- Resolución: 1 unidad

## Complemento a dos (complemento a la base) [enteros] (3/3)

Truco:  $C a 2 (X) = X$   
 $C a 2 (-X) = C a 1 (X) + 1$

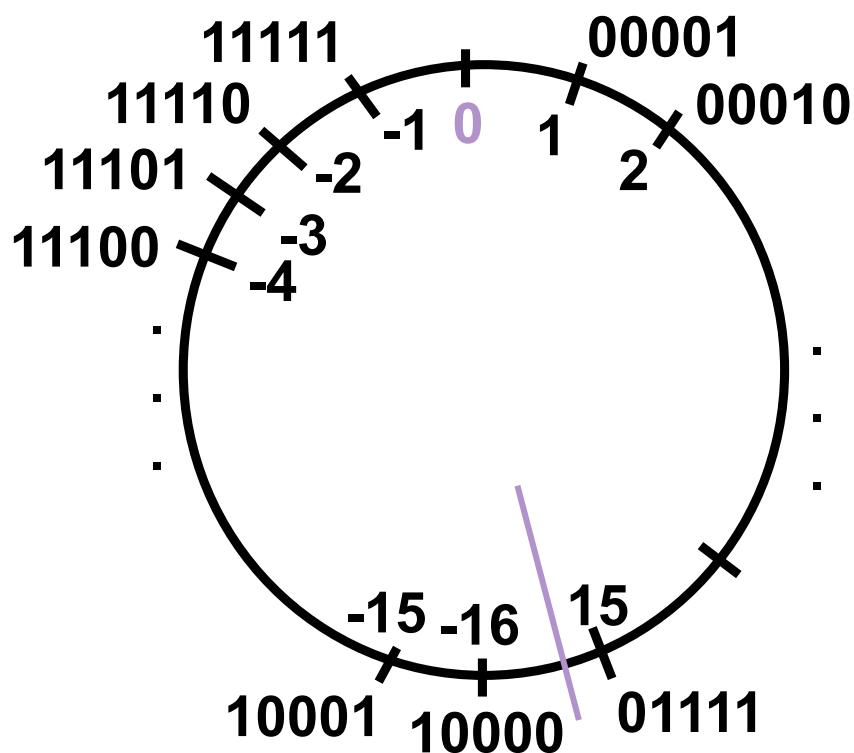
- ▶ Ejemplo: Para  $n=4 \Rightarrow +3 = 0011_2$
- ▶ Ejemplo: Para  $n=4 \Rightarrow -3 = 1101_2$ 
  - ▶  $1 \Rightarrow -$  (bit signo y también parte de magnitud)
  - ▶  $C a 2 (3) = C a 2(0011_2) = 2^4 - 3 = 13 \Rightarrow 1101_2$

- Rango de representación:  $[-2^{n-1}, 2^{n-1}-1]$
- Resolución: 1 unidad
- El 0 tiene una única representación (No  $\exists -0$ )
- Rango asimétrico

## Ejemplo comparativo (3 bits)

Decimal	Binario Puro	Signo magnitud	Complemento a uno	Complemento a dos
+7	111	N.D.	N.D.	N.D.
+6	110	N.D.	N.D.	N.D.
+5	101	N.D.	N.D.	N.D.
+4	100	N.D.	N.D.	N.D.
+3	011	011	011	011
+2	010	010	010	010
+1	001	001	001	001
+0	000	000	000	000
-0	N.D.	100	111	N.D.
-1	N.D.	101	110	111
-2	N.D.	110	101	110
-3	N.D.	111	100	101
-4	N.D.	N.D.	N.D.	100
-5	N.D.	N.D.	N.D.	N.D.
-6	N.D.	N.D.	N.D.	N.D.
-7	N.D.	N.D.	N.D.	N.D.

## Complemento a dos



$2^{N-1}$  no negativos  
 $2^{N-1}$  negativos  
Un cero

## Complemento a dos para 32 bits

$$0000 \dots 0000 \ 0000 \ 0000 \ 0000_{\text{dos}} = 0_{(10)}$$

$$0000 \dots 0000 \ 0000 \ 0000 \ 0001_{\text{dos}} = 1_{(10)}$$

$$0000 \dots 0000 \ 0000 \ 0000 \ 0010_{\text{dos}} = 2_{(10)}$$

...

$$0111 \dots 1111 \ 1111 \ 1111 \ 1101_{\text{dos}} = 2,147,483,645_{(10)}$$

$$0111 \dots 1111 \ 1111 \ 1111 \ 1110_{\text{dos}} = 2,147,483,646_{(10)}$$

$$\underline{0111 \dots 1111 \ 1111 \ 1111 \ 1111}_{\text{dos}} = \underline{2,147,483,647}_{(10)}$$

$$1000 \dots 0000 \ 0000 \ 0000 \ 0000_{\text{dos}} = -2,147,483,648_{(10)}$$

$$1000 \dots 0000 \ 0000 \ 0000 \ 0001_{\text{dos}} = -2,147,483,647_{(10)}$$

$$1000 \dots 0000 \ 0000 \ 0000 \ 0010_{\text{dos}} = -2,147,483,646_{(10)}$$

...

$$1111 \dots 1111 \ 1111 \ 1111 \ 1101_{\text{dos}} = -3_{(10)}$$

$$1111 \dots 1111 \ 1111 \ 1111 \ 1110_{\text{dos}} = -2_{(10)}$$

$$1111 \dots 1111 \ 1111 \ 1111 \ 1111_{\text{dos}} = -1_{(10)}$$

## Aritmética en complemento a dos

- ▶ Sumas y
- ▶ Para  $n = 5$  bits
- ▶ Sea  $X = 5$ 
  - ▶ En complemento a dos = 00101
- ▶ Sea  $Y = 7$ 
  - ▶ En complemento a uno = 00111
- ▶  $iX + Y?$ 
$$\begin{array}{rcl} X & = & 00101 \\ Y & = & \underline{00111}+ \\ X+Y & = & 01100 \end{array}$$
- ▶ El valor de 01100 en complemento a uno es 12
- ▶ restas de igual forma

## Aritmética en complemento a dos

- ▶ Para  $n = 5$  bits
- ▶ Sea  $X = -5$ 
  - ▶ En complemento a dos = complemento de 00101:  $11010 + 1 = 11011$
- ▶ Sea  $Y = -7$ 
  - ▶ En complemento a uno = complemento de 00111:  $11000 + 1 = 11001$
- ▶  $X + Y?$ 
  - $-X = 11011$
  - $-Y = \underline{11001+}$
  - $-(X+Y) = 110100$       Se produce un acarreo: se desprecia
- ▶ El valor es 10100. Su valor en complemento a dos = el valor negativo de su complemento a dos = complemento a uno:  $01011 + 1 = 01100 = > -12$

## Aritmética en complemento a dos

- ▶ Para  $n = 5$  bits
- ▶ Sea  $X = 8$ 
  - ▶ En complemento a dos = 01000
- ▶ Sea  $Y = 9$ 
  - ▶ En complemento a uno = 01001
- ▶  $X + Y?$

$$\begin{array}{rcl} X & = & 01000 \\ Y & = & \underline{01001} \\ X+Y & = & 10001 \end{array}$$

- ▶ Se obtiene un negativo  $\Rightarrow$  desbordamiento

## Aritmética en complemento a dos

- ▶ Para  $n = 5$  bits
- ▶ Sea  $X = -8$ 
  - ▶ En complemento a dos = complemento de 01000: 10111 + 1 = 11000
- ▶ Sea  $Y = -9$ 
  - ▶ En complemento a uno = complemento de 01001: 10110 + 1 = 10111
- ▶  $X + Y?$ 
$$\begin{array}{rcl} -X & = & 11000 \\ -Y & = & \underline{10111+} \\ -(X+Y) & = & \textcolor{red}{101111} \end{array}$$
Se produce un acarreo: se desprecia
- ▶ El valor 01111, como es positivo  $\Rightarrow$  desbordamiento

## ¿Porqué se desprecia el acarreo?

- ▶  $-X$  se representa como  $2^n - X$
- ▶  $-Y$  se representa como  $2^n - Y$
- ▶  $-(X + Y)$  se representa como  $2^n - (X+Y)$
  
- ▶ Cuando sumamos directamente  $-X - Y$  se obtiene

$$-X = 2^n - X$$

$$-Y = \underline{2^n - Y}$$

$$-(X+Y) = 2^n + 2^n - (X + Y)$$

Se corrige el resultado despreciando el acarreo

$$\Rightarrow 2^n - (X + Y)$$

## Desbordamientos en complemento a dos

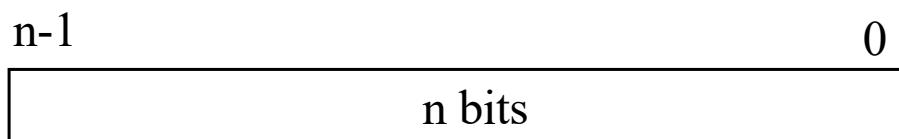
- ▶ Suma de dos negativos  $\Rightarrow$  positivo
- ▶ Suma de dos positivos  $\Rightarrow$  negativo

## Extensión de signo en complemento a dos

- ▶ ¿Cómo pasar de n bits a m bits, siendo n < m?
- ▶ Ejemplo:
  - ▶ n = 4, m = 8
  - ▶ Si  $X = 0110$  con 4 bits  $\Rightarrow X = \textcolor{blue}{0000}0110$  con 8 bits
  - ▶ Si  $X = 1011$  con 4 bits  $\Rightarrow X = \textcolor{blue}{1111}1011$  con 8 bits

## Representación en Exceso $2^{n-1}-1$ [enteros]

- ▶ El valor  $X$  con  $n$  bits se representa como  $X + 2^{n-1}-1$
- ▶ Se denomina sesgo a la cantidad  $2^{n-1}-1$



$$V(X) = \sum_{i=0}^{n-1} 2^i \cdot x_i - (2^{n-1} - 1)$$

- Rango de representación:  $[-(2^{n-1} - 1), 2^{n-1}]$
- Resolución: 1 unidad
- No existe ambigüedad con el 0

## Ejemplo comparativo (3 bits)

Decimal	Binario Puro	Signo magnitud	Complemento a uno	Complemento a dos	Exceso 3
+7	111	N.D.	N.D.	N.D.	N.D.
+6	110	N.D.	N.D.	N.D.	N.D.
+5	101	N.D.	N.D.	N.D.	N.D.
+4	100	N.D.	N.D.	N.D.	111
+3	011	011	011	011	110
+2	010	010	010	010	101
+1	001	001	001	001	100
+0	000	000	000	000	011
-0	N.D.	100	111	N.D.	N.D.
-1	N.D.	101	110	111	010
-2	N.D.	110	101	110	001
-3	N.D.	111	100	101	000
-4	N.D.	N.D.	N.D.	100	N.D.
-5	N.D.	N.D.	N.D.	N.D.	N.D.
-6	N.D.	N.D.	N.D.	N.D.	N.D.
-7	N.D.	N.D.	N.D.	N.D.	N.D.

# Ejercicio

Indique la representación de los siguientes números,  
razonando brevemente su respuesta:

1. **-32** en complemento a uno con **6 bits**
2. **-32** en complemento a dos con **6 bits**
3. **-10** en signo magnitud      con **5 bits**
4. **+14** en complemento a dos con **5 bits**

## Ejercicio (solución)

1. Con 6 bits **no es representable** en Cl:  
[- $2^{6-1}+1, \dots, -0, +0, \dots, 2^{6-1}-1]$
2. Cl + I -> **100000**
3. Signo=1, magnitud=1010 -> **11010**
4. Positivo -> Cl=C2=SM -> **01110**

## Ejercicio

► Usando 5 bits para representarlo, haga las siguientes sumas en complemento a uno:

- a)  $4 + 12$
- b)  $4 - 12$
- c)  $-4 - 12$

## Ejercicio (Solución)

- ▶ Usando 5 bits en complemento a uno:

a)  $4 + 12$

$$\begin{array}{r} 00100 \\ 01100 \\ \hline \end{array}$$

$10000 \Rightarrow$  se obtiene un negativo  $\Rightarrow -15 \Rightarrow \text{overflow}$

## Ejercicio (Solución)

- ▶ Usando 5 bits en complemento a uno:

b)  $4 - 12$

$$\begin{array}{r} 00100 \\ 10011 \\ \hline \end{array}$$

**10111**  $\Rightarrow -8$

## Ejercicio (Solución)

- ▶ Usando 5 bits en complemento a uno:

c) -4 - 12

$$\begin{array}{r} 11011 \\ 10011 \\ \hline \end{array}$$

101110  $\Rightarrow$  se obtiene un negativo  $\Rightarrow$  overflow



## **Parte IV**

### **Tema 3. Programación en ensamblador**



Grupo ARCOS

**uc3m | Universidad Carlos III de Madrid**

## Tema 3 (I)

Fundamentos de la programación en ensamblador

Estructura de Computadores

Grado en Ingeniería Informática

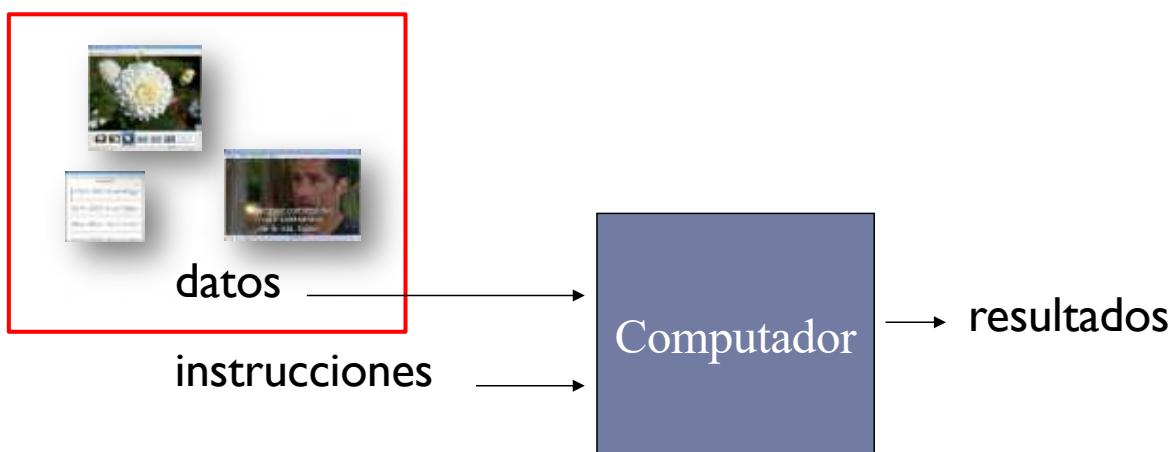


# Contenidos

1. Fundamentos básicos de la programación en ensamblador
2. Ensamblador del MIPS 32, modelo de memoria y representación de datos
3. Formato de las instrucciones y modos de direccionamiento
4. Llamadas a procedimientos y uso de la pila

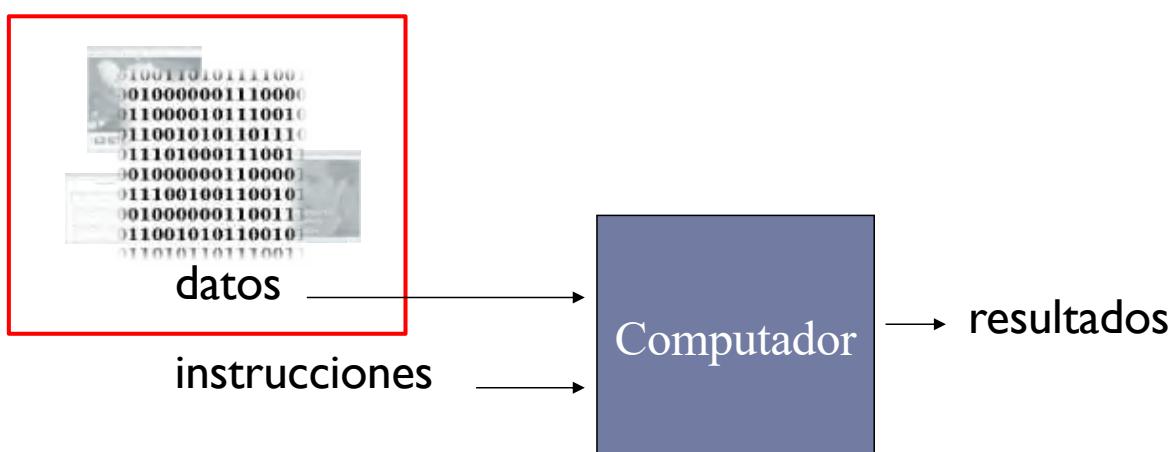
## Tipos de información: instrucciones y datos

- ▶ Representación de **datos** ...



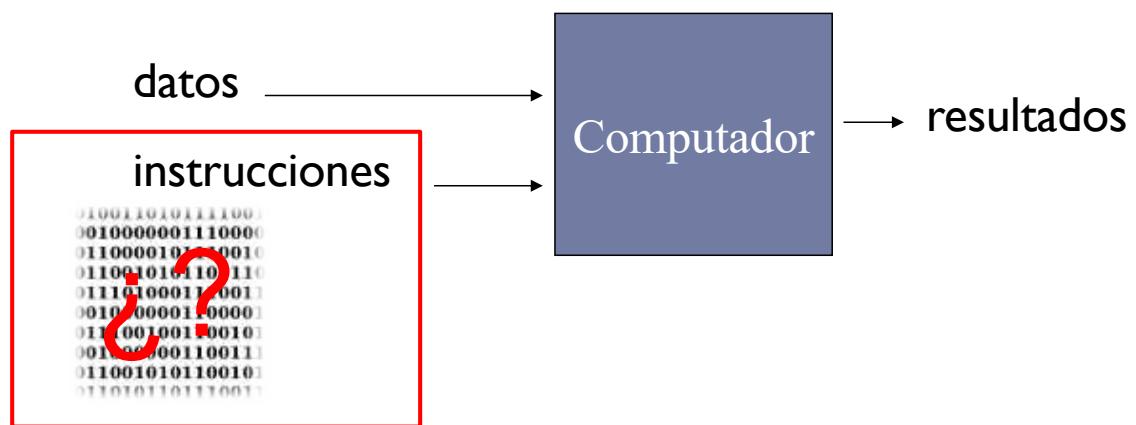
## Tipos de información: instrucciones y datos

- ▶ Representación de **datos** en **binario**.



## Tipos de información: instrucciones y datos

- ▶ ¿Qué sucede con las instrucciones?

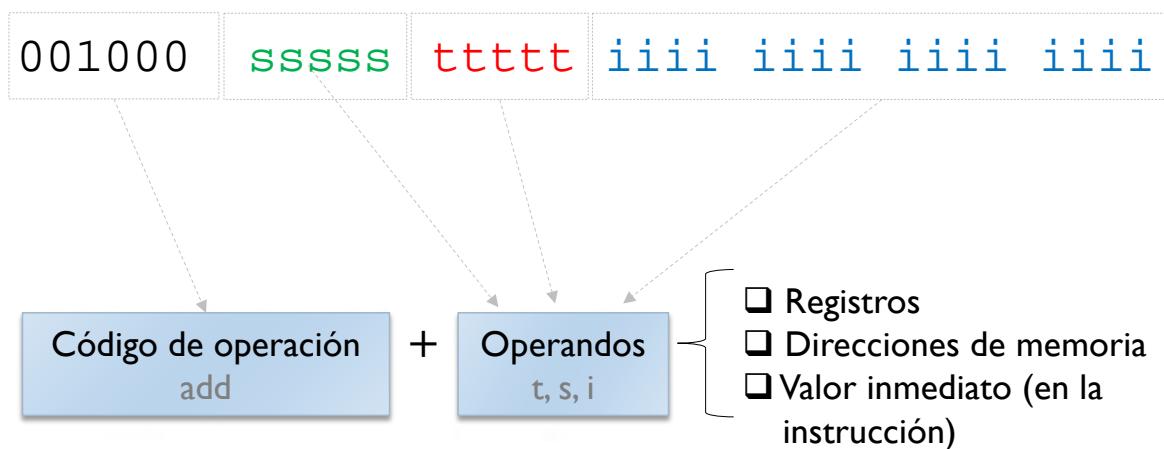


# Modelo de programación de un computador

- ▶ Un computador ofrece un modelo de programación formando por:
  - ▶ Juego de instrucciones (lenguaje ensamblador)
    - ▶ ISA: Instruction set Architecture
    - ▶ Una instrucción incluye:
      - Código de operación
      - Otros elementos: identificadores de registros, direcciones de memoria o números
  - ▶ Elementos de almacenamiento
    - ▶ Registros
    - ▶ Memoria
    - ▶ Registros de los controladores de E/S
  - ▶ Modos de ejecución

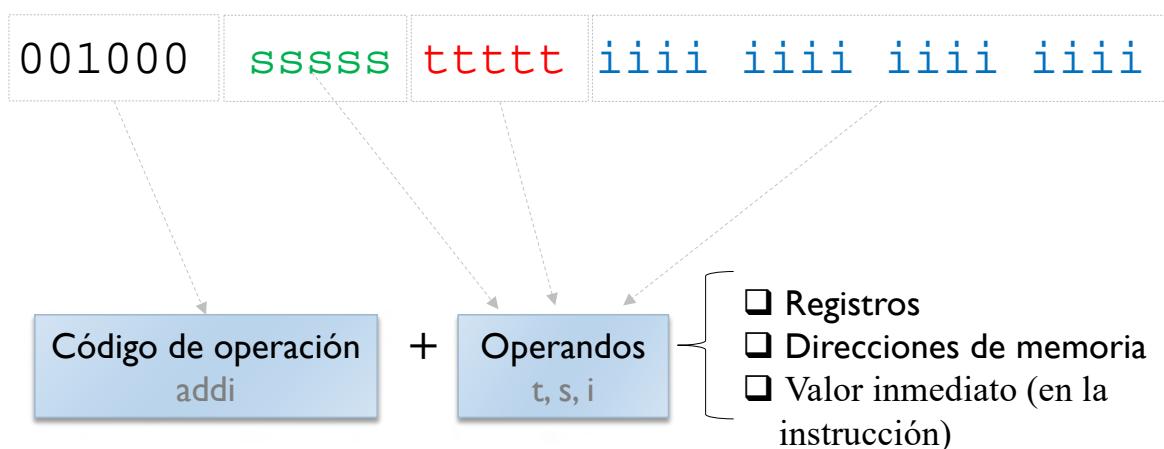
## Instrucción máquina

- ▶ Instrucción máquina: operación elemental que puede ejecutar directamente el procesador
- ▶ Ejemplo de instrucción en MIPS:
  - ▶ Suma de un registro (s) con un valor inmediato (i) y el resultado de la suma se almacena en registro (t)



## Propiedades de las instrucciones máquina

- ▶ Realizan una **única y sencilla tarea**
- ▶ Operan sobre un **número fijo de operandos**
- ▶ Incluyen toda la **información necesaria para su ejecución**



# Información incluida en una instrucción máquina

- ▶ La **operación a realizar**.
- ▶ Dónde se encuentran los **operandos**:
  - ▶ En registros
  - ▶ En memoria
  - ▶ En la propia instrucción (inmediato)
- ▶ Dónde dejar los resultados (como operando)
- ▶ Una referencia a la siguiente instrucción a ejecutar
  - ▶ De forma implícita, la siguiente instrucción
    - ▶ Un programa es una secuencia consecutiva de instrucciones máquina
  - ▶ De forma explícita en las instrucciones de bifurcación (como operando)



# Juego de instrucciones

- ▶ **Instruction Set Architecture (ISA)**
  - ▶ Conjunto de instrucciones de un procesador
  - ▶ Frontera entre el HW y el SW
- ▶ **Ejemplos:**
  - ▶ 80x86
  - ▶ MIPS
  - ▶ ARM
  - ▶ Power

# Características de un juego de instrucciones

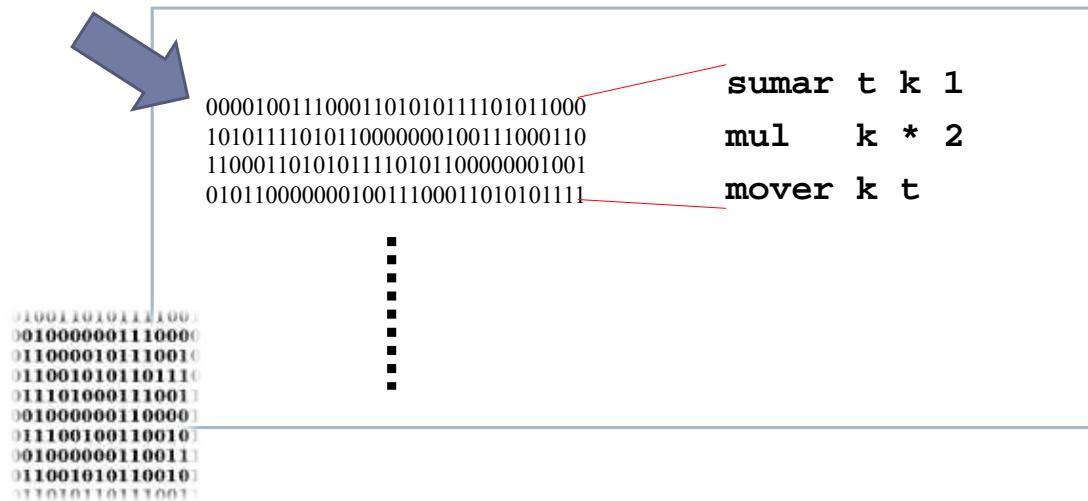
- ▶ **Operando:**
  - ▶ Registros, memoria, la propia instrucción
- ▶ **Direccionamiento de la memoria**
  - ▶ La mayoría utilizan direccionamiento por bytes
  - ▶ Ofrecen instrucciones para acceder a elementos de varios bytes a partir de una determinada posición
- ▶ **Modos de direccionamiento**
  - ▶ Especifican el lugar y la forma de acceder a los operandos (registro, memoria o la propia instrucción)
- ▶ **Tipo y tamaño de los operandos**
  - ▶ bytes: 8 bits
  - ▶ enteros: 16, 32, 64 bits
  - ▶ números en coma flotante: simple precisión, doble,...

# Características de un juego de instrucciones

- ▶ **Operaciones:**
  - ▶ Aritméticas, lógicas, de transfenrencia, control, ...
- ▶ **Instrucciones de control de flujo**
  - ▶ Saltos incondicionales
  - ▶ Saltos condicionales
  - ▶ Llamadas a procedimientos
- ▶ **Formato y codificación del juego de instrucciones**
  - ▶ Instrucciones de longitut fija o variable
    - ▶ 80x86: variable de 1 a 18 bytes
    - ▶ MIPS,ARM: fijo

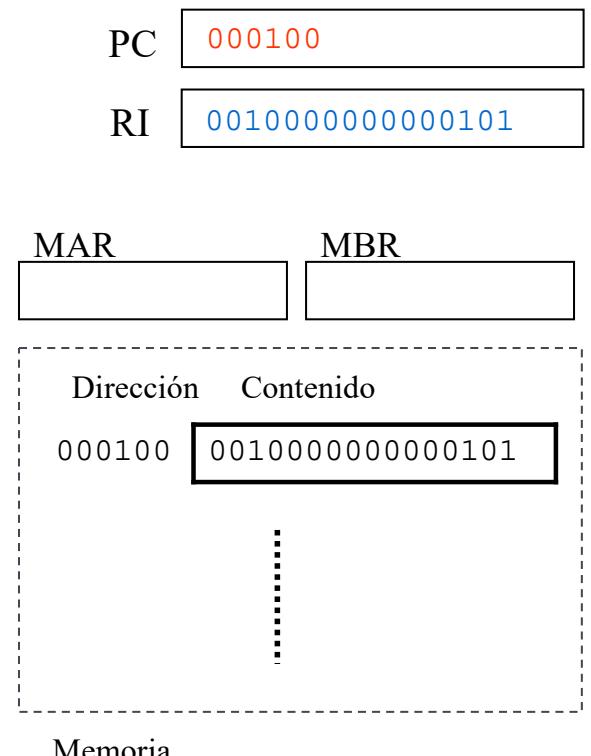
## Definición de programa

- ▶ **Programa:** lista ordenada de instrucciones máquina que se ejecutan en secuencia (por defecto).



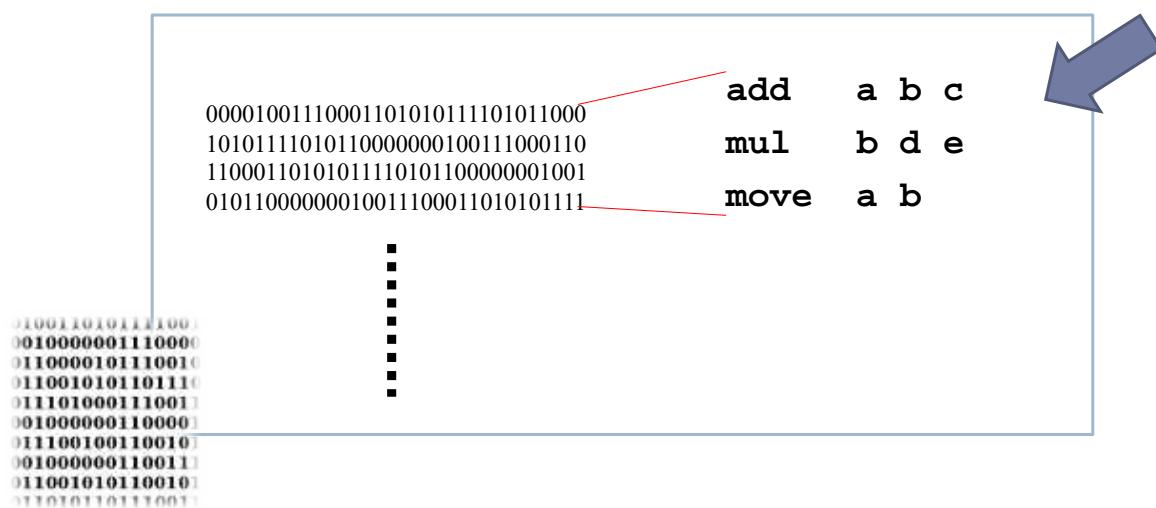
# Fases de ejecución de una instrucción

- ▶ **Lectura de la instrucción (ciclo de *fetch*)**
  - ▶  $\text{MAR} \leftarrow \text{PC}$
  - ▶ Lectura
  - ▶  $\text{MBR} \leftarrow \text{Memoria}$
  - ▶  $\text{PC} \leftarrow \text{PC} + 1$
  - ▶  $\text{RI} \leftarrow \text{MBR}$
- ▶ **Decodificación de la instrucción**
- ▶ **Ejecución de la instrucción**
- ▶ **Volver a *fetch***



## Definición de lenguaje ensamblador

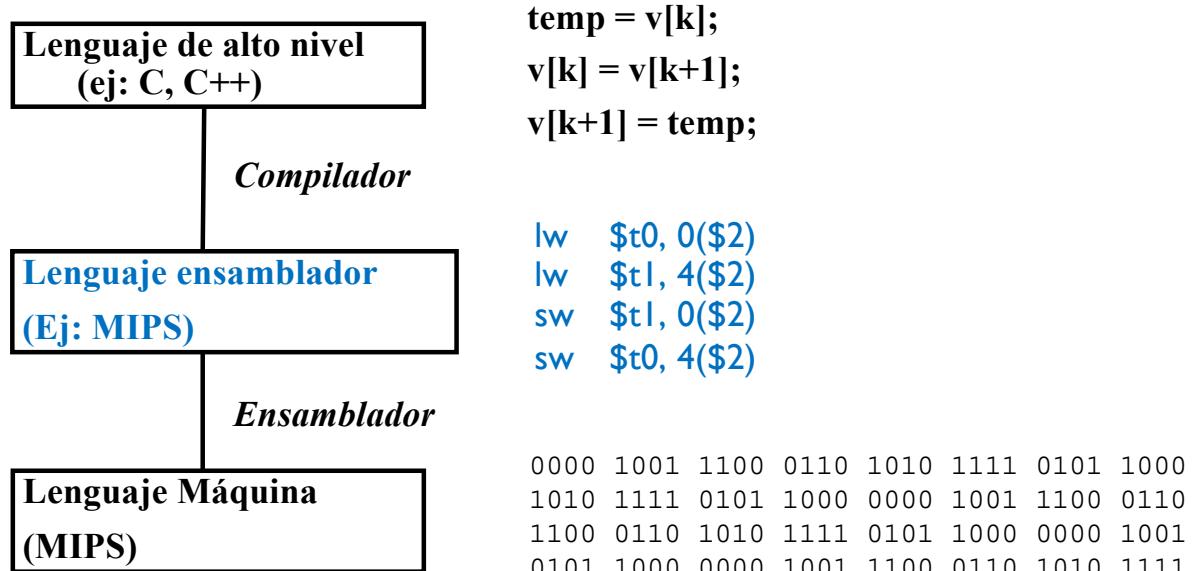
- ▶ **Lenguaje ensamblador:** lenguaje legible por un programador que constituye la representación más directa del código máquina específico de una arquitectura



## Definición de lenguaje ensamblador

- ▶ **Lenguaje ensamblador:** lenguaje legible por un programador que constituye la representación más directa del código máquina específico de una arquitectura de computadoras.
- ▶ Emplea códigos nemáticos para representar instrucciones
  - ▶ add – suma
  - ▶ lw – carga un dato de memoria
- ▶ Emplea nombres simbólicos para designar a datos y referencias
  - ▶ \$t0 – identificador de un registro
- ▶ Cada instrucción en ensamblador se corresponde con una instrucción máquina
  - ▶ add \$t1, \$t2, \$t3

# Diferentes niveles de lenguajes



# Proceso de compilación

Lenguaje de alto nivel

```
#include <stdio.h>
#define PI 3.1416
#define RADIO 20

int main ()
{
    int l;

    l=2*PI*RADIO;
    printf("long: %d\n",l);
    return (0);
}
```

Lenguaje ensamblador

```
.data
PI: .word 3.14156
RADIO: .word 20

.text
li $a0 2
la $t0 PI
lw $t0 ($t0)
la $t1 RADIO
lw $t1 ($t1)
mul $a0 $a0 $t0
mul $a0 $a0 $t1
li $v0 1
syscall
```

Lenguaje binario

```
)10011010111100]
)01000000111000(
)11000010111001(
)11001010110111(
)11101000111001]
)01000000110000]
)11100100110010]
)01000000110011]
)11001010110010]
)11010110111001]
```

# Compilación: ejemplo



## ▶ Edición de hola.c

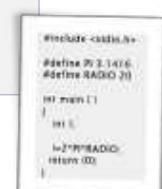
- ## ▶ gedit hola.c

```
int main ( )
{
    printf("Hola mundo...\n") ;
}
```

## ▶ Generación del programa `hola:`

- ▶ `gcc hola.c -o hola`

```
MZ®    ýÿ .      @  
€   ¤    Í!, LÍ!This program cannot be run in DOS  
mode.  
  
$      PE  L ,UŽI    ù    à  
8                      @          P  
@ ^.text    °          ` .rdata  
@ @.bss    @    0          €  À.idata
```



# Compilación: ejemplo

## ► Desensamblar hola:

► `objdump -d hola`

```
hola.exe:      formato del fichero pei-i386
Desensamblado de la sección .text:
00401000 <_WinMainCRTStartup>:
401000: 55          push    %ebp
...
40103f: c9          leave
401040: c3          ret
00401050 <_main>:
401050: 55          push    %ebp
401051: 89 e5        mov     %esp,%ebp
401053: 83 ec 08      sub    $0x8,%esp
401056: 83 e4 f0      and    $0xfffffffff0,%esp
401059: b8 00 00 00 00  mov    $0x0,%eax
40105e: 83 c0 0f      add    $0xf,%eax
401061: 83 c0 0f      add    $0xf,%eax
401064: c1 e8 04      shr    $0x4,%eax
401067: c1 e0 04      shl    $0x4,%eax
40106a: 89 45 fc      mov    %eax,%eax
40106d: 8b 45 fc      mov    %eax,0xfffffff(%ebp),%eax
401070: e8 1b 00 00 00  call   401090 <__chkstk>
401075: e8 a6 00 00 00  call   401120 <__main>
40107a: c7 04 24 00 20 40 00  movl  $0x402000,(%esp)
401081: e8 aa 00 00 00  call   401130 <_printf>
401086: c9          leave
401087: c3          ret
...
...
```



# Motivación para aprender ensamblador

```
#include <stdio.h>
#define PI 3.1416
#define RADIO 20

int main ()
{
    register int l;

    l=2*PI*RADIO;
    printf("long: %d\n",l);
    return (0);
}
```

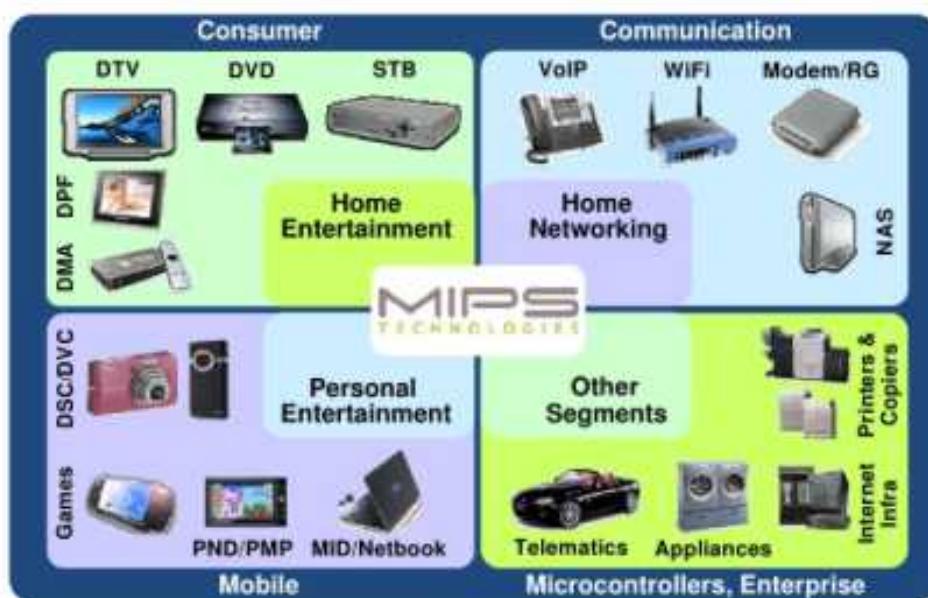
- ▶ Comprender qué ocurre cuando un computador ejecuta una sentencia de un lenguaje de alto nivel.
  - ▶ C, C++, Java, ...
- ▶ Poder determinar el impacto en tiempo de ejecución de una instrucción de alto nivel.
- ▶ Útil en dominios específicos:
  - ▶ Compiladores
  - ▶ Sistemas Operativos
  - ▶ Juegos
  - ▶ Sistemas empotrados
  - ▶ Etc.

# Objetivos

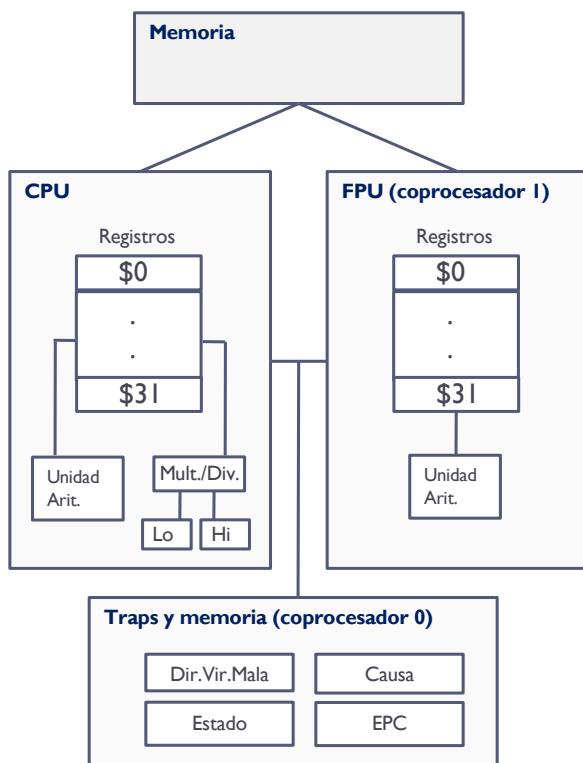
- ▶ Saber cómo se representan los elementos de un lenguaje de alto nivel en ensamblador:
  - ▶ Tipos de datos (int, char, ...)
  - ▶ Estructuras de control (if, while, ...)
- ▶ Poder escribir pequeños programas en ensamblador

```
.data  
PI: .word 3.14156  
RADIO: .word 20  
  
.text  
li $a0 2  
la $t0 PI  
lw $t0 ($t0)  
la $t1 RADIO  
lw $t1 ($t1)  
mul $a0 $a0 $t0  
mul $a0 $a0 $t1  
  
li $v0 1  
syscall
```

## Ejemplo de esamblador: MIPS 32



# Arquitectura del MIPS 32



## ▶ MIPS 32

- ▶ Procesador de 32 bits
- ▶ Tipo RISC
- ▶ CPU + coprocesadores auxiliares

## ▶ Coprocesador 0

- ▶ excepciones, interrupciones y sistema de memoria virtual

## ▶ Coprocesador I

- ▶ FPU (Unidad de Punto Flotante)

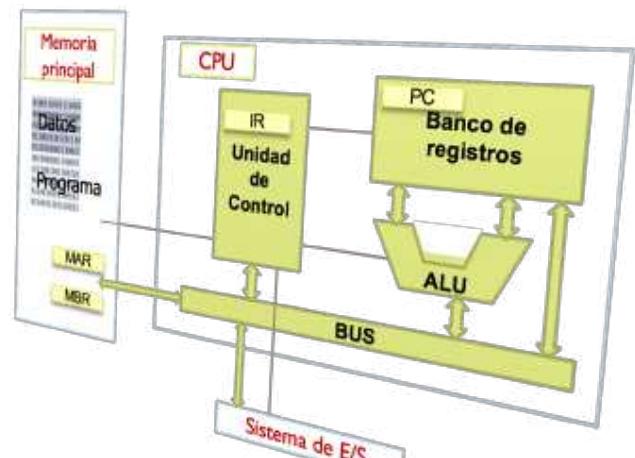
# Banco de registros (enteros)

Nombre registro	Número	Uso
zero	0	Constante 0
at	1	Reservado para el ensamblador
v0, v1	2, 3	Resultado de una rutina (o expresión)
a0, ..., a3	4, ..., 7	Argumento de entrada para rutinas
t0, ..., t7	8, ..., 15	Temporal ( <u>NO</u> se conserva entre llamadas)
s0, ..., s7	16, ..., 23	Temporal (se conserva entre llamadas)
t8, t9	24, 25	Temporal ( <u>NO</u> se conserva entre llamadas)
k0, k1	26, 27	Reservado para el sistema operativo
gp	28	Puntero al área global
sp	29	Puntero a pila
fp	30	Puntero a marco de pila
ra	31	Dirección de retorno (rutinas)

- ▶ Hay 32 registros
  - ▶ 4 bytes de tamaño (una palabra)
  - ▶ Se nombran con un \$ al principio
- ▶ Convenio de uso
  - ▶ Reservados
  - ▶ Argumentos
  - ▶ Resultados
  - ▶ Temporales
  - ▶ Punteros

## Tipo de instrucciones

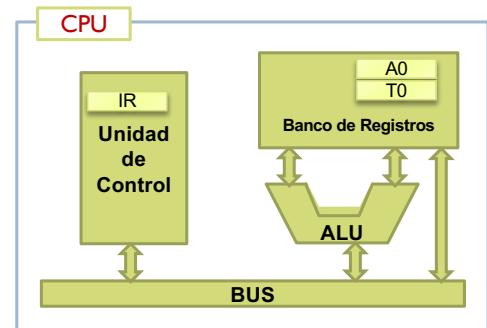
- ▶ Transferencias de datos
- ▶ Aritméticas
- ▶ Lógicas
- ▶ De desplazamiento, rotación
- ▶ De comparación
- ▶ Control de flujo (bifurcaciones, llamadas a procedimientos)
- ▶ De conversión
- ▶ De Entrada/salida
- ▶ Llamadas al sistema



# Transferencia de datos

- ▶ Copia datos:
  - ▶ entre registros
  - ▶ entre registros y memoria

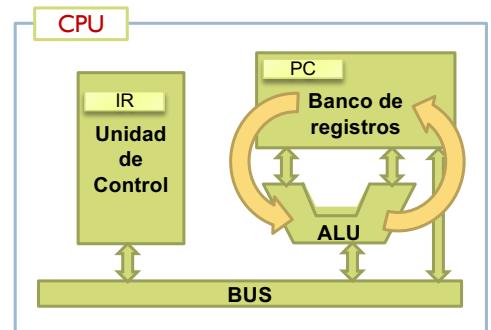
- ▶ Ejemplos:
  - ▶ Registro a registro  
`move $a0 $t0`
  - ▶ Carga inmediata  
`li $t0 5`



```
move $a0 $t0    # $a0 ← $t0
li      $t0 5    # $t0 ← 000...00101
```

# Aritméticas

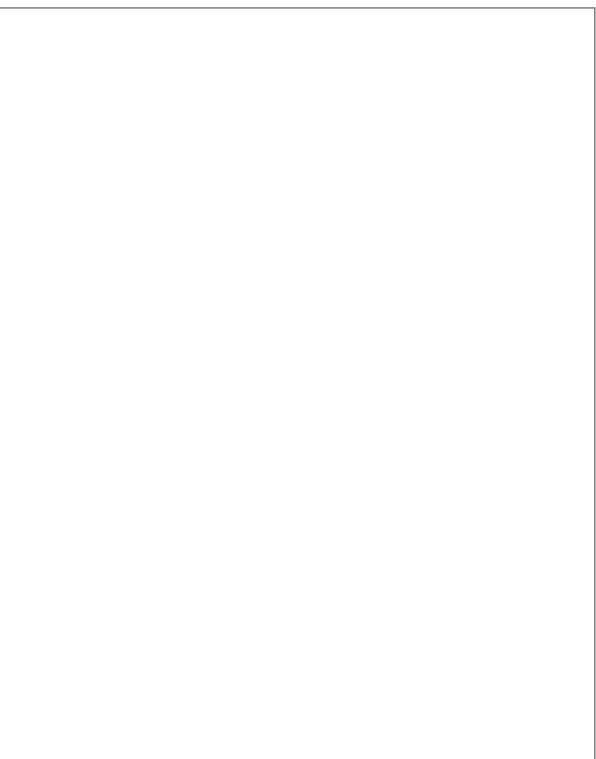
- ▶ Realiza operaciones aritméticas de enteros en la ALU o aritméticas de coma flotante (FPU)
- ▶ Ejemplos (ALU):
  - ▶ Sumar
    - `add $t0 $t1 $t2`       $\$t0 \leftarrow \$t1 + \$t2$
    - `addi $t0 $t1 5`       $\$t0 \leftarrow \$t1 + 5$
  - ▶ Restar
    - `sub $t0 $t1 $t2`
  - ▶ Multiplicar
    - `mul $t0 $t1 $t2`
  - ▶ División entera ( $5 / 2 = 2$ )
    - `div $t0 $t1 $t2`
  - ▶ Resto de la división ( $5 \% 2 = 1$ )
    - `rem $t0 $t1 $t2`       $\$t0 \leftarrow \$t1 \% \$t2$



# Ejemplo

```
int a = 5;  
int b = 7;  
int c = 8;  
int i;
```

```
i = a * (b + c)
```



# Ejemplo

```
int a = 5;  
int b = 7;  
int c = 8;  
int i;
```

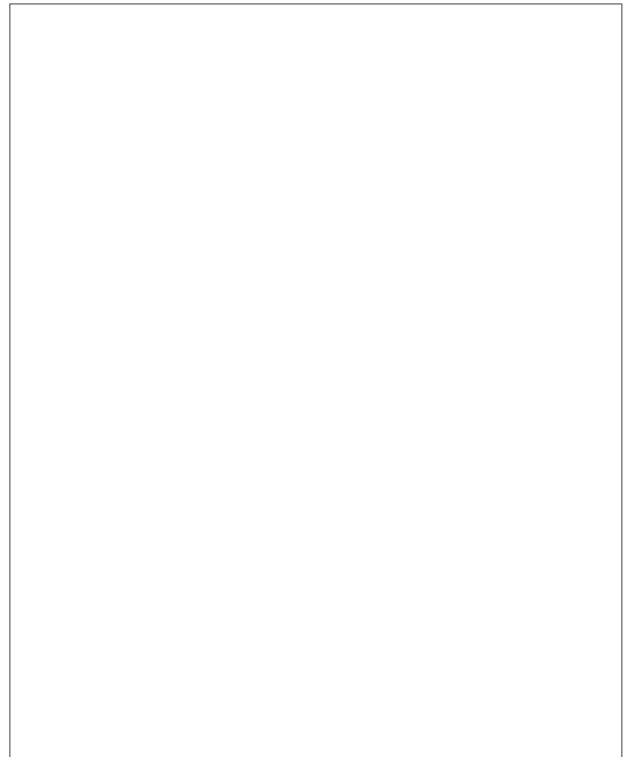
```
i = a * (b + c)
```

```
li $t1 5  
li $t2 7  
li $t3 8
```

```
add $t4 $t2 $t3  
mul $t4 $t4 $t1
```

# Ejercicio

```
int a = 5;  
int b = 7;  
int c = 8;  
int i;  
  
i = -(a * (b - 10) + c)
```



## Ejercicio (solución)

```
int a = 5;  
int b = 7;  
int c = 8;  
int i;
```

```
i = -(a * (b - 10) + c)
```

```
li $t1 5  
li $t2 7  
li $t3 8
```

```
li $t0 10  
sub $t4 $t2 $t0  
mul $t4 $t4 $t1  
add $t4 $t4 $t3  
li $t0 -1  
mul $t4 $t4 $t0
```

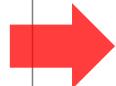
# Tipos de operaciones aritméticas

- ▶ Aritmética en **binario puro** o en **complemento a dos**
- ▶ Ejemplos:
  - ▶ Suma con signo (ca2)  
**add \$t0 \$t1 \$t2**
  - ▶ Suma inmediata con signo  
**addi \$t0 \$t1 -5**
  - ▶ Suma sin signo (binario puro)  
**addu \$t0 \$t1 \$t2**
  - ▶ Suma inmediata sin signo  
**addiu \$t0 \$t1 2**

- ▶ No **overflow**:  
**li \$t0 0x7FFFFFFF**  
**li \$t1 5**  
**addu \$t0 \$t0 \$t1**
- ▶ Con **overflow**:  
**li \$t0 0x7FFFFFFF**  
**li \$t1 1**  
**add \$t0 \$t0 \$t1**

# Ejercicio

```
li $t1 5  
li $t2 7  
li $t3 8  
  
li $t0 10  
sub $t4 $t2 $t0  
mul $t4 $t4 $t1  
add $t4 $t4 $t3  
li $t0 -1  
mul $t4 $t4 $t0
```



¿Y usando las nuevas instrucciones?

## Ejercicio (solución)

```
li $t1 5  
li $t2 7  
li $t3 8  
  
li $t0 10  
sub $t4 $t2 $t0  
mul $t4 $t4 $t1  
add $t4 $t4 $t3  
li $t0 -1  
mul $t4 $t4 $t0
```

```
li $t1 5  
li $t2 7  
li $t3 8  
  
addi $t4 $t2 -10  
mul $t4 $t4 $t1  
add $t4 $t4 $t3  
mul $t4 $t4 -1
```

# Lógicas

- ▶ Operaciones booleanas

- ▶ Ejemplos:

- ▶ AND

and \$t0 \$t1 \$t2 ( $\$t0 = \$t1 \& \$t2$ )

$$\begin{array}{r} \text{AND} \\ \hline \begin{array}{c} 1100 \\ 1010 \\ \hline 1000 \end{array} \end{array}$$

- ▶ OR

or \$t0 \$t1 \$t2 ( $\$t0 = \$t1 | \$t2$ )  
ori \$0 \$t1 80 ( $\$t0 = \$t1 | 80$ )

$$\begin{array}{r} \text{OR} \\ \hline \begin{array}{c} 1100 \\ 1010 \\ \hline 1110 \end{array} \end{array}$$

- ▶ NOT

not \$t0 \$t1 ( $\$t0 = ! \$t1$ )

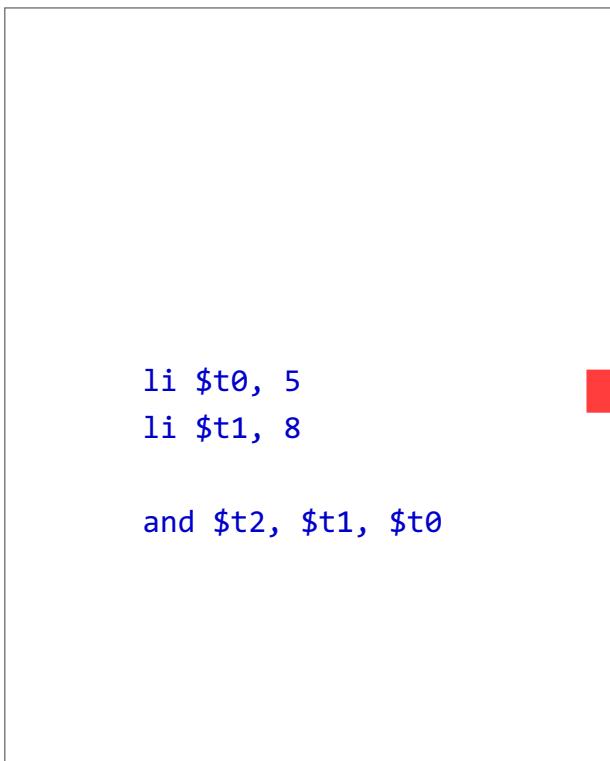
$$\begin{array}{r} \text{NOT} \\ \hline \begin{array}{c} 10 \\ 01 \end{array} \end{array}$$

- ▶ XOR

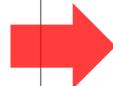
xor \$t0 \$t1 \$t2 ( $\$t0 = \$t1 ^ \$t2$ )

$$\begin{array}{r} \text{XOR} \\ \hline \begin{array}{c} 1100 \\ 1010 \\ \hline 0110 \end{array} \end{array}$$

# Ejercicio

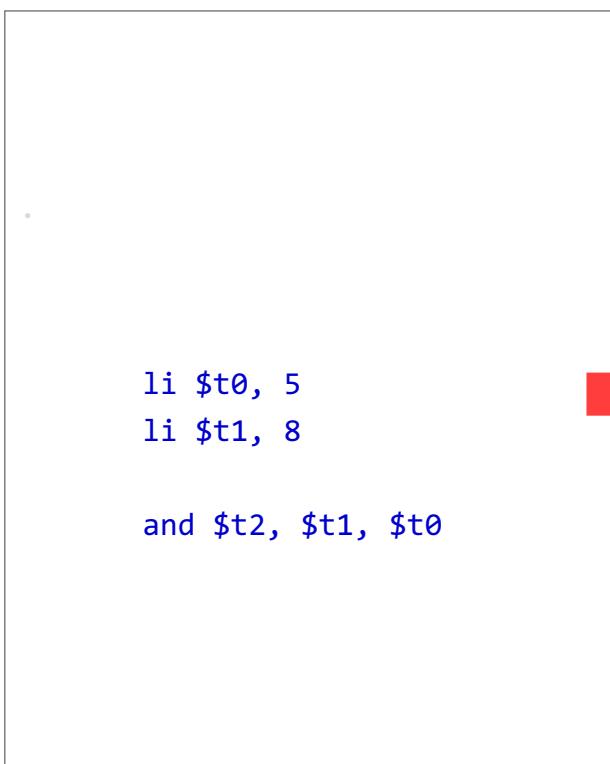


```
li $t0, 5  
li $t1, 8  
  
and $t2, $t1, $t0
```



¿Cuál será el valor  
almacenado en \$t2?

## Ejercicio (solución)



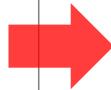
```
li $t0, 5  
li $t1, 8  
and $t2, $t1, $t0
```



00...0101	\$t0
00...1000	\$t1
<b>and 00...0000</b>	<b>\$t2</b>

# Ejercicio

```
li $t0, 5  
li $t1, 0x007FFFFF  
  
and $t2, $t1, $t0
```



¿Qué permite hacer un and con 0x007FFFFFF?

## Ejercicio (solución)

```
li $t0, 5  
li $t1, 0x007FFFFF  
  
and $t2, $t1, $t0
```

¿Qué permite hacer un and con 0x007FFFFFF?

Obtener los 23 bits menos significativos

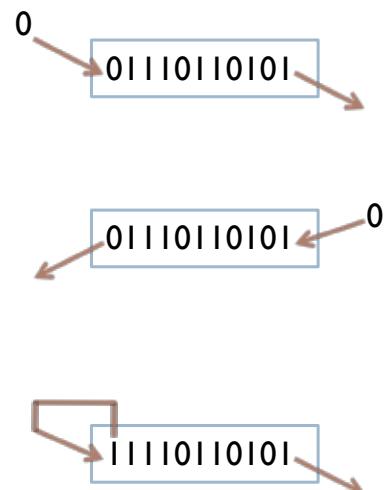
La constante usada para la selección de bits se denomina **máscara**.

# Desplazamientos

- ▶ De movimiento de **bits**

- ▶ Ejemplos:

- ▶ Desplazamiento **lógico** a la derecha  
**srl \$t0 \$t0 4**    ( $\$t0 = \$t0 \gg 4$  bits)
- ▶ Desplazamiento **lógico** a la izquierda  
**sll \$t0 \$t0 5**    ( $\$t0 = \$t0 \ll 5$  bits)
- ▶ Desplazamiento **aritmético**  
**sra \$t0 \$t0 2**    ( $\$t0 = \$t0 \gg 2$  bits)



## Ejercicio

```
li $t0, 5  
li $t1, 6  
  
sra $t0, $t1, 1
```

¿Cuál es el valor de \$t0?



## Ejercicio (solución)

```
li $t0, 5  
li $t1, 6  
  
sra $t0, $t1, 1
```

¿Cuál es el valor de \$t0?

000 .... 0110 \$t1

Se desplaza 1 bit a la derecha

000 ..... 0011 \$t0

## Ejercicio

```
li $t0, 5  
li $t1, 6  
  
sll $t0, $t1, 1
```

¿Cuál es el valor de \$t0?



## Ejercicio (solución)

```
li $t0, 5  
li $t1, 6  
  
sll $t0, $t1, 1
```

¿Cuál es el valor de \$t0?

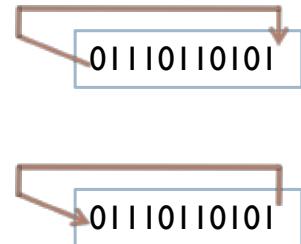
000 .... 0110 \$t1

Se desplaza 1 bit a la izquierda

000 ..... 1100 \$t0

# Rotaciones

- ▶ De movimiento de bits (2)
- ▶ Ejemplos:
  - ▶ Rotación a la izquierda  
`rol $t0 $t0 4` (\$t0 = \$t0 >> 4 bits)
  - ▶ Rotación a la derecha  
`ror $t0 $t0 5` (\$t0 = \$t0 << 5 bits)



# Ejercicio

Realice un programa que detecte el signo de un número almacenado \$t0 y deje en \$t1 un 1 si es negativo y un 0 si es positivo



## Ejercicio (solución)

Realice un programa que detecte el signo de un número almacenado \$t0 y deje en \$t1 un 1 si es negativo y un 0 si es positivo



```
li    $t0 -3
move $t1 $t0
rol   $t1 $t1 1
and   $t1 $t1 0x00000001
```

## Instrucciones de comparación

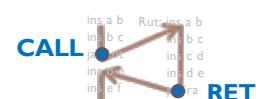
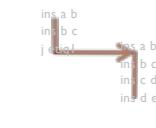
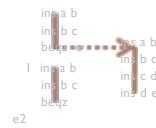
- ▶ **seq**   \$t0, \$t1, \$t2   if (\$t1 == \$t2)   \$t0 = 1; else \$t0 = 0
- ▶ **sneq** \$t0, \$t1, \$t2   if (\$t1 != \$t2)   \$t0 = 1; else \$t0 = 0
- ▶ **sge**   \$t0, \$t1, \$t2   if (\$t1 >= \$t2)   \$t0 = 1; else \$t0 = 0
- ▶ **sgt**   \$t0, \$t1, \$t2   if (\$t1 > \$t2)   \$t0 = 1; else \$t0 = 0
- ▶ **sle**   \$t0, \$t1, \$t2   if (\$t1 <= \$t2)   \$t0 = 1; else \$t0 = 0
- ▶ **slt**   \$t0, \$t1, \$t2   if (\$t1 < \$t2)   \$t0 = 1; else \$t0 = 0

# Instrucciones de comparación

- ▶ **seq \$t0, \$t1, \$t2** Set if equal
- ▶ **sneq \$t0, \$t1, \$t2** Set if no equal
- ▶ **sgc \$t0, \$t1, \$t2** Set if greater or equal
- ▶ **sgt \$t0, \$t1, \$t2** Set if greater than
- ▶ **sle \$t0, \$t1, \$t2** Set if less or equal
- ▶ **slt \$t0, \$t1, \$t2** Set if less than

# Control de Flujo

- ▶ Cambio de la secuencia de instrucciones a ejecutar (instrucciones de bifurcación)
- ▶ Distintos tipos:
  - ▶ Bifurcación o salto condicional:
    - ▶ Saltar a la posición etiqueta , si  $\$t0 <= \$t1$
    - ▶ Ej: `bne $t0 $t1 etiqueta`
  - ▶ Bifurcación o salto incondicional:
    - ▶ El salto se realiza siempre
    - ▶ Ej: `j etiqueta`
    - ▶ `b etiqueta`
  - ▶ Llamada a procedimiento:
    - ▶ Ej: `jal subrutina ..... jr $ra`



# Instrucciones de bifurcación

## ► Condicional:

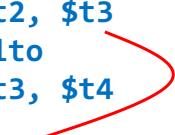
```
► beq    $t0    $t1    etiq    # salta a etiq1 si $t1 == $t0  
► bne    $t0    $t1    etiq    # salta a etiq1 si $t1 != $t0  
► beqz   $t1        etiq    # salta a etiq1 si $t1 == 0  
► bnez   $t1        etiq    # salta a etiq1 si $t1 != 0  
► bgt    $t0    $t1    etiq    # salta a etiq1 si $t0 > $t1  
► bge    $t0    $t1    etiq    # salta a etiq1 si $t0 >= $t1  
► blt    $t0    $t1    etiq    # salta a etiq1 si $t0 < $t1  
► ble    $t0    $t1    etiq    # salta a etiq1 si $t0 <= $t1
```

## ► Incondicional:

```
► b      etiq          # salta a etiq  
► j      etiq          # salta a etiq
```

**etiq** hace referencia una instrucción (representa a una dirección de memoria donde se encuentra la instrucción) a la que se salta:

```
add    $t1, $t2, $t3  
b      dir_salto  
add    $t2, $t3, $t4  
li     $t4, 1  
dir_salto: li    $t0, 4
```



# Ejercicio

Dada la siguiente expresión de un lenguaje de alto nivel

```
int a = 6;  
int b = 7;  
int c = 3;  
int d;  
  
d = (a+b) * (a+b);
```

Indique un fragmento de código en ensamblador del MIPS 32 que permita evaluar la expresión anterior.  
El resultado ha de almacenarse en el registro \$t5.

## Estructuras de control while

beq	\$t1 = \$t0
bnez	\$t1 = 0
bne	\$t1 != \$t0
bgt	\$t1 > \$t0
bge	\$t1 >= \$t0
blt	\$t1 < \$t0
ble	\$t1 <= \$t0

```
int i;

main ()
{
    i=0;
    while (i < 10) {

        /* acción */
        i = i + 1 ;
    }
}
```

## Estructuras de control while...

```
beq    $t1  = $t0
bnez   $t1  = 0
bne    $t1 != $t0
bgt    $t1 > $t0
bge    $t1 >= $t0
blt    $t1 < $t0
ble    $t1 <= $t0
```

```
int i;

main ()
{
    i=0;
    while (i < 10) {
        /* acción */
        i = i + 1 ;
    }
}
```

```
li $t0 0
li $t1 10
```

```
while:
```

```
bge $t0 $t1 fin
```

```
# acción
```

```
addi $t0 $t0 1
```

```
b while
```

```
fin: ...
```

# Ejercicio

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0



## Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0

$$1 + 2 + 3 + \dots + 10$$



NO vale  
;-)

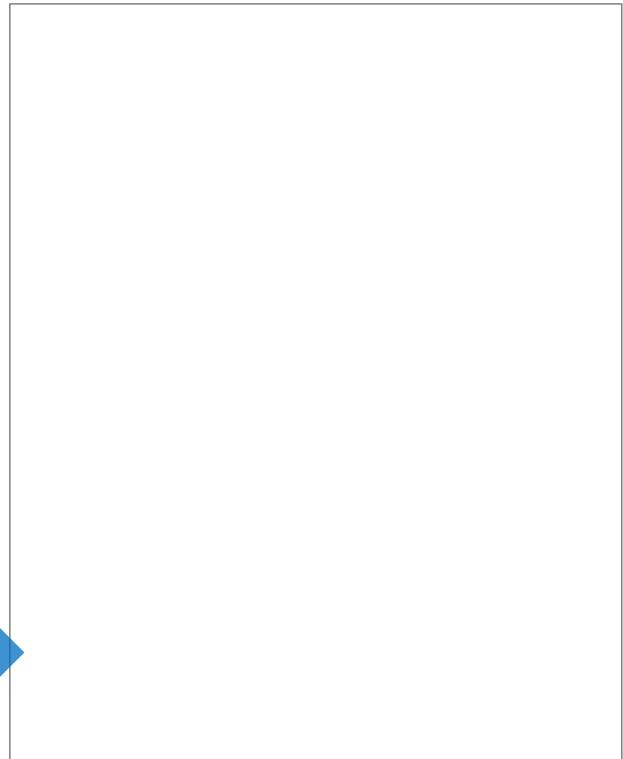
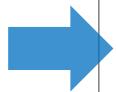
```
li $v0 0
add $v0 $v0 1
add $v0 $v0 2
add $v0 $v0 3
add $v0 $v0 4
add $v0 $v0 5
add $v0 $v0 6
add $v0 $v0 7
add $v0 $v0 8
add $v0 $v0 9
```

## Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0



```
int i, s;  
  
s=0;  
i=0;  
while (i <= 10)  
{  
    s = s + i ;  
    i = i + 1 ;  
}
```



## Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0



```
int i, s;  
  
s=0;  
i=0;  
while (i <= 10)  
{  
    s = s + i ;  
    i = i + 1 ;  
}
```



```
li $t0 0  
li $v0 0  
li $t2 10  
while1:  
    bgt $t0 t2 fin1  
    add $v0 $v0 $t0  
    add $t0 $t0 1  
    b while1  
fin1:
```

## Ejercicio

- ▶ Calcular el número de l's que hay en un registro (\$t0).  
Resultado en \$t3

## Ejercicio (solución)

- ▶ Calcular el número de 1's que hay en un registro (\$t0).  
Resultado en \$t3

```
i = 0;  
n = 45; #numero  
s = 0;  
while (i < 32)  
{  
    b = primer bit de n  
    s = s + b;  
    desplazar el contenido  
    de n un bit a la  
    derecha  
    i = i + 1;  
}
```

## Ejercicio (solución)

- ▶ Calcular el número de 1's que hay en un registro (\$t0).  
Resultado en \$t3

```
i = 0;  
n = 45; #numero  
s = 0;  
while (i < 32)  
{  
    b = primer bit de n  
    s = s + b;  
    desplazar el contenido  
    de n un bit a la  
    derecha  
    i = i + 1 ;  
}
```

```
i = 0;  
n = 45; #numero  
s = 0;  
while (i < 32)  
{  
    b = n & 1;  
    s = s + b;  
    n = n >> 1;  
    i = i + 1 ;  
}
```

## Ejercicio (solución)

- ▶ Calcular el número de 1's que hay en un registro (\$t0).  
Resultado en \$t3

```
i = 0;  
n = 45; #numero  
s = 0;  
while (i < 32)  
{  
    b = n & 1;  
    s = s + b;  
    n = n >> 1;  
    i = i + 1 ;  
}
```

```
li    $t0, 0      #i  
li    $t1, 45      #n  
li    $t2, 32  
li    $t3, 0      #s  
while: bge  $t0, t2, fin  
       and   $t4, $t1, 1  
       add   $t3, $t3, $t4  
       srl   $t1, $t1, 1  
       addi  $t0, $t0, 1  
b     while  
fin:  ...
```

# Ejemplo

Calcular el número de 1's que hay en un int en C/Java

```
int n = 45;
int b;
int i;
int s = 0;

for (i = 0; i < 32; i++) {
    b = n & 1;
    s = s + b;
    n = n >> 1;
}
printf("Hay %d\n", s);
```

## Ejemplo

- ▶ Calcular el número de 1's que hay en un int en C/Java

Otra solución:

```
int count[256] = {0,1,1,2,1,2,2,3,1, . . . 8};  
int i;  
int c = 0;  
  
for (i = 0; i < 4; i++) {  
    c = count[n & 0xFF];  
    s = s + c;  
    n = n >> 8;  
}  
printf("Hay %d\n", c);
```

## Ejercicio

- ▶ Obtener los 16 bits superiores de un registro (\$t0) y dejarlos en los 16 bits inferiores de otro (\$t1)

## Ejercicio (solución)

- ▶ Obtener los 16 bits superiores de un registro (\$t0) y dejarlos en los 16 bits inferiores de otro (\$t1)

```
srl    $t1,    $t0,    16
```



Se desplaza a la derecha 16  
Posiciones (de forma lógica)

# Estructuras de control

## if

```
int a=1;
int b=2;

main ()
{
    if (a < b) {
        a = b;
    }
    ...
}
```

beq	\$t1 = \$t0
bnez	\$t1 = 0
bne	\$t1 != \$t0
bgt	\$t1 > \$t0
bge	\$t1 >= \$t0
blt	\$t1 < \$t0
ble	\$t1 <= \$t0

## Estructuras de control

### if

```
int a=1;
int b=2;

main ()
{
    if (a < b) {
        a = b;
    }
    ...
}
```

```
beq    $t1  = $t0
bnez   $t1  = 0
bne    $t1 != $t0
bgt    $t1 >  $t0
bge    $t1 >= $t0
blt    $t1 <  $t0
ble    $t1 <= $t0
```

```
li    $t1 1
li    $t2 2

if_1: blt $t1 $t2 then_1
      b    fin_1
then_1: move $t1 $t2
fin_1: ...
```

# Estructuras de control

## if

```
int a=1;
int b=2;

main ()
{
    if (a < b) {
        a = b;
    }
    ...
}
```

beq	\$t1 = \$t0
bnez	\$t1 = 0
bne	\$t1 != \$t0
bgt	\$t1 > \$t0
bge	\$t1 >= \$t0
blt	\$t1 < \$t0
ble	\$t1 <= \$t0

```
li $t1 1
li $t2 2

if_2: bge $t1 $t2 fin_2
then_2: move $t1 $t2
fin_2: ...
```

## Estructuras de control if-else

```
int a=1;
int b=2;

main ()
{
    if (a < b){
        // acción 1
    } else {
        // acción 2
    }
}
```

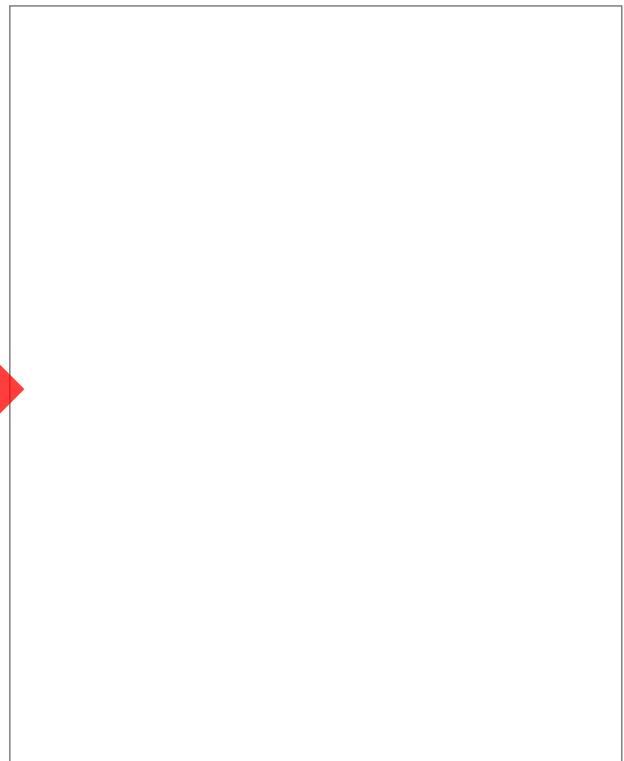
```
beq    $t1 = $t0
bnez   $t1 = 0
bne    $t1 != $t0
bgt    $t1 > $t0
bge    $t1 >= $t0
blt    $t1 < $t0
ble    $t1 <= $t0
```

```
li    $t1 1
li    $t2 2

if_3:  bge $t1 $t2 else_3
then_3: # acción 1
        b fi_3
else_3: # acción 2
        fi_3: ...
```

# Ejercicio

```
int b1 = 4;  
int b2 = 2;  
  
if (b2 == 8) {  
    b1 = 1;  
}  
...
```



## Ejercicio (solución)

```
int b1 = 4;  
int b2 = 2;  
  
if (b2 == 8) {  
    b1 = 1;  
}  
...
```

```
li    $t0 4  
li    $t1 2  
li    $t2 8  
  
bneq $t0 $t2 fin1  
li    $t1 1  
fin1: ...
```

## Ejercicio

- ▶ Determinar si el contenido de un registro (\$t2) es par.  
Si es par se almacena en \$t1 un 1, sino se almacena un 0

## Ejercicio (solución)

- ▶ Determinar si el contenido de un registro (\$t2) es par.  
Si es par se almacena en \$t1 un 1, sino se almacena un 0

```
    li  $t2,  9
    li  $t1,  2
    rem $t1,  $t2,  $t1      # se obtiene el resto
    bne $t1,  $0,  else     # cond.
then: li  $t1, 1
      b   fin                  # incond.
else: li  $t1, 0
fin: ...
```

## Ejercicio (otra solución)

- ▶ Determinar si el contenido de un registro (\$t2) es par.  
Si es par se almacena en \$t1 un 1, sino se almacena un 0

```
    li  $t2,  9
    li  $t1,  2
    rem $t3,  $t2,  $t1      # se obtiene el resto
    li  $t1, 0                 # suponer impar
    bne $t3,  $0,  fin        # si suposición ok, fin
    li  $t1, 1
fin: ...
```

## Ejercicio

- ▶ Determinar si el contenido de un registro (\$t2) es par.  
Si es par se almacena en \$t1 un 1, sino se almacena un 0.  
En este caso consultando el último bit

## Ejercicio (solución)

- ▶ Determinar si el contenido de un registro (\$t2) es par.  
Si es par se almacena en \$t1 un 1, sino se almacena un 0.  
En este caso consultando el último bit

```
    li  $t2, 9
    li  $t1, 1
    and $t1, $t2, $t1      # se obtiene el último bit
    beq $t1, $0  then      # cond.
else: li  $t1, 0
      b   fin                  # incond.
then: li  $t1, 1
fin: ...
```

# Ejercicio

- ▶ Calcular  $a^n$ 
  - ▶ a en \$t0
  - ▶ n en \$t1
  - ▶ El resultado en \$t2

```
a=8
n=4;
i=0;
p = 1;
while (i < n)
{
    p = p * a
    i = i + 1 ;
}
```

## Ejercicio (solución)

- ▶ Calcular  $a^n$ 
  - ▶ a en \$t0
  - ▶ n en \$t1
  - ▶ El resultado en \$t2

```
a=8  
n=4;  
i=0;  
p = 1;  
while (i < n)  
{  
    p = p * a  
    i = i + 1 ;  
}  
}
```

```
li    $t0, 8  
li    $t1, 4  
li    $t2, 1  
li    $t4, 0  
  
while: bge  $t4, $t1, fin  
       mul  $t2, $t2, $t0  
       addi $t4, $t4, 1  
       b     while  
fin:   move $t2, $t4
```

# Fallos típicos

## 1) Programa mal planteado

- ▶ No hace lo que se pide
- ▶ Hace incorrectamente lo que se pide

## 2) Programar directamente en ensamblador

- ▶ No codificar en pseudo-código el algoritmo a implementar

## 3) Escribir código ilegible

- ▶ No tabular el código
- ▶ No comentar el código ensamblador o no hacer referencia al algoritmo planteado inicialmente

Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 3 (II)

### Fundamentos de la programación en ensamblador

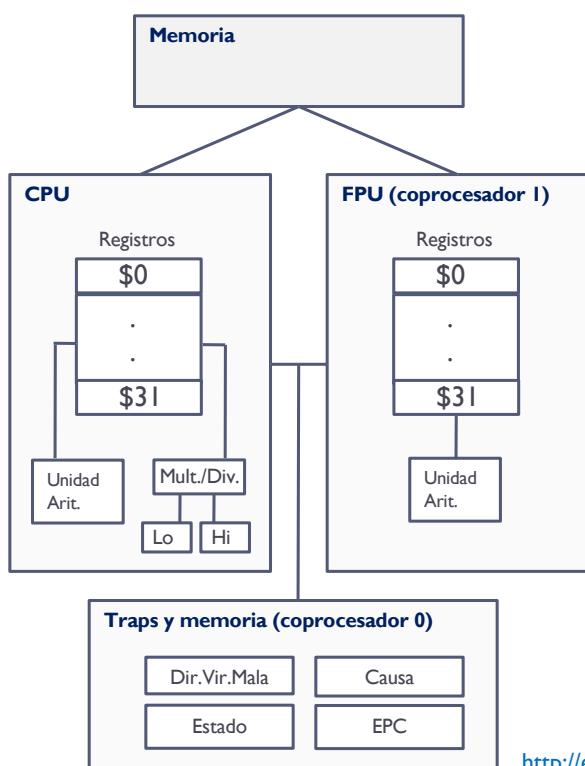
Estructura de Computadores  
Grado en Ingeniería Informática



# Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del MIPS 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila

# Arquitectura del MIPS 32



## ▶ MIPS 32

- ▶ Procesador de 32 bits
- ▶ Tipo RISC
- ▶ CPU + coprocesadores auxiliares

## ▶ Coprocesador 0

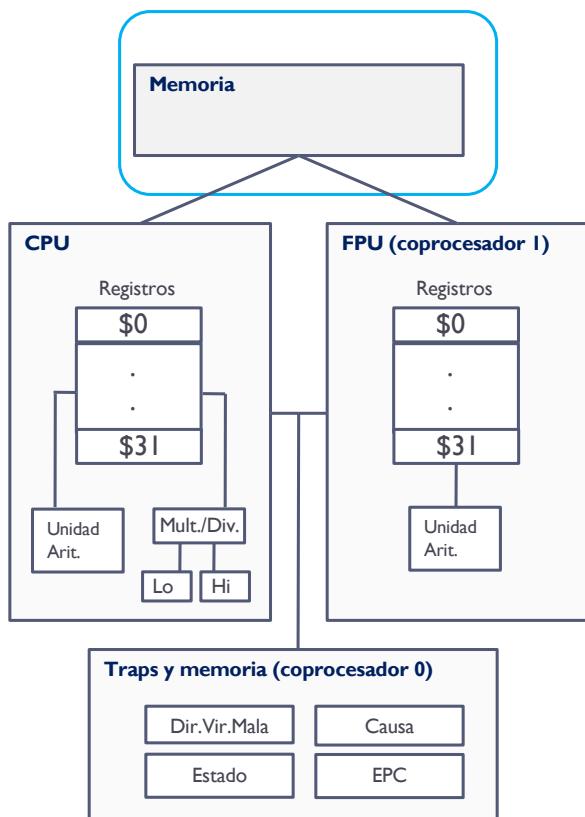
- ▶ excepciones, interrupciones y sistema de memoria virtual

## ▶ Coprocesador I

- ▶ FPU (Unidad de Punto Flotante)

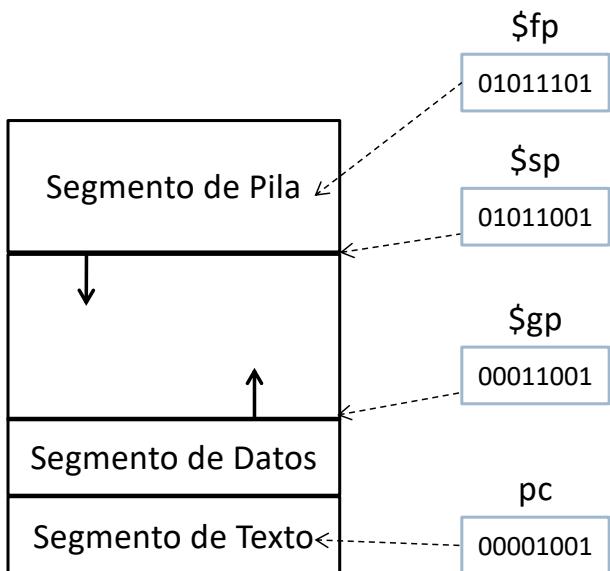
[http://es.wikipedia.org/wiki/MIPS\\_\(procesador\)](http://es.wikipedia.org/wiki/MIPS_(procesador))

# Arquitectura del MIPS 32



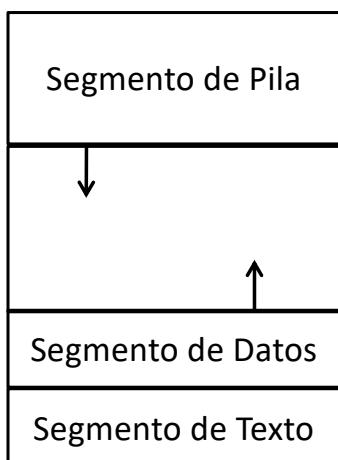
- ▶ Direcciones de memoria de 2 bits
- ▶ 4 GB direccionables

# Mapa de memoria de un proceso



- ▶ Los procesos dividen el espacio de memoria en segmentos lógicos para organizar el contenido:
  - ▶ Segmento de pila
    - ▶ Variables locales
    - ▶ Contexto de funciones
  - ▶ Segmento de datos
    - ▶ Datos estáticos
  - ▶ Segmento de código (texto)
    - ▶ Código

# Ejercicio

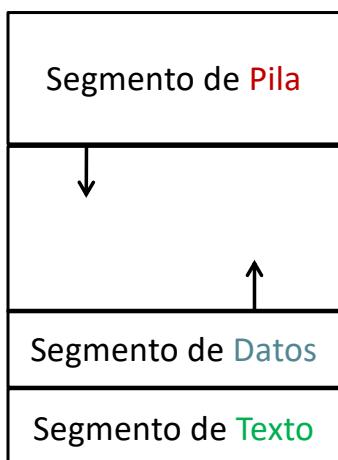


```
// variables globales
int a;

main ()
{
    // variables locales
    int b;

    // código
    return a + b;
}
```

# Ejercicio (solución)



```
// variables globales
int a;

main ()
{
    // variables locales
    int b;

    // código
    return a + b;
}
```

# Banco de registros (enteros) del MIPS 32

Nombre registro	Número	Uso
zero	0	Constante 0
at	1	Reservado para el ensamblador
v0, v1	2, 3	Resultado de una rutina (o expresión)
a0, ..., a3	4, ..., 7	Argumento de entrada para rutinas
t0, ..., t7	8, ..., 15	Temporal ( <u>NO</u> se conserva entre llamadas)
s0, ..., s7	16, ..., 23	Temporal (se conserva entre llamadas)
t8, t9	24, 25	Temporal ( <u>NO</u> se conserva entre llamadas)
k0, k1	26, 27	Reservado para el sistema operativo
gp	28	Puntero al área global
sp	29	Puntero a pila
fp	30	Puntero a marco de pila
ra	31	Dirección de retorno (rutinas)

- ▶ Hay 32 registros
  - ▶ 4 bytes de tamaño (una palabra)
  - ▶ Se nombran con un \$ al principio
- ▶ Convenio de uso
  - ▶ Reservados
  - ▶ Argumentos
  - ▶ Resultados
  - ▶ Temporales
  - ▶ Punteros

# Banco de registros del MIPS 32

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

- Hay 32 registros
  - 4 bytes de tamaño (una palabra)
  - Se nombran con un \$ al principio
- Convenio de uso
  - Reservados
  - Argumentos
  - Resultados
  - Temporales
  - Punteros

16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	

# Banco de registros del MIPS 32

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

\$zero

Valor cableado a cero

No puede modificarse

16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

# Banco de registros del MIPS 32

0	\$zero	16
1		17
2		18
3		19
4		20
5		21
6		22
7		23
8	\$t0	24
9	\$t1	25
10	\$t2	26
11	\$t3	27
12	\$t4	28
13	\$t5	29
14	\$t6	30
15	\$t7	31

# Banco de registros del MIPS 32

0	\$zero	\$s0	16
1		\$s1	17
2		\$s2	18
3		\$s3	19
4		\$s4	20
5		\$s5	21
6		\$s6	22
7		\$s7	23
8	\$t0	\$t8	24
9	\$t1	\$t9	25
10	\$t2		26
11	\$t3		27
12	\$t4		28
13	\$t5		29
14	\$t6		30
15	\$t7		31

# Banco de registros del MIPS 32

0	\$zero	\$s0	16
1		\$s1	17
2		\$s2	18
3	\$v0	\$s3	19
4	\$v1	\$s4	20
5	\$a0	\$s5	21
6	\$a1	\$s6	22
7	\$a2	\$s7	23
8	\$a3	\$t0	24
9		\$t1	25
10		\$t2	26
11		\$t3	27
12		\$t4	28
13		Paso de parámetros y Gestión de subrutinas	\$sp
14		\$t5	29
15		\$t6	\$fp
		\$t7	\$ra
			30
			31

# Banco de registros del MIPS 32

0	\$zero	\$s0	16
1	\$at	\$s1	17
2	\$v0	\$s2	18
3	\$v1	\$s3	19
4	\$a0	\$s4	20
5	\$a1	\$s5	21
6	\$a2	\$s6	22
7	\$a3	\$s7	23
8	\$t0	\$t8	24
9	\$t1	\$t9	25
10	\$t2	\$k0	26
11	\$t3	\$k1	27
12	\$t4	\$gp	28
13	\$t5	\$sp	29
14	\$t6	\$fp	30
15	\$t7	\$ra	31

Otros

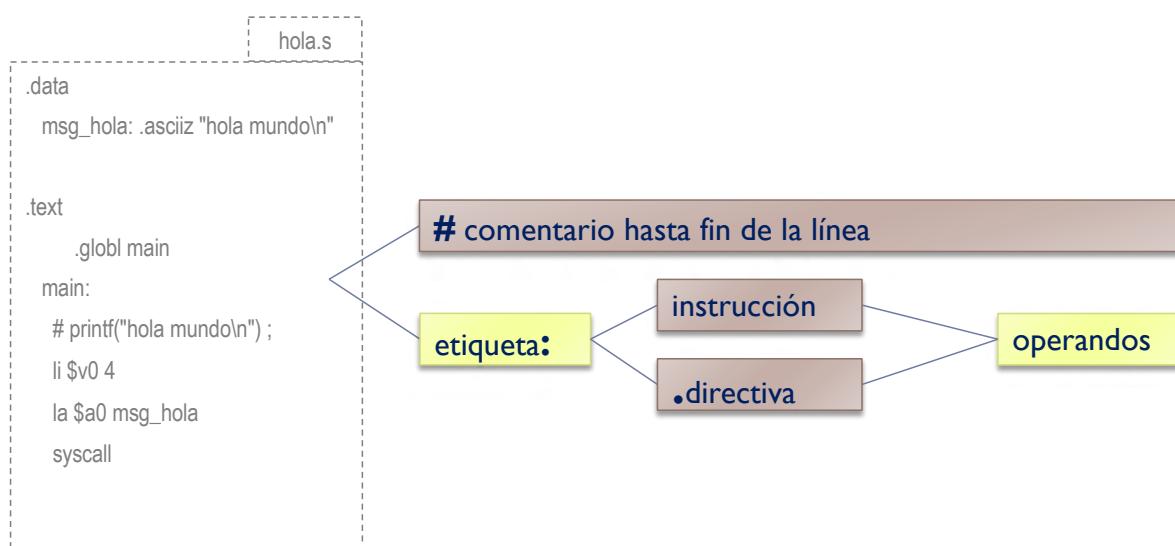
## Ejemplo: Hola mundo...

holas

```
.data
msg_hola: .asciiz "hola mundo\n"

.text
.globl main
main:
# printf("hola mundo\n") ;
li $v0 4
la $a0 msg_hola
syscall
```

# Ejemplo: Hola mundo...



## Ejemplo: Hola mundo...

hola.s

```
.data
    msg_hola: .asciiz "hola mundo\n"

.text
.globl main
main:
    # printf("hola mundo\n") ;
    li $v0 4
    la $a0 msg_hola
    syscall
```

## Ejemplo: Hola mundo...

hola.s

```
.data
    msg_hola: .asciiz "hola mundo\n"

.text
.globl main
main:           etiqueta: representa la dirección de memoria donde comienza la función main
    # printf("hola mundo\n") ;
    li $v0 4             comentarios
    la $a0 msg_hola      instrucciones
    syscall
```

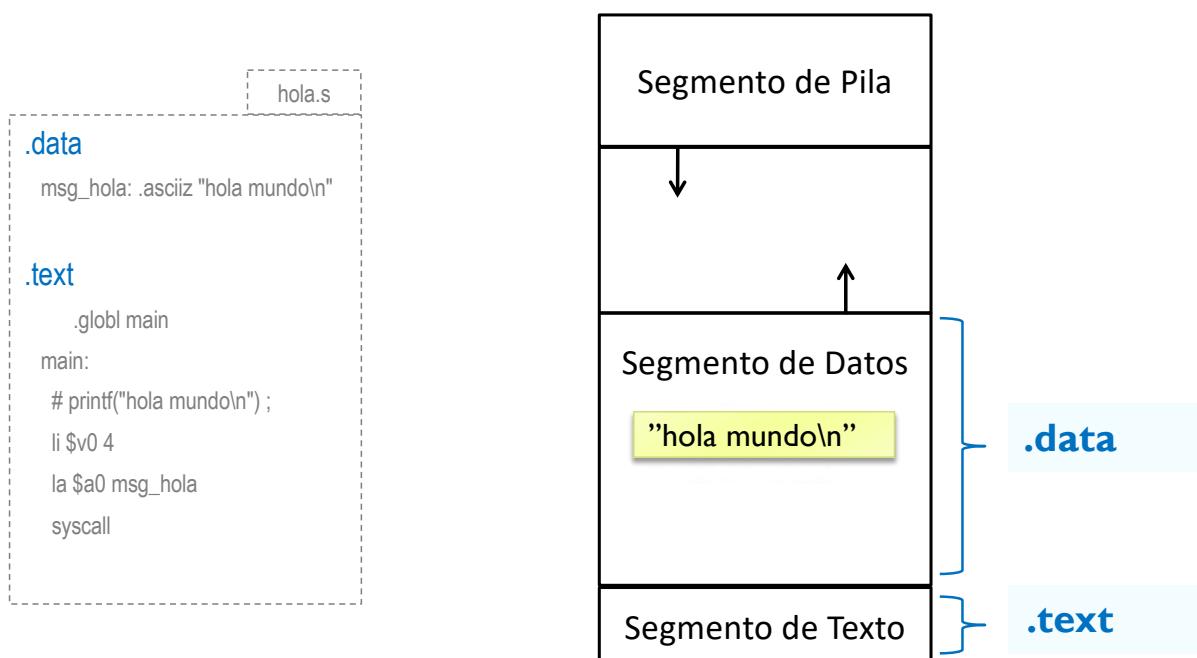
## Ejemplo: Hola mundo...

```
segmento de datos  
holas  
.data  
msg_hola: .asciiz "hola mundo\n"  
.text  
.globl main  
main:  
    # printf("hola mundo\n");  
    li $v0 4  
    la $a0 msg_hola  
    syscall
```

msg\_hola: representa la dirección de memoria donde comienza la cadena

segmento de código

# Programa en ensamblador: directivas de ensamblador (de preproceso)



# Programa en ensamblador: directivas de ensamblador

Directivas	Uso
<code>.data</code>	Siguientes elementos van al segmento de dato
<code>.text</code>	Siguientes elementos van al segmento de código
<code>.ascii "tira de caracteres"</code>	Almacena cadena caracteres NO terminada en carácter nulo
<code>.asciiz "tira de caracteres"</code>	Almacena cadena caracteres terminada en carácter nulo
<code>.byte I, 2, 3</code>	Almacena bytes en memoria consecutivamente
<code>.half 300, 301, 302</code>	Almacena medias palabras en memoria consecutivamente
<code>.word 800000, 800001</code>	Almacena palabras en memoria consecutivamente
<code>.float 1.23, 2.13</code>	Almacena float en memoria consecutivamente
<code>.double 3.0e21</code>	Almacena double en memoria consecutivamente
<code>.space 10</code>	Reserva un espacio de 10 bytes en el segmento actual
<code>.extern etiqueta n</code>	Declara que <code>etiqueta</code> es global de tamaño $n$
<code>.globl etiqueta</code>	Declara <code>etiqueta</code> como global
<code>.align n</code>	Alinea el siguiente dato en un límite de $2^n$

# Definición de datos estáticos

etiqueta (dirección) tipo de dato (directiva) valor

```
.data
cadena : .asciiz "Hola mundo\n"
i1: .word 10          # int i1=10
i2: .word -5         # int i2=-5
i3: .half 300        # short i3=300
c1: .byte 100         # char c1=100
c2: .byte 'a'         # char c2='a'
f1: .float 1.3e-4    # float f1=1.3e-4
d1: .double .001     # double d1=0.001

# int v[3] = { 0 , -1, 0xffffffff }; int w[100];
v: .word 0, -1, 0 xfffffff
w: .word 400
```

# Llamadas al sistema

- ▶ Muchos simuladores de ensamblador incluyen un pequeño “sistema operativo”
  - ▶ Ofrece 17 servicios.
- ▶ Invocación:
  - ▶ Código de servicio en \$v0
  - ▶ Otros parámetros en registros concretos
  - ▶ Invocación mediante instrucción máquina **syscall**

# Llamadas al sistema

Servicio	Código de llamada (\$v0)	Argumentos	Resultado
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer en \$v0
read_float	6		float en \$f0
read_double	7		double en \$f0
read_string	8	\$a0=buffer, \$a1=longitud	
sbrk	9	\$a0=cantidad	dirección en \$v0
exit	10		

# Llamadas al sistema

Servicio	Código de llamada (\$v0)	Argumentos	Resultado
print_char	11	\$a0 (código ASCII)	
read_char	12		\$v0 (código ASCII)
open	13	Equivalente a \$v0 = open(\$a0, \$a1, \$a2)	descriptor de fichero en \$v0
read	14	Equivalente a \$v0 = read (\$a0, \$a1, \$a2)	bytes leídos en \$v0
write	15	Equivalente a \$v0 = write(\$a0, \$a1, \$a2)	bytes escritos en \$v0
close	16	Equivalente a \$v0 = close(\$a0)	0 en \$v0
exit2	17	Termina el programa y hace que spim devuelva el código de error almacenado en \$a0	

## Ejemplo: Hola mundo...

hola.s

```
.data  
    msg_hola: .asciiz "hola mundo\n"
```

```
.text  
.globl main  
main:
```

```
# printf("hola mundo\n") ;
```

```
li $v0 4
```

```
la $a0 msg_hola
```

```
syscall
```

Servicio	Código de llamada	Argumentos
print_int	1	\$a0 = integer
print_float	2	\$f12 = float
print_double	3	\$f12 = double
print_string	4	\$a0 = string

instrucción de  
llamada al sistema

# Ejercicio

```
• • •  
int valor ;  
• • •  
readInt(&valor) ;  
valor = valor + 1 ;  
printInt(valor) ;  
• • •
```

Servicio	Código de llamada	Argumentos	Resultado
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer en \$v0
read_float	6		float en \$f0
read_double	7		double en \$f0
read_string	8	\$a0=buffer, \$a1=long.	
sbrk	9	\$a0=cantidad	dirección en \$v0
exit	10		

# Ejercicio (solución )

```
 . . .
int valor ;
. . .
readInt(&valor) ;
valor = valor + 1 ;
printInt(valor) ;
. . .
```

Servicio	Código	Argumentos	Resultado
print_int	1	\$a0 = integer	
read_int	5		integer en \$v0

```
 . . .
# readInt(&valor)
li $v0 5
syscall
sw $v0 valor

# valor = valor + 1
add $a0 $v0 1
sw $a0 valor

# printInt
li $v0 1
syscall
```

# Ejercicio

```
int x = 10;
int y = 20;
main() {
    print_string("La suma de ");
    print_int(x);
    print_string(" y de ");
    print_int(y);
    print_string(" es ");
    print_int(x+y);
    print_character("\n");
}
```

# Ejercicio (solución )

```
int x = 10;
int y = 20;
main() {
    print_string("La suma de ");
    print_int(x);
    print_string(" y de ");
    print_int(y);
    print_string(" es ");
    print_int(x+y);
    print_character("\n");
}
```

```
.rdata
m1:    .asciiz "La suma de "
m2:    .asciiz " y de "
m3:    .asciiz " es "
salto: .ascii "\n"

.data
x:     .word 10
y:     .word 20

.text
.globl main
main:
    la $a0, m1
    li $v0, 4
    syscall # print_string(m1)
    lw $a0, x
    li $v0, 1
    syscall # print_int(x)
    la $a0, m2
    li $v0, 4
    syscall # print_string(m2)
    lw $a0, y
    li $v0, 1
    syscall # print_int(y)
    la $a0, m3
    li $v0, 4
    syscall # print_string(m3)
    lw $t0, x
    lw $t1, y
    add $a0, $t0, $t1 # $a0= x + y
    li $v0, 1
    syscall # print_int($a0)
    lb $a0, salto
    li $v0, 11
    syscall # print_character(salto)
    jr $ra
```

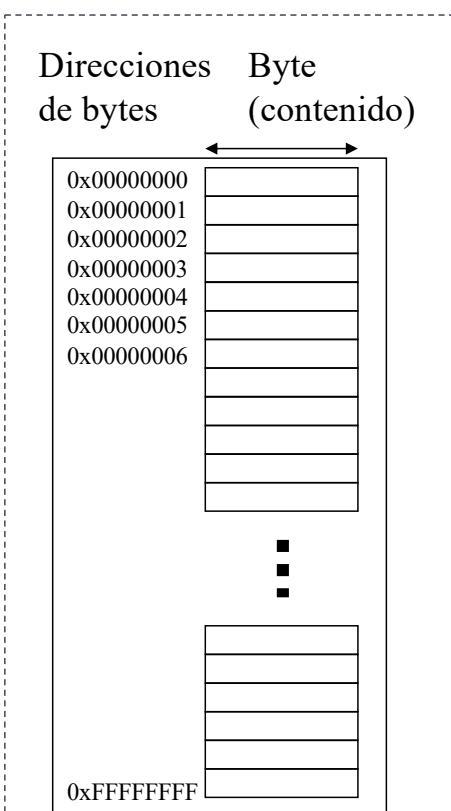
# Instrucciones y pseudoinstrucciones

- ▶ Una instrucción en ensamblador se corresponde con una única instrucción máquina
  - ▶ Ocupa 32 bits en el MIPS 32
  - ▶ `addi $t1,$t0,4`
- ▶ Una pseudoinstrucción se puede utilizar en un programa en ensamblador pero no se corresponde con ninguna instrucción máquina
  - ▶ Ej: `li $v0, 4`  
`move $t1,$t0`
- ▶ En el proceso de ensamblado se sustituyen por la secuencia de instrucciones máquina que realizan la misma funcionalidad.
  - ▶ Ej.: `ori $v0, $0, 4` sustituye a: `li $v0, 4`  
`addu $t1,$0,$t2` sustituye a: `move $t1, $t2`

## Otros ejemplos de pseudoinstrucciones

- ▶ Una pseudoinstrucción en ensamblador se puede corresponder con varias instrucciones máquina.
- ▶ `li $t1, 0x00800010`
  - ▶ No cabe en 32 bits, pero se puede utilizar como pseudoinstrucción.
  - ▶ Es equivalente a:
    - `lui $t1, 0x0080`
    - `ori $t1, $t1, 0x0010`

## Modelo de memoria del MIPS 32



La memoria se direcciona por bytes:

- Direcciones de 32 bits
- Contenido de cada dirección: un byte
- Espacio direccionable:  $2^{32}$  bytes = 4GB

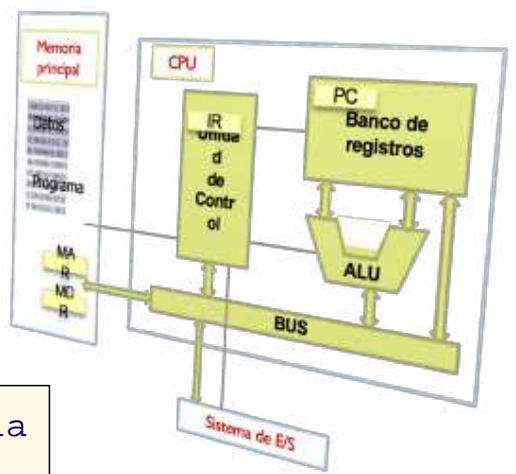
El acceso puede ser a:

- Bytes individuales
- Palabras (4 bytes consecutivos)

## Formato de las instrucciones de acceso a memoria

lw  
sw  
lb  
sb  
lbu

Registro, dirección de memoria



- Número que representa una dirección
- Etiqueta simbólica que representa una dirección
- (registro): representa la dirección almacenada en el registro
- num(registro): representa la dirección que se obtiene de sumar num con la dirección almacenada en el registro

## Formatos de las instrucciones de acceso memoria

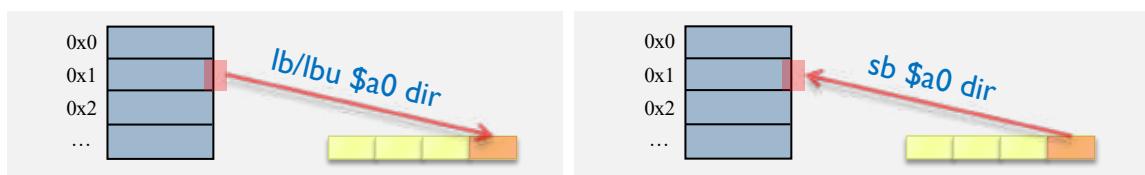
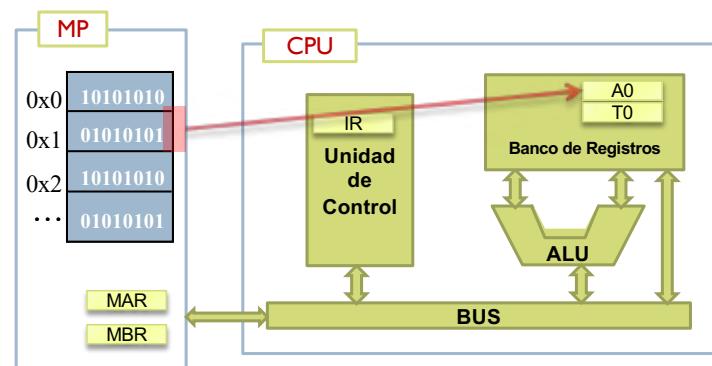
- ▶ **lbu \$t0 , 0xF000002**
  - ▶ Direccionamiento directo. Se carga en \$t0 el byte almacenado en la posición de memoria 0x0F000002
- ▶ **lbu \$t0 , etiqueta**
  - ▶ Direccionamiento directo. Se carga en \$t0 el byte almacenado en la posición de memoria etiqueta
- ▶ **lbu \$t0 , (\$t1)**
  - ▶ Direccionamiento indirecto de registro. Se carga en \$t0 el byte almacenado en la posición de memoria almacenada en \$t1
    - ▶ **lbu \$t0 , 80(\$t1)**
  - ▶ Direccionamiento relativo. Se carga en \$t0 el byte almacenado en la posición de memoria que se obtiene de sumar el contenido de \$t1 con 80

# Transferencia de datos bytes

- ▶ Copia un **byte** de memoria a un **registro** o viceversa

- ▶ Para bytes:

- ▶ Memoria a registro
  - lb \$a0, dir**
  - lbu \$a0, dir**
- ▶ Registro a memoria
  - sb \$t0, dir**



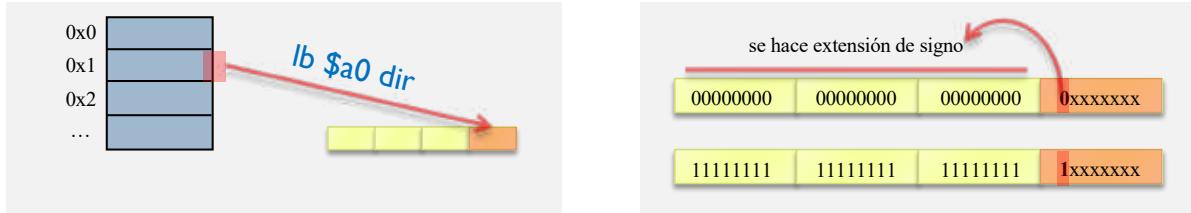
# Transferencia de datos

## Extensión de signo

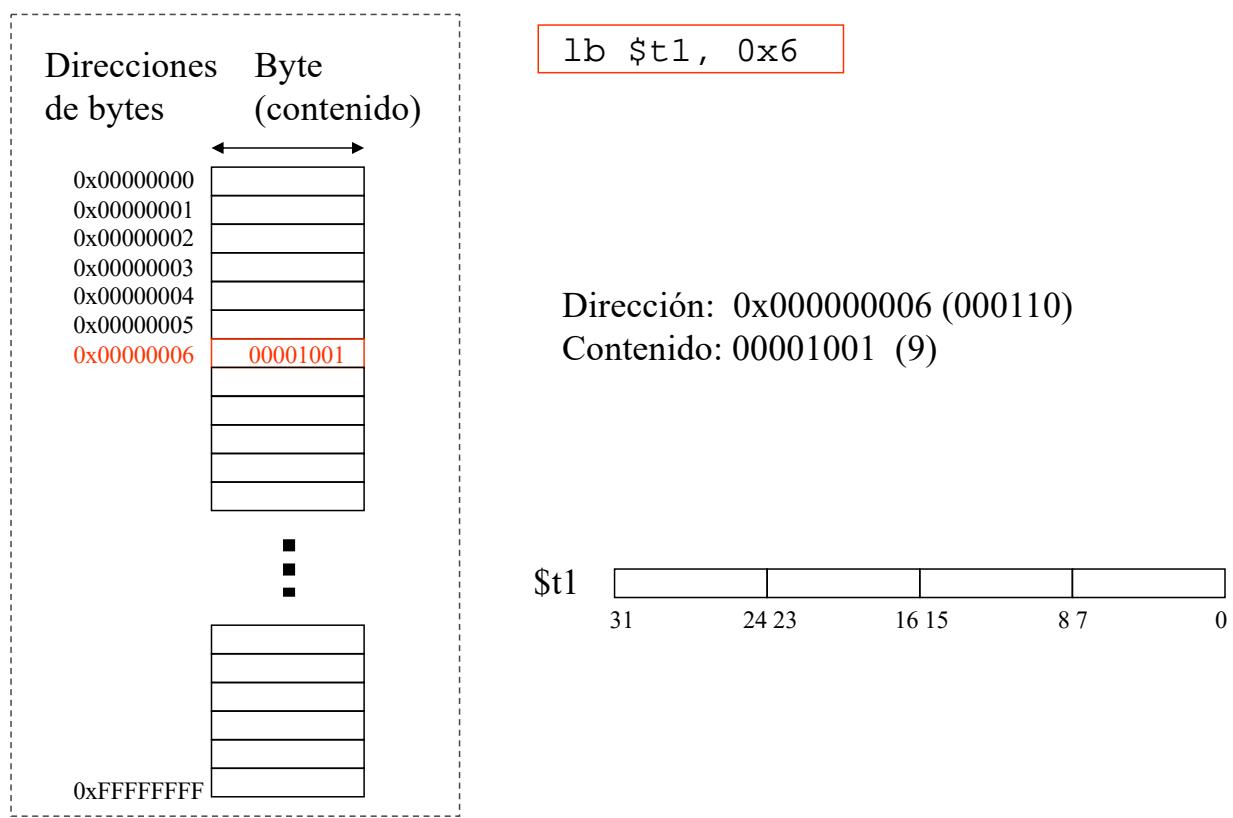
- ▶ Hay dos posibilidades a la hora de traer un byte de memoria a registro:
- ▶ A) Transferir sin signo, por ejemplo: `Ibu $a0, dir`



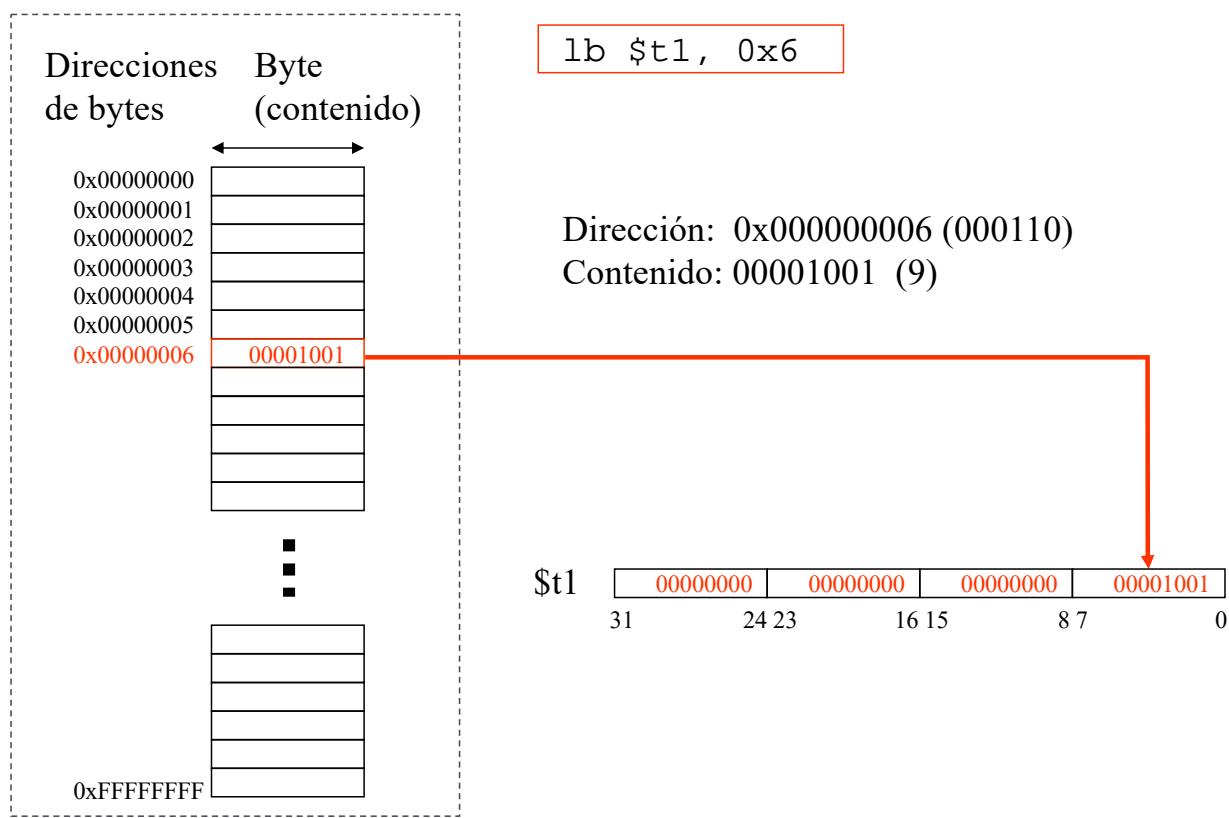
- ▶ B) Transferir con signo, por ejemplo: `lb $a0, dir`



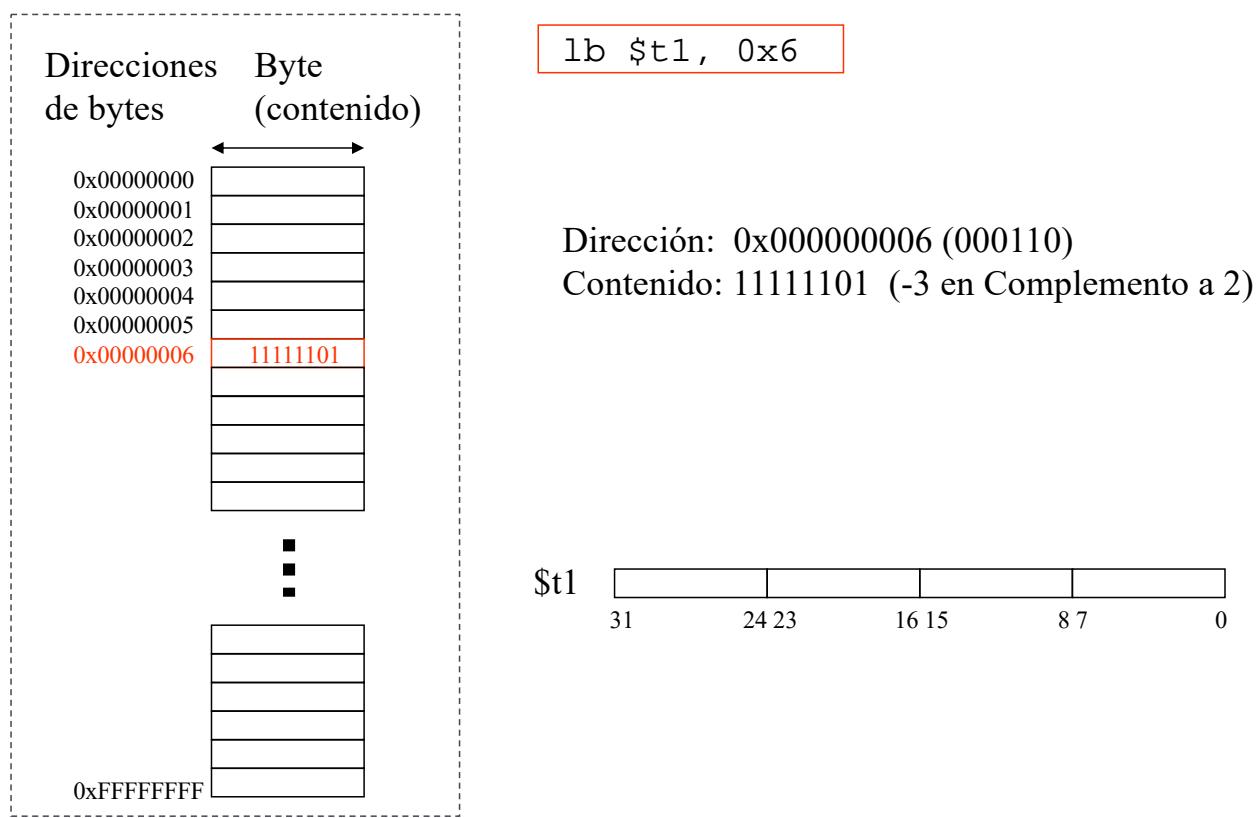
## Acceso a bytes con lb (load byte)



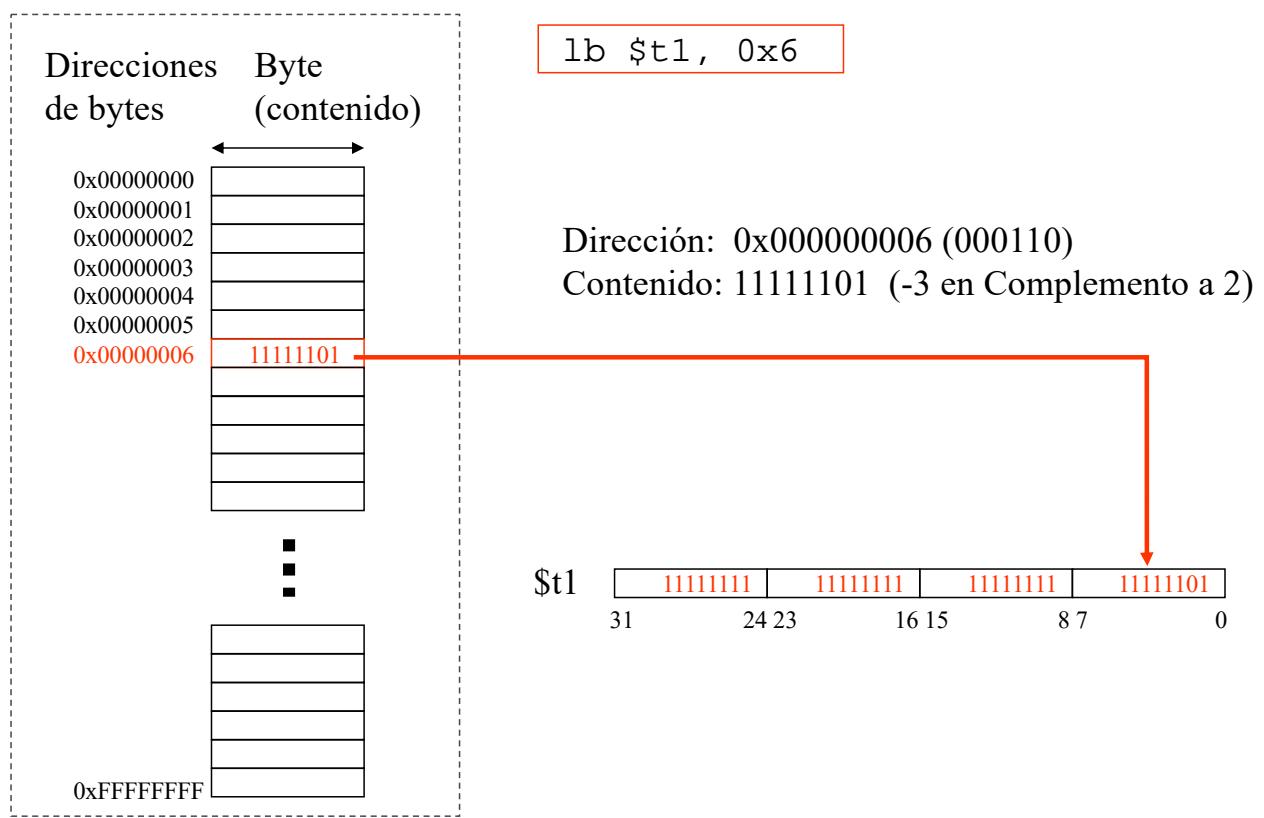
## Acceso a bytes con lb



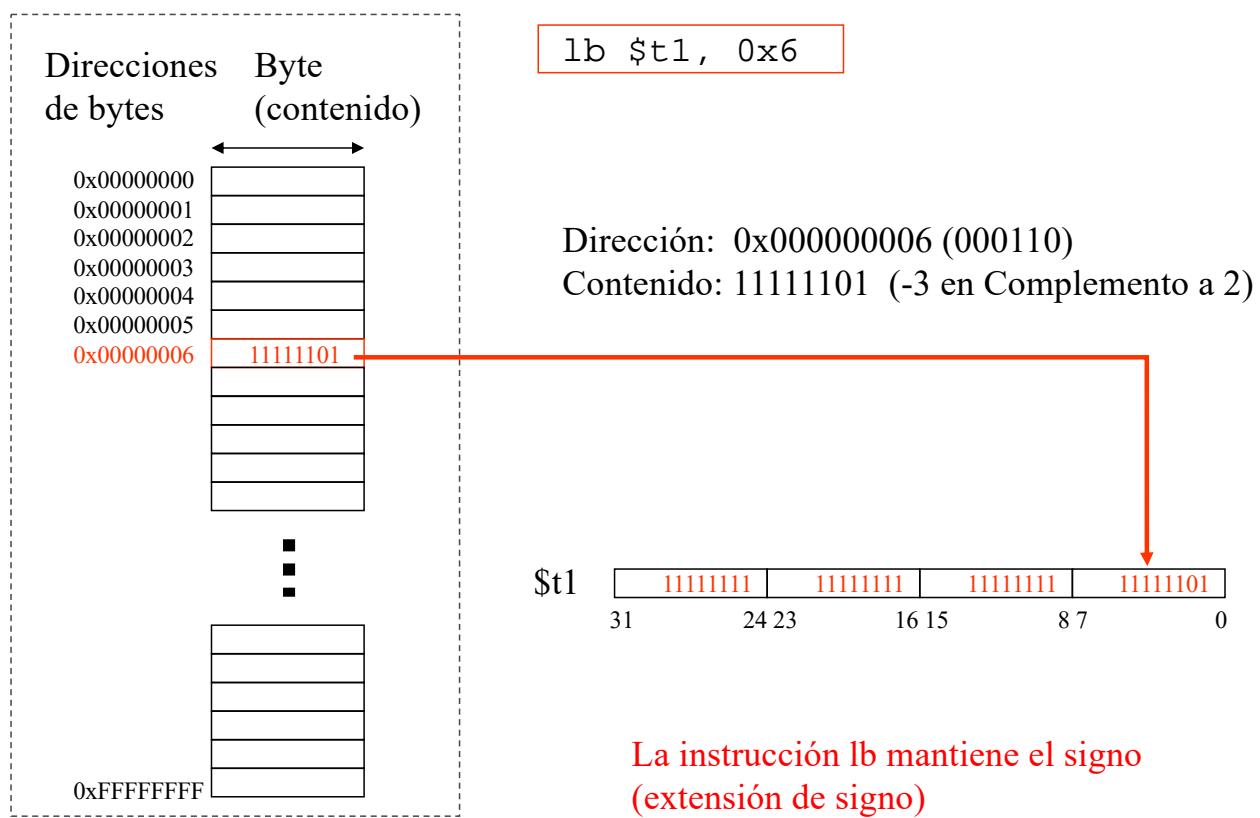
## Acceso a bytes con lb



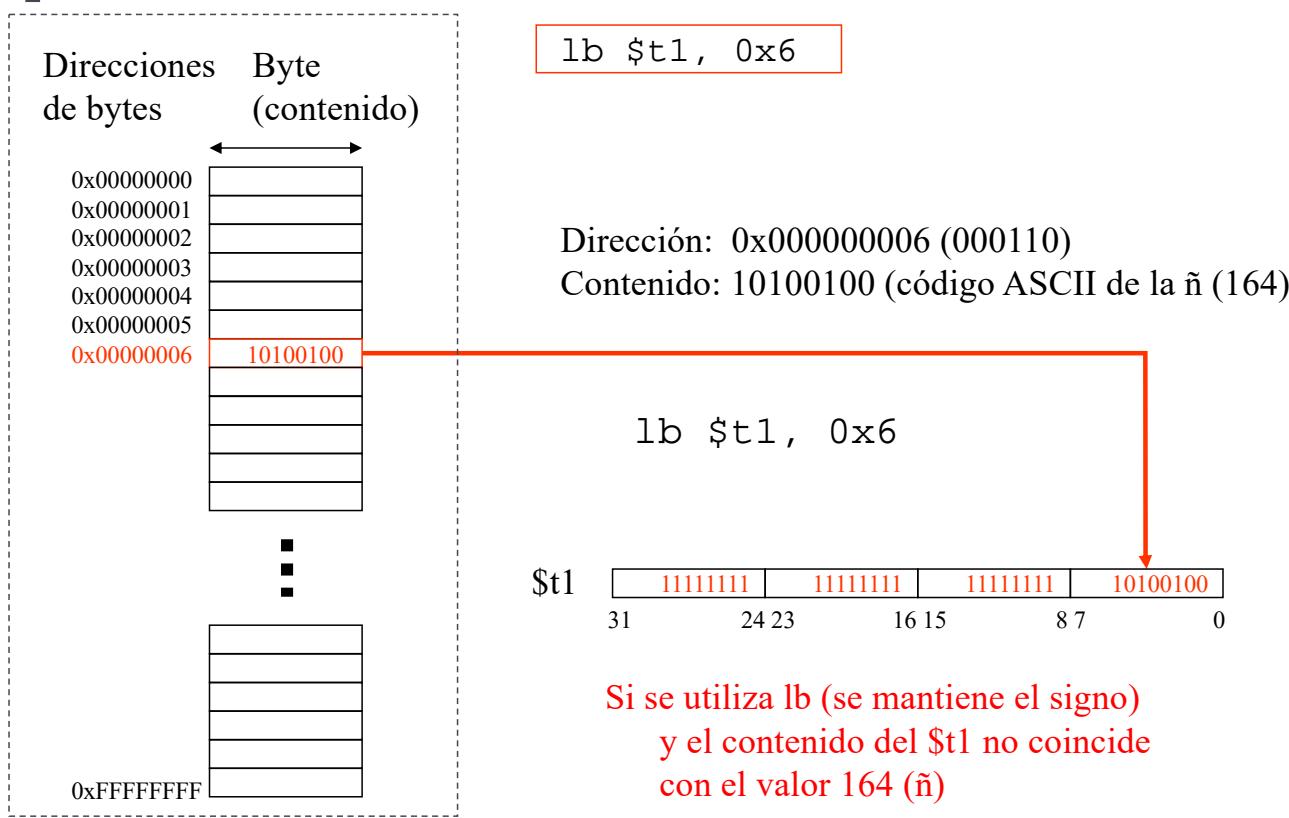
## Acceso a bytes con lb



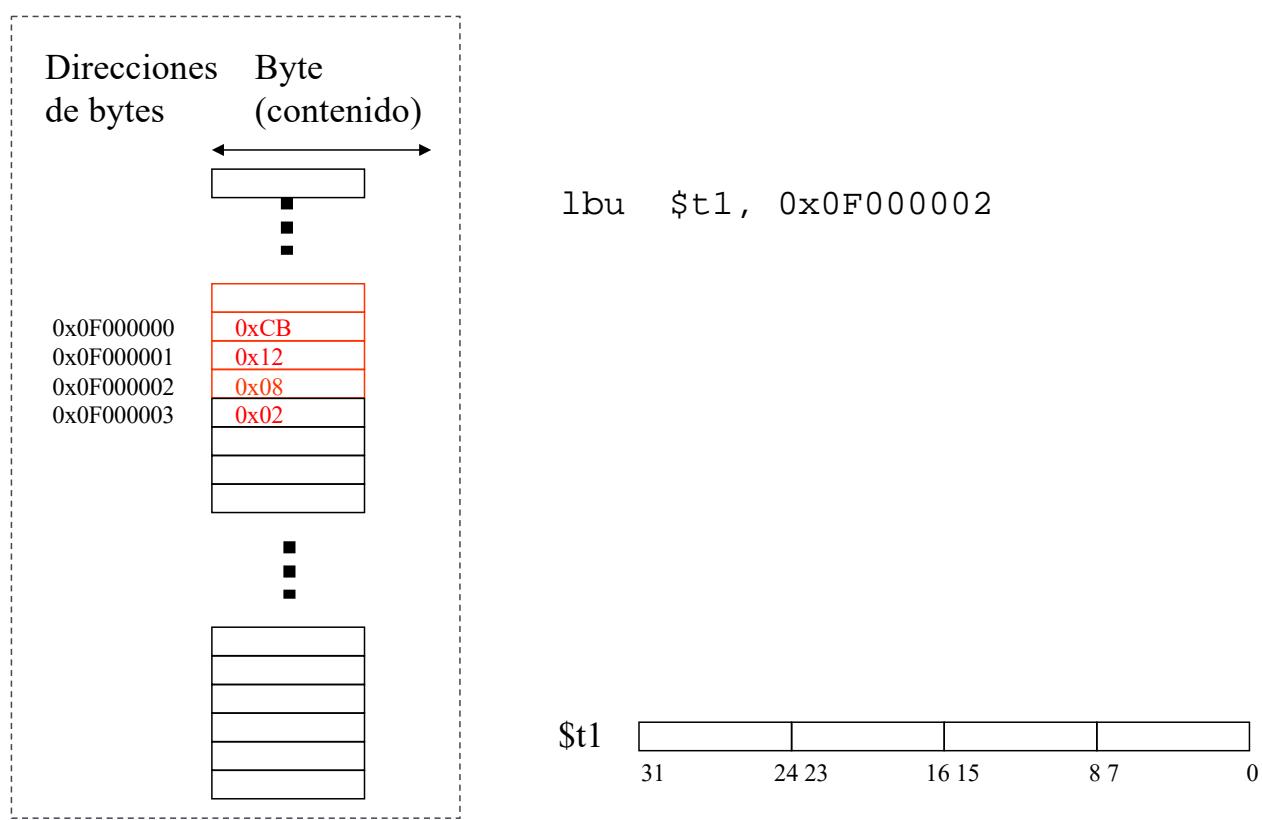
## Acceso a bytes con lb



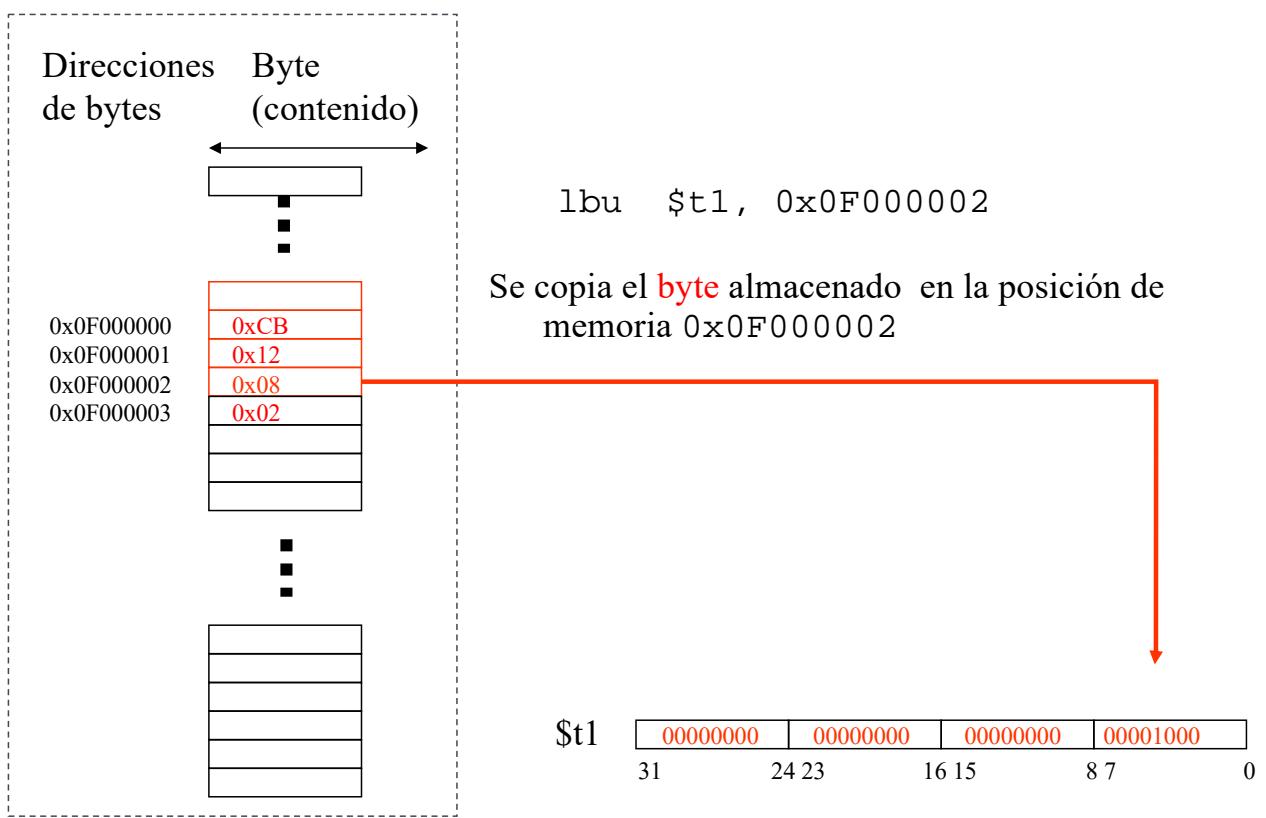
## Acceso a bytes con lb problemas accediendo a caracteres



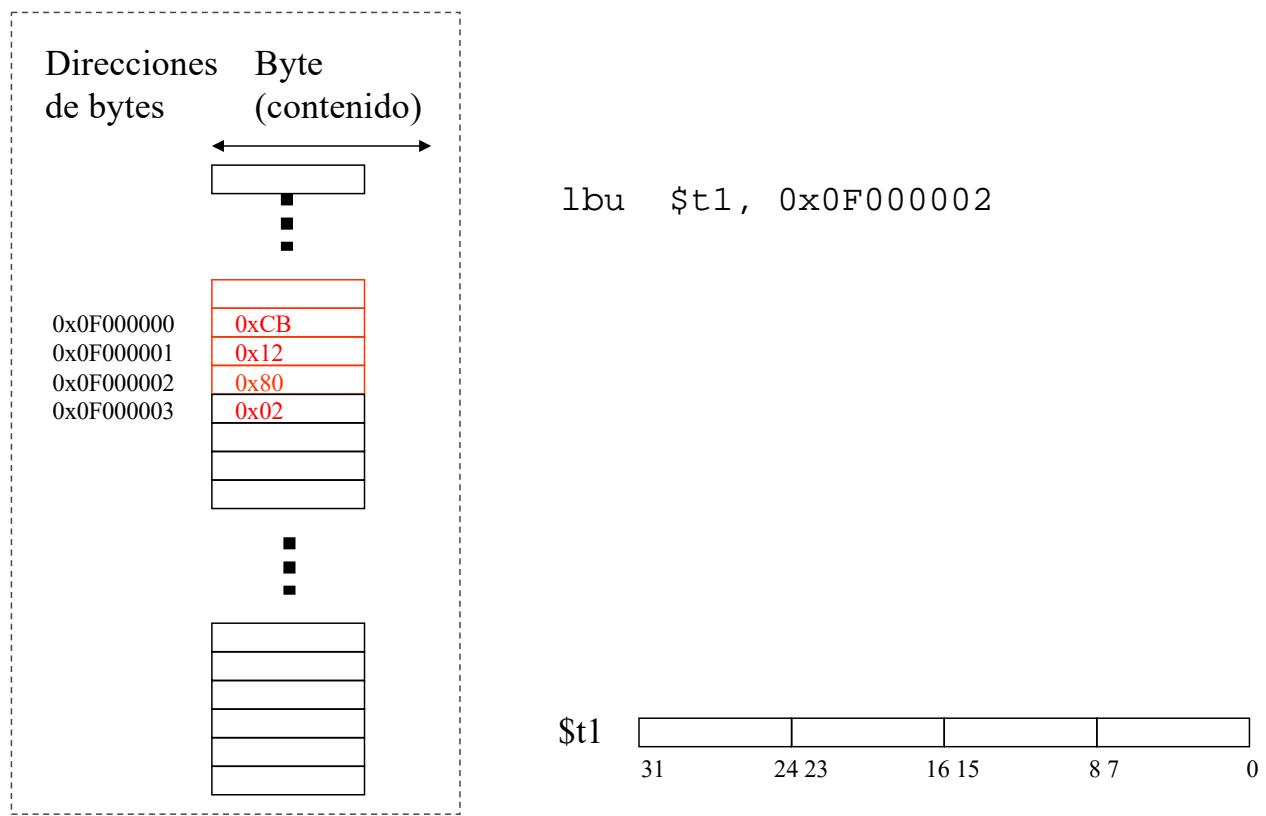
## Acceso a bytes con lbu (load byte unsigned)



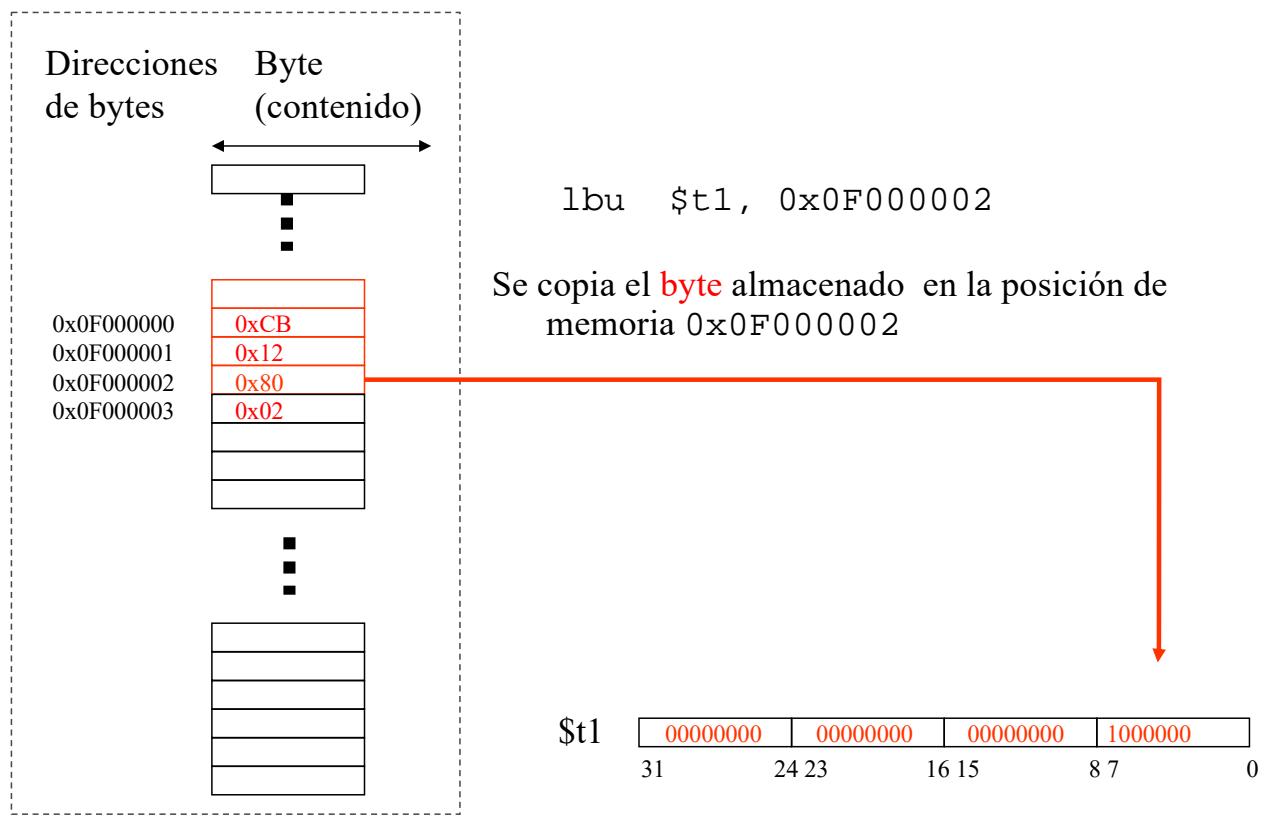
## Acceso a bytes con lbu (unsigned)



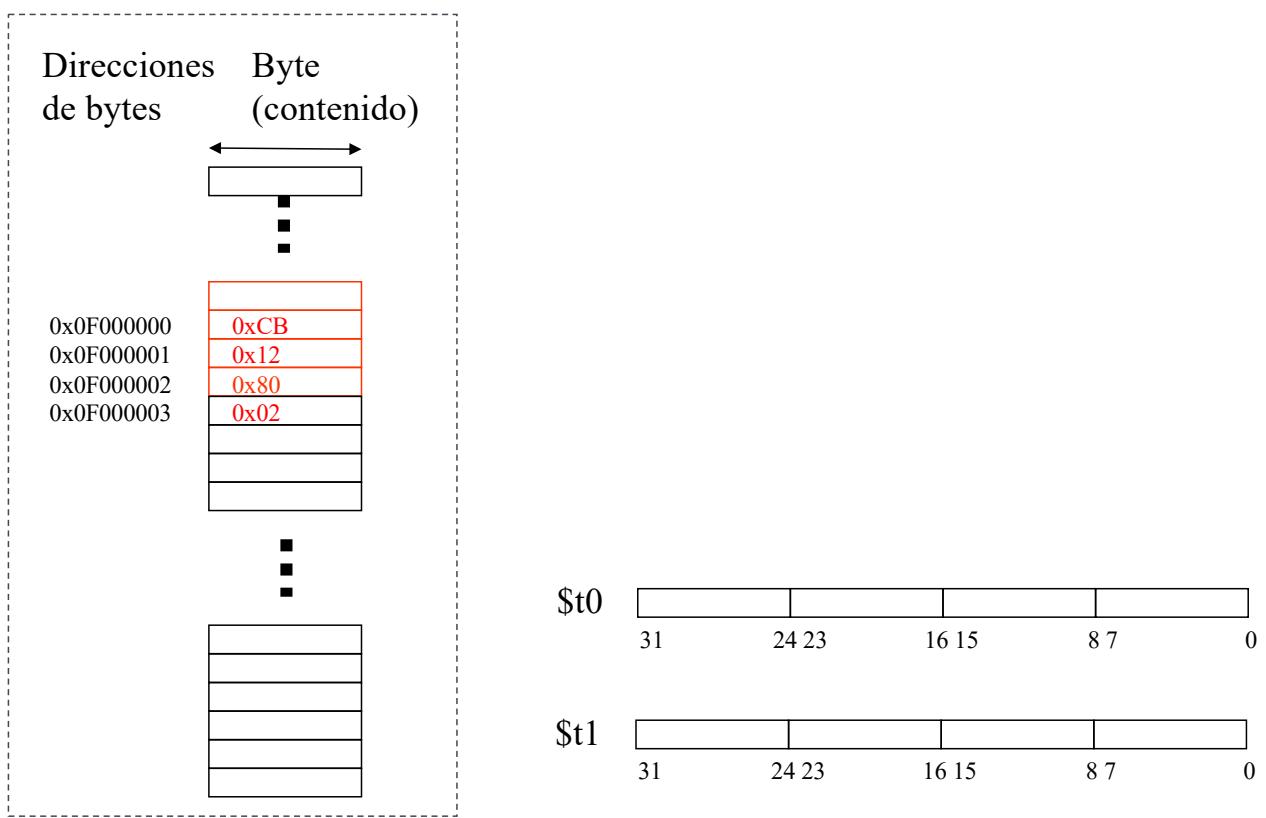
## Acceso a bytes con lbu (unsigned) No extiende el signo



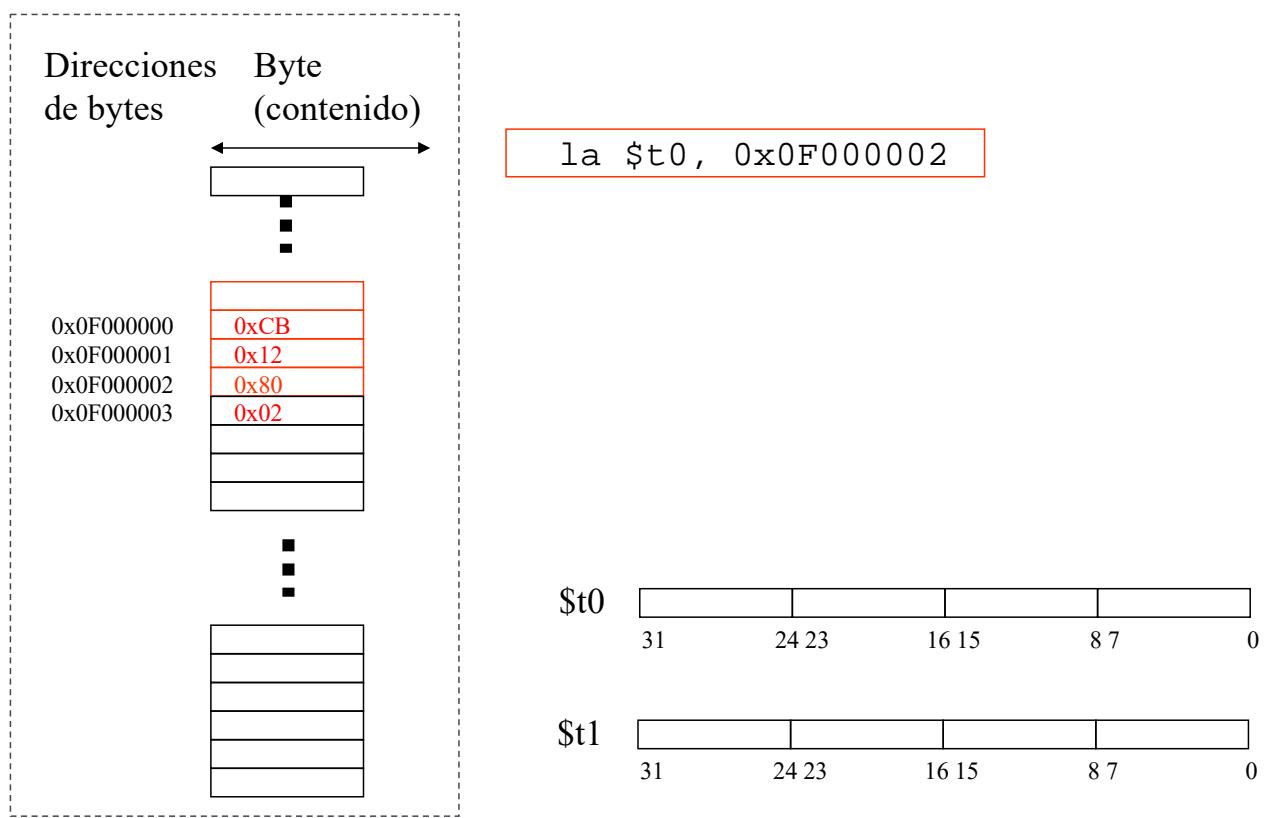
## Acceso a bytes con lbu (unsigned) No extiende el signo



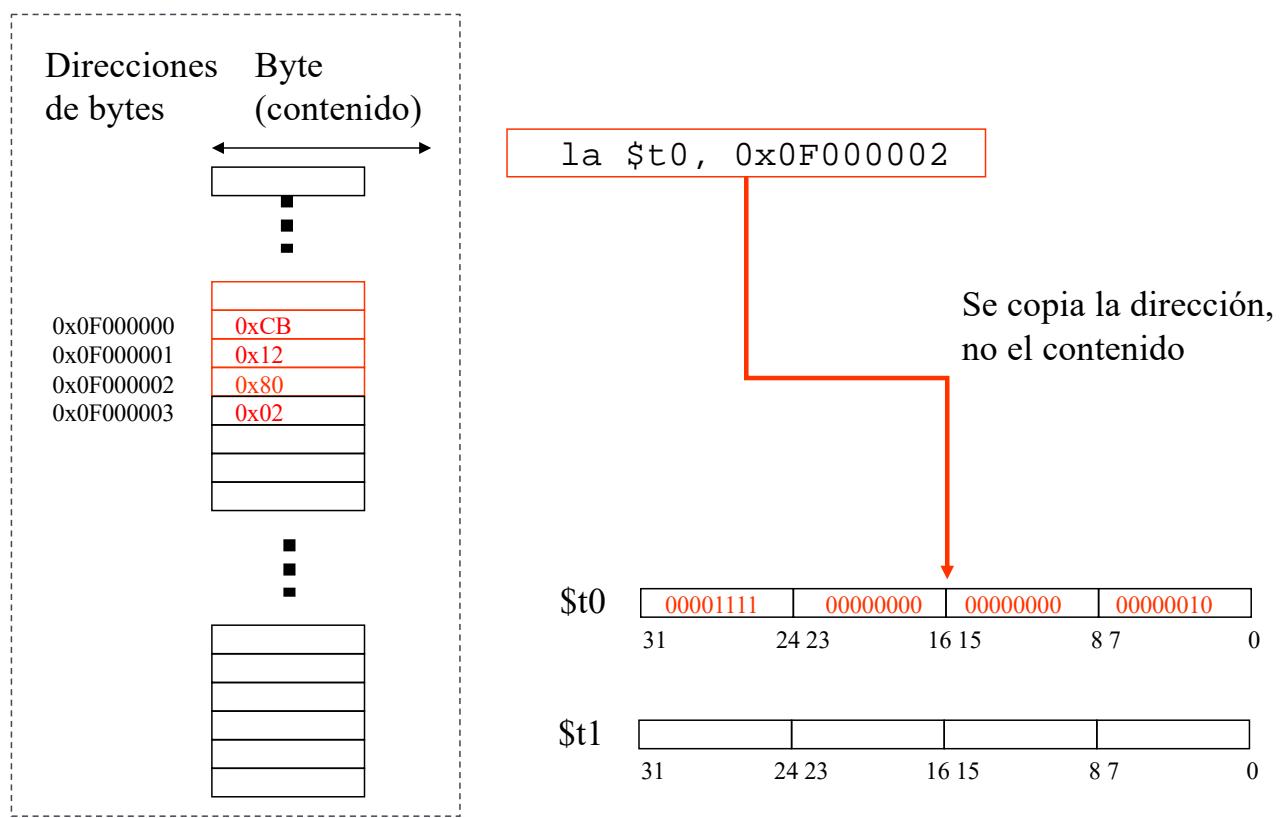
## Ejemplos de uso la (load address) y lbu



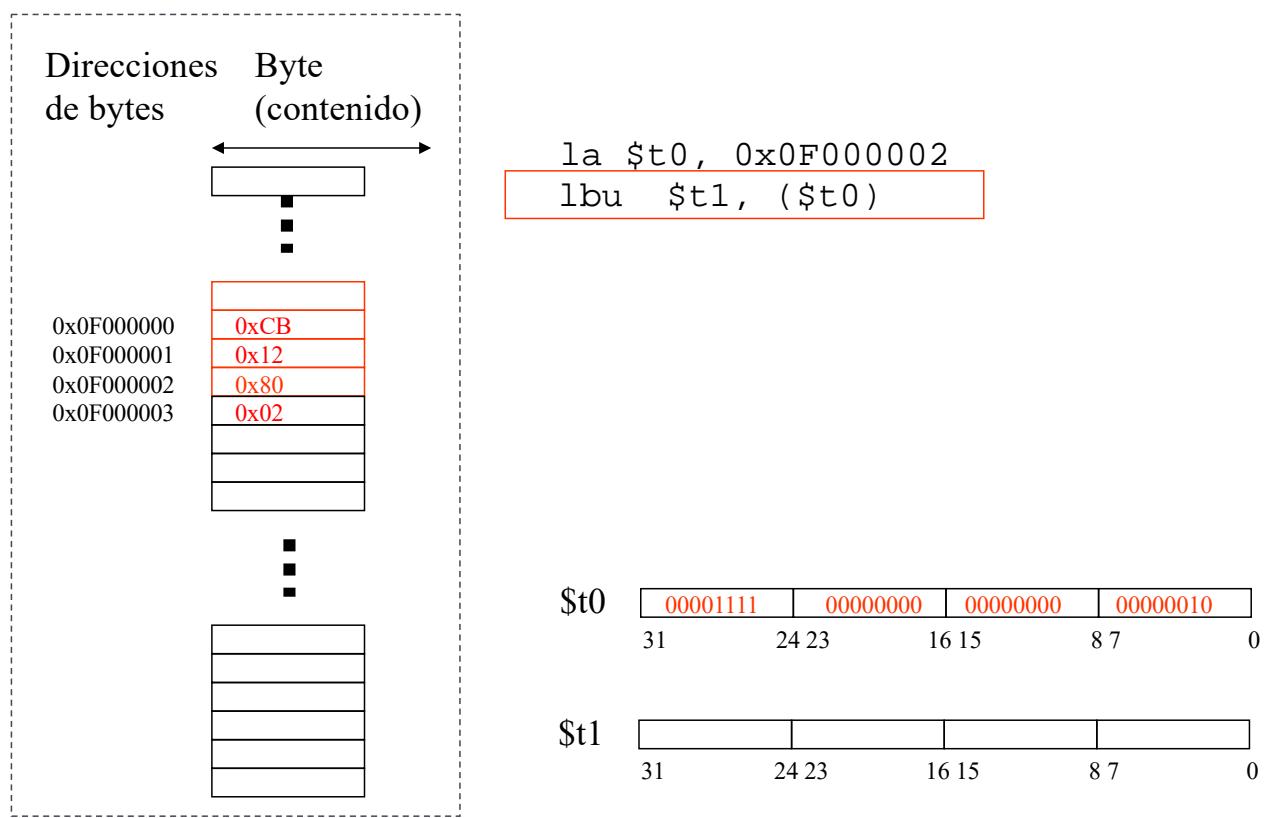
## Ejemplos de uso la y lbu



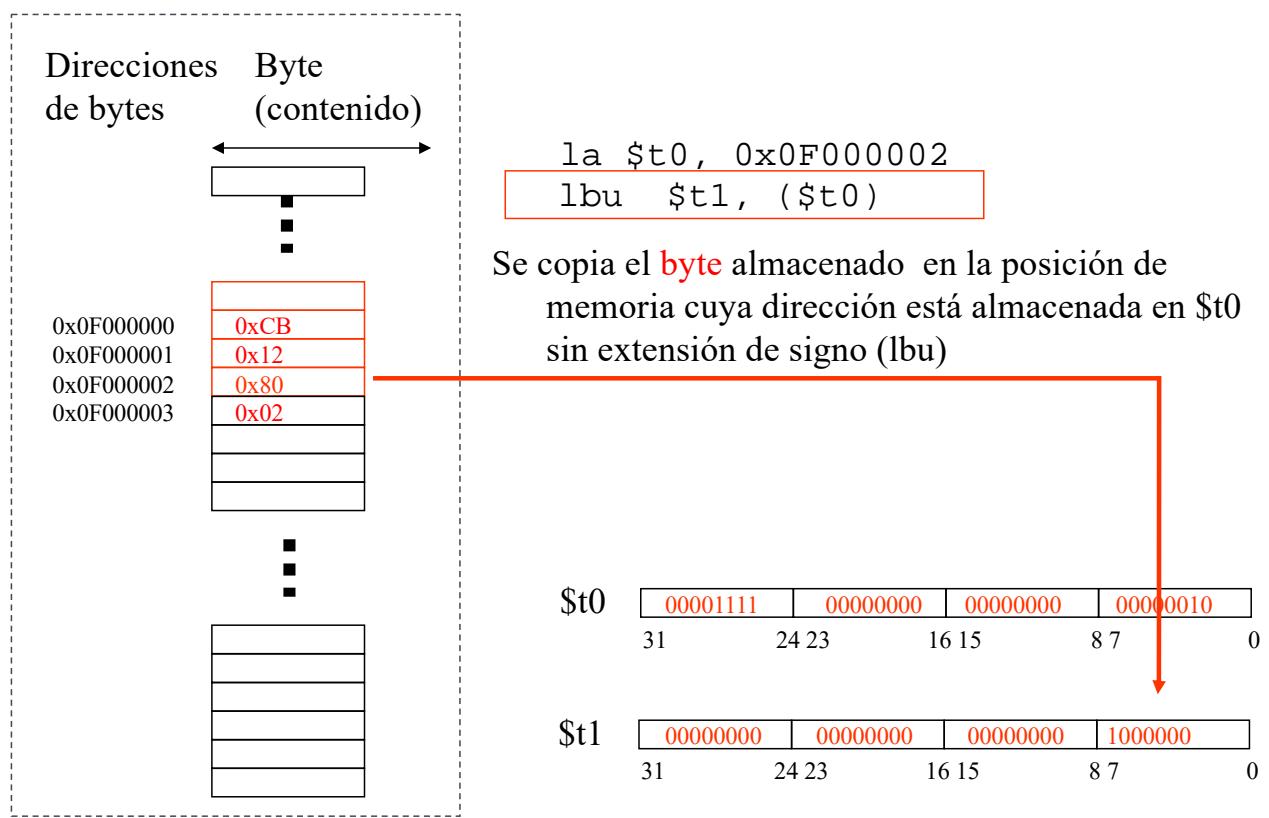
## Ejemplos de uso la y lbu



## Ejemplos de uso la y lbu



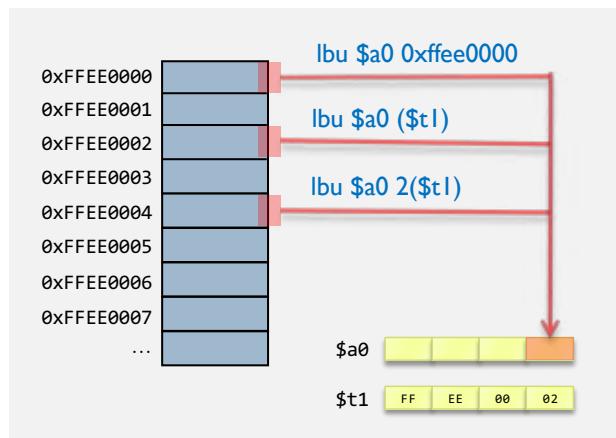
## Ejemplos de uso la y lbu



# Transferencia de datos

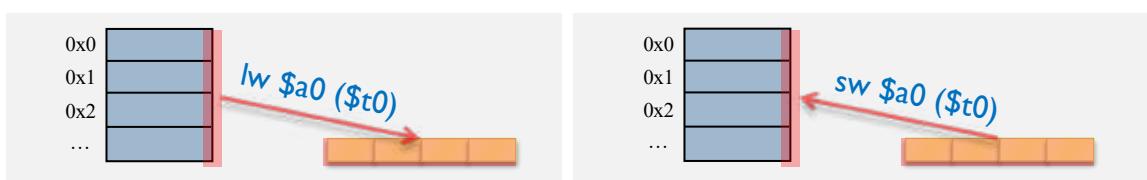
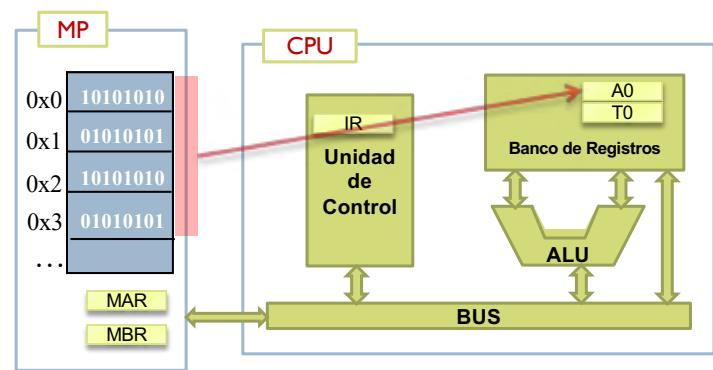
## Direccionamiento

- ▶ Hay tres posibilidades a la hora de indicar una posición de memoria:
- ▶ A) Directo:  
`Ibu $a0 0x0FFEE0000`
- ▶ B) Indirecto a registro:  
`Ibu $a0 ($t1)`
- ▶ C) Relativo a registro:  
`Ibu $a0 2($t1)`

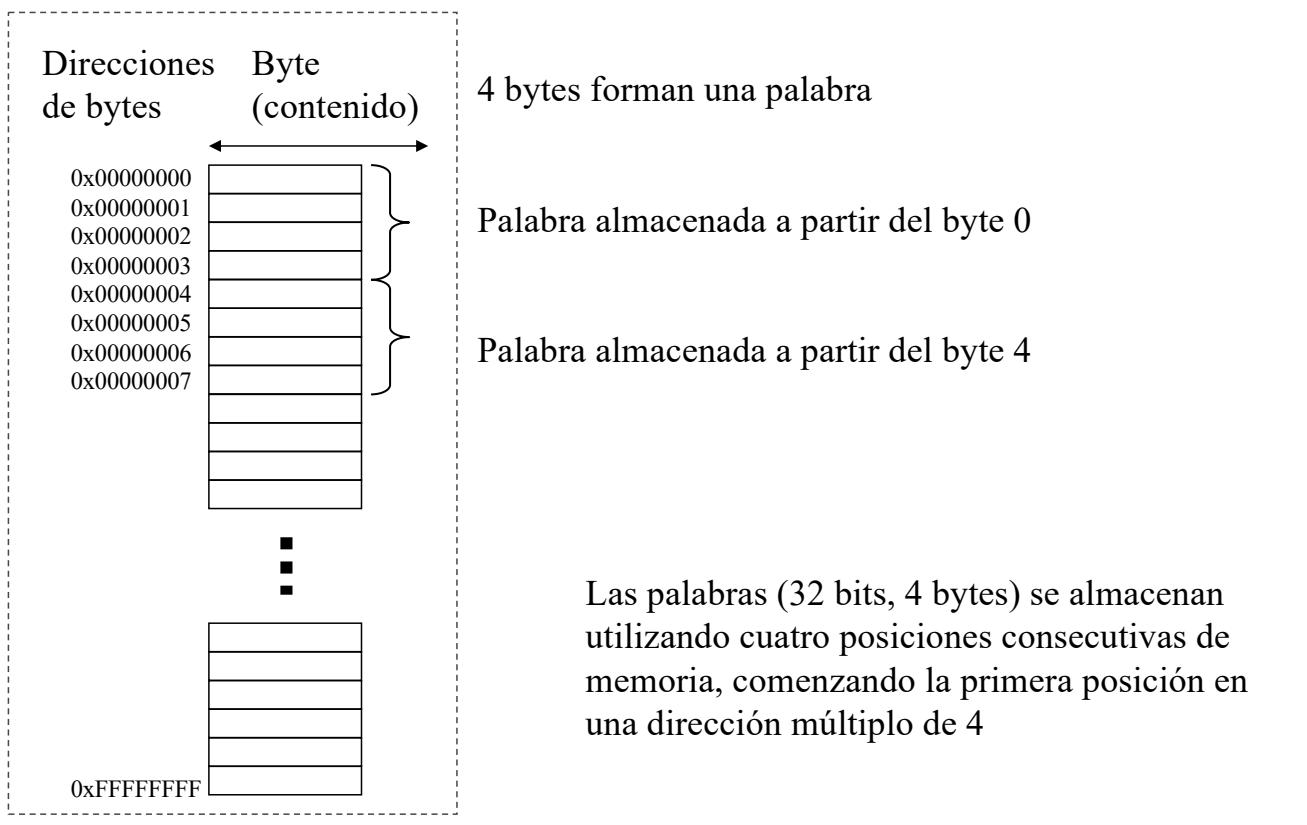


# Transferencia de datos palabras

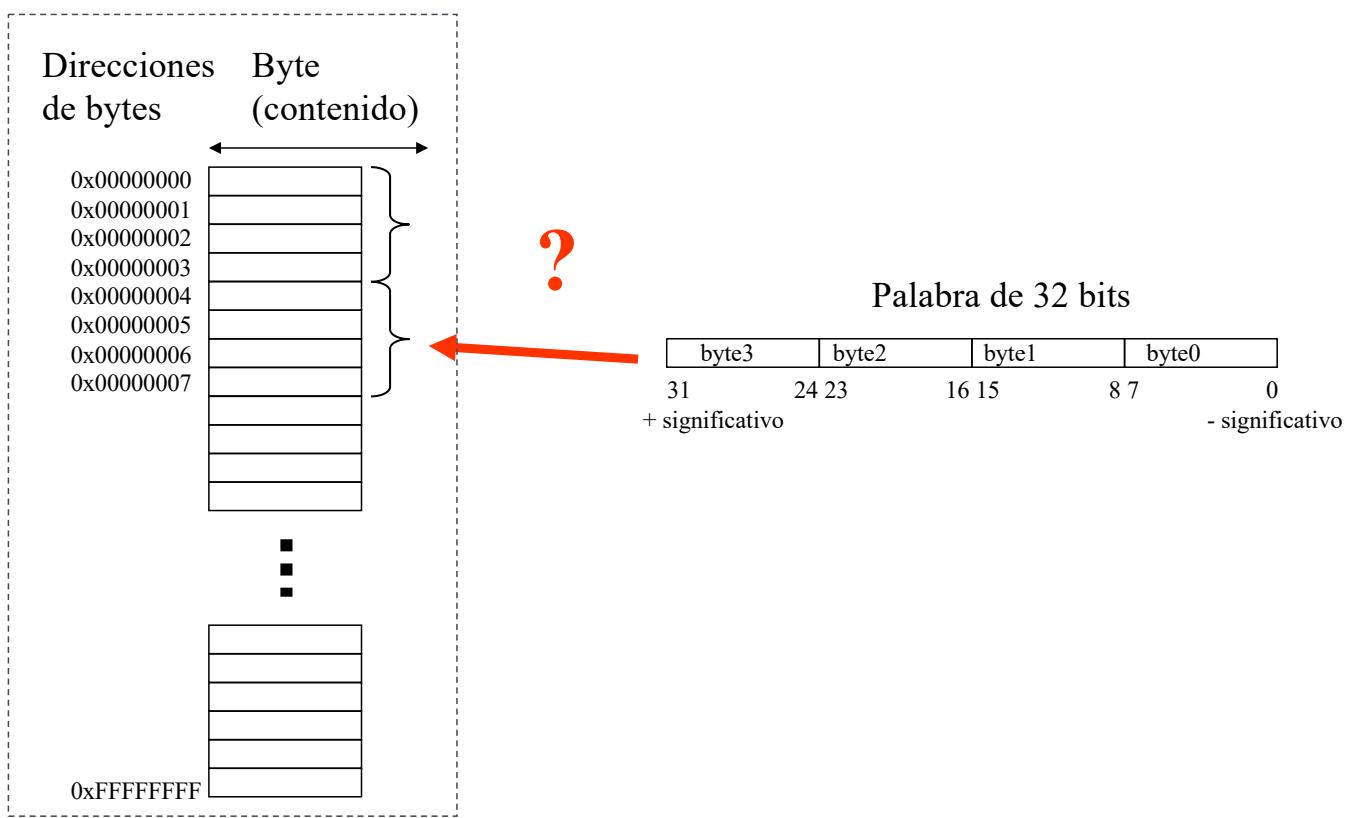
- ▶ Copia una palabra de memoria a un registro o viceversa
- ▶ Ejemplos:
  - ▶ Memoria a registro  
`lw $a0 ($t0)`
  - ▶ Registro a memoria  
`sw $a0 ($t0)`



# Acceso a palabras



# Acceso a palabras

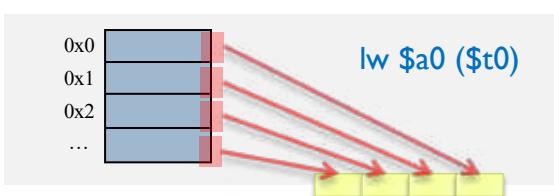


# Transferencia de datos ordenamiento de bytes

- ▶ Hay dos tipos de ordenamiento de bytes:

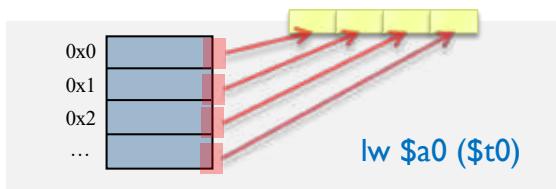
- ▶ Little-endian

(Dirección 'pequeña' termina la palabra...)



- ▶ Big-endian

(Dirección 'grande' termina la palabra...)



**IBM**  
**PowerPC™** (bi-endian)



# Almacenamiento de palabras en la memoria

Palabra de 32 bits

byte3	byte2	byte1	byte0
31	24 23	16 15	8 7 0
+ significativo			- significativo

A	byte3
A+1	byte2
A+2	byte1
A+3	byte0

BigEndian

A	byte0
A+1	byte1
A+2	byte2
A+3	byte3

LittleEndian

El número  $27_{(10)} = 11011_{(2)} = 000000000000000000000000000011011$

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

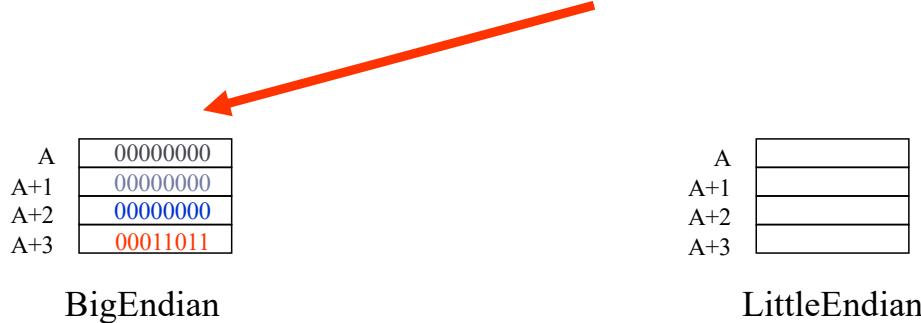
BigEndian

A	00011011
A+1	00000000
A+2	00000000
A+3	00000000

LittleEndian

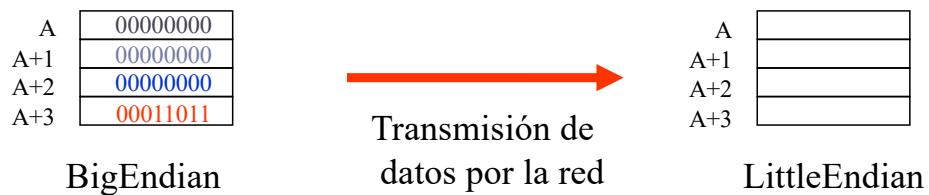
# Problemas en la comunicación entre computadores con arquitectura distinta

El número  $27_{(10)} = 11011_{(2)} = 0000000000000000\textcolor{blue}{000000000}\textcolor{red}{00011011}$



# Problemas en la comunicación entre computadores con arquitectura distinta

El número  $27_{(10)} = 11011_{(2)} = 0000000000000000\textcolor{blue}{000000000}\textcolor{red}{00011011}$



# Problemas en la comunicación entre computadores con arquitectura distinta

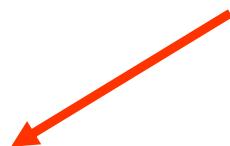
El número  $27_{(10)} = 11011_{(2)} = 0000000000000000\textcolor{blue}{00000000}\textcolor{red}{00011011}$

A	00000000
A+1	00000000
A+2	<b>00000000</b>
A+3	<b>00011011</b>

BigEndian

A	00000000
A+1	00000000
A+2	<b>00000000</b>
A+3	<b>00011011</b>

LittleEndian



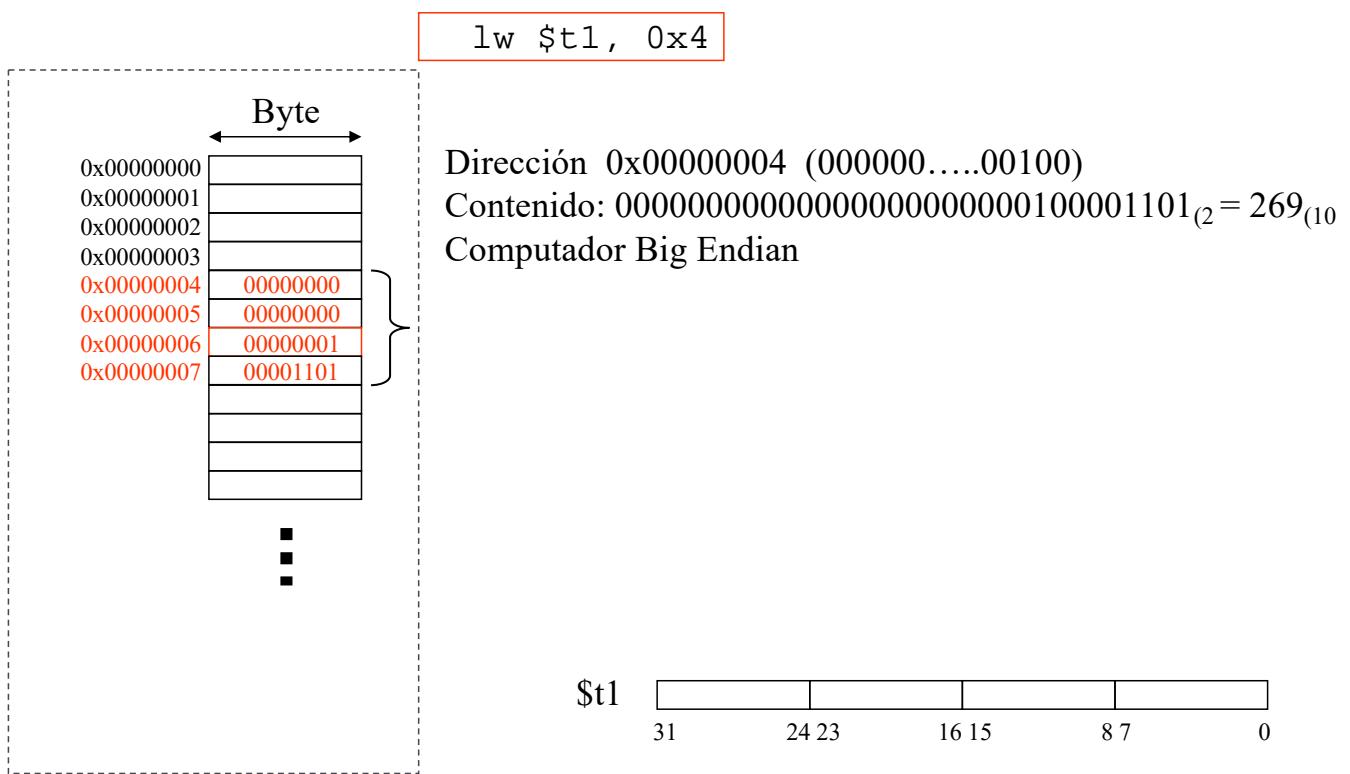
El número almacenado es: **00011011000000000000000000000000**  
que no es el 27

# Ejemplo

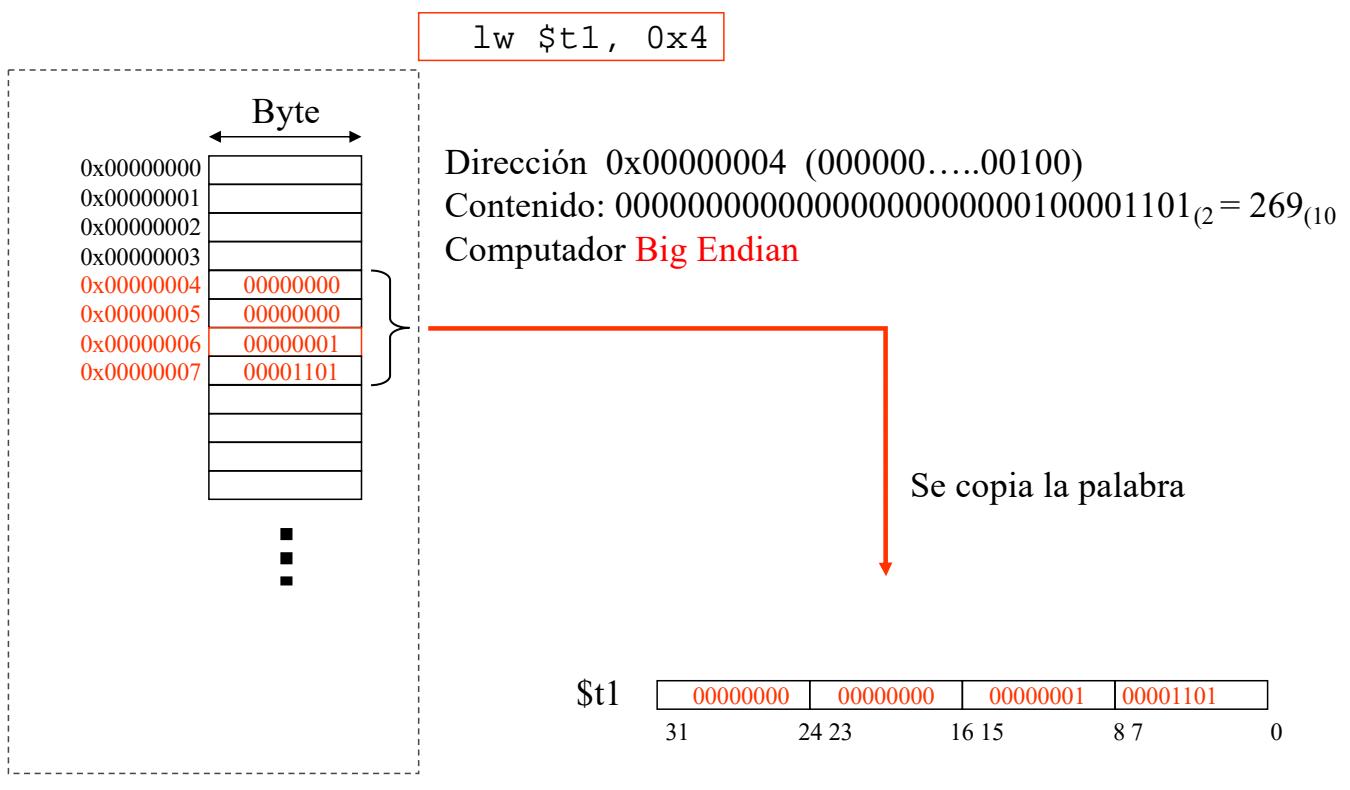
endian.s

```
.data  
  
b1: .byte 0x00, 0x11, 0x22, 0x33  
  
.text  
.globl main  
main:  
  
lw $t0 b1
```

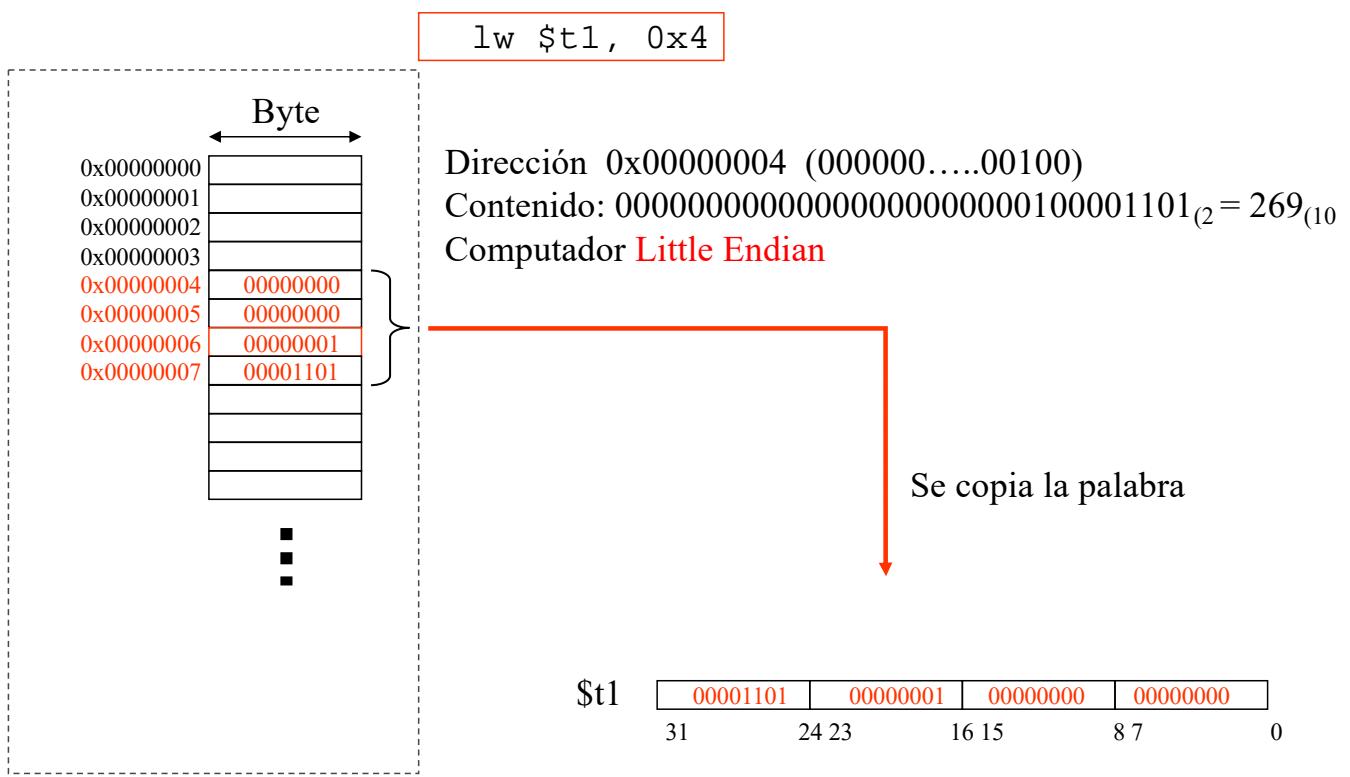
# Acceso a palabras



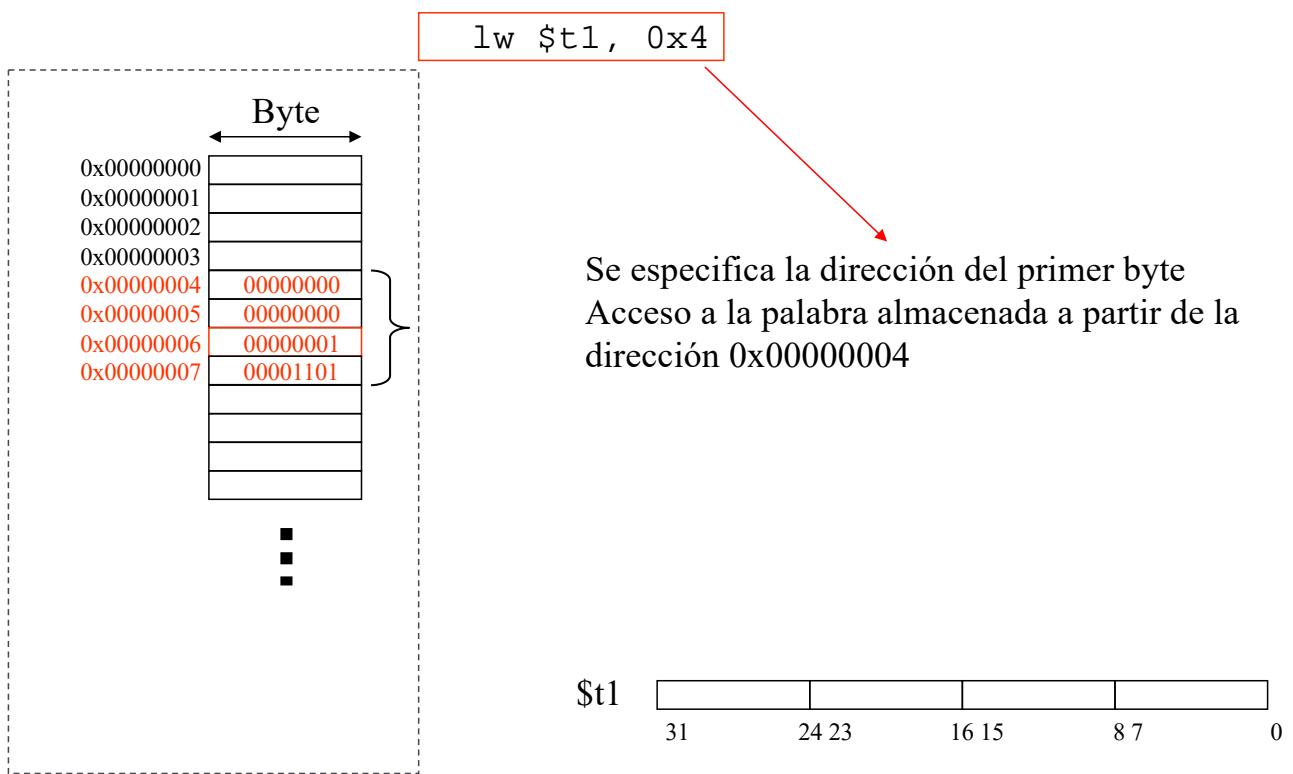
# Acceso a palabras



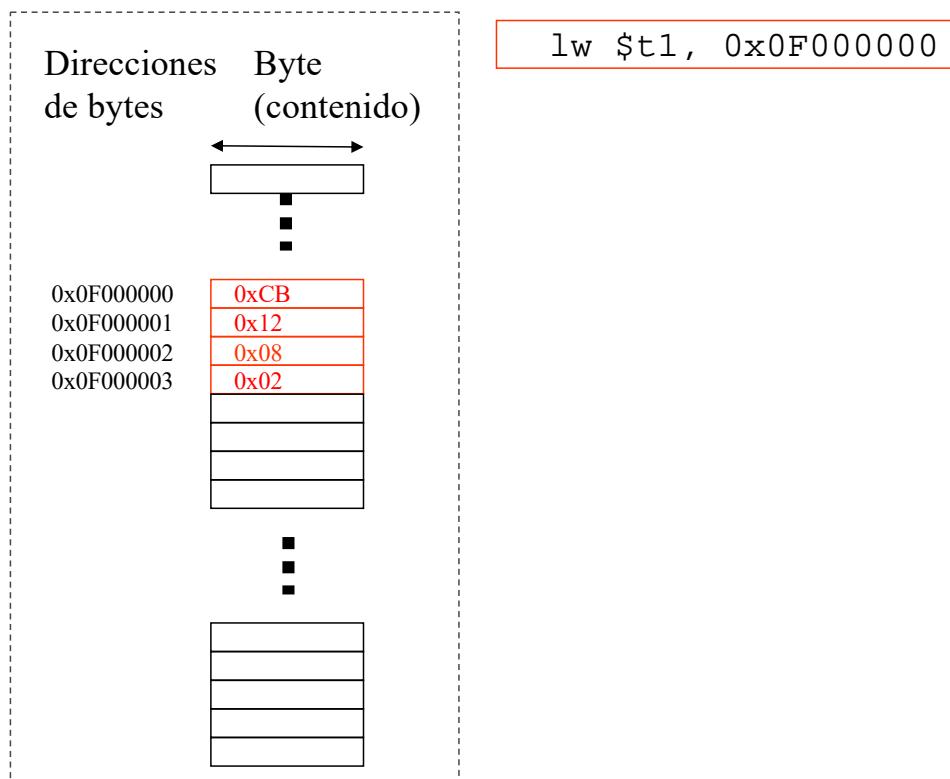
# Acceso a palabras



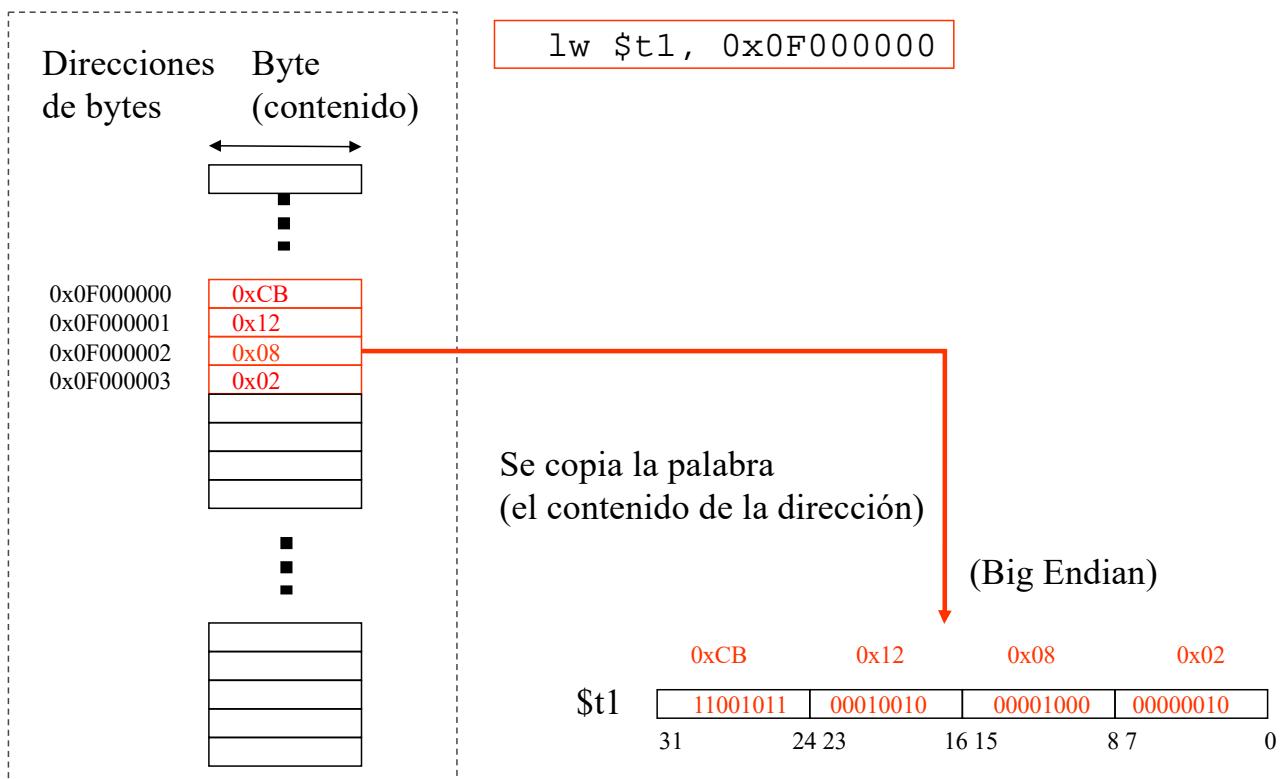
# Acceso a palabras



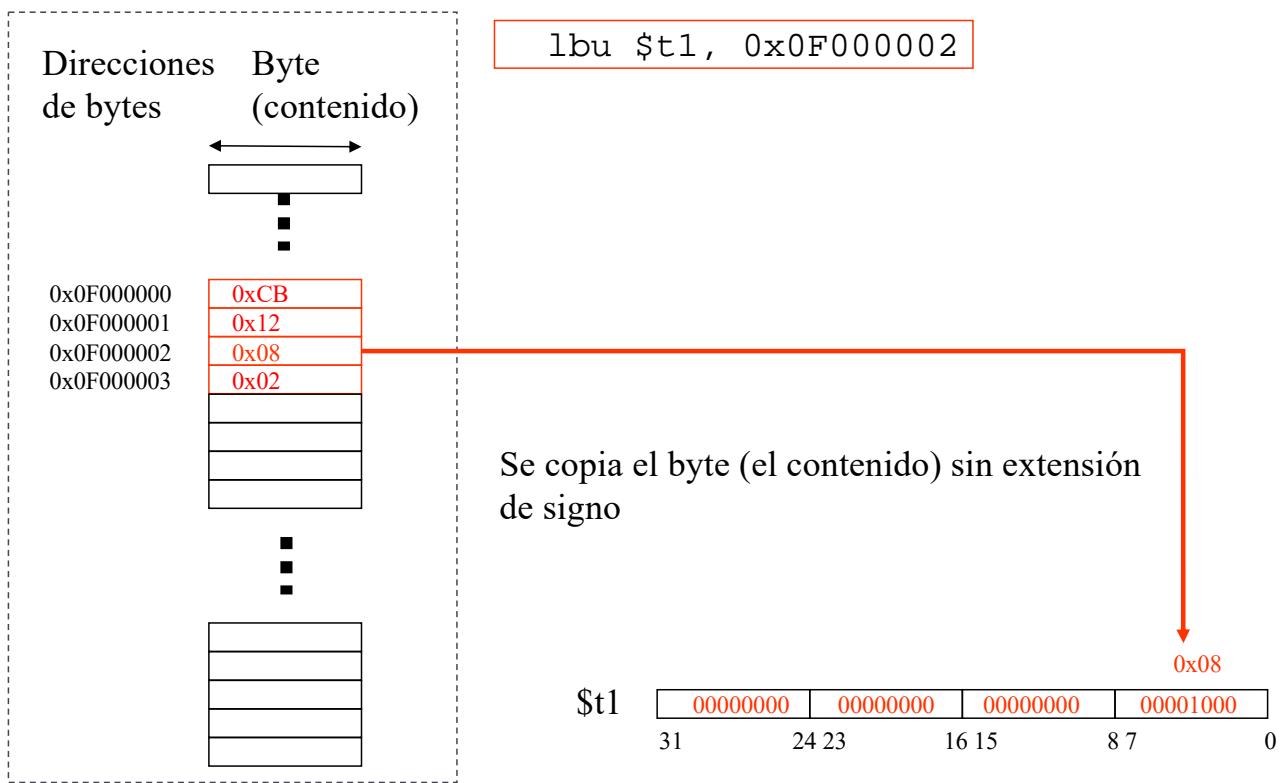
# Diferencias entre lw, lb, lbu, la



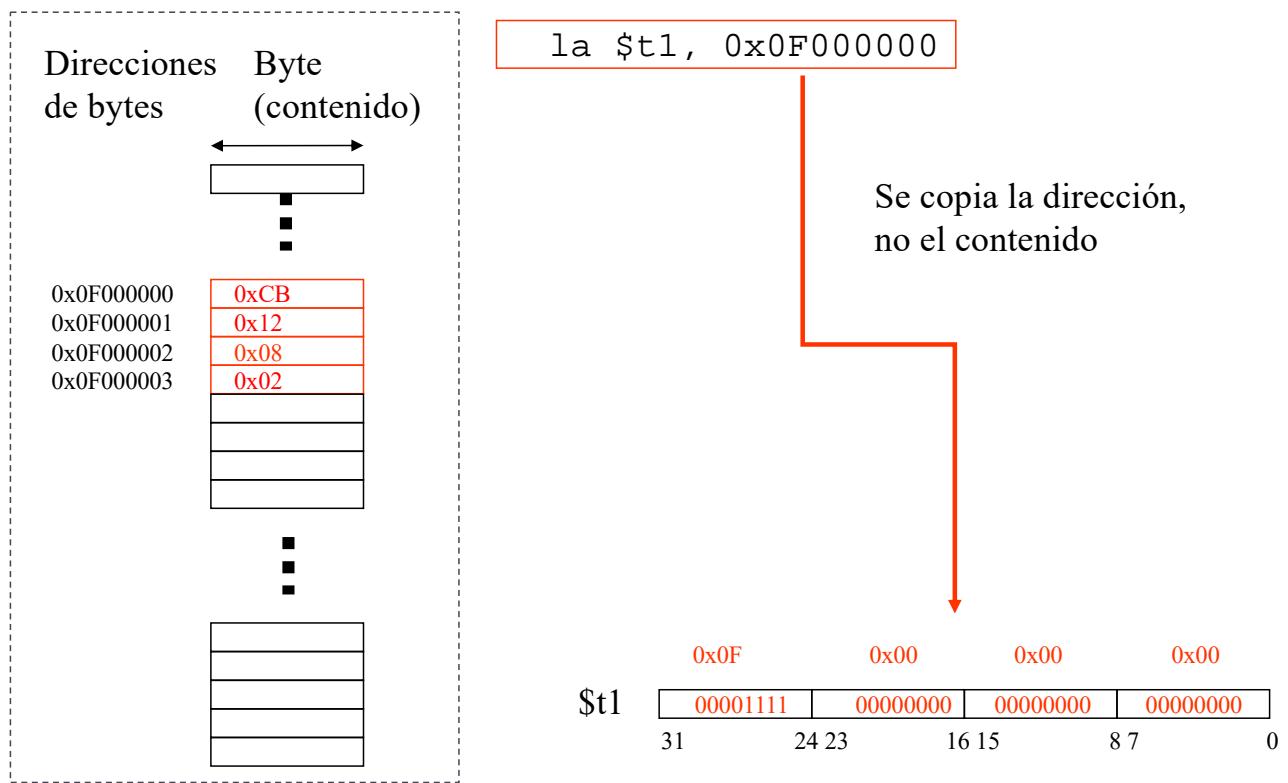
## Diferencias entre lw, lb, lbu, la



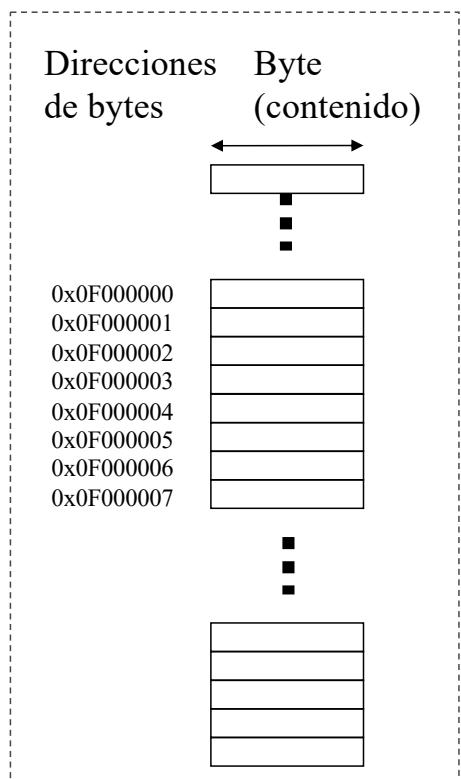
## Diferencias entre lw, lb, lbu, la



## Diferencias entre lw, lb, lbu, la

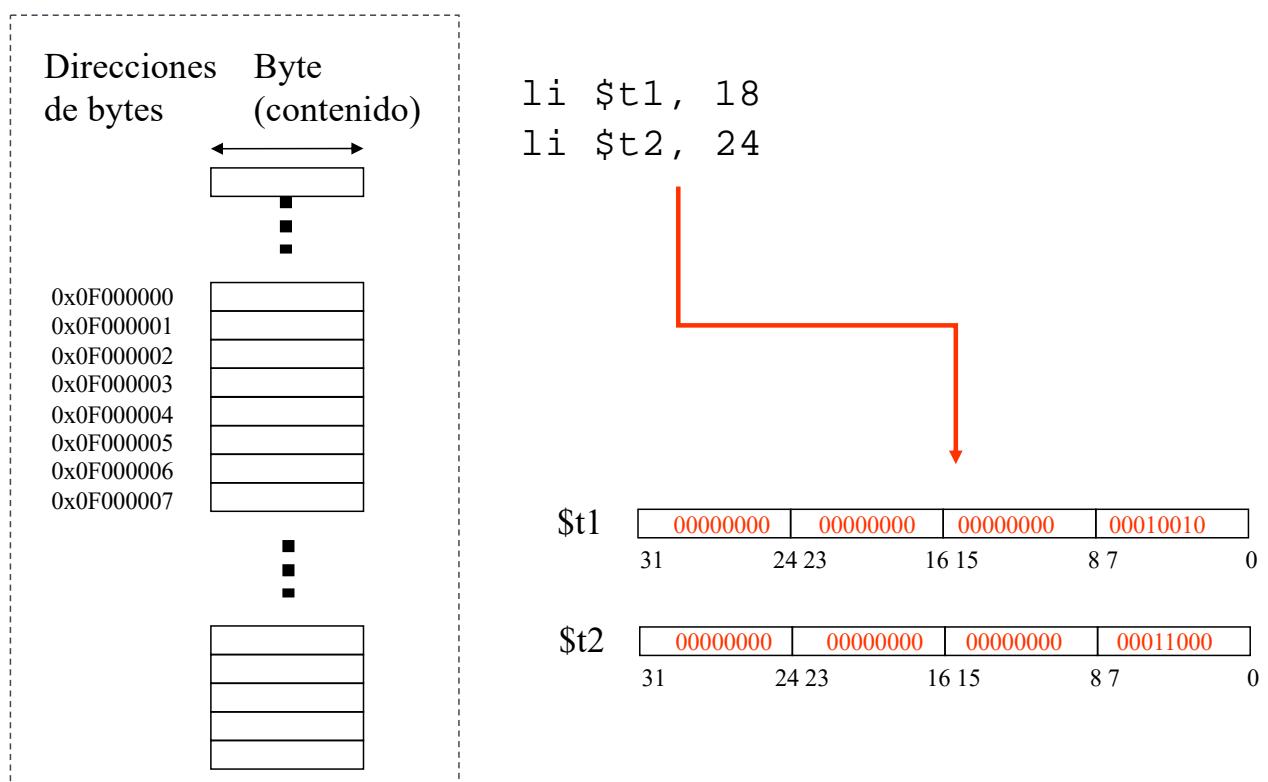


# Ejemplo

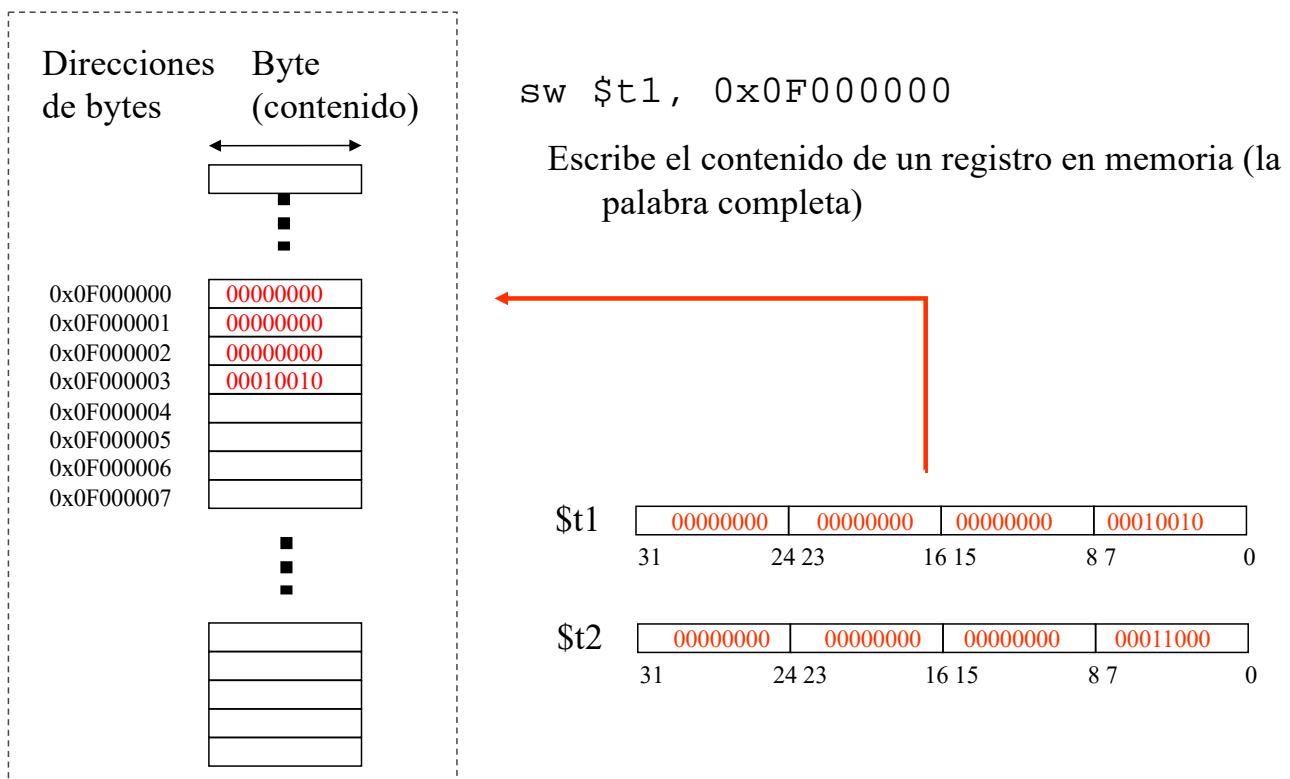


```
li $t1, 18  
li $t2, 24
```

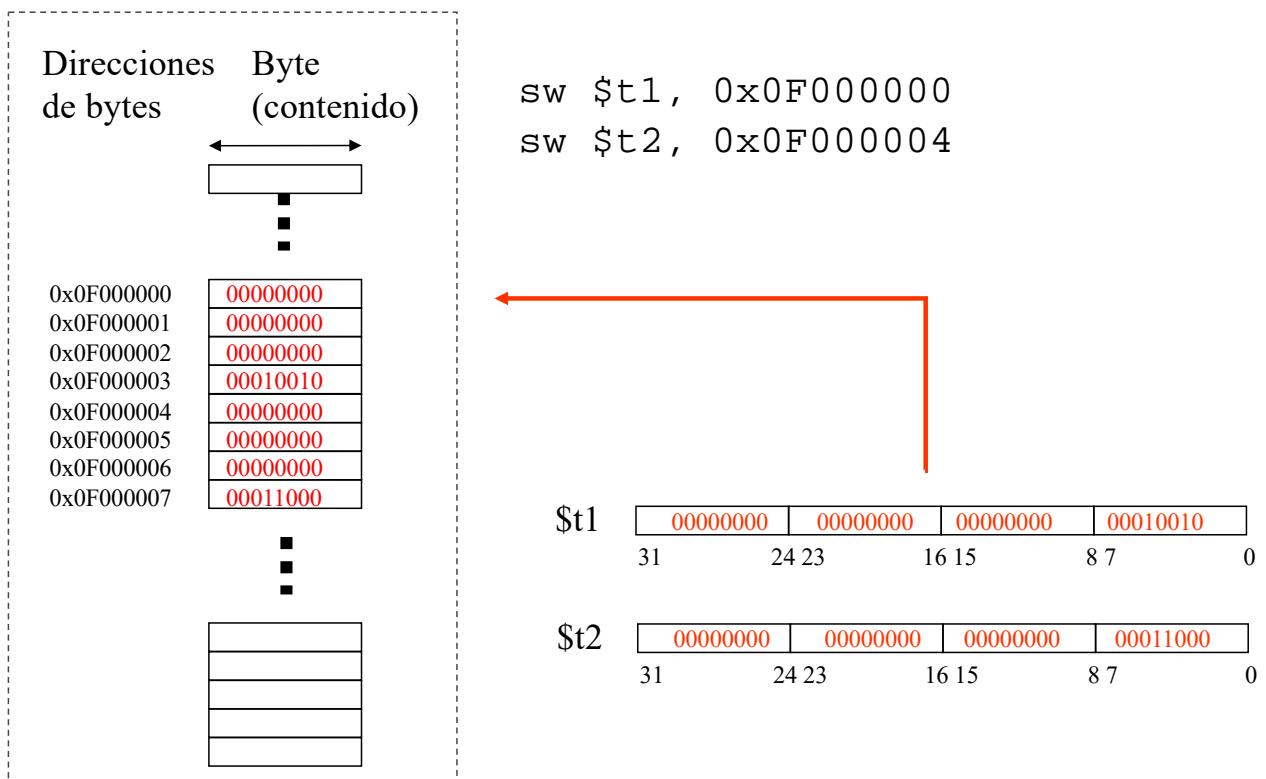
# Ejemplo



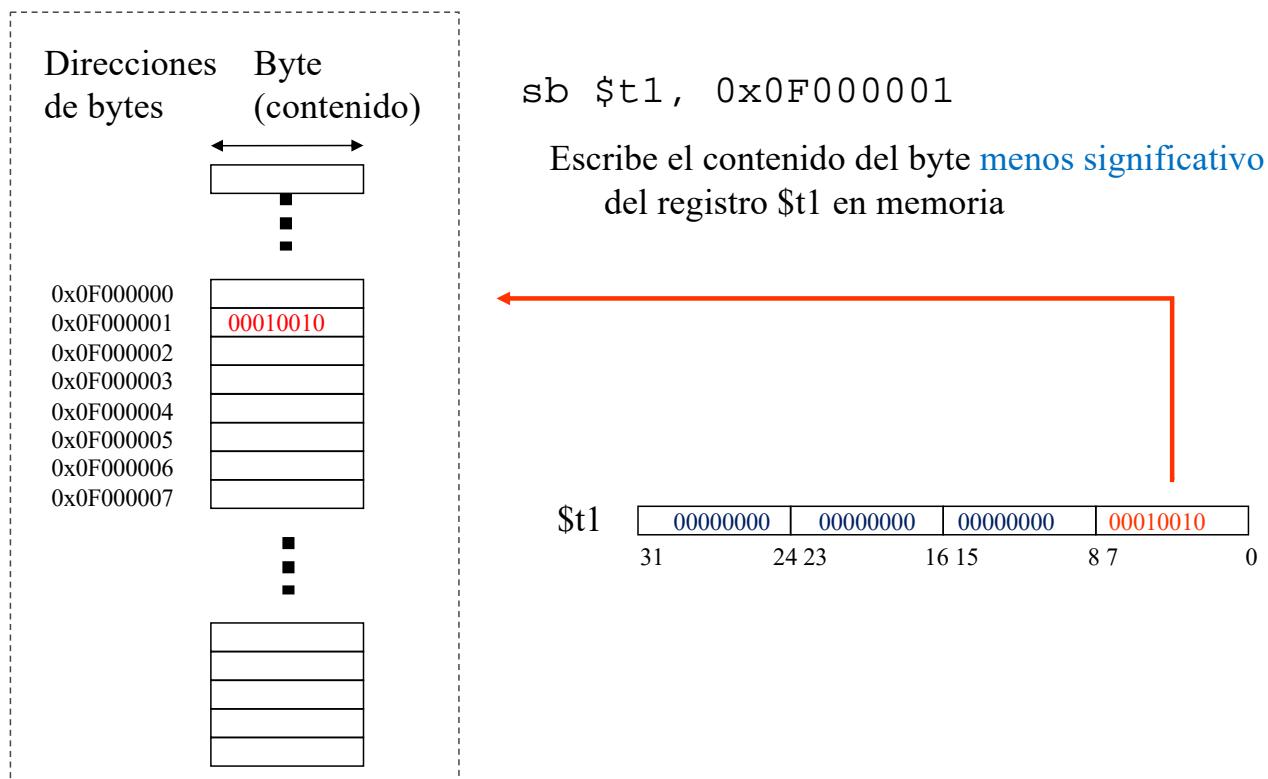
# Escritura en memoria de una palabra



# Escritura en memoria de una palabra

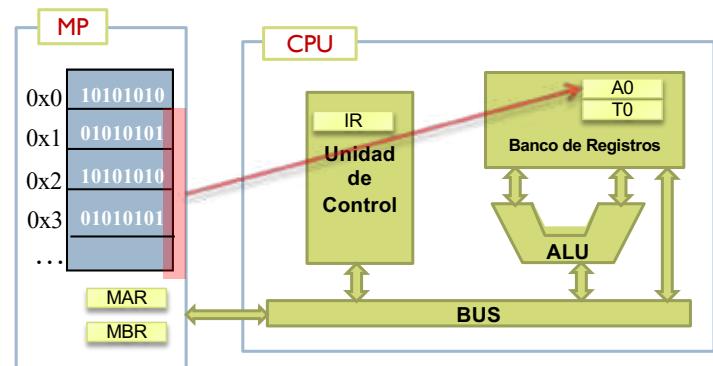


# Escritura en memoria de un byte



# Transferencia de datos alineamiento y tamaño de acceso

- ▶ Peculiaridades:
  - ▶ Alineamiento de los elementos en memoria
  - ▶ Tamaño de acceso por defecto



# Alineación de datos

## ▶ En general:

- ▶ Un dato que ocupa K bytes está alineado cuando la dirección D utilizada para accederlo cumple que:

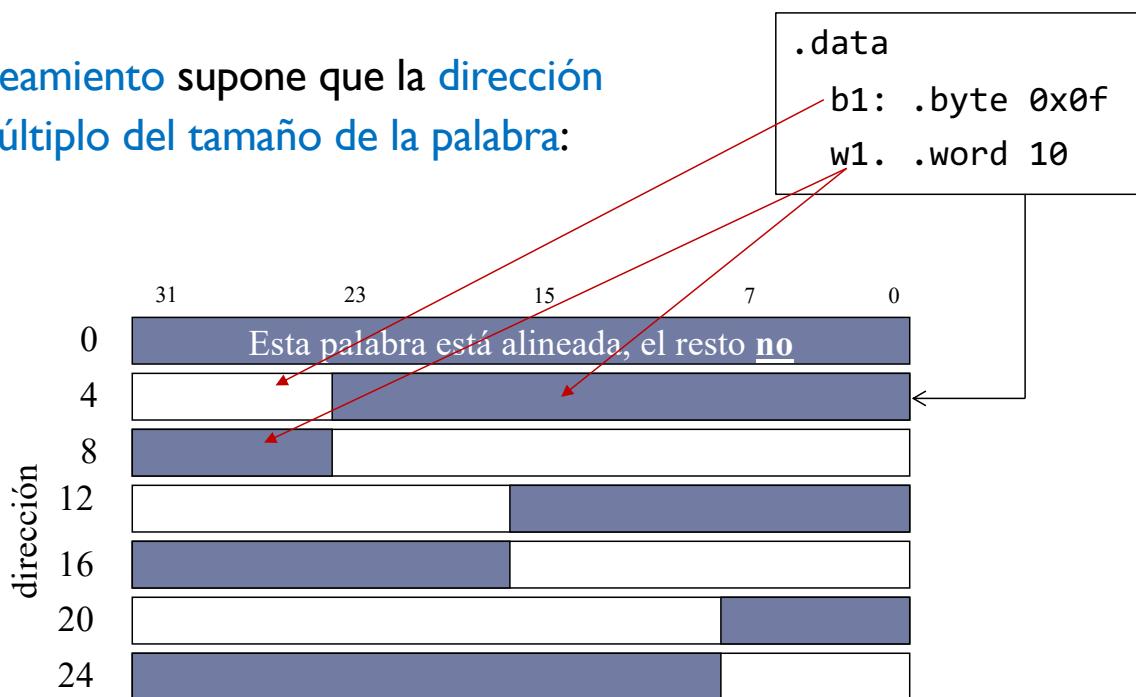
$$D \bmod K = 0$$

## ▶ La alineación supone que:

- ▶ Los datos que ocupan 2 bytes se encuentran en direcciones pares
- ▶ Los datos que ocupan 4 bytes se encuentran en direcciones múltiplo de 4
- ▶ Los datos que ocupan 8 bytes (double) se encuentran en direcciones múltiplo de 8

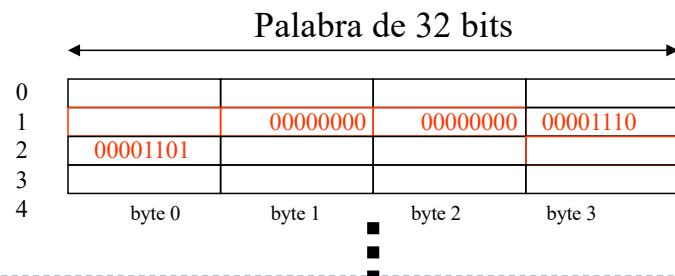
## Alineamiento

- El **alineamiento** supone que la **dirección** sea **múltiplo del tamaño de la palabra**:



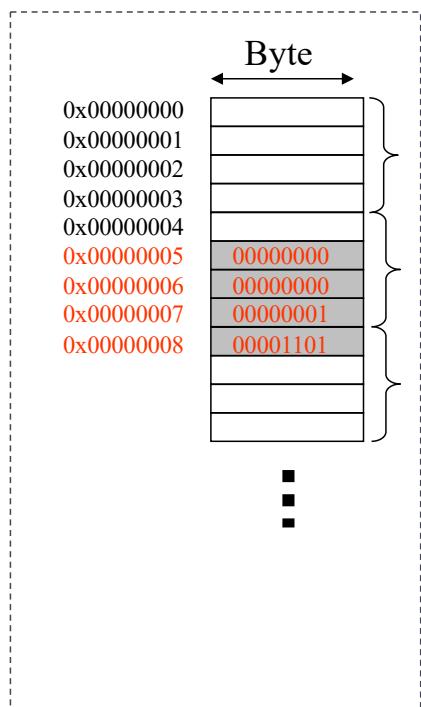
# Alineación de datos

- ▶ En general los computadores no permiten el acceso a datos no alineados
  - ▶ Objetivo: minimizar el número de accesos a memoria
  - ▶ El compilador se encarga de asignar a los datos las direcciones adecuadas
- ▶ Algunas arquitecturas como Intel permiten el acceso a datos no alineados
  - ▶ El acceso a un dato no alineado implica varios accesos a memoria



# Datos no alineados

lw \$t1, 0x05 ????

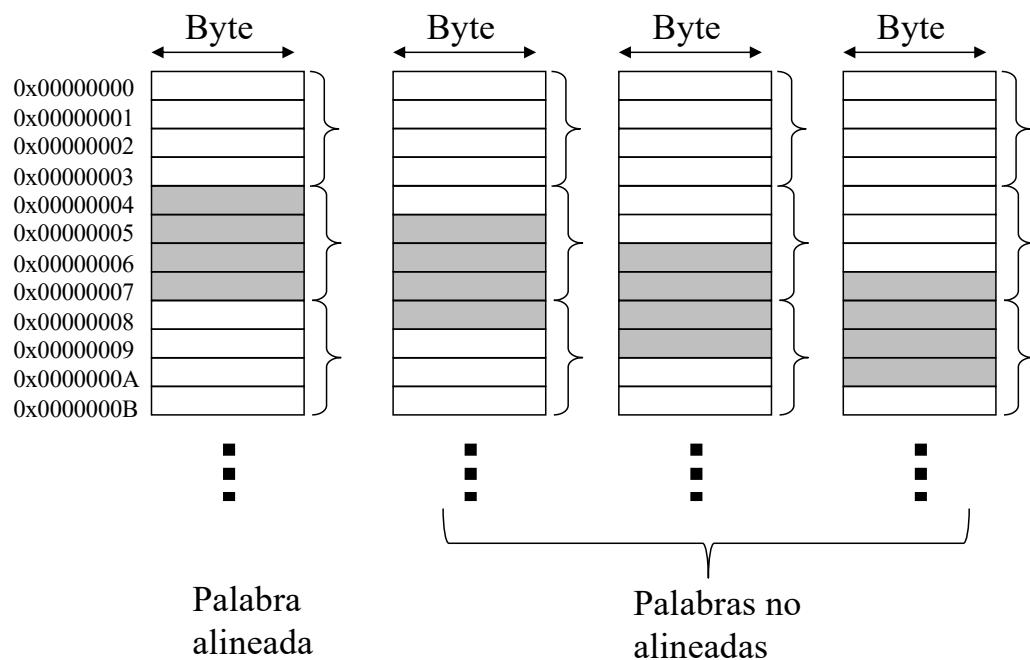


Palabras de memoria

La palabra que está almacenada a partir de la dirección 0x05 **no está alineada** porque se encuentra en dos palabras de memoria distintas

Una palabra tiene que almacenarse a partir de una dirección múltiplo de 4

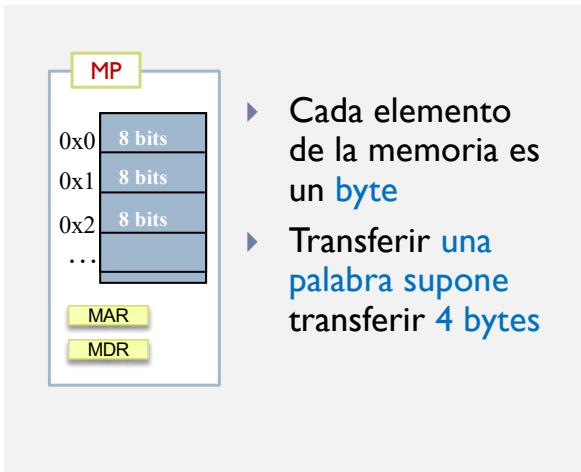
## Datos no alineados



# Direccionamiento a nivel de palabra o de byte

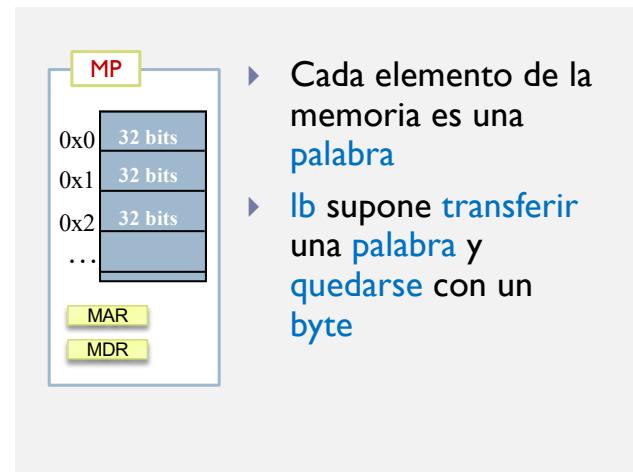
- ▶ La memoria principal es similar a un gran vector de una dimensión
- ▶ Una dirección de memoria es el índice del vector
- ▶ Hay dos tipos de direccionamiento:

- ▶ Direccionamiento por bytes



- ▶ Cada elemento de la memoria es un byte
- ▶ Transferir una palabra supone transferir 4 bytes

- ▶ Direccionamiento por palabras



- ▶ Cada elemento de la memoria es una palabra
- ▶ I<sub>b</sub> supone transferir una palabra y quedarse con un byte

# Resumen

- ▶ Un programa para poder ejecutarse debe estar cargado junto con sus datos en memoria
- ▶ Todas las instrucciones y los datos se almacenan en memoria, por tanto todo tiene una dirección de memoria
  - ▶ Las instrucciones y los datos
- ▶ En un computador como el MIPS 32 (de 32 bits)
  - ▶ Los registros son de 32 bits
  - ▶ En la memoria se pueden almacenar bytes (8 bits)
    - ▶ Instrucciones memoria → registro: lb, lbu, sb
    - ▶ Instrucciones registro → memoria: sb
  - ▶ En la memoria se pueden almacenar palabras (32 bits)
    - ▶ Instrucción memoria → registro: lw
    - ▶ Instrucción registro → memoria: sw

## Formatos de las instrucciones de acceso a memoria

lw  
sw  
lb  
sb  
lbu

Registro, dirección de memoria

- Número que representa una dirección
- Etiqueta simbólica que representa una dirección
- (registro): representa la dirección almacenada en el registro
- num(registro): representa la dirección que se obtiene de sumar num con la dirección almacenada en el registro

## Formatos de las instrucciones de acceso a memoria

- ▶ `lbu $t0, 0x0F000002`
  - ▶ Direccionamiento directo. Se carga en \$t0 el byte almacenado en la posición de memoria 0x0F000002
- ▶ `lbu $t0, etiqueta`
  - ▶ Direccionamiento directo. Se carga en \$t0 el byte almacenado en la posición de memoria etiqueta
- ▶ `lbu $t0, ($t1)`
  - ▶ Direccionamiento indirecto de registro. Se carga en \$t0 el byte almacenado en la posición de memoria almacenada en \$t1
- ▶ `lbu $t0, 80($t1)`
  - ▶ Direccionamiento relativo. Se carga en \$t0 el byte almacenado en la posición de memoria que se obtiene de sumar el contenido de \$t1 con 80

## Instrucciones de escritura en memoria

- ▶ `sw $t0, 0x0F000000`
  - ▶ Copia la palabra almacenada en `$t0` en la dirección `0x0F000000`
  
- ▶ `sb $t0, 0x0F000000`
  - ▶ Copia el byte almacenado en `$t0` (el menos significativo) en la dirección `0x0F000000`

# Tipos de datos en ensamblador

- ▶ Booleanos
- ▶ Caracteres
- ▶ Enteros
- ▶ Reales
- ▶ Vectores
- ▶ Cadenas de caracteres
- ▶ Matrices
- ▶ Otras estructuras

## Tipos de datos booleanos

```
bool_t b1 = false;  
bool_t b2 = true;  
...
```

```
main ()  
{  
    b1 = true ;  
    ...  
}
```

```
.data  
b1: .byte 0          # 1 byte  
b2: .byte 1  
...
```

```
.text  
.globl main  
main: la $t0 b1  
      li $t1 1  
      sb $t1 ($t0)  
...
```

## Tipos de datos caracteres

```
char c1 ;  
char c2 = 'a';  
...
```

```
main ()  
{  
    c1 = c2;  
    ...  
}
```

```
.data  
c1: .space 1      # 1 byte  
c2: .byte 'a'  
...
```

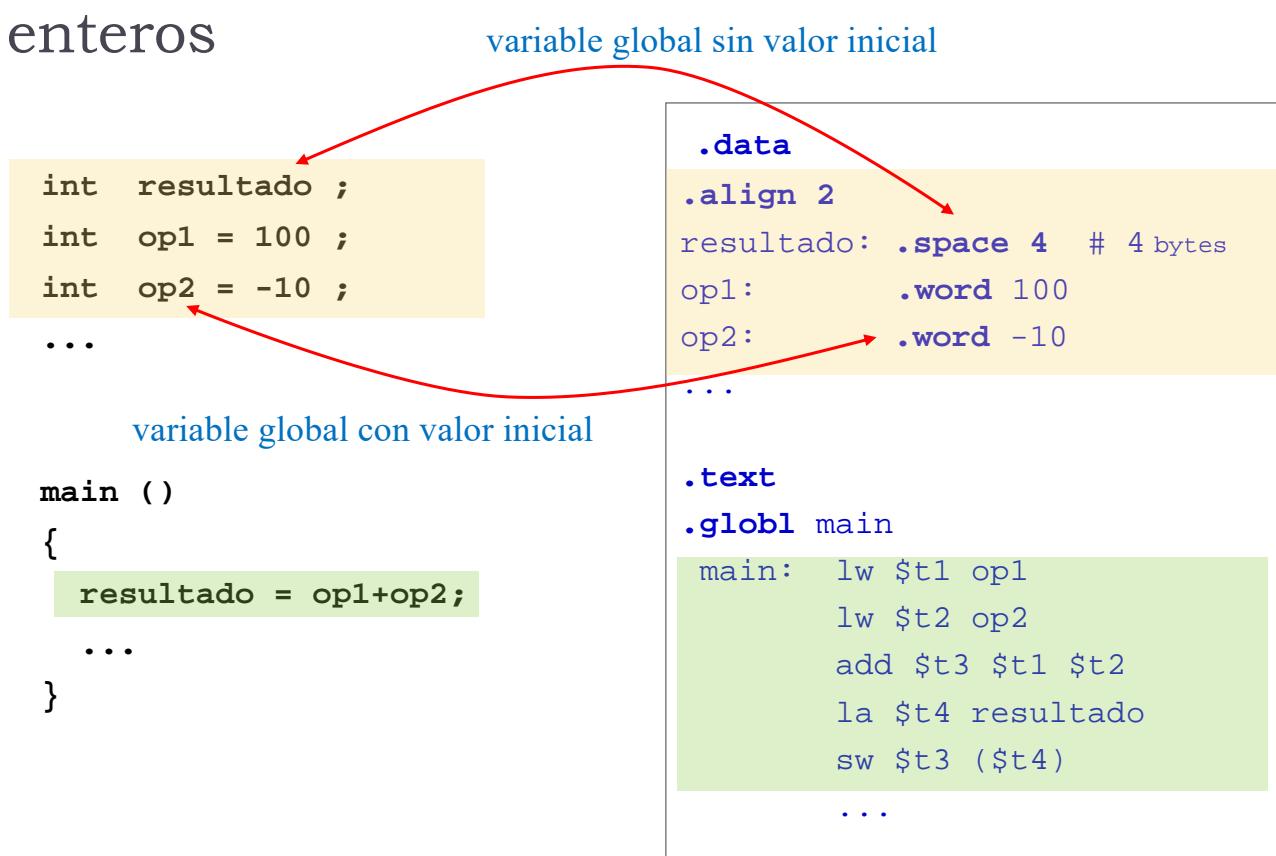
```
.text  
.globl main  
main: la $t0 c2  
      lbu $t1 c1  
      sb $t1 ($t0)  
...
```

## Tipos de datos enteros

```
int resultado ;
int op1 = 100 ;
int op2 = -10 ;
...
main ()
{
    resultado = op1+op2;
    ...
}
```

```
.data
.align 2
resultado: .space 4    # 4 bytes
op1:        .word 100
op2:        .word -10
...
.text
.globl main
main:   lw $t1 op1
        lw $t2 op2
        add $t3 $t1 $t2
        la $t4 resultado
        sw $t3 ($t4)
...
```

## Tipos de datos enteros



## Ejercicio

- ▶ Indique un fragmento de código en ensamblador con la misma funcionalidad que:

```
int b;
int a = 100 ;
int c = 5 ;
int d;

main ()
{
    d = 80;
    b = -(a+b*c+a);
}
```

Asumiendo que a, b, c y d son variables que residen en memoria

## Tipo de datos básicos float

```
float  resultado ;
float  op1 = 100 ;
floar  op2 = 2.5
...
main ()
{
    resultado = op1 + op2 ;
    ...
}
```

```
.data
.align 2
    resultado: .space 4 # 4 bytes
    op1:         .float 100
    op2:         .float 2.5
...
.text
.globl main
main: l.s    $f0 op1
      l.s    $f1 op2
      add.s  $f3 $f1 $f2
      s.s    $f3 resultado
      ...
```

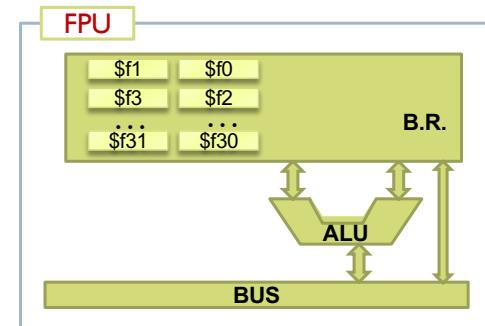
## Tipo de datos básicos double

```
double resultado ;
double op1 = 100 ;
double op2 = -10.27 ;
...
main ()
{
    resultado = op1 * op2 ;
    ...
}
```

```
.data
.align 2
resultado: .space 4
op1:       .double 100
op2:       .double -10.27
...
.text
.globl main
main: l.d    $f0 op1    # ($f0,$f1)
      l.d    $f2 op2    # ($f2,$f3)
      mul.d $f6 $f0 $f2
      s.d    $f6 resultado
      ...
```

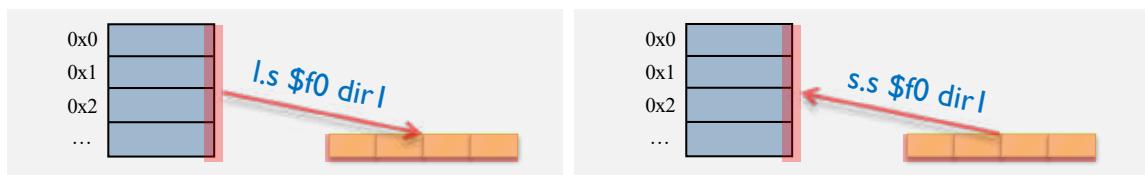
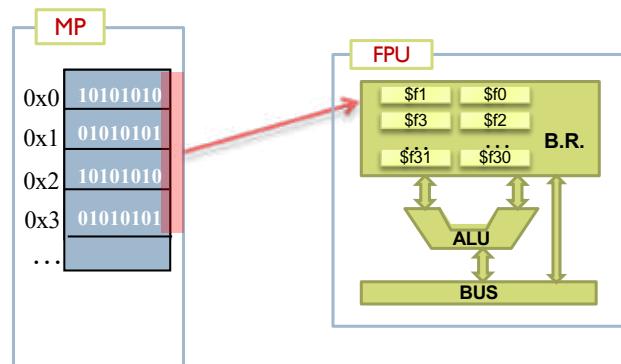
# Banco de registros de coma flotante

- ▶ El coprocesador I tiene 32 registros de 32 bits (4 bytes) cada uno
  - ▶ Es posible trabajar con simple o doble precisión
- ▶ Simple precisión (32 bits):
  - ▶ Del \$f0 al \$f31
  - ▶ Ej.: **add.s \$f0 \$f1 \$f5**  
 $f_0 = f_1 + f_5$
  - ▶ Otras operaciones:
    - ▶ **add.s, sub.s, mul.s, div.s, abs.s**
- ▶ Doble precisión (64 bits):
  - ▶ Se utilizan por parejas
  - ▶ Ej.: **add.d \$f0 \$f2 \$f8**  
 $(f_0, f_1) = (f_2, f_3) + (f_8, f_9)$
  - ▶ Otras operaciones:
    - ▶ **add.d, sub.d, mul.d, div.d, abs.d**



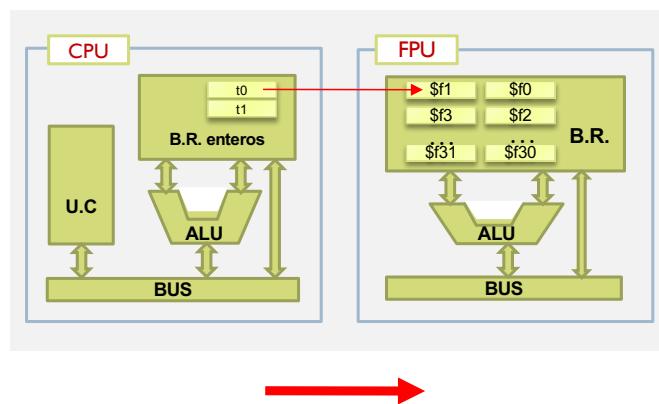
# Transferencia de datos IEEE 754

- ▶ Copia una **número de memoria** a un **registro** o viceversa
- ▶ Instrucciones:
  - ▶ Memoria a registro  
`I.s $f0 dir1`  
`I.d $f2 dir2`
  - ▶ Registro a memoria  
`s.s $f0 dir1`  
`s.d $f0 dir2`



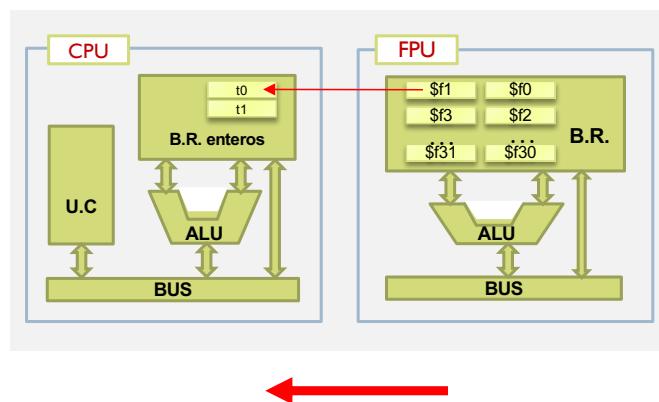
## Operaciones con registros (CPU, FPU)

mtc1 \$t0 \$f1



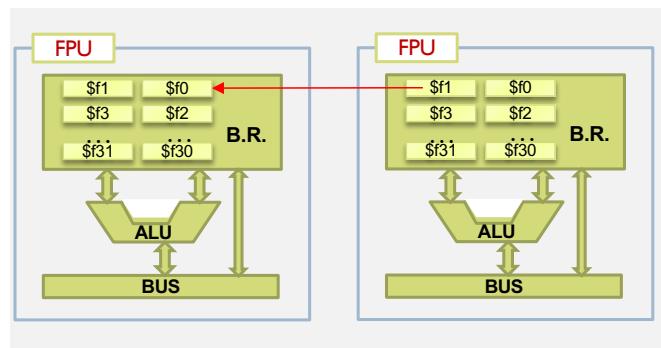
# Operaciones con registros (CPU, FPU)

mfc1 \$t0 \$f1



# Operaciones con registros (FPU, FPU)

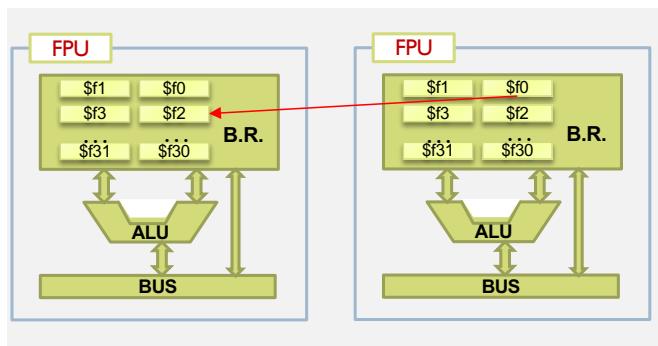
mov.s \$f0 \$f1



\$f0  $\leftarrow$  \$f1

# Operaciones con registros (FPU, FPU)

mov.d \$f0 \$f2



$(\$f0, \$f1) \leftarrow (\$f2, \$f3)$

## Operaciones de conversión

- ▶ **cvt.s.w \$f2 \$f1**
  - ▶ Convierte un entero (\$f1) a simple precisión (\$f2)
- ▶ **cvt.w.s \$f2 \$f1**
  - ▶ Convierte de simple precisión (\$f1) a entero (\$f2)
- ▶ **cvt.d.w \$f2 \$f0**
  - ▶ Convierte un entero (\$f0) a doble precisión (\$f2)
- ▶ **cvt.w.d \$f2 \$f0**
  - ▶ Convierte de doble precisión (\$f0) a entero (\$f2)
- ▶ **cvt.d.s \$f2 \$f0**
  - ▶ Convierte de simple precisión (\$f0) a doble (\$f2)
- ▶ **cvt.s.d \$f2 \$f0**
  - ▶ Convierte de doble precisión (\$f0) a simple(\$f2)

## Operaciones de carga

- ▶ **li.s \$f4, 8.0**
  - ▶ Carga el valor float 8.0 en el registro \$f4
- ▶ **li.d \$f2, 12.4**
  - ▶ Carga el valor double 12.4 en el registro \$f2 , par (\$f2,\$f3)

# Ejemplo

```
float PI      = 3.1415;
int   radio = 4;
float longitud;

longitud = PI * radio;
```

```
.text
.globl main

main:

    li.s    $f0  3.1415
    li     $t0  4

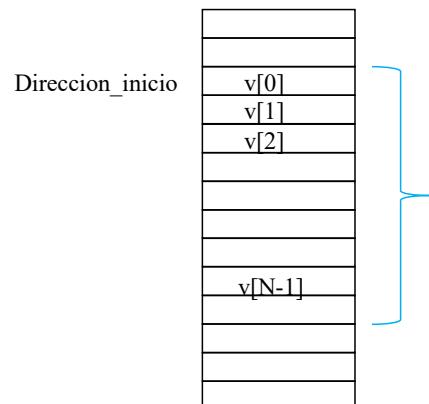
    mtc1   $t0 $f1      # 4 en Ca2
    cvt.s.w $f2 $f1      # 4 ieee754
    mul.s  $f3 $f2 $f0
```

# Tipo de datos básicos vectores

- ▶ Conjunto de elementos ordenados consecutivamente en memoria
- ▶ La dirección del elemento  $j$  se obtiene como:

$$\text{Direccion\_inicio} + j * p$$

Siendo  $p$  el tamaño de cada elemento



## Tipo de datos básicos vectores

```
int vec[5] ;  
...  
main ()  
{  
    vec[4] = 8;
```

```
.data  
.align 2 #siguiente dato alineado a 4  
vec: .space 20    #5 elem.*4 bytes  
  
.text  
main:  
    la $t1 vec  
    li $t2 8  
    sw $t2 16($t1)  
    ...
```

## Tipo de datos básicos vectores

```
int vec[5] ;  
...  
main ()  
{  
    vec[4] = 8;
```

```
.data  
.align 2 #siguiente dato alineado a 4  
vec: .space 20    #5 elem.*4 bytes
```

```
.text  
main:  
    li $t0 16  
    la $t1 vec  
    add $t3, $t1, $t0  
    li $t2 8  
    sw $t2, ($t3)  
    ...
```

## Tipo de datos básicos vectores

```
int vec[5] ;  
...  
  
main ()  
{  
    vec[4] = 8;  
}
```

```
.data  
.align 2      #siguiente dato alineado a 4  
vec: .space 20  #5 elem.*4 bytes  
  
.text  
main:  
    li $t2 8  
    li $t1 16  
    sw $t2 vec($t1)  
    ...
```

## Ejercicio

- ▶ Si V es un array de números enteros (int)
  - ▶ V representa la dirección de inicio de vector
- ▶ ¿En qué dirección se encuentra el elemento V[5]?
- ▶ ¿Qué instrucción permite cargar en el registro \$t0 el valor v[5]?

## Ejercicio (Solución)

- ▶ Si V es un array de números enteros (int)
  - ▶ V representa la dirección de inicio de vector
- ▶ ¿En qué dirección se encuentra el elemento V[5]?
  - ▶  $V + 5*4$
- ▶ ¿Qué instrucción permite cargar en el registro \$t0 el valor v[5]?
  - ▶ `lw $t1, 20`
  - ▶ `lw $t0, v($t1)`

## Ejercicio

- ▶ Escriba un programa en ensamblador equivalente a:

```
int vec[100] ;  
...  
  
main ()  
{  
    int i = 0;  
  
    for (i = 0; i < 100; i++)  
        vec[i] = 5;  
  
}
```

- ▶ Asumiendo que en \$a0 se encuentra almacenada la dirección del vector

## Ejercicio

- ▶ Escriba un programa en ensamblador equivalente a:

```
int vec[100] ;  
...  
  
main ()  
{  
    int i = 0;  
    suma = 0;  
  
    for (i = 0; i < 100; i++)  
        suma = suma + vec[i];  
  
}
```

- ▶ Asumiendo que en \$a0 se encuentra almacenada la dirección del vector y que el resultado ha de almacenarse en \$v0

## Tipo de datos básicos cadenas de caracteres

```
char c1 ;
char c2='h' ;
char *ac1 = "hola" ;
...
```

```
main ()
{
    printf("%s",ac1) ;
    ...
}
```

```
.data
c1:  .space 1      # 1 byte
c2:  .byte 'h'
ac1: .asciiz "hola"
...
```

```
.text
main:
    li $v0 4
    la $a0 ac1
    syscall
    ...
```

## Representación de cadenas de caracteres

```
// tira de caracteres (strings)
char c1[10] ;
char ac1[] = "hola" ;
```

```
.data

# strings
c1: .space 10      # 10 byte
ac1: .asciiz "hola" # 5 bytes (!)
ac2: .ascii  "hola" # 4 bytes
```

...	...	...	...		
ac1:	'h'	0x0108	ac2:	'h'	0x0108
	'o'	0x0109		'o'	0x0109
	'l'	0x010a		'l'	0x010a
	'a'	0x010b		'a'	0x010b
	0	0x010c		...	0x010c
	...	0x010d		...	0x010d

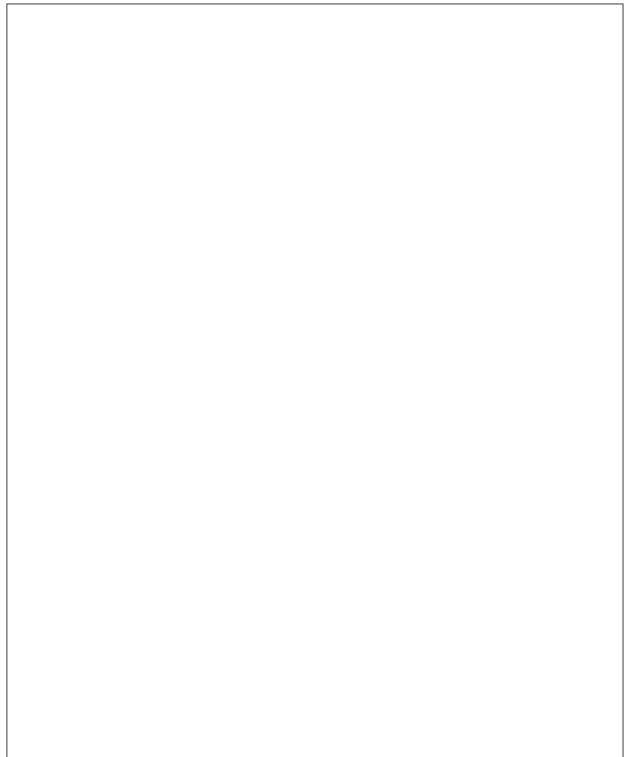
# Ejercicio

```
// variables globales

char v1;
int v2 ;
float v3 = 3.14 ;

char v4[10] ;
char v5 = "ec" ;

int v6[] = { 20, 22 } ;
```



## Ejercicio (solución)

```
// variables globales

char v1;
int v2 ;
float v3 = 3.14 ;

char v4 = "ec" ;

int v5[] = { 20, 22 } ;
```

```
.data

v1: .space 1
.align 2
v2: .space 4
v3: .float 3.14

v4: .asciiz "ec"

.align 2
v5: .word 20, 22
```

## Ejercicio (solución)

v1:	0	0x0100
	?	0x0101
	?	0x0102
	?	0x0103
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	...	

```
.data

v1: .space 1
.align 2
v2: .space 4
v3: .float 3.14

v4: .asciiz "ec"

.align 2
v5: .word 20, 22
```

# Ejercicio (solución)

v1:	0	0x0100
	?	0x0101
	?	0x0102
	?	0x0103
v2:	0	0x0104
	0	0x0105
	0	0x0106
	0	0x0107
v3:	(3.14)	0x0108
	(3.14)	0x0109
	(3.14)	0x010A
	(3.14)	0x010B
v4:	'e'	0x010C
	'c'	0x010D
	0	0x010E
		0x010F
v5:	(20)	0x0110
	(22)	0x0111
	...	0x0112

```
.data

v1: .space 1
.align 2
v2: .space 4
v3: .float 3.14

v4: .asciiz "ec"

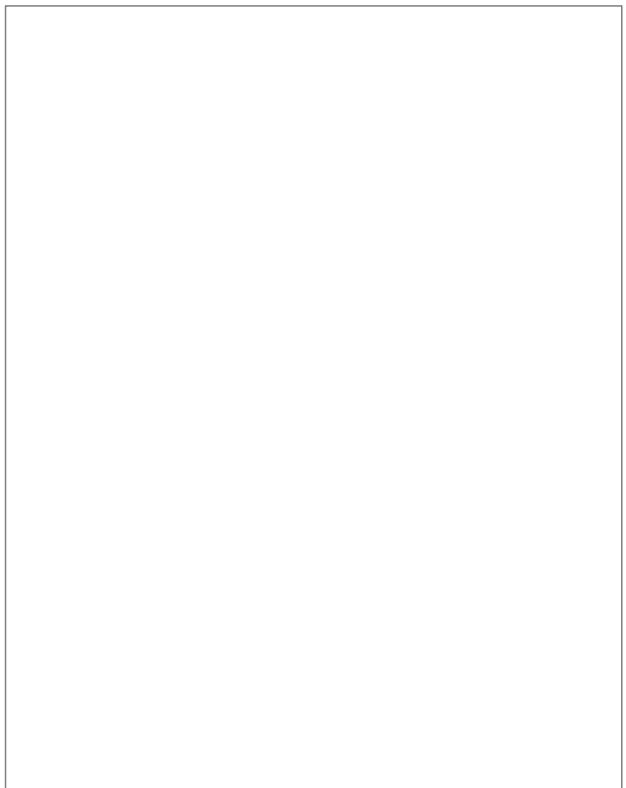
.align 2
v5: .word 20, 22
```

## Tipo de datos básicos

### Longitud de una cadena de caracteres

```
char c1 ;
char c2='h' ;
char *ac1 = "hola" ;
char *c;
...

main ()
{
    c = ac1; int l = 0;
    while (c[l] != NULL) {
        l++;
    }
    printf("%d", l);
    ...
}
```



# Tipo de datos básicos

## Longitud de una cadena de caracteres

```
char c1 ;
char c2='h' ;
char *acl = "hola" ;
char *c;

main ()
{
    c = acl; int l = 0;
    while (c[l] != NULL) {
        l++;
    }
    printf("%d", l);
    ...
}
```

```
.data
c1: .space 1      # 1 byte
c2: .byte 'h'
acl: .asciiz "hola"
.align 2
c: .space 4      #puntero => dirección
...
.text
.globl main
main: la $t0, acl
      li $a0, 0
      lbu $t1, ($t0)
      buc: beqz $t1, fin
            addi $t0, $t0, 1
            addi $a0, $a0, 1
            lbu $t1, ($t0)
            b buc

      fin: li $v0 1
            syscall
...

```

# Vectores y cadenas

## ▶ En general:

- ▶ `lw $t0, 4($s3) # $t0` ←———— `M[$s3+4]`
- ▶ `sw $t0, 4($s3) # M[$s3+4]` ←———— `$t0`

## Ejercicio

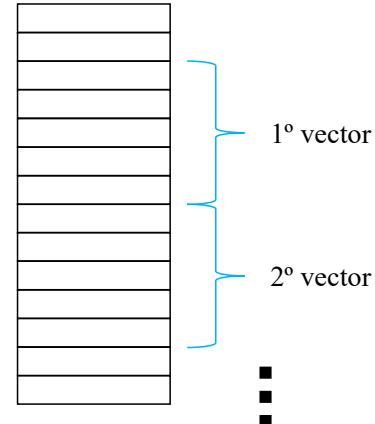
- ▶ Escriba un programa que:
  - ▶ Indique el número de veces que aparece un carácter en una cadena de caracteres
    - ▶ La dirección de la cadena se encuentra en \$a0
    - ▶ El carácter a buscar se encuentra en \$a1
    - ▶ El resultado se dejará en \$v0

## Tipos de datos básicos matrices

- ▶ Una matriz  $m \times n$  se compone de  $m$  vectores de longitud  $n$
- ▶ Normalmente se almacenan en memoria por filas
- ▶ El elemento  $a_{ij}$  se encuentra en la dirección:

$$\text{direccion\_inicio} + (i \cdot n + j) \times p$$

siendo  $p$  el tamaño de cada elemento



## Tipo de datos básicos matrices

```
int vec[5] ;
int mat[2][3] = {{11,12,13},
                  {21,22,23}};
...
main ()
{
    m[0][1] = m[0][0] +
                m[1][0] ;
    ...
}
```

```
.data
    .align 2    #siguiente dato alineado a 4
vec: .space 20    #5 elem.*4 bytes
mat: .word 11, 12, 13
        .word 21, 22, 23
...
.

.text
main:   lw $t1 mat+0
        lw $t2 mat+12
        add $t3 $t1 $t2
        sw  $t3 mat+4
...
```

## Ejemplo (enteros)

```
int vec[5] ;
int mat[2][3] = {{11,12,13},
                  {21,22,23}};
...
main ()
{
    mat[1][2] = mat[1][1] +
        mat[2][1];
    ...
}
```

```
.data
align 2
vec: .space 20      #5 elem.*4 bytes
mat: .word 11, 12, 13
      .word 21, 22, 23
...
.text
.globl main
main:  lw   $t1 mat+0
       lw   $t2 mat+12
       add $t3 $t1 $t2
       sw   $t3 mat+4
...

```

# Punteros en C

```
int a;
int *b;

main ()
{
    b = &a;
    *b = 2;
    ...
}
```

```
.data
align 2
a: .space 4      # int
b: .space 4      # dirección

.text
.globl main
main:   la $t0,  a
        sw $t0,  b

        li $t0,  2
        lw $t1,  b
        sw $t0,  ($t1)

.....
```

# Otros tipos de datos (estructuras de C)

```
struct Punto {  
    int x;  
    int y;  
};  
  
struct Punto p;
```

```
main ()  
{  
    p.x = 80;  
    p.y = 80;  
}
```

```
.data  
align 2  
p:  
p.x: .space 4  
p.y: .space 4  
...
```

```
.text  
main:  
    li $t0, 80  
    sw $t0, p.x  
    li $t1, 70  
    sw $t1, p.y
```

# Otros tipos de datos (estructuras de C)

```
struct Punto {  
    int x;  
    int y;  
};
```

```
struct Punto p;  
struct Punto q;
```

```
main ()  
{  
    p.x = 80;  
    p.y = 80;  
    q = p;  
}
```

```
.data  
align 2  
p:  
p.x: .space 4  
p.y: .space 4  
q:  
q.x: .space 4  
q.y: .space 4  
...
```

# Otros tipos de datos (estructuras de C)

```
struct Punto {  
    int x;  
    int y;  
};  
  
struct Punto p;  
struct Punto q;  
  
main ()  
{  
    p.x = 80;  
    p.y = 80;  
    q = p;  
}
```

```
.text  
main: li $t0, 80  
      sw $t0, p.x  
      li $t1, 70  
      sw $t1, p.y  
  
      lw $t0, p.x  
      sw $t0, q.x  
      lw $t0, p.y  
      sw $t0, q.y
```

## Otros tipos de datos (estructuras de C)

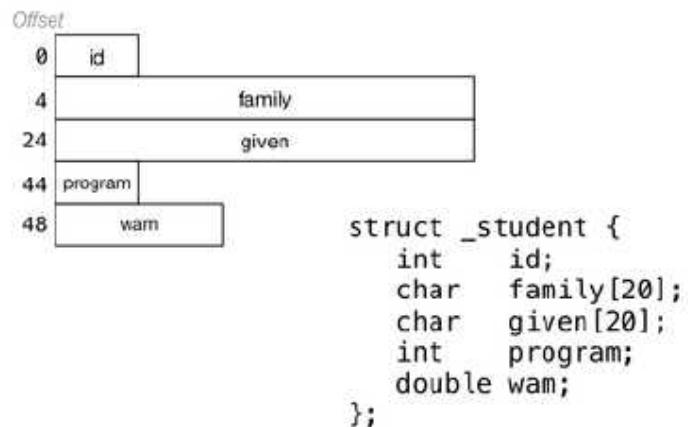
```
struct Punto {  
    int x;  
    char y[21];  
    int z;  
};  
  
struct Punto p;
```

```
main ()  
{  
    p.y[2] = 'b';  
}
```

```
.data  
align 2  
p:  
p.x: .space 4  
p.y: .space 21  
align 2  
p.z: .space 4
```

```
.text  
main:  
    li $t0, 'b'  
    la $t1, p.y  
    sw $t1, 2($t1)
```

# Otros tipos de datos (estructuras de C)



# Consejos

- ▶ **No programar directamente en ensamblador**
  - ▶ Mejor **primero hacer diseño** en DFD, Java/C/Pascal...
  - ▶ Ir traduciendo poco a poco el diseño a ensamblador
- ▶ **Comentar suficientemente el código y datos**
  - ▶ Por línea o por grupo de líneas  
comentar qué parte del diseño implementa.
- ▶ **Probar con suficientes casos de prueba**
  - ▶ Probar que el programa final funciona adecuadamente a las especificaciones dadas

# Ejercicio

- ▶ Escriba un programa que:
  - ▶ Cargue el valor -3.141516 en el registro \$f0
  - ▶ Permita obtener el valor del exponente y de la mantisa almacenada en el registro \$f0 (en formato IEEE 754)
    - ▶ Imprima el signo
    - ▶ Imprima el exponente
    - ▶ Imprima la mantisa

## Ejercicio (Solución)

```
.data

saltolinea: .asciiz "\n"

.text
.globl main
main:
    li.s $f0, -3.141516

    #se imprime
    mov.s $f12, $f0
    li $v0, 2
    syscall

    la $a0, saltolinea
    li $v0, 4
    syscall

    # se copia al procesador
    mfcl $t0, $f12
```

```
    li $s0, 0x80000000    #signo
    and $a0, $t0, $s0
    srl $a0, $a0, 31
    li $v0, 1
    syscall

    la $a0, saltolinea
    li $v0, 4
    syscall

    li $s0, 0x7F800000    #exponente
    and $a0, $t0, $s0
    srl $a0, $a0, 23
    li $v0, 1
    syscall

    la $a0, saltolinea
    li $v0, 4
    syscall

    li $s0, 0x007FFFFFFF  #mantisa
    and $a0, $t0, $s0
    li $v0, 1
    syscall

    jr $ra
```

Grupo ARCOS

**uc3m | Universidad Carlos III de Madrid**

## Tema 3 (III) Fundamentos de la programación en ensamblador

Estructura de Computadores  
Grado en Ingeniería Informática



# Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del MIPS 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila

## Formatos de las instrucciones de acceso a memoria (Repaso)

lw  
sw  
lb  
sb  
lbu

Registro, dirección de memoria

Número que representa una dirección  
Etiqueta simbólica que representa una dirección  
(registro): representa la dirección almacenada en el registro  
num(registro): representa la dirección que se obtiene de sumar num con la dirección almacenada en el registro  
**etiqueta + num:** representa la dirección que se obtiene de sumar etiqueta con num

# Instrucciones y pseudoinstrucciones del MIPS 32

- ▶ Una instrucción en ensamblador se corresponde con una instrucción máquina
  - ▶ Ocupa 32 bits
  - ▶ addi \$t1, \$t1, 2
- ▶ Una pseudoinstrucción en ensamblador se corresponde con varias instrucciones máquina.
  - ▶ li \$t1, 0x00800010
    - ▶ No cabe en 32 bits, pero se puede utilizar como pseudoinstrucción.
    - ▶ Es equivalente a:
      - lui \$t1, 0x0080
      - ori \$t1, \$t1, 0x0010

## Otro ejemplo de pseudoinstrucción del MIPS 32

- ▶ La pseudoinstrucción move

```
move      reg2, reg1
```

- ▶ Se convierte en:

```
add      reg2,$zero,reg1
```

# Información de una instrucción

- ▶ El tamaño de la instrucción se ajusta al de palabra (o múltiplo)
- ▶ Una instrucción máquina **se divide en campos:**
  - ▶ Operación a realizar
  - ▶ Operandos a utilizar
    - ▶ Puede haber operando implícitos
- ▶ El formato de una instrucción indica los campos y su tamaño:
  - ▶ Uso de formato sistemático
  - ▶ Tamaño de un campo limita los valores que codifica
    - ▶ Un campo de n bits permite codificar  $2^n$  valores distintos

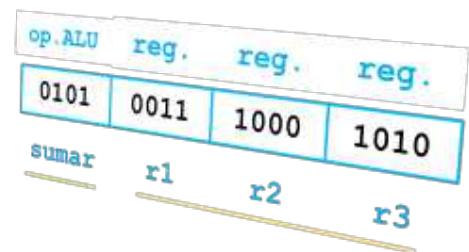
# Información de una instrucción

- ▶ Se utiliza unos pocos formatos:
  - ▶ Cada instrucción pertenece a un formato
  - ▶ Según el código de operación se conoce el formato asociado
- ▶ Ejemplo: formatos en MIPS

Tipo R aritméticas	<table border="1"><tr><td><b>op.</b></td><td><b>rs</b></td><td><b>rt</b></td><td><b>rd</b></td><td><b>shamt</b></td><td><b>func.</b></td></tr><tr><td>6 bits</td><td>5 bits</td><td>5 bits</td><td>5 bits</td><td>5 bits</td><td>6 bits</td></tr></table>	<b>op.</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>func.</b>	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
<b>op.</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>func.</b>								
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits								
Tipo I transferencia inmediato	<table border="1"><tr><td><b>op.</b></td><td><b>rs</b></td><td><b>rt</b></td><td></td><td colspan="2"><b>offset</b></td></tr><tr><td>6 bits</td><td>5 bits</td><td>5 bits</td><td></td><td colspan="2">16 bits</td></tr></table>	<b>op.</b>	<b>rs</b>	<b>rt</b>		<b>offset</b>		6 bits	5 bits	5 bits		16 bits	
<b>op.</b>	<b>rs</b>	<b>rt</b>		<b>offset</b>									
6 bits	5 bits	5 bits		16 bits									
Tipo J saltos	<table border="1"><tr><td><b>op.</b></td><td colspan="5"><b>offset</b></td></tr><tr><td>6 bits</td><td colspan="5">26 bits</td></tr></table>	<b>op.</b>	<b>offset</b>					6 bits	26 bits				
<b>op.</b>	<b>offset</b>												
6 bits	26 bits												

# Campos de una instrucción

- ▶ En los campos se codifica:
  - ▶ Operación a realizar (código Op.)
    - ▶ Instrucción y formato de la misma
  - ▶ Operandos a utilizar
    - ▶ Ubicación de los operandos
    - ▶ Ubicación del resultado
    - ▶ Ubicación de la siguiente instrucción (*si op. salto*)
      - Implícito:  $PC \leftarrow PC + '4'$  (apuntar a la siguiente instrucción)
      - Explícito: j 0x01004 (modifica el PC)



## Ubicaciones posibles para los operandos

- ▶ En la propia instrucción
- ▶ En los registros del procesador
- ▶ En memoria principal
- ▶ En unidades de Entrada/Salida (I/O)

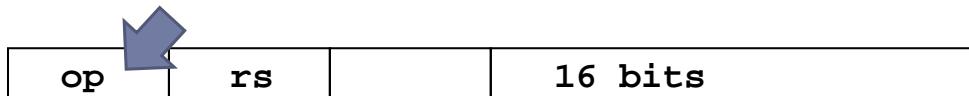
# Modos de direccionamiento

- ▶ El **modo de direccionamiento** es un procedimiento que permite determinar la ubicación de un operando, un resultado o una instrucción

- ▶ Implícito
- ▶ Inmediato
- ▶ Directo
  - ▶ a registro
  - ▶ a memoria
- ▶ Indirecto
  - ▶ a registro
  - ▶ a memoria
- ▶ Relativo
  - ▶ a registro índice
  - ▶ a registro base
  - ▶ a PC
  - ▶ a Pila

## Direccionamiento implícito

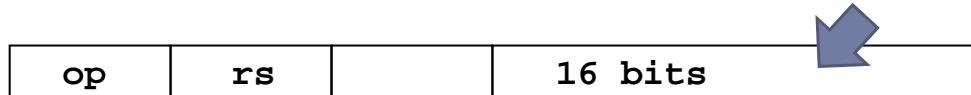
- ▶ El operando no está codificado en la instrucción, pero forma parte de esta
- ▶ Ejemplo: **beqz \$a0 etiqueta**
  - ▶ Si registro \$a0 es cero, salta a **etiqueta**.
  - ▶ \$a0 es un operando, \$zero es el otro (implícito)



- ▶ V/I (Ventajas/Inconvenientes)
  - ✓ Es rápido: no es necesario acceder a memoria.
  - ✗ Pero solo es posible en unos pocos casos.

## Direccionamiento inmediato

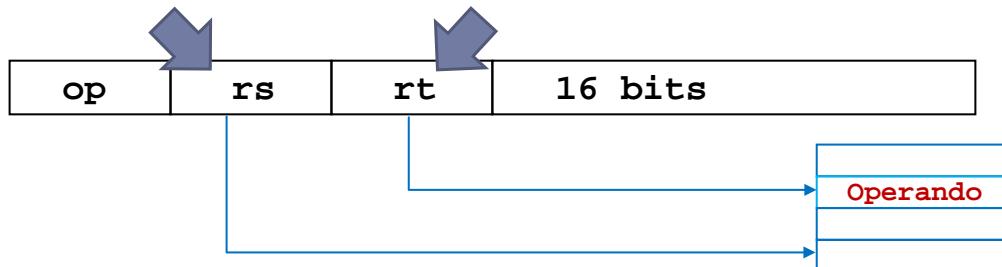
- ▶ El operando forma parte de la instrucción.
- ▶ Ejemplo: **li \$a0 0x4f51**
  - ▶ Carga en el registro \$a0 el valor inmediato **0x4f51**.
  - ▶ El valor 0x00004f51 está codificado en la propia instrucción.



- ▶ V/I
  - ✓ Es rápido: no es necesario acceder a memoria.
  - ✗ No siempre cabe el valor en una palabra:
    - ▶ No cabe en 32 bits, es equivalente a:
      - lui \$t1, 0x0080
      - ori \$t1, \$t1, 0x0010

## Direccionamiento directo a registro (direccionamiento de registro)

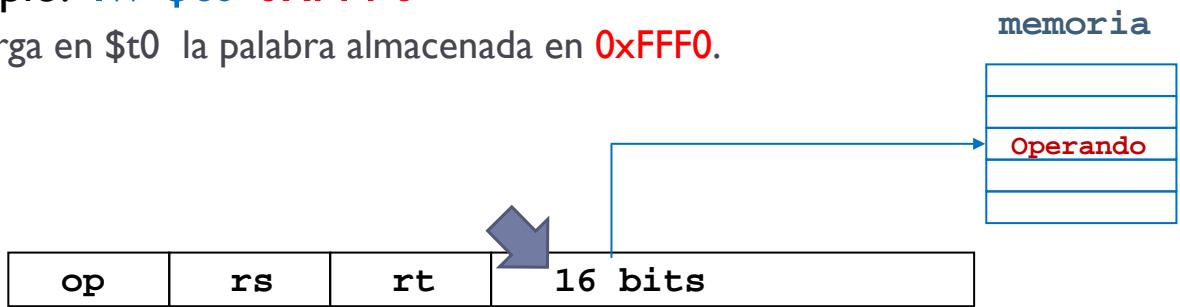
- ▶ El operando se encuentra en el registro.
- ▶ Ejemplo: **move \$a0 \$a1**
  - ▶ Copia en el registro **\$a0** el valor que hay en el registro **\$a1**.
  - ▶ El identificador de **\$a0** y **\$a1** está codificado en la instrucción.



- ▶ V/I
  - ✗ El número de registros está limitado.
  - ✓ Acceso a registros es rápido
  - ✓ El número de registros es pequeño => pocos bits para su codificación, instrucciones más cortas

## Direccionamiento directo a memoria

- ▶ El operando se encuentra en memoria, y la dirección está codificada en la instrucción.
- ▶ Ejemplo: **Iw \$t0 0xFFFF**
  - ▶ Carga en \$t0 la palabra almacenada en **0xFFFF**.



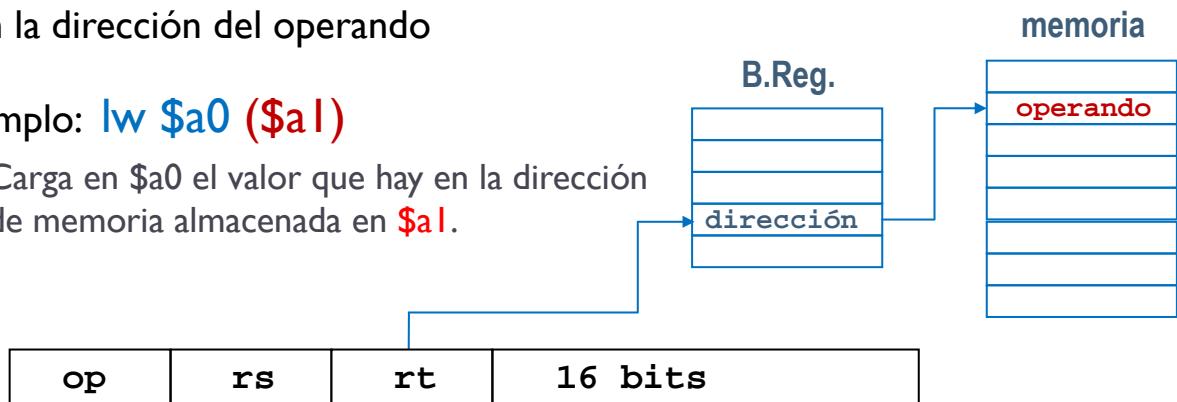
- ▶ V/I
  - ✗ Acceso a memoria es más lento comparado con los registros
  - ✗ Direcciones largas => instrucciones más largas
  - ✓ Acceso a un gran espacio de direcciones (capacidad > B.R.)

## Direccionamiento directo vs. indirecto

- ▶ En el direccionamiento directo se indica **dónde está el operando**:
  - ▶ En qué registro o en qué posición de memoria
- ▶ En el direccionamiento indirecto se indica **dónde está la dirección del operando**:
  - ▶ Hay que acceder a esa dirección en memoria
  - ▶ Se incorpora un nivel (o varios) de direccionamiento

## Direccionamiento indirecto de registro

- ▶ Se indica en la instrucción el registro con la dirección del operando
- ▶ Ejemplo: **lw \$a0 (\$a1)**
  - ▶ Carga en \$a0 el valor que hay en la dirección de memoria almacenada en **\$a1**.

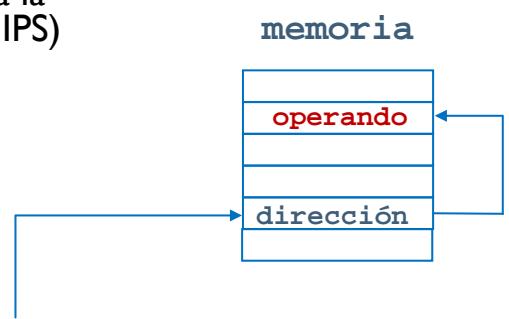


- ▶ V/I
- ✓ Amplio espacio de direcciones, instrucciones cortas

# Direccionamiento indirecto a memoria

- ▶ Se indica en la instrucción la dirección donde está la dirección del operando (no disponible en MIPS)
- ▶ Ejemplo: **LD RI [DIR]** (IEEE 694)
  - ▶ Carga en RI el valor que hay en la dirección de memoria que está almacenada en la dirección de memoria **DIR**.
  - ▶ .

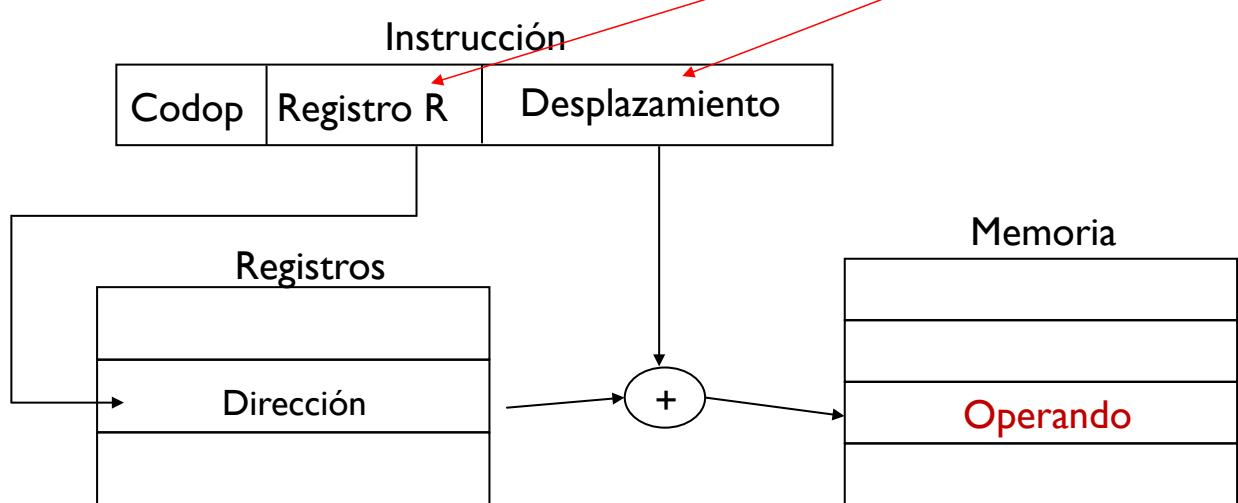
<b>op</b>	<b>rs</b>	<b>rt</b>	<b>16 bits</b>
-----------	-----------	-----------	----------------



- ▶ V/I
  - ✓ Amplio espacio de direcciones
  - ✓ El direccionamiento puede ser anidado, multnivel o en cascada
    - ▶ Ejemplo: LD RI [[[.RI]]]
- ✗ Puede requerir varios accesos memoria
- ✗ instrucciones más lentas de ejecutar

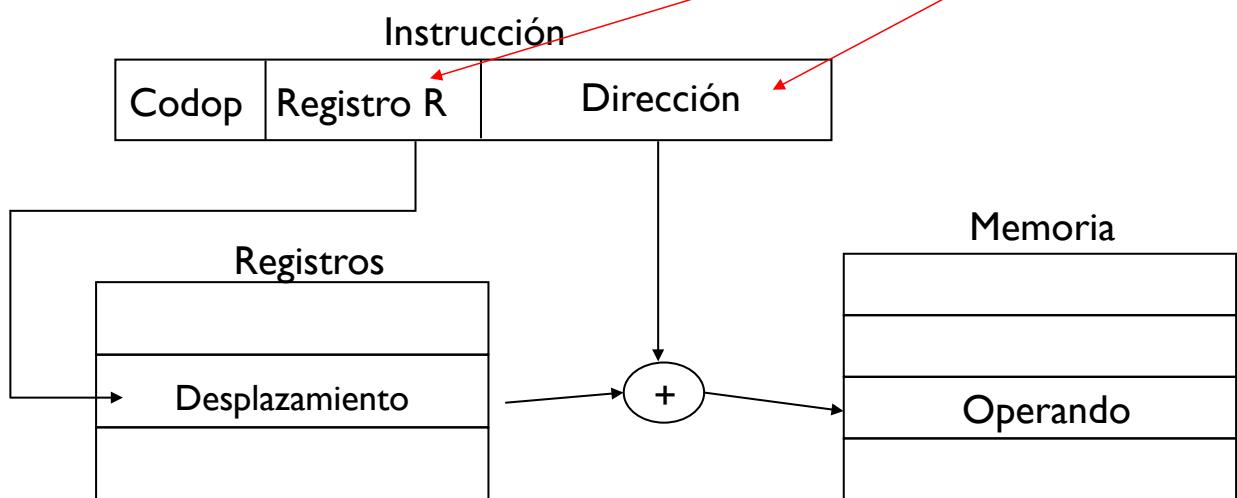
## Direccionamiento relativo a registro base

- Ejemplo: **lw \$a0 12(\$t1)**
  - Carga en \$a0 el contenido de la posición de memoria dada por  $\$t1 + 12$
  - Utiliza dos campos de la instrucción, \$t1 tiene la dirección base



## Direccionamiento relativo a registro índice

- Ejemplo: **lw \$a0 dir(\$t1)**
  - Carga en \$a0 el contenido de la posición de memoria dada por  $\$t1 + \text{dir}$
  - Utiliza dos campos: \$t1 representa el desplazamiento (índice) respecto a la dirección **dir**



# Utilidad: acceso a vectores

```
int v[5] ;
```

```
main ( )
```

```
{
```

```
    v[3] = 5 ;
```

```
    v[4] = 8 ;
```

```
}
```

```
.data
    .align 2#siguiente dato alineado a 4

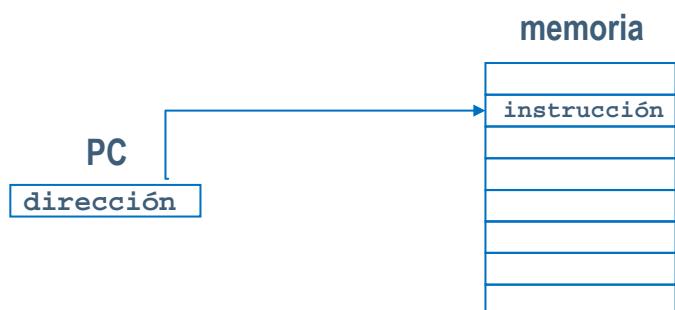
v: .space 20      # 5_int*4_bytes/int

.text
.globl main
main:
    la $t0 v
    li $t1 5
    sw $t1 12($t0)

    la $t0 16
    li $t1 8
    sw $t1 v($t0)
```

# Direccionamiento relativo al contador de programa

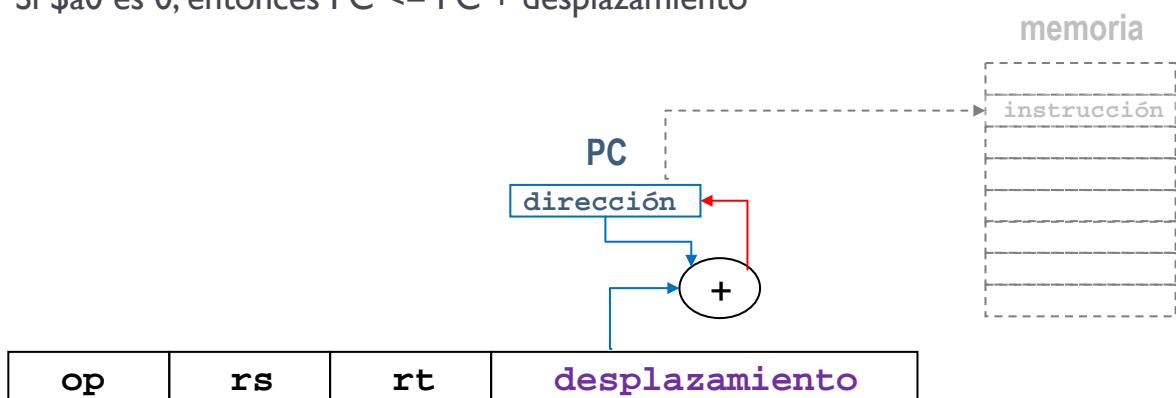
- ▶ El contador de programa PC:
  - ▶ Es un registro de 32 bits (4 bytes)
  - ▶ Almacena la dirección de la siguiente instrucción a ejecutar
    - ▶ Apunta a una palabra (4 bytes) con la instrucción a ejecutar



# Direccionamiento relativo al contador de programa

## ▶ Ejemplo: **beqz \$a0 etiqueta**

- ▶ La instrucción codifica etiqueta como el desplazamiento desde la dirección de memoria donde está esta instrucción, hasta la posición de memoria indicada en **etiqueta**.
- ▶ Si  $\$a0$  es 0, entonces  $PC \leq PC + \text{desplazamiento}$



# Contador de programa en el MIPS 32

- ▶ Los registros tienen 32 bits
- ▶ El contador de programa tiene 32 bits
- ▶ Las instrucciones ocupan 32 bits (una palabra)
- ▶ El contador de programa almacena la dirección donde se encuentra una instrucción
- ▶ La siguiente instrucción se encuentra 4 bytes después.
- ▶ Por tanto el contador de programa se actualiza:
  - ▶  $PC = PC + 4$

Dirección:	Instrucción:
0x00400000	or \$2,\$0,\$0
0x00400004	slt \$8,\$0,\$5
0x00400008	beq \$8,\$0,3
0x0040000c	add \$2,\$2,\$4
0x00400010	addi \$5,\$5,-1
0x00400014	j 0x100001

## Direccionamiento relativo a PC en el MIPS

- ▶ La instrucción `beq $t0, $1, etiqueta` se codifica en la instrucción:



- ▶ Etiqueta tiene que codificarse en el campo “Dato inmediato”
- ▶ ¿Cómo se actualiza el PC si  $$t0 == \$1$  y cuánto vale fin cuando se genera código máquina?

```
bucle:    beq    $t0, $1,  fin
           add    $t8, $t4, $t4
           addi   $t0, $0, -1
           j      bucle
fin:      . . .
```

## Direccionamiento relativo a PC en el MIPS

- ▶ Si se cumple la condición
  - ▶  $PC = PC + (\text{etiqueta}^* 4)$
- ▶ Por tanto en:

```
bucle:      beq    $t0, $1,  fin
            add    $t8, $t4, $t4
            addi   $t0, $0,  -1
            j      bucle
fin:        . . .
```

- ▶ **fin == 3**
  - ▶ Cuando se ejecuta una instrucción, el PC apunta a la siguiente

## Utilidad: desplazamientos en bucles

- ▶ **fin** representa la dirección donde se encuentra la instrucción move

```
        li    $t0 8
        li    $t1 4
        li    $t2 1
        li    $t4 0
while: bge  $t4 $t1 fin
       mul  $t2 $t2 $t0
       addi $t4 $t4 1
       b    while
fin:   move $t2 $t4
```

## Utilidad: desplazamientos en bucles

	Dirección	Contenido
li \$t0 8	0x0000100	li \$t0 8
li \$t1 4	0x0000104	li \$t1 4
li \$t2 1	0x0000108	li \$t2 1
li \$t4 0	0x000010C	li \$t4 0
<b>while:</b> bge \$t4 \$t1 <b>fin</b>	0x0000110	bge \$t4 \$t1 <b>fin</b>
mul \$t2 \$t2 \$t0	0x0000114	mul \$t2 \$t2 \$t0
addi \$t4 \$t4 1	0x0000118	addi \$t4 \$t4 1
b <b>while</b>	0x000011C	b <b>while</b>
<b>fin:</b> move \$t2 \$t4	0x0000120	move \$t2 \$t4

# Utilidad: desplazamientos en bucles

```

        li    $t0 8
        li    $t1 4
        li    $t2 1
        li    $t4 0
while:   bge  $t4 $t1 fin
            mul  $t2 $t2 $t0
            addi $t4 $t4 1
            b     while
fin:      move $t2 $t4

```

**fin** representa un desplazamiento respecto al PC actual => **3**  
 $PC = PC + 3 * 4$

**while** representa un desplazamiento respecto al PC actual =>**-4**  
 $PC = PC + (-4)*4$

Dirección	Contenido
0x0000100	li \$t0 8
0x0000104	li \$t1 4
0x0000108	li \$t2 1
0x000010C	li \$t4 0
0x0000110	bge \$t4 \$t1 <b>fin</b>
0x0000114	mul \$t2 \$t2 \$t0
0x0000118	addi \$t4 \$t4 1
0x000011C	b <b>while</b>
0x0000120	move \$t2 \$t4

# Utilidad: desplazamientos en bucles

```

        li    $t0 8
        li    $t1 4
        li    $t2 1
        li    $t4 0
while:   bge  $t4 $t1 fin
            mul  $t2 $t2 $t0
            addi $t4 $t4 1
            b     while
fin:      move $t2 $t4

```

**fin** representa un desplazamiento respecto al PC actual => **3**  
 $PC = PC + 3 * 4$

**while** representa un desplazamiento respecto al PC actual =>**-4**  
 $PC = PC + (-4)*4$

Dirección	Contenido
0x0000100	li \$t0 8
0x0000104	li \$t1 4
0x0000108	li \$t2 1
0x000010C	li \$t4 0
0x0000110	bge \$t4 \$t1 <b>3</b>
0x0000114	mul \$t2 \$t2 \$t0
0x0000118	addi \$t4 \$t4 1
0x000011C	b <b>-4</b>
0x0000120	move \$t2 \$t4

## Diferencia entre las instrucción b y j

Instrucción j dirección

op.	dirección
6 bits	26 bits

Dirección de salto =>  $PC = \text{dirección}$

Instrucción b desplazamiento

op.			desplazamiento
6 bits	5 bits	5 bits	16 bits

Dirección de salto =>  $PC = PC + \text{desplazamiento} * 4$   
permite que el código sea reubicable en memoria

# Ejercicio

- ▶ Dadas estas 2 instrucciones para realizar un salto incondicional:

- ▶ 1) **j etiqueta**



- ▶ 2) **b etiqueta2**

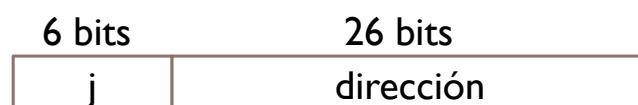


- ▶ Donde en la primera se carga la dirección en PC y en la segunda se suma el desplazamiento a PC (siendo este un número en complemento a dos)
- ▶ Se pide:
  - ▶ Indique razonadamente cual de las dos opciones es más apropiada para bucles pequeños.

# Ejercicio (solución)

## ▶ Ventajas de la opción 1:

- ▶ El cálculo de la dirección es más rápido, solo cargar
- ▶ El rango de direcciones es mayor, mejor para bucles grandes



## ▶ Ventajas de la opción 2:

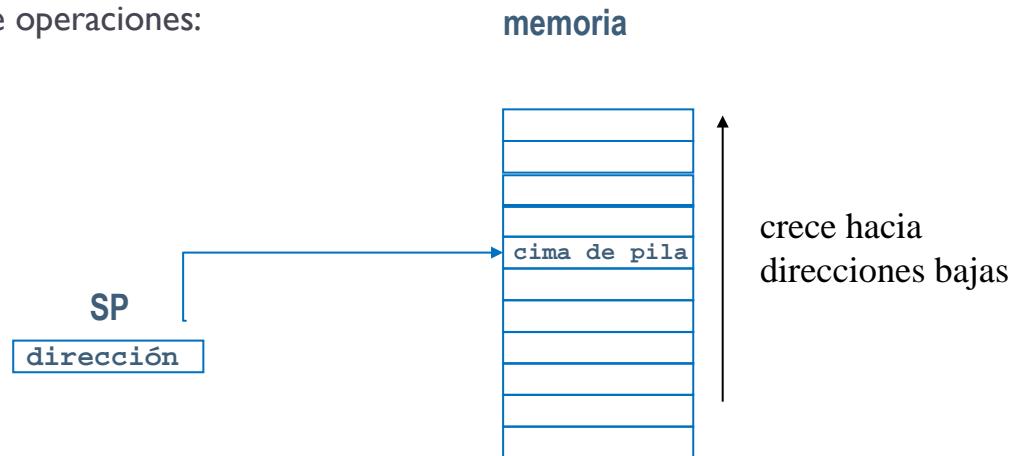
- ▶ El rango de direcciones a las que se puede saltar es menor (bucles pequeños)
- ▶ Permite un código reubicable



## ▶ La opción 2 sería más apropiada

# Direccionamiento relativo a pila

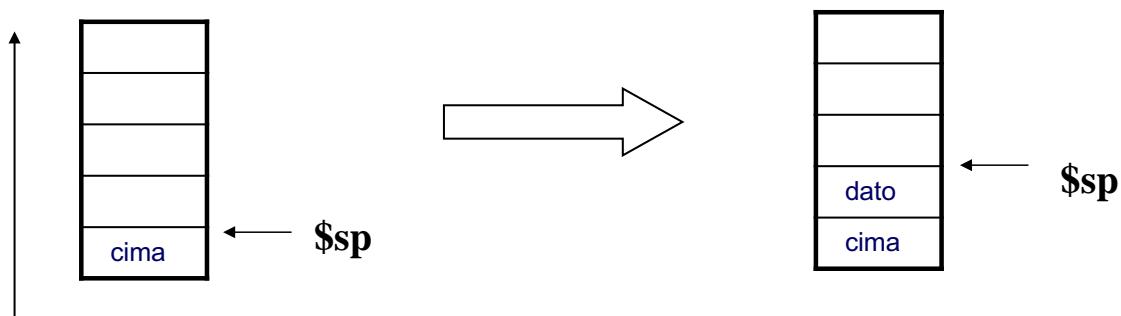
- ▶ El puntero de pila SP (*Stack Pointer*):
  - ▶ Es un registro de 32 bits (4 bytes) en el MIPS
  - ▶ Almacena la dirección de la cima de pila
    - ▶ Apunta a una palabra (4 bytes)
- ▶ Dos tipos de operaciones:
  - ▶ [push](#)
  - ▶ [pop](#)



# Operación PUSH

**PUSH Reg**

Apila el contenido del registro (dato)

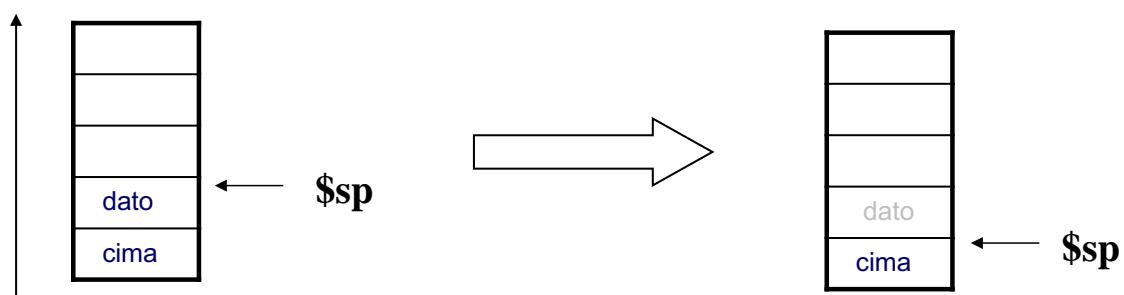


crece hacia direcciones bajas

# Operación POP

## POP Reg

Desapila el contenido del registro (dato)  
Copia dato en el registro Reg



crece hacia direcciones bajas

# Direccionamiento de pila en el MIPS

- ▶ MIPS no dispone de instrucciones PUSH o POP.
- ▶ El registro puntero de pila (\$sp) es visible al programador.
  - ▶ Se va a asumir que el puntero de pila apunta al último elemento de la pila

## PUSH \$t0

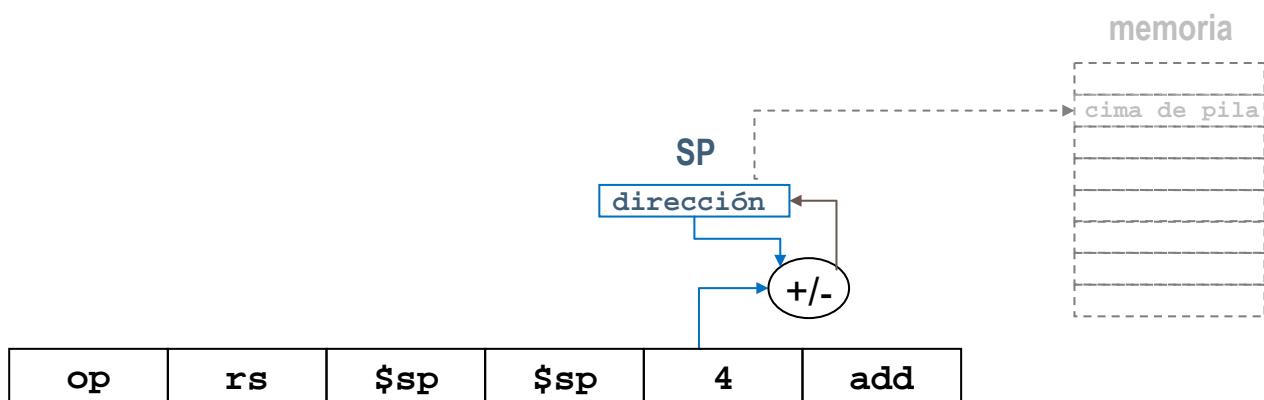
```
addu $sp, $sp, -4  
sw    $t0, ($sp)
```

## POP \$t0

```
lw    $t0, ($sp)  
addu $sp, $sp, 4
```

# Operación PUSH en MIPS

- ▶ Ejemplo: **push \$a0**
  - ▶ addu \$sp \$sp -4 # \$SP = \$SP - 4
  - ▶ sw \$a0 (\$sp) # memoria[\$SP] = \$a0



# Modos de direccionamiento en MIPS

## ▶ Direccionamientos:

- |   |                      |
|---|----------------------|
| ▶ Inmediato   | valor                |
| ▶ De registro   | \$r                  |
| ▶ Directo   | dir                  |
| ▶ Indirecto de registro                                       | (\$r)                |
| ▶ Relativo a registro   | valor(\$r)           |
| ▶ valor puede representar una dirección (registro base)       |                      |
| ▶ valor puede representar un desplazamiento (registro índice) |                      |
| ▶ Relativo a PC   | beq etiqueta         |
| ▶ Relativo a pila   | desplazamiento(\$sp) |

# Ejercicio

- ▶ Indique el tipo de direccionamiento usado en las siguientes instrucciones MIPS:
  1. li \$t1 4
  2. lw \$t0 4(\$a0)
  3. bnez \$a0 etiqueta

# Ejercicio (solución)

1. **li \$t1 4**

- ▶ **\$t1** → directo a registro
- ▶ **4** → inmediato

2. **lw \$t0 4(\$a0)**

- ▶ **\$t0** → directo a registro
- ▶ **4(\$a0)** → relativo a registro base

3. **bnez \$a0 etiqueta**

- ▶ **\$a0** → directo a registro
- ▶ **etiqueta** → relativo a contador de programa

# Ejemplos de tipos de direccionamiento

# Juego de instrucciones

- ▶ Queda **definido** por:
  - ▶ Conjunto de instrucciones
  - ▶ Formato de la instrucciones
  - ▶ Registros
  - ▶ Modos de direccionamiento
  - ▶ Tipos de datos y formatos

# Juego de instrucciones

- ▶ Distintas formas para la **clasificación** de un juego de instrucciones:
  - ▶ Complejidad del juego de instrucciones
    - ▶ CISC vs RISC
  - ▶ Modo de ejecución
    - ▶ Pila
    - ▶ Registro
    - ▶ Registro-Memoria, Memoria-Registro, ...

# Formato de instrucciones

- ▶ Una instrucción máquina es autocontenido e incluye:
  - ▶ Código de operación
  - ▶ Dirección de los operandos
  - ▶ Dirección del resultado
  - ▶ Dirección de la siguiente instrucción
  - ▶ Tipos de representación de los operandos
- ▶ Una instrucción se divide en **campos**
- ▶ Ejemplo de campos en una instrucción del MIPS:



# Formato de instrucciones

- ▶ Una instrucción normalmente ocupa una palabra pero puede ocupar más en algunos computadores
  - ▶ En el caso del MIPS todas las instrucciones ocupan una palabra
- ▶ Campo de código:
  - ▶ Con  $n$  bits se pueden codificar  $2^n$  instrucciones
  - ▶ Si se quiere codificar más se utiliza un campo de extensión
  - ▶ Ejemplo: en el MIPS las instrucciones aritméticas tienen como código de op = 0. La función concreta se codifica en el campo func.

Tipo R  
aritméticas



## Formato de una instrucción

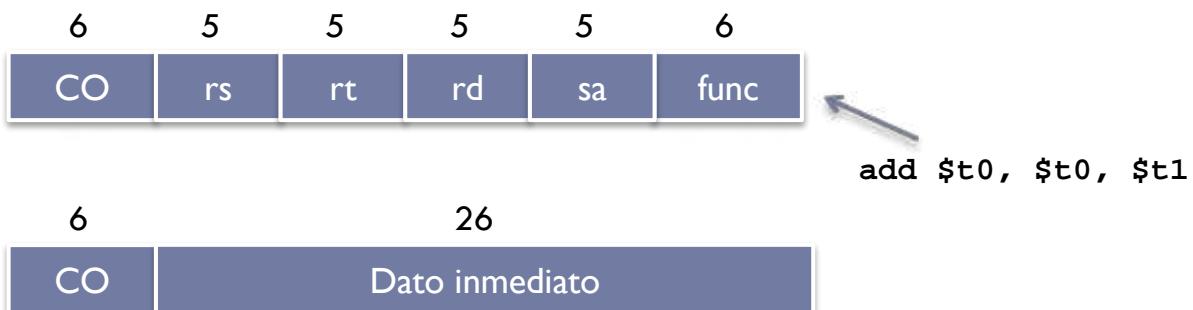
- ▶ Especifica el significado de cada uno de los bits que forma la instrucción.
- ▶ Longitud del formato: Número de bits que componen la instrucción.
- ▶ La instrucción se divide en campos.
- ▶ Normalmente una arquitectura ofrece unos pocos formatos de instrucción.
  - ▶ Simplicidad en el diseño de la unidad de control.
- ▶ Uso sistemático:
  - ▶ Campos del mismo tipo siempre igual longitud.
  - ▶ Selección mediante código de operación.
    - ▶ Normalmente el primer campo.

# Longitud de formato

## ▶ Alternativas:

- ▶ Longitud única: Todas las instrucciones tienen la misma longitud de formato.
  - ▶ MIPS32: 32 bits
  - ▶ PowerPC: 32 bits
- ▶ Longitud variable: Distintas instrucciones tienen distinta longitud de formato.
  - ▶ ¿Cómo se sabe la longitud de la instrucción? → Cod. Op.
  - ▶ IA32 (Procesadores Intel): Número variable de bytes.

## Ejemplo: Formato de las instrucciones del MIPS

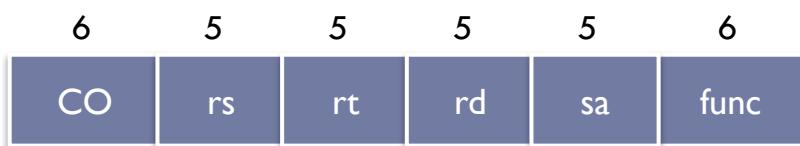


# Ejemplo de formato en el MIPS

- ▶ **MIPS Instruction:**

- ▶ `add $8 , $9 , $10`

- ▶ **Formato a utilizar :**



- ▶ Representación decimal de cada campo:

0	9	10	8	0	32
---	---	----	---	---	----

- ▶ Representación binaria de cada campo:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

# Ejemplo de formato en el MIPS

- ▶ **MIPS Instruction:**

- ▶ addi \$21,\$22,-50

- ▶ **Formato a utilizar :**



Representación decimal de cada campo:

8	22	21	-50
---	----	----	-----

Representación binaria de cada campo

001000	10110	10101	1111111111001110
--------	-------	-------	------------------

## ¿Cómo utilizar addi con un valor de 32 bits?

- ▶ **¿Qué ocurre si se utiliza desde el ensamblador?**
  - ▶ `addi $t0,$t0, 0xABABCD`
  - ▶ El valor inmediato es de 32 bits. Esta instrucción no se puede codificar en una palabra de 32 bits.

## ¿Cómo utilizar addi con un valor de 32 bits?

- ▶ **¿Qué ocurre si se utiliza desde el ensamblador?**
  - ▶ addi \$t0,\$t0, 0xABABCD
  - ▶ El valor inmediato es de 32 bits. Esta instrucción no se puede codificar en una palabra de 32 bits.
- ▶ **Solución:**
  - ▶ Desde el ensamblador se puede utilizar, pero al final se traduce en:

```
lui      $at, 0xABAB
ori      $at, $at, 0xCD
add      $t0, $t0, $at
```
  - ▶ El registro \$at está reservado para el ensamblador por convenio

## Ejercicio

- ▶ ¿Cómo sabe la unidad de control el formato de la instrucción que está ejecutando?
- ▶ ¿Cómo sabe la unidad de control el número de operandos de una instrucción?
- ▶ ¿Cómo sabe la unidad de control el formato de cada operación?

## Código de operación

- ▶ **Tamaño fijo:**
  - ▶  $n$  bits →  $2^n$  códigos de operación.
  - ▶  $m$  códigos de operación →  $\log_2 m$  bits.
- ▶ **Campos de extensión**
  - ▶ MIPS (instrucciones aritméticas-lógicas)
  - ▶ Op = 0; la instrucción está codificada en func

Tipo R  
aritméticas



- ▶ **Tamaño variable:**
  - ▶ Instrucciones más frecuentes = Tamaños más cortos.

# Ejercicio

- ▶ Sea un computador de 16 bits de tamaño de palabra, que incluye un repertorio con 60 instrucciones máquina y con un banco de registros que incluye 8 registros.

Se pide:

Indicar el formato de la instrucción **ADDx R1 R2 R3**, donde R1, R2 y R3 son registros.

## Ejercicio (solución)

palabra -> 16 bits  
60 instrucciones  
8 registros (en BR)  
ADDx R1(reg.), R2(reg.), R3(reg.)

- ▶ Palabra de 16 bits define el tamaño de la instrucción

16 bits



## Ejercicio (solución)

palabra -> 16 bits  
60 instrucciones  
8 registros (en BR)  
ADDx R1(reg.), R2(reg.), R3(reg.)

- ▶ Para 60 instrucciones se necesitan 6 bits (mínimo)



## Ejercicio (solución)

- palabra -> 16 bits
- 60 instrucciones
- 8 registros (en BR)
- ADDx R1(reg.), R2(reg.), R3(reg.)

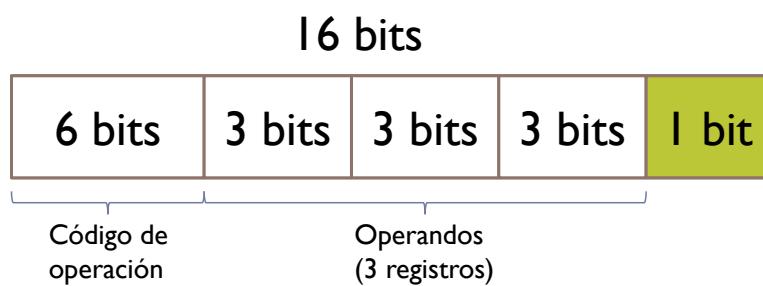
- ▶ Para 8 registros se necesitan 3 bits (mínimo)



## Ejercicio (solución)

palabra -> 16 bits  
60 instrucciones  
8 registros (en BR)  
ADDx R1(reg.), R2(reg.), R3(reg.)

- ▶ Sobra 1 bit ( $16-6-3-3-3 = 1$ ), usado de relleno



## Ejercicio

- ▶ Sea un computador de 16 bits, que direcciona la memoria por bytes y que incluye un repertorio con 60 instrucciones máquina. El banco de registros incluye 8 registros. Indicar el formato de la instrucción ADDV RI, R2, M, donde RI y R2 son registros y M es una dirección de memoria.

## Ejercicio

- ▶ Sea un computador de 32 bits, que direcciona la memoria por bytes. El computador incluye 64 instrucciones máquina y 128 registros. Considere la instrucción SWAPM dir1, dir2, que intercambia el contenido de las posiciones de memoria dir1 y dir2. Se pide:
  - ▶ Indicar el espacio de memoria direccionable en este computador.
  - ▶ Indicar el formato de la instrucción anterior.
  - ▶ Especifique un fragmento de programa en ensamblador del MIPS 32 equivalente a la instrucción máquina anterior.
  - ▶ Si se fuerza a que la instrucción quepa en una palabra, qué rango de direcciones se podría contemplar considerando que las direcciones se representan en binario puro.

## Ejercicio

- ▶ Sea un computador de 32 bits, que direcciona la memoria por bytes. El computador incluye 64 instrucciones máquina y 128 registros. Considere la instrucción SWAPM dir1, dir2, que intercambia el contenido de las posiciones de memoria dir1 y dir2. Se pide:
  - ▶ Indicar el espacio de memoria direccionable en este computador.
  - ▶ Indicar el formato de la instrucción anterior.
  - ▶ Especifique un fragmento de programa en ensamblador del MIPS 32 equivalente a la instrucción máquina anterior.
  - ▶ si se fuerza a que la instrucción quepa en una palabra, qué rango de direcciones se podría contemplar considerando que las direcciones se representan en binario puro.

# Juego de instrucciones

- ▶ Distintas formas para la **clasificación** de un juego de instrucciones:
  - ▶ Complejidad del juego de instrucciones
    - ▶ CISC vs RISC
  - ▶ Modo de ejecución
    - ▶ Pila
    - ▶ Registro
    - ▶ Registro-Memoria, Memoria-Registro, ...

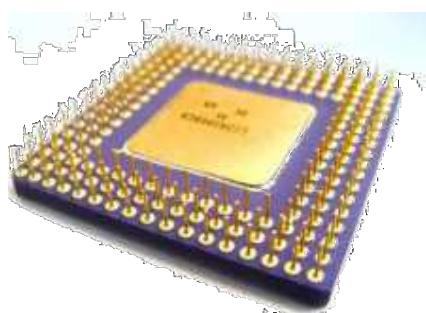
# CISC

- ▶ *Complex Instruction Set Computer*
- ▶ Muchas instrucciones
- ▶ Complejidad variable
  - ▶ Instrucciones complejas
    - ▶ Más de una palabra
    - ▶ Unidad de control más compleja
    - ▶ Mayor tiempo de ejecución
- ▶ Diseño irregular

# CISC vs RISC

## ▶ Observación:

- ▶ Alrededor del 20% de las instrucciones ocupa el 80% del tiempo total de ejecución de un programa
- ▶ El 80% de las instrucciones no se utilizan casi nunca
- ▶ 80% del silicio infrautilizado, complejo y costoso



# RISC

- ▶ *Reduced Instruction Set Computer*
- ▶ Juegos de instrucciones reducidos
- ▶ Instrucciones simples y ortogonales
  - ▶ Ocupan una palabra
  - ▶ Instrucciones sobre registros
  - ▶ Uso de los mismos modos de direccionamiento para todas las instrucciones (alto grado de ortogonalidad)
- ▶ Diseño más compacto:
  - ▶ Unidad de control más sencilla y rápida
  - ▶ Espacio sobrante para más registros y memoria caché

# Juego de instrucciones

- ▶ Distintas formas para la **clasificación** de un juego de instrucciones:
  - ▶ Complejidad del juego de instrucciones
    - ▶ CISC vs RISC
  - ▶ Modo de ejecución
    - ▶ Pila
    - ▶ Registro
    - ▶ Registro-Memoria, Memoria-Registro, ...

# Modelo de ejecución

- ▶ Una máquina tiene un **modelo de ejecución** asociado.
  - ▶ Modelo de ejecución indica el número de direcciones y tipo de operandos que se pueden especificar en una instrucción.
- ▶ Modelos de ejecución:
  - ▶ 0 direcciones → Pila
  - ▶ 1 dirección → Registro acumulador
  - ▶ 2 direcciones → Registros, Registro-Memoria y Memoria-Memoria
  - ▶ 3 direcciones → Registros, Registro-Memoria y Memoria-Memoria

# Modelo de 3 direcciones

## ► Registro-Registro:

- ▶ Los 3 operandos son registros.
- ▶ Requiere operaciones de carga/almacenamiento.
- ▶ **ADD .R0,.R1,.R2**

## ► Memoria-Memoria:

- ▶ Los 3 operandos son direcciones de memoria.
- ▶ **ADD /DIR1,/DIR2,/DIR3**

## ► Registro-Memoria:

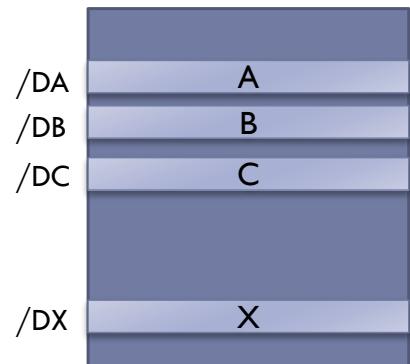
- ▶ Híbrido.
- ▶ **ADD .R0,/DIR1,/DIR2**
- ▶ **ADD .R0,.R1,/DIR1**

# Ejercicio

- ▶ Sea la siguiente expresión matemática:

- ▶  $X = A + B * C$

Donde los operandos están en memoria tal y como se describe en la figura:

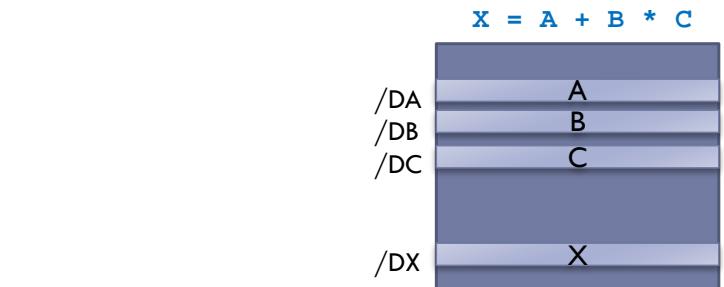


Para los modelos R-R y M-M, indique:

- ▶ El número de instrucciones
- ▶ Accesos a memoria
- ▶ Accesos a registros

# Ejercicio (solución)

- ▶ Memoria-Memoria:



```
MUL /DX, /DB, /DC  
ADD /DX, /DX, /DA
```

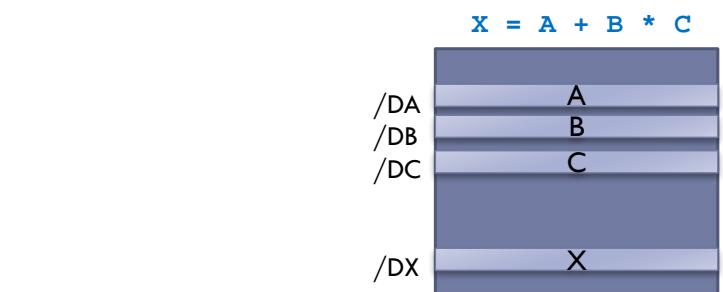
- ▶ Registro-Registro:

```
LOAD R0, /DB  
LOAD R1, /DC  
MUL R0, R0, R1  
LOAD R2, /DA  
ADD R0, R0, R2  
STORE R0, /DX
```

# Ejercicio (solución)

## ▶ Memoria-Memoria:

- ▶ 2 instrucciones
- ▶ 6 accesos a memoria
- ▶ 0 accesos a registros



MUL /DX, /DB, /DC  
ADD /DX, /DX, /DA

## ▶ Registro-Registro:

- ▶ 6 instrucciones
- ▶ 4 accesos a memoria
- ▶ 10 accesos a registros

LOAD R0, /DB  
LOAD R1, /DC  
MUL R0, R0, R1  
LOAD R2, /DA  
ADD R0, R0, R2  
STORE R0, /DX

# Modelo de 2 direcciones

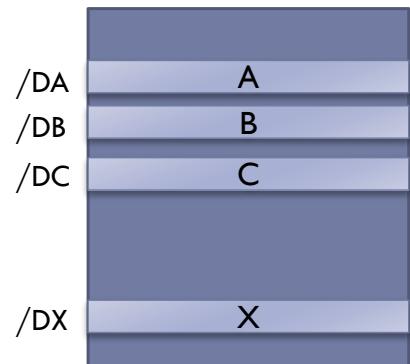
- ▶ **Registro-Registro:**
  - ▶ Los 2 operandos son registros.
  - ▶ Requiere operaciones de carga/almacenamiento.
  - ▶ **ADD R0, RI            (R0 <- R0 + RI)**
- ▶ **Memoria-Memoria:**
  - ▶ Los 2 operandos son direcciones de memoria.
  - ▶ **ADD /DIR1,/DIR2    (MP[DIR1] <- MP[DIR1] + MP[DIR2])**
- ▶ **Registro-Memoria:**
  - ▶ Híbrido.
  - ▶ **ADD R0,/DIR1        (R0 <- R0 + MP[DIR1])**

## Ejercicio

- ▶ Sea la siguiente expresión matemática:

- ▶  $X = A + B * C$

Donde los operandos están en memoria tal y como se describe en la figura:

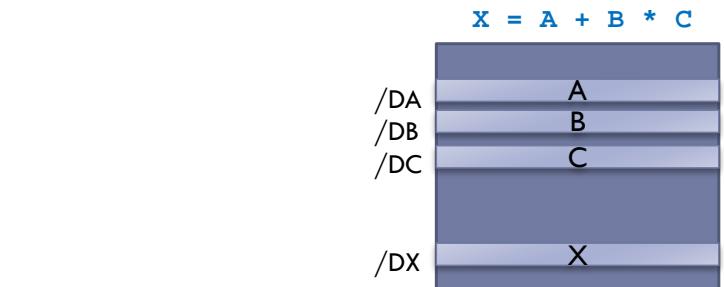


Para los modelos R-R y M-M, indique:

- ▶ El número de instrucciones
- ▶ Accesos a memoria
- ▶ Accesos a registros

# Ejercicio (solución)

## ▶ Memoria-Memoria:



```
MOVE  /DX, /DB  
MUL   /DX, /DC  
ADD   /DX, /DA
```

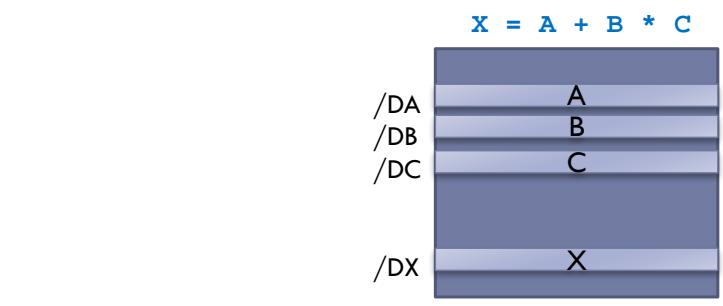
## ▶ Registro-Registro:

```
LOAD  R0, /DB  
LOAD  R1, /DC  
MUL   R0, R1  
LOAD  R2, /DA  
ADD   R0, R2  
STORE R0, /DX
```

## Ejercicio (solución)

### ▶ Memoria-Memoria:

- ▶ 3 instrucciones
- ▶ 6 accesos a memoria
- ▶ 0 accesos a registros



MOVE /DX, /DB  
MUL /DX, /DC  
ADD /DX, /DA

### ▶ Registro-Registro:

- ▶ 6 instrucciones
- ▶ 4 accesos a memoria
- ▶ 8 accesos a registros

LOAD R0, /DB  
LOAD R1, /DC  
MUL R0, R1  
LOAD R2, /DA  
ADD R0, R2  
STORE R0, /DX

## Modelo de 1 dirección

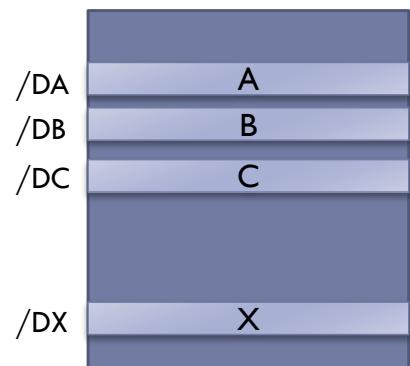
- ▶ Todas las operaciones utilizan un operando implícito:
  - ▶ Registro acumulador
  - ▶ **ADD RI**                            **(AC <- AC + RI)**
- ▶ Operaciones de carga y almacenamiento siempre sobre el acumulador.
- ▶ Posibilidad de movimiento entre el registro acumulador y otros registros

# Ejercicio

- ▶ Sea la siguiente expresión matemática:

- ▶  $X = A + B * C$

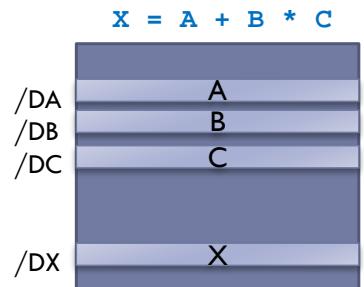
Donde los operandos están en memoria tal y como se describe en la figura:



Para el modelo de 1 dirección, indique:

- ▶ El número de instrucciones
- ▶ Accesos a memoria
- ▶ Accesos a registros

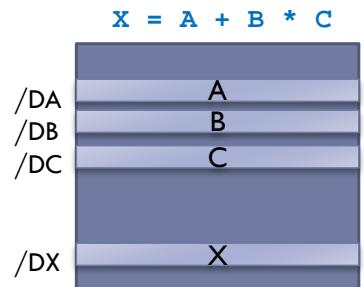
## Ejercicio (solución)



- ▶ Modelo de I sola dirección:

```
LOAD  /DB
MUL   /DC
ADD   /DA
STORE /DX
```

## Ejercicio (solución)



### ► Modelo de I sola dirección:

- 4 instrucciones
- 4 accesos a memoria
- 0 accesos a registros

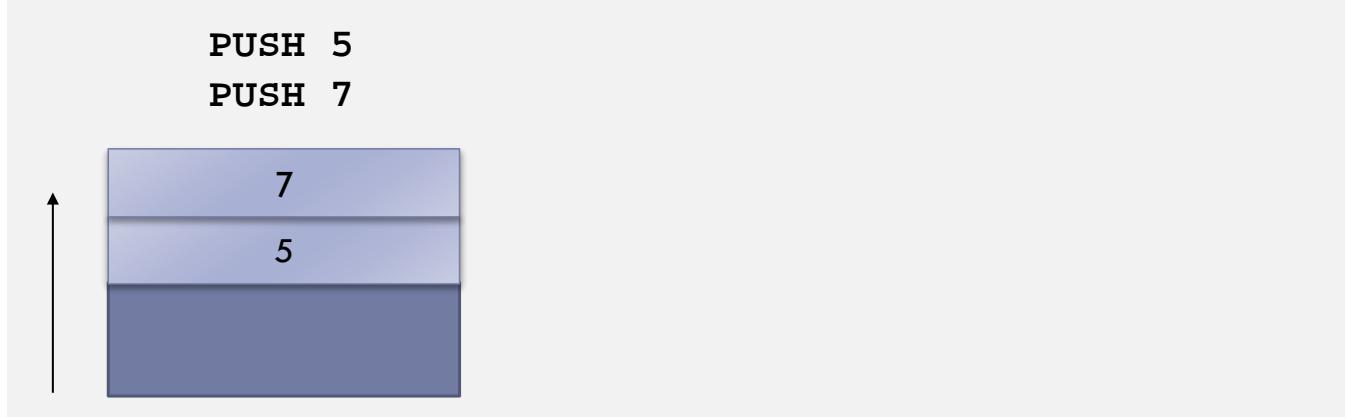
LOAD /DB  
MUL /DC  
ADD /DA  
STORE /DX

# Modelo de 0 direcciones

- ▶ Todas las operaciones referidas a la **pila**:
  - ▶ Los operandos están en la cima de la pila.
    - ▶ Al hacer la operación se retiran de la pila.
  - ▶ El resultado se coloca en la cima de la pila.
  - ▶ **ADD**
- ▶ Dos operaciones especiales:
  - ▶ **PUSH**
  - ▶ **POP**

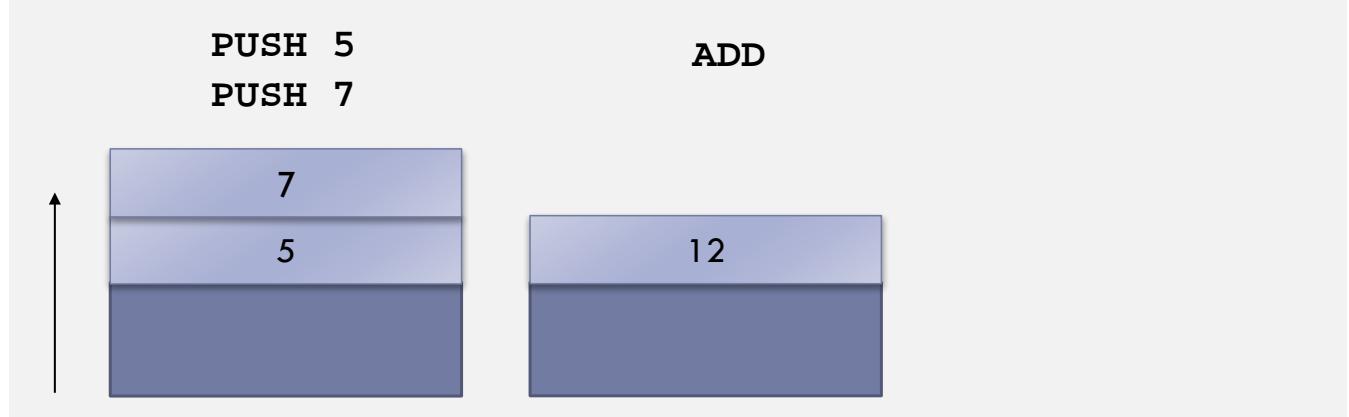
# Ejemplo

```
push 5  
push 7
```



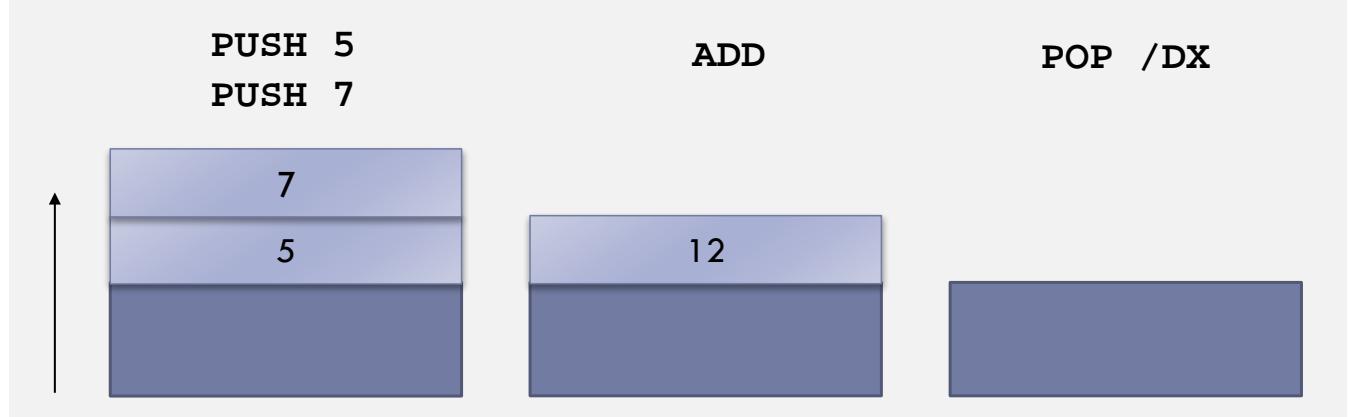
# Ejemplo

```
push 5  
push 7  
add
```



# Ejemplo

```
push 5  
push 7  
add  
pop /dx
```

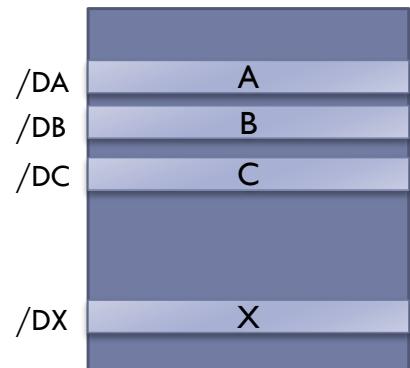


# Ejercicio

- ▶ Sea la siguiente expresión matemática:

- ▶  $X = A + B * C$

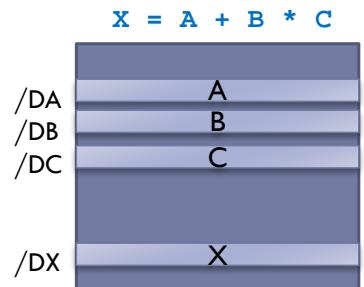
Donde los operandos están en memoria tal y como se describe en la figura:



Para el modelo de 0 dirección, indique:

- ▶ El número de instrucciones
- ▶ Accesos a memoria
- ▶ Accesos a registros

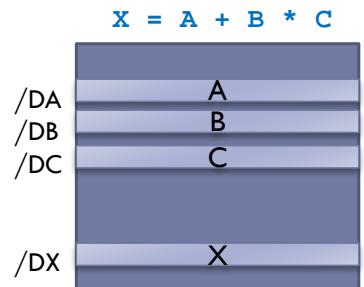
## Ejercicio (solución)



### ► Modelo de 0 direcciones:

```
PUSH /DB  
PUSH /DC  
MUL  
PUSH /DA  
ADD  
POP /DX
```

## Ejercicio (solución)



### ► Modelo de 0 direcciones:

- ▶ 6 instrucciones
- ▶ 4 accesos a memoria (datos)
- ▶ 10 accesos a memoria (pila)
- ▶ 0 accesos a registros

PUSH /DB  
PUSH /DC  
MUL  
PUSH /DA  
ADD  
POP /DX

Grupo ARCOS

**uc3m | Universidad Carlos III de Madrid**

## Tema 3 (IV) Fundamentos de la programación en ensamblador

Estructura de Computadores  
Grado en Ingeniería Informática



# Contenido

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del MIPS 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ **Llamadas a procedimientos y uso de la pila**

# Procedimientos

- ▶ Un procedimiento (función, subrutina) es un subprograma que realiza una tarea específica cuando se le invoca
  - ▶ Recibe argumentos o parámetros de entrada
  - ▶ Devuelve algún resultado
- ▶ En ensamblador un procedimiento se asocia con un nombre simbólico que denota su dirección de inicio (la dirección de memoria donde se encuentra la primera instrucción de la subrutina)

## Pasos en la ejecución de un procedimiento/función

- ▶ Situar los parámetros en un lugar donde el procedimiento pueda accederlos
- ▶ Transferir el control al procedimiento
- ▶ Adquirir los recursos de almacenamiento necesarios para el procedimiento
- ▶ Realizar la tarea deseada
- ▶ Poner el resultado en un lugar donde el programa o procedimiento que realiza la llamada pueda accederlo
- ▶ Devolver el control al punto de origen

# Funciones en un lenguaje de alto nivel

```
int main() {
    int z;
    z=factorial(x);
    print_int(z);
}

int factorial(int x) {
    int i;
    int r=1;
    for (i=1;i<=x;i++) {
        r*=i;
    }
    return r;
}
```

# Funciones en un lenguaje de alto nivel

```
int main() {  
    int z;  
    z=factorial(x);  
    print_int(z);  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```



# Funciones en un lenguaje de alto nivel

```
int main() {  
    int z;  
    z=factorial(x);  
    print_int(z);  
}
```

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

# Funciones en un lenguaje de alto nivel

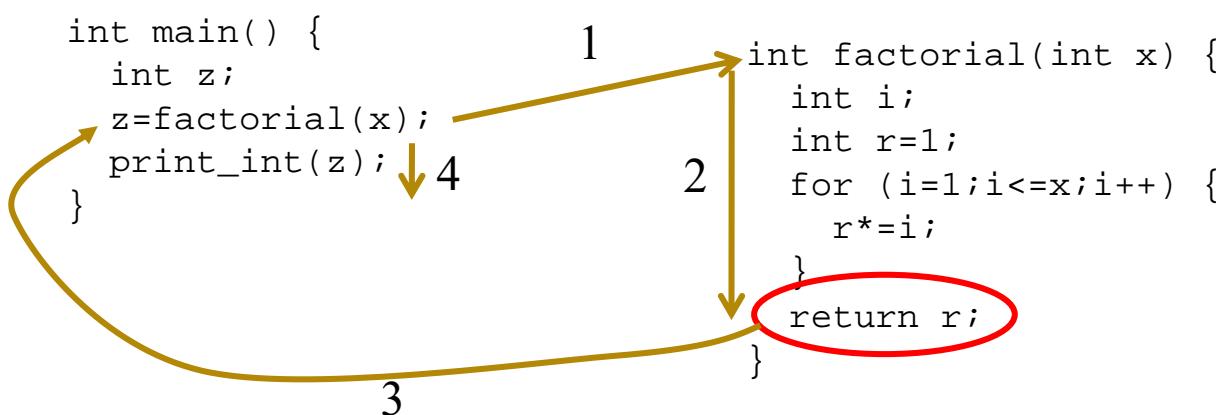
```
int main() {  
    int z;  
    z=factorial(x);  
    print_int(z);  
}
```

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

# Funciones en un lenguaje de alto nivel

```
int main() {  
    int z;  
    z=factorial(x);  
    print_int(z); ↓  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

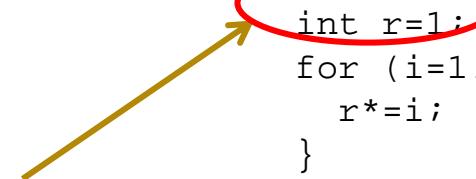
# Funciones en un lenguaje de alto nivel



# Funciones en un lenguaje de alto nivel

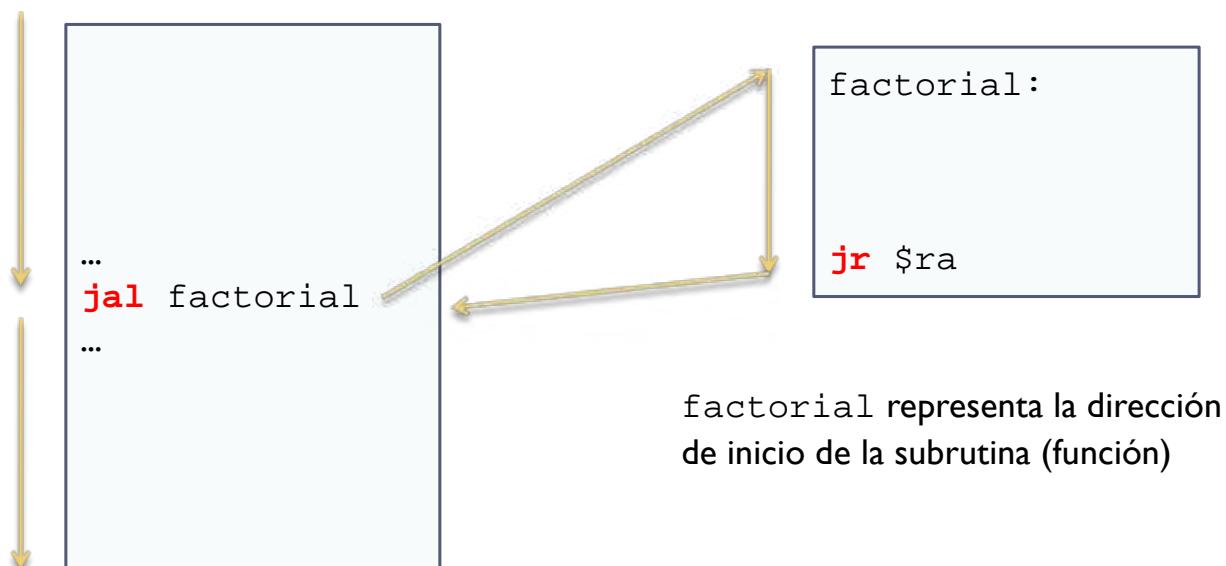
```
int main() {  
    int z;  
    z=factorial(x);  
    print_int(z);  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

Variables locales

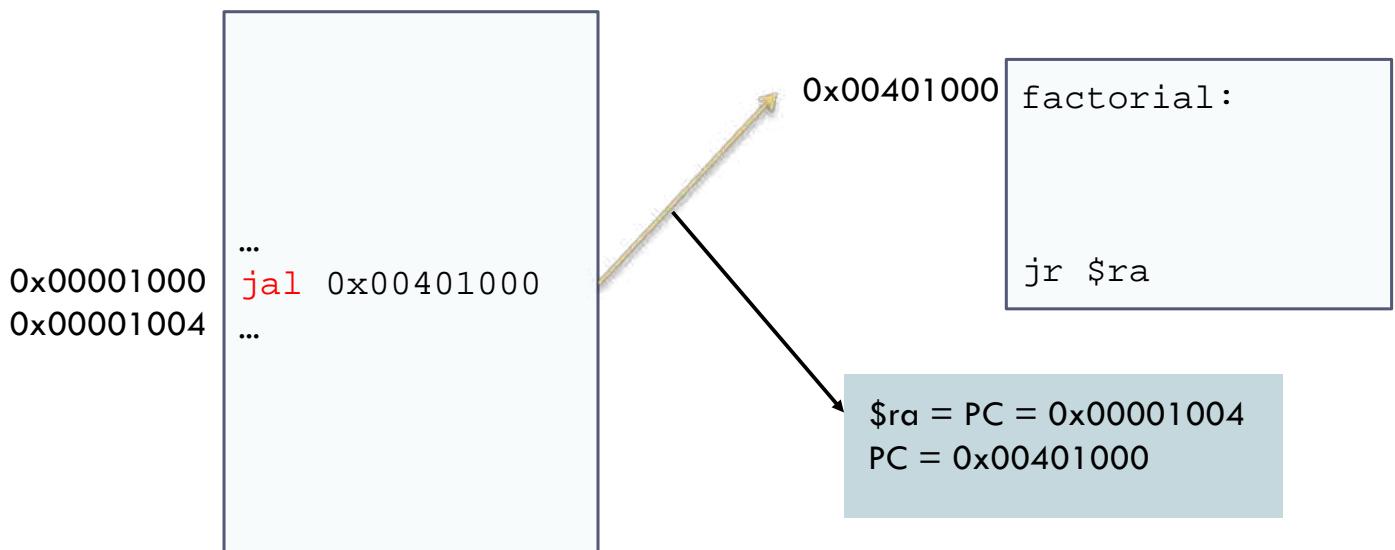


# Llamadas a funciones en MIPS

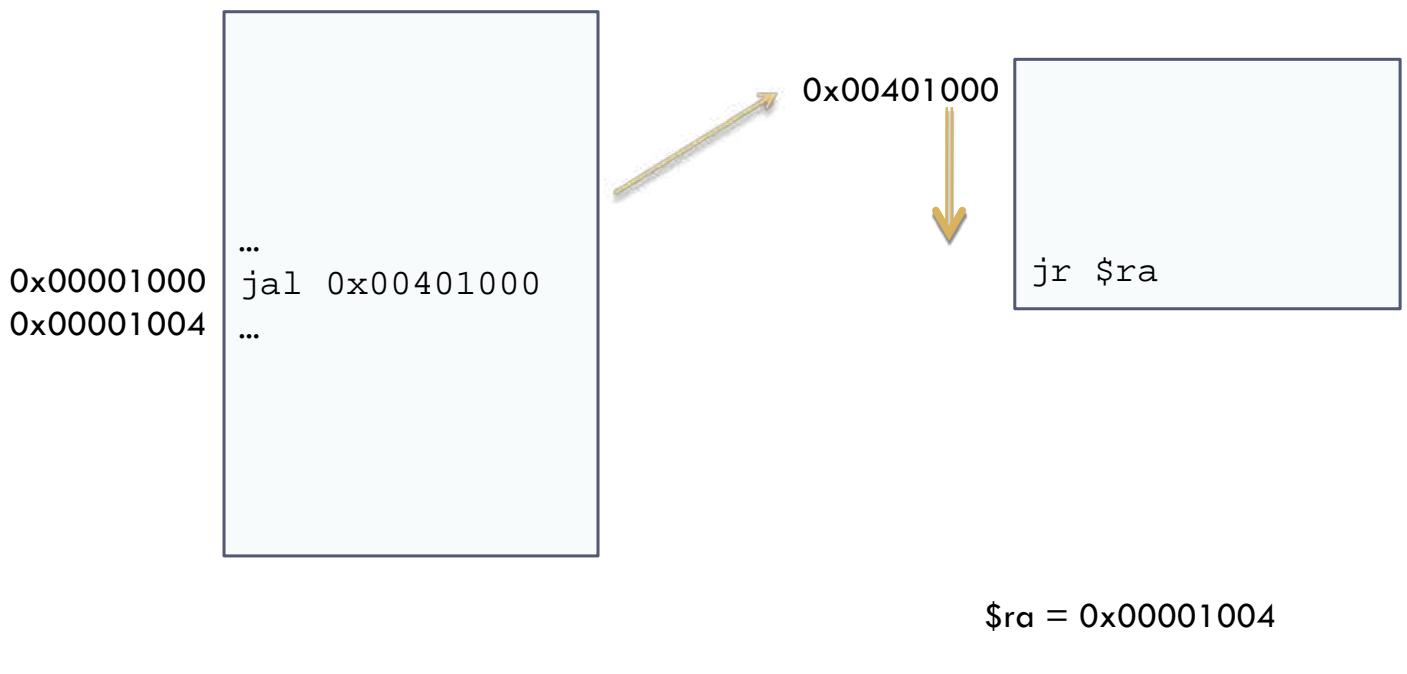
Llamada a función en el MIPS (instrucción jal)



## Llamadas a funciones en MIPS

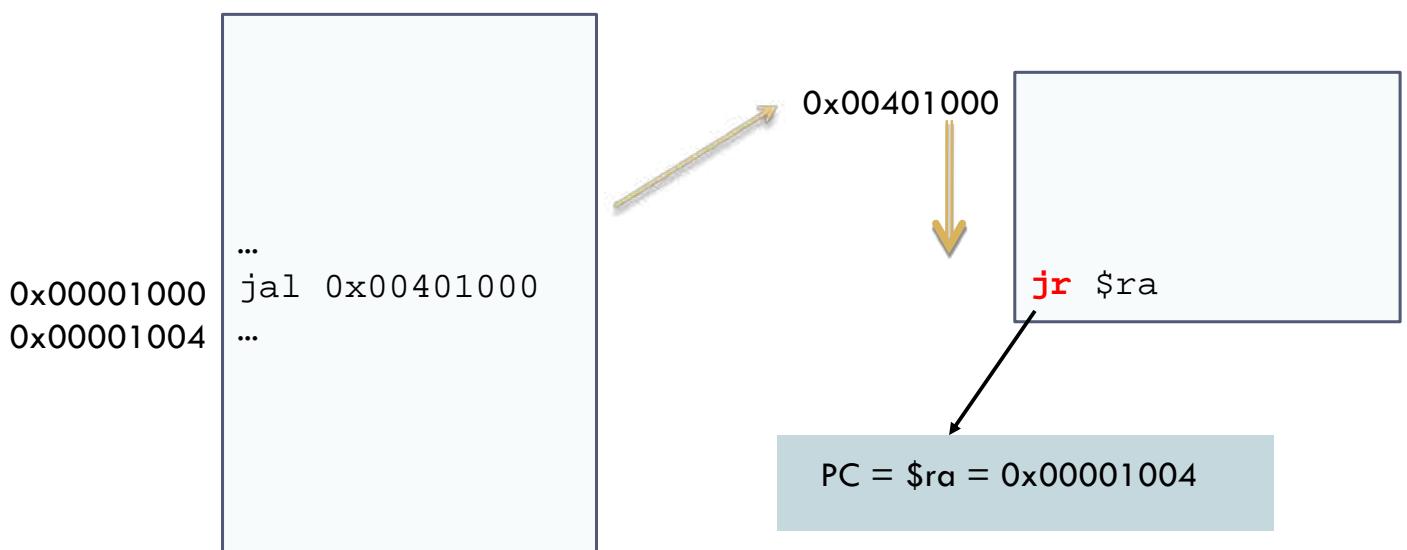


## Llamadas a funciones en MIPS



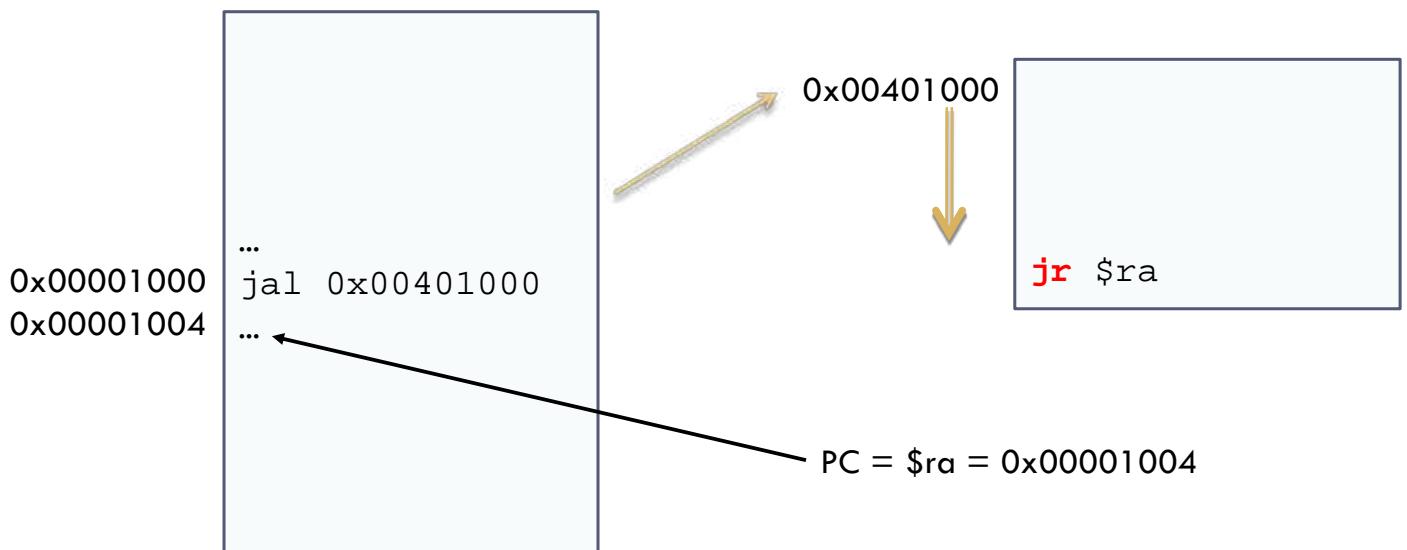
# Llamadas a funciones en MIPS

Retorno de subrutina (instrucción jr )



**\$ra = 0x00001004**

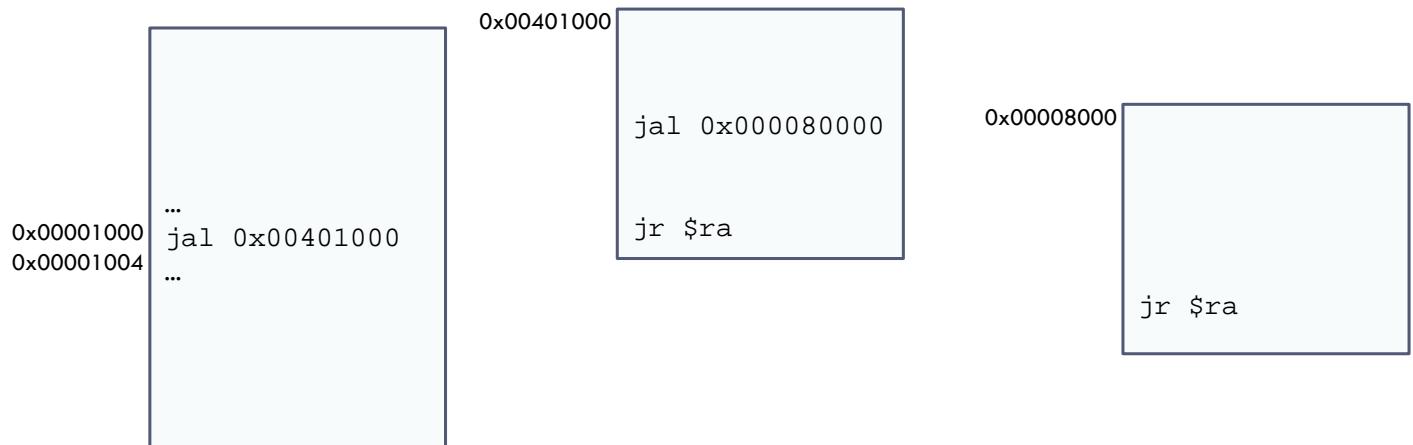
## Llamadas a funciones en MIPS



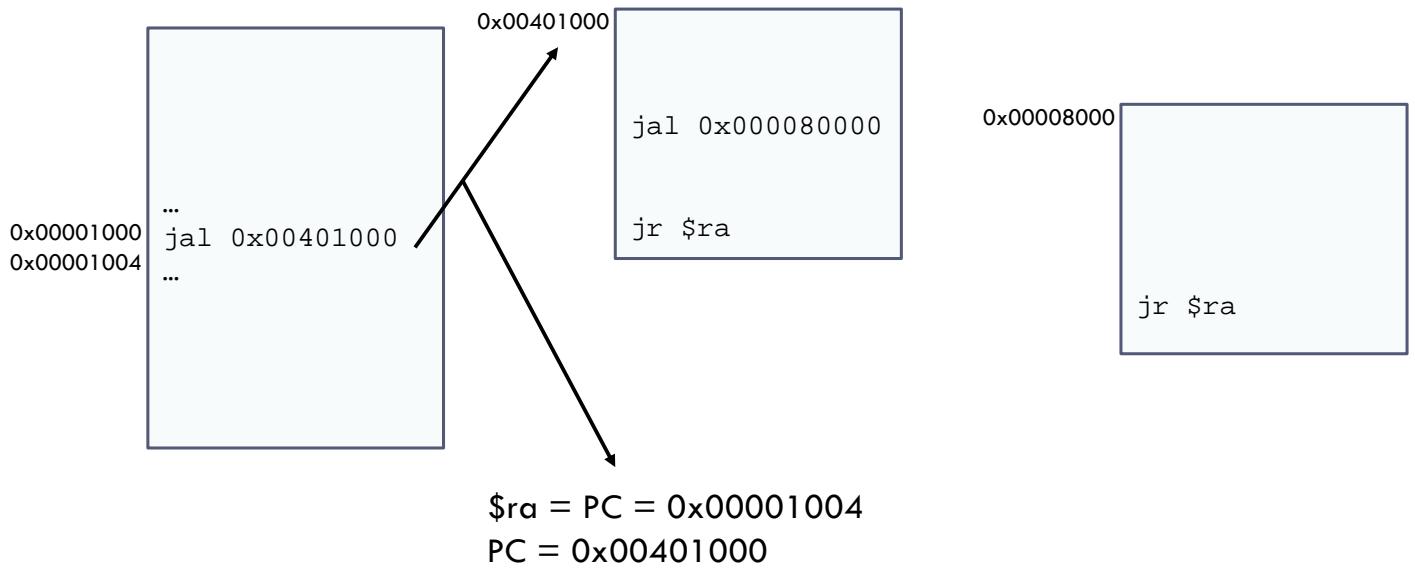
## Instrucciones jal / jr

- ▶ ¿Qué hace la instrucción jal?
  - ▶ \$ra  $\leftarrow$  \$PC
  - ▶ \$PC  $\leftarrow$  Dirección de salto
- ▶ ¿Qué hace la instrucción jr?
  - ▶ \$PC  $\leftarrow$  \$ra

## Llamadas anidadas

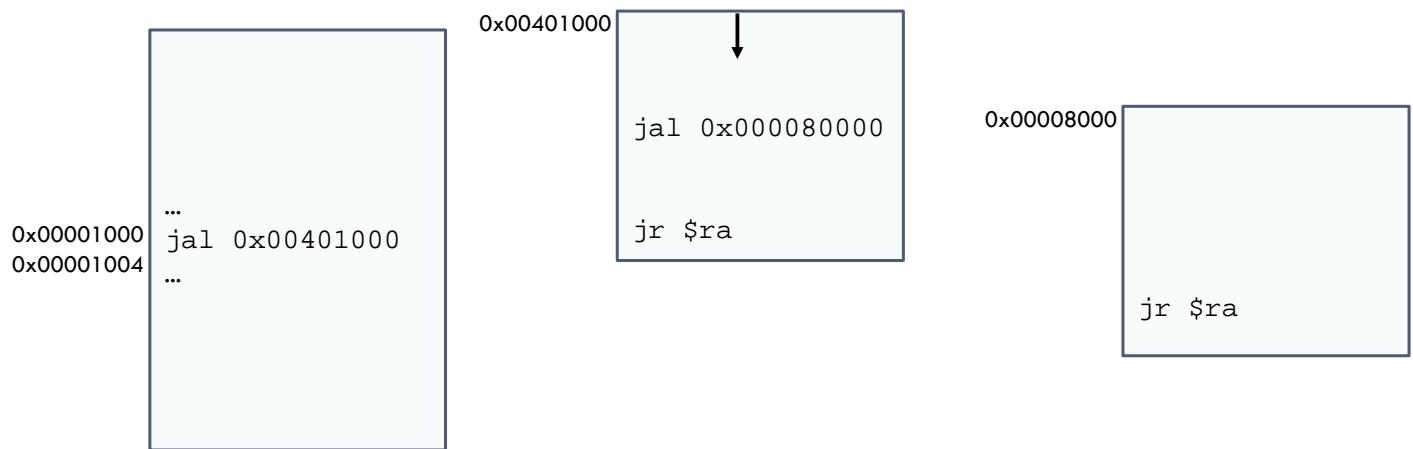


# Llamadas anidadas



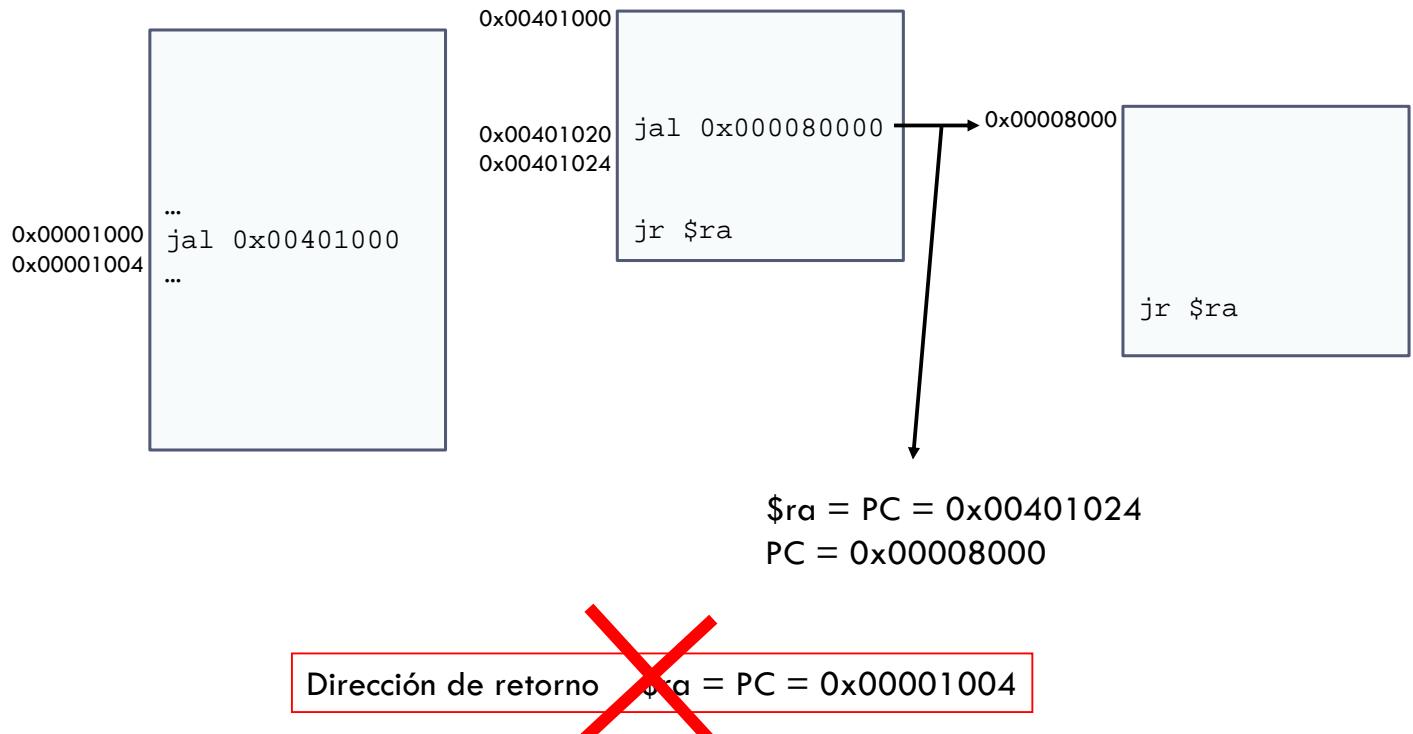
Dirección de retorno     $\$ra = PC = 0x00001004$

# Llamadas anidadas

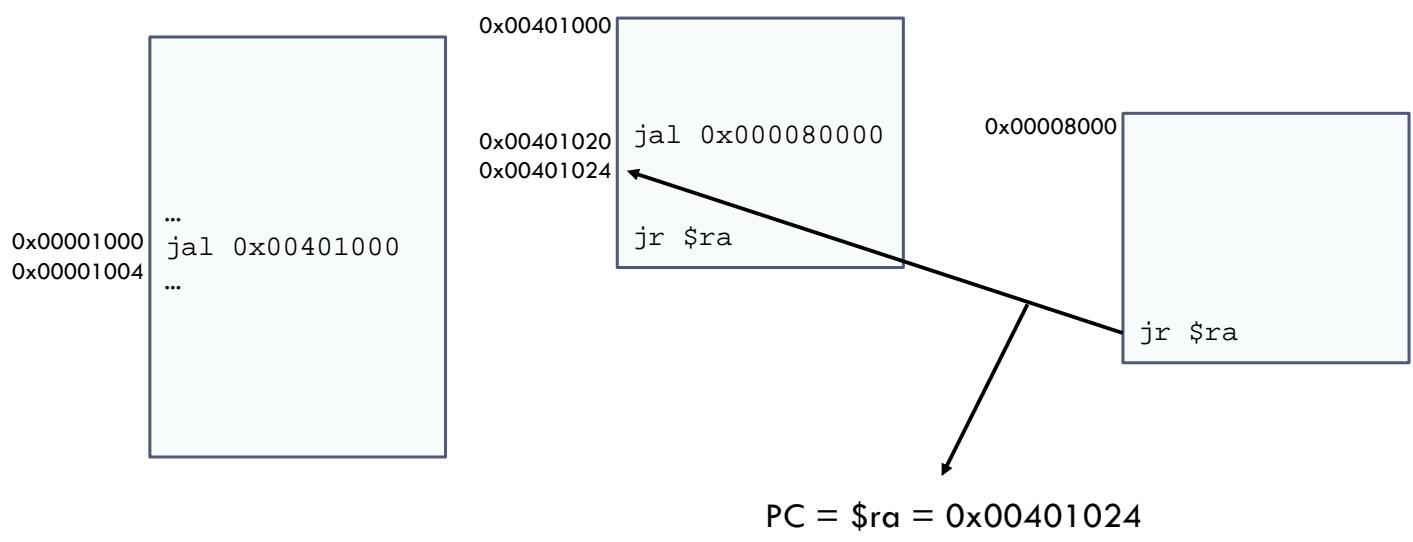


Dirección de retorno \$ra = PC = 0x00001004

# Llamadas anidadas

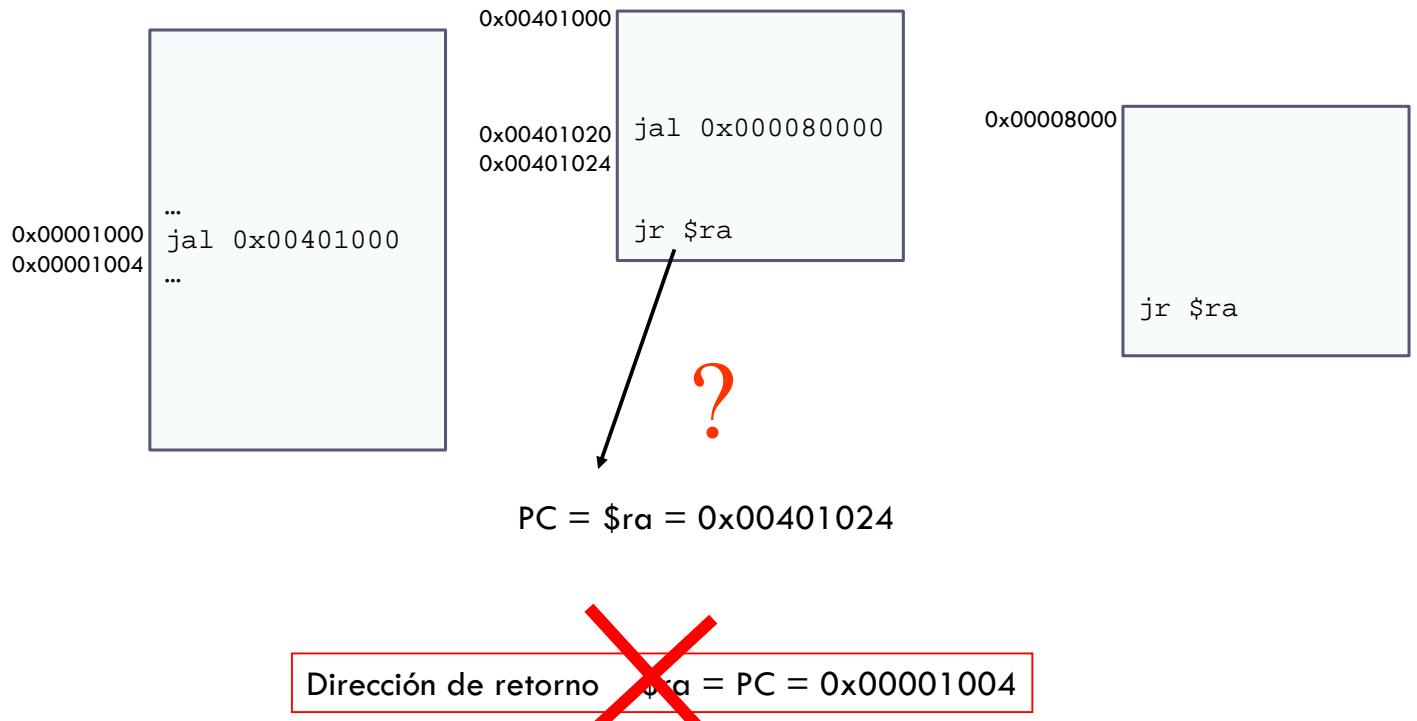


# Llamadas anidadas

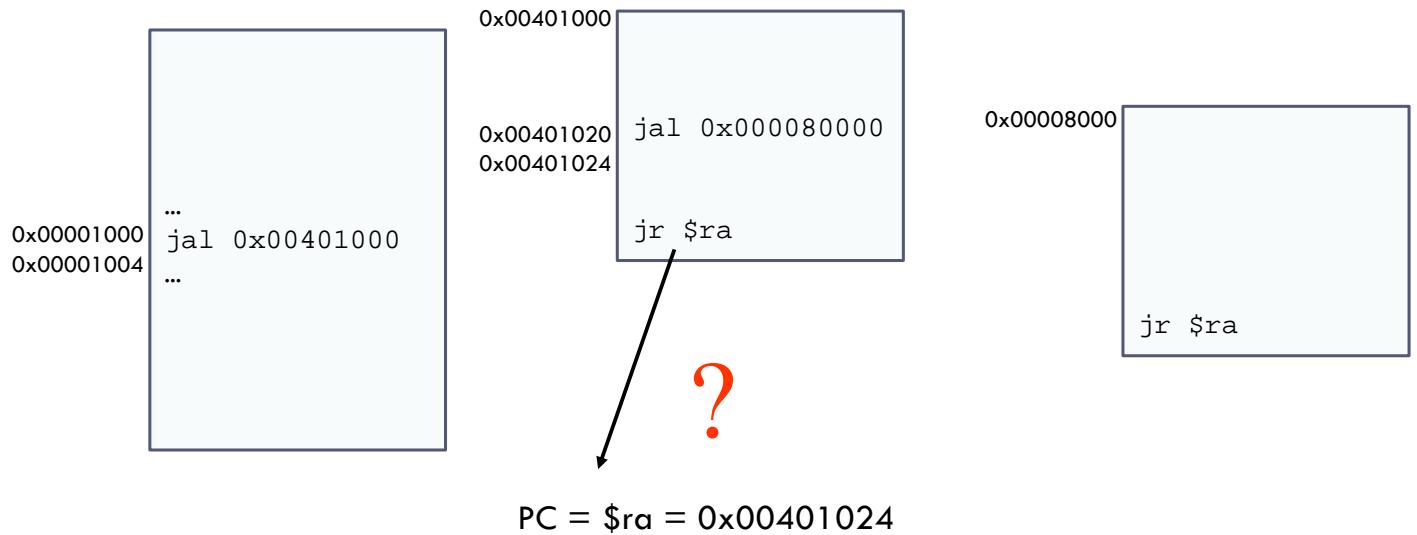


Dirección de retorno  ~~$\$ra = PC = 0x00001004$~~

# Llamadas anidadas



# Llamadas anidadas

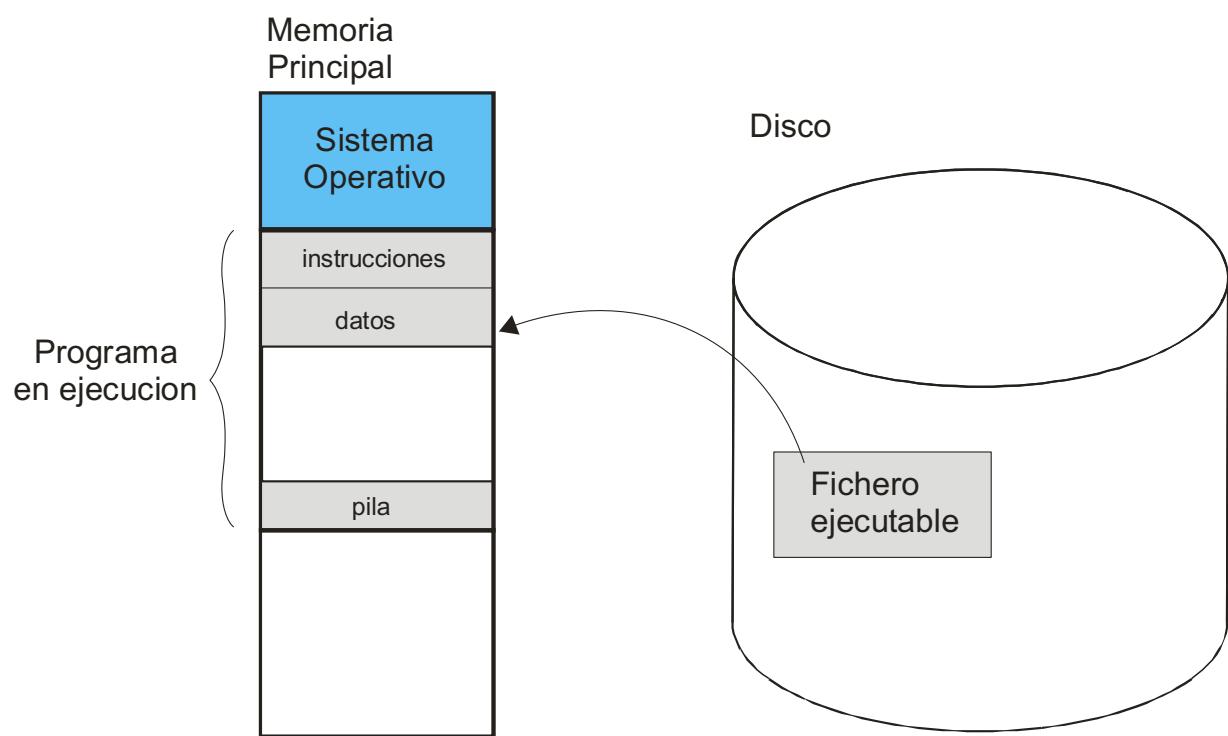


Se ha perdido la dirección de retorno

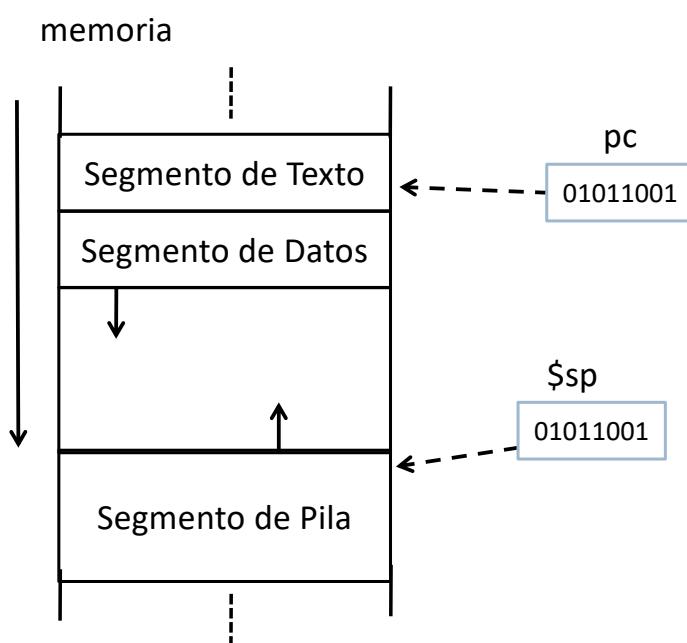
## ¿Dónde guardar la dirección de retorno?

- ▶ El computador dispone de dos elementos para almacenamiento:
  - ▶ Registros
  - ▶ Memoria
- ▶ No se pueden utilizar los registros porque su número es limitado
- ▶ Se guarda en memoria principal
  - ▶ En una zona del programa que se denomina **pila**

# Ejecución de un programa



# Mapa de memoria de un proceso

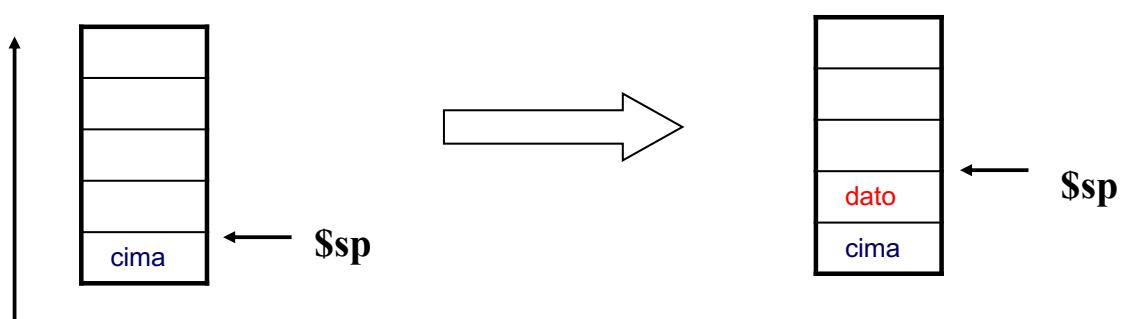


- ▶ El programa de usuario se divide en segmentos:
  - ▶ Segmento de código (texto)
    - ▶ Código, instrucciones máquina
  - ▶ Segmento de datos
    - ▶ Datos estáticos, variables globales
  - ▶ Segmento de pila
    - ▶ Variables locales
    - ▶ Contexto de funciones

# Pila

**PUSH Reg**

Apila el contenido del registro (dato)

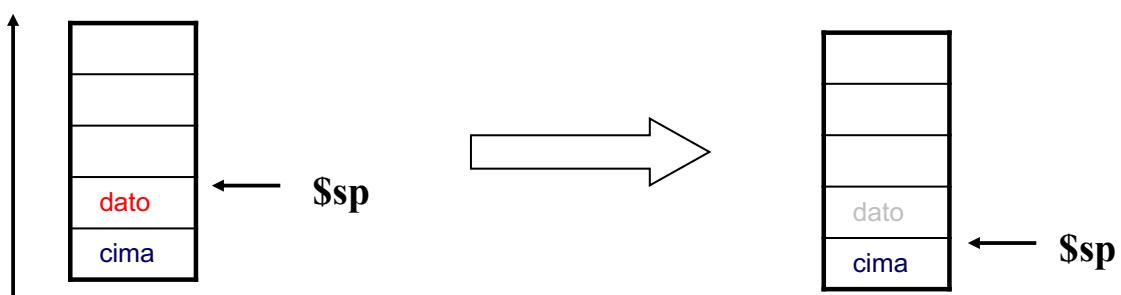


crece hacia direcciones bajas

# Pila

## POP Reg

Desapila el contenido del registro (dato)  
Copia dato en el registro Reg



crece hacia direcciones bajas

## Antes de empezar

- ▶ MIPS no dispone de instrucciones PUSH o POP.
- ▶ El registro puntero de pila (\$sp) es visible al programador.
  - ▶ Se va a asumir que el puntero de pila apunta al último elemento de la pila

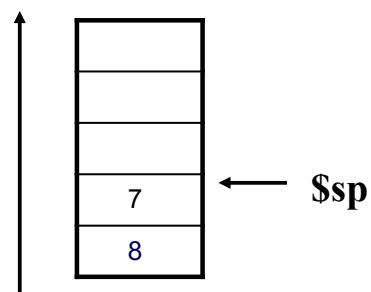
### PUSH \$t0

```
addu $sp, $sp, -4  
sw    $t0, ($sp)
```

### POP \$t0

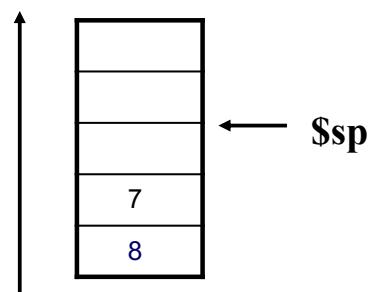
```
lw    $t0, ($sp)  
addu $sp, $sp, 4
```

## Operación PUSH en el MIPS



Estado inicial: el registro puntero de pila (\$sp) apunta al último elemento situado en la cima de la pila

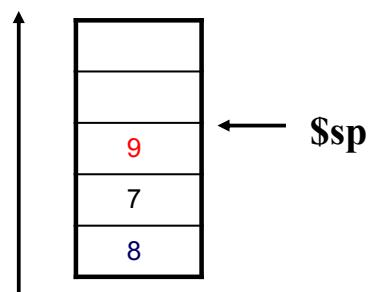
## Operación PUSH en el MIPS



**addu \$sp, \$sp, -4**

Se resta 4 al registro puntero de pila para poder insertar una nueva palabra en la pila

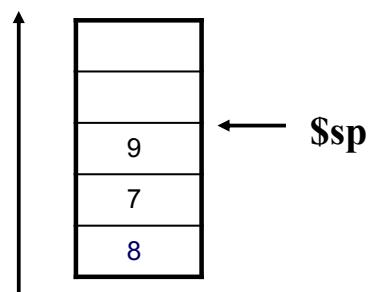
## Operación PUSH en el MIPS



```
li $t2, 9  
sw $t2 ($sp)
```

Se inserta el contenido del registro \$t2 en la cima de la pila

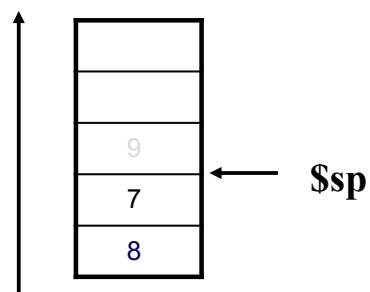
## Operación POP en el MIPS



**lw \$t2 (\$sp)**

Se copia en \$t2 el dato almacenado en la cima de la pila (9)

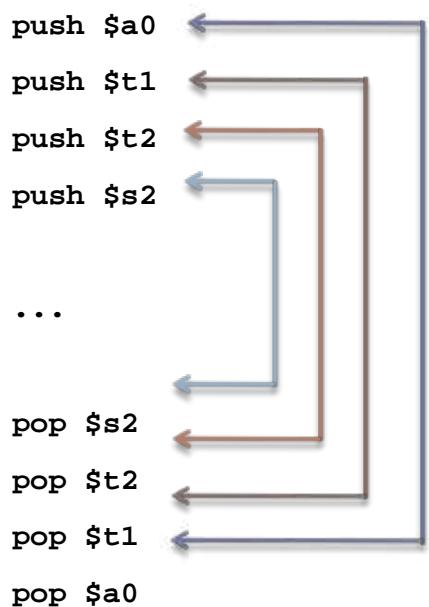
# Operación POP en el MIPS



**addu \$sp, \$sp, 4**

Se actualiza el registro puntero de pila para apuntar a la nueva cima de la pila. El dato desapilado (9) sigue estando en memoria pero será sobrescrito en futuras operaciones PUSH

# Pila uso de push y pop consecutivos



# Pila uso de push y pop consecutivos

```
push $a0
push $t1
push $t2
push $s2
```

...

```
pop $s2
pop $t2
pop $t1
pop $a0
```

```
addu $sp $sp -4
sw   $a0  ($sp)
addu $sp $sp -4
sw   $t1  ($sp)
addu $sp $sp -4
sw   $t2  ($sp)
addu $sp $sp -4
sw   $s2  ($sp)
```

...

```
lw $s2 ($sp)
addu $sp $sp 4
lw $t2 ($sp)
addu $sp $sp 4
lw $t1 ($sp)
addu $sp $sp 4
lw $a0 ($sp)
addu $sp $sp 4
```

# Pila uso de push y pop consecutivos

```
push $a0
push $t1
push $t2
push $s2
```

...

```
pop $s2
pop $t2
pop $t1
pop $a0
```

```
addu $sp $sp -16
sw   $a0  12($sp)
sw   $t1  8($sp)
sw   $t2  4($sp)
sw   $s2    ($sp)
```

...

```
lw   $s2    ($sp)
lw   $t2  4($sp)
lw   $t1  8($sp)
lw   $a0  12($sp)
addu $sp $sp 16
```

## Ejemplo

(1) Se parte de un código en lenguaje de alto nivel

```
int main() {  
    int z;  
    z=factorial(5);  
    print_int(z);  
    .  
    .  
    .  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```



## Ejemplo

### (2) Pensar en el paso de parámetros

- ▶ Los **parámetros** en el MIPS se pasan en \$a0, \$a1, \$a2 y \$a3
- ▶ Los **resultados** en el MIPS se recogen en \$v0, \$v1
- ▶ Más adelante se verá con más detalle
- ▶ Si se necesita pasar más de cuatro parámetros, los cuatro primeros en los registros \$a0, \$a1, \$a2 y \$a3 y el resto en la pila
  
- ▶ En la llamada `z=factorial(5);`
- ▶ Un parámetro de entrada: en \$a0
- ▶ Un resultado en \$v0

## Ejemplo

### (3) Se pasa a ensamblador cada función

El parámetro se pasa en \$a0  
El resultado se devuelve en \$v0

```
int main() {  
    int z;  
    z=factorial(5);  
    print_int(z);  
    ...  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}  
  
li $a0, 5      # argumento  
jal factorial # llamada  
move $a0, $v0 # resultado  
li $v0, 1  
syscall       # llamada para  
               # imprimir un int  
...  
  
factorial: li  $s1, 1  #s1 para r  
            li  $s0, 1  #s0 para i  
            bucle: bgt $s0, $a0, fin  
                    mul $s1, $s1, $s0  
                    addi $s0, $s0, 1  
                    b    bucle  
            fin:  move $v0, $s1 #resultado  
                  jr   $ra
```

# Ejemplo

## (4) Se analizan los registros que se modifican

```
int factorial(int x) {          factorial: li    $s1, 1    #s1 para r
    int i;                      li    $s0, 1    #s0 para i
    int r=1;                     bucle: bgt   $s0, $a0, fin
    for (i=1;i<=x;i++) {        mul    $s1, $s1, $s0
        r*=i;                   addi   $s0, $s0, 1
    }                           b      bucle
    return r;                   fin:   move   $v0, $s1  #resultado
}                                jr    $ra
```



La función factorial trabaja (modifica) con los registros \$s0, \$s1

Si estos registros se modifican dentro de la función, podría afectar a la función que realizó la llamada (la función main)

Por tanto, la función factorial debe guardar el valor de estos registros en la pila al principio y restaurarlos al final

# Ejemplo

## (5) Se guardan los registros en la pila

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```



```
factorial: addu $sp, $sp, -8  
           sw   $s0, 4($sp)  
           sw   $s1, ($sp)  
           li   $s1, 1    #s1 para r  
           li   $s0, 1    #s0 para i  
bucle:    bgt  $s0, $a0, fin  
           mul   $s1, $s1, $s0  
           addi  $s0, $s0, 1  
           b     bucle  
fin:      move  $v0, $s1  #resultado  
           lw    $s1, ($sp)  
           lw    $s0, 4($sp)  
           addu $sp, $sp, 8  
           jr   $ra
```

No es necesario guardar \$ra. La rutina factorial es terminal

Se guarda en la pila \$s0 y \$s1 porque se modifican

Si se hubiera utilizado \$t0 y \$t1 no habría hecho falta hacerlo (los registros \$t no se preservan)

## Ejemplo 2

```
int main()
{
    int z;

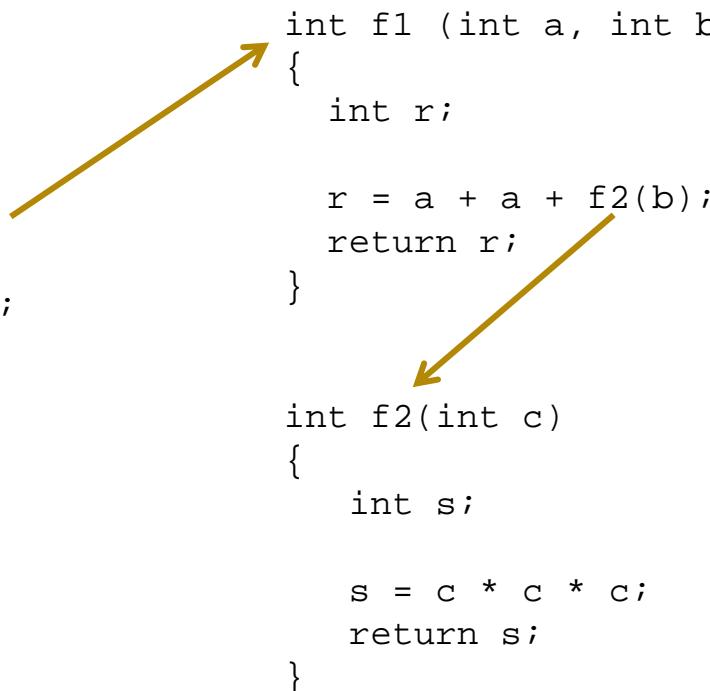
    z=f1(5, 2);
    print_int(z);
}

int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}

int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```



## Ejemplo 2. Invocación

```
int main()
{
    int z;
    z=f1(5, 2);
    print_int(z);
}
```



```
li    $a0, 5          # primer argumento
li    $a1, 2          # segundo argumento
jal   f1              # llamada
move  $a0, $v0        # resultado
li    $v0, 1          # llamada para
syscall                 # imprimir un int
```

Los parámetros se pasan en \$a0 y \$a1  
El resultado se devuelve en \$v0

## Ejemplo 2. Cuerpo de f1

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```



```
f1: add    $s0, $a0, $a0
     move   $a0, $a1
     jal    f2
     add    $v0, $s0, $v0
     jr    $ra
```

```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

## Ejemplo 2. Se analizan los registros que se modifican en f1

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```

```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

```
f1: add    $s0, $a0, $a0
     move   $a0, $a1
     jal    f2
     add    $v0, $s0, $v0
     jr    $ra
```

f1 modifica \$s0 y \$ra, por lo tanto se guardan en la pila  
El registro \$ra se modifica en la instrucción jal f2  
El registro \$a0 se modifica al pasar el argumento a f2, pero  
por convenio la función f1 no tiene porque guardarla en la pila  
solo si lo utiliza después de realizar la llamada a f2

## Ejemplo 2. Cuerpo de f1 guardando en la pila los registros que se modifican

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```

```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

```
f1: addu $sp, $sp, -8
    sw   $s0, 4($sp)
    sw   $ra,  ($sp)
```

```
add   $s0, $a0, $a0
move  $a0, $a1
jal   f2
add   $v0, $s0, $v0
```

```
lw   $ra,  ($sp)
lw   $s0, 4($sp)
addu $sp, $sp, 8
```

```
jr   $ra
```

## Ejemplo 2. Cuerpo de f2

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```

```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

```
f2: mul $t0, $a0, $a0
    mul $v0, $t0, $a0
    jr $ra
```

La función f2 no modifica el registro \$ra porque no llama a ninguna otra función  
El registro \$t0 no es necesario guardar su valor porque no se ha de preservar su valor según convenio

## Convención en el uso de los registros en el MIPS

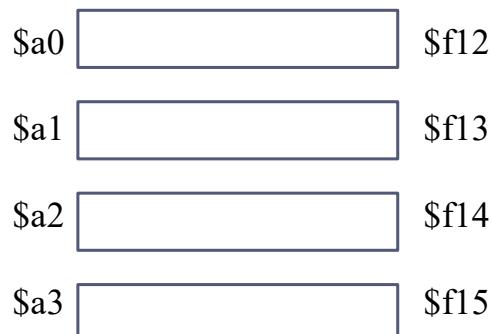
Registro	Uso	Preservar el valor
\$v0-\$v1	Resultados	No
\$a0..\$a3	Argumentos	No
\$t0..\$t9	Temporales	No
\$s0..\$s7	Temporales a Salvar	Si
\$sp	Puntero de pila	Si
\$fp	Puntero marco de pila	Si
\$ra	Dirección de retorno	Si

## Convención en el uso de los registros de coma flotante en el MIPS

Registro	Uso	Preservar el valor
\$f0-\$f3	Resultados	No
\$f4..\$f11	Temporales	No
\$f12-\$f15	Argumentos	No
\$f16-\$f19	Temporales	No
\$f20-\$f31	Temporales a salvar	Si

## Paso de argumentos detallado en MIPS

- ▶ Se utilizan los registros \$ai y \$fi, pero no se pueden pasar más de 16 bytes en registros teniendo en cuenta \$a y \$f



- ▶ Si el primer argumento es entero:
  - ▶ Todos los argumentos se pasan en \$a0 .. \$a3 (aunque el resto sean de tipo float o double)
  - ▶ El resto en la pila

## Paso de argumentos detallado en MIPS

- ▶ Si el primer argumento es float o double
  - ▶ Se pasa en \$f12 (float) y \$f14-\$f15 (double)
  - ▶ Para el resto se utiliza \$ai o \$fi hasta llegar a 16 bytes, el resto en la pila



## Ejemplos de llamadas

Argumentos de la función	Asignación en registros y pila
d1, d2	\$f12-\$f13, \$f14-\$f15
s1, d2	\$f12, \$f14-\$f15
s1, s2	\$f12, \$f13
n1, n2, n3, n4	\$a0, \$a1, \$a2, \$a3
d1, n1, d2	\$f12-\$f13, \$a2, pila
d1, n1, n2	\$f12-\$f13, \$a2, \$a3
n1, n2, n3, d1	\$a0, \$a1, \$a2, pila
n1, n2, n3, s1	\$a0, \$a1, \$a2, \$a3
n1, n2, d1	\$a0, \$a1, (\$a2,\$a3)
d1, s1, s2	\$f12-\$f13, \$f14, \$a3

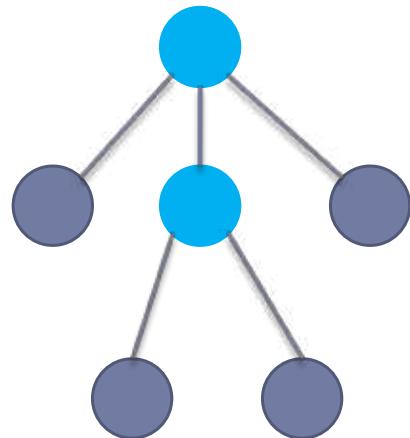
n=int, d=double, s=float

## Retorno de resultados en MIPS

- ▶ Se usa \$v0 y \$v1 para valores de tipo entero
- ▶ Se usa \$f0 para valores de tipo float
- ▶ Se usa \$f0-\$f1 para valores de tipo double
- ▶ En caso de estructuras o valores complejos han de dejarse en pila. El espacio lo reserva la función que realiza la llamada

## Tipos de subrutinas

- ▶ **Subrutina terminal.** 
  - ▶ No invoca a ninguna otra subrutina.
- ▶ **Subrutina no terminal.** 
  - ▶ Invoca a alguna otra subrutina.



## Activación de procedimientos Marco de pila

- ▶ El **marco de pila o registro de activación** es el mecanismo que utiliza el compilador para activar los procedimientos (subrutinas) en los lenguajes de alto nivel
- ▶ El marco de pila lo construyen en la pila el procedimiento llamante y el llamado
- ▶ Su manipulación se hace a través de dos registros:
  - ▶ \$sp: puntero de pila, que apunta siempre a la cima de la pila
  - ▶ \$fp: puntero de marco de pila, que marca la zona de la pila que pertenece al procedimiento llamado
- ▶ El **registro marco de pila** (\$fp) se utiliza en el procedimiento llamado para:
  - ▶ Acceder a los parámetros que se pasan en la pila
  - ▶ Acceder a las variables locales del procedimiento

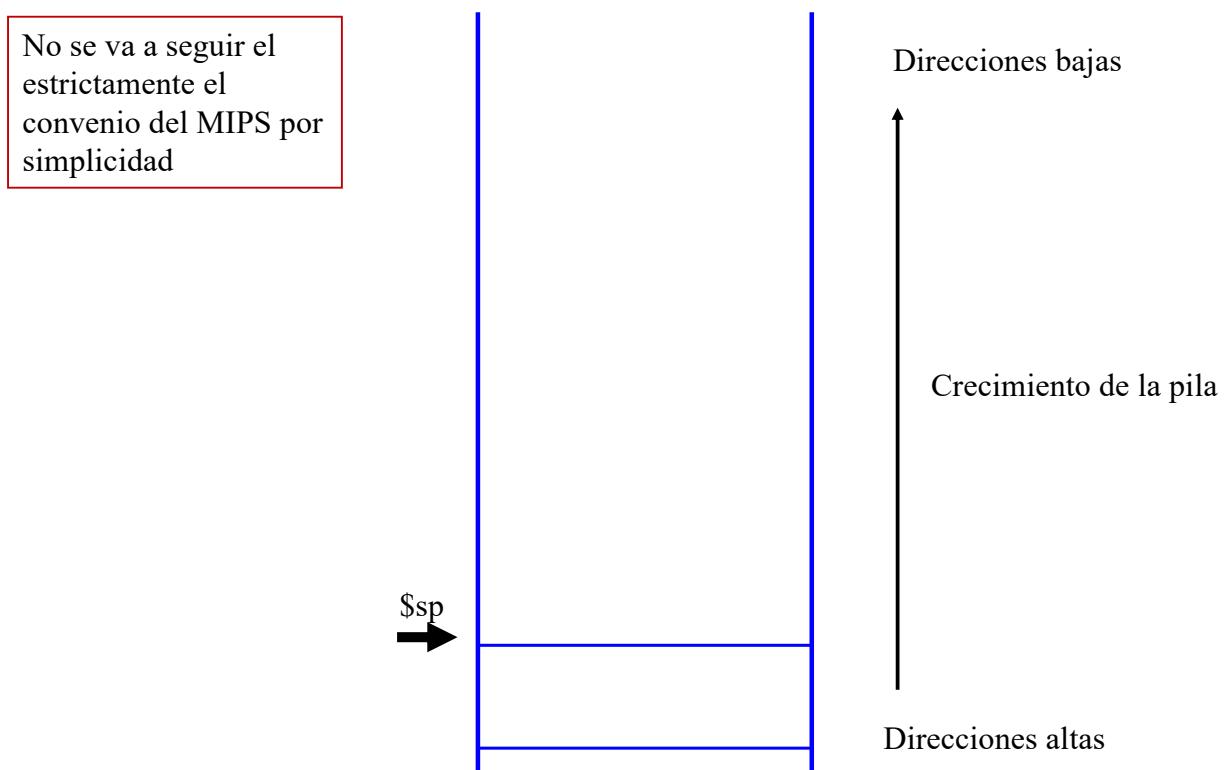
## Marco de pila

- ▶ El marco de pila almacena:
  - ▶ Los parámetros introducidos por el procedimiento llamante en caso de ser necesarios
  - ▶ El registro marco de pila del procedimiento llamante (antiguo \$fp)
  - ▶ Los registros guardados por la función (incluyen al registro \$ra en caso de procedimientos no terminales)
  - ▶ Variables locales

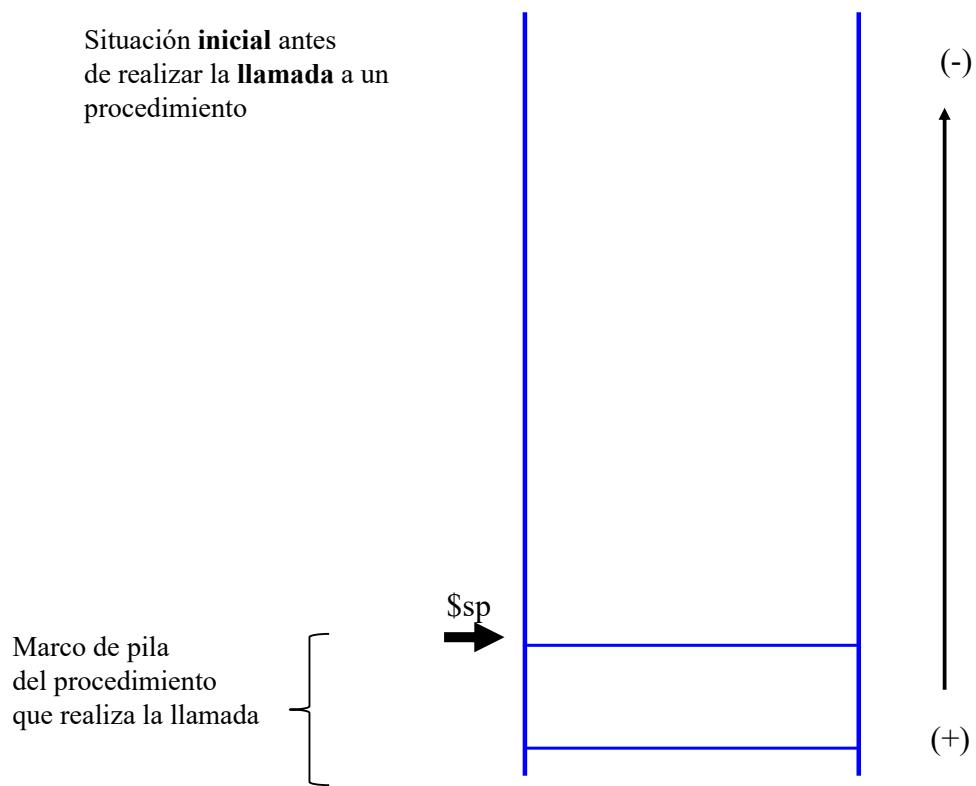
# Procedimiento general de llamadas a subrutinas

Subrutina llamante	Subrutina llamada
Salvaguarda de registros que no quiera que modifique la subrutina llamada (\$t, \$a, ..)	
Paso de parámetros, reserva de espacio para valores a devolver si es necesario	
<b>Llamada a subrutina (jal)</b>	
	Reserva del marco de pila
	Salvaguarda de registros (\$ra, \$fp, \$s)
	<b>Ejecución de subrutina</b>
	Restauración de valores guardados
	Copiar valores a devolver en el espacio reservado por el llamante
	Liberación de marco de pila
	<b>Salida de subrutina (jr \$ra)</b>
Recuperar valores devueltos	
Restauración de registros guardados, liberación del espacio de pila reservado	

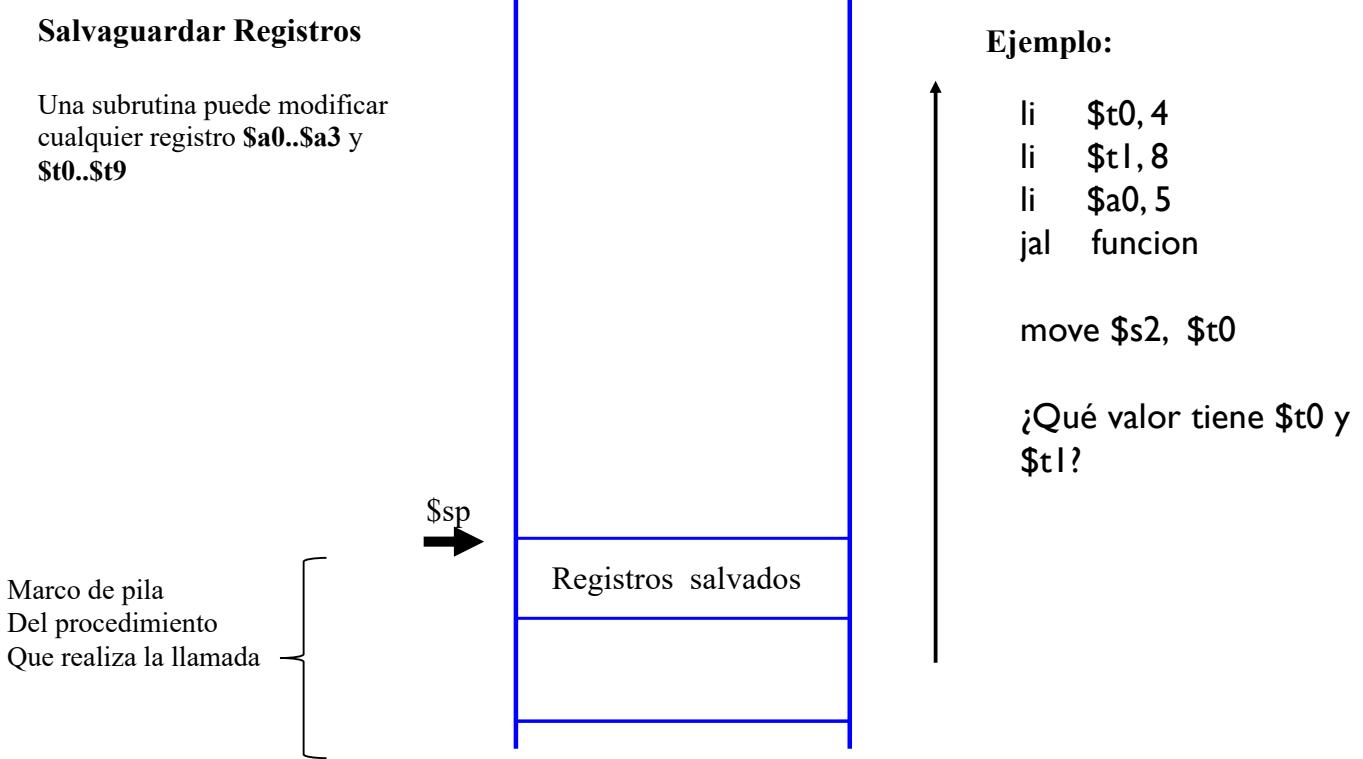
## Construcción del marco de pila subrutina llamante



## Construcción del marco de pila subrutina llamante



## Construcción del marco de pila subrutina llamante



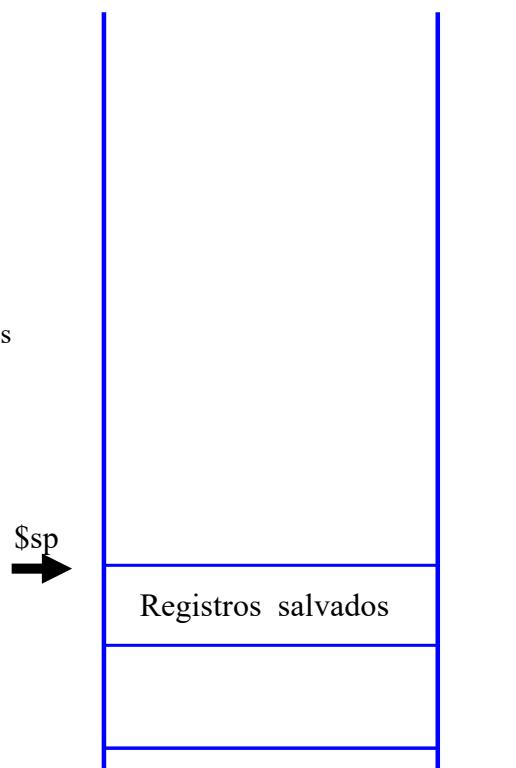
# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro **\$a0..\$a3** y **\$t0..\$t9**

Para preservar su valor, la subrutina que realiza la llamada debe guardar en la pila los valores de esos registros

Marco de pila  
Del procedimiento  
Que realiza la llamada



## Ejemplo:

```
li $t0, 4
li $t1, 8
li $a0, 5
jal funcion
```

move \$s2, \$t0

¿Qué valor tiene \$t0 y \$t1?

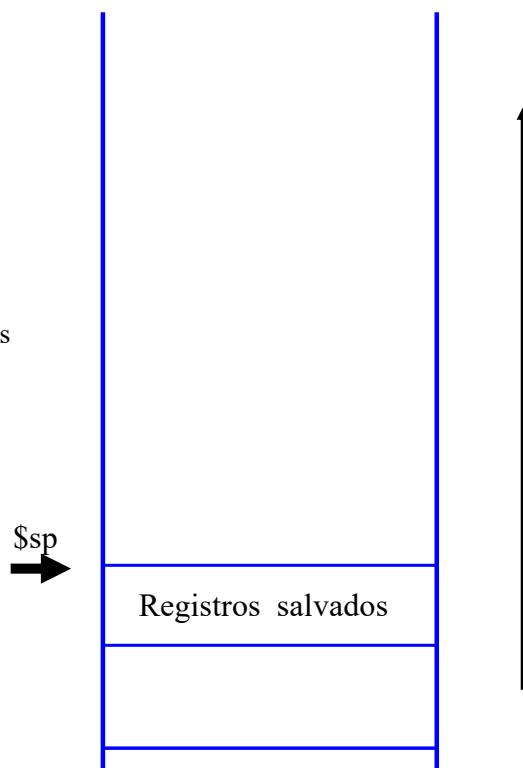
# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro **\$a0..\$a3** y **\$t0..\$t9**

Para preservar su valor, la subrutina que realiza la llamada debe guardar en la pila los valores de esos registros

Marco de pila  
Del procedimiento  
Que realiza la llamada



## Ejemplo:

```
subu $sp $sp 8
sw  $t0 ($sp)
sw  $t1 4($sp)
```

```
li   $a0, 5
jal  funcion
```

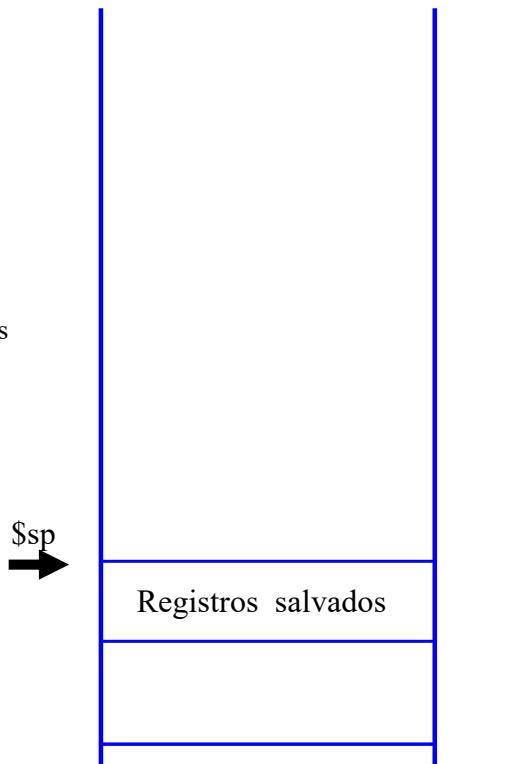
# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro \$a0..\$a3 y \$t0..\$t9

Para preservar su valor, la subrutina que realiza la llamada debe guardar en la pila los valores de esos registros  
(habrá que restaurarlos después)

Marco de pila  
Del procedimiento  
Que realiza la llamada



## Ejemplo:

```
subu $sp $sp 8
sw $t0 ($sp)
sw $t1 4($sp)
```

```
li $a0, 5
jal funcion
```

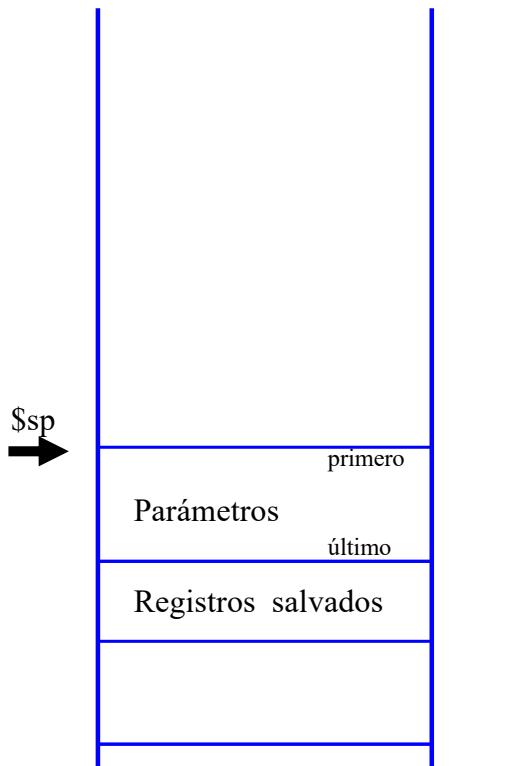
```
lw $t0 ($sp)
lw $t1 4($sp)
addu $sp $sp 8
```

# Construcción del marco de pila subrutina llamante

## Paso de parámetros:

Antes de realizar la llamada el procedimiento llamante:  
Deja los parámetros en **\$ai (\$f)**  
El resto de parámetros en la pila

Marco de pila  
Del procedimiento  
Que realiza la llamada



## Ejemplo (6 parámetros):

```
li $a0, 1
li $a1, 2
li $a3, 3
li $a4, 4

subu $sp, $sp, 8

li $t0, 5
sw $t0, 4($sp)

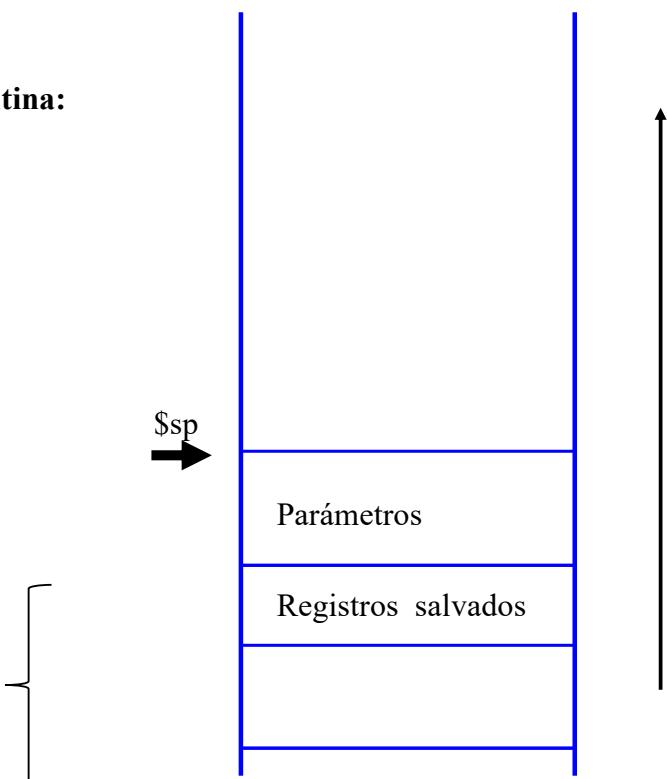
li $t0, 6
sw $t0, ($sp)
```

# Construcción del marco de pila subrutina llamante

**Llamada a subrutina:**

jal subrutina

Marco de pila  
Del procedimiento  
Que realiza la llamada



# Construcción del marco de pila subrutina llamada

**Reserva del marco de pila:**

$$\$sp = \$sp - \text{tamaño marco}$$

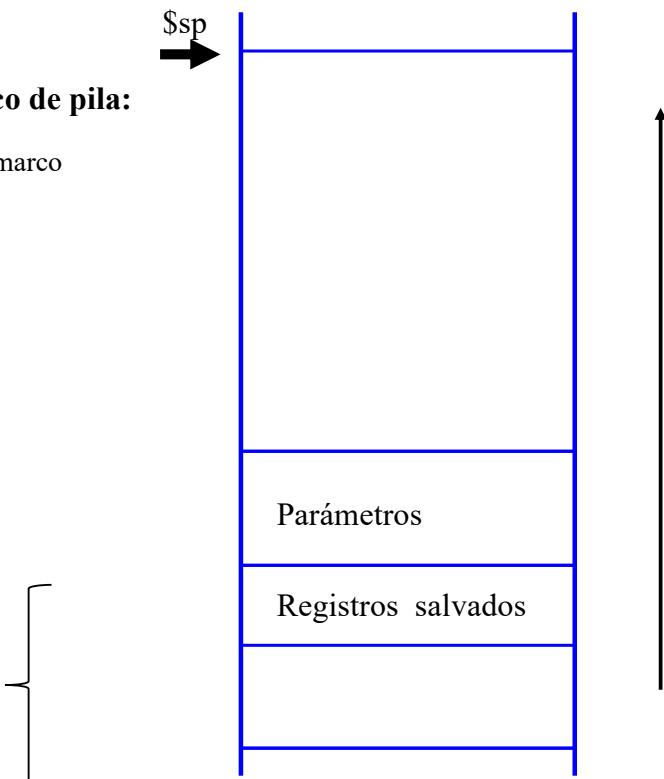
**Espacio para:**

$\$ra, \$fp$

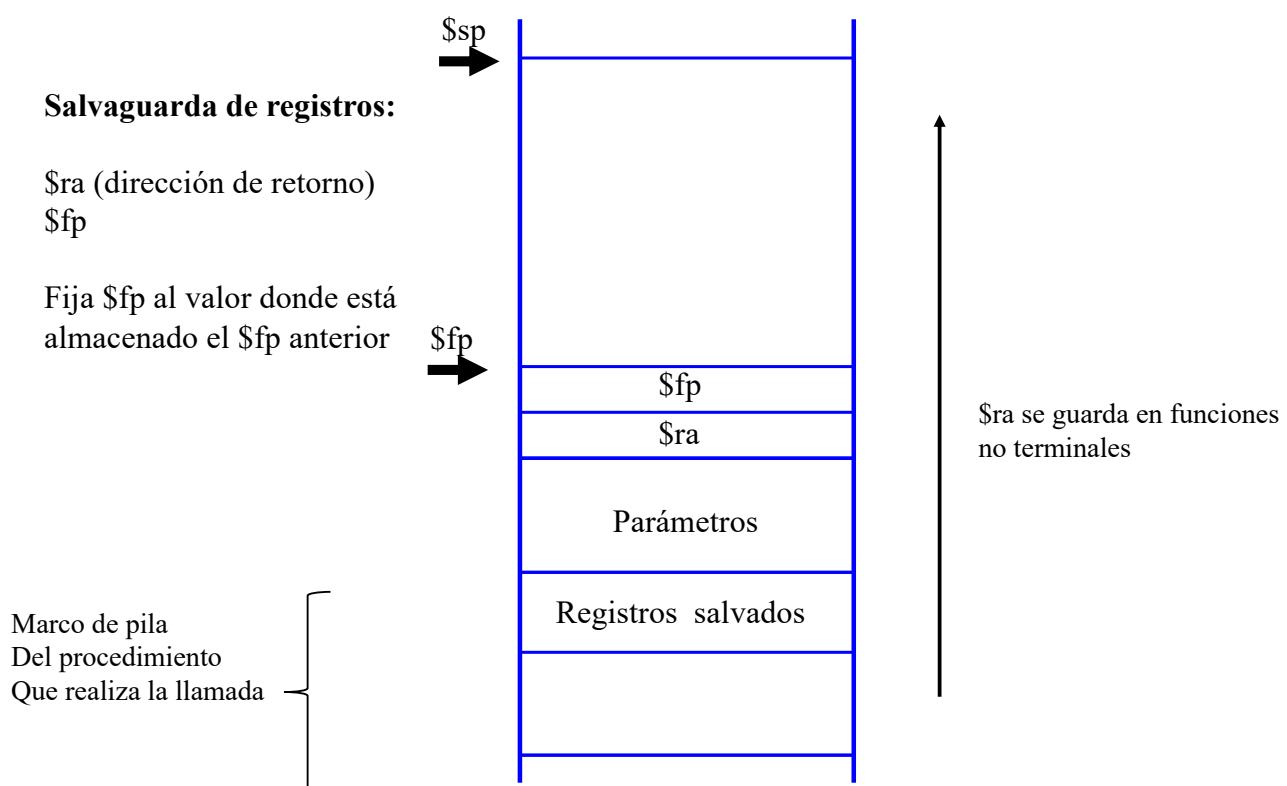
$\$s0\dots\$s9$

variables locales

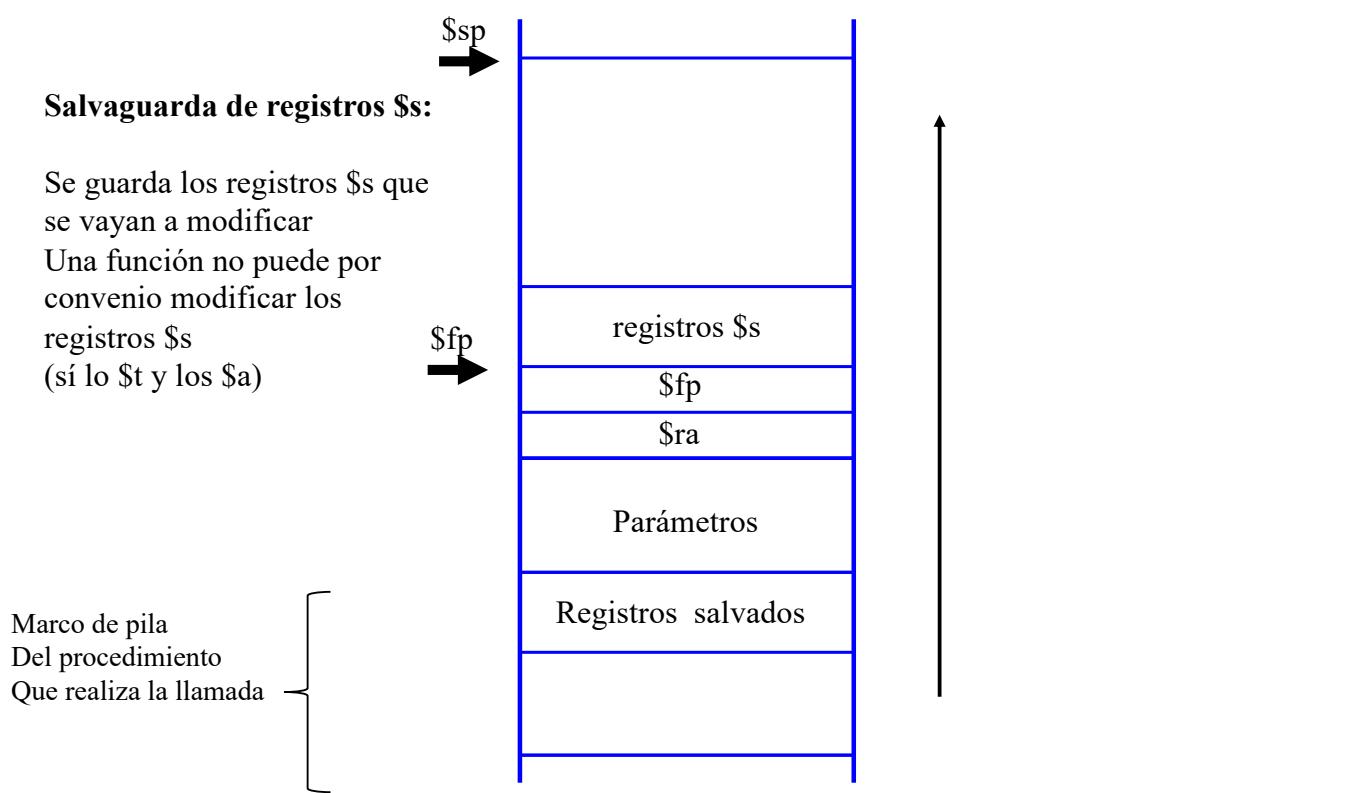
Marco de pila  
Del procedimiento  
Que realiza la llamada



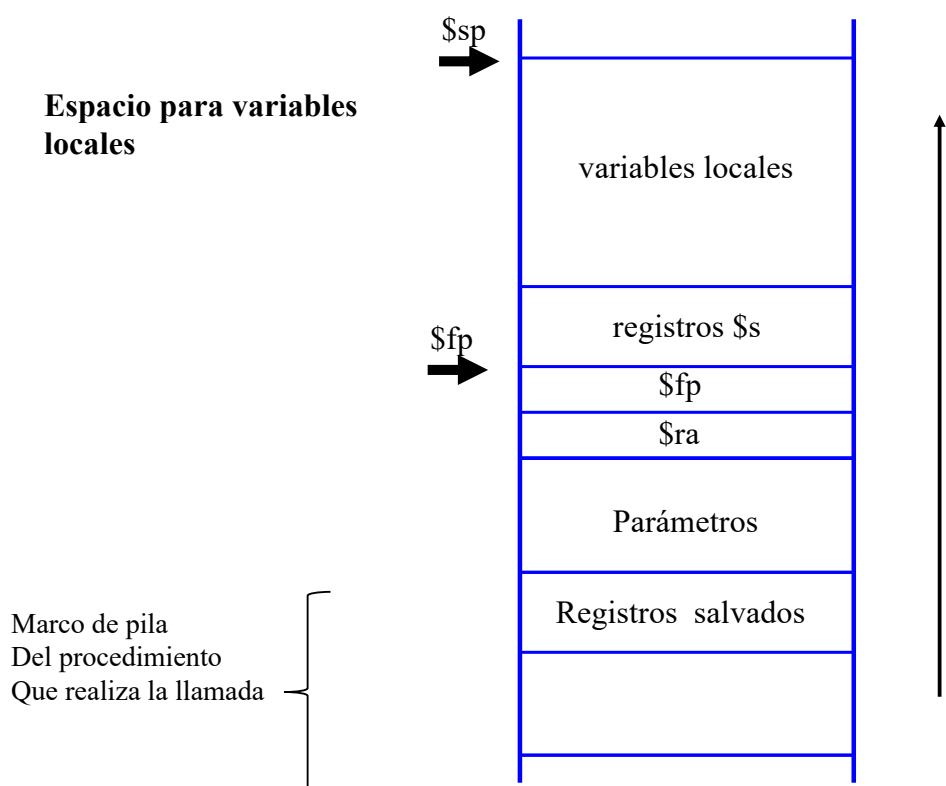
## Construcción del marco de pila subrutina llamada



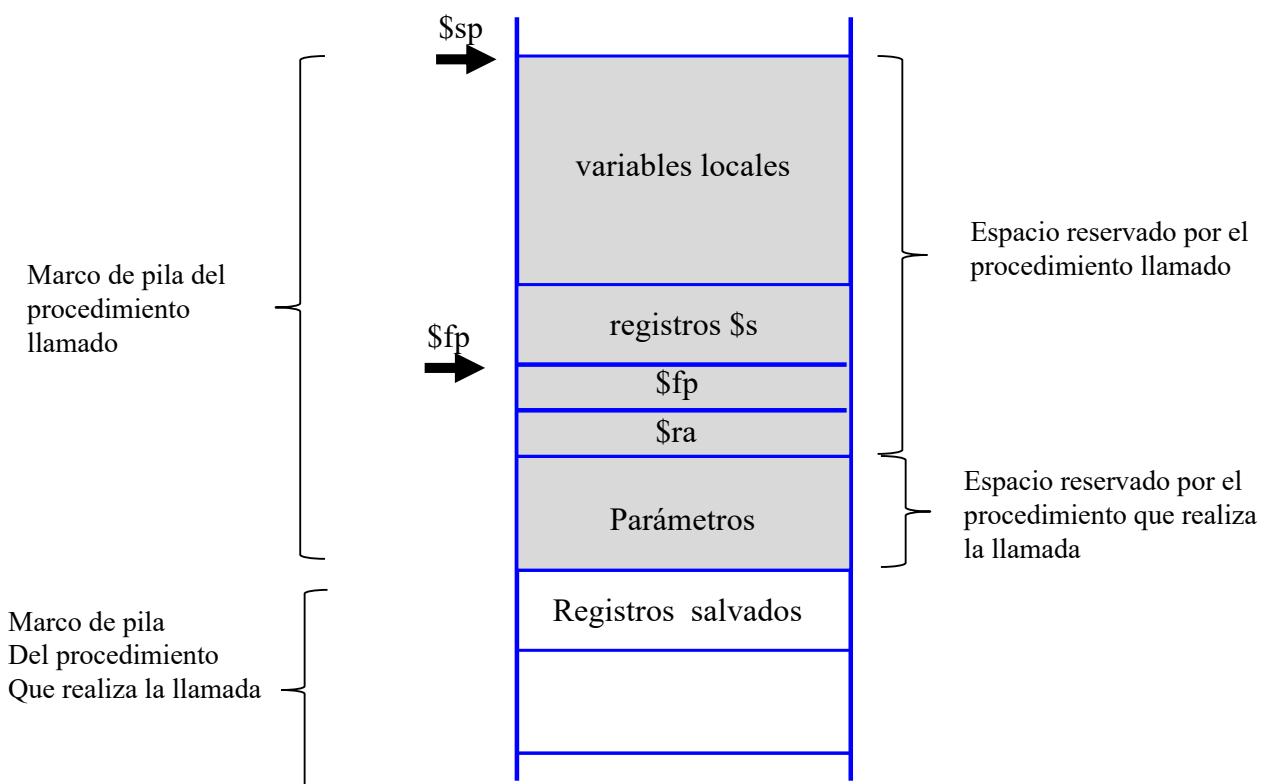
## Construcción del marco de pila subrutina llamada



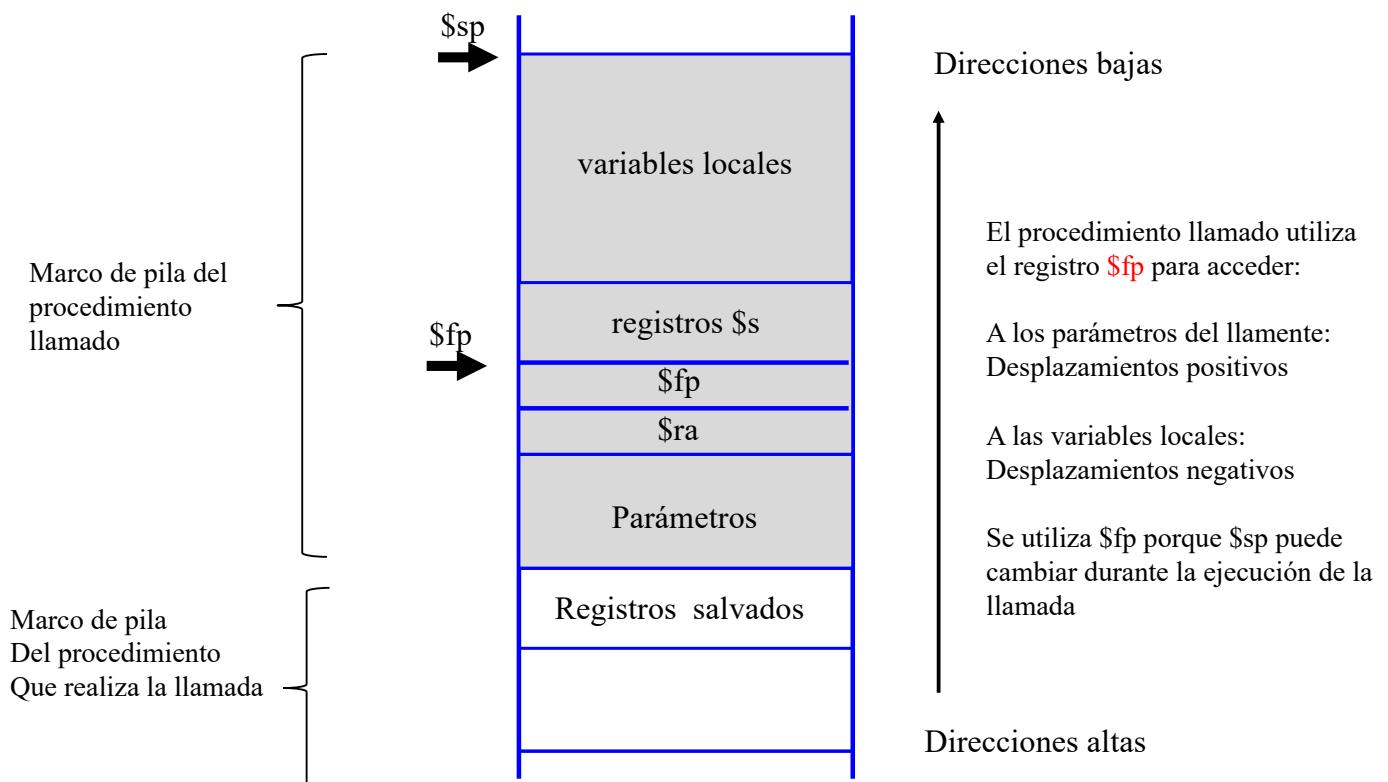
## Construcción del marco de pila subrutina llamada



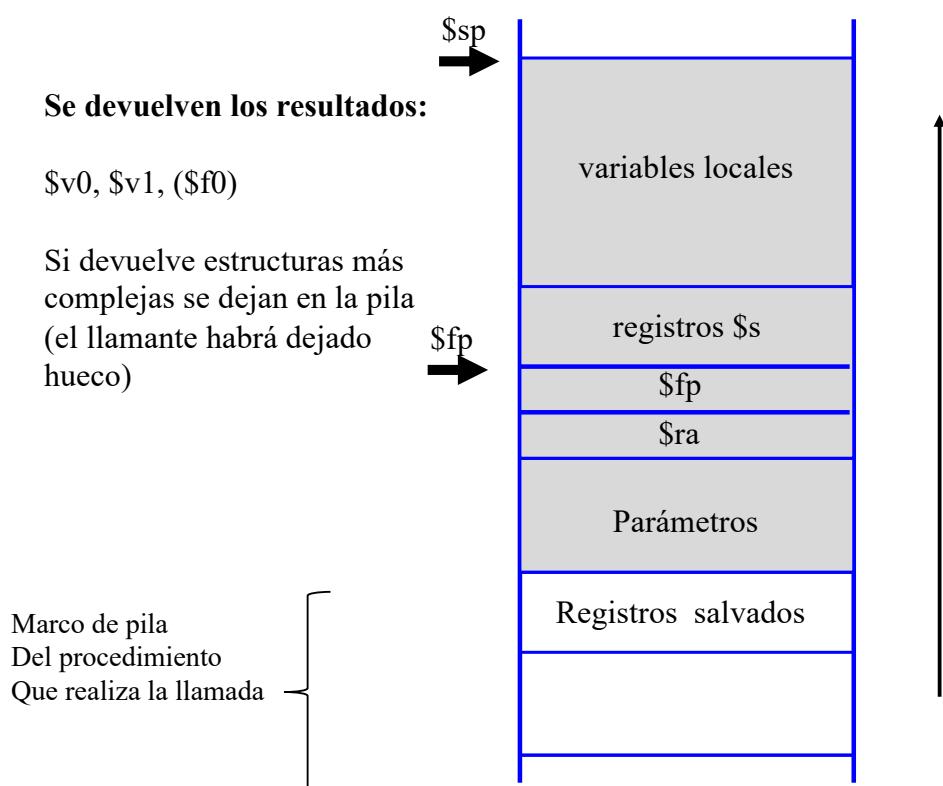
# Construcción del marco de pila



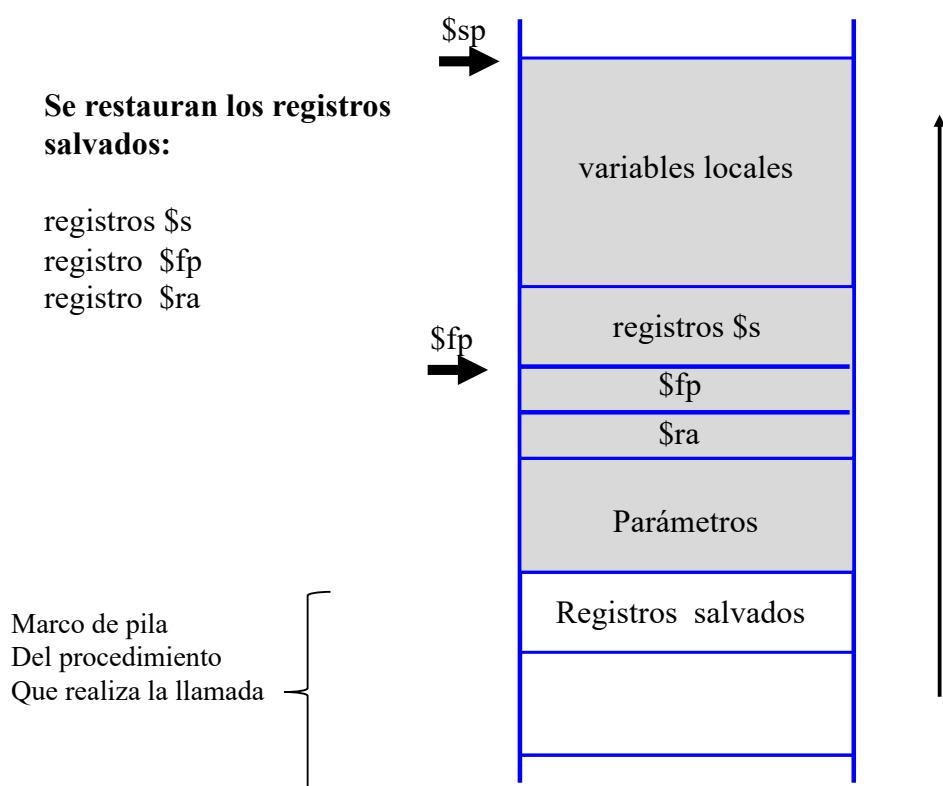
# Marco de pila



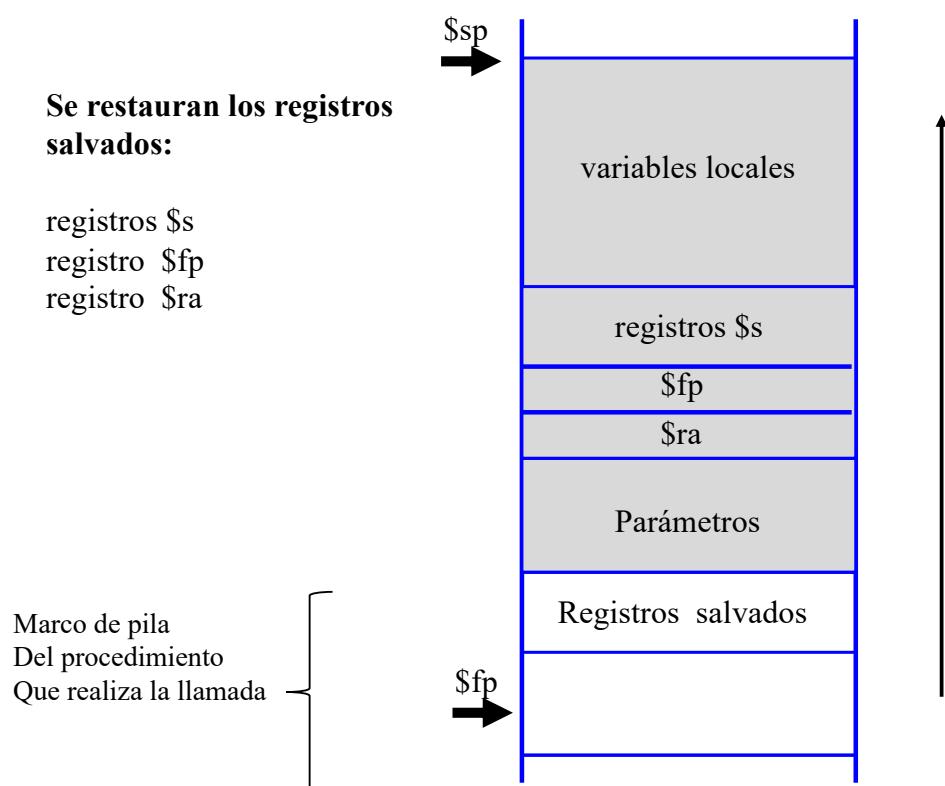
## Finalización de la subrutina subrutina llamada



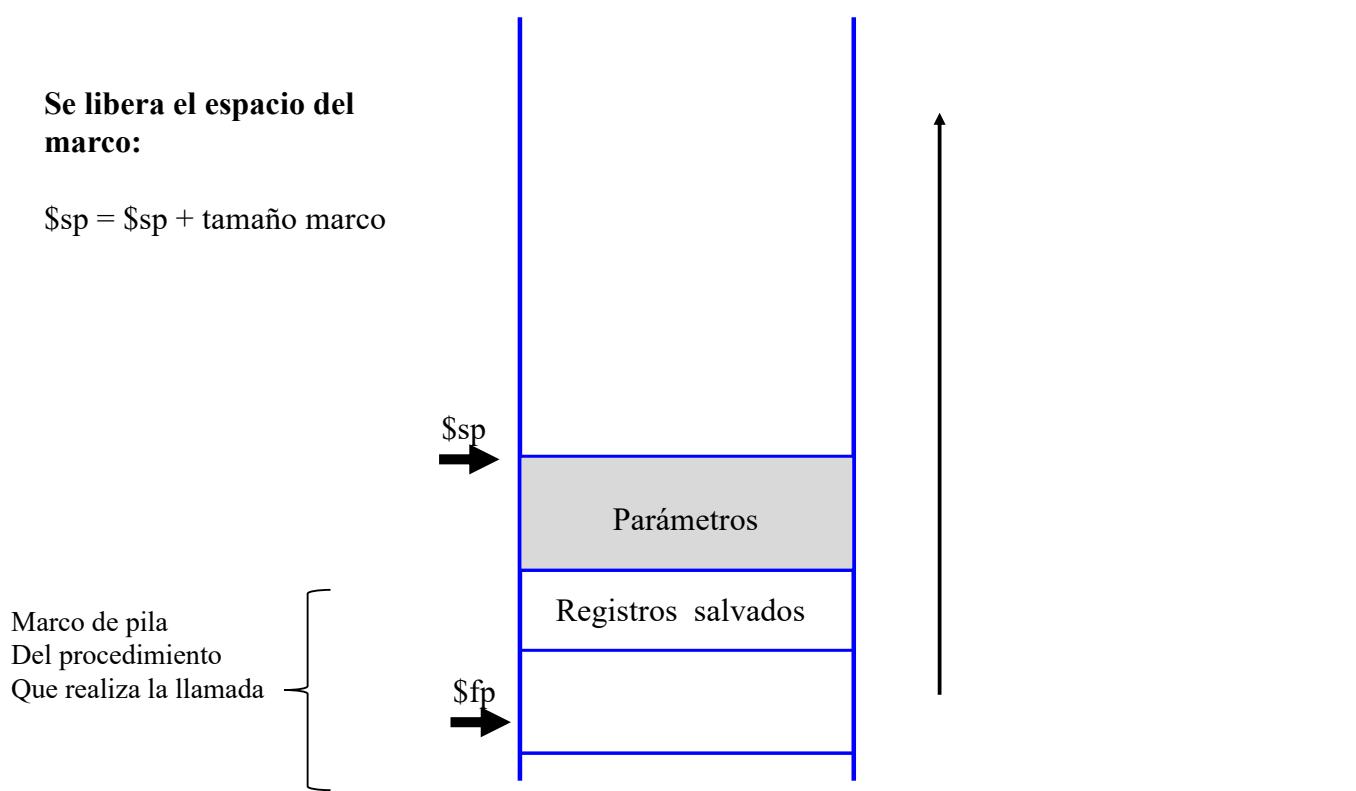
## Finalización de la subrutina subrutina llamada



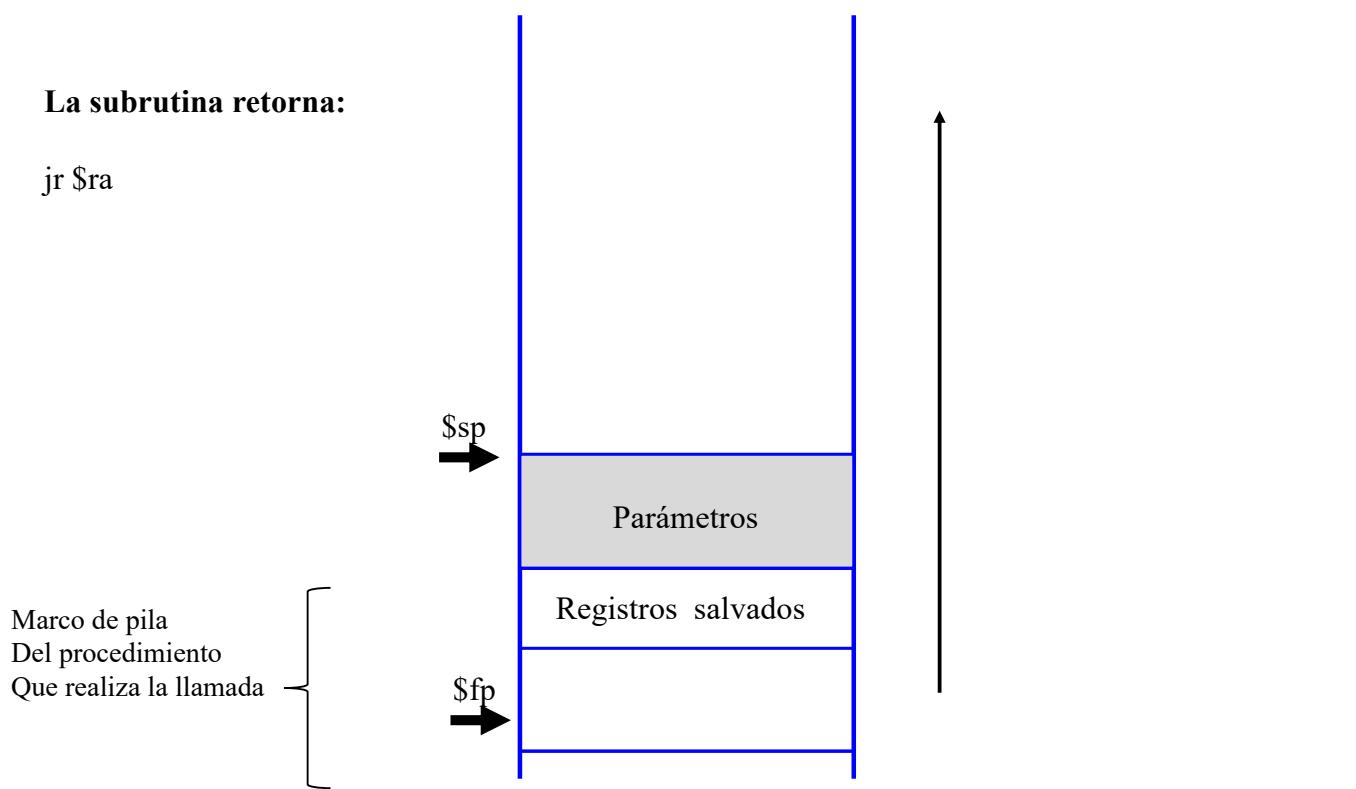
## Finalización de la subrutina subrutina llamada



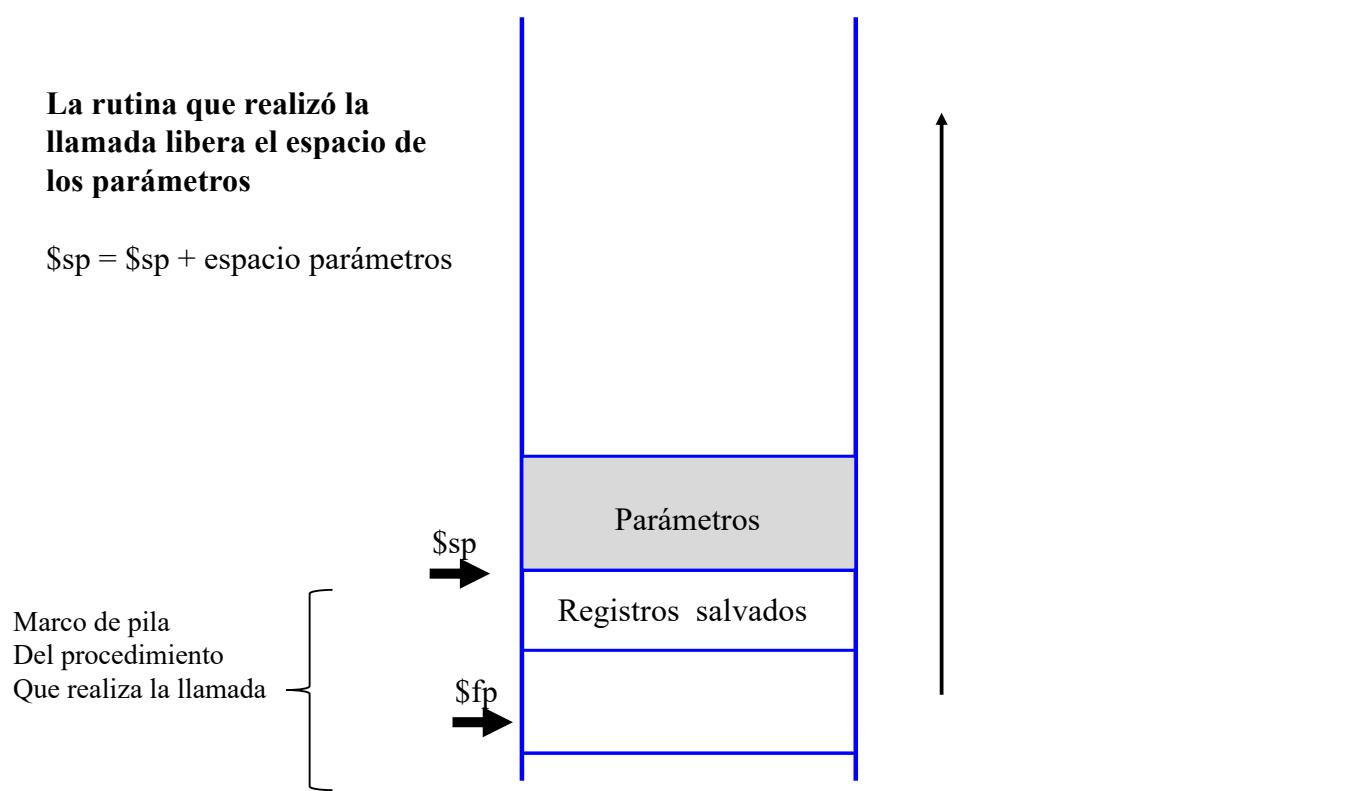
## Finalización de la subrutina subrutina llamada



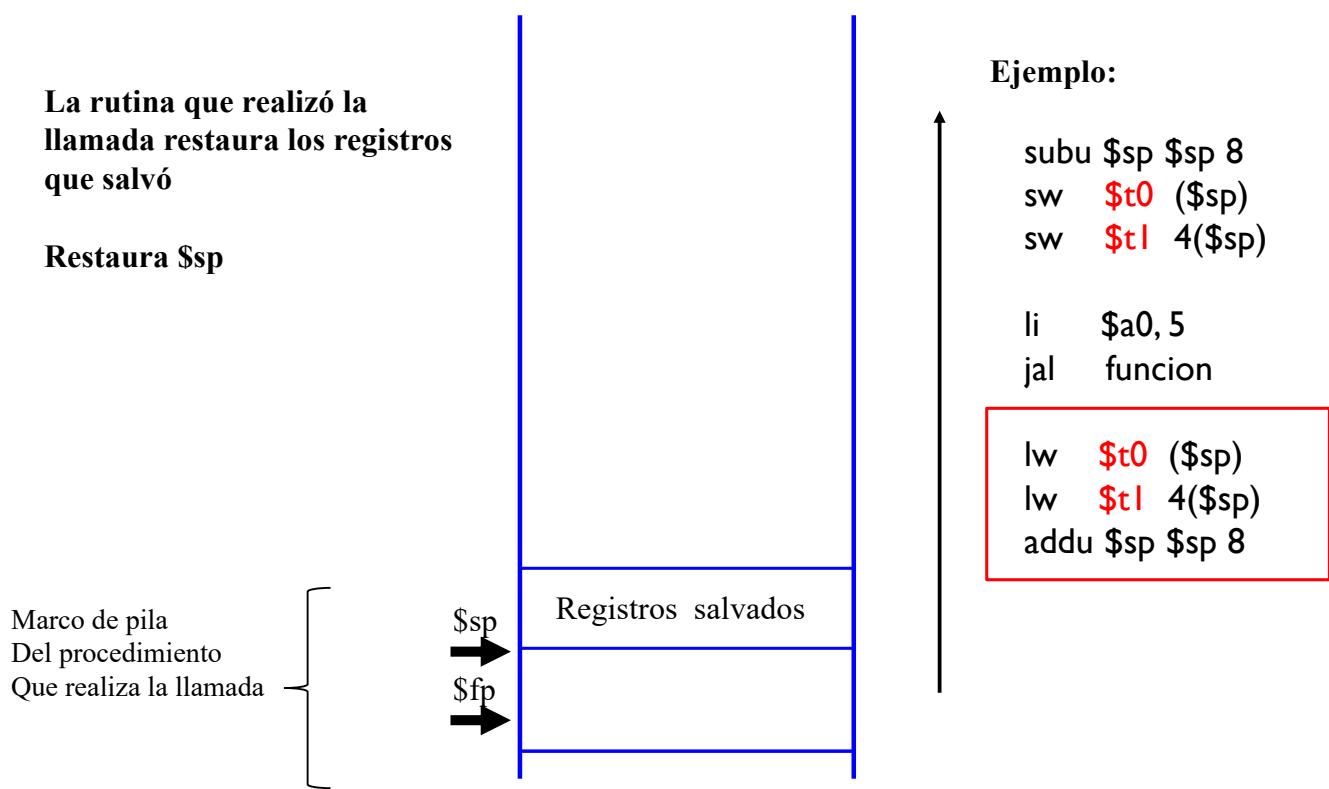
## Finalización de la subrutina subrutina llamada



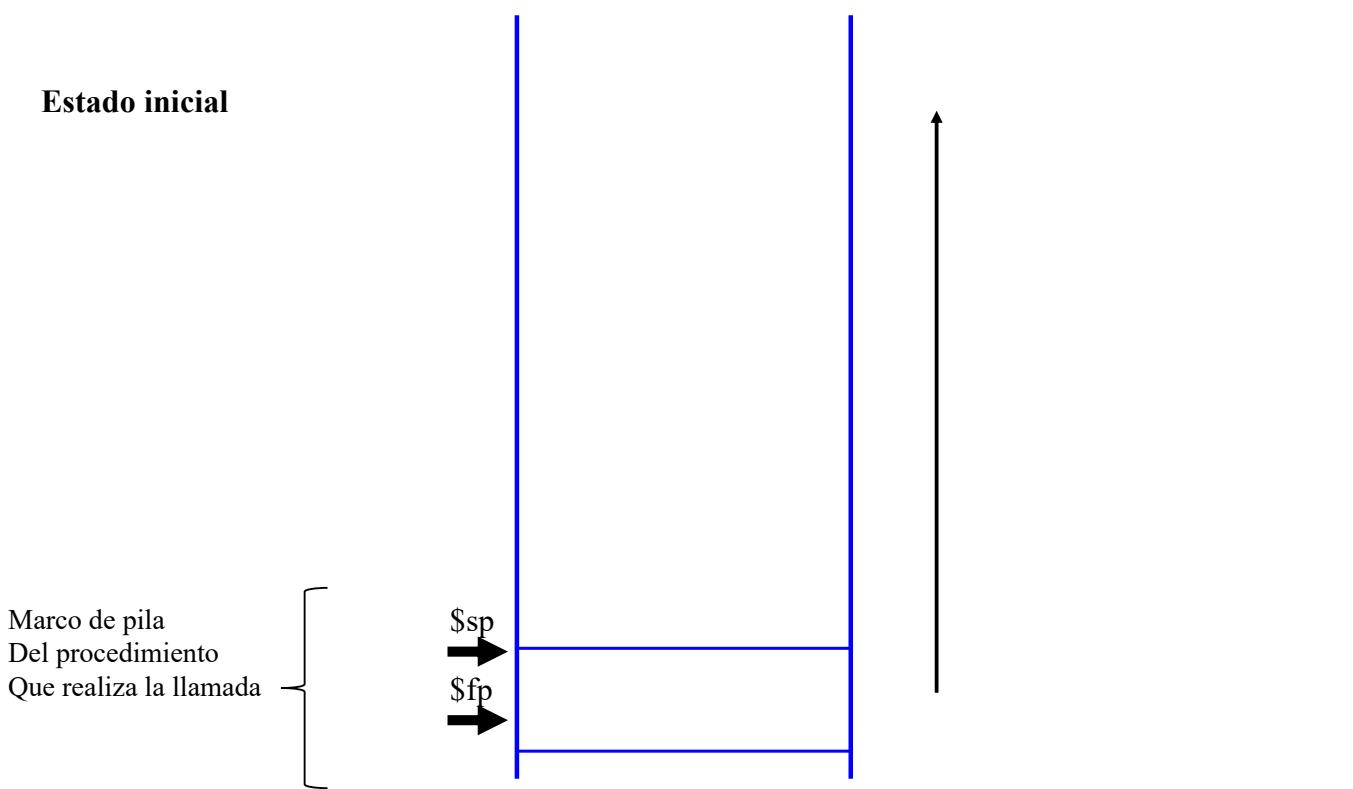
## Finalización de la subrutina subrutina llamante



## Finalización de la subrutina subrutina llamante



## Estado después de finalizar la llamada



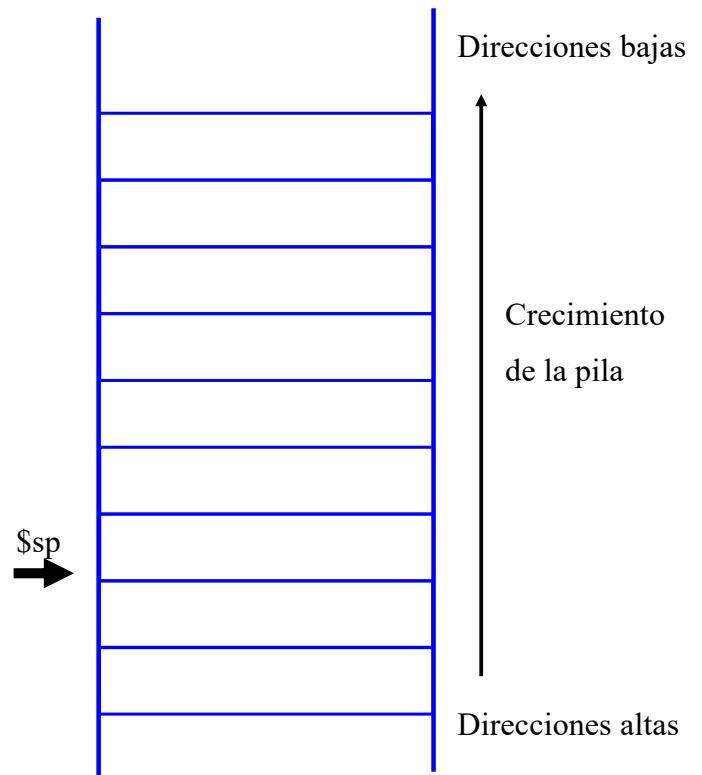
## Acceso a parámetros y variables locales usando el marco de pila

```
int f ( int n1, n2, n3,
        n4, n5, n6 )
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

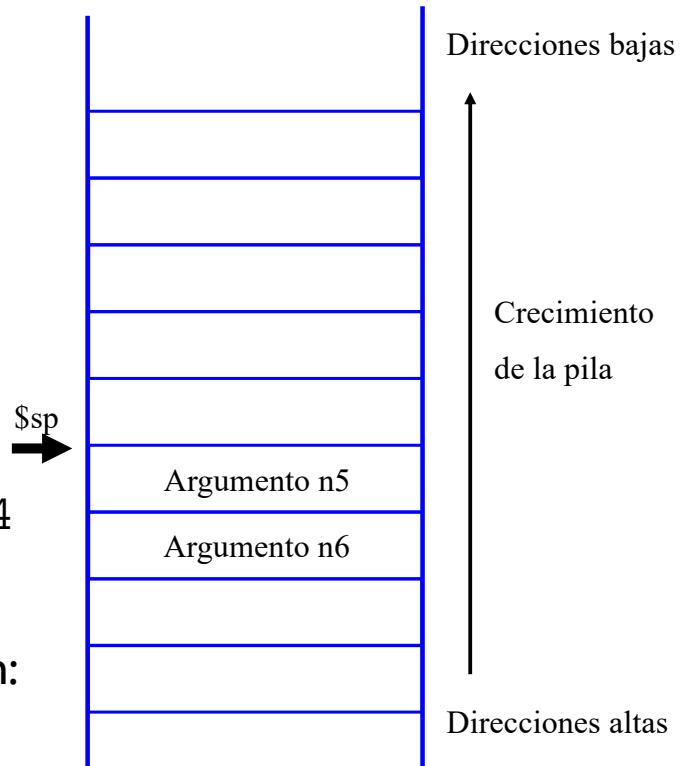
Si se realiza una llamada a f( )



## Acceso a parámetros y variables locales usando el marco de pila

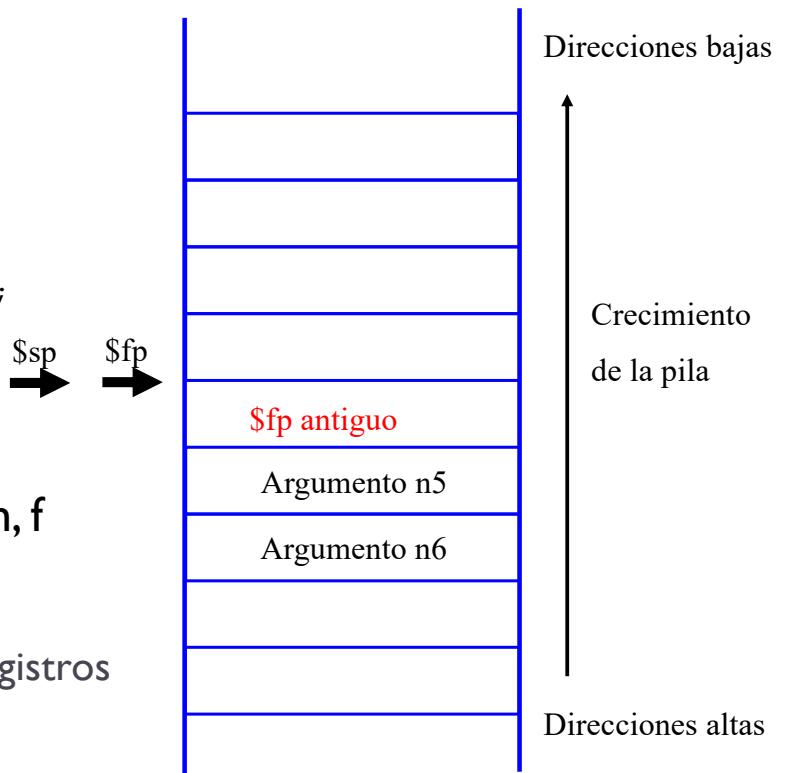
```
int f (int n1, n2, n3,  
       n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

- ▶ Los parámetros n1, n2, n3 y n4 se pasan:
  - ▶ En \$a0, \$a1, \$a2, \$a3
- ▶ Los parámetros n5, n6 se pasan:
  - ▶ En la pila



## Acceso a parámetros y variables locales usando el marco de pila

```
int f (int n1, n2, n3,  
       n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```



- ▶ Una vez invocada la función, f debe
  - ▶ Guardar una copia de \$fp
  - ▶ Guardar una copia de los registros a preservar
  - ▶ \$ra no, porque es terminal

## Acceso a parámetros y variables locales usando el marco de pila

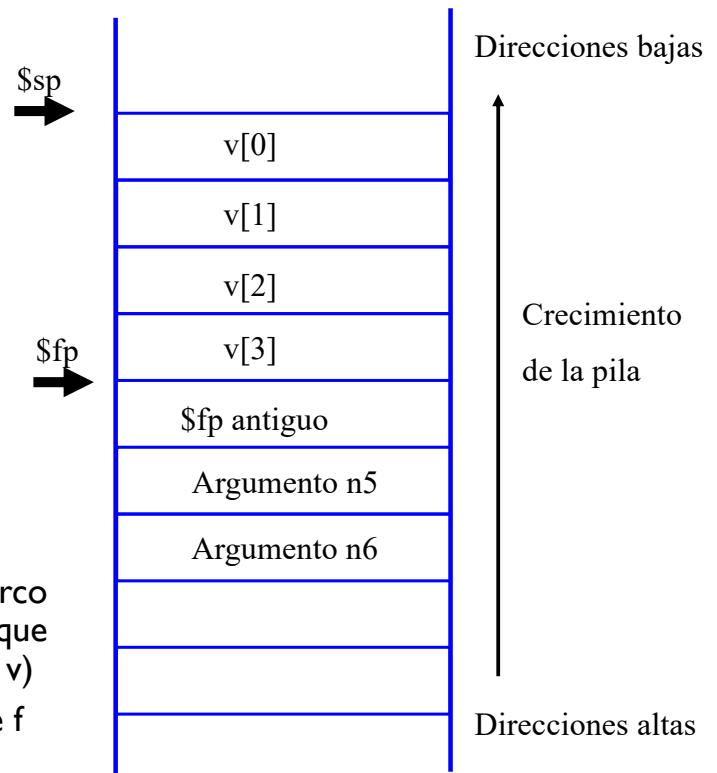
```

int f (int n1, n2, n3,
       n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}

```



- ▶ A continuación, `f` debe **reservar** en el marco de pila **espacio** para las variables locales que no quepan en registros (en este ejemplo `v`)
- ▶ Para este ejemplo se ha considerado que `f` no modifica ningún registro

## Acceso a parámetros y variables locales usando el marco de pila

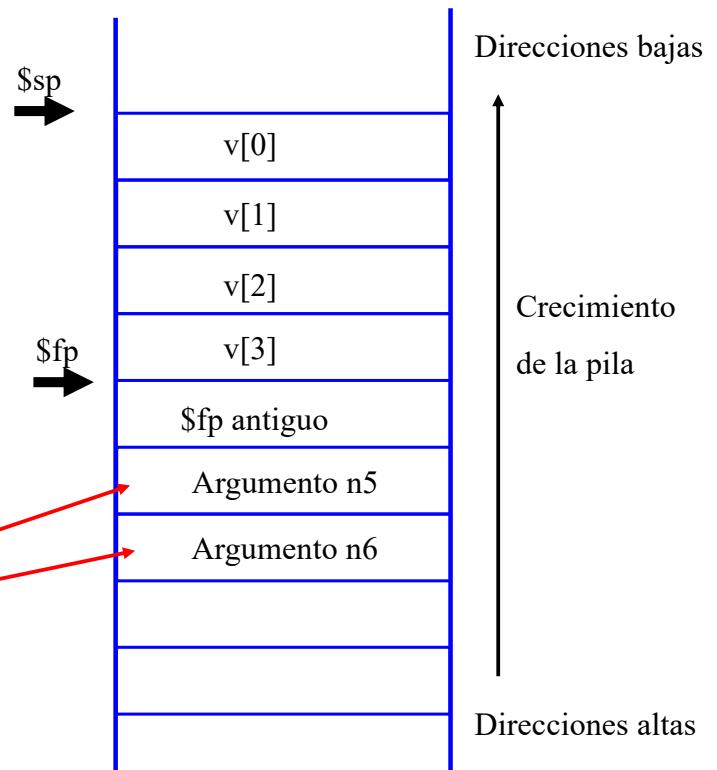
```

int f (int n1, n2, n3,
       n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
  
```

- ▶ El valor de n1 está en \$a0
- ▶ El valor de n5 está en 4(\$fp)
- ▶ El valor de n6 está en 8(\$fp)
- ▶ El valor de v[ 3 ] está en -4(\$fp)
- ▶ El valor de v[ 0 ] está en -16(\$fp)



Código para realizar la llamada a  
f(int n1, n2, n3, n4, n5, n6)

Para la llamada: f( 3, 4, 23, 12, 6, 7 );

```
li    $a0, 3
li    $a1, 4
li    $a2, 23
li    $a3, 12
addu $sp, $sp, -8
li    $t0, 6
sw    $t0, ($sp)
li    $t0, 7
sw    $t0, 4($sp)
jal   f
```

Los cuatro primeros en registros \$ai

El resto en la pila

## Código de la función f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,          f:  
       n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

## Código de la función f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3, n4, n5, n6)          f:      addu    $sp, $sp, -4
{                                                 sw       $fp, ($sp)
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

Se guarda el valor de \$fp (\$fp antiguo)

## Código de la función f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,          f:      addu    $sp, $sp, -4
       n4, n5, n6)           sw      $fp, ($sp)
{                                         move    $fp, $sp
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

Se fija \$fp a la posición donde está  
guardado el actual \$fp

## Código de la función f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,          f:      addu    $sp, $sp, -4
       n4, n5, n6)                 sw      $fp, ($sp)
{                                         move    $fp, $sp
    int v[4];                         addu    $sp, $sp, -16
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

Se deja hueco para el vector v (16 bytes  
para 4 elementos de tipo int)

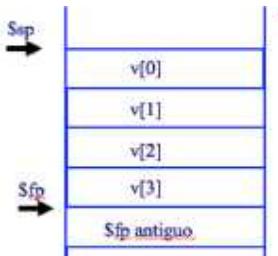
## Código de la función f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,
       n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

f:	
addu	\$sp, \$sp, -4
sw	\$fp, (\$sp)
move	\$fp, \$sp
addu	\$sp, \$sp, -16
add	\$t0, \$a0, \$a1
add	\$t0, \$t0, \$a2
add	\$t0, \$t0, \$a3
lw	\$t1, 4(\$fp)
add	\$t0, \$t0, \$t1
lw	\$t1, 8(\$fp)
add	\$t0, \$t0, \$t1



Se calcula la suma `n1+n2+n3+n4+n5+n6`

## Código de la función f(int n1, n2, n3, n4, n5, n6)

```

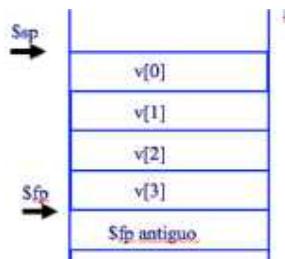
int f (int n1, n2, n3,
       n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}

```

Bucle



<b>f:</b> <b>bucle:</b>	addu \$sp, \$sp, -4 sw \$fp, (\$sp) move \$fp, \$sp addu \$sp, \$sp, -16 add \$t0, \$a0, \$a1 add \$t0, \$t0, \$a2 add \$t0, \$t0, \$a3 lw \$t1, 4(\$fp) add \$t0, \$t0, \$t1 lw \$t1, 8(\$fp) add \$t0, \$t0, \$t1 li \$t1, 0 # indice move \$t2, \$fp addi \$t2, \$t2, -16 # desplaz. li \$t3, 3 bgt \$t1, \$t3, fin sw \$t0, (\$t2) addi \$t2, \$t2, 4 addi \$t1, \$t1, 1 b bucle
----------------------------	---

# Código de la función

## $f(\text{int } n1, \text{ int } n2, \text{ int } n3, \text{ int } n4, \text{ int } n5, \text{ int } n6)$

```

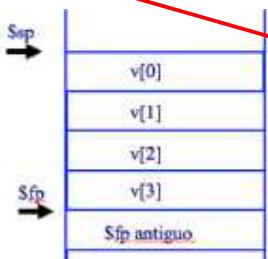
int f (int n1, n2, n3,
       n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}

```

Preparar  
el valor  
a retornar  
en \$v0



f:	addu      \$sp, \$sp, -4
	sw      \$fp, (\$sp)
	move      \$fp, \$sp
	addu      \$sp, \$sp, -16
	add      \$t0, \$a0, \$a1
	add      \$t0, \$t0, \$a2
	add      \$t0, \$t0, \$a3
	lw      \$t1, 4(\$fp)
	add      \$t0, \$t0, \$t1
	lw      \$t1, 8(\$fp)
	add      \$t0, \$t0, \$t1
	li      \$t1, 0    # indice
	move      \$t2, \$fp
	addi      \$t2, \$t2, -16 # desplaz.
	li      \$t3, 3
	bgt      \$t1, \$t3, fin
	sw      \$t0, (\$t2)
	addi      \$t2, \$t2, 4
	addi      \$t1, \$t1, 1
	b      bucle
	lw      \$v0, -12(\$fp)

bucle:

fin:

## Código de la función f(int n1, n2, n3, n4, n5, n6)

```

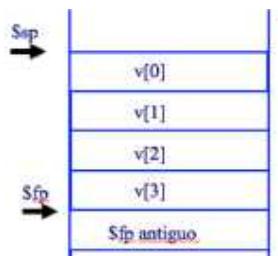
int f (int n1, n2, n3,
       n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}

```

Se restaura  
el valor de \$fp



f:	addu    \$sp, \$sp, -4 sw      \$fp, (\$sp) move   \$fp, \$sp addu    \$sp, \$sp, -16 add    \$t0, \$a0, \$a1 add    \$t0, \$t0, \$a2 add    \$t0, \$t0, \$a3 lw     \$t1, 4(\$fp) add    \$t0, \$t0, \$t1 lw     \$t1, 8(\$fp) add    \$t0, \$t0, \$t1 li     \$t1, 0      # indice move   \$t2, \$fp addi   \$t2, \$t2, -16 # desplaz. li     \$t3, 3 bgt    \$t1, \$t3, fin sw     \$t0, (\$t2) addi   \$t2, \$t2, 4 addi   \$t1, \$t1, 1 bucle
bucle:	
fin:	lw     \$v0, -12(\$fp) lw     \$fp, (\$fp)

## Código de la función f(int n1, n2, n3, n4, n5, n6)

```

int f (int n1, n2, n3,
       n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}

f:      addu    $sp, $sp, -4
        sw      $fp, ($sp)
        move   $fp, $sp
        addu   $sp, $sp, -16
        add    $t0, $a0, $a1
        add    $t0, $t0, $a2
        add    $t0, $t0, $a3
        lw     $t1, 4($fp)
        add    $t0, $t0, $t1
        lw     $t1, 8($fp)
        add    $t0, $t0, $t1
        li     $t1, 0      # indice
        move   $t2, $fp
        addi   $t2, $t2, -16 # desplaz.
        li     $t3, 3
        bgt   $t1, $t3, fin
        sw     $t0, ($t2)
        addi   $t2, $t2, 4
        addi   $t1, $t1, 1
        bucle
        b      bucle
fin:    lw     $v0, -12($fp)
        lw     $fp, ($fp)
        addu   $sp, $sp, 20
        jr     $ra

```

Se restaura  
la pila  
y se retorna

## Código posterior a la llamada a f(int n1, n2, n3, n4, n5, n6)

Para la llamada: f(3, 4, 23, 12, 6, 7);

```
li      $a0, 3
li      $a1, 4
li      $a2, 23
li      $a3, 12
addu   $sp, $sp, -8
li      $t0, 6
sw      $t0, 0($sp)
li      $t0, 7
sw      $t0, 4($sp)
jal    f
addu   $sp, $sp, 8
move   $a0, $v0
li      $v0, 1
syscall
```

Los cuatro primeros en registros \$ai

El resto en la pila

Deja el puntero de pila en el estado anterior

En \$v0 está el valor devuelto: lo imprime

## Variables locales en registros

- ▶ Siempre que se puede, las variables locales (int, double, char, ...) se almacenan en registros

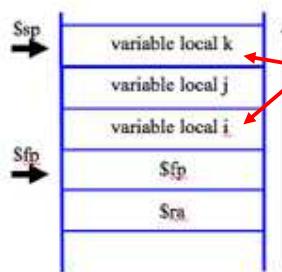
```
int f(...)  
{  
    int i, j, k;  
  
    i = 0;  
    j = 1;  
    k= i + j;  
    . . .  
}
```

```
f: . . .  
    li $t0, 0  
    li $t1, 1  
    add $t2, $t0, $t1  
    . . .
```

## Variables locales en pila

- ▶ Si no se pueden utilizar registros (no hay suficientes) se usa la pila

```
int f(...)  
{  
    int i, j, k;  
  
    i = 0;  
    j = 1;  
    k= i + j;  
    . . .  
}
```



```
f: . . .  
#espacio para variables loc.  
addu $sp, $sp, 12  
  
li $t0, 0  
sw $t0, -4($fp)  
li $t1, 1  
sw $t1, -8($fp)  
  
li $t0, -4($fp)  
li $t1, -8($fp)  
add $t2, $t0, $t1  
sw $t2, -12($fp)  
. . .
```

# Convenio de paso de parámetros

- ▶ **Convenio que describe:**
  - ▶ Uso del banco de registros generales.
  - ▶ Uso del banco de registros FPU.
  - ▶ Uso de la pila.
  - ▶ Afecta a código llamante y código llamado.
- ▶ **Distintos compiladores usan distintos convenios.**
  - ▶ ABI → *Application Binary Interface*.

## Convenio del MIPS

- ▶ El puntero de pila siempre alineado a doble palabra (múltiplo de 8)
- ▶ El procedimiento llamado reserva espacio para los registros \$a0..\$a3
  - ▶ El mínimo marco de pila ocupa 24 bytes

```
f:    addu $sp, $sp, -24
          sw   $ra, 20($sp)
          sw   $fp, 16$(sp)
```
- ▶ Durante el tema se ha utilizado un convenio más simplificado y ligeramente distinto al empleado realmente en el MIPS

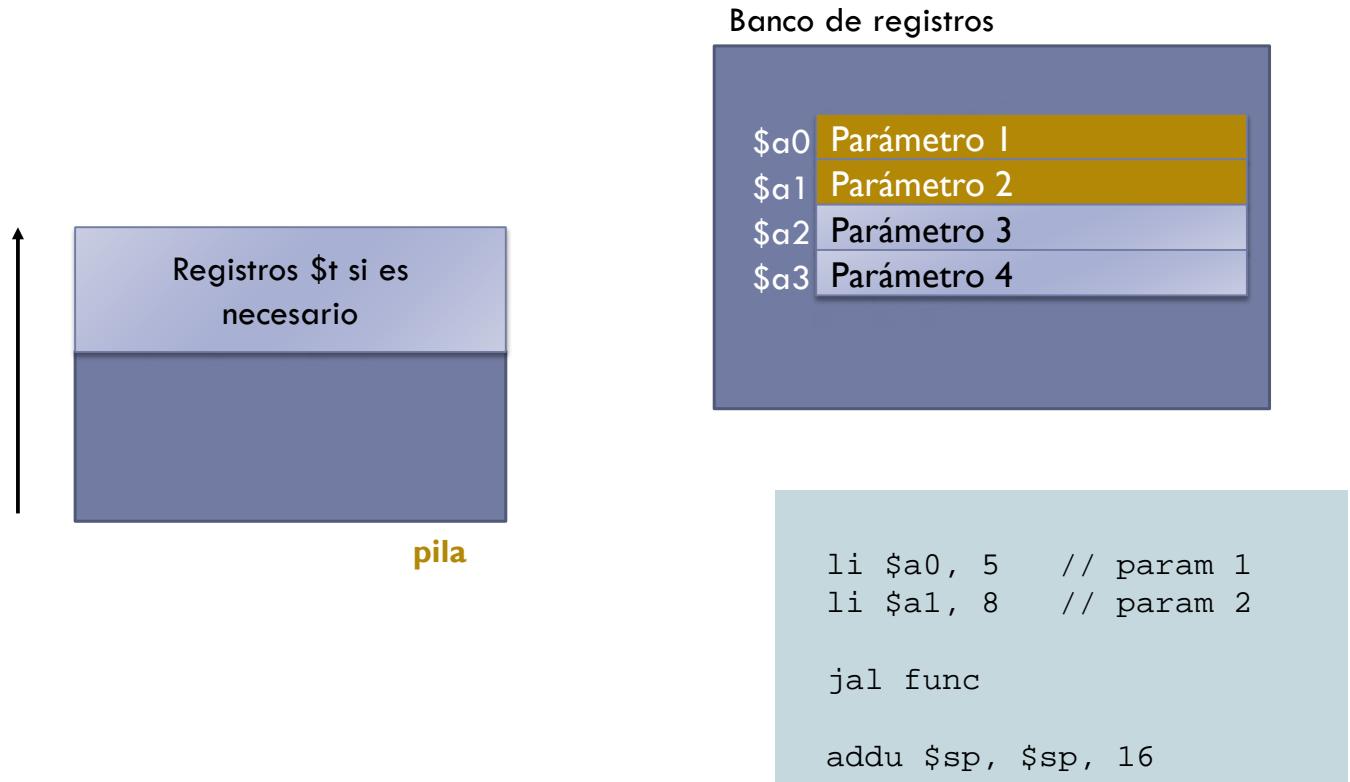
# Ejercicio

Considere una función denominada `func` que recibe tres parámetros de tipo entero y devuelve un resultado de tipo entero, y considere el siguiente fragmento del segmento de datos:

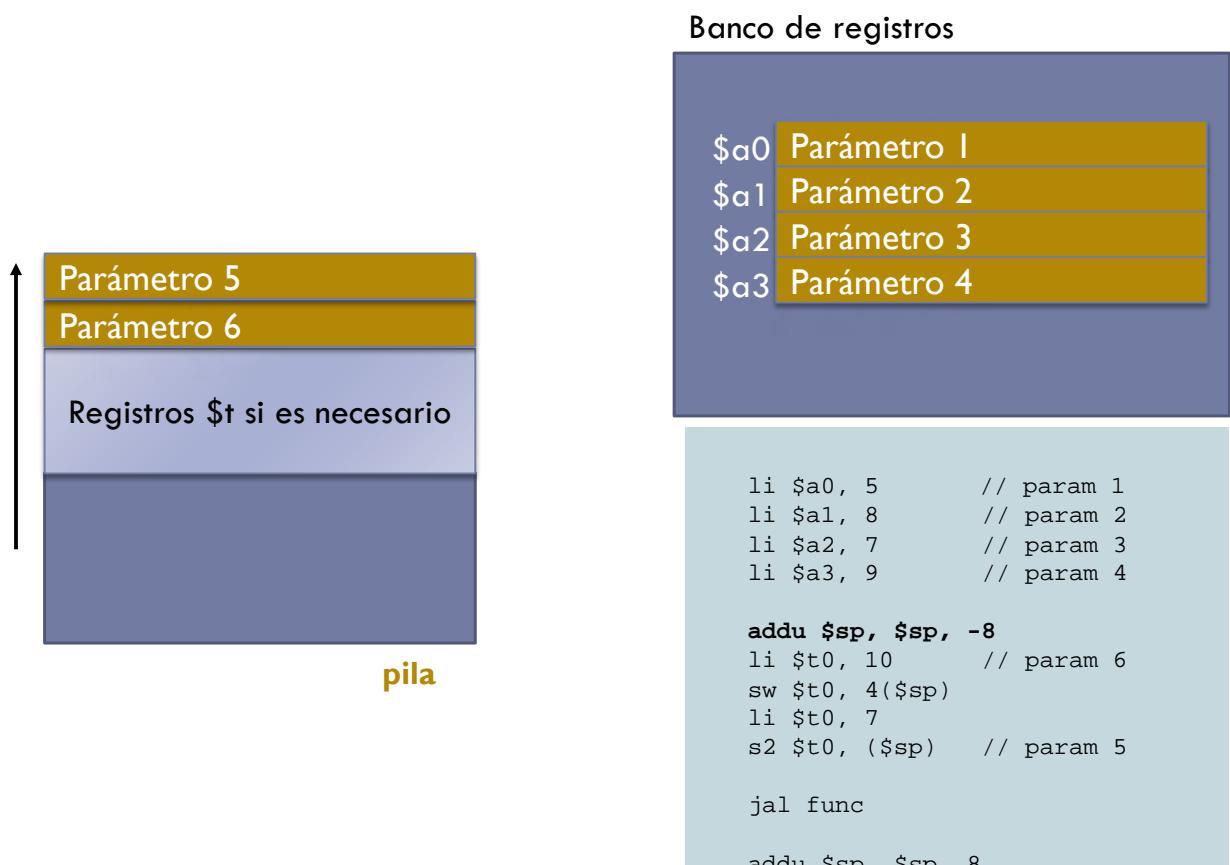
```
.data  
    a: .word 5  
    b: .word 7  
    c: .word 9  
  
.text
```

Indique el código necesario para poder llamar a la función anterior pasando como parámetros los valores de las posiciones de memoria `a`, `b` y `c`. Una vez llamada a la función deberá imprimirse el valor que devuelve la función.

## Paso de 2 parámetros



## Paso de 6 parámetros



## Llamada a subrutina Subrutina llamante

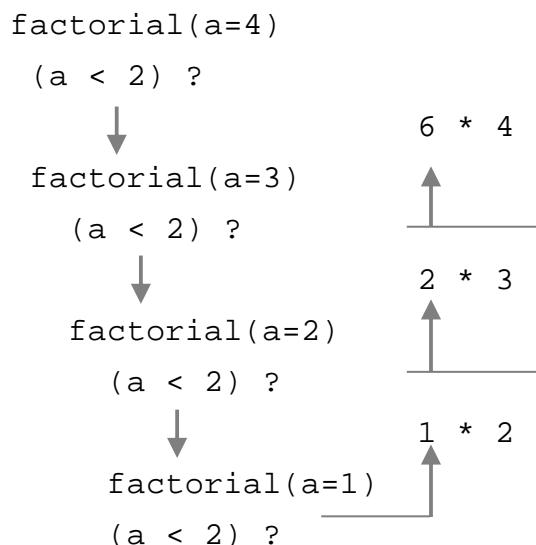
- ▶ Instrucción de salto “*and link*”
  - ▶ jal etiqueta
  - ▶ bal etiqueta
  - ▶ bltzal \$reg, etiqueta
  - ▶ bgezal \$reg, etiqueta
  - ▶ jalr \$reg
  - ▶ jalr \$reg, \$reg

## Ejemplo: factorial

```
int factorial ( int a )
{
    if (a < 2) then
        return 1 ;
    return a * factorial(a-1) ;
}

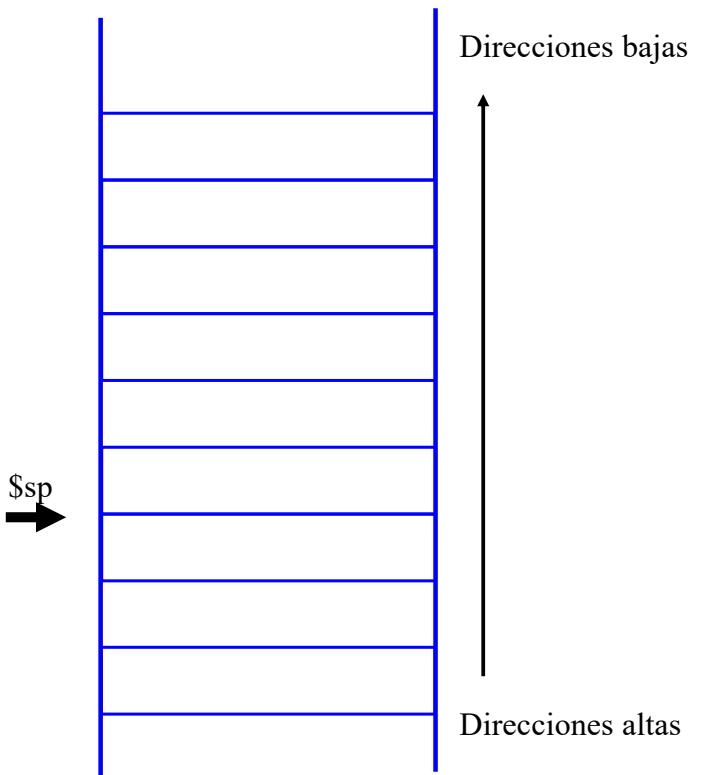
void main () {
    int resultado ;
    resultado=factorial(4) ;

    printf("f(4)=%d",resultado) ;
}
```



## Ejemplo: factorial

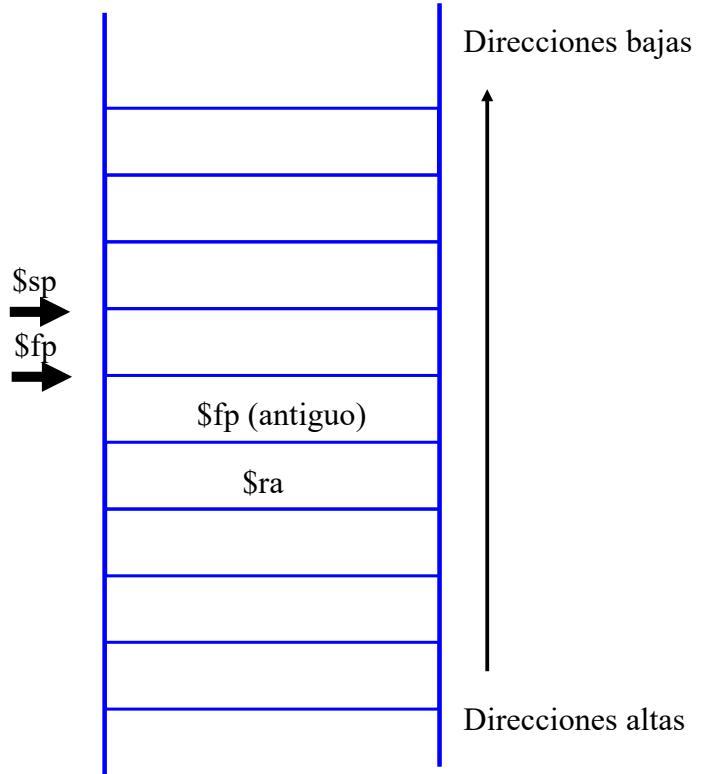
Justo antes de la llamada



## Ejemplo: factorial

factorial:

```
# frame stack
subu $sp $sp 12
sw    $ra 8($sp)
sw    $fp 4($sp)
addu $fp $sp 4
```



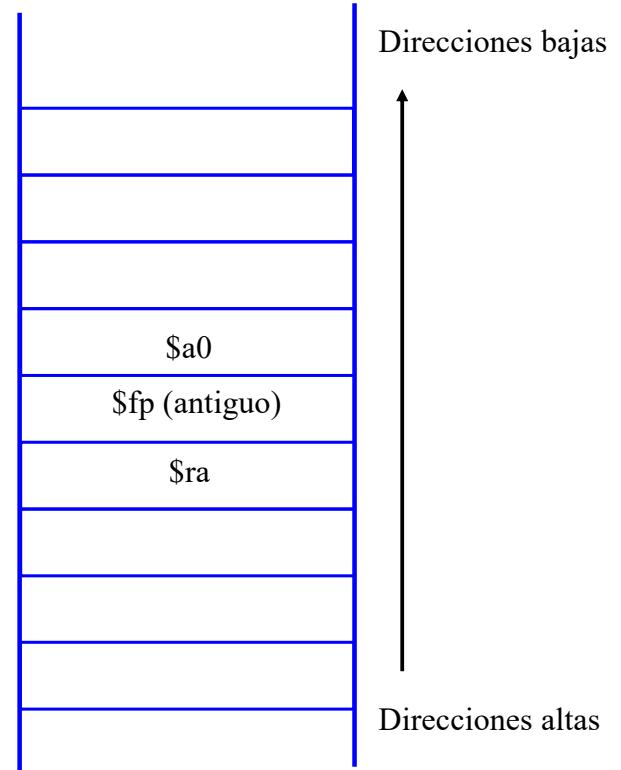
## Ejemplo: factorial

factorial:

```
# frame stack
    subu $sp $sp 12
    sw    $ra 8($sp)
    sw    $fp 4($sp)
    addu $fp $sp 4

    bge $a0 2 b_else
    li   $v0 1
    b    b_efl

b_else: sw $a0 -4($fp)
        addi $a0 $a0 -1
        jal factorial
        lw   $v1 -4($fp)
        mul $v0 $v0 $v1
```



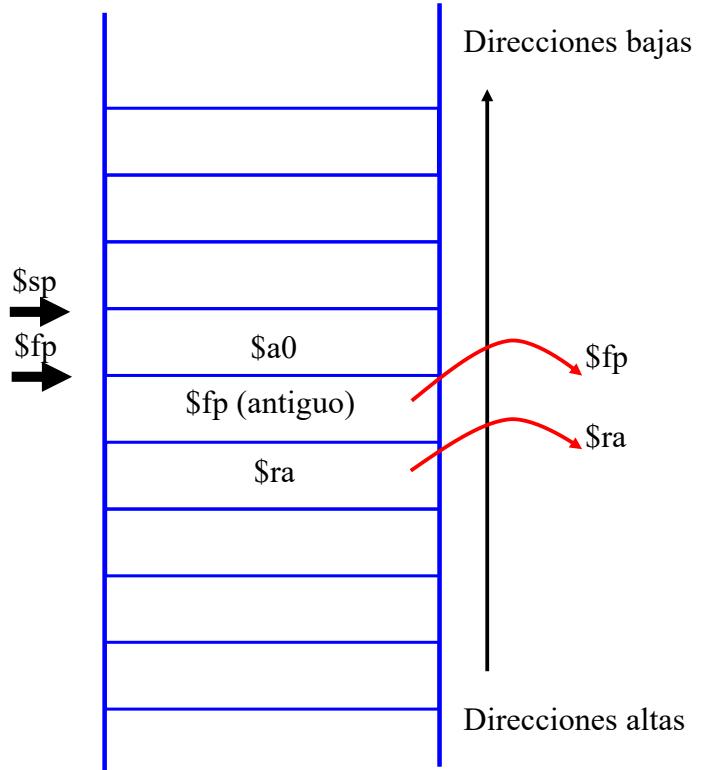
## Ejemplo: factorial

factorial:

```
# frame stack
    subu $sp $sp 12
    sw    $ra 8($sp)
    sw    $fp 4($sp)
    addu $fp $sp 8

    bge $a0 2 b_else
    li   $v0 1
    b    b_efs

b_else: sw $a0 -4($fp)
        addi $a0 $a0 -1
        jal factorial
        lw $v1 -4($fp)
        mul $v0 $v0 $v1
# end frame stack
b_efs:  lw $ra 8($sp)
        lw $fp 4($sp)
```



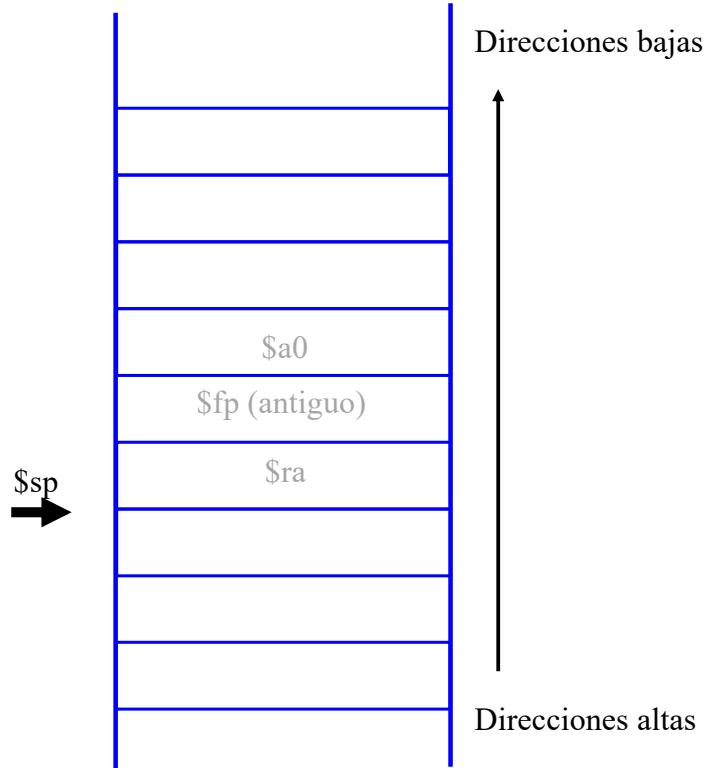
## Ejemplo: factorial

factorial:

```
# frame stack
    subu $sp $sp 12
    sw    $ra 8($sp)
    sw    $fp 4($sp)
    addu $fp $sp 8

    bge $a0 2 b_else
    li   $v0 1
    b    b_efs

b_else: sw $a0 -4($fp)
        addi $a0 $a0 -1
        jal factorial
        lw $v1 -4($fp)
        mul $v0 $v0 $v1
# end frame stack
b_efs:  lw $ra 8($sp)
        lw $fp 4($sp)
        addu $sp $sp 12
        jr $ra
```



# Paso de parámetros en C

```
int f1(int a, int b) {          f1: add $v0, $a0, $a1  
    int z;                      jr $ra  
    z = a + b;  
}
```

En C, todos los parámetros se pasan por valor

# Paso de parámetros en C

```
int f2(int a, int *b) {          f2: lw $t0, ($a1)
    int z;                      add $v0, $a0, $t0
    z = a + *b;                 sw $v0, ($a1)
    *b = z;                     jr $ra
}
}
```

En C, todos los parámetros se pasan por valor

- En este caso b es una dirección de memoria, pero se pasa por valor
- La función puede modificar el contenido de esa posición de memoria, puesto que se pasa su dirección

# Paso de parámetros en C

```
.data:  
    a: .word 8  
  
.text:  
  
main:  
    . . .  
    # llamada: f1(a, 9)  
    lw    $a0, a      # se pasa el valor de a  
    li    $a1, 9      # se pasa el valor 9  
    jal   f1  
    move $a0, $v0  
    li    $v0, 1  
    syscall         # se imprime el valor que devuelve  
    . . .
```

# Llamadas a las funciones anteriores

```
.data:  
    a: .word 8  
.text:  
    f2: lw $t0, ($a1)  
          add $v0, $a0, $t0  
          sw $v0, ($a1)  
          jr $ra  
  
    main:  
        . . .  
        # llamada: f2(1, &a)  
        li $a0, 1      # se pasa el valor 1  
        la $a1, a      # se pasa la dirección de a  
        jal f2  
        move $a0, $v0  
        li $v0, 1  
        syscall         # se imprime el valor que devuelve  
        lw $a0, a  
        syscall         # se imprime el valor de a  
    . . .
```

# Asignación dinámica de memoria en SPIM

## ▶ Llamada al sistema sbrk() en SPIM

- ▶ \$a0: número de bytes a reservar
- ▶ \$v0 = 9 (código de llamada al sistema)
- ▶ Devuelve en \$v0 la dirección del bloque reservado
- ▶ En SPIM no hay una llamada al sistema para liberar memoria (free)

```
int *p;  
  
p = malloc(20*sizeof(int));  
  
p[0] = 1;  
p[1] = 4;
```

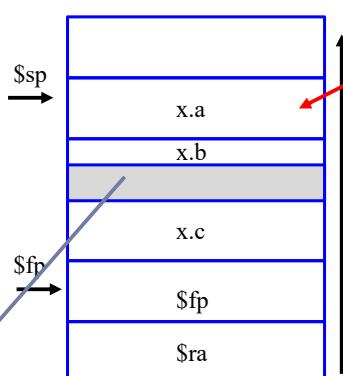
```
# se reservan 80 bytes  
li $a0, 80  
li $v0, 9 # código de  
# llamada  
syscall  
  
move $a0, $v0  
li $t0, 1  
sw $t0, ($a0)  
li $t0, 4  
sw $t0, 4($a0)
```

## Uso de estructuras de C (structs)

- ▶ Las variables locales de tipo struct se asignan en la pila
- ▶ C puede pasar estructuras completas a las funciones
  - ▶ Se pasan en la pila
- ▶ Una función en C puede devolver una estructura
  - ▶ La función que llama reserva espacio en la pila para que la función llamada deje allí el resultado a devolver

## Variables locales de tipo struct

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
int f(...)  
{  
    struct S x;  
  
    x.a = 2;  
    x.b = 'a';  
    x.c = 3;  
    . . .  
}
```

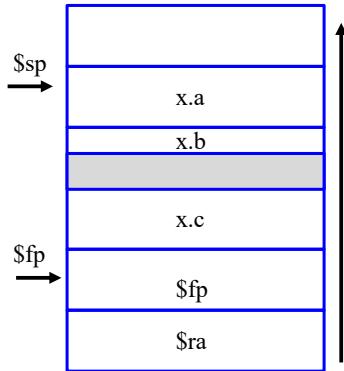


hueco para alineamiento de `x.c`

```
# se reserva el struct  
  
addu $sp, $sp, -12  
  
addu $t0, $fp, -12  
  
li    $t1, 2  
sw   $t1, 0($t0) #x.a  
  
li    $t1, 'a'  
sb   $t1, 4($t0) #x.b  
  
li    $t1, 3  
sw   $t1, 8($t0) #x.c
```

## Llamadas a funciones con struct

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
void f1(struct S p) {  
    . . .  
}  
  
int f2(...)  
{  
    struct S x;  
  
    x.a = 2;  
    x.b = 'a';  
    x.c = 3;  
    f1(x);  
    . . .  
}
```



```
# se reserva el struct  
  
addu $sp, $sp, -12  
addu $t0, $fp, -12  
li $t1, 2  
sw $t1, 0($t0)  
li $t1, 'a'  
sb $t1, 4($t0)  
li $t1, 3  
sw $t1, 8($t0)
```

# Llamadas a funciones con struct

```

struct S {
    int a;
    char b;
    int c;
}

void f1(struct S p) {
    . . .
}

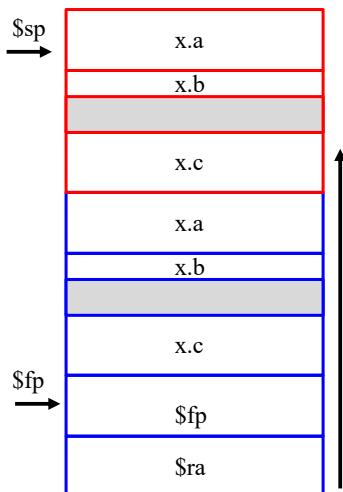
int f2(...)

{
    struct S x;

    x.a = 2;
    x.b = 'a';
    x.c = 3;
    f1(x);
    . . .
}

```

se copian los argumentos de f1 en la pila (paso por valor)



# se reserva el struct

```

addu $sp, $sp, -12
addu $t0, $fp, -12
li   $t1, 2
sw   $t1, 0($t0)
li   $t1, 'a'
sb   $t1, 4($t0)
li   $t1, 3
sw   $t1, 8($t0)

```

# proceso de llamada

```

addu $sp, $sp, -12
lw   $t1, 0($t0) #x.a
sw   $t1, 0($sp)

lb   $t1, 4($t0) #x.b
sb   $t1, 4($sp)

lw   $t1, 8($t0) #x.c
sw   $t1, 8($sp)
jal  f1
addu $sp, $sp, 12

```

## Llamadas a funciones con punteros a struct

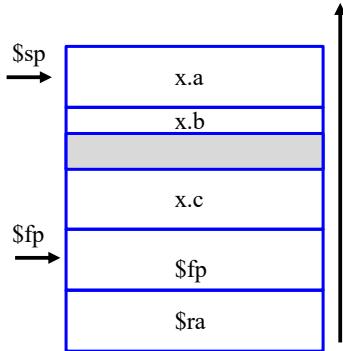
```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
void f1(struct S *p) {  
    . . .  
}  
  
int f2(...)  
{  
    struct S x;  
  
    x.a = 2;  
    x.b = 'a';  
    x.c = 3;  
    f1(&x);  
    . . .
```

En este caso se pasa  
una dirección  
en \$a0  
no se copia nada en la  
pila

# se reserva el struct

```
addu $sp, $sp, -12  
addu $t0, $fp, -12  
li $t1, 2  
sw $t1, 0($t0)  
li $t1, 'a'  
sb $t1, 4($t0)  
li $t1, 3  
sw $t1, 8($t0)
```

addi \$a0, \$fp, -12  
jal f1



## Funciones que devuelven structs

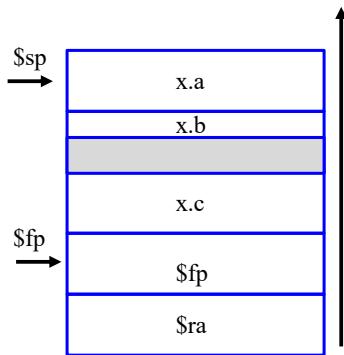
```
struct S {
    int a;
    char b;
    int c;
}

struct S f1() {
    . . .
}

int f2(...)

{
    struct S x;

    x= f1();
    . . .
}
```



```
# se reserva el struct

addu $sp, $sp, -12
addu $t0, $fp, -12
li $t1, 2
sw $t1, 0($t0)
li $t1, 'a'
sb $t1, 4($t0)
li $t1, 3
sw $t1, 8($t0)
```

# Funciones que devuelven structs

```

struct S {
    int a;
    char b;
    int c;
}

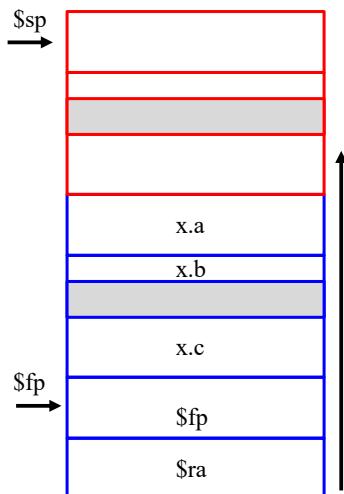
struct S f1() {
    struct S p;

    return p;
}
int f2(...)
{
    struct S x;

    x= f1();
    . . .

```

la función que llama reserva espacio en la pila para el resultado



# se reserva el struct

```

addu $sp, $sp, -12
addu $t0, $fp, -12
li $t1, 2
sw $t1, 0($t0)
li $t1, 'a'
sb $t1, 4($t0)
li $t1, 3
sw $t1, 8($t0)

```

```

addu $sp, $sp, -12
jal f1
#recupera el valor
lw $t0, 0($sp)
sw $t0, -12($fp)

lb $t0, 4($sp)
sb $t0, -8($sp)

lw $t0, 8($sp)
sw $t0, -4($fp)

addu $sp, $sp, 12

```

# Ejercicio

```
struct S {
    int a;
    char b;
    int c;
}

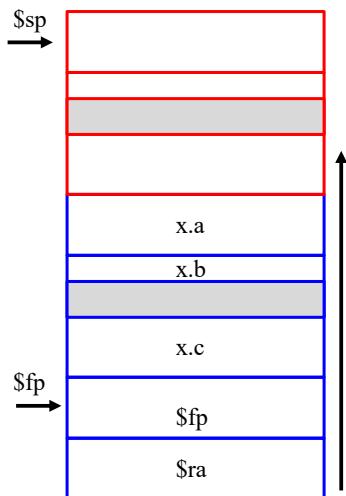
struct S f1() {
    struct S p;

    return p;
}

int f2(...)
{
    struct S x;

    x= f1();
    . . .
}
```

Escriba el código necesario para que f1() devuelva la estructura y la copie en la pila



# Traducción y ejecución de programas

- ▶ Elementos que intervienen en la traducción y ejecución de un programa:
  - ▶ Compilador
  - ▶ Ensamblador
  - ▶ Enlazador
  - ▶ Cargador

# Código compilado frente a interpretado

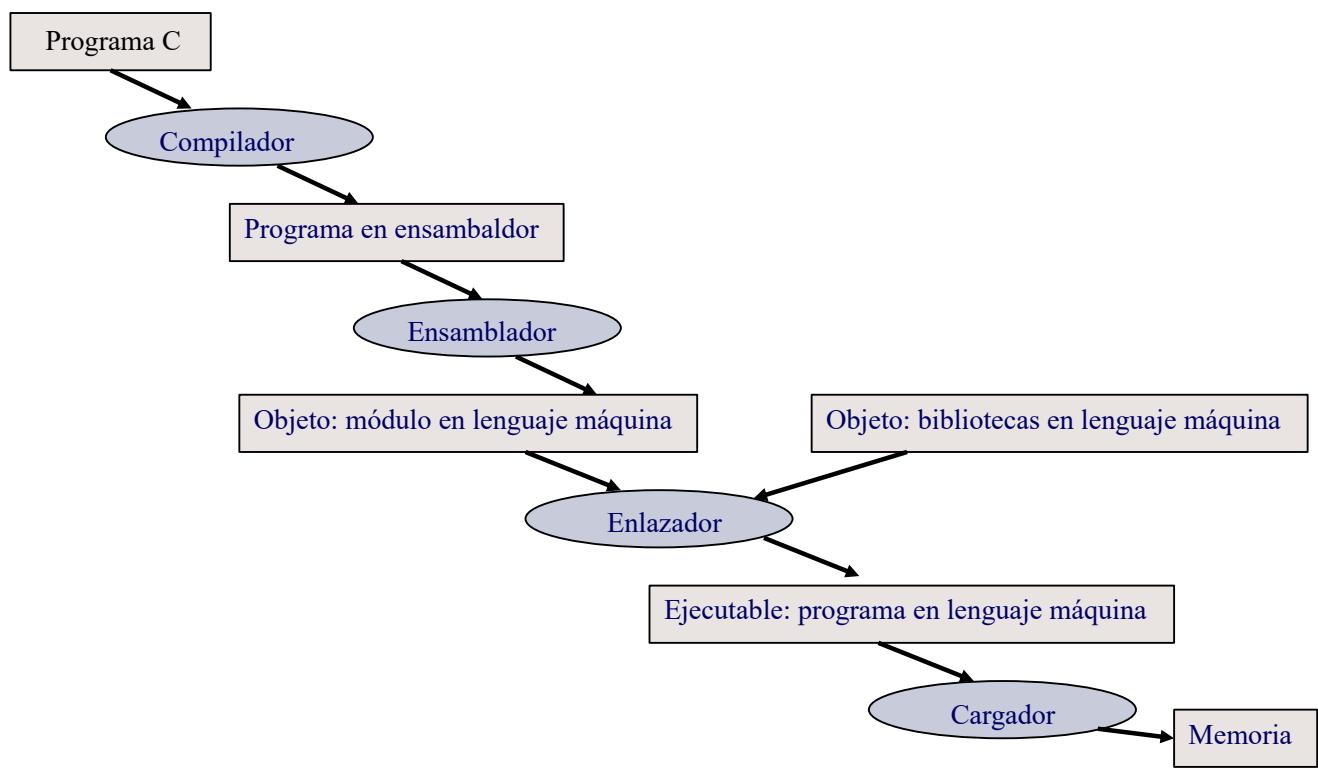
## ▶ Código compilado:

- ▶ Los programas son traducidos a código máquina de un computador
  - ▶ El código es ejecutado directamente por el computador
  - ▶ Generalmente más eficiente

## ▶ Código interpretado:

- ▶ Un interprete es un programa que ejecuta otros programas
- ▶ Un interprete ejecuta un conjunto de instrucciones independientes de la máquina. Las instrucciones son ejecutadas por un programa
- ▶ Ejemplo: Java es traducido a un *byte code* que es ejecutado por un interprete (*Java Virtual Machine*)
- ▶ Generalmente es más fácil escribir un interprete. Mayor portabilidad

## Etapas en la traducción y ejecución de un programa (programa en C)



# Compilador

- ▶ Entrada: lenguaje de alto nivel (C, C++, ...)
- ▶ Salida: código en lenguaje ensamblador
- ▶ Puede contener pseudoinstrucciones
- ▶ Una **pseudoinstrucción** es una instrucción que entiende el ensamblador pero que no tiene correspondencia directa con una instrucción en lenguaje máquina
  - ▶ `move $t1, $t2`  $\Rightarrow$  `or $t1, $t2, $zero`

## Ensamblador

- ▶ Entrada: código en lenguaje ensamblador
- ▶ Salida: Código objeto escrito en lenguaje máquina
- ▶ En ensamblador convierte las pseudoinstrucciones a instrucciones máquina
- ▶ Analiza las sentencias en ensamblador de forma independiente, sentencia a sentencia
- ▶ El ensamblador produce un fichero objeto (.o)

## Análisis de sentencias en ensamblador

- ▶ Se comprueba si la instrucción es correcta (código de operación, operandos, direccionamientos válidos, ...)
- ▶ Se comprueba si la sentencia tiene etiqueta. Si la tiene comprueba que el código simbólico no está repetido y le asigna el valor correspondiente a la posición de memoria que habrá de ocupar la instrucción o el dato.
- ▶ Construye una **tabla de símbolos** con todas las etiquetas simbólicas
  - ▶ En una primera fase o pasada se determinan todos los valores que no conllevan referencias adelantadas
  - ▶ En una segunda fase o pasada se resuelven aquellas etiquetas que han quedado pendientes

# Formato de un fichero objeto

- ▶ **Cabecera del fichero.** Describe el tamaño y posición de los elementos dentro del fichero
- ▶ **Segmento de texto:** contiene el código máquina
- ▶ **Segmento de datos:** contiene los datos de las variables globales
- ▶ **Información de reubicación:** identifica instrucciones o palabras de datos que dependen de una dirección absoluta cuando el programa se cargue en memoria
  - ▶ Cualquier etiqueta de `j` or `jal` (internas o externas)
  - ▶ Direcciones de datos
- ▶ **Tabla de símbolos:** etiquetas no definidas en este módulo (referencias externas)
- ▶ **Información de depuración.** Permite asociar instrucciones máquina con código C e interpretar las estructuras de datos

## Enlazador

- ▶ Entrada: ficheros en código objeto
- ▶ Salida: Código ejecutable
- ▶ Combina varios archivos objeto (.o) en un único fichero ejecutable
- ▶ Resuelve todas las referencias (instrucciones de salto y direcciones de datos)
- ▶ En enlazador asume que la primera palabra del segmento de texto está en la dirección 0x00000000
- ▶ Permite la compilación separada de ficheros
  - ▶ El cambio en un fichero no implica recompilar todo el programa completo
  - ▶ Permite el uso de funciones de biblioteca (.a)

## Enlazador

.o file 1

text 1

data 1

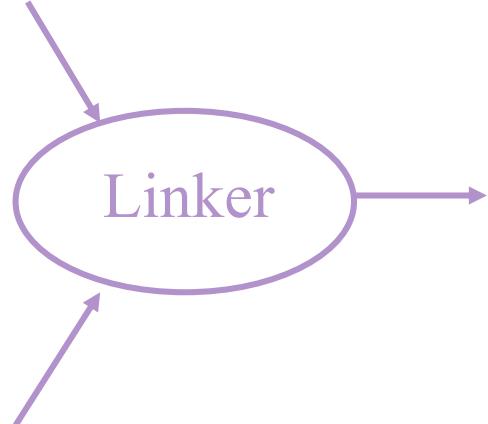
info 1

.o file 2

text 2

data 2

info 2



a.out

Relocated text 1

Relocated text 2

Relocated data 1

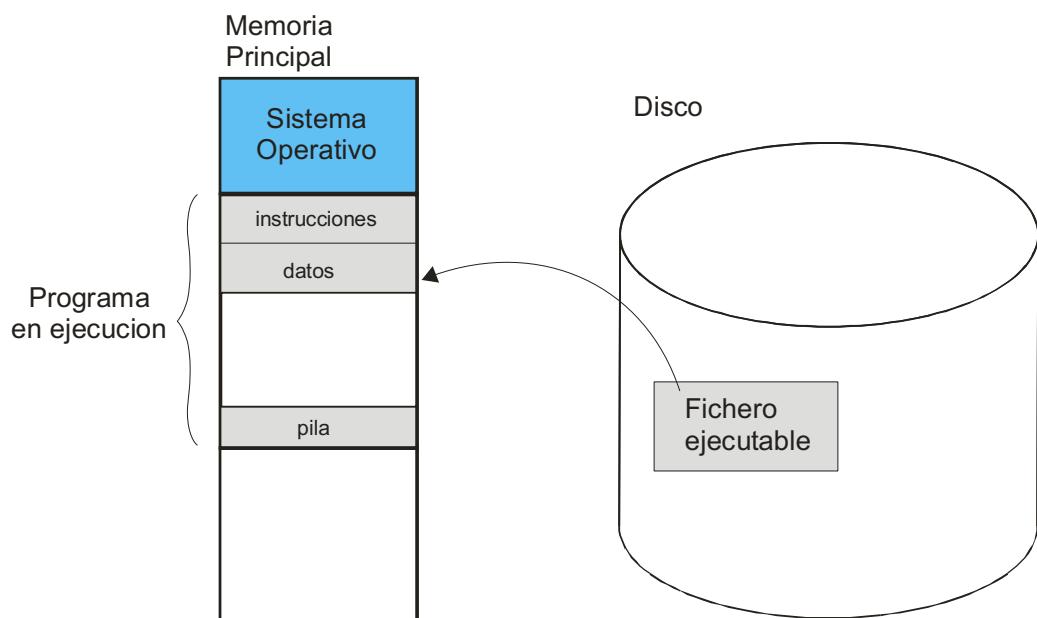
Relocated data 2

## Formato de un fichero ejecutable

- ▶ **Cabecera del fichero.** Describe el tamaño y posición de los elementos dentro del fichero. Incluye la dirección de inicio del programa
- ▶ **Segmento de texto:** contiene el código máquina
- ▶ **Segmento de datos:** contiene los datos de las variables globales con valor inicial
- ▶ **Información de reubicación:** en caso de utilizar bibliotecas dinámicas

# Cargador

- ▶ Lee un fichero ejecutable (a.out) y lo carga en memoria



# Cargador

- ▶ Forma parte del sistema operativo
- ▶ Lee la cabecera del ejecutable para determinar el tamaño de los segmentos de texto y datos
- ▶ Crea un nuevo espacio de direcciones en memoria para ubicar el segmento de texto, datos y pila
- ▶ Copia las instrucciones y los datos con valor inicial del fichero ejecutable (disco) a memoria
- ▶ Copia los argumentos que se pasan al programa en la pila
- ▶ Inicializa los registros. Fija el PC y el SP a sus posiciones

# Bibliotecas

- ▶ Una **biblioteca** es una colección de objetos normalmente relacionados entre sí
- ▶ Los módulos objetos de los programas pueden incluir referencias a símbolos definidos en alguno de los objetos de una biblioteca (funciones o variables exportadas)
- ▶ Las **bibliotecas del sistema** son un conjunto de bibliotecas predefinidas que ofrecen servicios a las aplicaciones
- ▶ Tipos:
  - ▶ Bibliotecas estáticas: se enlazan con los ficheros objeto para producir un fichero ejecutable que incluye todas las referencias resueltas. Un cambio en la biblioteca implica volver a enlazar y generar el ejecutable
  - ▶ Bibliotecas dinámicas (**DLL, dynamically linked library**)

## Bibliotecas dinámicas

- ▶ Las rutinas de las bibliotecas no se enlazan en el archivo ejecutable y no se cargan hasta que el programa se ejecuta
- ▶ El programa incluye información para la localización de las bibliotecas y la actualización de las referencias externas durante la ejecución
- ▶ Ventajas:
  - ▶ Da lugar a ejecutables más pequeños.
  - ▶ Solo se carga de la biblioteca aquello que se utiliza durante la ejecución.
  - ▶ El cambio en una biblioteca no afecta al ejecutable. No se necesita volver a generar un nuevo ejecutable.

# Ejemplo

## C ⇒ ASM ⇒ Obj ⇒ Exe ⇒ Ejecución

### Programa C: ejemplo.c

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    int i, sum = 0;
    for (i = 1; i <= 10; i++)
        sum = sum + i + i;

    printf ("La suma 1 + ... +10 es %d\n", sum);
}
```

`printf()`: función de biblioteca en libc.a

# Compilación

```
.text
.align 2
.globl main
main:
    subu $sp,$sp,24
    sw $ra, 20($sp)
    sw $a0, 4($sp)
    sw $a1, 8($sp)

    li $t0, 0
    li $t1, 0
bucle:
    bgt $t0, 10, fin
    add $t1, $t1, $t0
    addi $t0, $t0, 1
    b bucle

fin:
    la $a0, str
    li $a1, $t1
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    lw $a0, 4($sp)
    lw $a1, 8($sp)
    addiu $sp,$sp,24
    jr $ra
.data
.align 0

str:
    .asciiz "La suma
    i + . . . +10 es
    %d\n"
```

# Compilación

```
.text
    .align 2
    .globl main
main:
    subu $sp,$sp,24
    sw $ra, 20($sp)
    sw $a0, 4($sp)
    sw $a1, 8($sp)

    li $t0, 0
    li $t1, 0
bucle:
    bgt $t0, 10, fin
    add $t1, $t1, $t0
    addi $t0, $t0, 1
    b bucle

fin:
    la $a0, str
    li $a1, $t1
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    lw $a0, 4($sp)
    lw $a1, 8($sp)
    addiu $sp,$sp,24
    jr $ra
.data
.align 0 7 pseudo-instructions
.data:
str:.asciiz " La
suma 1 + ... +10
es %d\n"
```

# Compilación

## Eliminación de pseudoinstrucciones

```
.text
.align 2
.globl main
main:
    addiu $29,$29,-24
    sw $31, 20($29)
    sw $4, 4($29)
    sw $5, 8($29)
    ori $8, $0, 0
    ori $9, $0, 0
bucle:
    slti $1, $8, 11
    beq $1, $0, fin
    add $9, $9, $8
    addi $8, $8, 1
    bgez $0, bucle
fin:
    lui $4, l.str
    ori $4, $4, r.str
    addu $4, $0, $9
    jal printf
    addu $2, $0, $0
    lw $31, 20($29)
    lw $4, 4($29)
    lw $5, 8($29)
    addiu $29,$29,24
    jr $31
```

# Compilación

## Asignación de direcciones

00	addiu \$29,\$29,-24	2c	lui \$4, l.str
04	sw \$31, 20(\$29)	30	ori \$4, \$4, r.str
08	sw \$4, 4(\$29)	34	addu \$4, \$0, \$9
0c	sw \$5, 8(\$29)	38	jal printf
10	ori \$8, \$0, 0	3c	addu \$2, \$0, \$0
14	ori \$9, \$0, 0	40	lw \$31, 20(\$29)
18	slti \$1, \$8, 11	44	lw \$4, 4(\$29)
1c	beq \$1, \$0, fin	48	lw \$5, 8(\$29)
20	add \$9, \$9, \$8	4c	addiu \$29,\$29,24
24	addi \$8, \$8, 1	50	jr \$31
28	bgez \$0, bucle		

# Compilación

## Creación de la tabla de símbolos y de reubicación

### ■ Tabla de símbolos

Etiqueta	dirección (en modulo)	tipo
main:	0x00000000	global text
bucle:	0x0000001c	local text
str:	0x00000000	local data

### ■ Información de reubicación

Dirección	tipo Instr.	Dependencia
0x0000002c	lui	l.str
0x00000030	ori	r.str
0x00000038	jal	printf

# Compilación

## Resolver etiquetas relativas a PC

00 addiu \$29,\$29,-24	2c lui \$4, l.str
04 sw \$31, 20(\$29)	30 ori \$4, \$4, r.str
08 sw \$4, 4(\$29)	34 addu \$4, \$0, \$9
0c sw \$5, 8(\$29)	38 jal printf
10 ori \$8, \$0, 0	3c addu \$2, \$0, \$0
14 ori \$9, \$0, 0	40 lw \$31, 20(\$29)
18 slti \$1, \$8, 11	44 lw \$4, 4(\$29)
1c beq \$1, \$0, <b>3</b>	48 lw \$5, 8(\$29)
20 add \$9, \$9, \$8	4c addiu \$29,\$29,24
24 addi \$8, \$8, 1	50 jr \$31
28 bgez \$0, <b>-4</b>	

## Segmento de texto en el fichero objeto

0x000000	00100111011101111111111101000
0x000004	1010111101111100000000000010100
0x000008	1010111101001000000000000000100
0x00000c	10101111010010100000000000001000
0x000010	10001101000000000000000000000000000
0x000014	101011010010000000000000000011100
0x000018	00101000010100000000000000001011
0x00001c	00010000010000000000000000000011
0x000020	0000000100101000100100000100000
0x000024	001000010000100000000000000000001
0x000028	000001000000000011111111111100
0x00002c	001110000000100000000000000000000
0x000030	001101001000010000000000000000000
0x000034	0000000100100100000000000000100001
0x000038	000011000000000000000000000000000
0x00003c	000000000000000000001000001000001
0x000040	1000111101111100000000000010100
0x000044	1000111101001000000000000000100
0x000048	10001111010010100000000000001000
0x00004c	000000111100000000000000000011000
0x000050	00000000000000001110100000001000

# Enlazado

- ▶ Combinar ejemplo.o y libc.a
- ▶ Crear las direcciones de memoria absolutas
- ▶ Modificar y mezclar las tablas de símbolos y de reubicación
- ▶ Symbol Table

Label	Address
main:	0x00000000
loop:	0x0000001c
str:	0x10000430
printf:	0x000003b0
	...

- ▶ Relocation Information

Address	Instr.Type	Dependency
0x00000030	lui	0x0000002c
0x00000038	ori	l.str
	jal	r.str
		printf
		...

## Enlazado Resolver las direcciones

00	addiu \$29,\$29,-24	2c	lui \$4, 4096
04	sw \$31, 20(\$29)	30	ori \$4, \$4, 1072
08	sw \$4, 4(\$29)	34	addu \$4, \$0, \$9
0c	sw \$5, 8(\$29)	38	jal 812
10	ori \$8, \$0, 0	3c	addu \$2, \$0, \$0
14	ori \$9, \$0, 0	40	lw \$31, 20(\$29)
18	slti \$1, \$8, 11	44	lw \$4, 4(\$29)
1c	beq \$1, \$0, 3	48	lw \$5, 8(\$29)
20	add \$9, \$9, \$8	4c	addiu \$29,\$29,24
24	addi \$8, \$8, 1	50	jr \$31
28	bgez \$0, -4		

# Enlazado

- ▶ **Generación del fichero ejecutable**
  - ▶ Único segmento de texto
  - ▶ Único segmento de datos
  - ▶ Cabecera con información sobre las secciones

# Guía rápida del ensamblador del MIPS 32

## Registros MIPS y usos

Nombre del Registro	Número	Uso
zero	0	Constante 0
at	1	Reservada para ensamblador
v0	2	Evaluación de expresiones y resultado de funciones
v1	3	Evaluación de expresiones y resultado de funciones
a0	4	Argumento 1
a1	5	Argumento 2
a2	6	Argumento 3
a3	7	Argumento 4
t0...t7	8...15	Temporal (No se guarda valor entre llamadas)
s0...s7	16...23	Temporal (Se guarda valor entre llamadas)
t8, t9	24, 25	Temporal (No se guarda valor entre llamadas)
k0, k1	26, 27	Reservado para el kernel del Sistema operativo
gp	28	Puntero al área global
sp	29	Puntero de pila
fp	30	Puntero de marco de pila
ra	31	Dirección de retorno, usada por llamadas a función

## Llamadas al sistema

Servicio	Código de llamada	Argumentos	Resultado
print_int	1	\$a0 = entero	
print_float	2	\$f12 = real (32 bits)	
print_double	3	\$FP12 = real (64 bits)	
print_string	4	\$a0 = cadena	
read_int	5		Entero (en \$v0)
read_float	6		Real 32 bits (en \$f0)
read_double	7		Real 64 bits (en \$FP0)
read_string	8	\$a0 = buffer, \$a1 = longitud	
sbrk	9	\$a0 = cantidad	Dirección (en \$v0)
exit	10		
print_char	11	\$a0 = byte	
read_char	12		\$v0 (cód ASCII)

## Directivas del ensamblador

.ascii cadena	Almacena la cadena en memoria, pero no termina con NULL ('\0')
.asciiz cadena	Almacena la cadena en memoria y coloca un NULL ('\0') al final de esta
.byte b1, ... bn	Almacena N valores en bytes sucesivos de memoria
.data	Las siguientes definiciones de datos que aparezcan se almacenan en el segmento de datos. Puede llevar un argumento que indica la dirección a partir de donde se empezarán a almacenar los datos.
.double d1, ..., dn	Almacena N valores reales de doble precisión en direcciones consecutivas de memoria.
.extern etiqueta n	Declara que los datos almacenados a partir de <i>etiqueta</i> ocupan N bytes y que <i>etiqueta</i> es un símbolo global. Esta directiva permite al ensamblador almacenar datos en una zona del segmento de datos que puede ser accedida a través del registro \$gp.
.float f1, ..., fn	Almacena N valores reales de precisión simple en posiciones consecutivas de memoria.
.globl simbolo	Declara un símbolo global que se puede referenciar desde otros programas.
.half h1, ..., hn	Almacena N números de 16 bits en medias palabras consecutivas.
.text	Las instrucciones que siguen a esta directiva se ponen en el segmento de código. Puede llevar un parámetro que indica dónde empieza la zona de código.
.word w1, ..., wn	Almacena N cantidades de 32 bits (1 palabra) en posiciones consecutivas de memoria

## Instrucciones aritméticas y lógicas

En todas las instrucciones siguientes, Src2 puede ser tanto un registro como un valor inmediato (un entero de 16 bits) y en aquellas donde pone inm solo acepta un valor inmediato

add Rdest, Rsrc1, Src2	Suma con desbordamiento
addi Rdest, Rsrc1, inm	Suma un número inmediato con desbordamiento
addu Rdest, Rsrc1, Src2	Suma sin desbordamiento
addiu Rdest, Rsrc1, inm	Suma un número inmediato sin desbordamiento
and Rdest, Rsrc1, Src2	Operación lógica AND
andi Rdest, Rsrc1, inm	Operación lógica AND con un número inmediato
div Rsrc1, Rsrc2	Divide con desbordamiento. Deja el cociente en el registro <i>lo</i> y el resto en el registro <i>hi</i>
divu Rsrc1, Rsrc2	Divide sin desbordamiento. Deja el cociente en el registro <i>lo</i> y el resto en el registro <i>hi</i>
div Rdest, Rsrc1, Rrc2	Divide con desbordamiento
divu Rdest, Rsrc1, Rrc2	Divide sin desbordamiento
mul Rdest, Rsrc1, Src2	Multiplica sin desbordamiento
mult Rsrc1, Rsrc2	Multiplica, la parte baja del resultado se deja en el registro <i>lo</i> y la parte alta en el registro <i>hi</i>
multu Rsrc1, Rsrc2	Multiplica sin desbordamiento, la parte baja del resultado se deja en el registro <i>lo</i> y la parte alta en el registro <i>hi</i>
mod Rdest, Rsrc1, Rsrc2	Módulo de la división con desbordamiento
modu Rdest, Rsrc1, Rsrc2	Módulo de la división sin desbordamiento
nop	No realiza ninguna operación
nor Rdest, Rsrc1, Src2	Operación Lógica NOR
or Rdest, Rsrc1, Src2	Operación Lógica OR
ori Rdest, Rsrc1, inm	Operación Lógica OR con un inmediato
rem Rdest, Rsrc1, Rsrc2	Módulo de la división con desbordamiento
rotr rdest, rsrc1, inm	Rotación hacia la derecha de inm bits
sll Rdest, Rsrc1, inm	Desplazamiento lógico de bits a la izquierda

srl Rdest, Rsrc1, imm	Desplazamiento lógico de bits a la derecha
sra Rdest, Rsrc1, imm	Desplazamiento aritmético de bits a la derecha
sub Rdest, Rsrc1, Src2	Resta (con desbordamiento)
subu Rdest, Rsrc1, Src2	Resta (sin desbordamiento)
xor Rdest, Rsrc1, Src2	Operación Lógica XOR
xori Rdest, Rsrc1, imm	Operación Lógica XOR con número inmediato

## Instrucciones de manipulación de constantes

li Rdest, inmediato	Cargar valor inmediato
lui Rdest, inmediato	Cargar los 16 bits de la parte baja del valor inmediato en la parte alta del registro. Los bits de la parte baja se pone a 0.

## Instrucciones de comparación

En todas las instrucciones siguientes, Src2 puede ser un registro o un valor inmediato (de 16 bits).

slt Rdest, Rsrc1, Rsrc2	Pone Rdest a 1 si Rsrc1 es menor a Rsrc2, en otro caso pone 0 (para números con signo).
sltu Rdest, Rsrc1, Rsrc2	Pone Rdest a 1 si Rsrc1 es menor a Rsrc2, en otro caso pone 0 (para números sin signo).
slti Rdest, Rsrc1, imm	Pone Rdest a 1 si Rsrc1 es menor a imm, en otro caso pone 0 (para números con signo).
sltiu Rdest, Rsrc1, imm	Pone Rdest a 1 si Rsrc1 es menor a imm, en otro caso pone 0 (para números sin signo).

## Instrucciones de almacenamiento

sb Rsrc, dirección	Almacena el byte más bajo de Rsrc en la dirección indicada.
sh Rsrc, dirección	Almacena la media palabra (16 bits) baja de un registro en la dirección de memoria indicada.
sw Rsrc, dirección	Almacena la Rsrc en la dirección indicada.

## Instrucciones de bifurcación y salto

En todas las instrucciones siguientes, Src2 puede ser un registro o un valor inmediato. Las instrucciones de bifurcación (branch) usan un desplazamiento de 16 bits con signo; por lo que se puede saltar  $2^{15+1}$  instrucciones hacia delante o  $2^{15}$  instrucciones hacia atrás. Las instrucciones de salto (jump) contienen un campo de dirección de 26 bits.

b etiqueta	Bif. incondicional a la instrucción que está en etiqueta.
beq Rsrc1, Src2, etiqueta	Bif. condicional si Rsrc1 es igual a Src2.
beqz Rsrc, etiqueta	Bif. condicional si el registro Rsrc es igual a 0.
bge Rsrc1, Src2, etiqueta	Bif. condicional si el registro Rsrc1 es mayor o igual a Src2 (con signo).
bgeu Rsrc1, Src2, etiq	Bif. condicional si el registro Rsrc1 es mayor o igual a Src2 (sin signo).
bgez Rsrc, etiqueta	Bif. condicional si el registro Rsrc es mayor o igual a 0.
bgezal Rsrc, etiqueta	Bif. condicional si el registro Rsrc es mayor o igual a 0. Guarda la dirección actual en el registro \$ra (\$31)
bgt Rsrc1, Src2, etiqueta	Bif. condicional si el registro Rsrc1 es mayor que Src2 (con signo).
bgtu Rsrc1, Src2, etiqueta	Bif. condicional si el registro Rsrc1 es mayor que Src2 (sin signo).
bgtz Rsrc, etiqueta	Bif. condicional si Rsrc es mayor que 0.
ble Rsrc1, Src2, etiqueta	Bif. condicional si Rsrc1 es menor o igual a Src2 (con signo).
bleu Rsrc1, Src2, etiqueta	Bif. condicional si Rsrc1 es menor o igual a Src2 (sin signo).
blez Rsrc, etiqueta	Bif. condicional si Rsrc es menor o igual a 0.
blt Rsrc1, Src2, etiqueta	Bif. condicional si Rsrc1 es menor que Src2 (con signo).
bltu Rsrc1, Src2, etiqueta	Bif. condicional si Rsrc1 es menor que Src2 (sin signo).
bltz Rsrc, etiqueta	Bif. condicional si Rsrc es menor que 0.
bne Rsrc1, Src2, etiqueta	Bif. condicional si Rsrc1 no es igual a Src2.
bnez Rsrc, etiqueta	Bif. condicional si Rsrc no es igual a 0.
j etiqueta	Salto incondicional.
jal etiqueta	Salto incondicional, almacena la dirección actual en \$ra (\$31).
jalr Rsrc	Salto incondicional, almacena la dirección actual en \$ra (\$31).
jalr Rsrc1, Rsrc2	Salto incondicional, almacena la dirección actual en Rsrc1.
jr Rsrc	Salto incondicional.

## Instrucciones de carga

la Rdest, dirección	Carga dirección en Rdest (el valor de dirección, no el contenido)
lb Rdest, dirección	Carga el byte de la dirección especificada y extiende el signo
lbu Rdest, dirección	Carga el byte de la dirección especificada, no extiende el signo
lh Rdest, dirección	Carga 16 bits de la dirección especificada, se extiende el signo
lhu Rdest, dirección	Carga 16 bits de la dirección especificada, no se extiende signo
lw Rdest, dirección	Carga una palabra de la dirección especificada.

## Instrucciones de transferencia de datos

move Rdest, Rsrc	Mueve el contenido del registro Rsrc al registro Rdest.
mfhi Rdest	Mueve el contenido del registro HI al registro Rdest.
mflo Rdest	Mueve el contenido del registro LO al registro Rdest.
mthi Rsrc	Mueve el contenido del registro Rsrc al registro HI.
mtlo Rsrc	Mueve el contenido del registro Rsrc al registro LO.

## Instrucciones aritméticas del coprocesador de coma flotante

abs.s fd, fs, ft	Valor absoluto de un número real de 32 bits.
abs.d fd, fs, ft	Valor absoluto de un número real de 64 bits.
add.s fd, fs, ft	Suma los registros fs y ft y almacena el resultado en fd (float)
add.d fd, fs, ft	Suma los registros fs y ft y almacena el resultado en fd (double)
div.s fd, fs, ft	Divide fs entre ft y deja el resultado en fd (float)
div.d fd, fs, ft	Divide fs entre ft y deja el resultado en fd (double)
mul.s fd, fs, ft	Multiplica los registros fs y ft y deja su resultado en fd. (float)
mul.d fd, fs, ft	Multiplica los registros fs y ft y deja su resultado en fd. (double)
rsqrt.s fd, fs	1.0 entre el resultado de la raíz cuadrada de fs (float)
rsqrt.d fd, fs	1.0 entre el resultado de la raíz cuadrada de fs (double)
sqrts fd, fs	Raíz cuadrada de fs (float)
sqrtd fd, fs	Raíz cuadrada de fs (double)
sub.s fd, fs, ft	Resta (float)
sub.d fd, fs, ft	Resta (double)

## Instrucciones de carga y almacenamiento del coprocesador de coma flotante

l.s fs, dirección	Carga en fs el valor del float (32 bits) que se encuentra a partir de la dirección especificada.
l.d fd, dirección	Carga fd con el valor del double (64 bits) que se encuentra a partir de la dirección especificada.
s.s fs, dirección	Almacena el registro fs a partir de la dirección indicada. (float)
s.d fd, dirección	Almacena un double (64 bits) en la dirección indicada, el valor de 64 bits proviene de fd.
li.s fs, valor	Carga en el registro fs del coprocesador matemático el valor (float)
li.d fd, valor	Carga en el registro fd del coprocesador matemático el valor (double)

El campo dirección se puede representar utilizando direccionamiento absoluto, indirecto de registro o relativo a registro

## Instrucciones de transferencia de datos entre registros

mfc1 Rdest, CPsrc	Mueve el contenido del registro CPsrc del coprocesador en coma flotante al registro de la CPU Rdest.
mtc1 Rsrc, CPdest	Mueve el contenido del registro Rsrc de la CPU al registro Cpdest del coprocesador en coma flotante.
mov.s fd, fs	Mueve el contenido del registro fs al registro fd. (float)
mov.d fd, fs	Mueve el contenido del registro fs al registro fd. (double)

## Instrucciones de conversión

cvt.d.s fd, fs	Convierte un float en un double, el resultado se guarda en fd
cvt.d.w fd, Rsrc	Convierte un entero en un double, el resultado se guarda en fd
cvt.s.d fd, fs	Convierte un double en un float, el resultado se guarda en fd
cvt.s.w fd, Rsrc	Convierte un entero en un float, el resultado se guarda en fd
cvt.w.s Rdest, fs	Convierte un float en un entero, el resultado se guarda en Rdest
cvt.w.d Rdest, fs	Convierte un double en un entero, el resultado se guarda en Rdest



## **Parte V**

### **Tema 4. El procesador**



Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 4 (I) El procesador

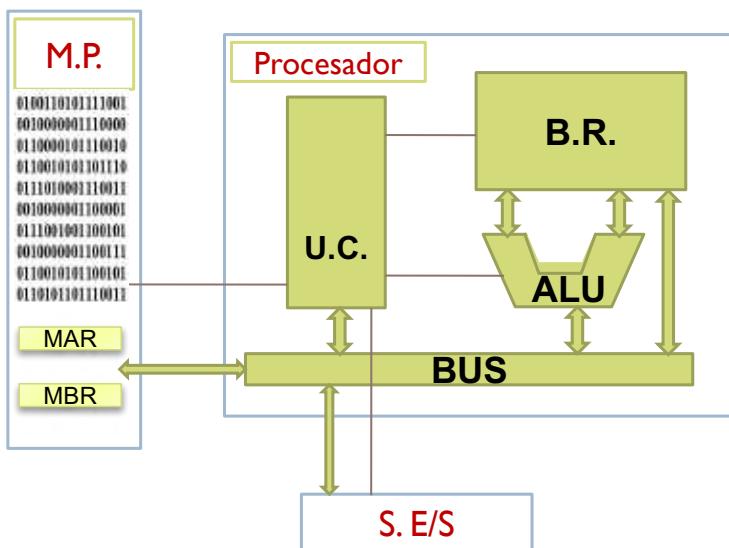
Estructura de Computadores  
Grado en Ingeniería Informática



# Contenido

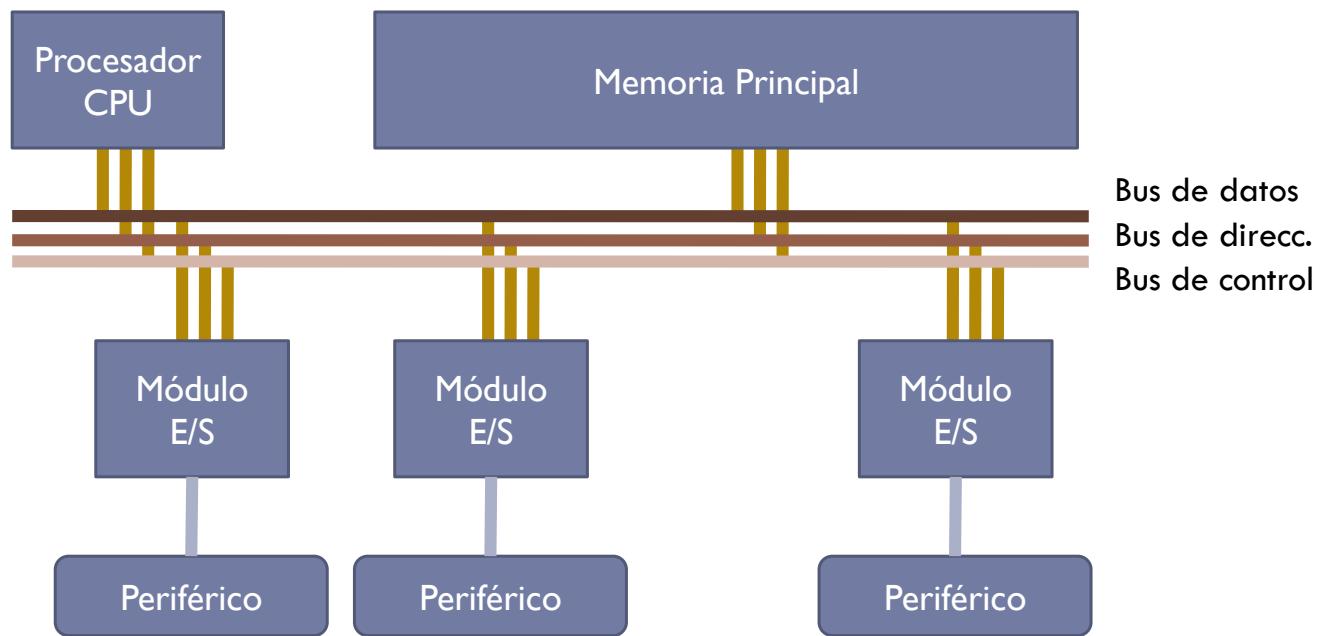
1. Elementos de un computador
2. Organización del procesador
3. La unidad de control
4. Ejecución de instrucciones
5. Modos de ejecución
6. Arranque de un computador
7. Interrupciones
8. Diseño de la unidad de control
9. Arranque de un computador
10. Prestaciones y paralelismo

# Motivación

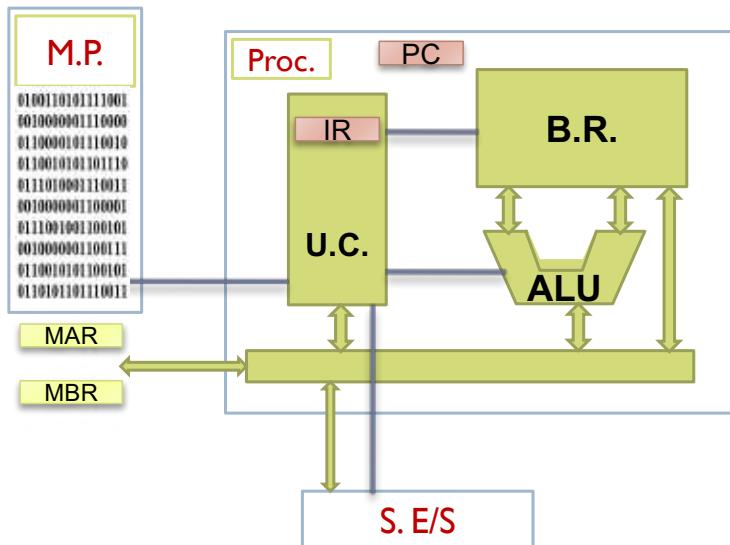


- En el tema 3 se estudian las instrucciones máquina
- En el tema 4 se estudia cómo se ejecutan las instrucciones en el computador

# Componentes de un computador

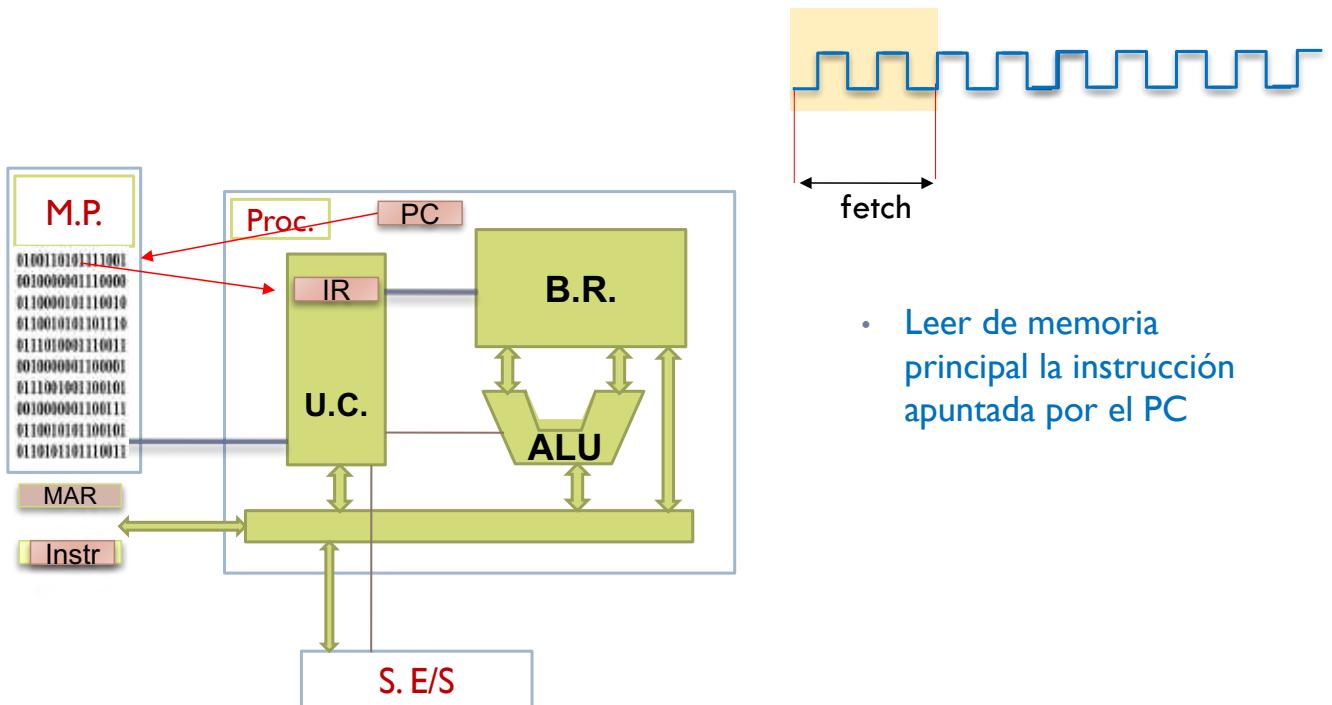


# Funcionamiento básico de la UC



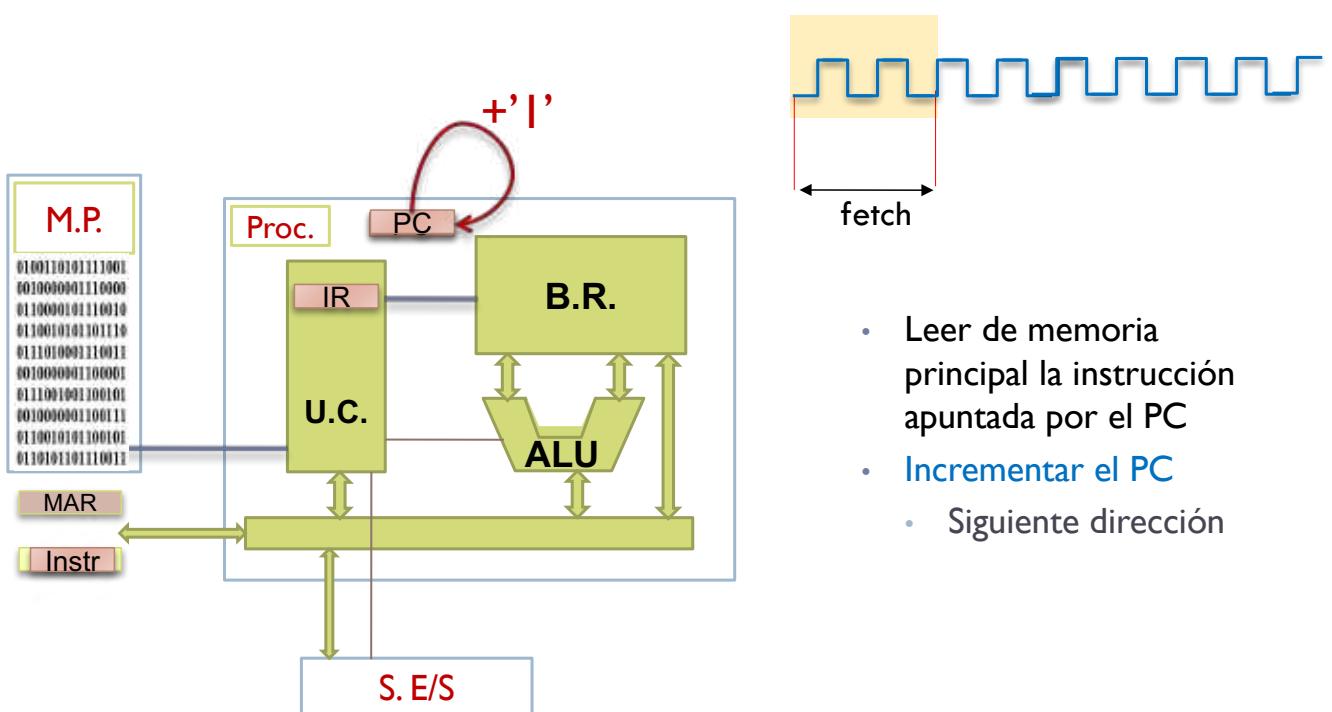
- Ejecutar instrucciones máquina

## Funcionamiento básico de la UC

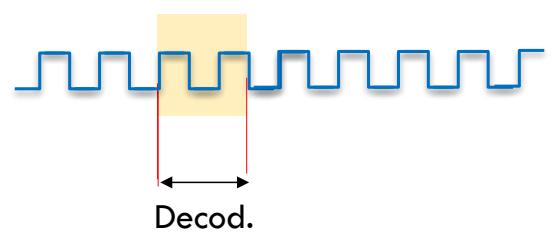
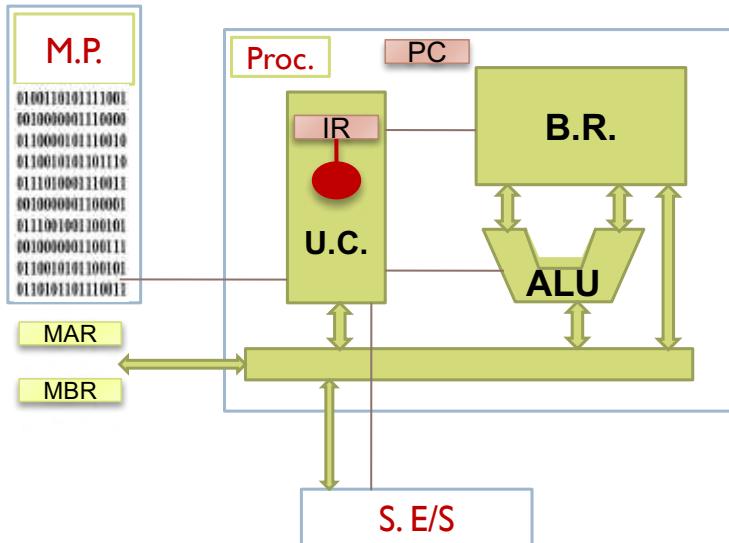


- Leer de memoria principal la instrucción apuntada por el PC

# Funcionamiento básico de la UC

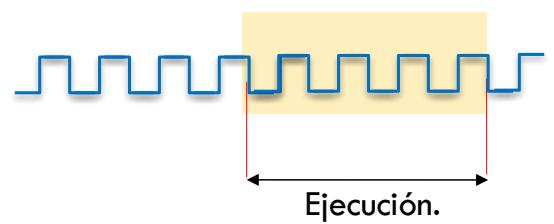
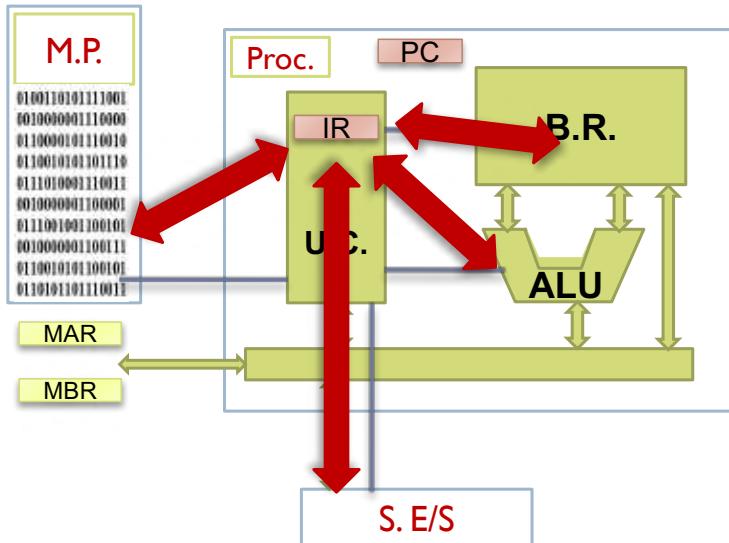


# Funcionamiento básico de la UC



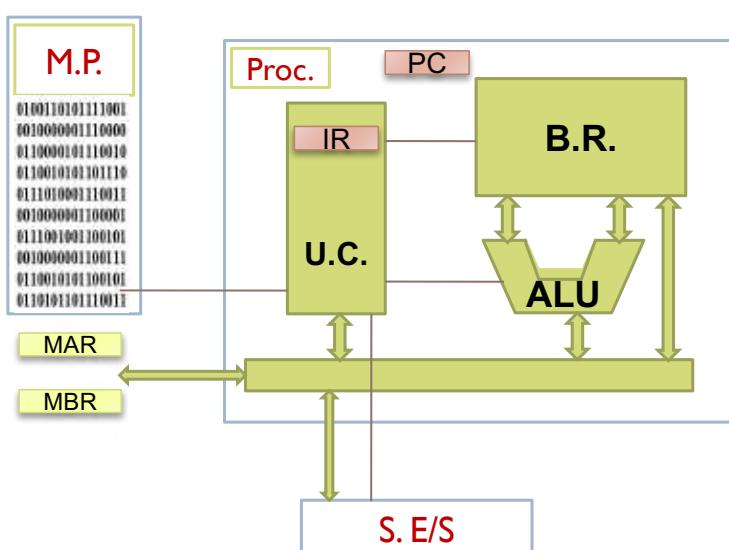
- Leer de memoria principal la instrucción apuntada por el PC
- Incrementar el PC
- Decodificación

# Funcionamiento básico de la UC



- Leer de memoria principal la instrucción apuntada por el PC
- Incrementar PC
- Decodificar instrucción
- Ejecución

# Otras funciones de la UC

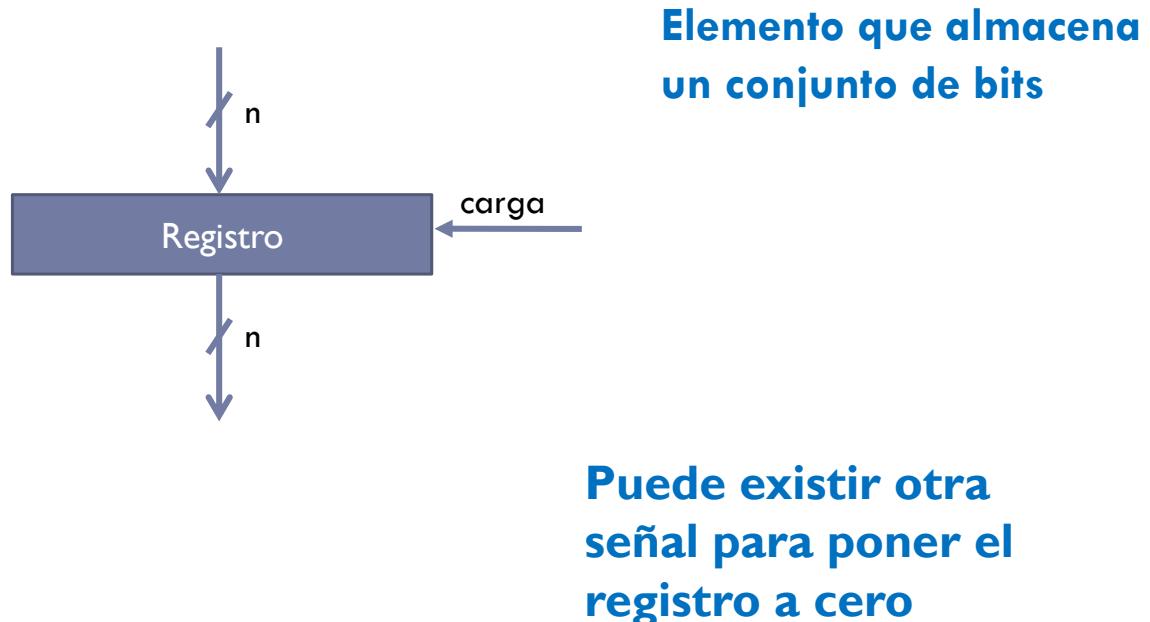


- **Resolver situaciones anómalas**
  - Instrucciones ilegales
  - Accesos a memoria ilegales
  - ...
- **Atender las interrupciones**
- **Controlar la comunicación con los periféricos**

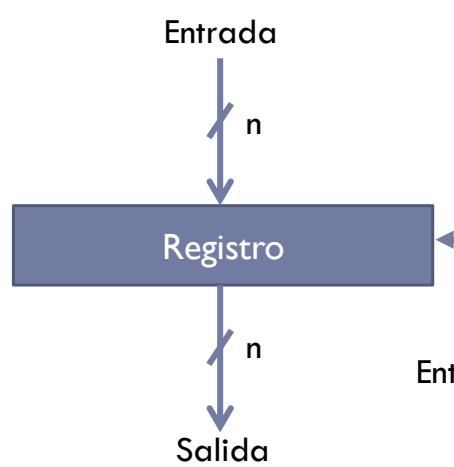
# Componentes del procesador

- ▶ Banco de registros
- ▶ Unidad aritmético-lógica
- ▶ Unidad de control
- ▶ Memoria caché

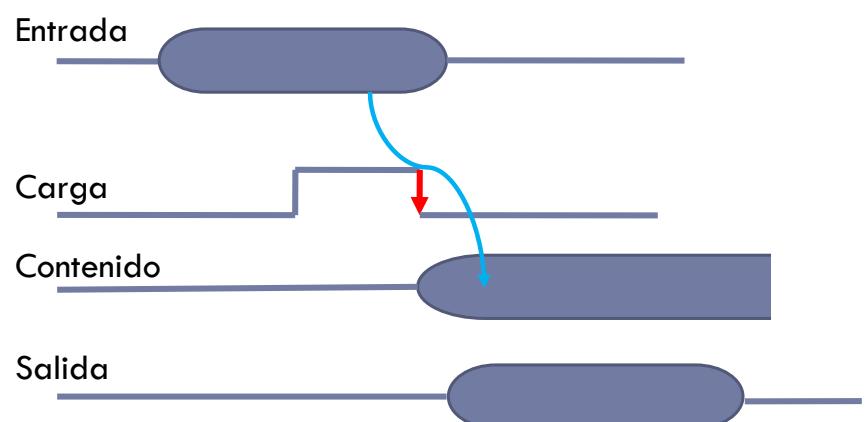
# Registros



# Registros



Puede existir otra señal para poner el registro a cero



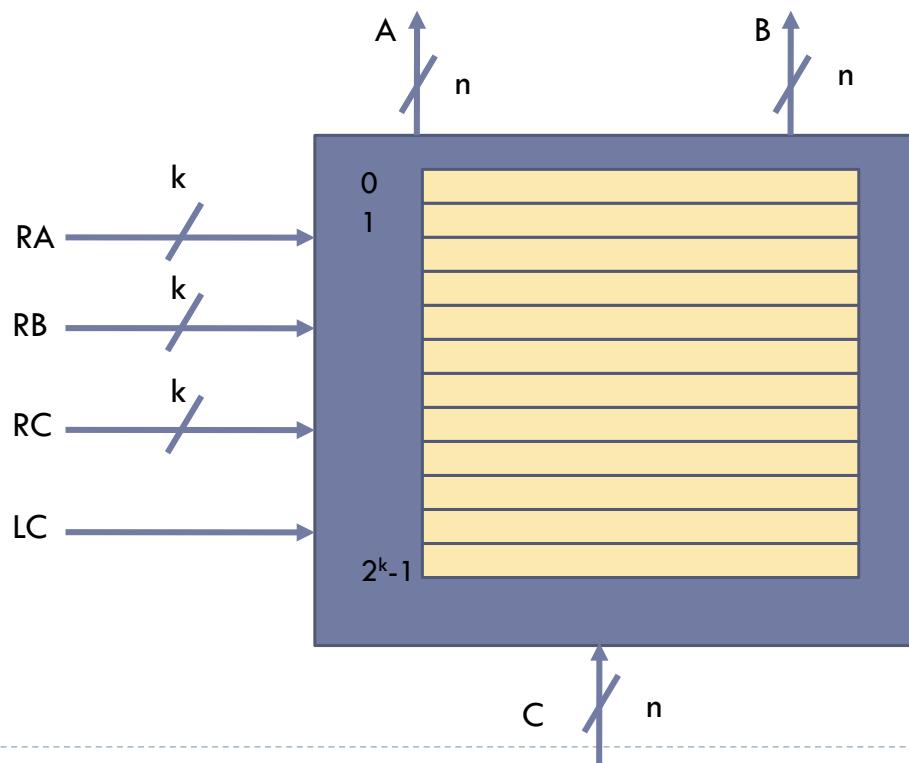
# Tipos de registros

- ▶ **Visibles al programador**
  - ▶ En el MIPS: \$t0, \$t1, ....
- ▶ **Registros no visibles**
  - ▶ Registros temporales
- ▶ **Registros de control y estado**
  - ▶ Contador de programa, PC (*program counter*)
  - ▶ Registro de instrucción, IR (*instruction register*)
  - ▶ Registro de direcciones de memoria, MAR (*memory address register*)
  - ▶ Registro de datos de memoria, MBR (*memory buffer register*)
  - ▶ Registro de estado, SR (*status register*)

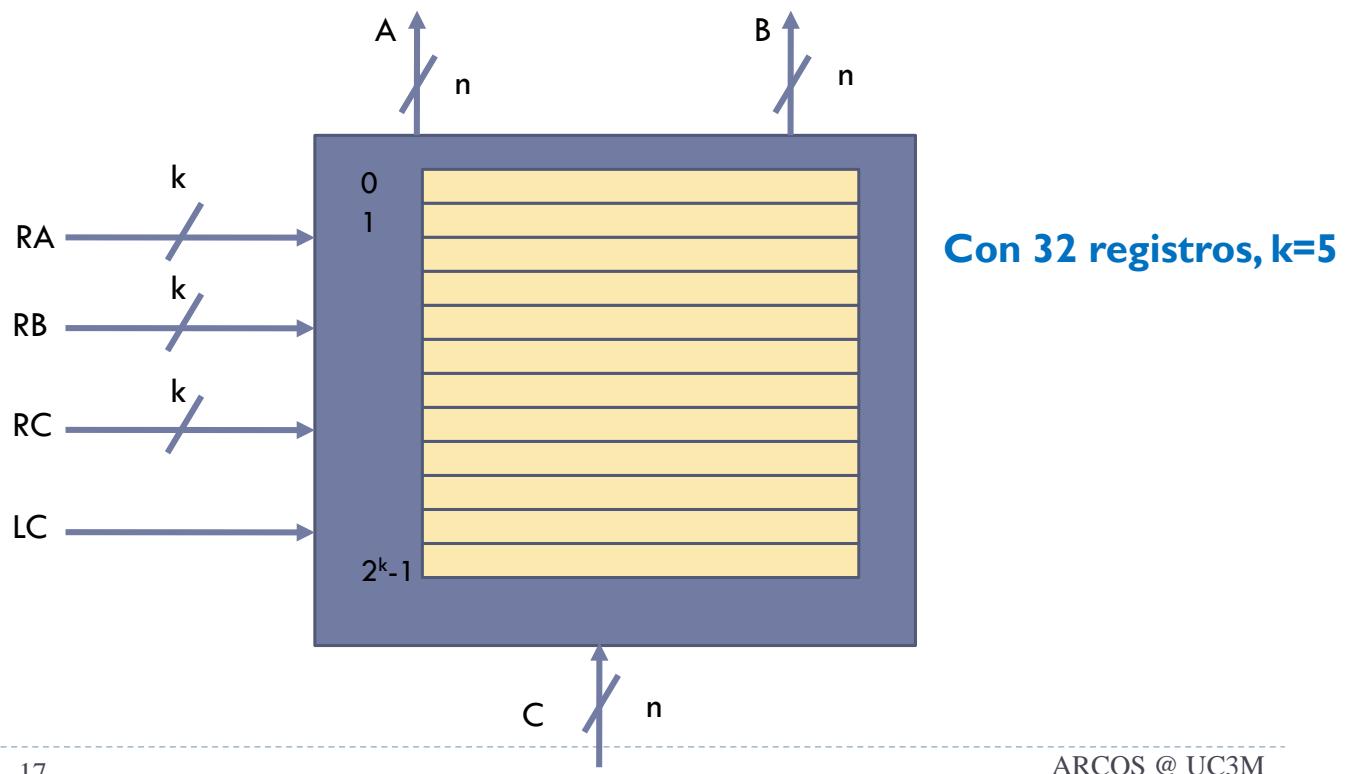
## Banco de registros

- ▶ Agrupación de registros.
- ▶ Típicamente un número de registros potencia de 2.
  - ▶  $n$  registros  $\rightarrow \log_2 n$  bits para seleccionar cada registro
  - ▶  $k$  bits de selección  $\rightarrow 2^k$  registros
- ▶ Elemento fundamental de almacenamiento.
  - ▶ Acceso muy rápido.

## Banco de registros

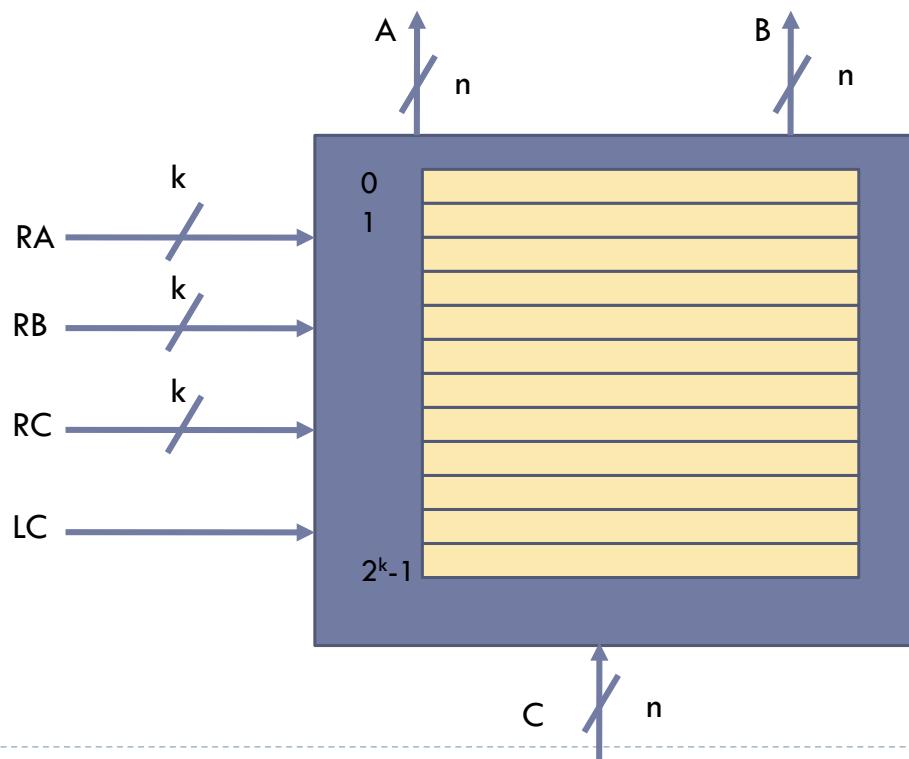


## Banco de registros

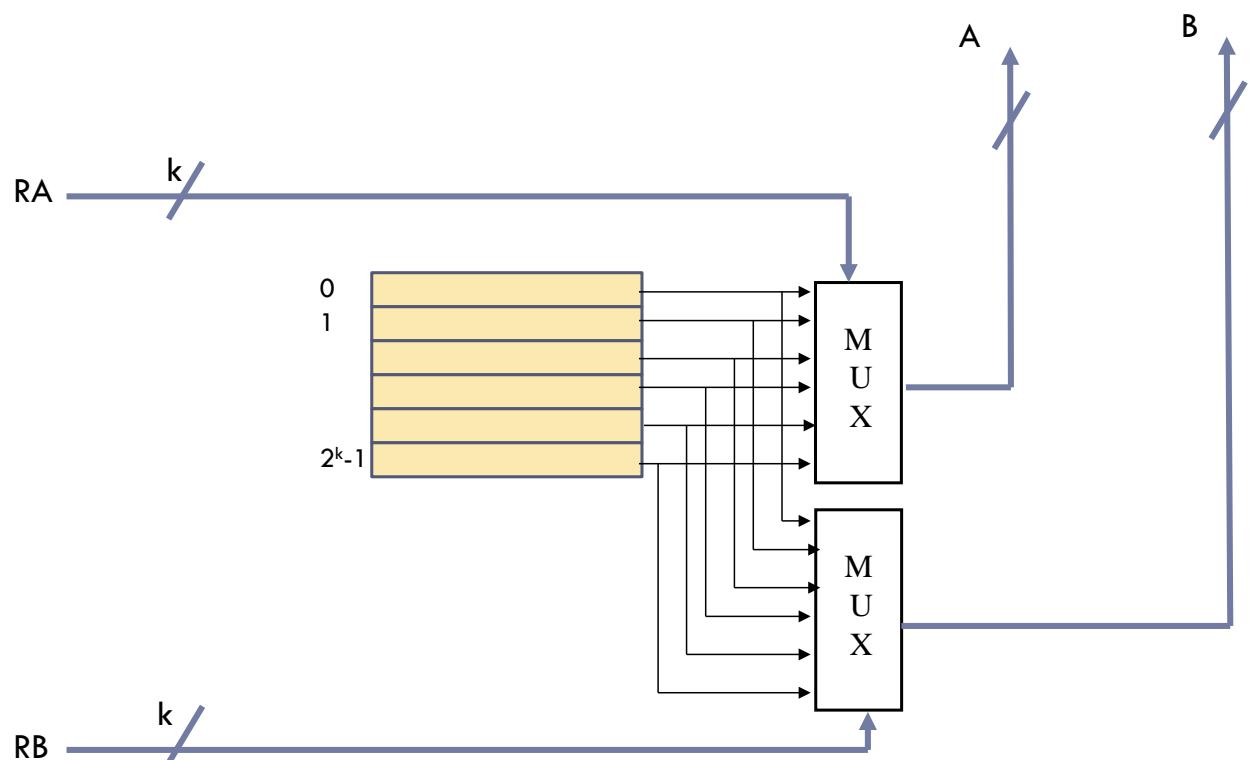


## Banco de registros

¿Qué valor tiene que tener RA para sacar por A el contenido del registro 14?

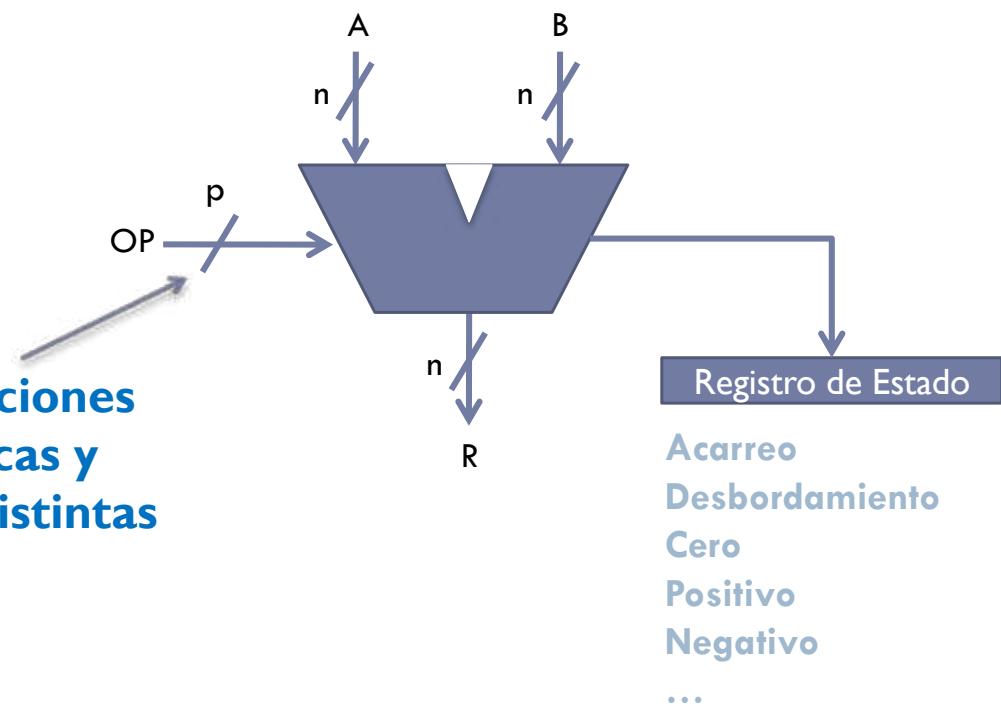


## Esquema para lectura

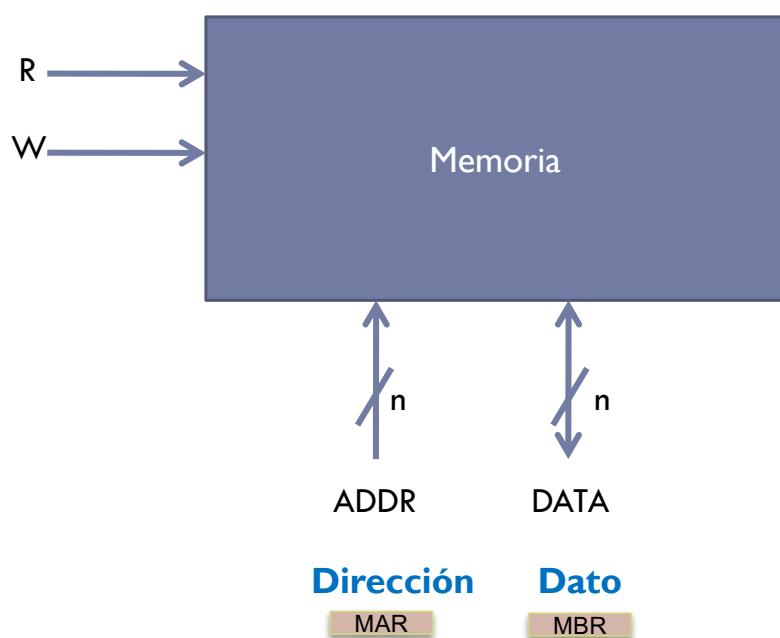


# Unidad aritmético lógica

2<sup>p</sup> operaciones  
aritméticas y  
lógicas distintas



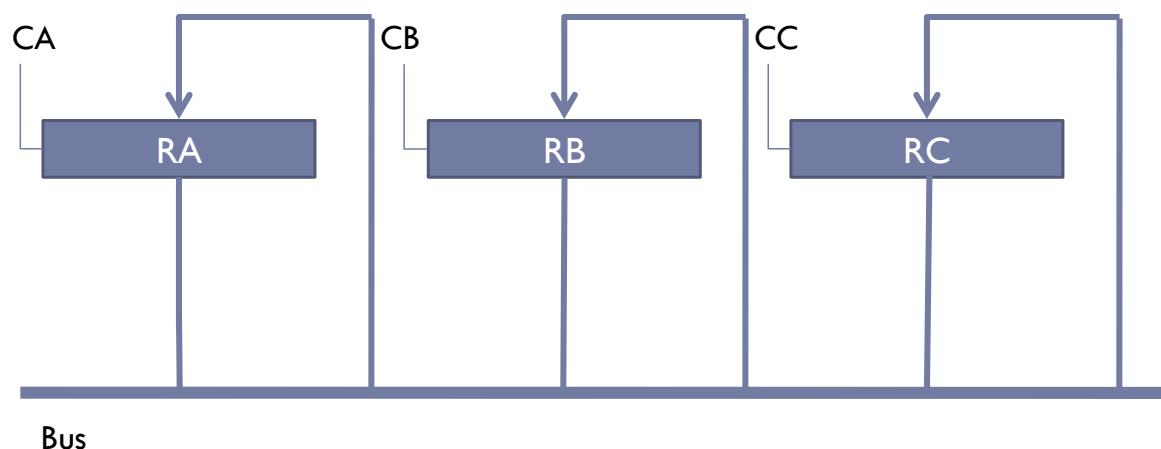
## Acceso a la memoria



## Unidad de control

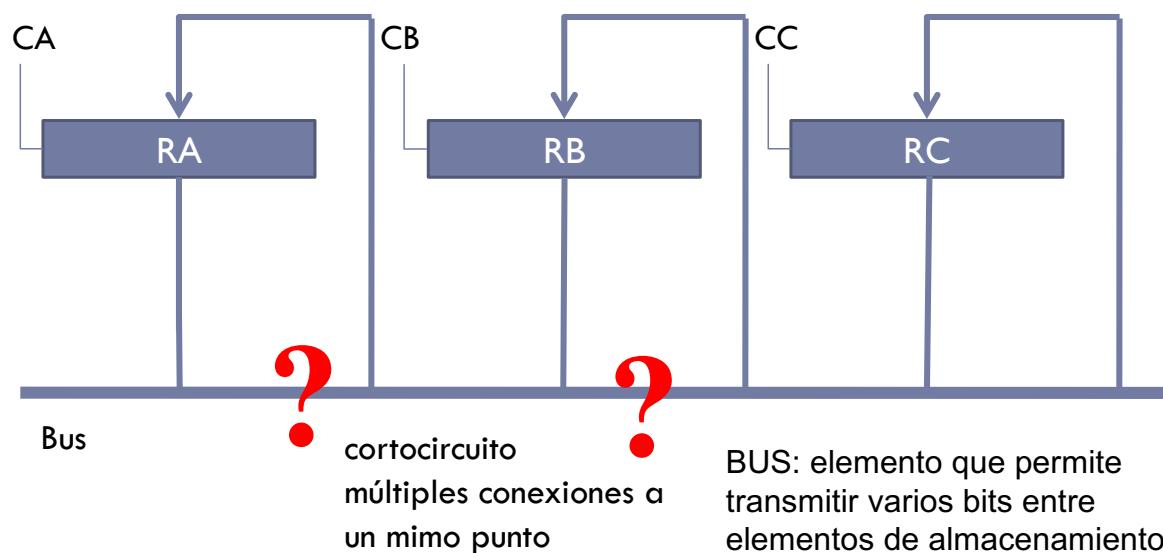


## Conexión de registros a un bus



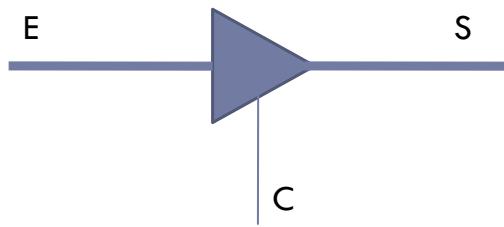
BUS: elemento que permite  
Transmitir varios bits entre  
Elementos de almacenamiento

## Conexión de registros a un bus



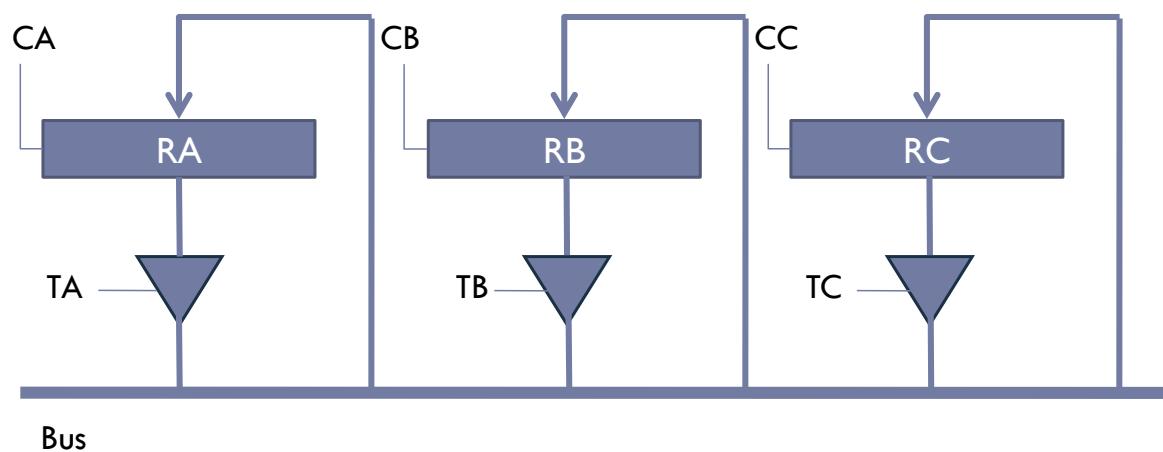
## Búfer triestado

- ▶ Tipo especial de puerta lógica que puede poner su salida en alta impedancia ( $Z$ )
- ▶ Útil para permitir múltiples conexiones a un mismo punto

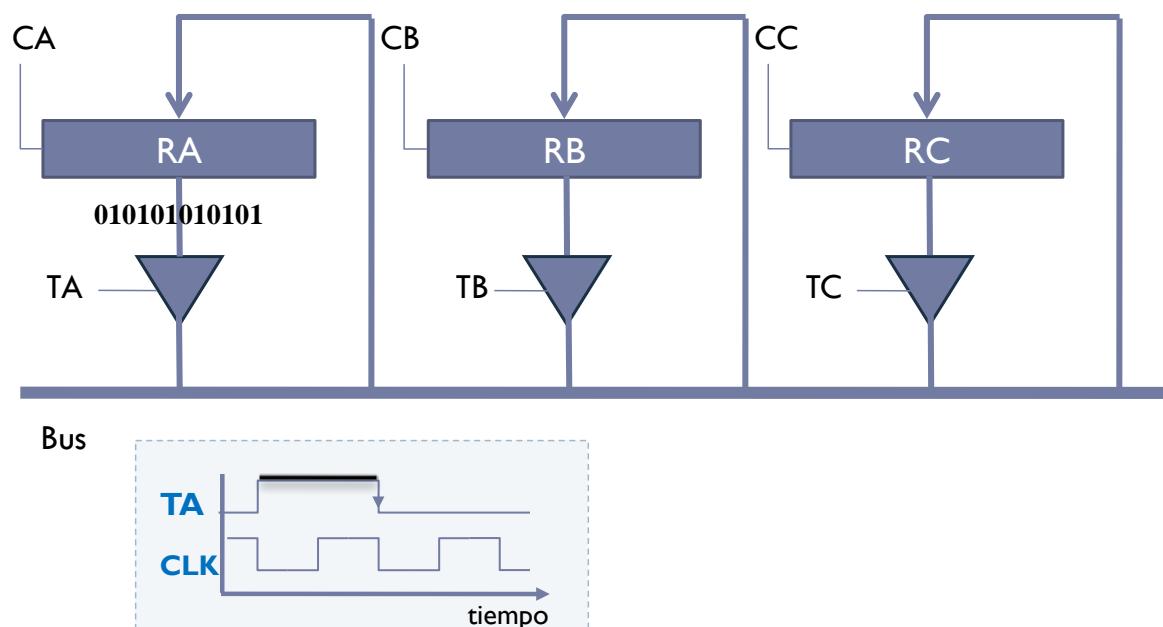


E	C	S
0	0	Z
1	0	Z
0	1	0
1	1	1

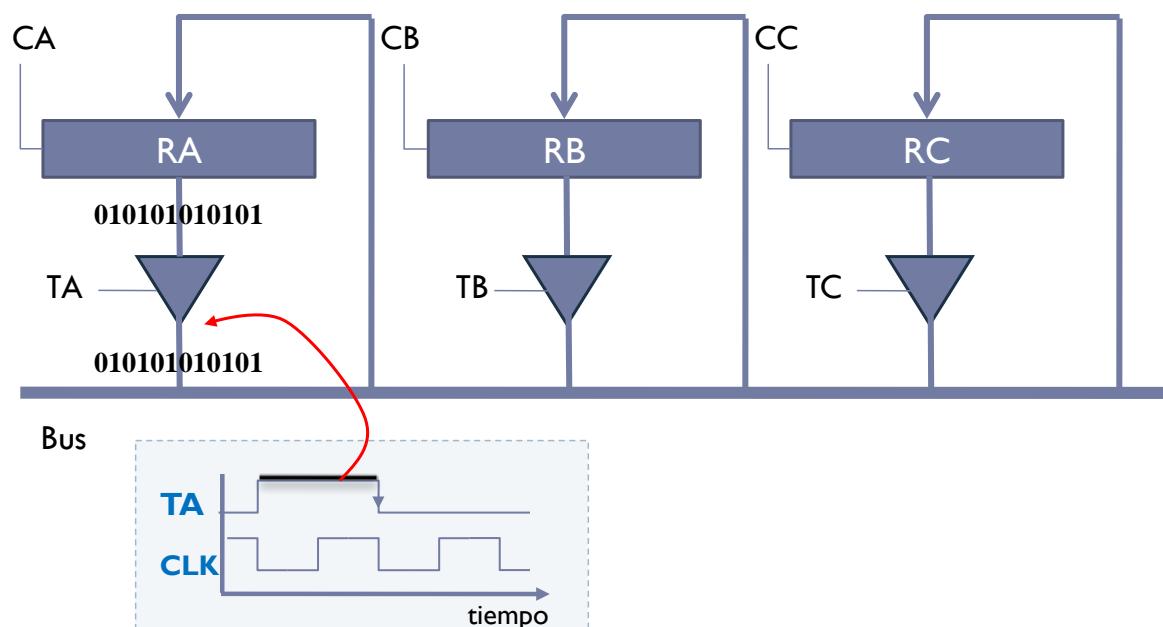
## Acceso a un bus



## Acceso a un bus

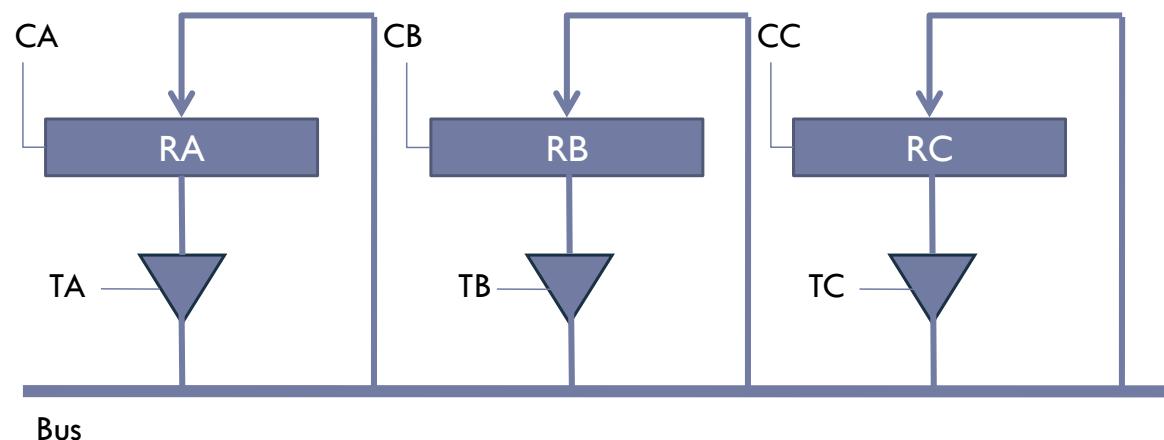


## Acceso a un bus

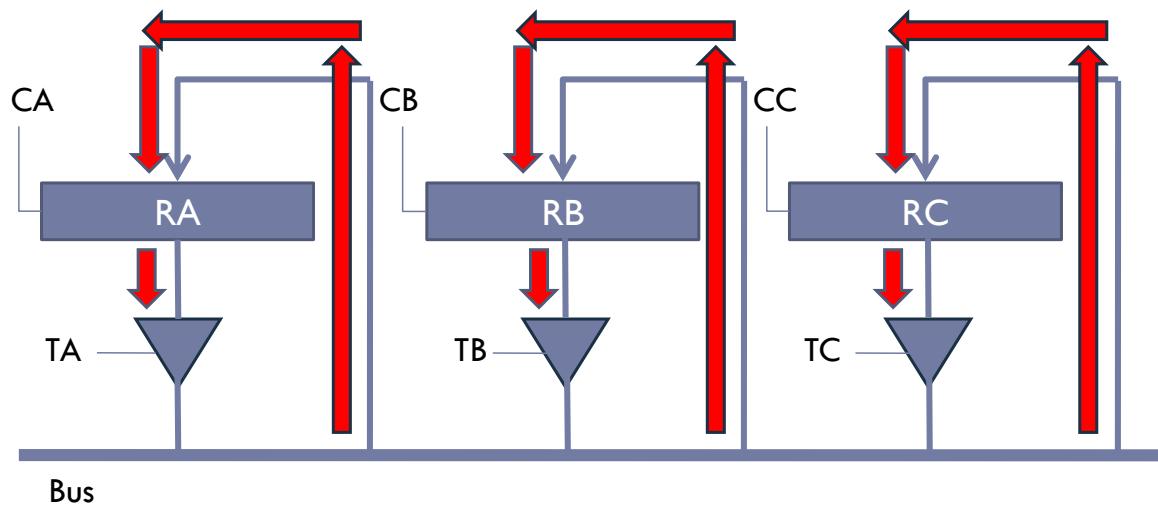


## Ejemplo

- ▶ ¿Qué señales de control hay que activar para cargar el contenido de RA en RB?



## Camino de datos ( $RB \leftarrow RA$ )

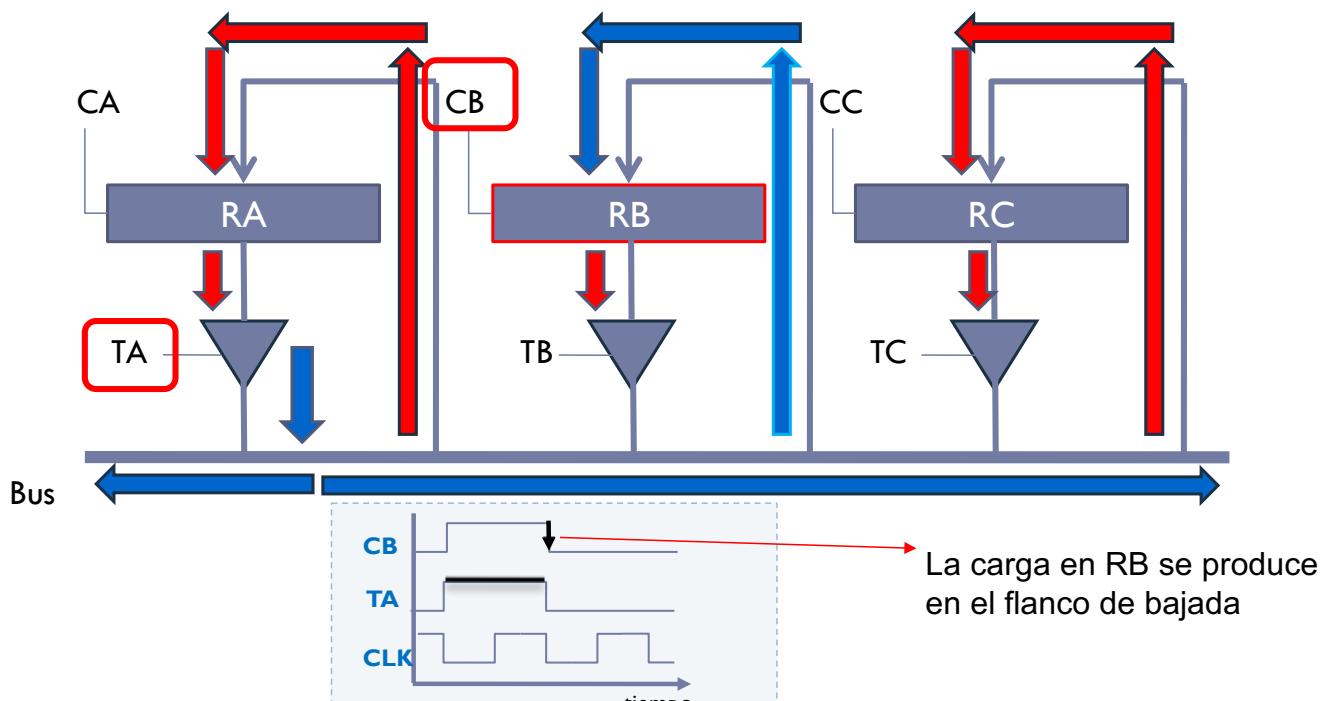


**Situación con todas las señales desactivadas**

## Camino de datos ( $RB \leftarrow RA$ )

### IMPORTANTE

No se puede activar dos o más triestados al mismo bus y al mismo tiempo



## Lenguaje nivel RT

- ▶ Lenguaje de nivel de transferencia de registros.
  - ▶ Registro1  $\leftarrow$  Registro2
- ▶ Especifica lo que ocurre en el computador mediante transferencias de datos entre registros.

# Operaciones elementales

## ▶ Operaciones de transferencia

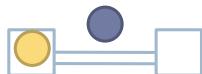
- ▶  $\text{MAR} \leftarrow \text{PC}$



Reg  $\leftarrow$  Reg

## ▶ Operaciones de proceso

- ▶  $\text{RI} \leftarrow \text{R2} + \text{RT2}$

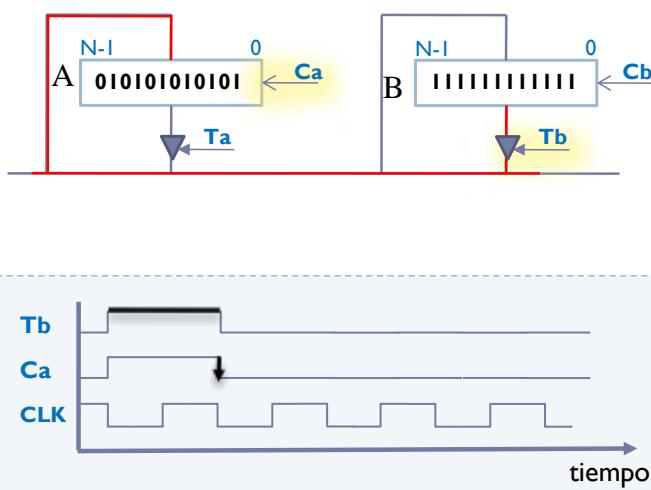


Reg  $\leftarrow \varphi(\text{Reg}, \text{Reg})$

## ▶ Lenguaje RT

- ▶ Lenguaje de nivel de transferencia de registros.
- ▶ Especifica lo que ocurre en el computador mediante transferencias de datos entre registros.

# Ejemplo de operación elemental de transferencia



## ▶ Operación elemental de transferencia:

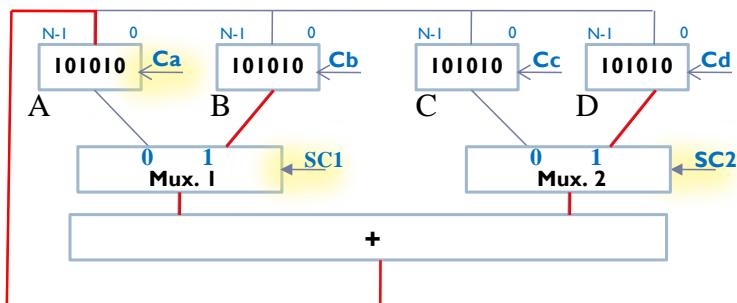
- ▶ Elemento de almacenamiento origen
- ▶ Elemento de almacenamiento destino
- ▶ Se establece un camino

xx:  $A \leftarrow B$  [ $T_b, Ca$ ]

## ▶ IMPORTANTE

- ▶ Establecer el camino entre origen y destino en un mismo ciclo
- ▶ En un mismo ciclo NO:
  - ▶ se puede atravesar un registro

# Ejemplo de operación elemental de procesamiento



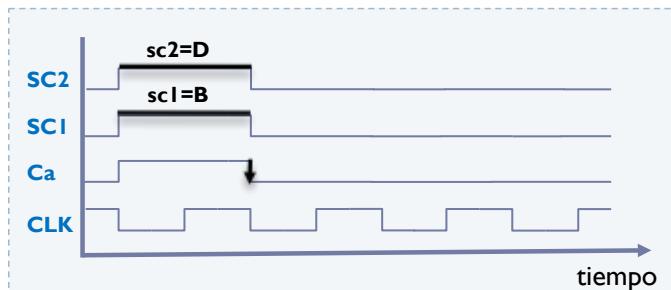
## ▶ Operación elemental de procesamiento:

- ▶ Elemento(s) de origen
- ▶ Elemento destino
- ▶ Operación de transformación en el camino

yy:  $A \leftarrow B + D$  [ $SC_1 = b, SC_2 = d, Ca$ ]

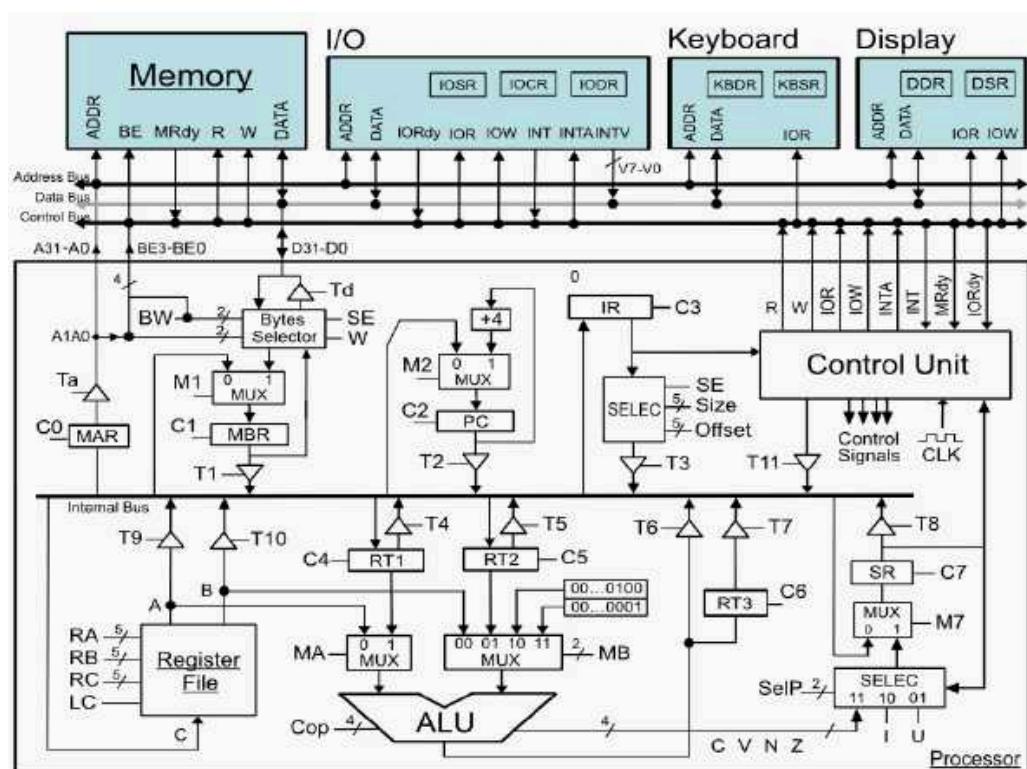
## ▶ IMPORTANTE

- ▶ Establecer el camino entre origen y destino en un mismo ciclo
- ▶ En un mismo ciclo NO se puede atravesar un registro



# Estructura de un computador elemental

## Simulador WepSIM

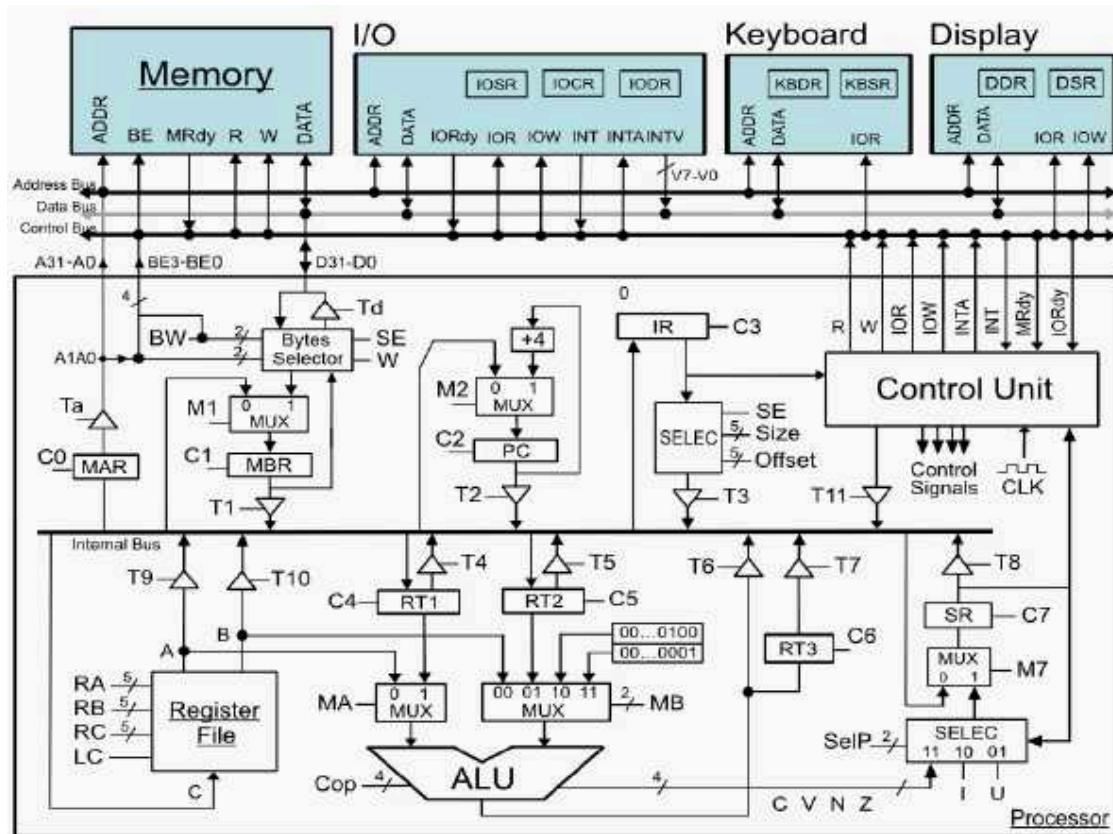


<https://wepsim.github.io/wepsim/>

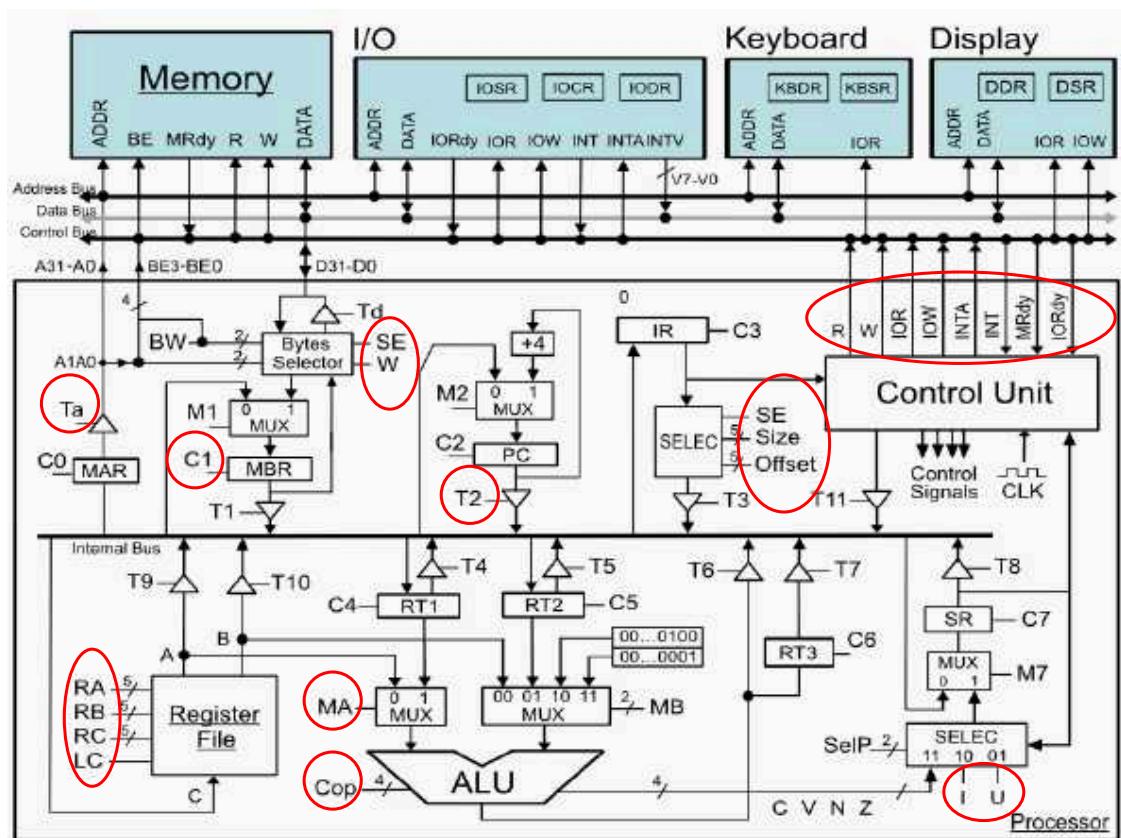
# Características

- ▶ **Computador de 32 bits**
- ▶ **La memoria se direcciona por bytes**
  - ▶ Un ciclo para las operaciones de lectura y escritura
- ▶ **Banco de 32 registros visibles**
  - ▶ R0..R31
  - ▶ Asumir como en el MIPS:
    - ▶ R0 = 0 y SP = R29
- ▶ **Registros no visibles al usuario**
  - ▶ RT1, RT2, RT3: no visibles
- ▶ **Otros registros de control y estado**
  - ▶ MAR, MBR, PC, SR, IR
- ▶ **Simulador WepSIM**
  - ▶ <https://wepsim.github.io/wepsim/>

# Estructura de un computador elemental



## Señales de control



# Señales de control

- ▶ Señales de acceso a memoria
- ▶ Señales de carga en registros
- ▶ Señales de control de las puertas triestado
- ▶ Señales de selección de los MUX
- ▶ Señales de control del banco de registros (RA, RB, RC y LC)
- ▶ Otras señales de selección

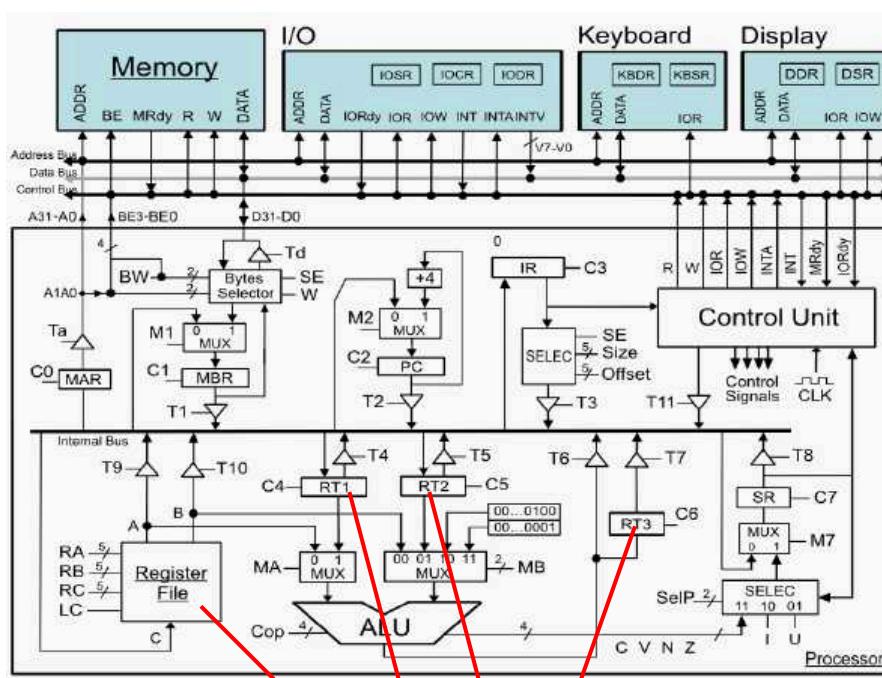
## Nomenclatura:

- Ry: Selección de registros del banco de registros
- Mx: Selección en multiplexor
- Tx: Señal de activación triestado
- Cx: Señal de carga de registro

# Registros

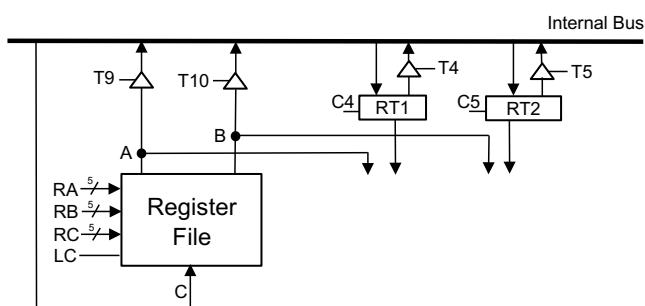
- ▶ **Registros del banco de registros**
- ▶ **Registros de control y estado:**
  - ▶ PC: contador de programa
  - ▶ IR: registro de instrucción
  - ▶ SP: puntero de pila (en el banco de registros)
  - ▶ MAR: registro de direcciones de memoria
  - ▶ MBR: registro de datos de memoria
  - ▶ SR: registro de estado
- ▶ **Registros no visibles al usuario: RT1, RT2, RT3.**

# Estructura de un computador elemental



Banco de registros y  
registros auxiliares (RT1, RT2 y RT3)

# Señales de control



## Nomenclatura:

- Ry -> Identificador de registro para el punto y
- Mx -> Selección en multiplexor
- Tx -> Señal de activación triestado
- Cx -> Señal de carga de registro

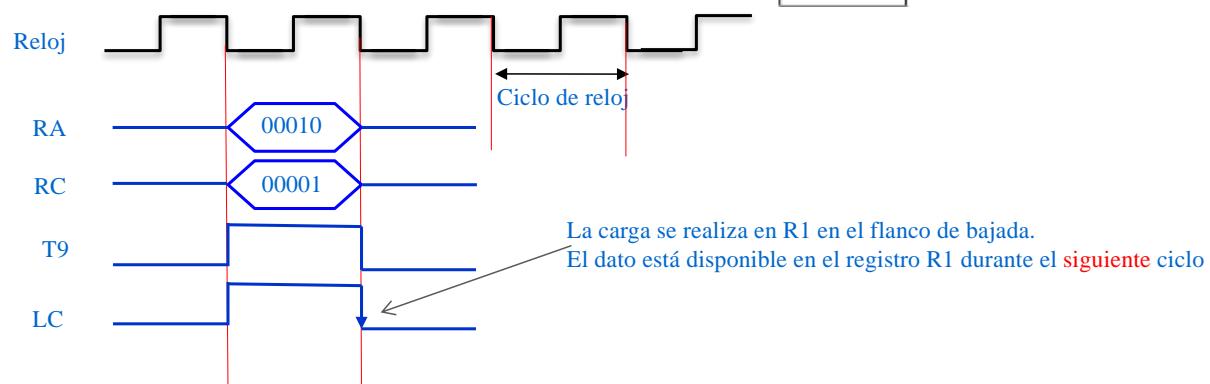
## ▶ Banco de registros y registros RT1 y RT2

- ▶ RA – salida de registro RA a A
- ▶ RB – salida de registro RB a B
- ▶ RC – salida de registro RC a E
- ▶ LC – activa la escritura para RC
- ▶ T9 – copia de A al bus interno
- ▶ T10 – copia de B al bus interno
- ▶ C4 – del bus interno al RT1
- ▶ T4 – salida de RT1 al bus interno
- ▶ C5 – del bus interno al RT2
- ▶ T5 – salida de RT2 al bus interno

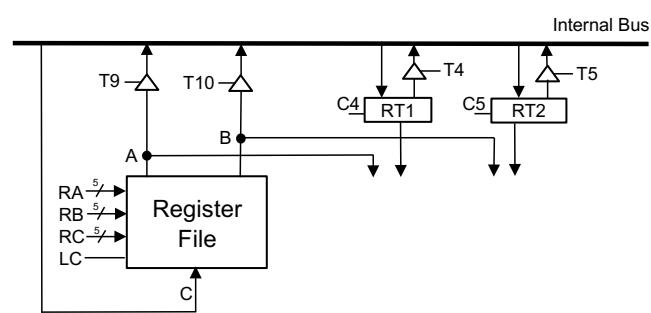
# Uso del banco de registros

- ▶ Señales de control a activar en un ciclo para la operación elemental  $R1 \leftarrow R2$ , donde R1 es el registro 1 y R2 el registro 2 del Banco de registros:

- ▶ RA = 00010
- ▶ RC = 00001
- ▶ T9 y LC
- ▶ El resto a 0

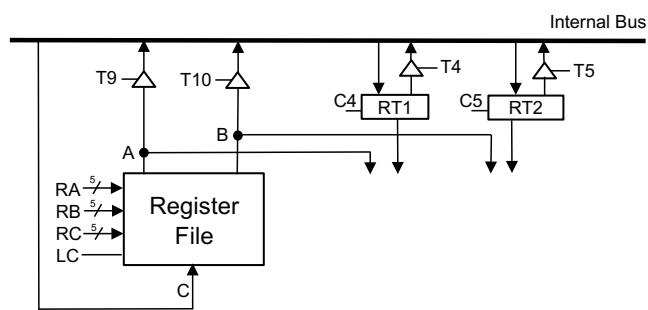


## Ejemplo operaciones elementales en registros



▶ **SWAP R1 R2**

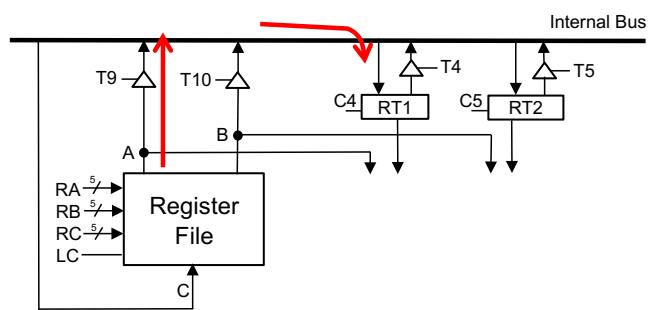
## Ejemplo operaciones elementales en registros



### ▶ SWAP RI R2

O. Elemental	Señales

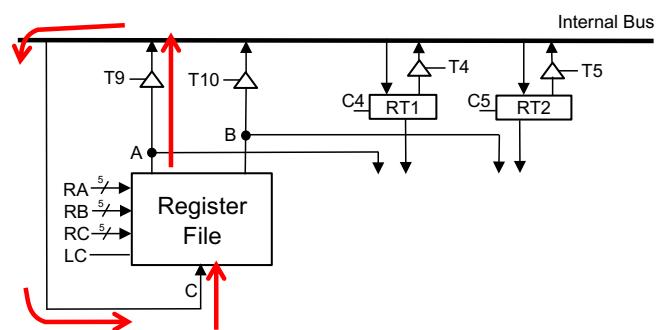
# Ejemplo operaciones elementales en registros



## ▶ SWAP RI R2

O. Elemental	Señales
RT1 ← R1	RA=00001, T9, C4

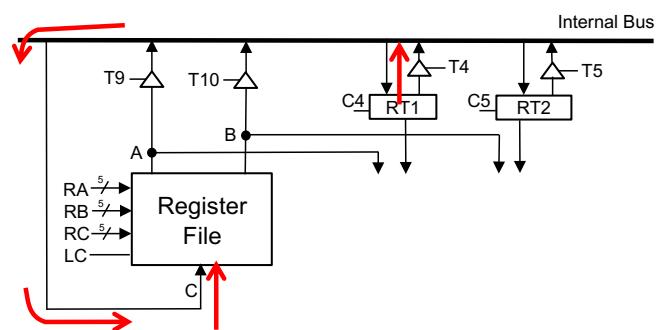
# Ejemplo operaciones elementales en registros



## ▶ SWAP R1 R2

O. Elemental	Señales
RT1 ← R1	RA=00001, T9, C4
R1 ← R2	RA=2 (00010), T9, RC=1, LC

## Ejemplo operaciones elementales en registros

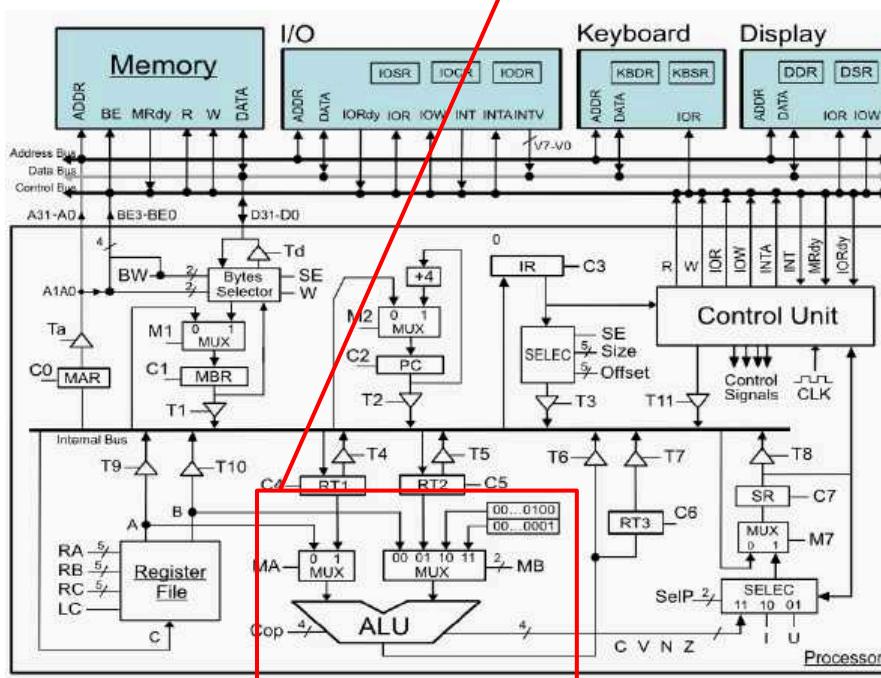


### ► SWAP R1 R2

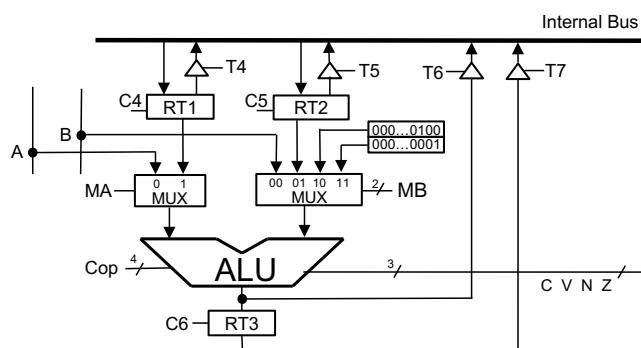
O. Elemental	Señales
RT1 ← R1	RA=1, T9, C4
R1 ← R2	RA=2 (00010), T9, RC=1, LC
R2 ← RT1	T4, RC=2 (00010), LC

# Estructura de un computador elemental

Unidad Aritmetico-Lógica (ALU)



# Señales de control

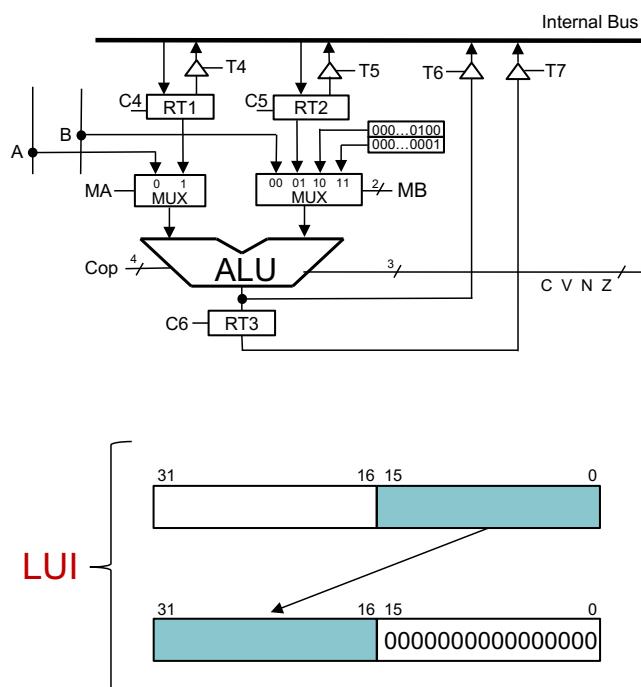


## ▶ ALU

- ▶ MA – selección de operando A
- ▶ MB – selección de operando B
- ▶ Cop – código de operación

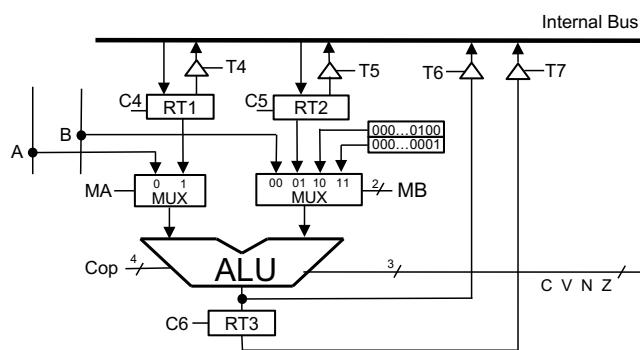
Cop (Cop <sub>3</sub> -Cop <sub>0</sub> )	Operación
0000	NOP
0001	A <b>and</b> B
0010	A <b>or</b> B
0011	<b>not</b> (A)
0100	A <b>xor</b> B
0101	<b>Shift Right Logical</b> (A) B= number of bits to shift
0110	<b>Shift Right Arithmetic</b> (A) B= number of bits to shift
0111	<b>Shift left</b> (A) B= number of bits to shift
1000	<b>Rotate Right</b> (A) B= number of bits to rotate
1001	<b>Rotate Left</b> (A) B= number of bits to rotate
1010	A <b>+ B</b>
1011	A <b>- B</b>
1100	A <b>* B</b> (with overflow)
1101	A <b>/ B</b> (integer division)
1110	A <b>% B</b> (integer division)
1111	<b>LUI</b> (A)

# Señales de control



Cop (Cop <sub>3</sub> -Cop <sub>0</sub> )	Operación
0000	NOP
0001	A <b>and</b> B
0010	A <b>or</b> B
0011	<b>not</b> (A)
0100	A <b>xor</b> B
0101	<b>Shift Right Logical</b> (A) B= number of bits to shift
0110	<b>Shift Right Arithmetic</b> (A) B= number of bits to shift
0111	<b>Shift left</b> (A) B= number of bits to shift
1000	<b>Rotate Right</b> (A) B= number of bits to rotate
1001	<b>Rotate Left</b> (A) B= number of bits to rotate
1010	A <b>+ B</b>
1011	A <b>- B</b>
1100	A <b>* B</b> (with overflow)
1101	A <b>/ B</b> (integer division)
1110	A <b>% B</b> (integer division)
1111	<b>LUI</b> (A)

# Señales de control



Resultado	C	V	N	Z
Resultado positivo (0 se considera +)	0	0	0	0
Resultado == 0	0	0	0	1
Resultado negativo	0	0	1	0
Desbordamiento de la operación	0	1	0	0
División por cero	0	1	0	1
Acarreo en el bit 32	1	0	0	0

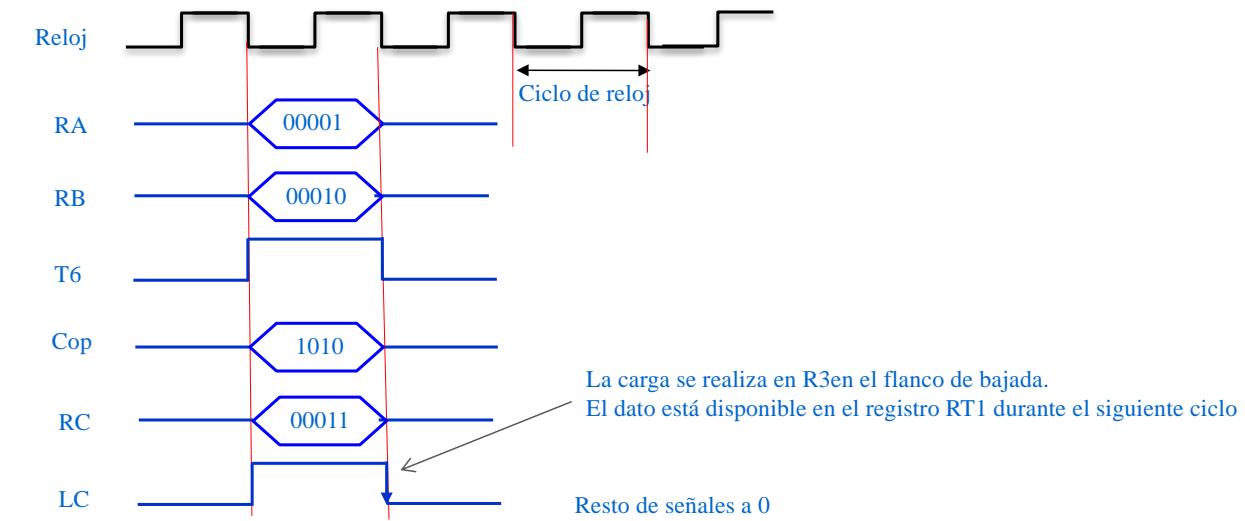
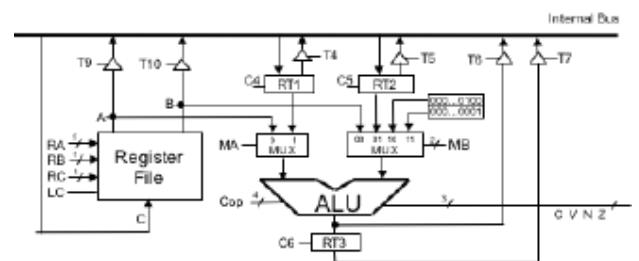
Cop (Cop <sub>3</sub> -Cop <sub>0</sub> )	Operación
0000	NOP
0001	A and B
0010	A or B
0011	not (A)
0100	A xor B
0101	Shift Right Logical (A) B= number of bits to shift
0110	Shift Right Arithmetic( A) B= number of bits to shift
0111	Shift left (A) B= number of bits to shift
1000	Rotate Right (A) B= number of bits to rotate
1001	Rotate Left (A) B= number of bits to rotate
1010	A + B
1011	A - B
1100	A * B (with overflow)
1101	A / B (integer division)
1110	A % B (integer division)
1111	LUI (A)

# Ejemplo

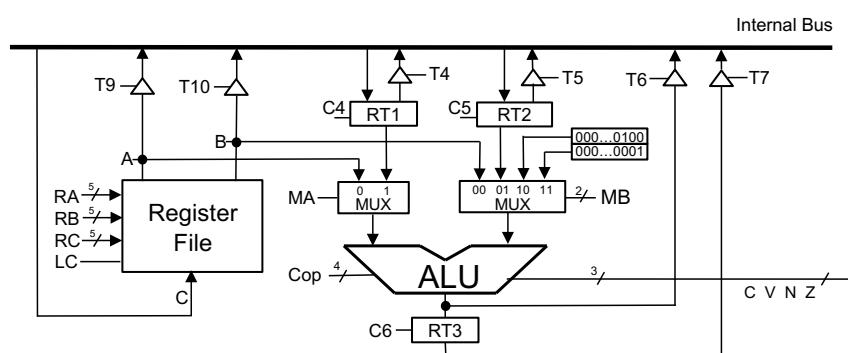
## operaciones elementales en ALU

► ADD R3 RI R2

O. Elemental	Señales
R3 $\leftarrow$ R1 + R2	RA=R1, RB=R2, Cop=+, T6, RC=R3, LC=1



# Ejemplo operaciones elementales en ALU

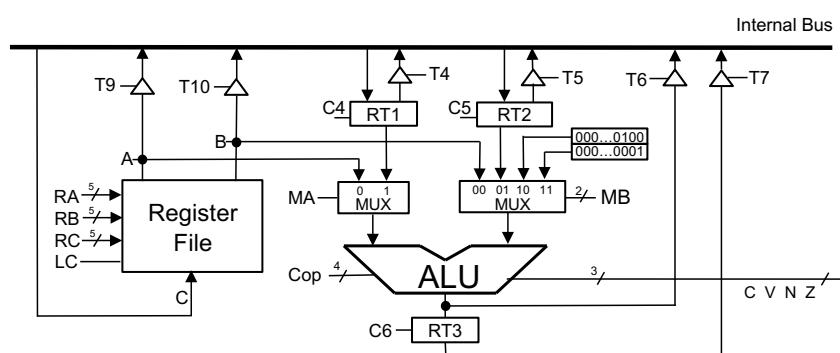


▶ **SWAP RI R2**

▶ **SWAP RI, R2 sin R<sub>tmp</sub>**

O. Elemental	Señales
RT1 ← R1	RA=1, T9, C4
R1 ← R2	RA=2, T9, RC=1, LC
R2 ← RT1	T4, RC=2, LC

# Ejemplo operaciones elementales en ALU



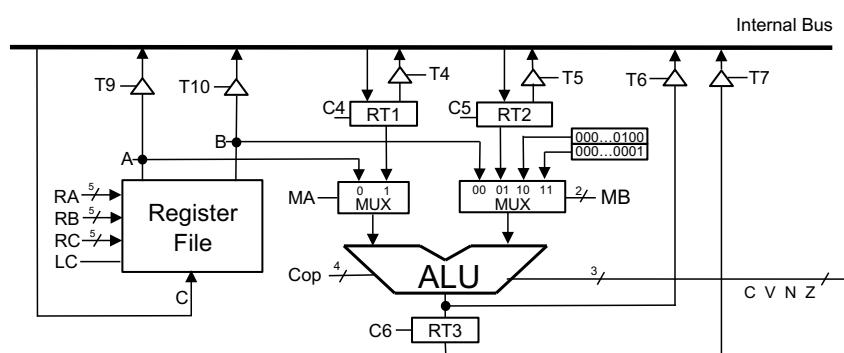
▶ SWAP RI R2

O. Elemental	Señales
RT1 ← R1	RA=1, T9, C4
R1 ← R2	RA=2, T9, RC=1, LC
R2 ← RT1	T4, RC=2, LC

▶ SWAP RI, R2 sin R<sub>tmp</sub>

O. Elemental
R1 ← R1 ^ R2
R2 ← (R1 ^ R2) ^ R2
R1 ← (R1 ^ R2) ^ R1

# Ejemplo operaciones elementales en ALU



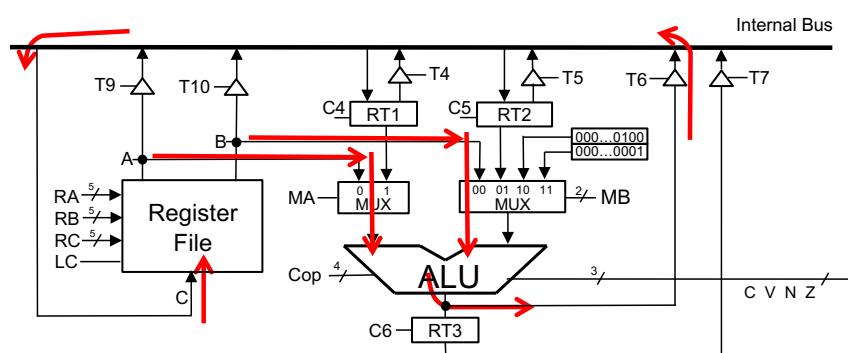
▶ SWAP RI R2

O. Elemental	Señales
RT1 ← R1	RA=1, T9, C4
R1 ← R2	RA=2, T9, RC=1, LC
R2 ← RT1	T4, RC=2, LC

▶ SWAP RI, R2 sin R<sub>tmp</sub>

O. Elemental	Señales
R1 ← R1 ^ R2	RA=1, RB=2, Cop=^, T6, RC=1
R2 ← R1 ^ R2	RA=1, RB=2, Cop=^, T6, RC=2
R1 ← R1 ^ R2	RA=1, RB=2, Cop=^, T6, RC=1

# Ejemplo operaciones elementales en ALU



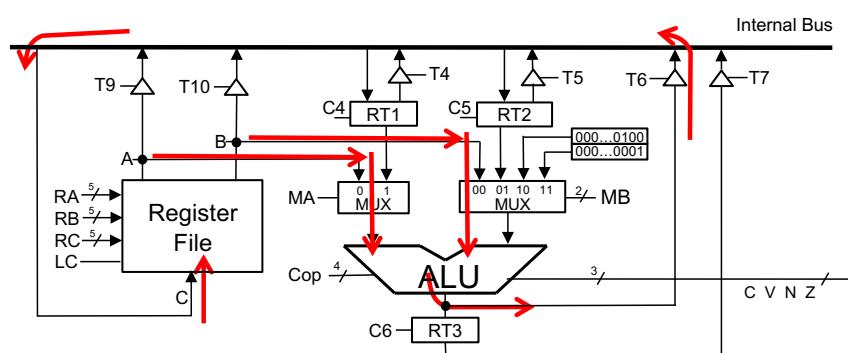
▶ SWAP RI R2

O. Elemental	Señales
RT1 ← R1	RA=1, T9, C4
R1 ← R2	RA=2, T9, RC=1, LC
R2 ← RT1	T4, RC=2, LC

▶ SWAP RI, R2 sin R<sub>tmp</sub>

O. Elemental	Señales
R1 ← R1 ^ R2	RA=1, RB=2, Cop=^, T6, RC=1
R2 ← R1 ^ R2	RA=1, RB=2, Cop=^, T6, RC=2
R1 ← R1 ^ R2	RA=1, RB=2, Cop=^, T6, RC=1

## Ejemplo operaciones elementales en ALU



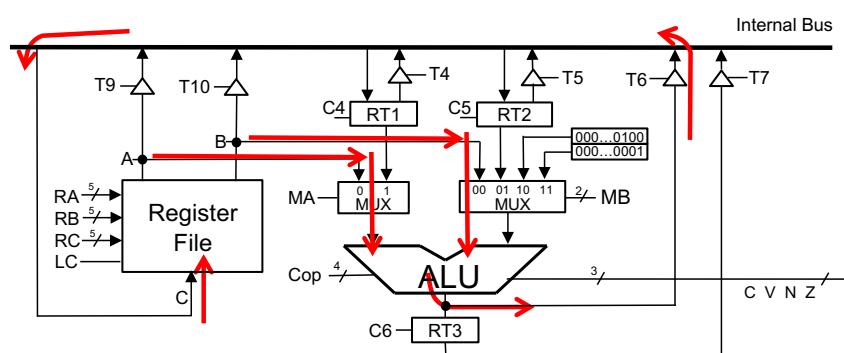
▶ SWAP RI R2

O. Elemental	Señales
RT1 $\leftarrow$ R1	RA=1, T9, C4
R1 $\leftarrow$ R2	RA=2, T9, RC=1, LC
R2 $\leftarrow$ RT1	T4, RC=2, LC

▶ SWAP RI, R2 sin R<sub>tmp</sub>

O. Elemental	Señales
R1 $\leftarrow$ R1 ^ R2	RA=1, RB=2, Cop= $\wedge$ , T6, RC=1
R2 $\leftarrow$ R1 ^ R2	RA=1, RB=2, Cop= $\wedge$ , T6, RC=2
R1 $\leftarrow$ R1 ^ R2	RA=1, RB=2, Cop= $\wedge$ , T6, RC=1

## Ejemplo operaciones elementales en ALU



▶ SWAP RI R2

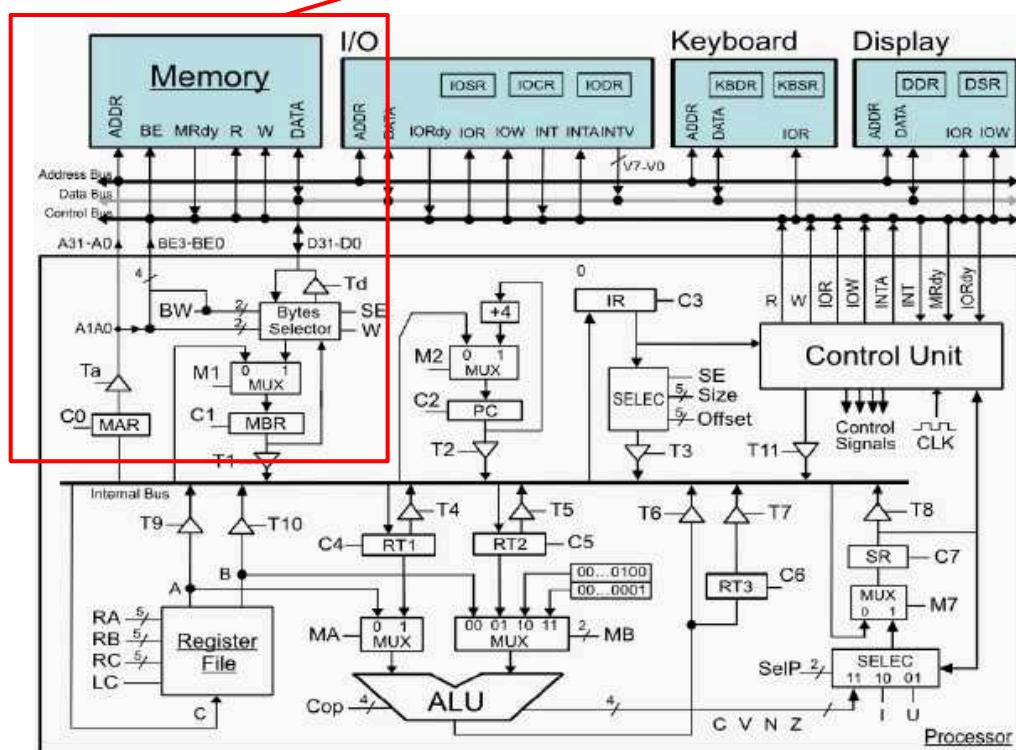
O. Elemental	Señales
RT1 $\leftarrow$ R1	RA=1, T9, C4
R1 $\leftarrow$ R2	RA=2, T9, RC=1, LC
R2 $\leftarrow$ RT1	T4, RC=2, LC

▶ SWAP RI, R2 sin R<sub>tmp</sub>

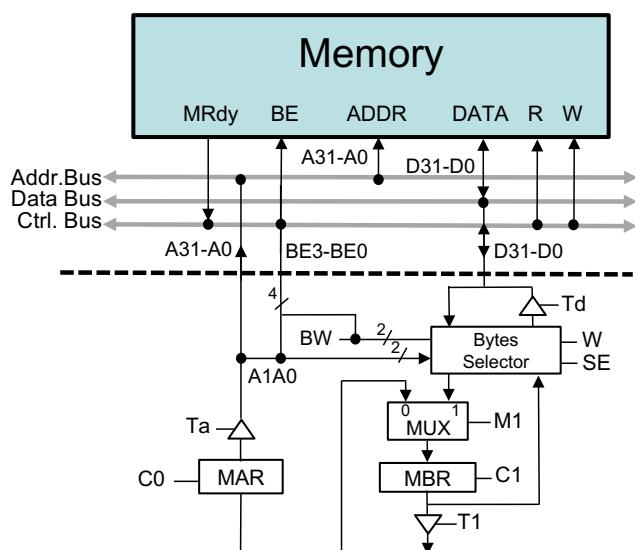
O. Elemental	Señales
R1 $\leftarrow$ R1 ^ R2	RA=1, RB=2, Cop= $\wedge$ , T6, RC=1
R2 $\leftarrow$ R1 ^ R2	RA=1, RB=2, Cop= $\wedge$ , T6, RC=2
R1 $\leftarrow$ R1 ^ R2	RA=1, RB=2, Cop= $\wedge$ , T6, RC=1

# Estructura de un computador elemental

Memoria principal,  
registro de direcciones y de datos



# Señales de control



## Nomenclatura:

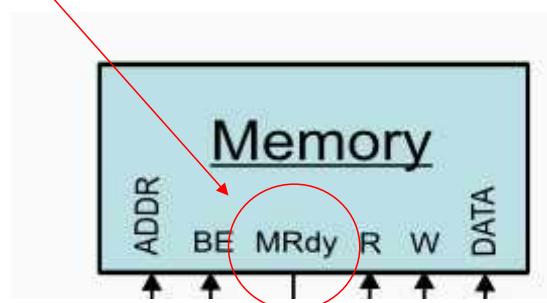
- MAR -> registro de direcciones
- MBR -> registro de datos

## ▶ Memoria principal

- ▶ R – lectura
- ▶ W – escritura
- ▶  $BE3-BE0 = A1A0 + BW$ 
  - ▶ Tamaño acceso (byte, palabra, media palabra)
- ▶ C0 – del bus interno al MAR
- ▶ C1 – del bus de datos al MBR
- ▶ Ta – salida de MAR al bus de direcciones
- ▶ Td – salida de MBR al bus de datos
- ▶ TI – salida de MBR al bus interno
- ▶ MI – selección para MBR: de memoria o bus interno

## Acceso a Memoria

- ▶ Síncrono: la memoria requiere un número determinado de ciclos
- ▶ Asíncrono: la memoria indica cuándo finaliza la operación



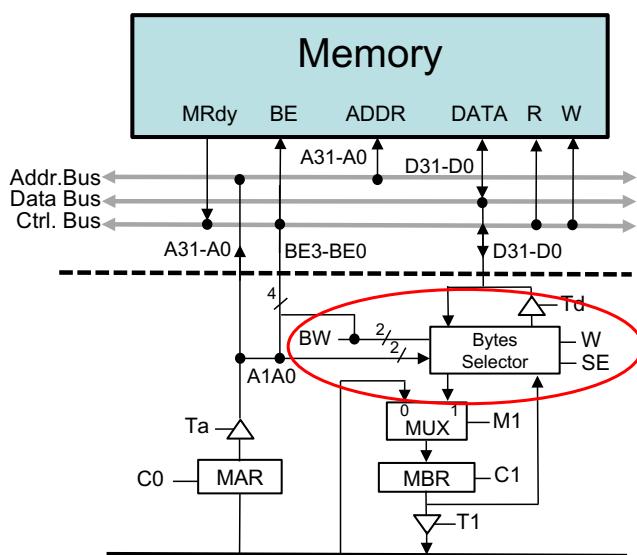
# Señales BE (Byte-Enable) para lectura

Bytes en memoria				Selección de bytes				Salida al BUS			
D31-D24	D23-D16	D15-D8	D7-D0	BE3	BE2	BE1	BE0	D31-D24	D23-D16	D15-D8	D7-D0
Byte 3	Byte 2	Byte I	Byte 0	0	0	0	0	---	---	---	Byte 0
Byte 3	Byte 2	Byte I	Byte 0	0	0	0	I	---	---	Byte I	---
Byte 3	Byte 2	Byte I	Byte 0	0	0	I	0	--	Byte 2	---	---
Byte 3	Byte 2	Byte I	Byte 0	0	0	I	I	Byte 3	---	---	---
Byte 3	Byte 2	Byte I	Byte 0	0	I	0	X	---	---	Byte I	Byte 0
Byte 3	Byte 2	Byte I	Byte 0	0	I	I	X	Byte 3	Byte 2	---	---
Byte 3	Byte 2	Byte I	Byte 0	I	I	X	X	Byte 3	Byte 2	Byte I	Byte 0

# Señales BE (Byte-Enable) para escritura

Dato en el bus				Selección de bytes				Bytes escritos en memoria			
D31-D24	D23-D16	D15-D8	D7-D0	BE3	BE2	BE1	BE0	D31-D24	D23-D16	D15-D8	D7-D0
Byte 3	Byte 2	Byte I	Byte 0	0	0	0	0	---	---	---	Byte 0
Byte 3	Byte 2	Byte I	Byte 0	0	0	0	I	---	---	Byte I	---
Byte 3	Byte 2	Byte I	Byte 0	0	0	I	0	--	Byte 2	---	---
Byte 3	Byte 2	Byte I	Byte 0	0	0	I	I	Byte 3	---	---	---
Byte 3	Byte 2	Byte I	Byte 0	0	I	0	X	---	---	Byte I	Byte 0
Byte 3	Byte 2	Byte I	Byte 0	0	I	I	X	Byte 3	Byte 2	---	---
Byte 3	Byte 2	Byte I	Byte 0	I	I	X	X	Byte 3	Byte 2	Byte I	Byte 0

# Tamaño de acceso a memoria



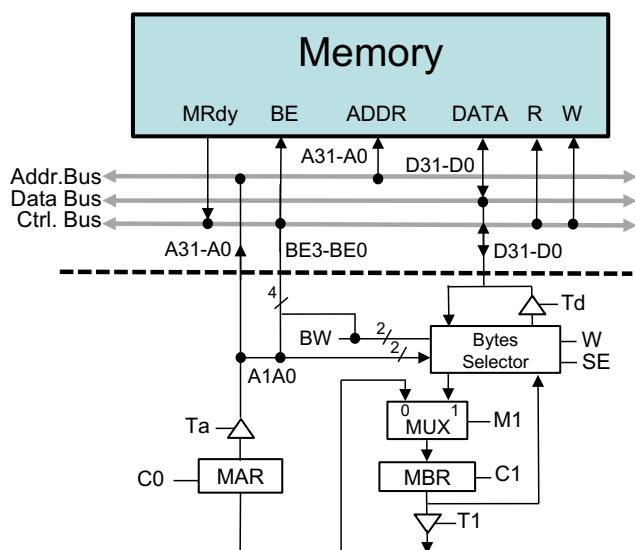
- ▶ Bytes Selector: selecciona qué bytes se almacenan en MBR en lectura y se vuelcan al bus en escritura
- ▶ Acceso a bytes: BW=0
- ▶ Acceso a media palabra: BW=01
- ▶ Acceso a palabra: BW =11
- ▶ SE: extensión de signo
  - ▶ 0: no extiende el signo en accesos más pequeños de una palabra
  - ▶ 1: extiende el signo en accesos más pequeños de una palabra

## Nomenclatura:

- MAR -> registro de direcciones
- MBR -> registro de datos

# Ejemplo operaciones elementales para usar la memoria

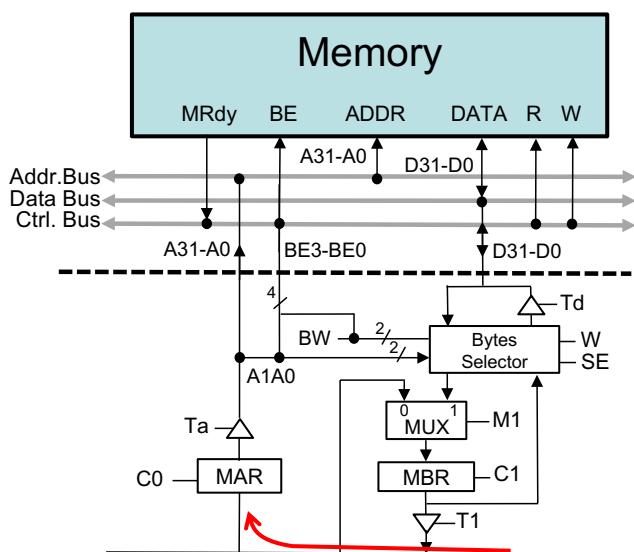
## ▶ Lectura



# Ejemplo

Acceso a memoria síncrona de 1 ciclo

## ▶ Lectura

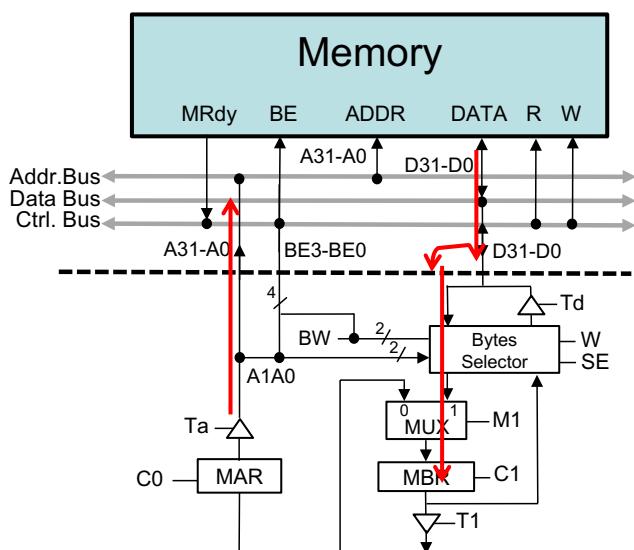


O. Elemental	Señales
MAR ← <dirección>	..., C0

# Ejemplo

Acceso a memoria síncrona de 1 ciclo

## ▶ Lectura de una palabra

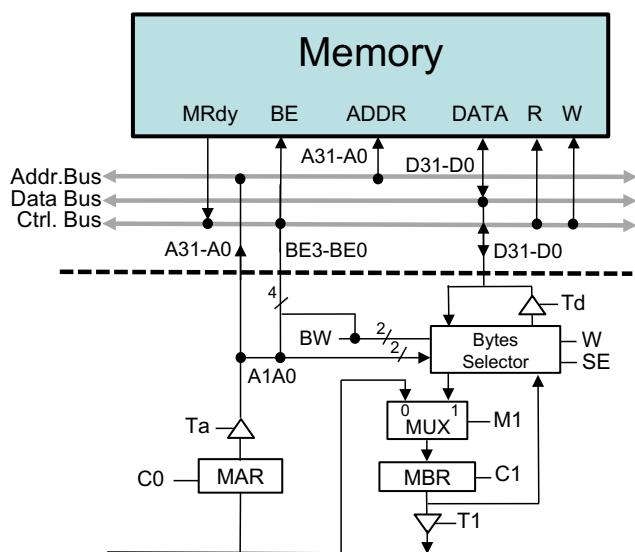


O. Elemental	Señales
MAR ← <dirección>	..., C0
<b>MBR ← MP[MAR]</b>	<b>Ta, R, M1, C1, BW=11</b>

# Ejemplo

Acceso a memoria síncrona de 1 ciclo

## ▶ Lectura de una palabra



## O. Elemental Señales

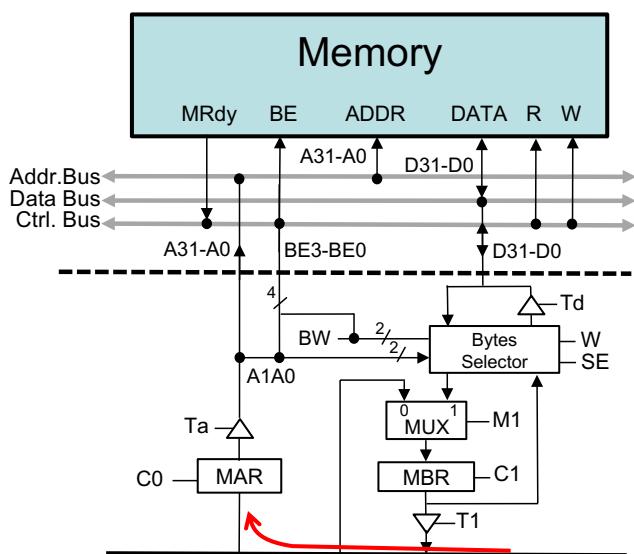
O. Elemental	Señales
MAR ← <dirección>	..., C0
MBR ← MP[MAR]	Ta, R, M1, C1, BW=11

## ▶ Escritura de una palabra

# Ejemplo

Acceso a memoria síncrona de 1 ciclo

## ▶ Lectura



O. Elemental	Señales
MAR ← <dirección>	..., C0
MBR ← MP[MAR]	Ta, R, M1, C1

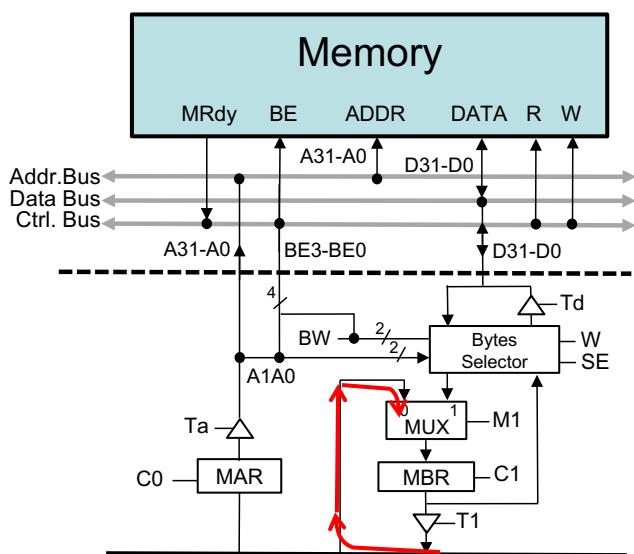
## ▶ Escritura de una palabra

O. Elemental	Señales
MAR ← <dirección>	..., C0

# Ejemplo

Acceso a memoria síncrona de 1 ciclo

## ▶ Lectura



O. Elemental	Señales
MAR ← <dirección>	..., C0
MBR ← MP[MAR]	Ta, R, M1, C1

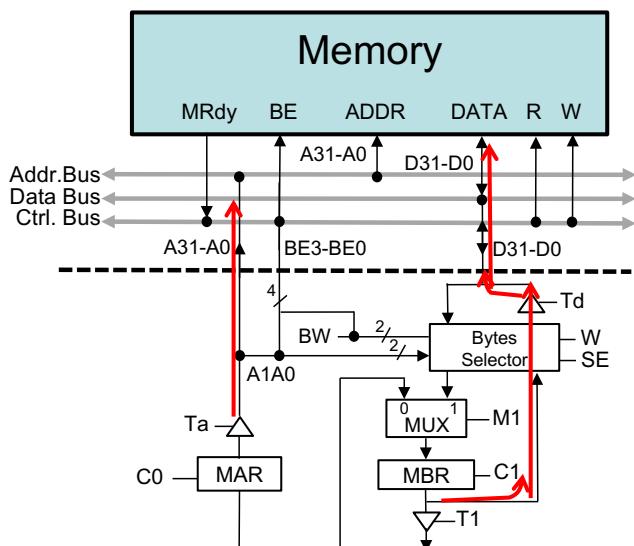
## ▶ Escritura

O. Elemental	Señales
MAR ← <dirección>	..., C0
MBR ← <dato>	..., C1

# Ejemplo

Acceso a memoria síncrona de 1 ciclo

## ▶ Lectura



### O. Elemental

MAR ← <dirección>

### Señales

..., C0

MBR ← MP[MAR]

Ta, R, M1, C1

## ▶ Escritura

### O. Elemental

MAR ← <dirección>

### Señales

..., C0

MBR ← <dato>

..., C1

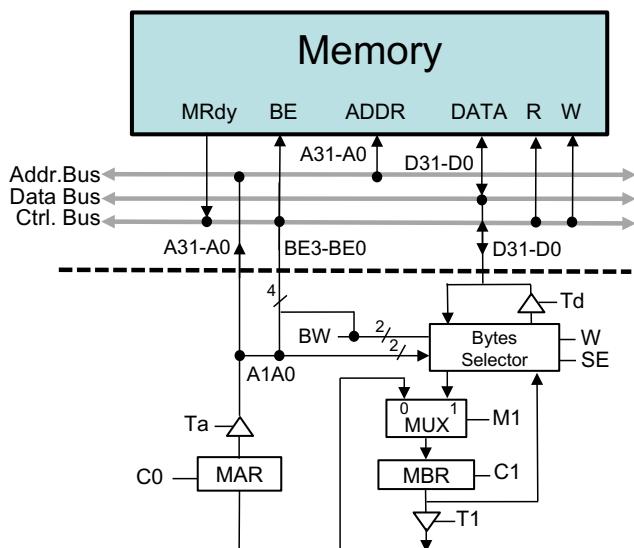
Ciclo de escritura

Ta, Td, W,  
BW=11

# Ejemplo

Acceso a memoria síncrona de 1 ciclo

## ▶ Lectura



O. Elemental	Señales
MAR ← <dirección>	..., C0
MBR ← MP[MAR]	Ta, R, M1, C1

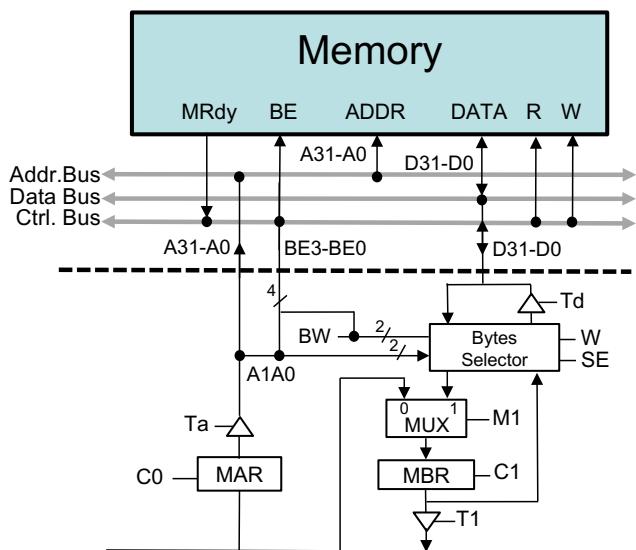
## ▶ Escritura

O. Elemental	Señales
MAR ← <dirección>	..., C0
MBR ← <dato>	..., C1
Ciclo de escritura	Ta, Td, W, BW=11

# Ejemplo

Acceso a memoria síncrona de 2 ciclo

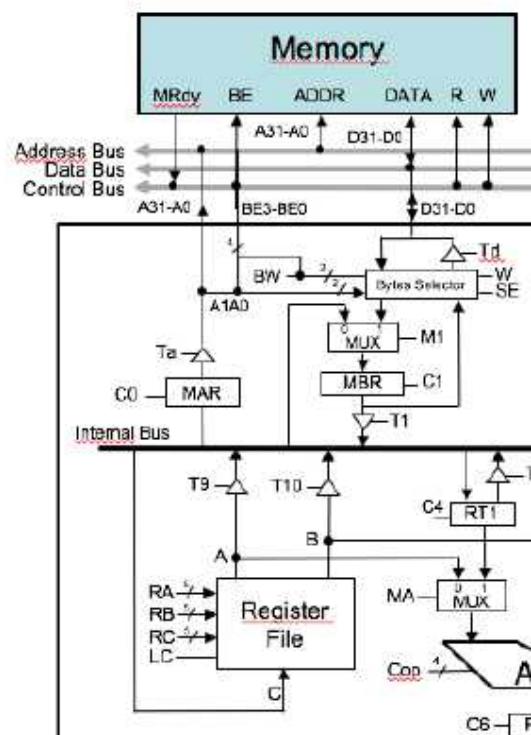
## ▶ Lectura de una palabra



O. Elemental	Señales
MAR ← <dirección>	..., C0
ciclo de lectura	Ta, R,
MBR ← MP[MAR]	Ta, R, M1, C1, BW=11

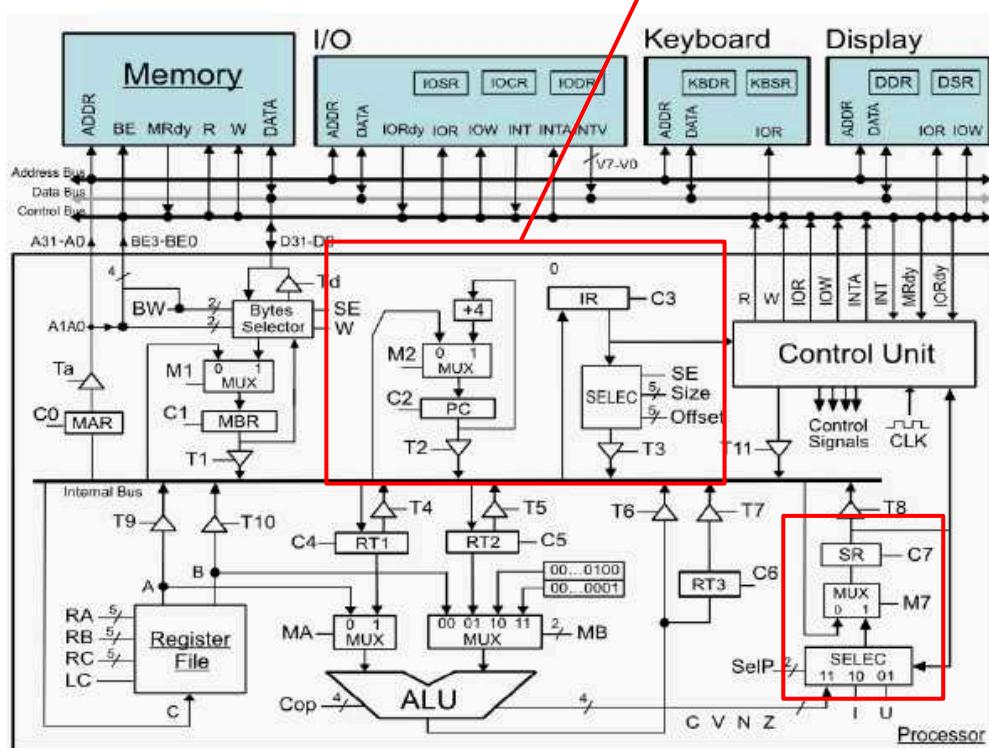
# Ejemplo de acceso a memoria

- ▶ Señales de control a activar para realizar la operación de lectura (una palabra)
   
**R2 ← Memoria[RI]:**
  - ▶ Primer ciclo: MAR ← RI
    - ▶ RA = 00001, T9, C0
  - ▶ Segundo ciclo: MBR ← Memoria
    - ▶ Ta, R, CI, MI, BW=11
  - ▶ Tercer ciclo: R2 ← MBR
    - ▶ TI, RC= 00010, LC



# Estructura de un computador elemental

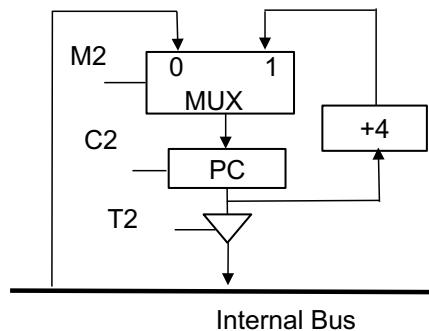
## Registros SR, PC y IR



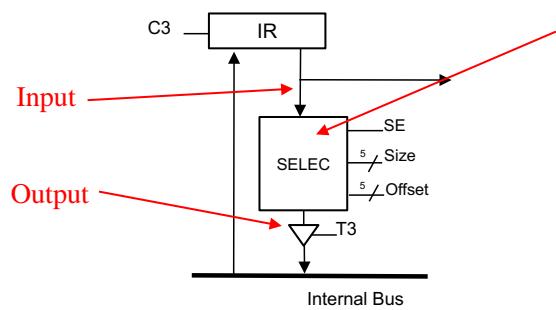
# Contador de programa

## ▶ Contador de programa PC:

- ▶ C2, M2
  - ▶  $PC \leftarrow PC + 4$
- ▶ C2 – del bus interno al PC
- ▶ T2 – de PC a bus interno

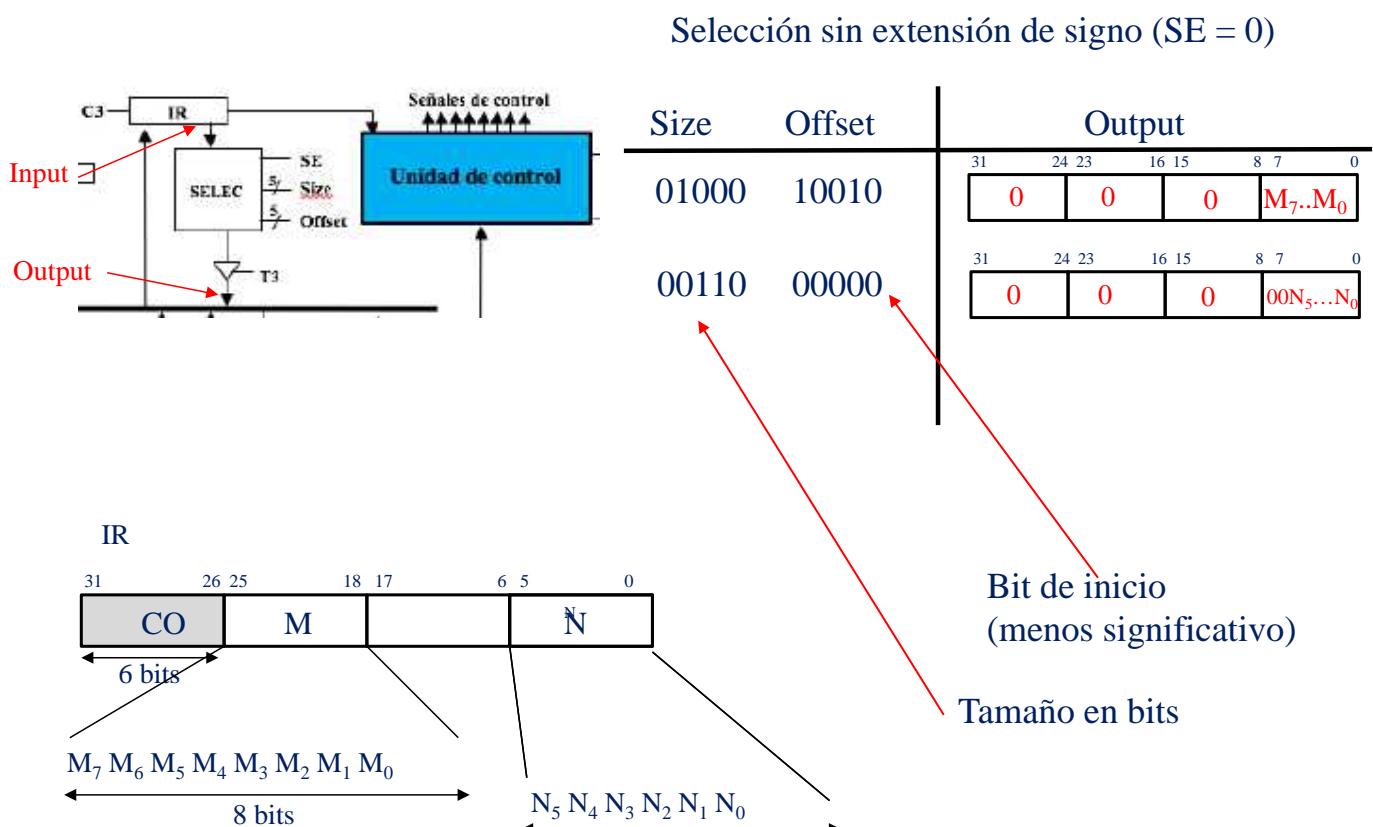


# Registro de instrucción



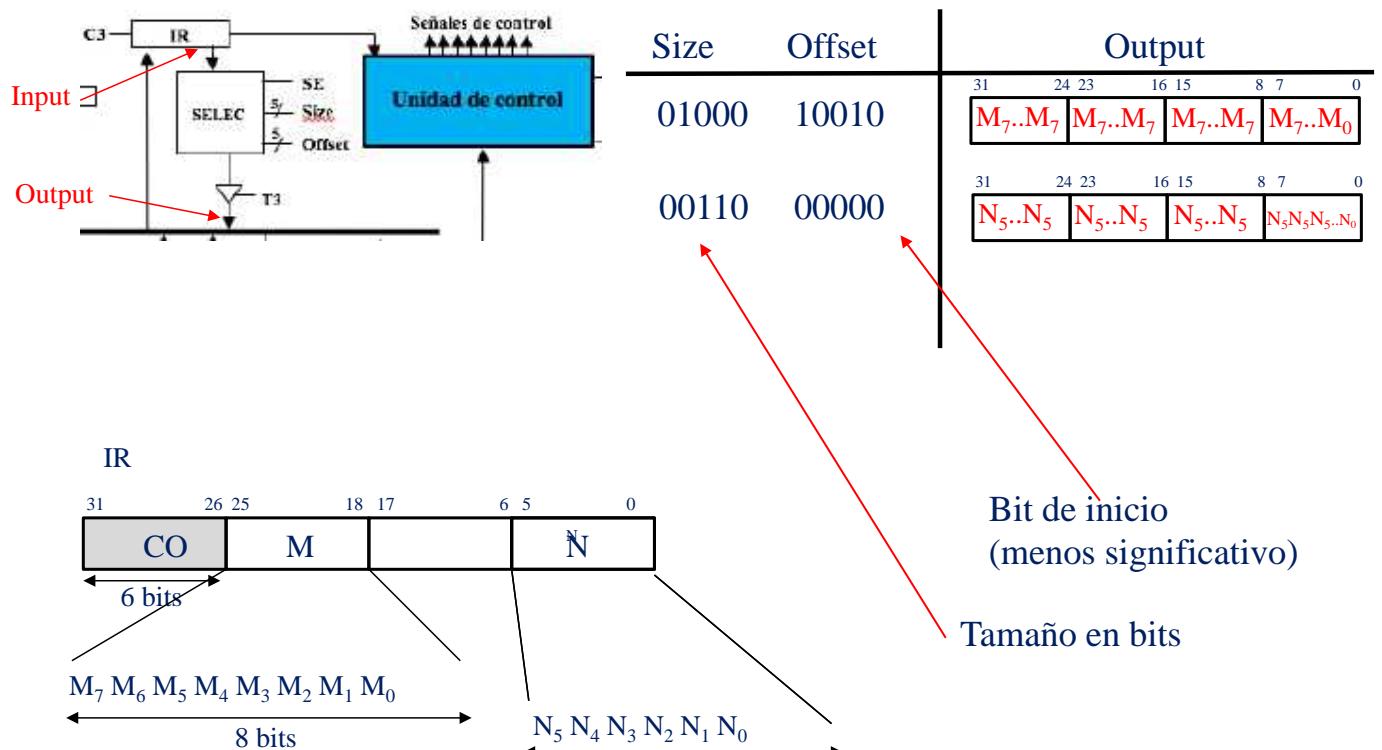
- ▶ C3 – del bus interno al IR
- ▶ SELEC: Transfiere el contenido de IR al bus
  - ▶ Size: Tamaño
  - ▶ Offset: desplazamiento
    - ▶ Bit de inicio (menos significativo)
  - ▶ SE: extensión de signo

# Registro Selector



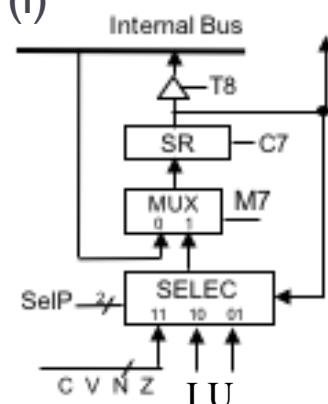
# Registro Selector

Selección con extensión de signo (SE = 1)



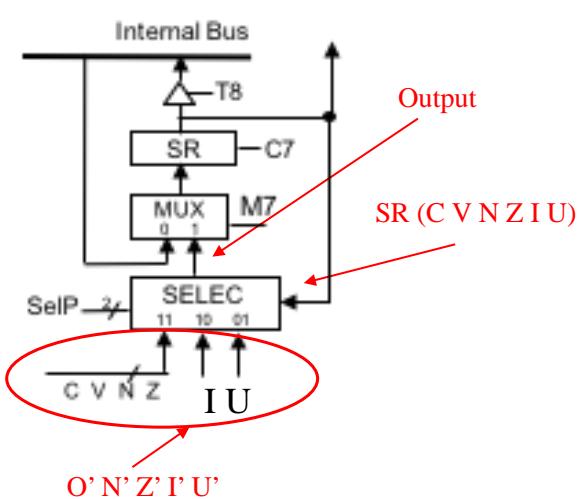
# Registro de estado

- ▶ Almacena información (bits de estado) sobre el estado del programa que se está ejecutando en el procesador:
  - ▶ Resultado de la última operación en la ALU: C,V,N,Z
  - ▶ Si el procesador está ejecutando en modo núcleo o modo usuario (U)
  - ▶ Si las interrupciones están habilitadas o no (I)
- ▶ Señales
  - ▶ C7 – de bus interno al SR
  - ▶ SelP,M7 – flags de ALU,I,o U a SR
  - ▶ T8 – del SR al bus interno



# Registro de estado

Operación de SELEC:



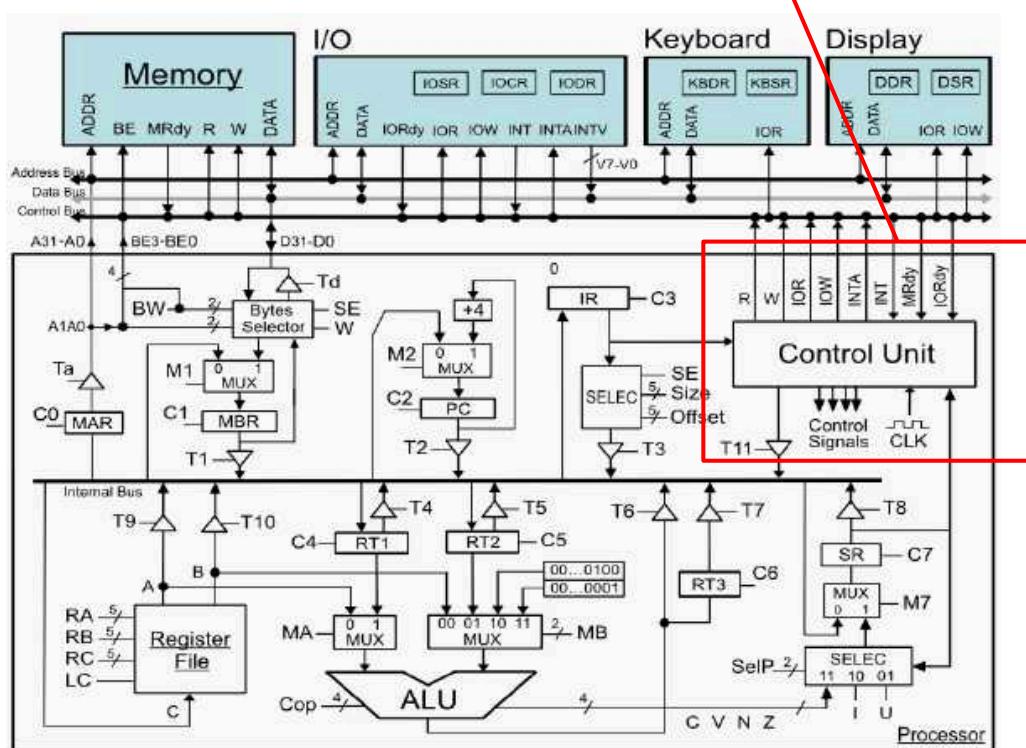
if ( $\text{SelP1} = 1 \text{ AND } \text{SelP0} == 1$ )  
Output = **C' V' N' Z' I U**

if ( $\text{SelP1} == 1 \text{ AND } \text{SelP0} == 0$ )  
Output = **C V N Z I' U**

if ( $\text{SelP1} == 0 \text{ AND } \text{SelP0} == 1$ )  
Output = **C V N Z I U'**

# Estructura de un computador elemental

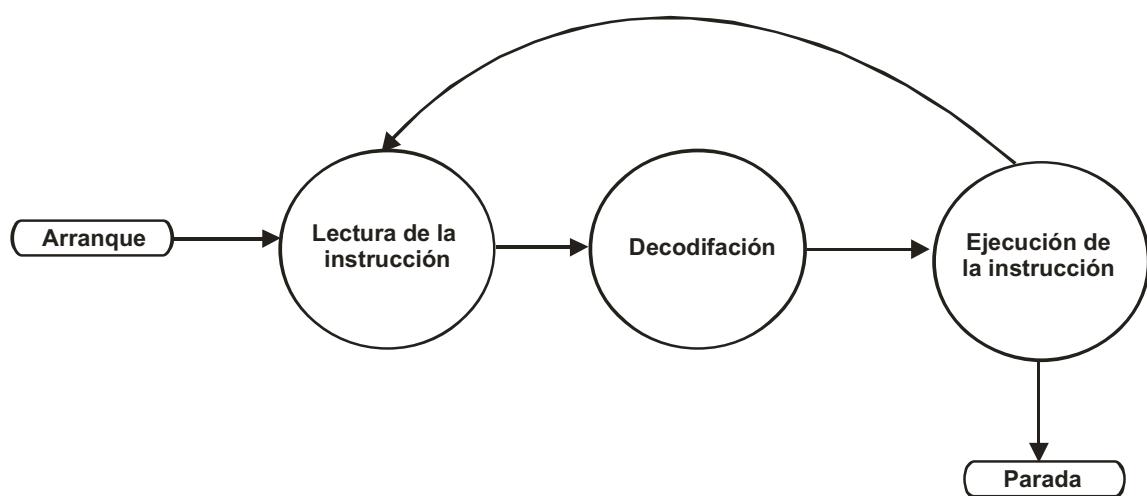
## Unidad de Control (UC)



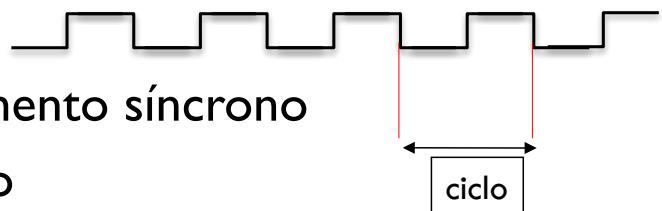
# Unidad de control

## Fases de ejecución de una instrucción

- ▶ Funciones básicas
  - ▶ Lectura de instrucciones de la memoria
  - ▶ Decodificación
  - ▶ Ejecución de instrucciones



## Reloj

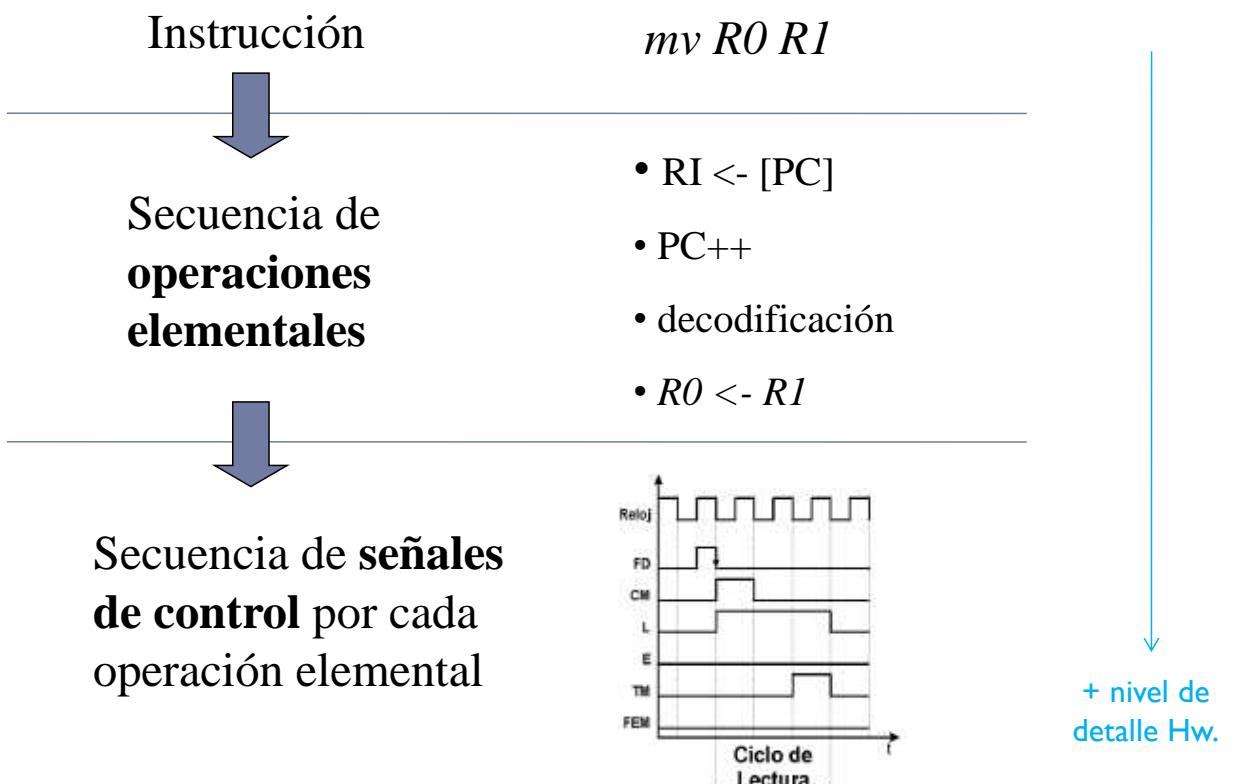


- ▶ Un computador es un elemento síncrono
- ▶ Controla el funcionamiento
- ▶ El reloj temporiza las operaciones
  - ▶ En un ciclo de reloj se ejecutan una o más operaciones elementales siempre que no haya conflicto
  - ▶ Durante el ciclo se mantienen activadas las señales de control necesarias
- ▶ En un mismo ciclo se puede realizar
  - ▶  $\text{MAR} \leftarrow \text{PC}$  y  $\text{RT3} \leftarrow \text{RT2} + \text{RT1}$
- ▶ En un mismo ciclo **no** se puede realizar
  - ▶  $\text{MAR} \leftarrow \text{PC}$  y  $\text{RI} \leftarrow \text{RT3}$  **¿por qué?**

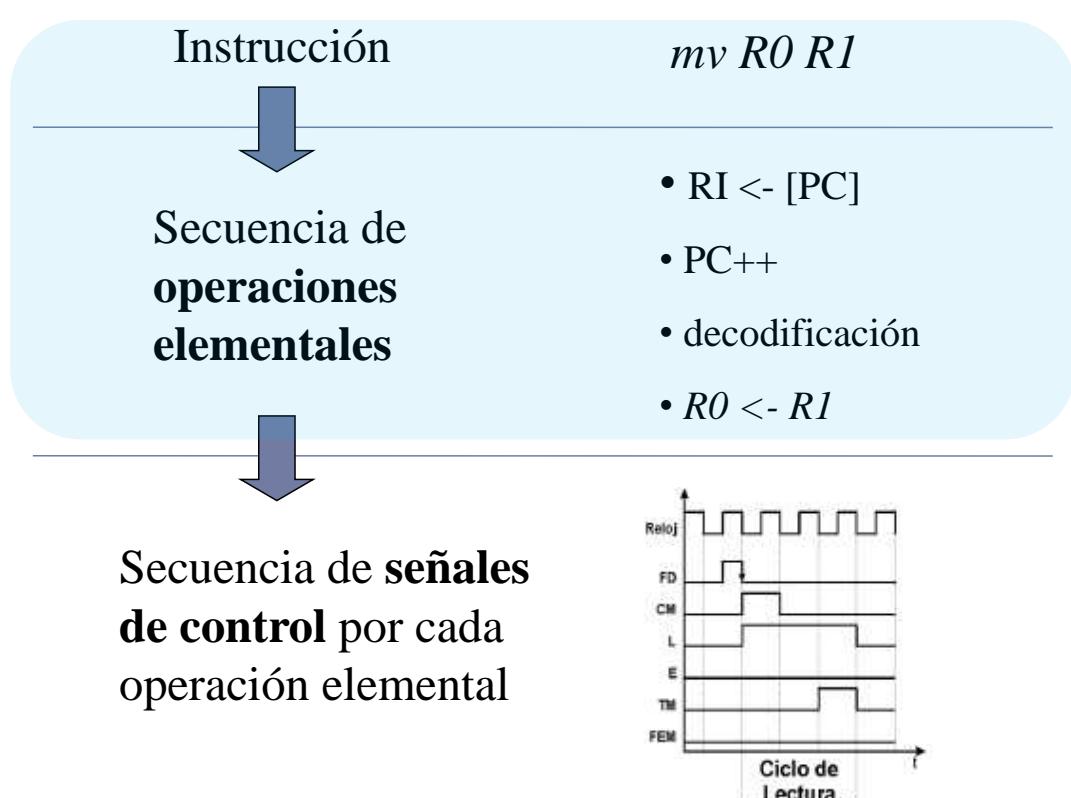
## Ejercicio

- ▶ ¿Cuál es la duración del ciclo de un computador con una frecuencia de reloj de 1 GHz?

# Descripción de la actividad de la U.C.



# Descripción de la actividad de la U.C.



# Fases de ejecución de una instrucción

- ▶ **Lectura de la instrucción, captación o fetch**
  - ▶ Leer la instrucción almacenada en la dirección de memoria indicada por PC y llevarla a RI.
  - ▶ Incremento del PC
- ▶ **Decodificación**
  - ▶ Análisis de la instrucción en RI para determinar:
    - ▶ La operación a realizar.
    - ▶ Direcciónamiento a aplicar.
    - ▶ Señales de control a activar
- ▶ **Ejecución**
  - ▶ Generación de las señales de control en cada ciclo de reloj.

## Lectura de una instrucción

Ciclo	Op. Elemental
C1	$\text{MAR} \leftarrow \text{PC}$
C2	$\text{PC} \leftarrow \text{PC} + 4$
C3	$\text{MBR} \leftarrow \text{MP}$
C4	$\text{IR} \leftarrow \text{MBR}$



Ciclo	Op. Elemental
C1	$\text{MAR} \leftarrow \text{PC}$
C2	$\text{PC} \leftarrow \text{PC} + 4,$ $\text{MBR} \leftarrow \text{MP}$
C3	$\text{IR} \leftarrow \text{MBR}$

Posibilidad de operaciones simultáneas

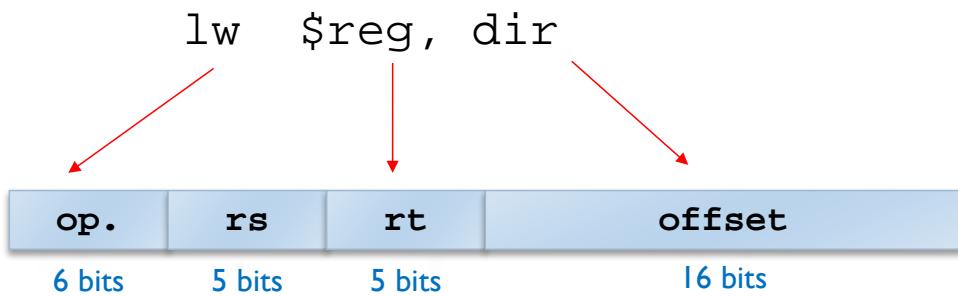
## Señales de control del ciclo de fetch

- ▶ Especificación de las señales de control activas en cada ciclo de reloj.
  - ▶ Se puede generar a partir del nivel RT.

Ciclo	Op. Elemental	Señales de control activadas
C1	$\text{MAR} \leftarrow \text{PC}$	T2, C0
C2	$\text{PC} \leftarrow \text{PC} + 4$ , $\text{MBR} \leftarrow \text{MP}$	C2, M1 Ta, R, CI, MI, BW=11
C3	$\text{IR} \leftarrow \text{MBR}$	T1, C3

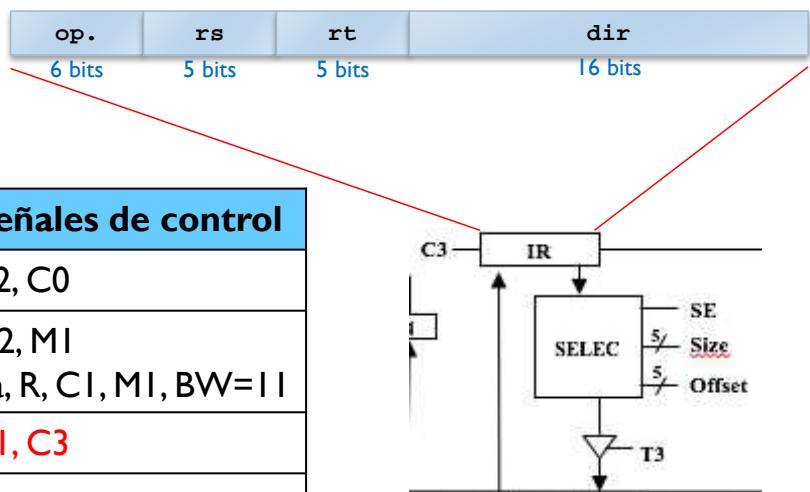
## Ejecución de la instrucción de MIPS

- ▶ lw \$reg, dir



## Ejecución de lw \$reg, dir

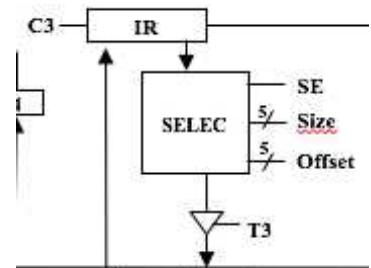
Ciclo	Op. Elemental	Señales de control
C1	MAR $\leftarrow$ PC	T2, C0
C2	PC $\leftarrow$ PC + 4, MBR $\leftarrow$ MP	C2, MI Ta, R, CI, MI, BW=11
C3	IR $\leftarrow$ MBR	T1, C3
C4		
C5		
C6		
C7		



## Ejecución de lw \$reg, dir

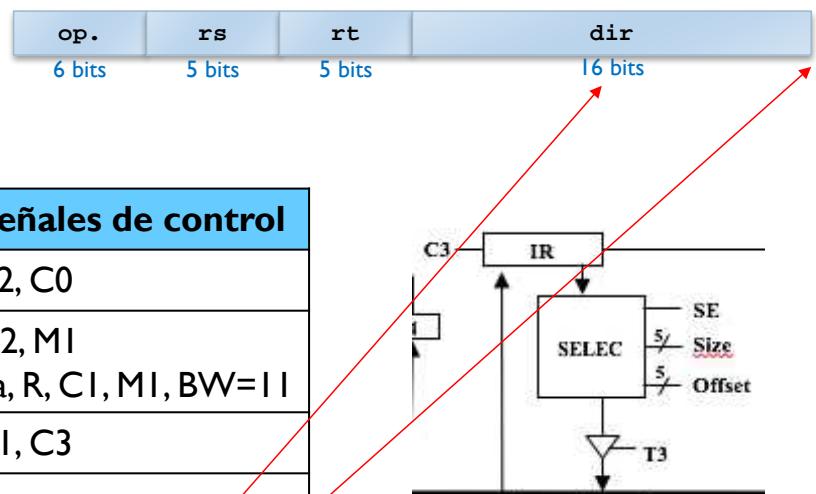


Ciclo	Op. Elemental	Señales de control
C1	MAR $\leftarrow$ PC	T2, C0
C2	PC $\leftarrow$ PC + 4, MBR $\leftarrow$ MP	C2, MI Ta, R, CI, MI, BW=11
C3	IR $\leftarrow$ MBR	T1, C3
C4	Decodificación	
C5		
C6		
C7		



## Ejecución de lw \$reg, dir

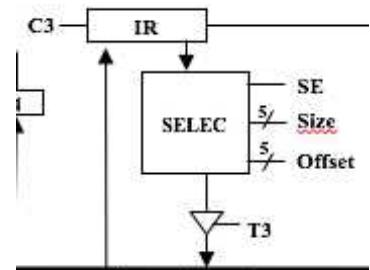
Ciclo	Op. Elemental	Señales de control
C1	MAR $\leftarrow$ PC	T2, C0
C2	PC $\leftarrow$ PC + 4, MBR $\leftarrow$ MP	C2, MI Ta, R, CI, MI, BW=11
C3	IR $\leftarrow$ MBR	T1, C3
C4	Decodificación	
C5	MAR $\leftarrow$ RI(dir)	C0, T3, Size = 10000 Offset = 00000
C6		
C7		



## Ejecución de lw \$reg, dir



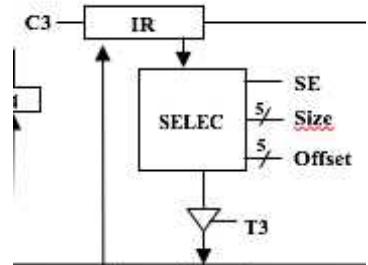
Ciclo	Op. Elemental	Señales de control
C1	MAR $\leftarrow$ PC	T2, C0
C2	PC $\leftarrow$ PC + 4, MBR $\leftarrow$ MP	C2, MI Ta, R, CI, MI, BW=11
C3	IR $\leftarrow$ MBR	T1, C3
C4	Decodificación	
C5	MAR $\leftarrow$ RI(dir)	C0, T3, Size =10000 Offset = 00000
C6	MBR $\leftarrow$ MP	Ta, R, CI, MI, BW=11
C7		



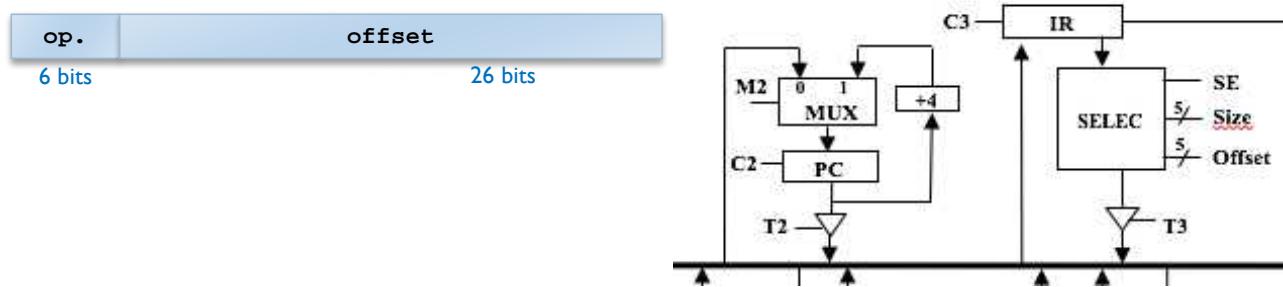
## Ejecución de lw \$reg, dir



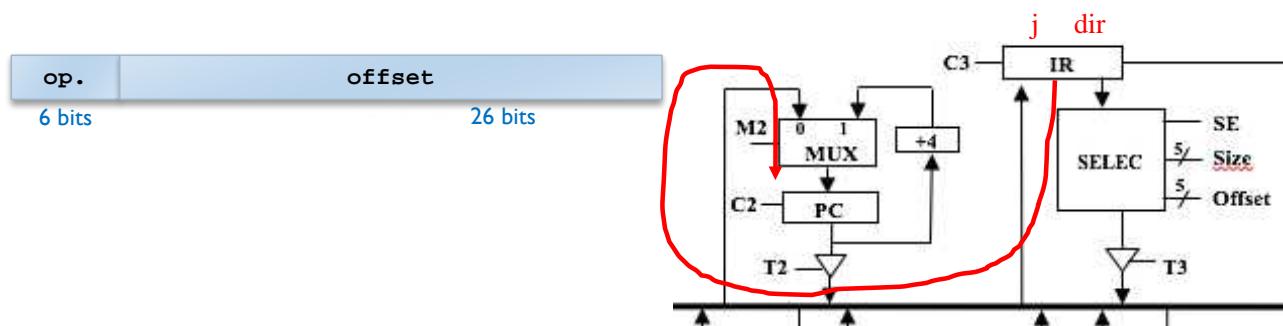
Ciclo	Op. Elemental	Señales de control
C1	MAR $\leftarrow$ PC	T2, C0
C2	PC $\leftarrow$ PC + 4, MBR $\leftarrow$ MP	C2, MI Ta, R, CI, MI, BW=11
C3	IR $\leftarrow$ MBR	T1, C3
C4	Decodificación	
C5	MAR $\leftarrow$ RI(dir)	C0, T3, Size =10000 Offset = 00000
C6	MBR $\leftarrow$ MP	Ta, R, CI, MI, BW=11
C7	\$reg $\leftarrow$ MBR	T1, RC=id de \$reg LC



## Ejecución de j dir



## Ejecución de j dir



Ciclo	Op. Elemental	Señales de control
C1	$\text{MAR} \leftarrow \text{PC}$	T2, C0
C2	$\text{PC} \leftarrow \text{PC} + 4$ , $\text{MBR} \leftarrow \text{MP}$	C2, MI Ta, R, CI, MI, BW=11
C3	$\text{IR} \leftarrow \text{MBR}$	T1, C3
C4	Decodificación	
C5	$\text{PC} \leftarrow \text{RI}(\text{dir})$	C2, T3, Size = 11010 (26) Offset = 00000

# Ejercicio

## ▶ Instrucciones que caben en una palabra:

- ▶ sw \$reg, dir
- ▶ add \$rd, \$r01, \$r02
- ▶ addi \$rd, \$r01, inm
- ▶ lw \$reg1, desp(\$reg2)
- ▶ j dir
- ▶ jr \$reg
- ▶ beq \$r01, \$r02, desp

## beqz \$reg, desplaz

Ciclo	Op. Elemental
C1	$MAR \leftarrow PC$
C2	$PC \leftarrow PC + 4$ , $MBR \leftarrow MP$
C3	$IR \leftarrow MBR$
C4	Decodificación
C5	$$reg + \$0$
C6	Si $SR.Z == 0$ salto a fetch
C7	$RT2 \leftarrow PC$
C8	$RT1 \leftarrow IR(desplaz)$
C9	$RT1 \leftarrow RT1 * 4$
C10	$PC \leftarrow RT1 + RT2$

Si  $$reg == 0$   
 $PC \leftarrow PC + desp * 4$

## Instrucciones que ocupan varias palabras

Ejemplo: addm R1 , dir       $R1 \leftarrow R1 + MP[dir]$

Formato:

addm	R1		Dir (dirección)
1 <sup>a</sup> palabra		2 <sup>a</sup> palabra	

Ciclo	Op. Elemental
C1	$MAR \leftarrow PC$
C2	$PC \leftarrow PC + 4$ , $MBR \leftarrow MP$
C3	$IR \leftarrow MBR$
C4	Decodificación
C5	$MAR \leftarrow PC$

Ciclo	Op. Elemental
C6	$MBR \leftarrow MP$ , $PC \leftarrow PC + 4$
C7	$MAR \leftarrow MBR$
C8	$MBR \leftarrow MP$
C9	$RTI \leftarrow MBR$
C10	$RI \leftarrow RI + RTI$

# Ejemplo

ADD (R<sub>2</sub>) R<sub>3</sub> (R<sub>4</sub>)

## A. Fetch + Decodif.

- 1.- MAR  $\leftarrow$  PC
- 2.- RI  $\leftarrow$  Memoria(MAR)
- 3.- PC  $\leftarrow$  PC + "4"
- 4.- Decodificación de la instrucción

## B. Traer operandos

- 5.- MAR  $\leftarrow$  R<sub>4</sub>
- 6.- MBR  $\leftarrow$  Memoria(MAR)
- 7.- RTI  $\leftarrow$  MBR

## C. Ejecutar

- 8.- MBR  $\leftarrow$  R<sub>3</sub> + RTI

## D. Guardar resultados

- 9.- MAR  $\leftarrow$  R<sub>2</sub>
- 10.- Memoria(MAR)  $\leftarrow$  MBR

## Recordatorio

- ▶ **No es posible atravesar un registro en el ciclo de reloj**
- ▶ **No es posible llevar a un bus dos valores a la vez.**

# Modos de ejecución

## ▶ Modo usuario

- ▶ El procesador no puede **ejecutar instrucciones privilegiadas** (ejemplo: instrucciones de E/S, de habilitación de interrupciones, ...)
- ▶ Si un proceso de usuario ejecuta una instrucción privilegiada se produce una interrupción

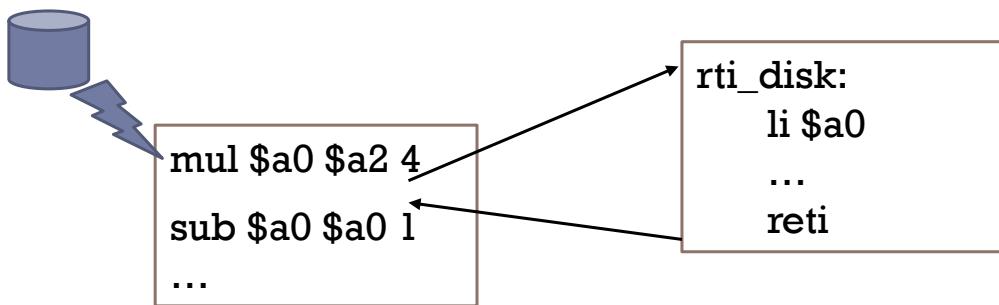
## ▶ Modo núcleo

- ▶ Reservado al sistema operativo
  - ▶ El procesador puede ejecutar todo el repertorio de instrucciones
- ▶ Se indica con un bit situado en el **registro de estado (U)**

# Inerrupciones

- ▶ Señal que llega a la unidad de control y que rompe la secuencia normal de ejecución
- ▶ Causas:
  - ▶ Cuando ocurre un error en la ejecución de la instrucción (división por cero, ...)
  - ▶ Ejecución de una instrucción ilegal
  - ▶ Acceso a una posición de memoria ilegal
  - ▶ Cuando un periférico solicita la atención del procesador
  - ▶ El reloj. Interrupciones de reloj
- ▶ Cuando se genera una interrupción se detiene el programa actual y se transfiere la ejecución a otro programa que atiende la interrupción

# Idea de interrupción

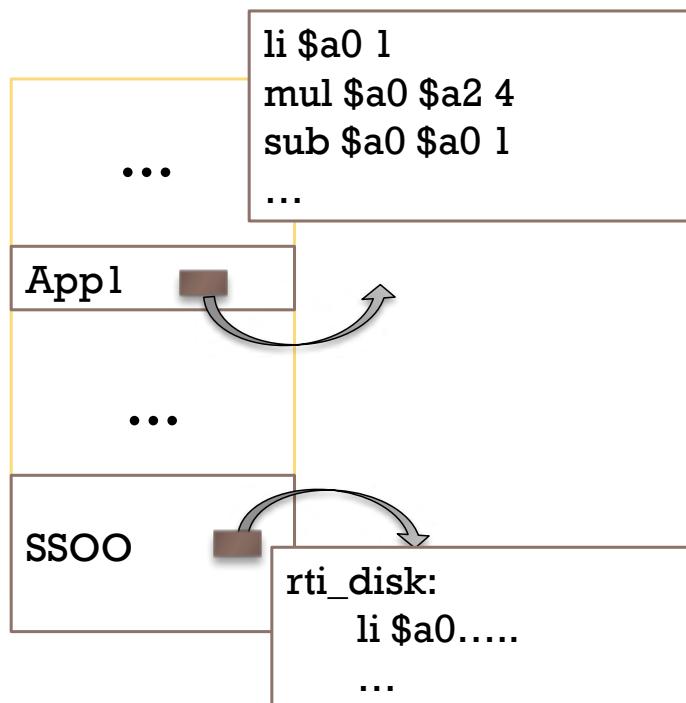


- ▶ Señal que llega a la U.C. y que rompe la secuencia normal de ejecución: se detiene el programa actual y se ejecuta otro que atiende la interrupción.
- ▶ Ejemplo de causas:
  - ▶ Cuando un periférico solicita la atención del procesador
  - ▶ Cuando ocurre un error en la ejecución de la instrucción, ...

# Clasificación de las interrupciones

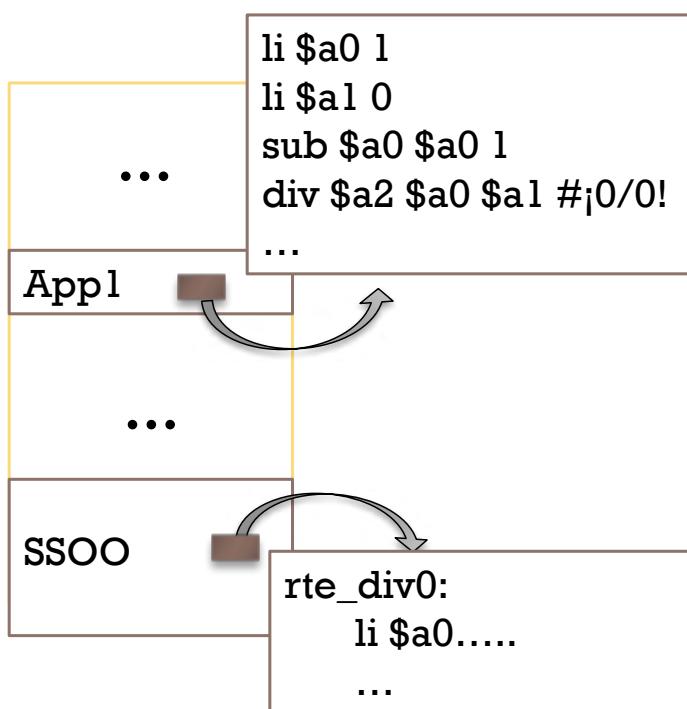
- ▶ **Excepciones hardware síncronas**
  - ▶ División por cero, acceso a una posición de memoria ilegal,
- ▶ **Excepciones hardware asíncronas**
  - ▶ Fallos o errores en el HW
- ▶ **Interrupciones externas**
  - ▶ Periféricos, interrupción del reloj
- ▶ **Llamadas al sistema**
  - ▶ Instrucciones máquina especiales que generan una interrupción para activar al sistema operativo

## Excepciones hardware asíncronas e Interrupciones externas



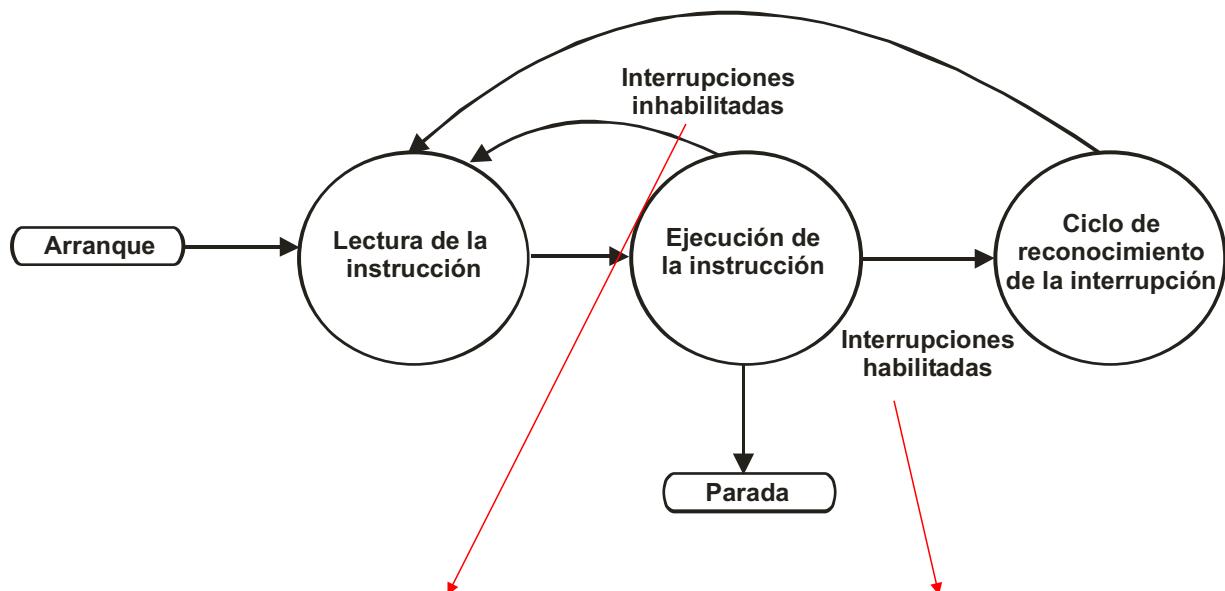
- ▶ Originan una ruptura de secuencia no programada
  - ▶ **Al final microprograma de la instrucción en curso ver si hay interrupción pendiente, y si la hay...**
  - ▶ ...Bifurcación a subrutina del S.O. que la trata
- ▶ Posteriormente, restituye el estado y devuelve el control al programa interrumpido.
- **Causa asíncrona a la ejecución del programa en curso**
  - ▶ Atención a periférico
  - ▶ Etc.

## Excepciones hardware síncronas



- ▶ Originan una ruptura de secuencia no programada
  - ▶ Dentro del microprograma de la instrucción en curso...
  - ▶ ...Bifurcación a subrutina del S.O. que la trata
- ▶ Posteriormente, restituye el estado y devuelve el control al programa interrumpido o finaliza su ejecución
- Causa síncrona a la ejecución del programa en curso
  - ▶ División entre cero
  - ▶ Etc.

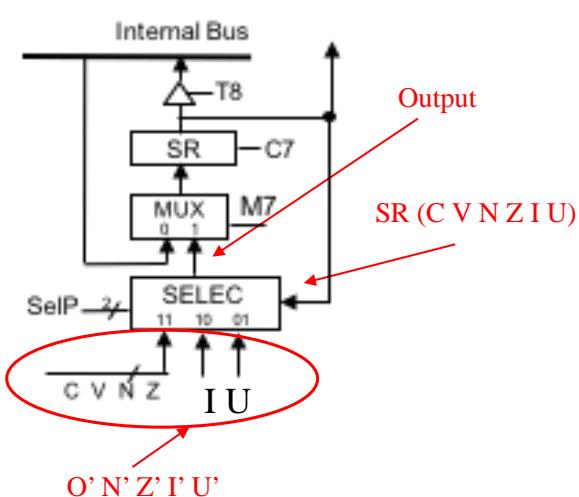
# Tratamiento de las interrupciones



Se indica con un bit situado en el **registro de estado (I)**

# Activación del registro de estado

Operación de SELEC:



if ( $\text{SelP1} = 1 \text{ AND } \text{SelP0} == 1$ )  
Output =  $C' V' N' Z' I' U$

if ( $\text{SelP1} == 1 \text{ AND } \text{SelP0} == 0$ )  
Output =  $C V N Z I' U$

if ( $\text{SelP1} == 0 \text{ AND } \text{SelP0} == 1$ )  
Output =  $C V N Z I U'$

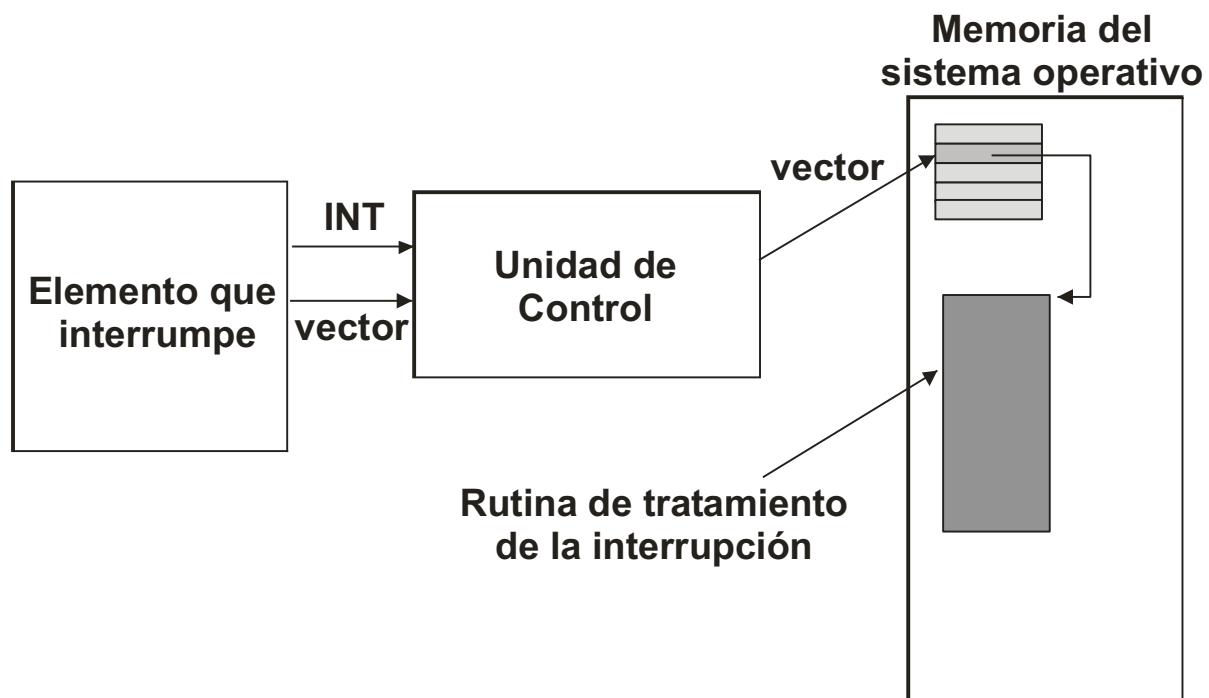
## Ciclo de reconocimiento de la interrupción

- ▶ Durante este ciclo la Unidad de control realiza los siguientes pasos:
  - ▶ Comprueba se hay activada una señal de interrupción.
  - ▶ Si está activada:
    - ▶ Salva el contador de programa y el registro de estado
    - ▶ Pasa de modo usuario a modo núcleo
    - ▶ Obtiene la dirección de la rutina de tratamiento de la interrupción
    - ▶ Almacena en el contador de programa la dirección obtenida (de esta forma la siguiente instrucción será la de la rutina de tratamiento)

## Rutina de tratamiento de la interrupción

- ▶ Forma parte del código del sistema operativo
- ▶ Salva el resto de registros del procesador
- ▶ Atiende la interrupción
- ▶ Restaura los registros del procesador utilizados por el programa interrumpido
- ▶ Ejecuta una instrucción máquina especial: RETI
  - ▶ Restaura el registro de estado del programa interrumpido (fijando de nuevo el modo del procesador a modo usuario)
  - ▶ Restaura el contador de programa (de forma que la siguiente instrucción es la del programa interrumpido).

## Interrupciones vectorizadas



## Inerrupciones vectorizadas

- ▶ El elemento que interrumpe suministra el **vector de interrupción**
- ▶ Este vector es un índice en una tabla que contiene la dirección de la rutina de tratamiento de la interrupción.
- ▶ La UC lee el contenido de esta entrada y carga el valor en el PC
- ▶ Cada sistema operativo rellena esta tabla con las direcciones de cada una de las rutinas de tratamiento, que son dependientes de cada sistema operativo.

# Interrupciones en un PC con Windows

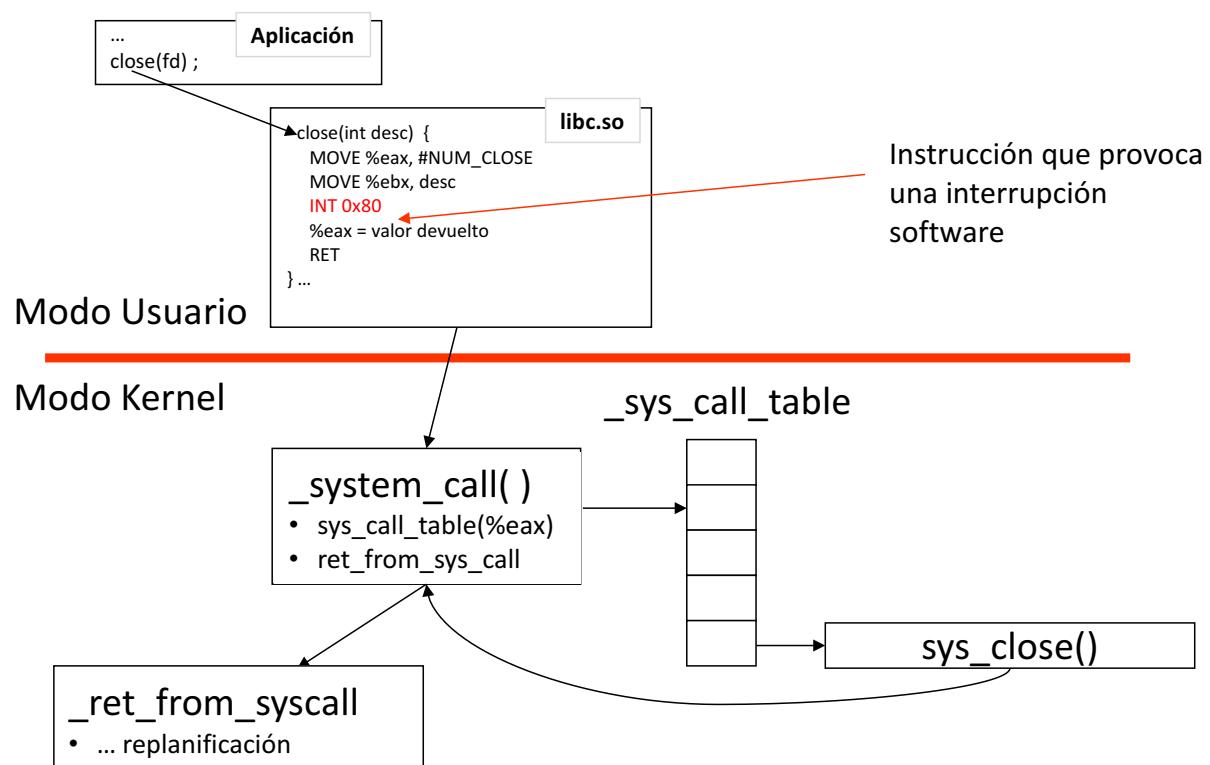
Información del sistema			
Archivo	Editar	Ver	Ayuda
<b>Resumen del sistema</b>			
Recurso de hardware	Recurso	Dispositivo	Estado
Conflictos/uso compartido	IRQ 0	Cronómetro del sistema	OK
DMA	IRQ 1	Teléfono PS/2 estándar	OK
Hardware forzado	IRQ 8	Sistema CMOS/reloj en tiempo real	OK
E/S	IRQ 11	Controladora de SMBus de la familia Intel(R) ICH10 - 3A30	OK
IRQ 12	IRQ 12	Mouse PS/2 de Microsoft	OK
IRQ 13	IRQ 13	Procesador de datos numéricos	OK
IRQ 16	IRQ 16	Controladora estándar PCI IDE de doble canal	OK
IRQ 17	IRQ 16	Controladora de host universal USB de la familia Intel(R) ICH10 - 3A37	OK
IRQ 17	IRQ 17	Puerto raíz PCI Express 1 de la familia Intel(R) ICH10 - 3A40	OK
IRQ 18	IRQ 17	Puerto raíz PCI Express 5 de la familia Intel(R) ICH10 - 3A48	OK
IRQ 18	IRQ 18	Controladora de host universal USB de la familia Intel(R) ICH10 - 3A36	OK
IRQ 18	IRQ 18	Puerto raíz PCI Express 3 de la familia Intel(R) ICH10 - 3A44	OK
IRQ 18	IRQ 18	Controladora de host mejorado USB de la familia Intel(R) ICH10 - 3A3C	OK
IRQ 19	IRQ 18	Realtek PCIe GBE Family Controller	OK
IRQ 19	IRQ 19	Controladora de host VIA compatible con OHCI 1394	OK
IRQ 19	IRQ 19	Controladora de host universal USB de la familia Intel(R) ICH10 - 3A29	OK
IRQ 19	IRQ 19	Puerto raíz PCI Express 4 de la familia Intel(R) ICH10 - 3A46	OK
IRQ 20	IRQ 19	Controladora de host universal USB de la familia Intel(R) ICH10 - 3A29	OK
IRQ 20	IRQ 20	Controladora estándar PCI IDE de doble canal	OK
IRQ 21	IRQ 20	Controladora de host universal USB de la familia Intel(R) ICH10 - 3A38	OK
IRQ 22	IRQ 21	Controladora de High Definition Audio	OK
IRQ 23	IRQ 22	Controladora de host mejorado USB de la familia Intel(R) ICH10 - 3A3A	OK
IRQ 23	IRQ 23	Controladora de host universal USB de la familia Intel(R) ICH10 - 3A34	OK
IRQ 24	IRQ 23	NVIDIA GeForce GTX 290	OK
IRQ 81	IRQ 24	Sistema Microsoft compatible con ACPI	OK
IRQ 82	IRQ 81	Sistema Microsoft compatible con ACPI	OK
IRQ 83	IRQ 82	Sistema Microsoft compatible con ACPI	OK
IRQ 84	IRQ 83	Sistema Microsoft compatible con ACPI	OK

## Interrupciones Software. Llamadas al sistema y sistemas operativos

- ▶ El mecanismo de llamadas al sistema es el que permite que los programas de usuario puedan solicitar los servicios que ofrece el sistema operativo
  - ▶ Cargar programas en memoria para su ejecución
  - ▶ Acceso a los dispositivos periféricos
- ▶ Similar a las llamadas al sistema que ofrece el simulador QtSPIM

# Interrupciones software

## Llamadas al sistema (ejemplo: Linux)



## Interrupciones del reloj y sistemas operativos

- ▶ La señal que gobierna la ejecución de las instrucciones máquina se divide mediante un divisor de frecuencia para generar una interrupción externa cada cierto intervalo de tiempo (pocos milisegundos)
- ▶ Estas **interrupciones de reloj** o tics son interrupciones periódicas que permite que el sistema operativo entre a ejecutar de forma periódica evitando que un programa de usuario monopolice la CPU
  - ▶ Permite alternar la ejecución de diversos programas en un sistema dado la apariencia de ejecución simultánea
  - ▶ Cada vez que llega una interrupción de reloj se suspende al programa y se salta al sistema operativo que ejecuta el **planificador** para decidir el **siguiente** programa a ejecutar

## Ejercicio

- ▶ Señales de control a activar en caso de que se haya producido una interrupción
  - ▶ Se obtiene el vector de interrupción del bus de datos

Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 4 (II) El procesador

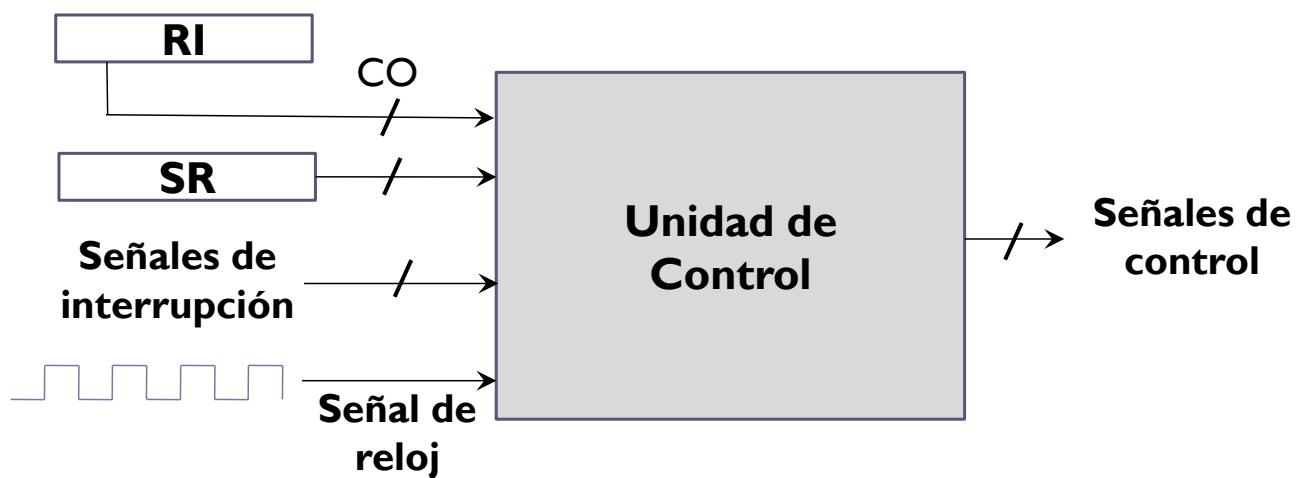
Estructura de Computadores  
Grado en Ingeniería Informática



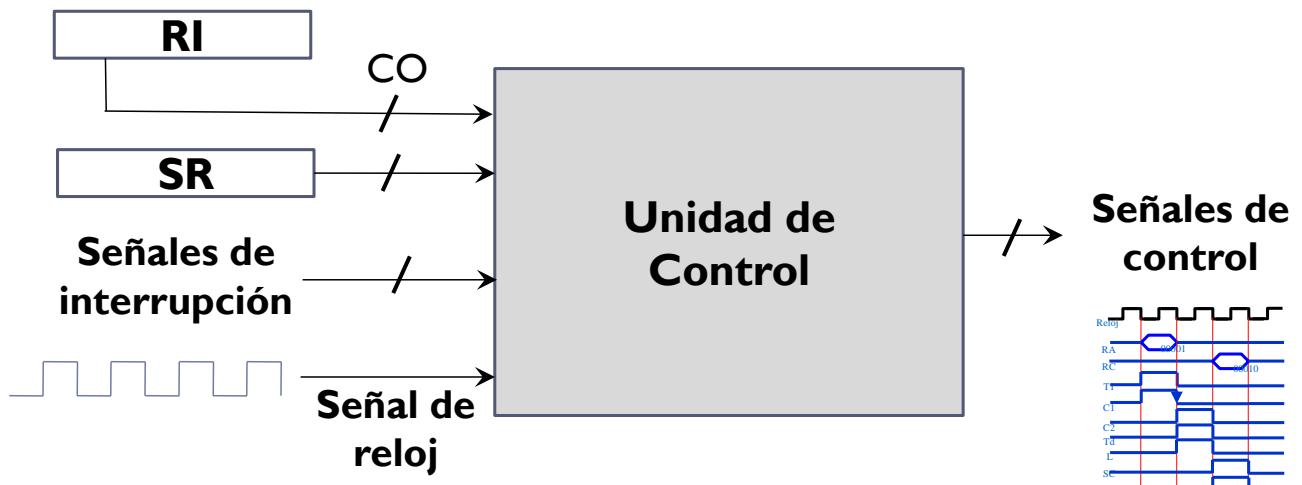
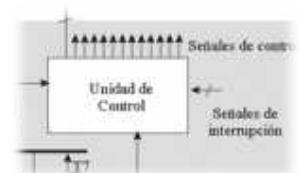
# Contenidos

1. Elementos de un computador
2. Organización del procesador
3. La unidad de control
4. Ejecución de instrucciones
5. Modos de ejecución
6. Interrupciones
7. Diseño de la unidad de control
8. Arranque de un computador
9. Prestaciones y paralelismo

# Unidad de control



# Unidad de control



- ▶ Cada una de las señales de control es función del valor de:
  - ▶ El contenido del RI
  - ▶ El contenido de RE
  - ▶ El momento del tiempo

# Diseño de la unidad de control

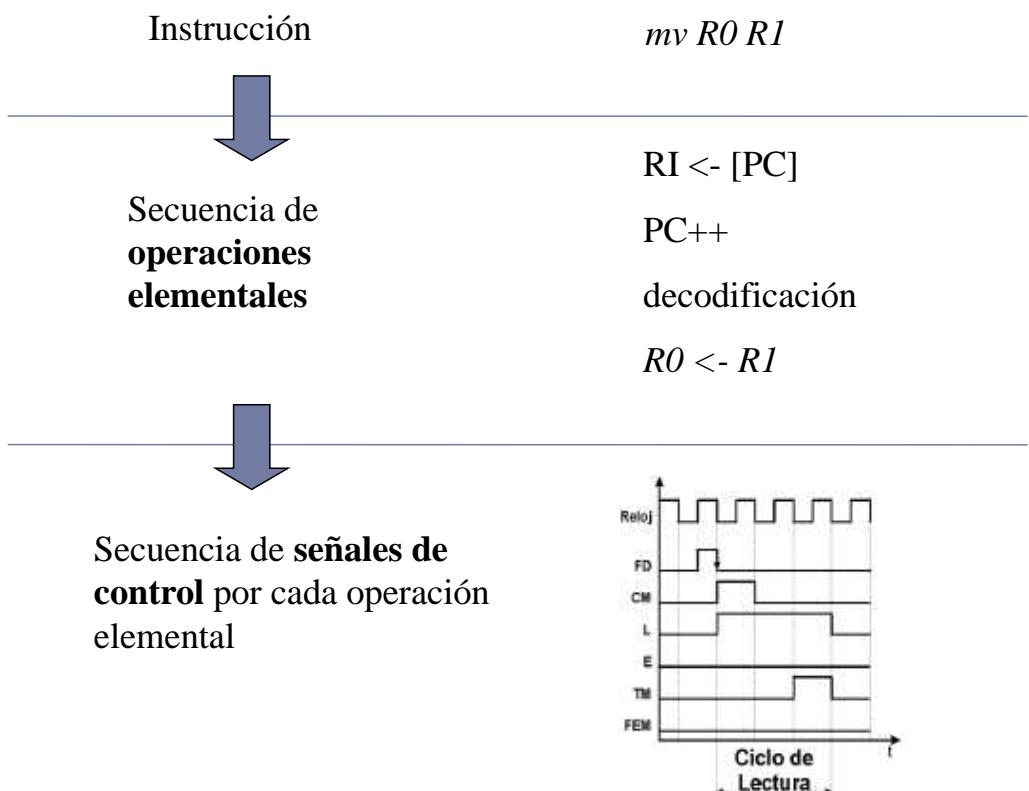
- ▶ Para cada instrucción máquina:
  1. Definir el comportamiento en lenguaje de transferencia de registro (RT) en cada ciclo de reloj
  2. Traducir el comportamiento a valores de cada señal de control en cada ciclo de reloj
  3. Diseñar un circuito que genere el valor de cada señal de control en cada ciclo de reloj

# Diseño de la unidad de control

## ▶ Para cada instrucción máquina:

1. Definir el comportamiento en lenguaje de transferencia de registro (RT) en cada ciclo de reloj
2. Traducir el comportamiento a valores de cada señal de control en cada ciclo de reloj
3. Diseñar un circuito que genere el valor de cada señal de control en cada ciclo de reloj

# Diseño de la unidad de control



# Diseño de la unidad de control

- ▶ Para cada instrucción máquina:
  1. Definir el comportamiento en lenguaje de transferencia de registro (RT) en cada ciclo de reloj
  2. Traducir el comportamiento a valores de cada señal de control en cada ciclo de reloj
  3. Diseñar un circuito que genere el valor de cada señal de control en cada ciclo de reloj

# Técnicas de control

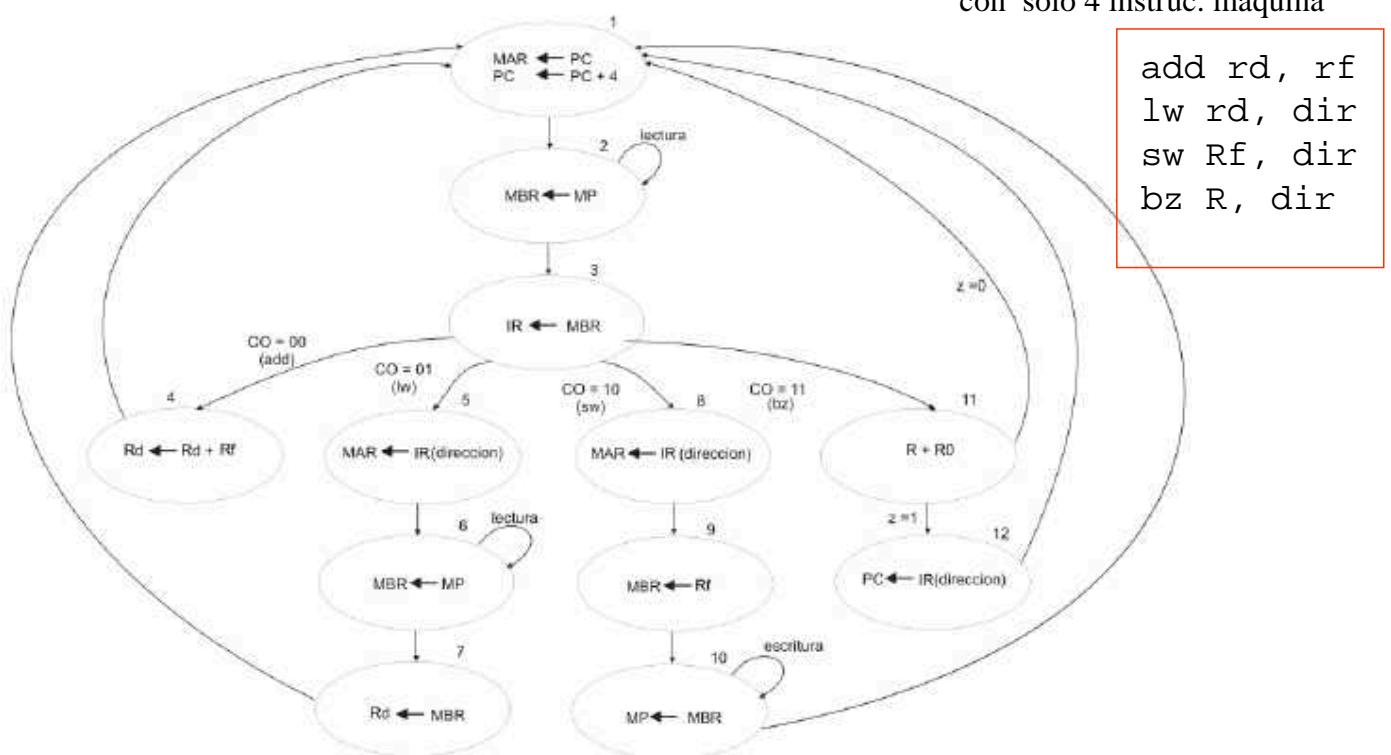
- ▶ Unidad de control cableada
- ▶ Unidad de control microprogramada

# Ejemplo

- ▶ Diseño de una unidad de control para un juego de 4 instrucciones máquina:
- ▶ Instrucciones a considerar:
  - ▶ add Rd, Rf:       $Rd \leftarrow Rd + Rf$
  - ▶ lw Rd, dir:       $Rd \leftarrow MP[dir]$
  - ▶ sw Rf, dir:       $MP[dir] \leftarrow Rf$
  - ▶ bz R, dir:      if ( $R == 0$ )  $PC \leftarrow dir$

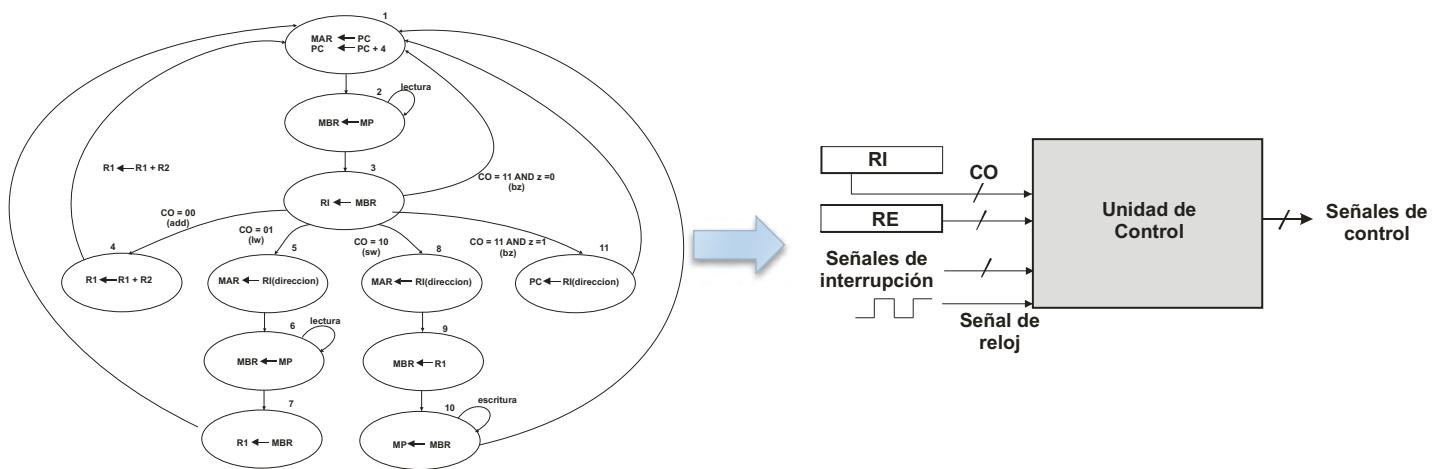
# Máquina de estados para el ejemplo

Ejemplo para un computador con solo 4 instruc. máquina



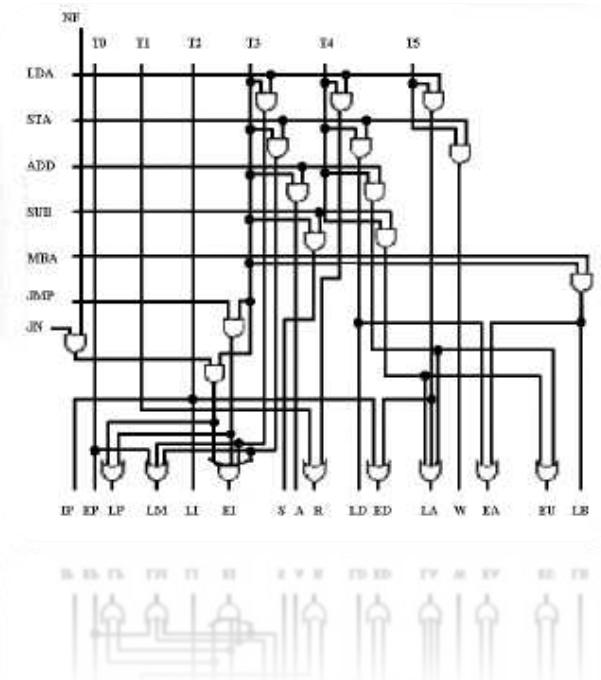
# Técnicas de control

- ▶ **Dos técnicas** de diseñar y construir una unidad de control:
  - Lógica cableada
  - Lógica almacenada (micropogramación)



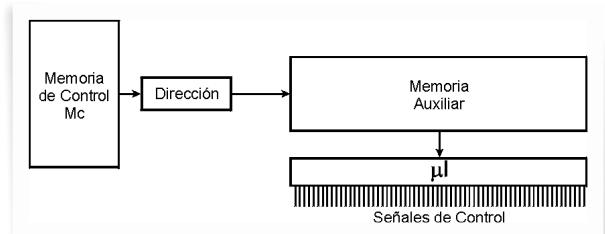
# Unidad de control cableada

- ▶ Construcción mediante puertas lógicas, siguiendo los métodos de diseño lógico.
- ▶ Características:
  - ▶ Laborioso y costoso el diseño y puesta a punto del circuito
  - ▶ Difícil de modificar:
    - ▶ rediseño completo.
  - ▶ Muy rápida  
(usado en computadores RISC)

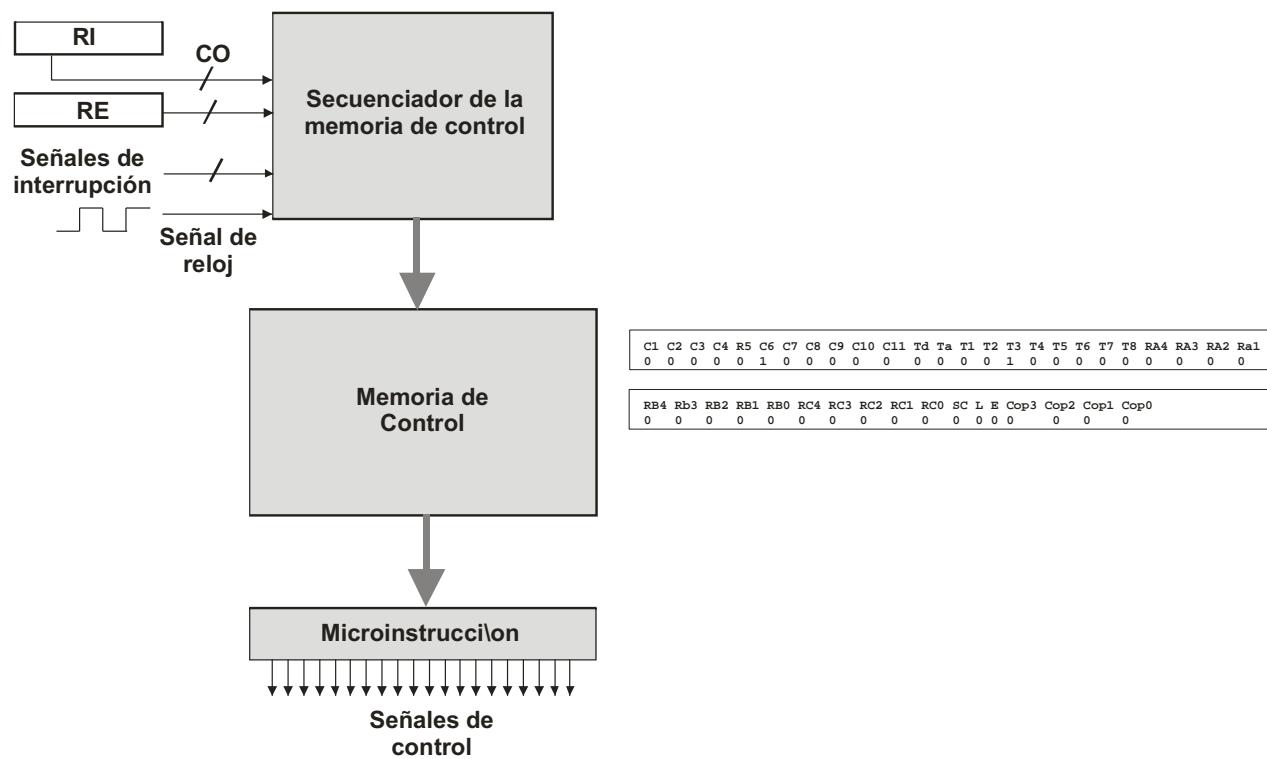


# Unidad de control almacenada. Microprogramación

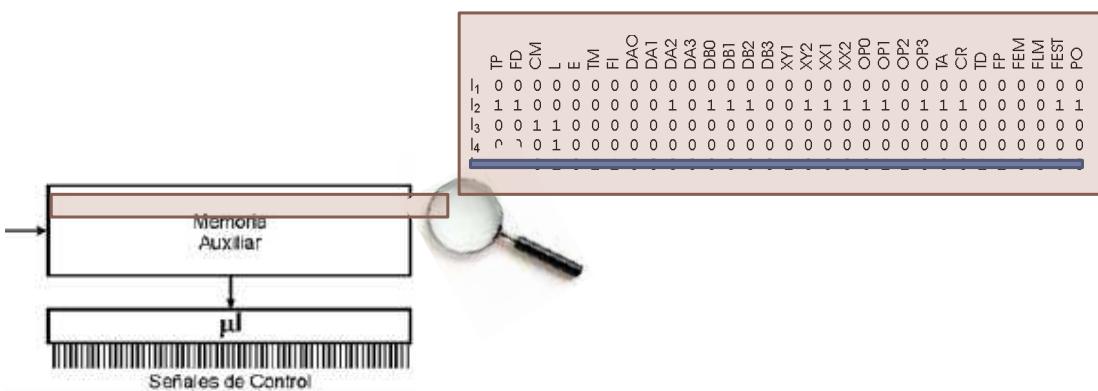
- ▶ Idea básica:  
Emplear una memoria (**memoria de control**)  
donde almacenar las señales de cada  
ciclo de cada instrucción.
- ▶ Características:
  - ▶ Fácil modificación
    - ▶ Actualización, ampliación, etc..
    - ▶ Ej.: Ciertas consolas, routers, etc.
  - ▶ Fácil tener instrucciones complejas
    - ▶ Ej.: Rutinas de diagnóstico, etc.
  - ▶ Fácil tener varios juegos de instrucciones
    - ▶ Se pueden emular otros computadores.
  - ▶ HW simple  $\Rightarrow$  difícil microcódigo



# Estructura general de una unidad de control microprogramada

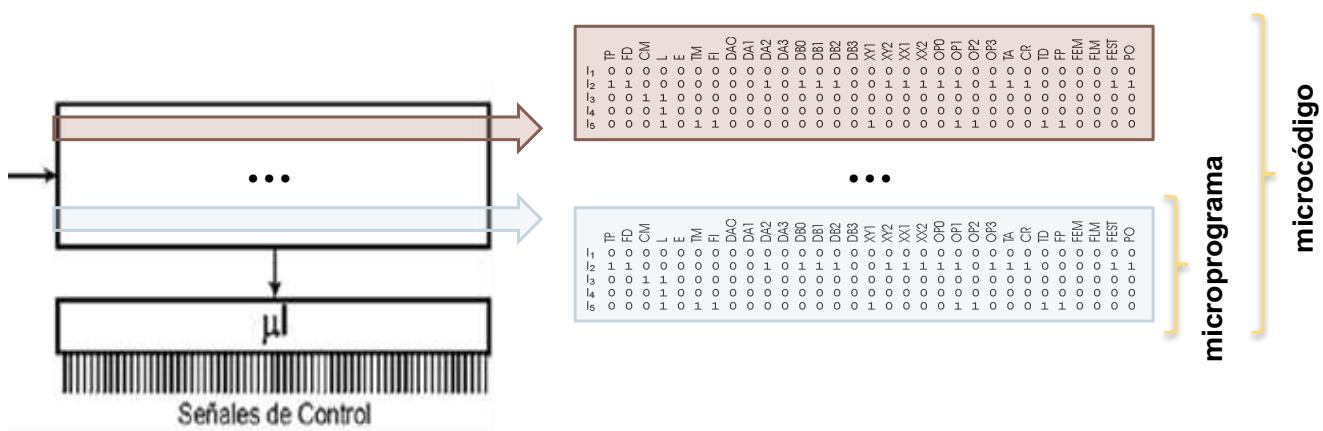


# Unidad de control almacenada. Microinstrucciones



- ▶ **Microinstrucción:** A cada palabra que define el valor de cada señal de control en un ciclo de una instrucción/fetch/CRI
- ▶ **Las microinstrucciones**
  - ▶ tienen un bit por cada señal de control.
  - ▶ cadena de 1's y 0's que representa el estado de cada señal de control durante un período de una instrucción.

# Unidad de control almacenada. Microprograma



- ▶ **Microprograma:** conjunto ordenado de microinstrucciones, que representan el cronograma de una instrucción máquina.
- ▶ **Microcódigo:** conjunto de los microprogramas de una máquina.

# Contenido de la memoria de control



- ▶ **FETCH: traer sig. Instrucción**
  - ▶ Ciclo Reconocimiento Int.
  - ▶  $IR \leftarrow \text{Mem}[PC]$ ,  $PC++$ , salto-a-C.O.
- ▶ **Microprograma:**  
uno por instrucción de ensamblador
  - ▶ Traer resto de operandos (si hay)
    - ▶ Actualizar PC en caso de más operandos
  - ▶ Realizar la instrucción
  - ▶ Salto a FETCH

# Estructura de la unidad de control microprogramada

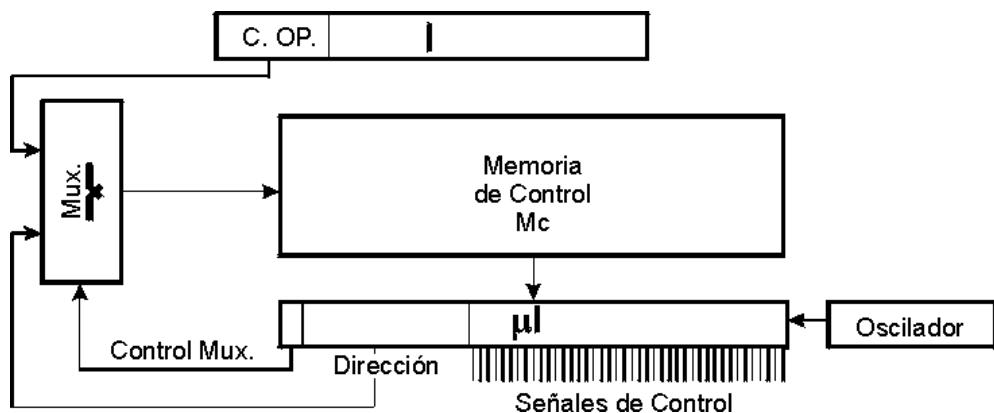
## ► Tres condiciones básicas:

1. Memoria de control suficiente para almacenar todos los microprogramas correspondientes a todas las instrucciones.
2. Procedimiento para asociar a cada instrucción su microprograma
  - ▶ Procedimiento que convierta el código de operación de la instrucción en la dirección de la memoria de control donde empieza su microprograma.
3. Mecanismo de secuenciación para ir leyendo las sucesivas microinstrucciones, y para bifurcar a otro microprograma cuando termina el que se está ejecutando.

## ► Dos alternativas:

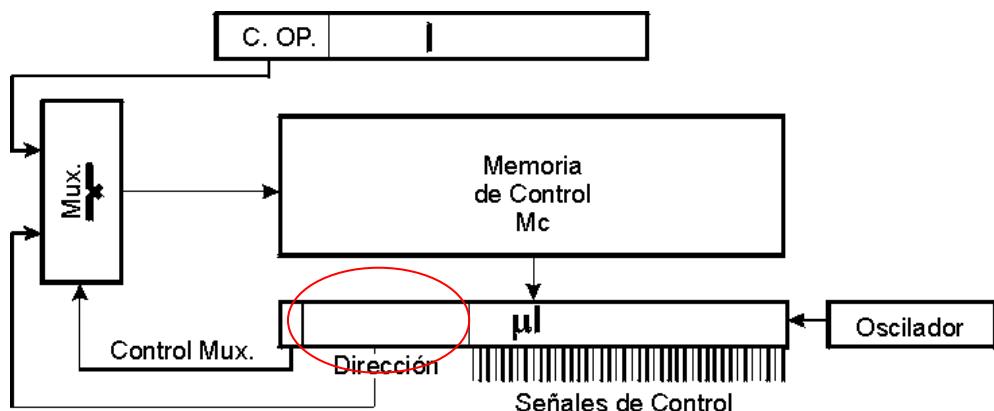
1. Secuenciamiento explícito.
2. Secuenciamiento implícito.

# Estructura de UC microprogramada con secuenciamiento explícito



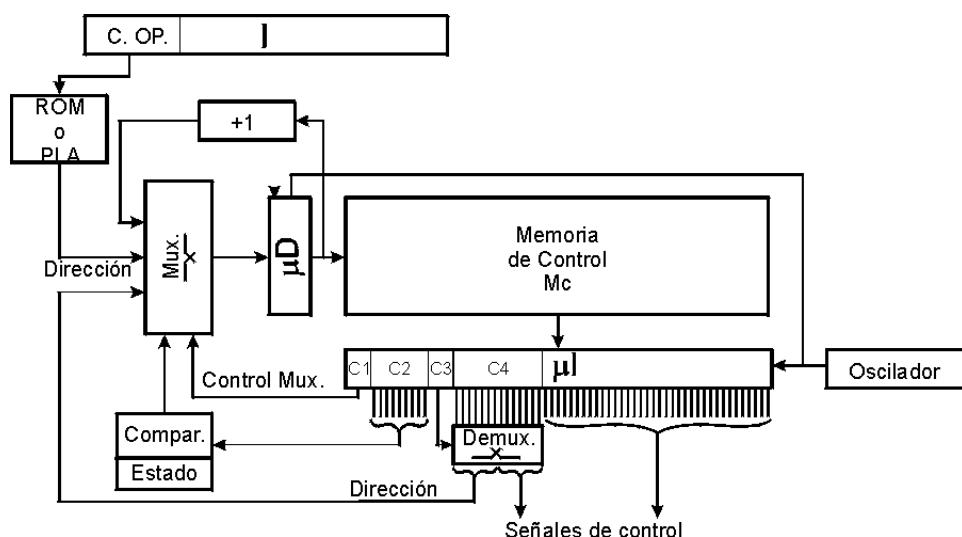
- ▶ Memoria de control guarda todos los  $\mu$ programas, donde cada  $\mu$ instrucción proporciona la  $\mu$ dirección de la  $\mu$ instrucción siguiente
- ▶ El CO representa la  $\mu$ Dirección de la primera  $\mu$ instrucción asociado a la instrucción máquina

## Estructura de UC microprogramada con secuenciamiento explícito



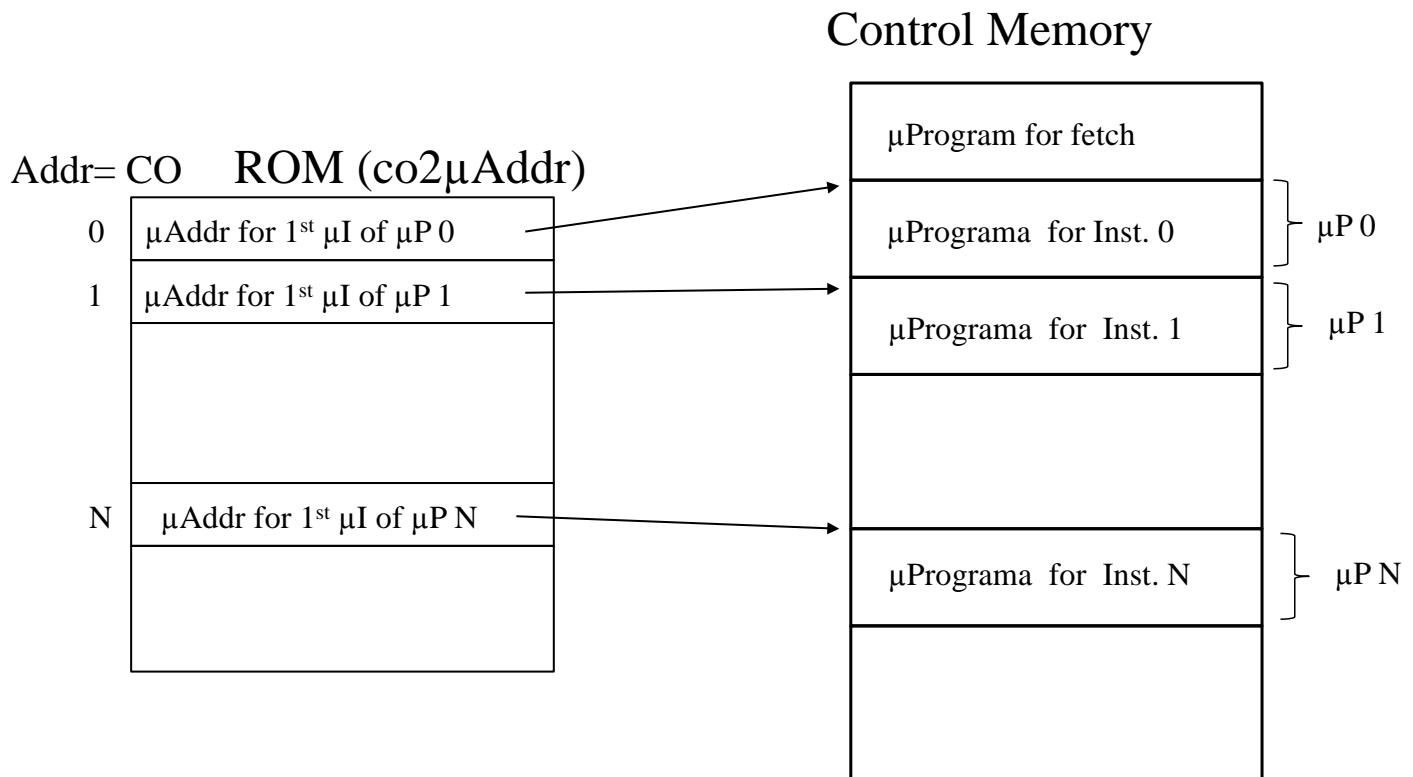
- ▶ Memoria de control guarda todos los  $\mu$ programas, donde cada  $\mu$ instrucción proporciona la  $\mu$ dirección de la  $\mu$ instrucción siguiente
- ▶ **Problema:** gran cantidad de memoria de control para el secuenciamiento de instrucciones, necesario almacenar la  $\mu$ dirección siguiente

# Estructura de U.C. microprogramada con secuenciamiento implícito

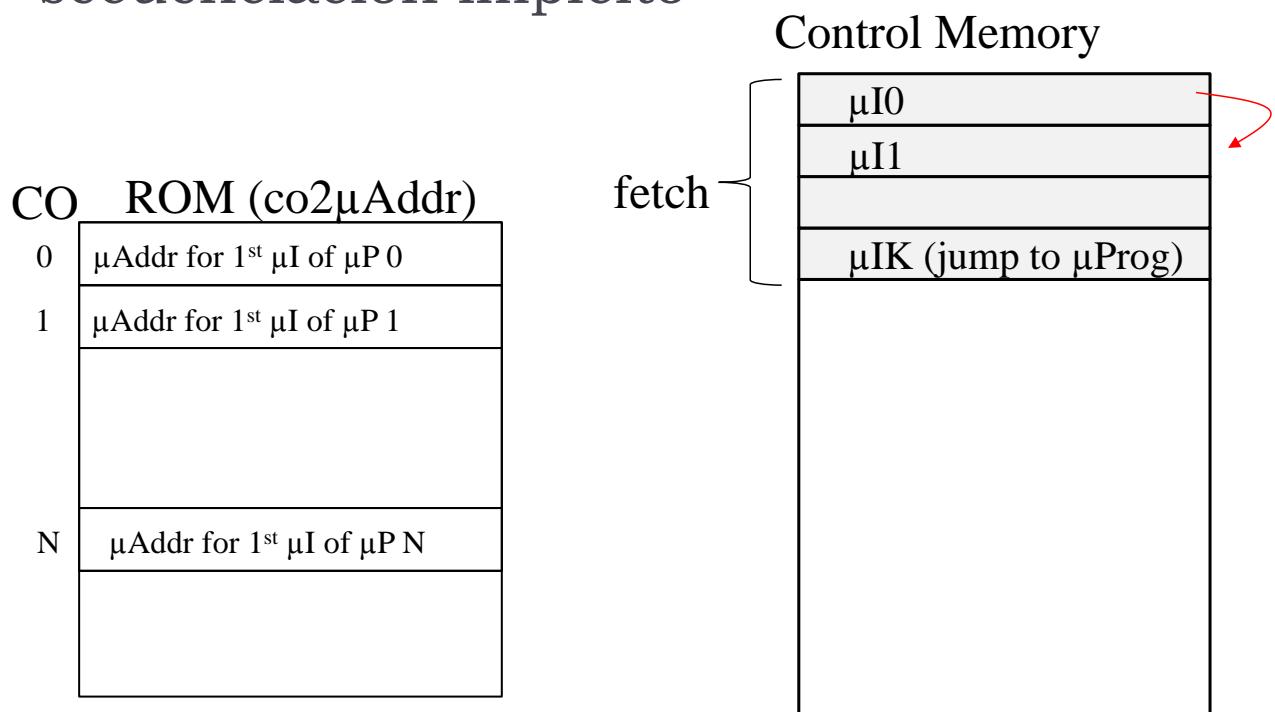


- ▶ Memoria de control guarda todos los microprogramas de forma consecutiva en la memoria de control
- ▶ La ROM/PLA asocia a cada instrucción su microprograma (primera μdirección)
- ▶ Siguiente μinstrucción (+1), μbifurcaciones condicionales o μbucles

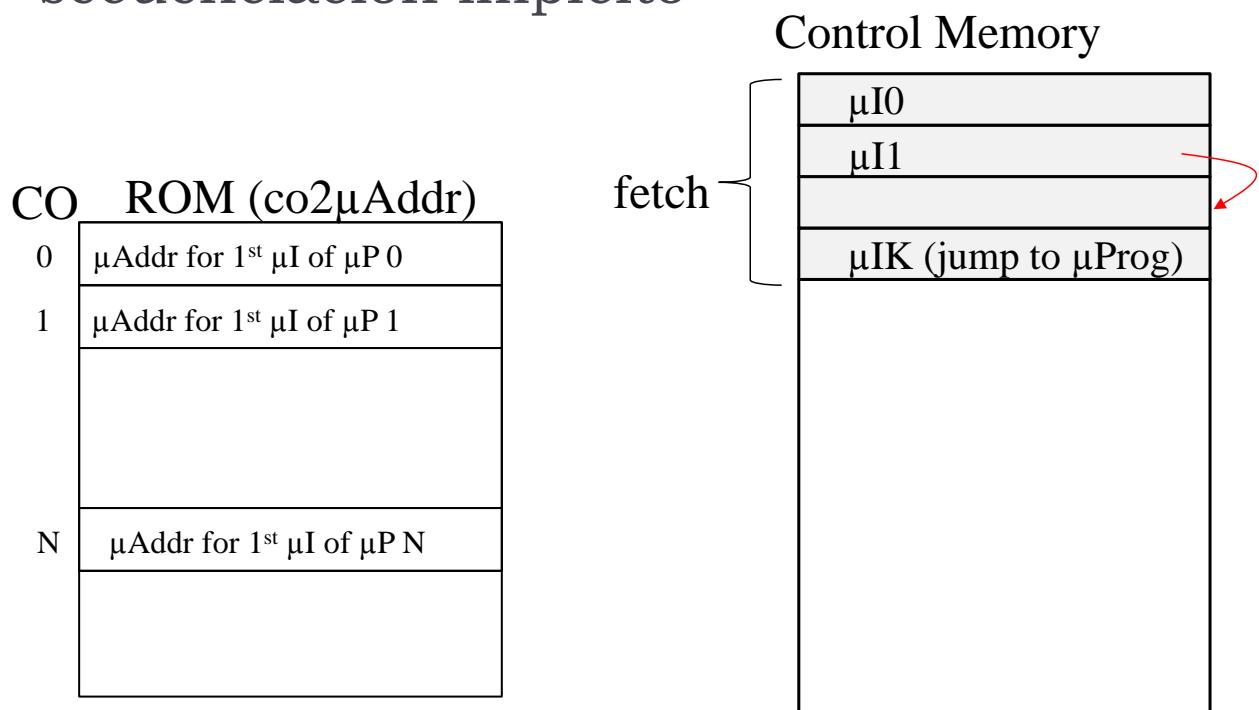
## Ejemplo de funcionamiento de una UC con secuenciación impícito



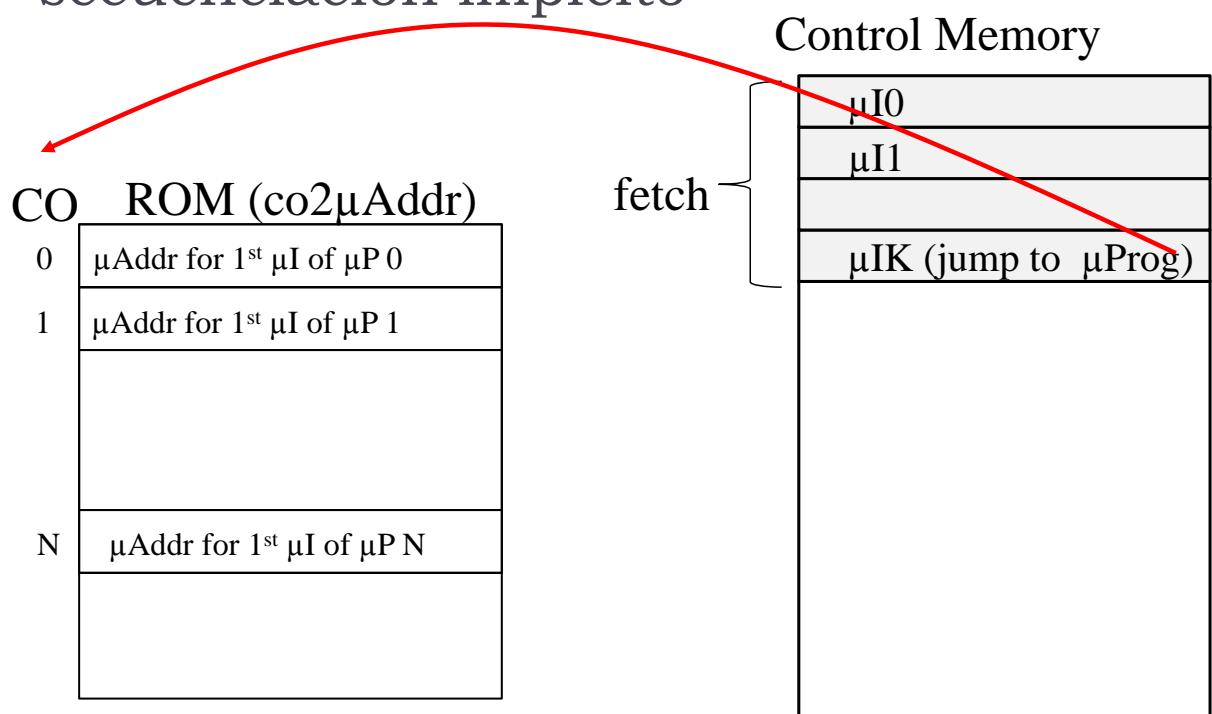
## Ejemplo de funcionamiento de una UC con secuenciación implícito



## Ejemplo de funcionamiento de una UC con secuenciación implícito

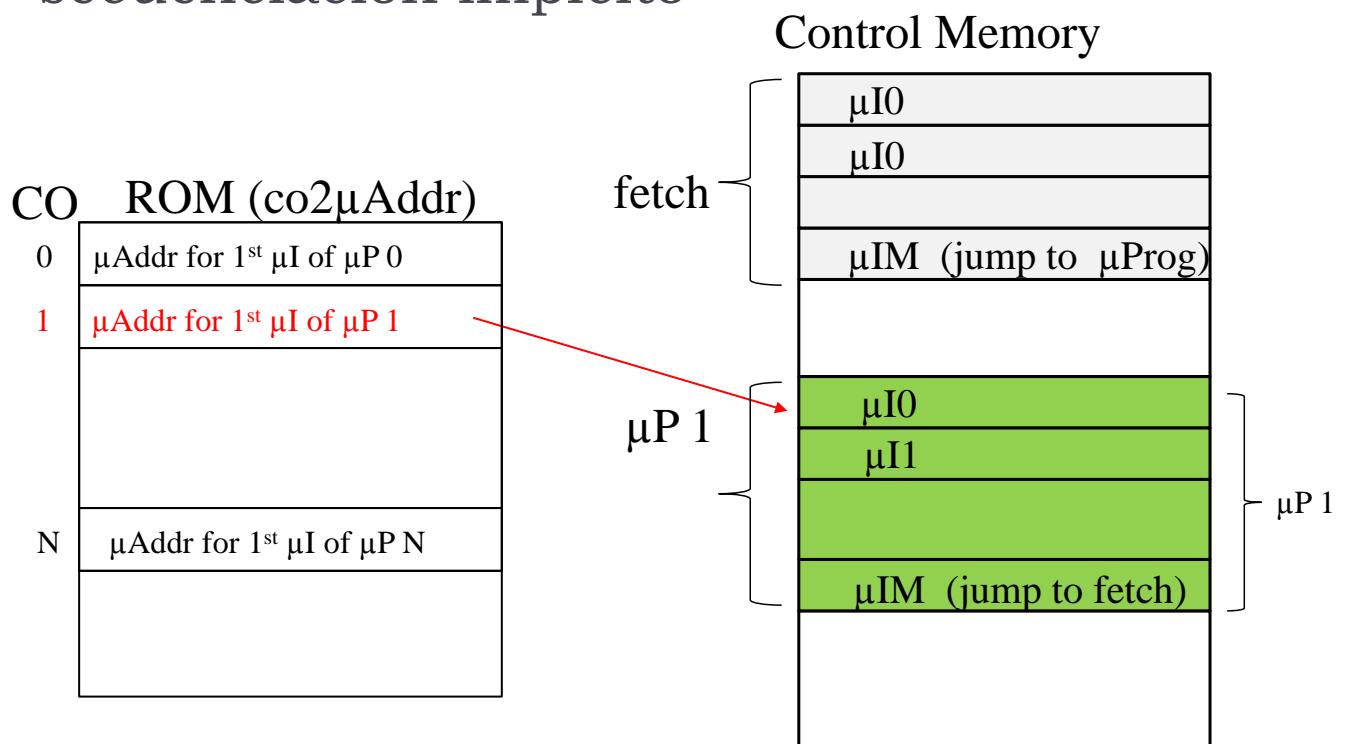


## Ejemplo de funcionamiento de una UC con secuenciación implícito

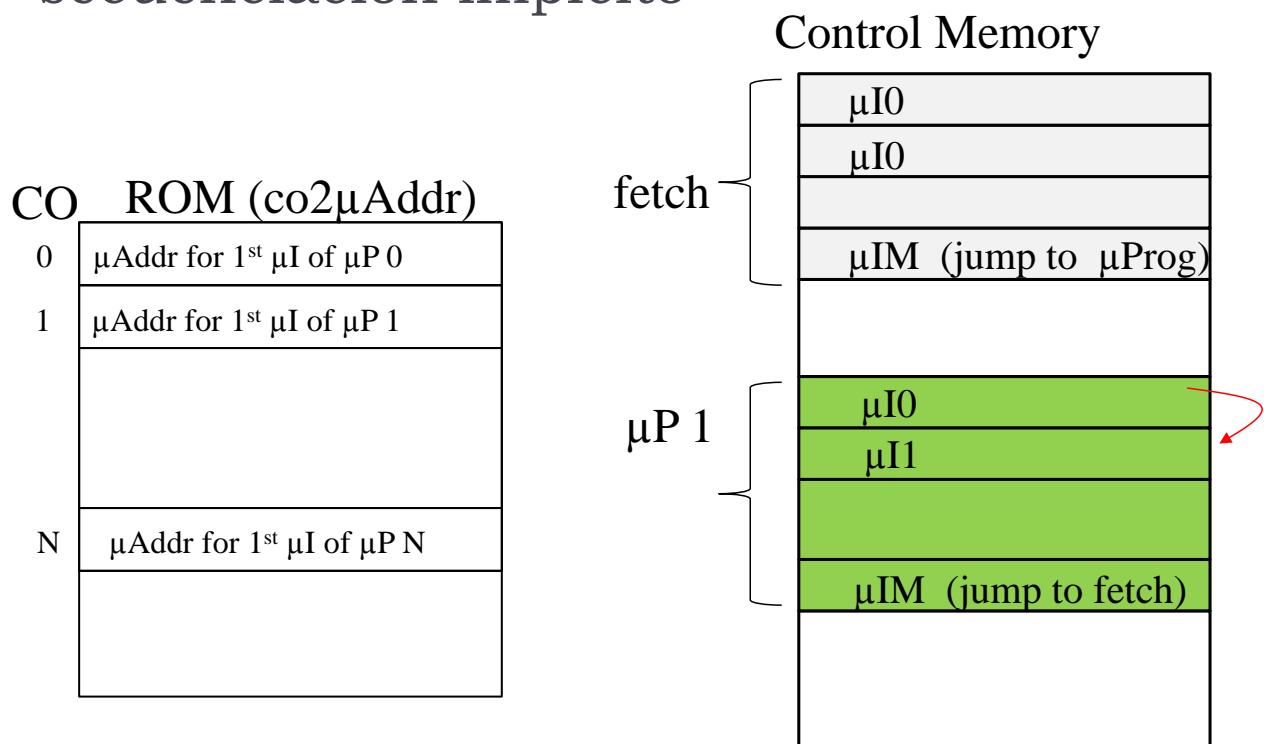


En el Registro de Instrucción el CO

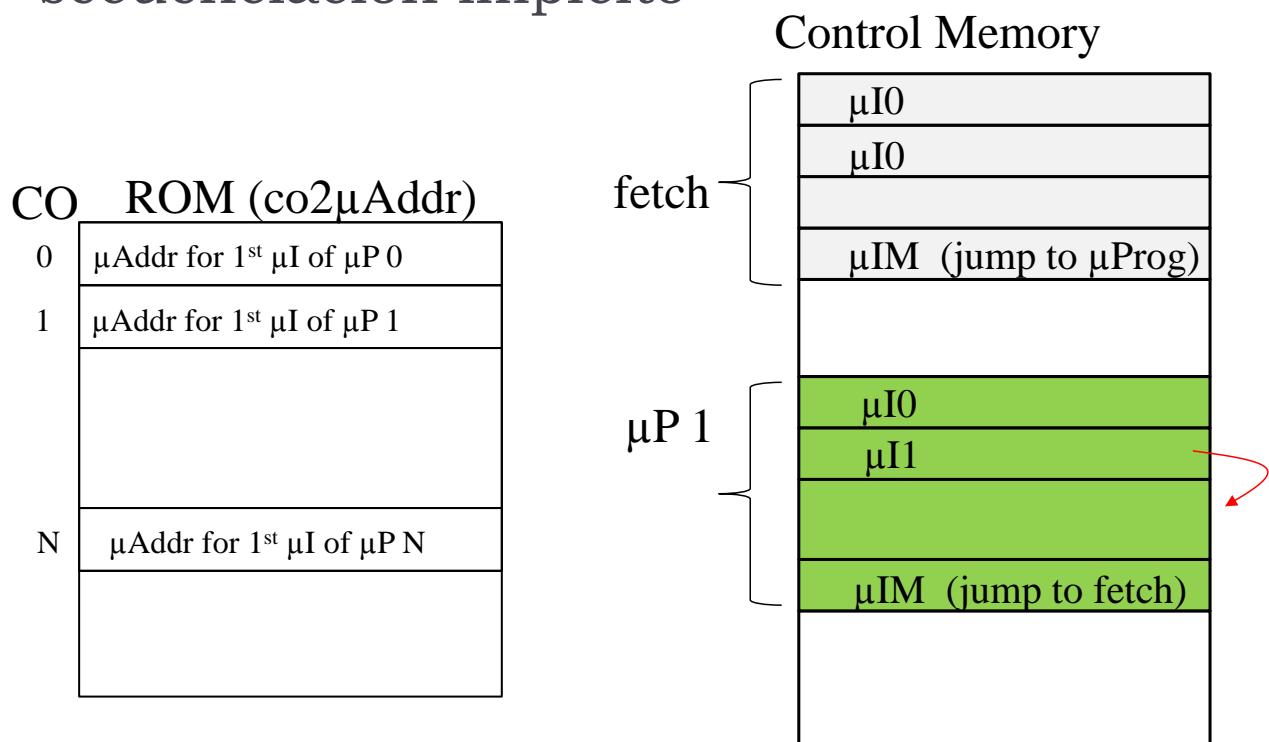
## Ejemplo de funcionamiento de una UC con secuenciación impícito



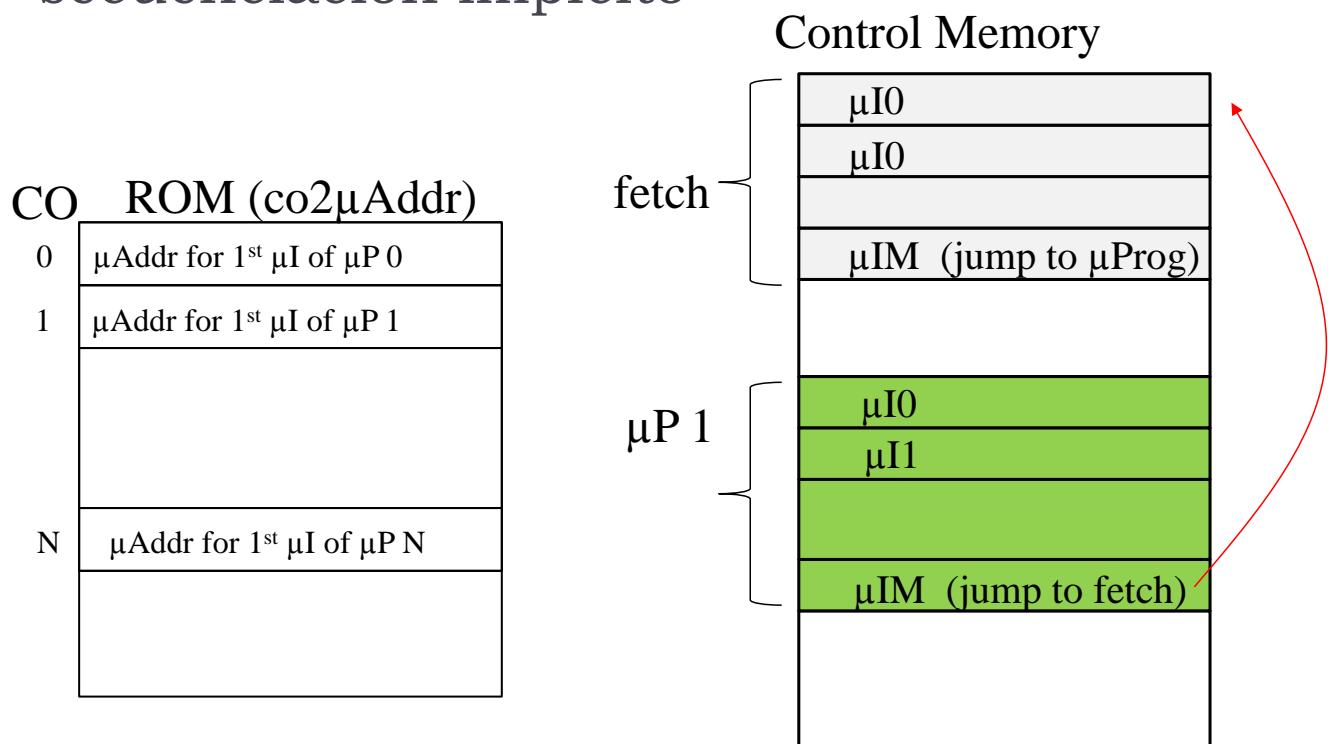
## Ejemplo de funcionamiento de una UC con secuenciación impícito



## Ejemplo de funcionamiento de una UC con secuenciación implícito



## Ejemplo de funcionamiento de una UC con secuenciación impícito



# Formato de las microinstrucciones

- ▶ **Formato de la microinstrucción:** especifica el nº de bits y el significado de cada uno de ellos.

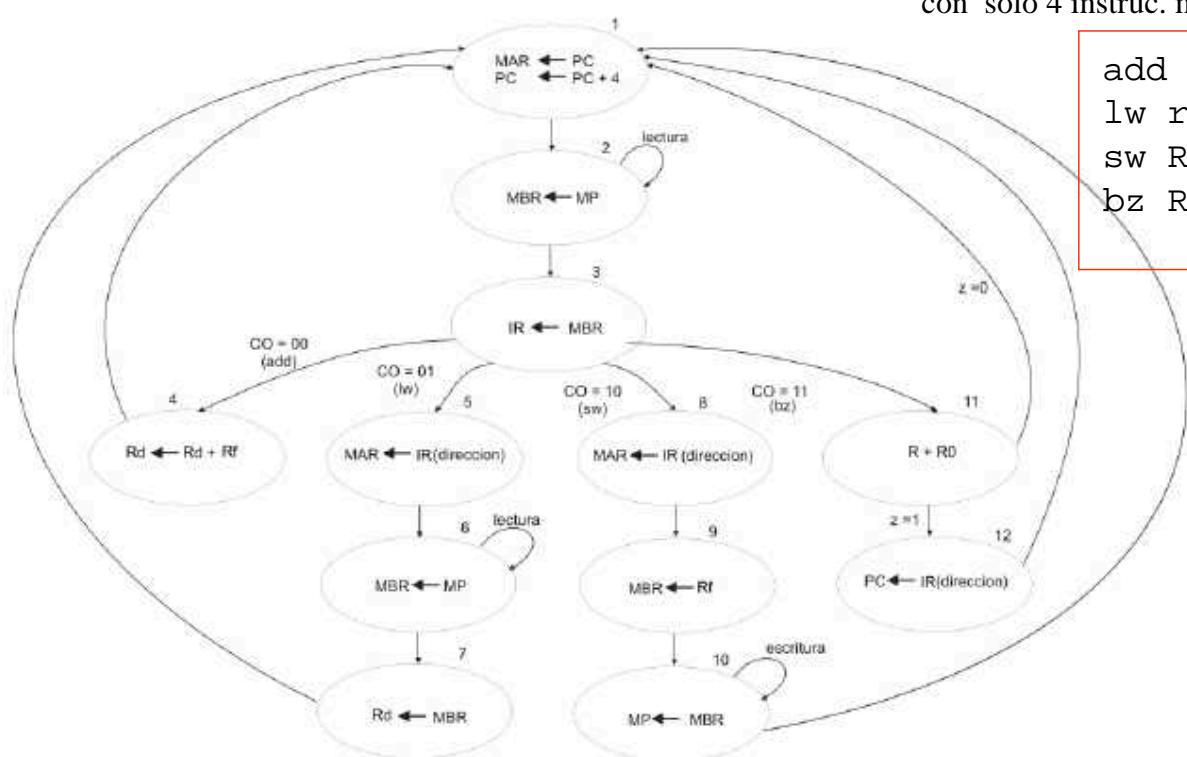


- ▶ Las señales se agrupan por **campos**:
  - ▶ Señales triestado de acceso a bus
  - ▶ Señales de gobierno de la ALU
  - ▶ Señales de gobierno del banco de registros
  - ▶ Señales de gobierno de la memoria
  - ▶ Señales de control de los multiplexores

# Máquina de estados del ejemplo

Ejemplo para un computador con solo 4 instruc. máquina

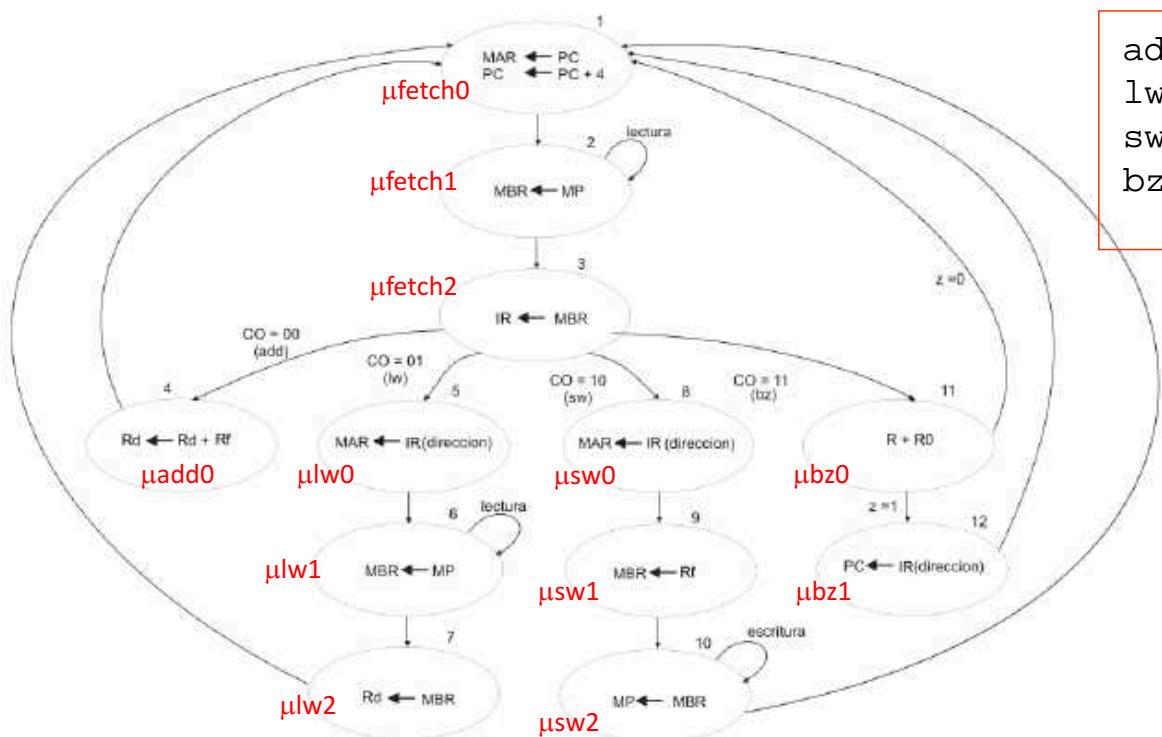
add rd, rf  
lw rd, dir  
sw Rf, dir  
bz R, dir



# Microinstrucciones para el ejemplo

Ejemplo para un computador con solo 4 instruc. máquina

add rd, rf  
 lw rd, dir  
 sw Rf, dir  
 bz R, dir



## Microdódigo para el ejemplo

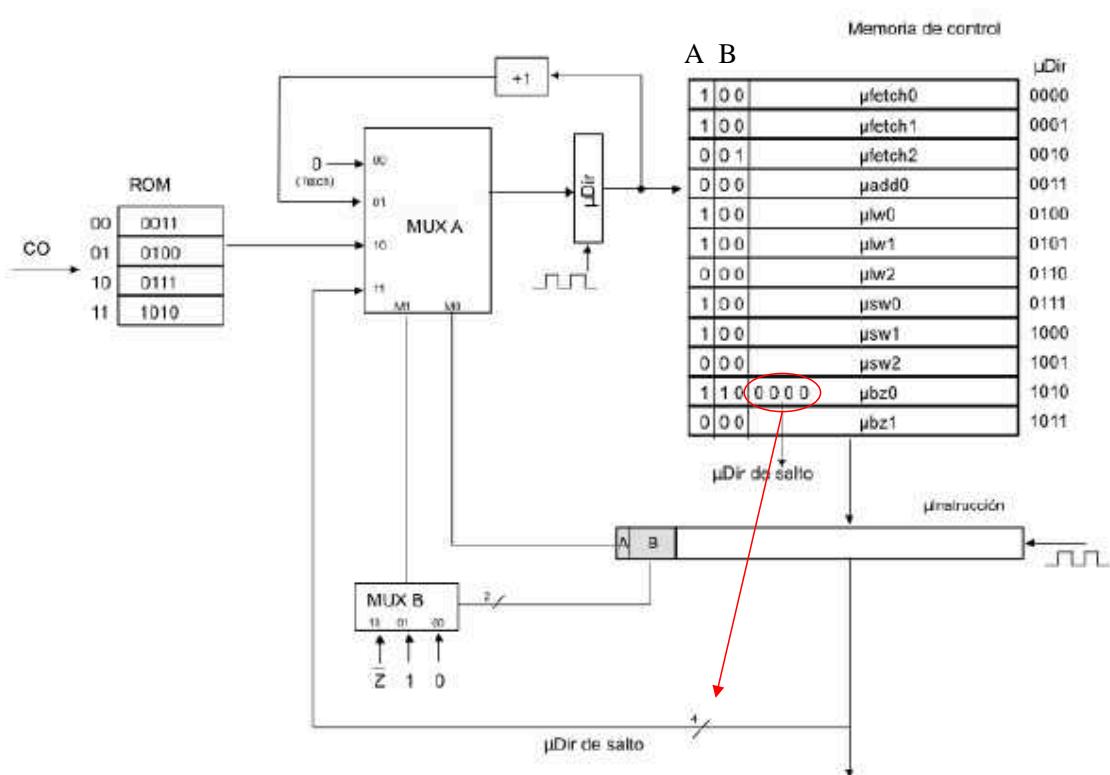
```

add r1, r2
lw r1, dir
bz dir
sw r1

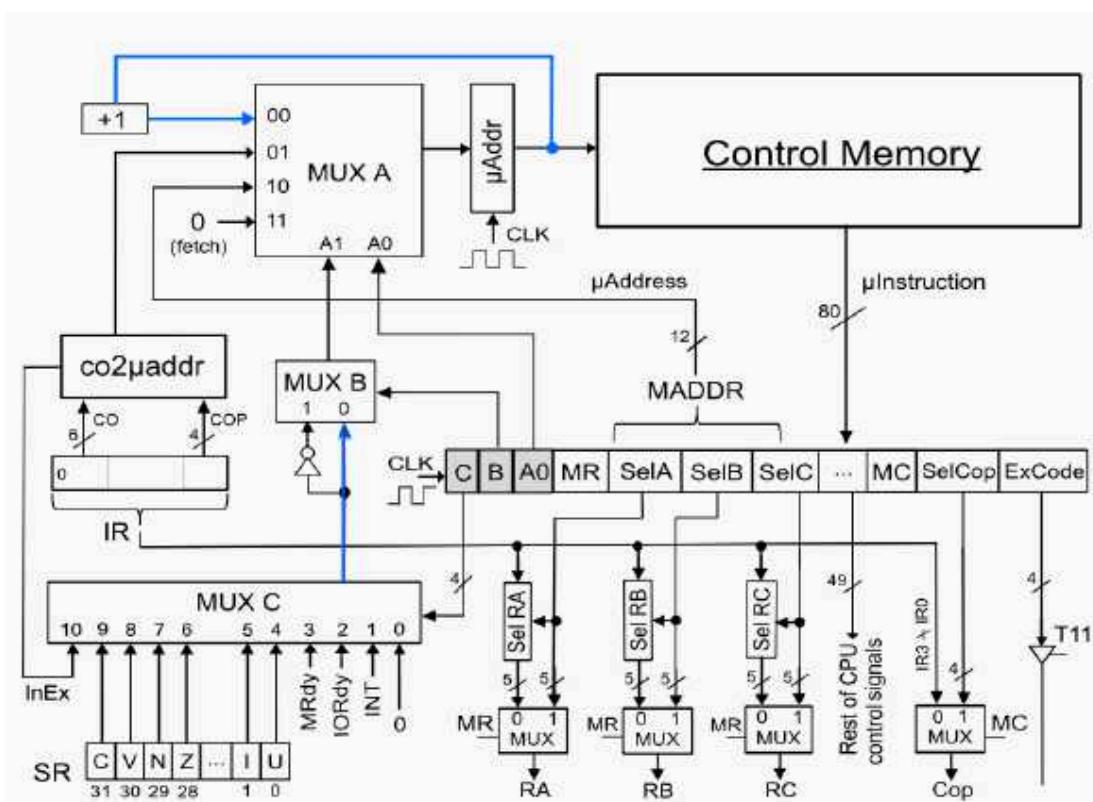
```

	C0	C1	C2	C3	C4	C5	C6	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	LE	MA	MBI	MB0	M1	M2	M7	R	W	Ta	Td
ufetch0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
ufetch1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
ufetch2	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
μadd0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
μlw0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
μlw1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
μlw2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
μsw0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
μsw1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
μsw3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
μbz0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
μbz1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

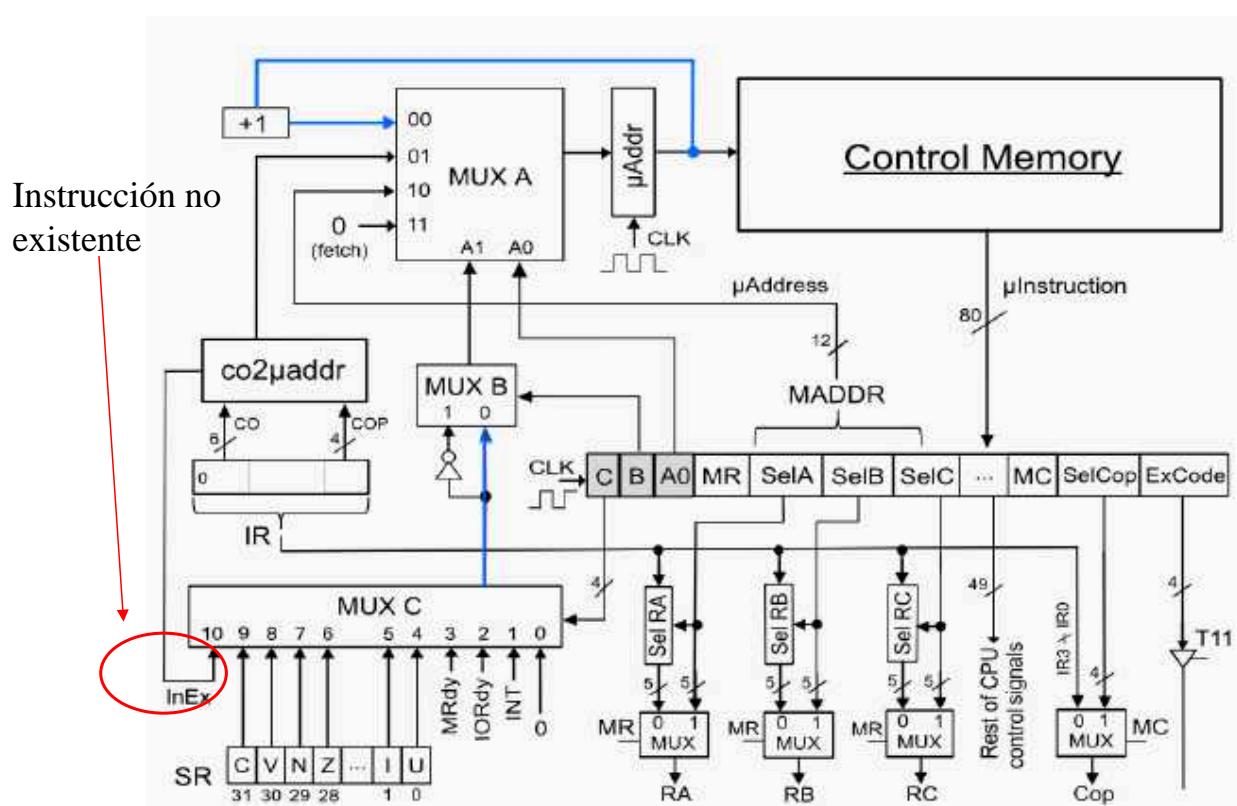
## Ejemplo de unidad de control microprogramada para el ejemplo



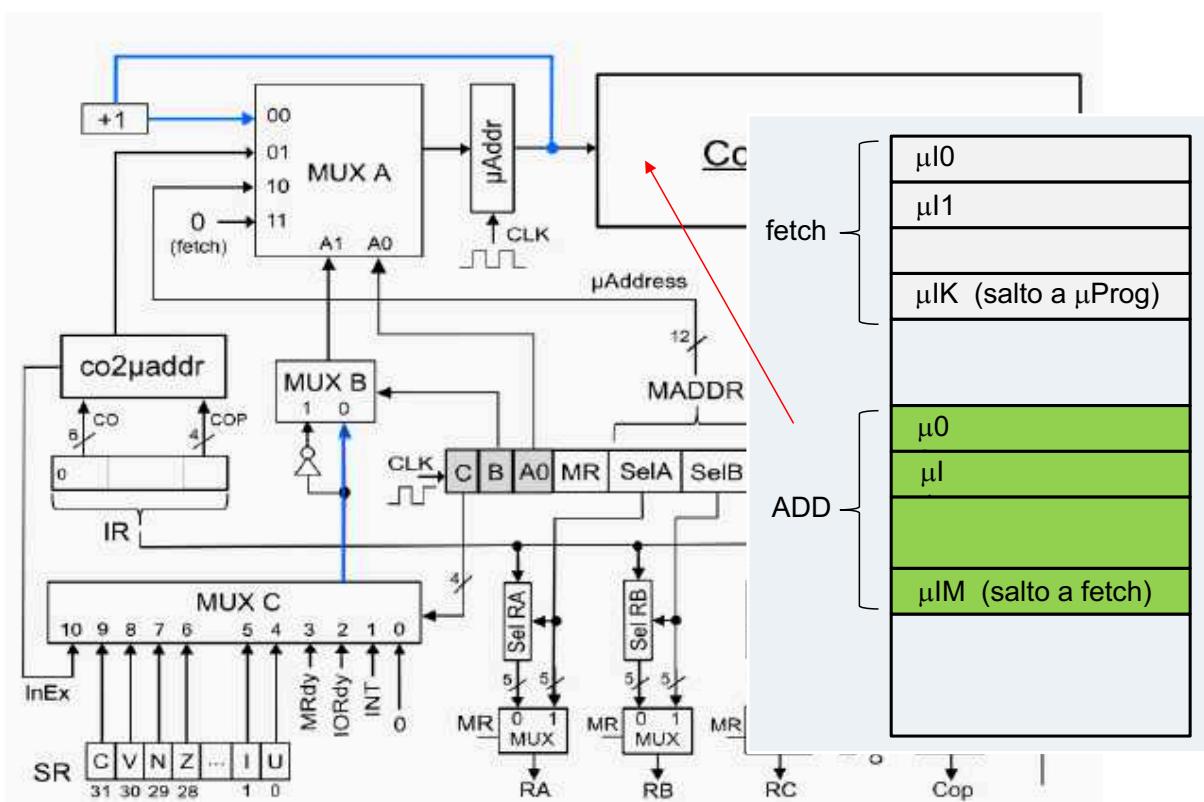
# Unidad de control de WepSIM



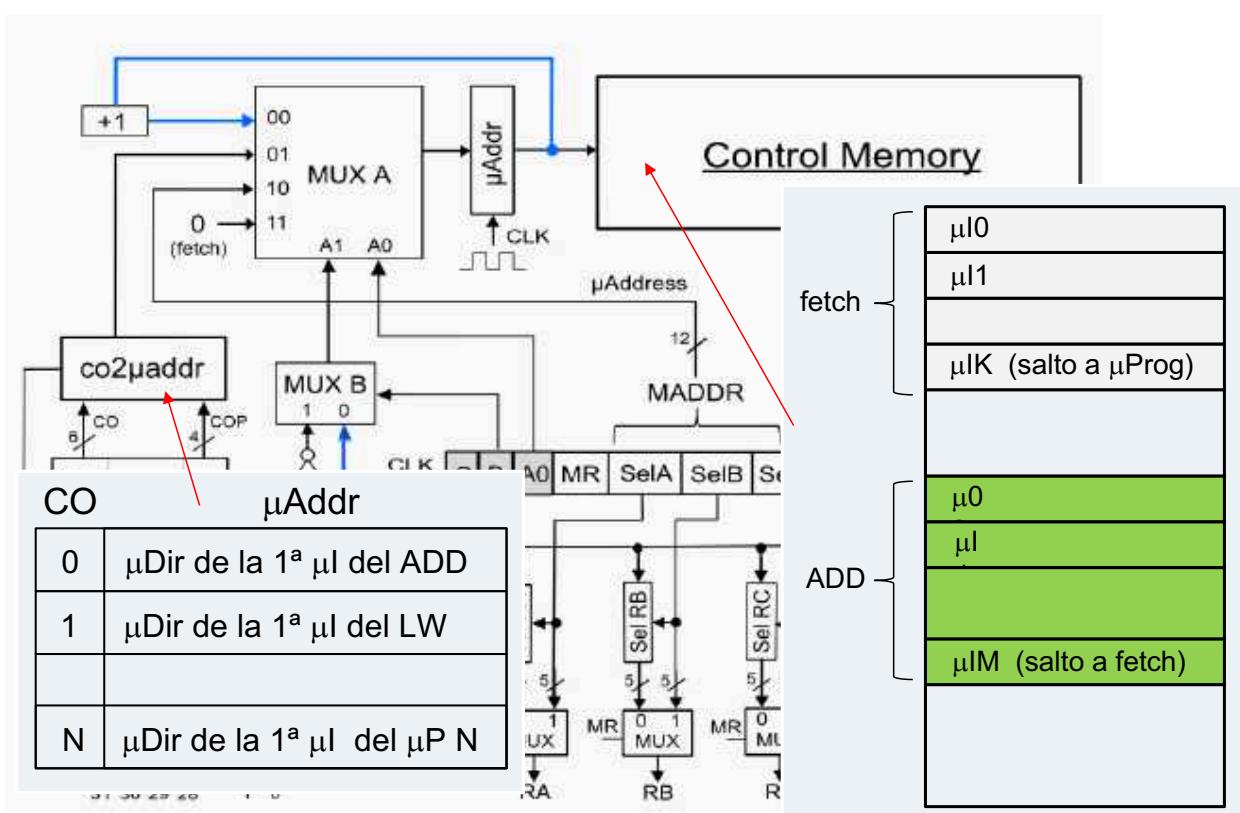
# Unidad de control de WepSIM



# Unidad de control de WepSIM

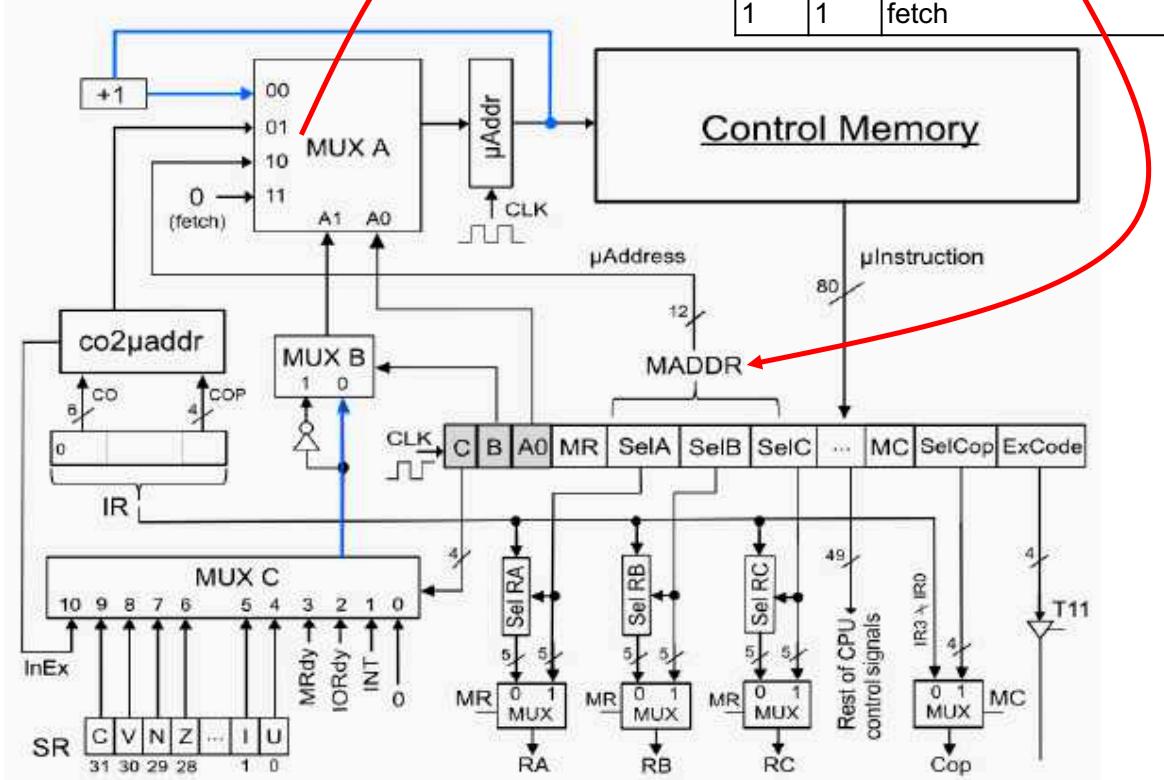


# Unidad de control de WepSIM



## Unidad de control de WepSIM

A1	A0	Salida
0	0	Sig $\mu$ Dir
0	1	Salto a $\mu$ Prog. (ROM)
1	0	$\mu$ Dir de salto
1	1	fetch



# Unidad de control de WepSIM

A0	B	C3	C2	C1	C0	Acción
0	0	0	0	0	0	Siguiente μDirección
0	I	0	0	0	0	Salto incondicional a MADDR
0	0	0	0	0	I	Salto condicional a MADDR si INT = I (*)
0	I	0	0	I	0	Salto condicional a MADDR si IORdy = 0 (*)
0	I	0	0	I	I	Salto condicional a MADDR si MRdy = 0 (*)
0	0	0	I	0	0	Salto condicional a MADDR si U = I (*)
0	0	0	I	0	I	Salto condicional a MADDR si I = I (*)
0	0	0	I	I	0	Salto condicional a MADDR si Z = I (*)
0	0	0	I	I	I	Salto condicional a MADDR si N = I (*)
0	0	I	0	0	0	Salto condicional a MADDR si O = I (*)
I	0	0	0	0	0	Salto a μProg. (ROM c02μaddr)
I	I	0	0	0	0	Salto a fetch (μDir = 0)

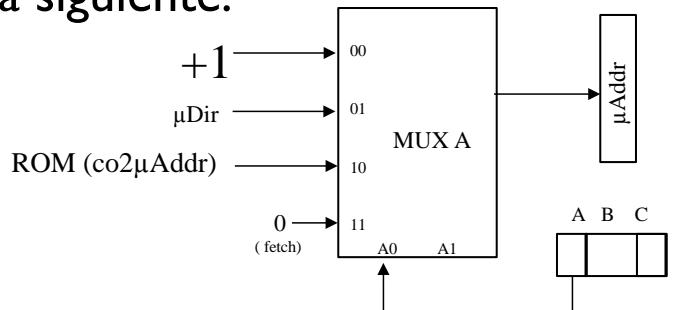
- ▶ (\*) Si no se cumple la condición → Siguiente μDirección
- ▶ Resto de entradas → funcionamiento indefinido

## Ejemplo

- ▶ Salto a la  $\mu$ Dirección 000100011100 (12 bits) si Z = 1.

En caso contrario se salta a la siguiente:

- ▶ A0 = 0
- ▶ B= 0
- ▶ C = 0110
- ▶  $\mu$ Addr = 000100011100



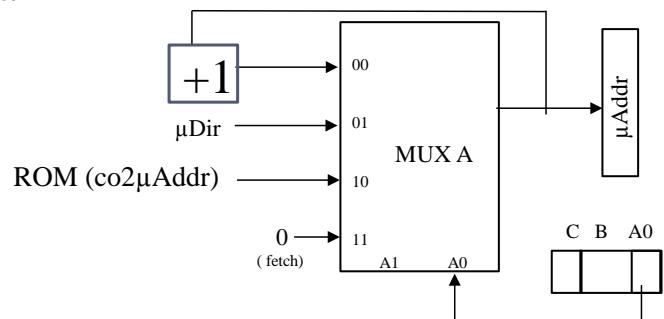
- ▶ Salto incondicional a la  $\mu$ Dirección 000100011111

- ▶ A0 = 0
- ▶ B= 1
- ▶ C = 0000
- ▶  $\mu$ Addr = 000100011111

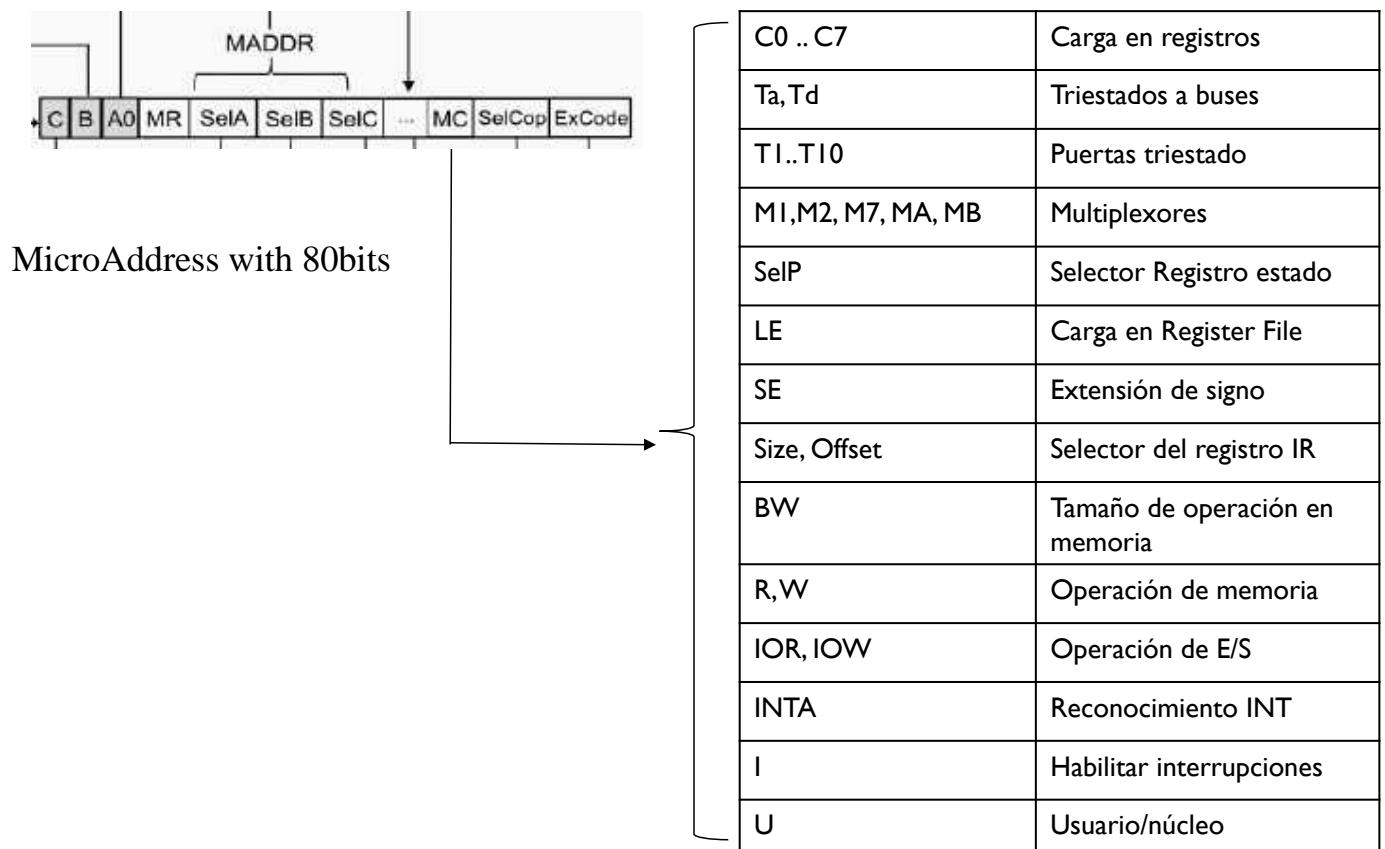
$\mu$ Dirección codificada  
en los bits 72-61 de la  
 $\mu$ Instrucción

# Ejemplo

- ▶ En la última  $\mu$ Instrucción del fetch saltar a la primera  $\mu$ Dirección del  $\mu$ Programa asociado al código de operación leído de memoria
  - ▶  $A_0 = 1$
  - ▶  $B = 0$
  - ▶  $C = 0000$



## Formato de la microinstrucción



## Ejemplo

### operaciones elementales con la UC

- ▶ **Salto a la dirección 000100011100 (12 bits) si Z = 1.**  
**En caso contrario se salta a la siguiente.**

O. Elemental	Señales
Si (Z) $\mu\text{PC}=000100011100$	A0=0, B=0, C=0110 <sub>2</sub> , mADDR=000100011100 <sub>2</sub>

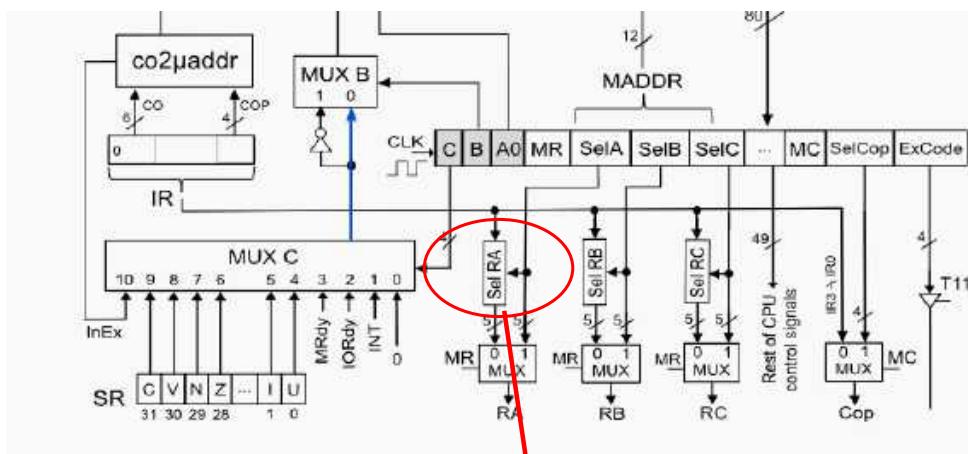
- ▶ **Salto incondicional a la dirección 000100011111**

O. Elemental	Señales
$\mu\text{PC}=000100011111$	A0=0, B=1, C=0000 <sub>2</sub> , mADDR=000100011111 <sub>2</sub>

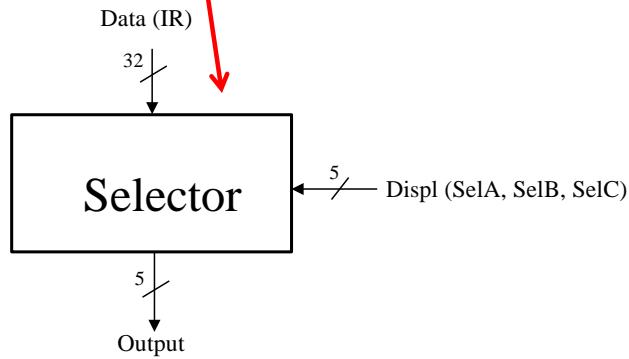
- ▶ **Salto a la primera  $\mu$ dirección del  $\mu$ programa asociado al CO**

O. Elemental	Señales
Salto a CO	A0=1, B=0, C=0000 <sub>2</sub>

# Selector de registros

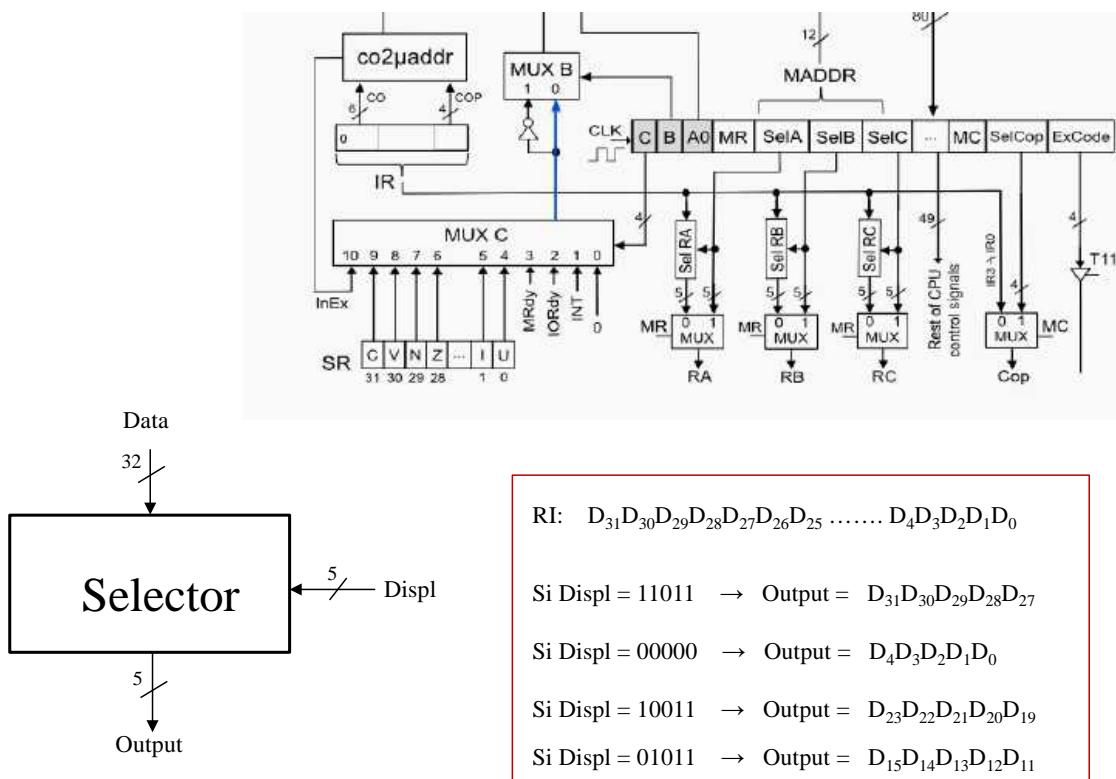


Selecciona 5 bits de un conjunto de 32 bits desde la posición indicada en Displ (bit inferior)

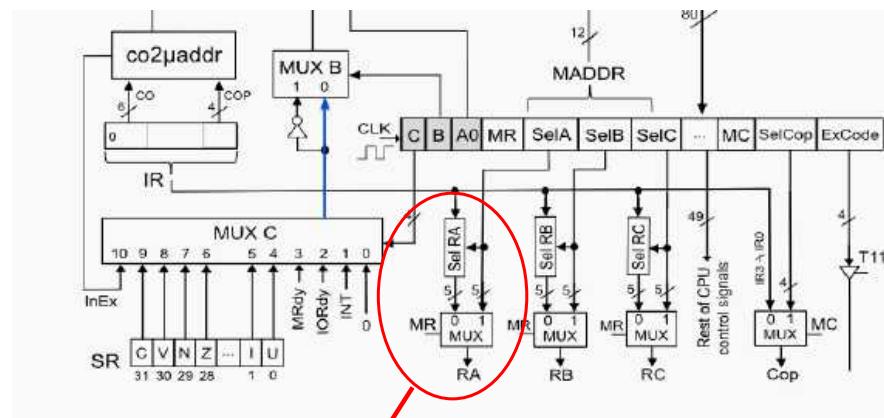


# Selector de registros

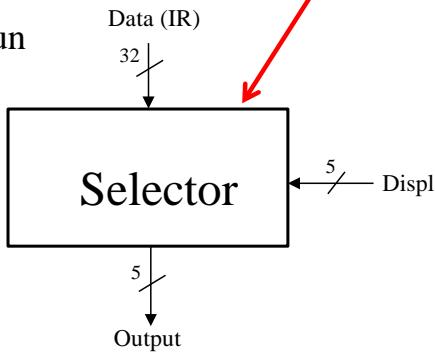
## Ejemplo



# Selector de registros



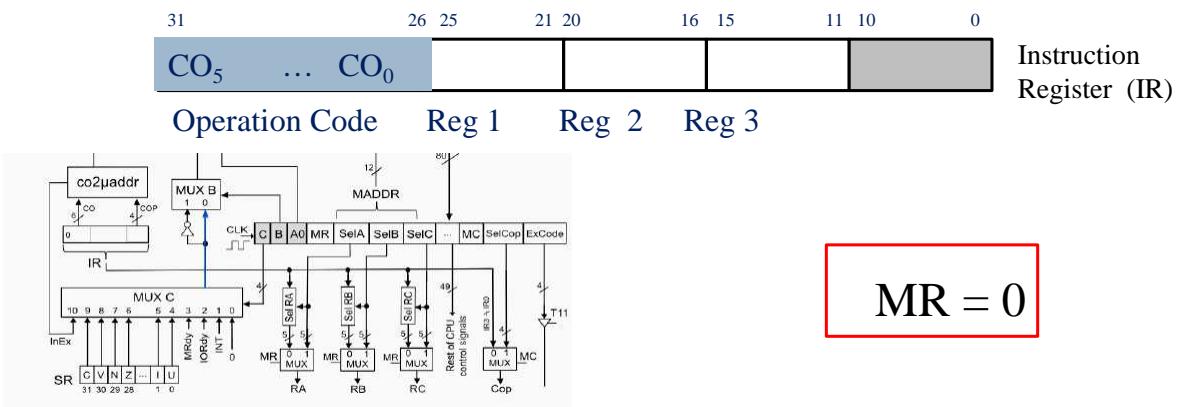
Selecciona 5 bits de un conjunto de 32 bits desde la posición indicada en Displ



- Si  $MR = 1$ , RA se obtiene directamente de la  $\mu$ Instrucción
- Si  $MR = 0$ , RA se obtiene de un campo de la instrucción (en IR)

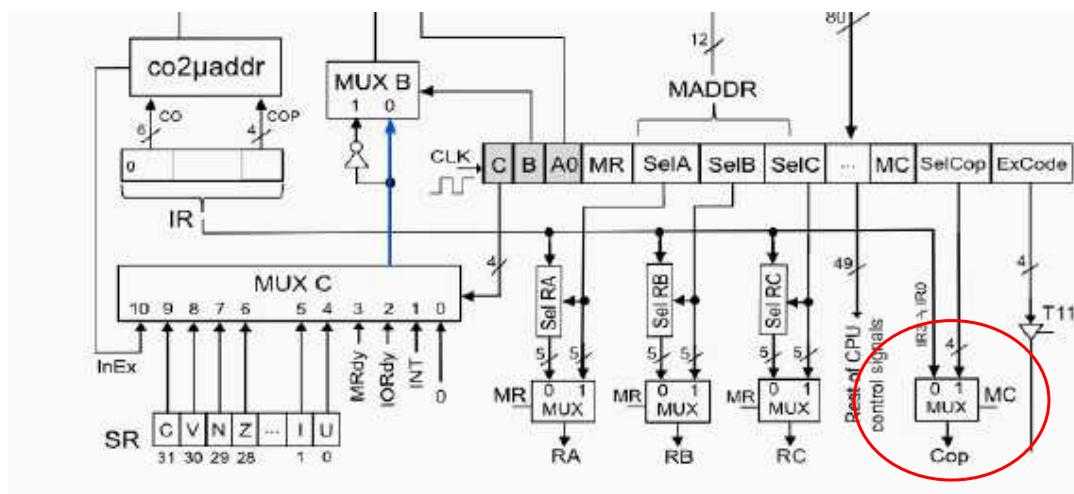
# Selector de registros

- ▶ Si el formato de una instrucción almacenada en IR es:



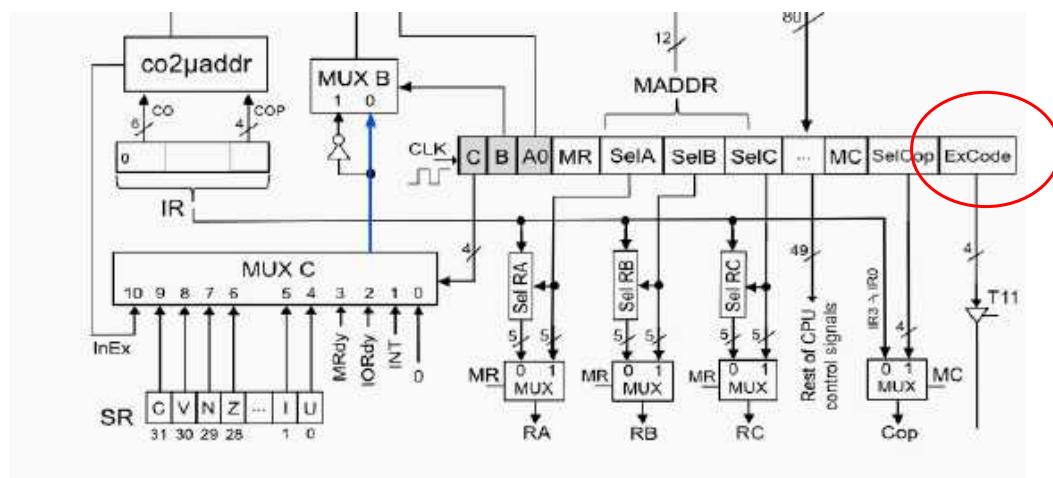
- ▶ Si se quiere seleccionar el campo con el Reg 2 en la puerta B del banco de registros → SelB = 10000 (RB se obtiene de los bits 20...16 del IR)
- ▶ Si se quiere seleccionar el campo con el Reg 3 en la puerta A del banco de registros → SelA = 01011 (RA se obtiene de los bits 15...11 del IR)
- ▶ Si se quiere seleccionar el campo con el Reg 1 en la puerta C del banco de registros → SelC = 10101 (RC se obtiene de los bits 25...21 del IR)

## Selección del código de operación de la ALU



- Si  $MC = 1$ , el código de operación de la ALU se obtiene directamente de la microinstrucción (**SelCop**)
- Si  $MC = 0$ , el código de operación de la ALU se obtiene de los cuatro últimos bits almacenados en el registro de instrucción

## Código de excepción



- ExCode: vector de interrupción a utilizar cuando se produce una excepción en la ejecución de la instrucción.

# Ejemplo

## ▶ Instrucciones a microprogramar con WepSIM\*:

Instrucción	Cód. Oper.	Significado
ADD Rd, Rf1, Rf2	000000	$Rd \leftarrow Rf1 + Rf2$
LI R, valor	000001	$R \leftarrow \text{valor}$
LW R, dir	000010	$R \leftarrow MP[\text{dir}]$
SW R, dir	000011	$MP[\text{dir}] \leftarrow R$
BEQ Rf1, Rf2, despl	000100	if ( $Rf1 == Rf2$ ) $PC \leftarrow PC + \text{desp}$
J dir	000101	$PC \leftarrow \text{dir}$
HALT	000110	Parada, bucle infinito

\* Memoria de un ciclo

# Microprograma de las instrucciones

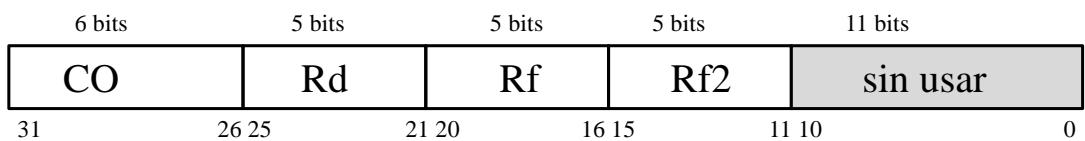
## ▶ FETCH

Ciclo	Op. Elemental	Señales activadas (resto a 0)	C	B	A0
0	MAR $\leftarrow$ PC	T2, C0	0000	0	0
1	MBR $\leftarrow$ MP	Ta, R, BW = 11, CI, MI	0000	0	0
	PC $\leftarrow$ PC + 4	M2, C2	0000	0	0
2	IR $\leftarrow$ MBR	T1, C3	0000	0	0
3	Decodificación		0000	0	1

# Microprograma de las instrucciones

## ▶ ADD Rd, Rf1, Rf2

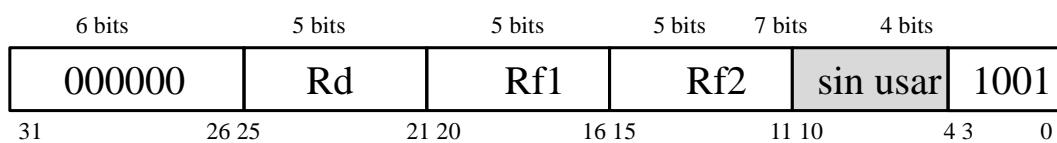
Ciclo	Op. Elemental	Señales activadas (resto a 0)	C      B    A0
0	$Rd \leftarrow Rf1 + Rf2$	$Cop = 1010$ $SelP=11, C7, M7$ $T6, LC$ $SelA = 10000 (16)$ $SelB = 01011 (11)$ $SelC = 10101 (21)$	0000    I    I



## Microprograma de las instrucciones (otra)

### ► ADD Rd, Rf1, Rf2

Ciclo	Op. Elemental	Señales activadas (resto a 0)	C      B    A0
0	$Rd \leftarrow Rf1 + Rf2$	SelCop = 1010, MC SelP=11, C7, M7 T6, LC SelA = 10000 (16) SelB = 01011 (11) SelC = 10101 (21)	0000    I    I



# Microprograma de las instrucciones

## ► LI R, valor

Ciclo	Op. Elemental	Señales activadas (resto a 0)	C      B      A0
0	$R \leftarrow IR$ (valor)	LC SelC = 10101 (21) T3, Size = 10000 Offset= 00000 SE=1	0000    I    I

6 bits                  5 bits                  5 bits                  16 bits

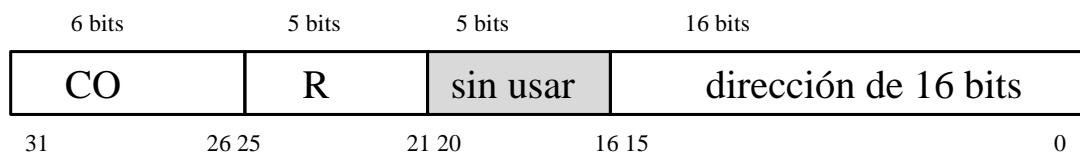
CO	R	sin usar	número de 16 bits
----	---	----------	-------------------

31                  26 25                  21 20                  16 15                  0

# Microprograma de las instrucciones

- ▶ LW R dir, con memoria síncrona de un ciclo

Ciclo	Op. Elemental	Señales activadas (resto a 0)	C    B    A0
0	MAR $\leftarrow$ IR (dir)	T3, C0 Size = 10000, Offset= 00000	0000    0    0
1	MBR $\leftarrow$ MP[MAR]	Ta, R, BW = 11, CI, MI	0000    0    0
2	R $\leftarrow$ MBR	T1, LC, SelC = 10101	0000    1    1



# Microprograma de las instrucciones

- ▶ LW R dir, con memoria asíncrona (MRdy=1 indica el fin)

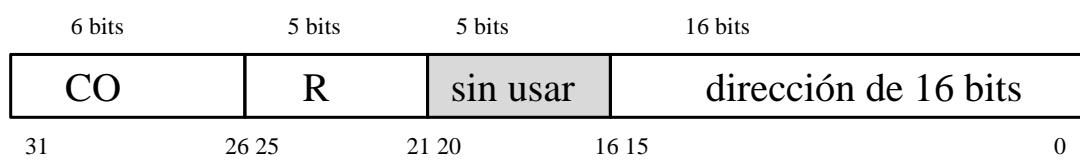
Ciclo	Op. Elemental	Señales activadas (resto a 0)	C      B      A0
0	MAR $\leftarrow$ IR (dir)	T3, C0 Size = 10000, Offset= 00000	0000    0    0
1	while (!MRdy) MBR $\leftarrow$ MP[MAR]	Ta, R, BW =11, CI, MI, MADDR= $\mu$ Add de esta microinstrucción	0011    1    0
2	R $\leftarrow$ MBR	T1, LC, SelC = 10101	0000    1    1

Se ejecuta esta microinstrucción mientras MRdy==0

# Microprograma de las instrucciones

- ▶ SW R dir, con memoria síncrona de un ciclo

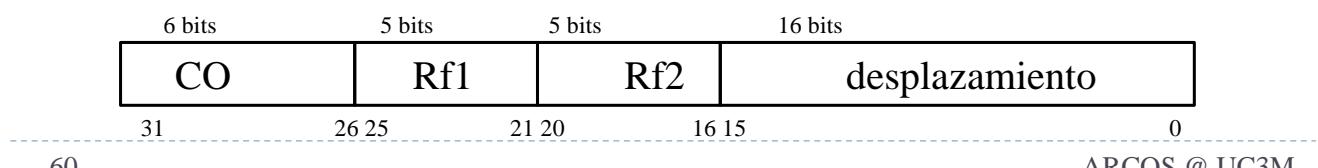
Ciclo	Op. Elemental	Señales activadas (resto a 0)	C    B    A0
0	MBR $\leftarrow$ R	T9, C1, SelA=10101	0000 0 0
1	MAR $\leftarrow$ IR(dir)	T3, C0, Size = 10000, offset= 00000	0000 0 0
2	MP[dir] $\leftarrow$ MBR	Td, Ta, BW = 11, W	0000 1 1



# Microprograma de las instrucciones

## ► BEQ Rf1, Rf2, desp

Ciclo	Op. Elemental	Señales activadas (resto a 0)	C      B    A0
0	Rf1 - Rf2	SelCop = 1011, MC, C7, M7 SelP = 11, SelA = 10101 SelB = 10000	0000  0  0
II	If ( $Z == 0$ ) goto fetch else next	MADDR = 0	0110  1  0
2	RT1 $\leftarrow$ PC	T2, C4	0000  0  0
3	RT2 $\leftarrow$ IR (dir)	Size = 10000 Offset = 00000, T3,C5	0000  0  0
4	PC $\leftarrow$ RT1 + RT2	SelCop = 1010, MC, MA, MB=01, T6,C2,	0000  1  1



# Microprograma de las instrucciones

▶ J dir

Ciclo	Op. Elemental	Señales activadas (resto a 0)	C    B    A0
0	PC ← IR (dir)	C2,T3, size = 10000, offset= 00000	0000   I   I



# Especificación de los microprogramas en WepSIM

**Lista de microcódigos  
especificación de registros  
pseudoinstrucciones**

## Especificación de los microprogramas en WepSIM

```
begin
{
    fetch:  (T2, C0=1),
            (Ta, R, BW=11, CI, MI),
            (M2, C2, TI, C3),
            (A0, B=0, C=0)
}
```

## Especificación de los microprogramas en WepSIM

```
ADD R1,R2,R3{
    co=000000,
    nwords=1,
    R1=reg(25,21),
    R2=reg(20,16),
    R3=reg(15,11),
    {
        (SelCop=1010, MC, SelP=11, M7,C7,T6,LC,
         SelA=01011, SelB=10000, SelC=10101,
         A0=1, B=1, C=0)
    }
}
```

# Especificación de los microprogramas en WepSIM

```
BEQ R1, R2, desp{
    co=000100,
    nwords=1,
    R1=reg(25,21),
    R2=reg(20,16),
    desp=address(15,0)rel,
    {
        (T8, C5),
        (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7),
        (A0=0, B=1, C=110, MADDR=bck2ftch),
        (T5, M7=0, C7),
        (T2, C4),
        (SE=1, OFFSET=0, SIZE=10000, T3, C5),
        (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0),
    bck2ftch: (T5, M7=0, C7),
        (A0=1, B=1, C=0)
    }
}
```

etiqueta, representa  
μdirección de salto

# Especificación de registros

```
registers{
    0=$zero,           15=$t7,
    1=$at,            16=$s0,
    2=$v0,            17=$s1,
    3=$v1,            18=$s2,
    4=$a0,            19=$s3,
    5=$a1,            20=$s4,
    6=$a2,            21=$s5,
    7=$a3,            22=$s6,
    8=$t0,            23=$s7,
    9=$t1,            24=$t8,
    10=$t2,           25=$t9,
    11=$t3,           26=$k0,
    12=$t4,           27=$k1,
    13=$t5,           28=$gp,
    14=$t6,           29=$sp (stack_pointer),
                      30=$fp,
                      31=$ra
}
```

# Arranque del computador

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)



# Arranque del computador

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)
- ▶ Se ejecuta el **programa iniciador**
  - ▶ Test del sistema (POST)



A screenshot of a BIOS setup screen. The screen displays system information including the processor (Intel Core2 Extreme CPU X9650 @ 4.00GHz), memory (2096064K DRAM), and two IDE channels (each with two drives). The screen also shows the detection of two IDE drives and the BIOS version (Award Modular BIOS v6.00PG, An Energy Star® Ally). The bottom of the screen shows the copyright notice (© 1994-2007, Award Software, Inc.) and the date (09/19/2007).

```
Awrd Modular BIOS v6.00PG, An Energy Star® Ally
Copyright (C) 1994-2007, Award Software, Inc.

Intel X38 MB00-For X38-DQ6 F4

Main Processor : Intel(R) Core(TM)2 Extreme CPU X9650 @ 4.00GHz(3Ghzx2)
CPUID:0024 Patch ID:0000
Memory Testing : 2096064K DRAM

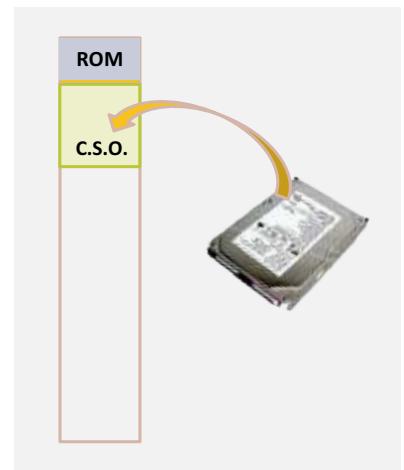
Memory Runs at Dual Channel Interleaved
IDE Channel 0 Slave : 400 Mhz 800MHz-00F0W6 12.01B01
IDE Channel 1 Slave : 400 Mhz 800MHz-00F0W6 12.01B01

Detecting IDE drives ...
IDE Channel 4 Master : None
IDE Channel 4 Slave : None
IDE Channel 5 Master : None
IDE Channel 5 Slave : None

<DEL>:BIOS Setup <F9>:XpressRecovery2 <F12>:Boot Menu <End>:Off Each
09/19/2007-X38-1010-0A79069C-00
```

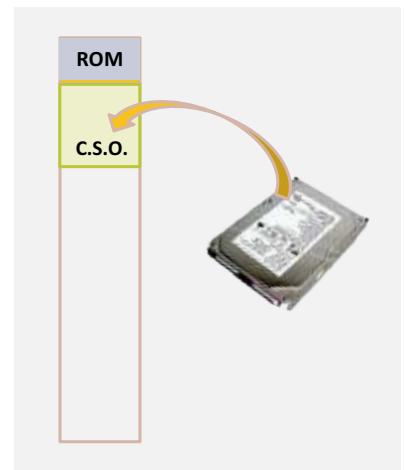
# Arranque del computador

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)
- ▶ Se ejecuta el **programa iniciador**
  - ▶ Test del sistema (POST)
  - ▶ Carga en memoria el **cargador del sistema operativo (MBR)**



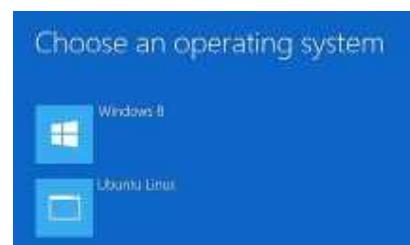
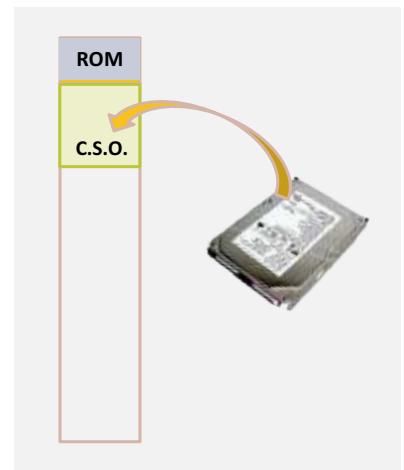
# Arranque del computador

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)
- ▶ Se ejecuta el **programa iniciador**
  - ▶ Test del sistema (POST)
  - ▶ Carga en memoria el **cargador del sistema operativo (MBR)**



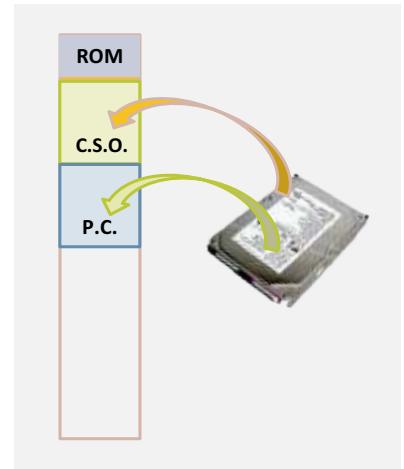
# Arranque del computador

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)
- ▶ Se ejecuta el **programa iniciador**
  - ▶ Test del sistema (POST)
  - ▶ Carga en memoria el **cargador del sistema operativo** (**MBR**)
- ▶ Se ejecuta el **cargador del sistema operativo**
  - ▶ Establece opciones de arranque



# Arranque del computador

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)
- ▶ Se ejecuta el **programa iniciador**
  - ▶ Test del sistema (POST)
  - ▶ Carga en memoria el **cargador del sistema operativo** (**MBR**)
- ▶ Se ejecuta el **cargador del sistema operativo**
  - ▶ Establece opciones de arranque
  - ▶ Carga el **programa de carga**



# Arranque del computador

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)
- ▶ Se ejecuta el **programa iniciador**
  - ▶ Test del sistema (POST)
  - ▶ Carga en memoria el **cargador del sistema operativo**
- ▶ Se ejecuta el **cargador del sistema operativo**
  - ▶ Establece opciones de arranque
  - ▶ Carga el **programa de carga**
- ▶ Se ejecuta el **programa de carga**
  - ▶ Establece estado inicial para el S.O.
  - ▶ Carga el sistema operativo y lo ejecuta

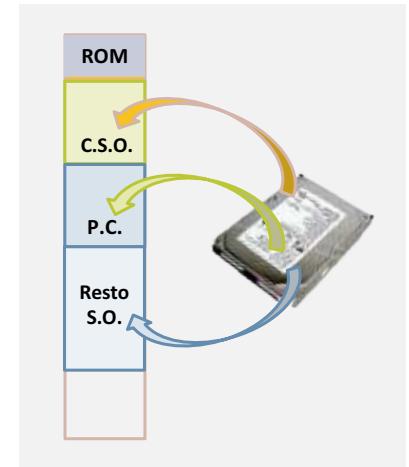


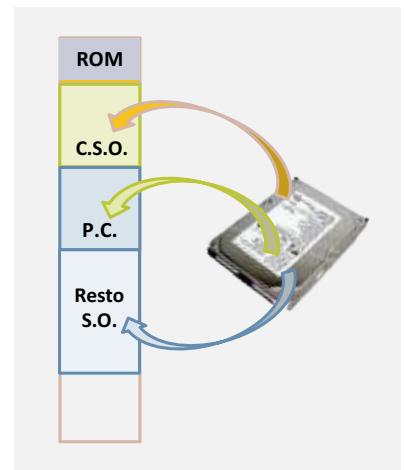
Imagen de la terminal de Linux que muestra el proceso de arranque. Se observan mensajes como "Configuring clock", "Setting up system time from the hardware clock (localtime)", "Initialising boot-time system settings", "Initialising network", "Setting up modules", "Initialising random", "Setting up initrd", "Setting up devtmpfs", "Setting up fakec0 console and c0l", "Loading kernel", "rc0.d[8] >> Going to runlevel 0", "Starting services at runlevel 0", "Starting daemons", "Starting a process to maintain GRUB mode", "GRUB2 English Recovery", "Translating grub2img to 3807200 bytes", "Resizing 616e400" y "Resizing 616e400".

```
Configuring clock
Setting up system time from the hardware clock (localtime)
Initialising boot-time system settings
Initialising network
Setting up modules
Initialising random
Setting up initrd
Setting up devtmpfs
Setting up fakec0 console and c0l
Loading kernel
rc0.d[8] >> Going to runlevel 0
Starting services at runlevel 0
Starting daemons
Starting a process to maintain GRUB mode
GRUB2 English Recovery
Translating grub2img to 3807200 bytes
Resizing 616e400
Resizing 616e400
```

# Arranque del computador

## resumen

- ▶ El Reset carga en los registros sus valores predefinidos
  - ▶ PC ← dirección de arranque del **programa iniciador** (en memoria ROM)
- ▶ Se ejecuta el **programa iniciador**
  - ▶ Test del sistema (POST)
  - ▶ Carga en memoria el **cargador del sistema operativo** (**MBR**)
- ▶ Se ejecuta el **cargador del sistema operativo**
  - ▶ Establece opciones de arranque
  - ▶ Carga el **programa de carga**
- ▶ Se ejecuta el **programa de carga**
  - ▶ Establece estado inicial para el S.O.
  - ▶ Carga el sistema operativo y lo ejecuta



# Tiempo de ejecución de un programa

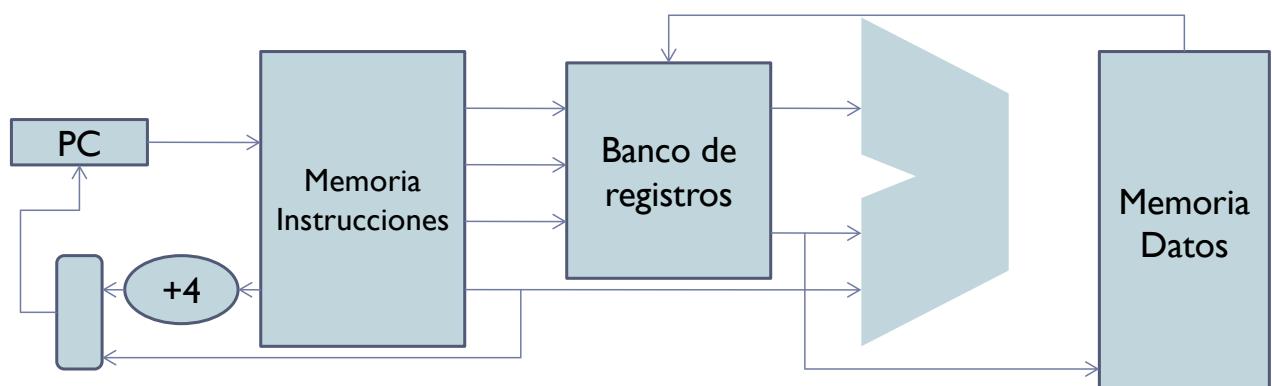
$$\text{Tiempo}_{\text{ejecución}} = \text{NI} \times \text{CPI} \times t_{\text{ciclo\_CPU}} + \text{NI} \times \text{AMI} \times t_{\text{ciclo\_mem}}$$

- ▶ **NI** es el número de instrucciones máquina del programa
- ▶ **CPI** es el número medio de ciclos de reloj necesario para ejecutar una instrucción
- ▶  **$t_{\text{ciclo\_CPI}}$**  es el tiempo que dura el ciclo de reloj del procesador
- ▶ **AMI** es el número medio de accesos a memoria por instrucción
- ▶  **$t_{\text{ciclo\_mem}}$**  es el tiempo de un acceso a memoria

## Factores que afecta al tiempo de ejecución

	NI	CPI	$t_{ciclo\_CPI}$	AMI	$t_{ciclo\_mem}$
Programa	✓			✓	
Compilador	✓	✓		✓	
Juego de instrucciones (ISA)	✓	✓	✓	✓	
Organización		✓	✓		✓
Tecnología			✓		✓

## Modelo de procesador basado en camino de datos (sin bus interno)



# Paralelismo a nivel de instrucción

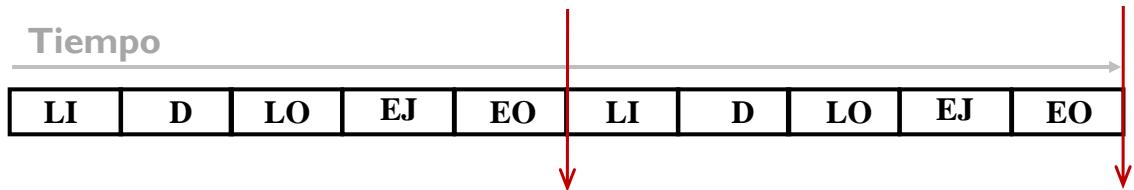
- ▶ Procesamiento concurrente de varias instrucciones
- ▶ Combinación de elementos que trabajan en paralelo:
  - ▶ **Procesadores segmentados:** utilizan técnicas de pipeline para procesar varias instrucciones simultáneamente
  - ▶ **Procesadores superescalares:** procesador segmentado que puede ejecutar varias instrucciones en paralelo cada una de ellas en una unidad segmentada diferente
  - ▶ **Procesadores multicore:** procesador que combina dos o más procesadores independientes en un solo empaquetado

# Segmentación de instrucciones



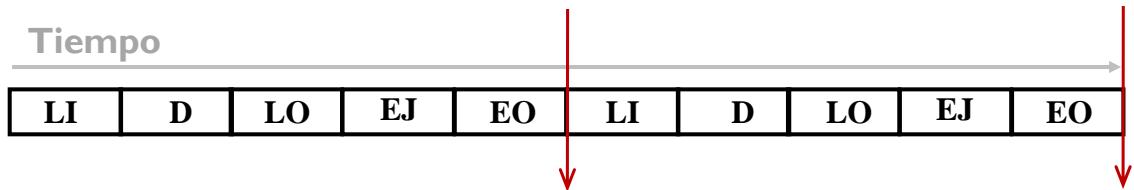
- ▶ Etapas de ejecución de una instrucción:
  - ▶ **LI:** Lectura de la instrucción e incremento del PC
  - ▶ **D:** Decodificación
  - ▶ **LO:** Lectura de Operandos
  - ▶ **EJ:** Ejecución de la instrucción
  - ▶ **EO:** Escritura de Operandos

# Segmentación de instrucciones sin pipeline



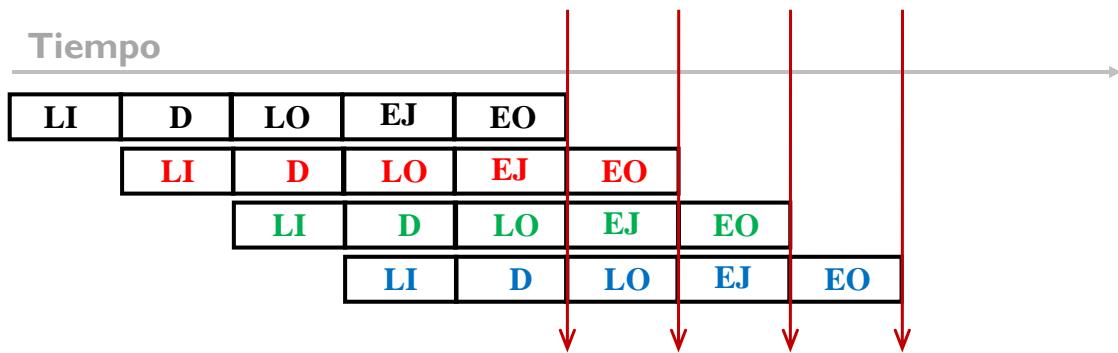
- ▶ Etapas de ejecución de una instrucción:
  - ▶ **LI**: Lectura de la instrucción e incremento del PC
  - ▶ **D**: Decodificación
  - ▶ **LO**: Lectura de Operandos
  - ▶ **EJ**: Ejecución de la instrucción
  - ▶ **EO**: Escritura de Operandos

## Segmentación de instrucciones sin pipeline



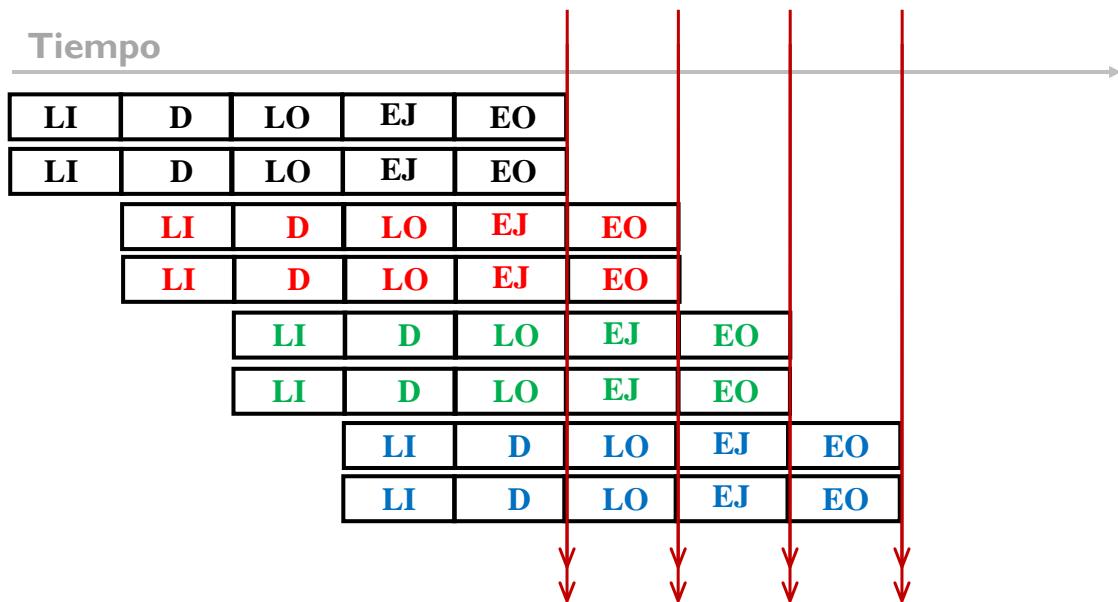
- ▶ Si cada fase dura  $N$  ciclos de reloj, entonces
  - ▶ Una instrucción se ejecuta en  $5*N$  ciclos de reloj
  - ▶ **Cada  $N$  ciclos de reloj se ejecuta  $1/5$  de instrucción**

## Segmentación de instrucciones con pipeline



- ▶ Si cada fase dura  $N$  ciclos de reloj, entonces
  - ▶ Una instrucción se ejecuta en  $5*N$  ciclos de reloj
  - ▶ **Cada  $N$  ciclos de reloj se ejecuta 1 de instrucción**

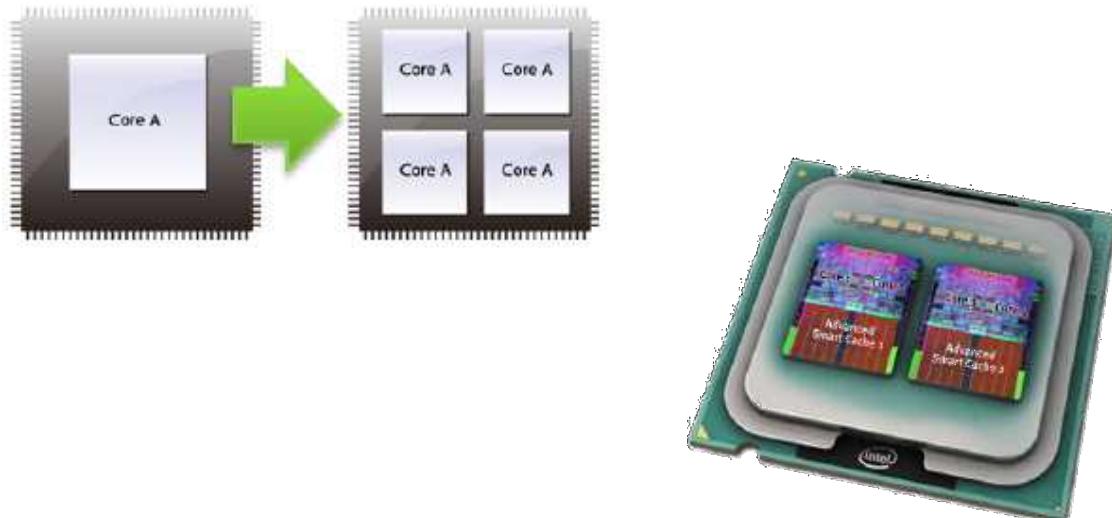
# Superescalar



- ▶ Pipeline con varias unidades funcionales en paralelo

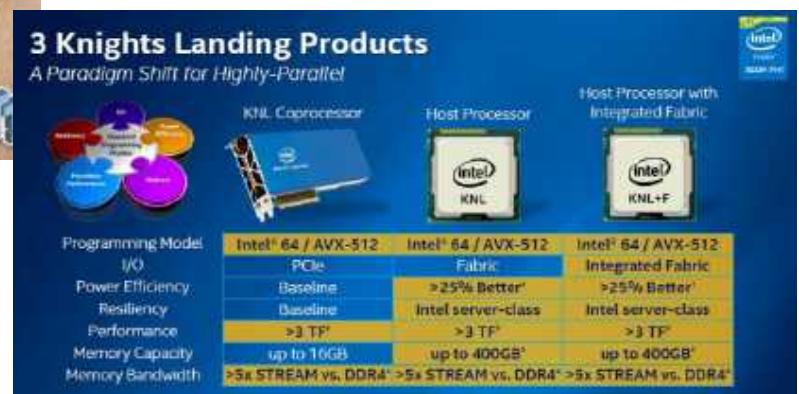
# Multicore

- ▶ Múltiples procesadores en el mismo encapsulado



# Multicore

- ▶ Múltiples procesadores en el mismo encapsulado



<http://wccftech.com/intel-knights-landing-detailed-16-gb-highbandwidth-ondie-memory-384-gb-ddr4-system-memory-support-8-billion-transistors/>



## **Parte VI**

### **Tema 5. Jerarquía de memoria**



Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 5 Jerarquía de Memoria

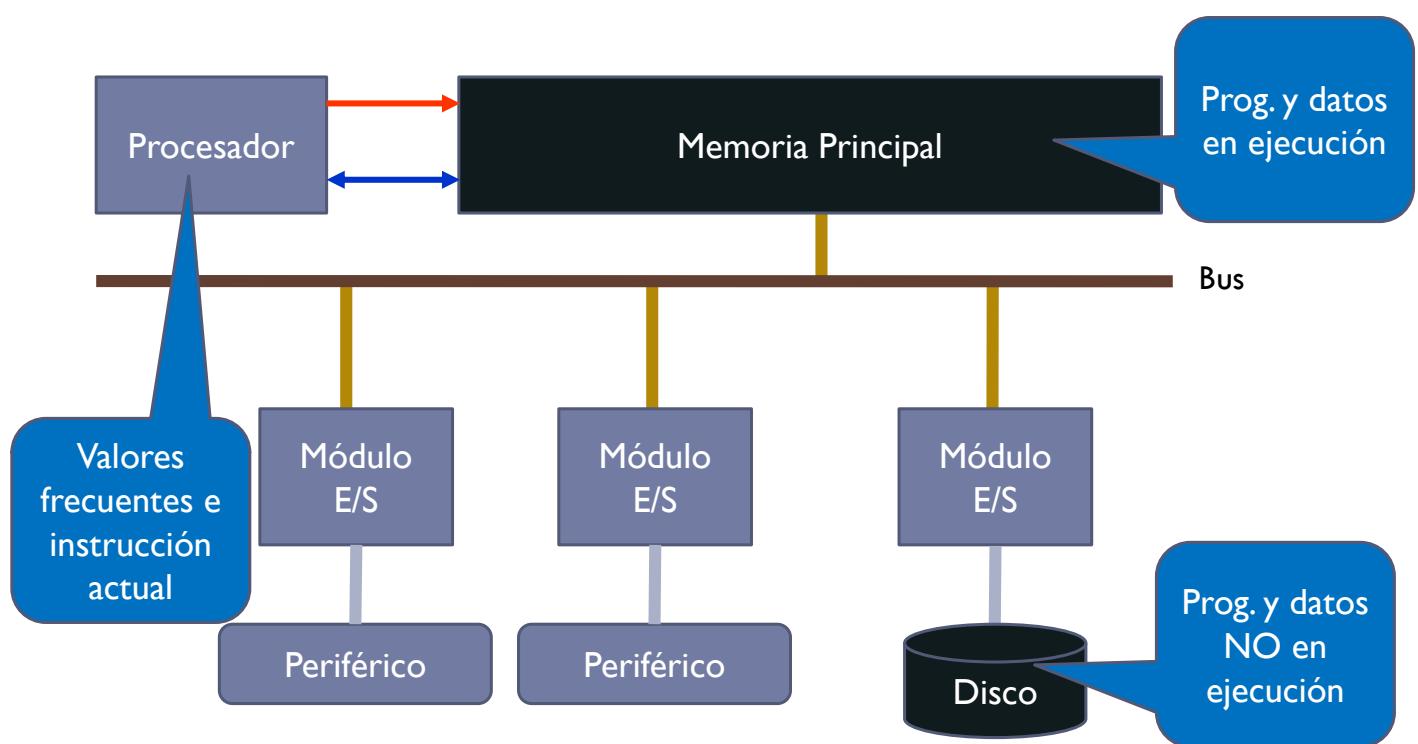
Estructura de Computadores  
Grado en Ingeniería Informática



# Contenidos

1. **Tipos de memoria**
2. **Jerarquía de memoria**
3. **Memoria principal**
4. **Memoria caché**

# Visión general del computador



# Tipos de memoria (hasta el momento)

Procesador

Banco de  
registros

- Almacena pocos datos
- Tiempo de acceso a un registro: orden de ns.

Memoria principal

- Más capacidad (GB).
- Tiempo de acceso: 40-100 ns.
- 1 acceso a memoria = muchos ciclos de reloj

Disco

- Capacidad de almacenamiento casi ilimitada.
- Tiempo de acceso lento: orden de milisegundos

# Distintos tipos de dispositivos físicos

## ▶ Memorias semiconductoras

- ▶ Circuitos electrónicos
- ▶ Ej.: RAM, ROM y Flash



## ▶ Memorias magnéticas

- ▶ Información sobre una superficie magnetizada
- ▶ Ej.: Discos duros y cintas

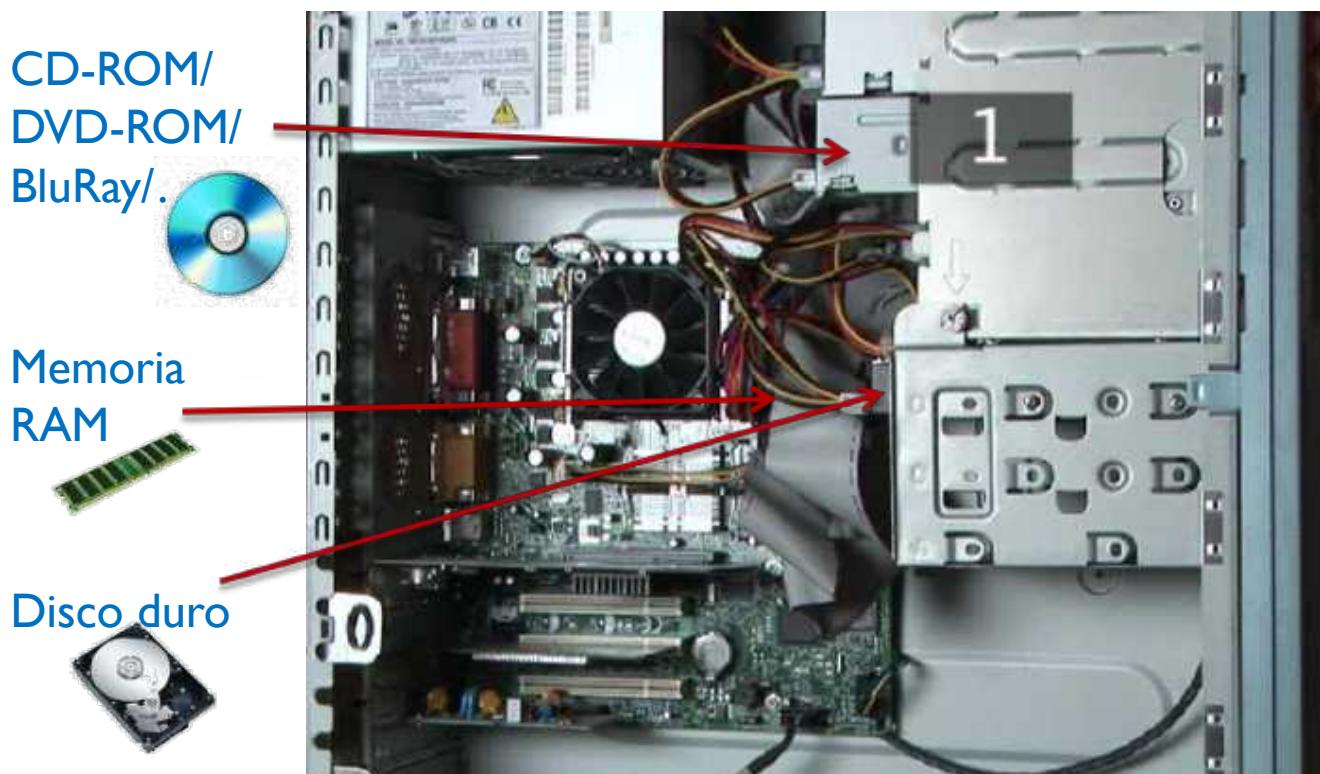


## ▶ Memorias ópticas

- ▶ Información grabada con un láser que genera pequeñas perforaciones sobre una superficie
- ▶ Ej.: CD, DVD y blu-ray



## ¿Dónde se encuentra?



# Principales características

- ▶ **Permanencia de los datos**
  - ▶ Volátiles: RAM
  - ▶ No volátiles: ROM, Flash,
- ▶ **Tipos de operaciones**
  - ▶ Memoria de lectura y escritura (RAM)
  - ▶ Memoria se solo lectura (ROM)
- ▶ **Organización**
  - ▶ Unidad de almacenamiento: bits, palabras, bloques, ...
  - ▶ Modo de acceso:
    - ▶ Secuencial (cinta magnética),
    - ▶ Aleatorio (RAM): se puede acceder en cualquier orden. Mismo tiempo de acceso
- ▶ **Prestaciones**
  - ▶ Tiempo de acceso: tiempo entre presentar dirección y obtener un dato
  - ▶ Ancho de banda o Velocidad de transferencia: cantidad de datos accedidos por unidad de tiempo
- ▶ **Otras**
  - ▶ Capacidad: cantidad de datos que es posible almacenar
  - ▶ Coste: precio por unidad de dato almacenable

# Unidades de tamaño

- ▶ Normalmente se expresa en octetos o bytes:
  - ▶ byte      | byte = 8 bits
  - ▶ kilobyte    | KB = 1.024 bytes       $2^{10}$  bytes
  - ▶ megabyte   | MB = 1.024 KB       $2^{20}$  bytes
  - ▶ gigabyte    | GB = 1.024 MB       $2^{30}$  bytes
  - ▶ terabyte    | TB = 1.024 GB       $2^{40}$  bytes
  - ▶ petabyte    | PB = 1.024 TB       $2^{50}$  bytes
  - ▶ exabyte    | EB = 1.024 PB       $2^{60}$  bytes
  - ▶ zettabyte   | ZB = 1.024 EB       $2^{70}$  bytes
  - ▶ yottabyte   | YB = 1.024 ZB       $2^{80}$  bytes

## Unidades de tamaño (cuidado)

- ▶ En **comunicación** se suele usar el kilobit y no el kilobyte (**I Kb <> I KB**) y potencias de 10
  - ▶ I Kb = 1.000 bits
  - ▶ I KB = 1.000 bytes
- ▶ En **almacenamiento (discos duros)** algunos fabricantes no utilizan potencias de dos, sino potencias de 10:
  - ▶ kilobyte I KB = 1.000 bytes  $10^3$  bytes
  - ▶ megabyte I MB = 1.000 KB  $10^6$  bytes
  - ▶ gigabyte I GB = 1.000 MB  $10^9$  bytes
  - ▶ terabyte I TB = 1.000 GB  $10^{12}$  bytes
  - ▶ .....

# Evolución del rendimiento

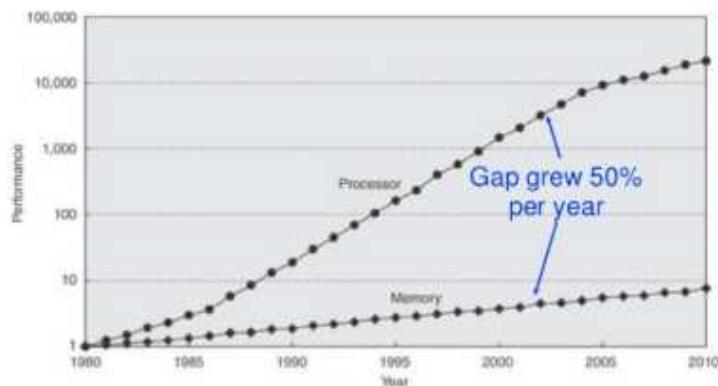
## ▶ Procesadores

- ▶ 1980-2000: Incremento medio del 60% anual.

## ▶ Memorias DRAM

- ▶ 1980-2000: Incremento medio del 7% anual.

## ▶ La distancia entre memoria y procesador es mayor cada año



Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

# Número de accesos a memoria

```
int i;  
int s = 0;  
for (i=0; i < 1000; i++)  
    s = s + i;
```

- ▶ ¿Cuántos accesos a memoria se generan en este fragmento de memoria?

# Número de accesos a memoria

```
int i;
int s = 0;
for (i=0; i < 1000; i++)
    s = s + i;
i=0;

li    $t0, 0      // s
li    $t1, 0      // i
li    $t2, 1000
bucle: bge  $t1, $t2, fin
        add   $t0, $t0, $t1
        addi  $t1, $t1, 1
        b     bucle
fin:   li    $t1, 0
```

# Número de accesos a memoria

```
int i;
int s = 0;
for (i=0; i < 1000; i++)
    s = s + i;
i=0;
```

	li \$t0, 0 // s
	li \$t1, 0 // i
	li \$t2, 1000
	bucle: bge \$t1, \$t2, fin
	add \$t0, \$t0, \$t1
	addi \$t1, \$t1, 1
	b bucle
	fin: li \$t1, 0

**Solución:**  $3 + 4 \times 1000 + 1 + 1 = 4005$

# Número de accesos a memoria

```
int i;                                li    $t0, 0      // s
int s = 0;                             li    $t1, 0      // i
for (i=0; i < 1000; i++)           li    $t2, 1000
    s = s + i;                      bucle: bgt  $t1, $t2, fin
                                         add   $t0, $t0, $t1
                                         addi  $t1, $t1, 1
                                         b     bucle
                                         fin:  li    $t1, 1
```

**Solución:**  $3 + 4 \times 1000 + 1 + 1 = 4005$

Con una memoria de 60 ns el tiempo total sería 240300 ns

Un procesador típico dedicaría más del 98% de su tiempo a esperar datos de memoria

# Número de accesos a memoria

```
int v[1000]; // global  
  
int i;  
for (i=0; i < 1000; i++)  
    v[i] = 0;
```

# Número de accesos a memoria

```
int v[1000]; // global      .data:  
                           v: .space 4000  
  
int i;  
for (i=0; i < 1000; i++)    .text:  
    v[i] = 0;  
                           li    $t0, 0      // i  
                           li    $t1, 0      // i de v  
                           li    $t2, 1000 // componentes  
bucle: bge   $t0, $t2, fin  
       sw    $0, v($t1)  
       addi  $t0, $t0, 1  
       addi  $t1, $t1, 4  
       b     bucle  
fin:
```

# Número de accesos a memoria

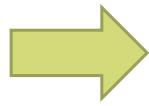
```
int v[1000]; // global      .data:  
                           v: .space 4000  
  
int i;  
for (i=0; i < 1000; i++)    .text:  
    v[i] = 0;  
                                li    $t0, 0      // i  
                                li    $t1, 0      // i de v  
                                li    $t2, 1000 // componentes  
bucle: bgt    $t0, $t2, fin  
        sw     $0, v($t1)  
        addi   $t0, $t0, 1  
        addi   $t1, $t1, 4  
        b      bucle
```

**Solución:**  $3 + 5 \times 1000 + 1 + 1000$  (acceso adicional de sw) = 6004

# Contenidos

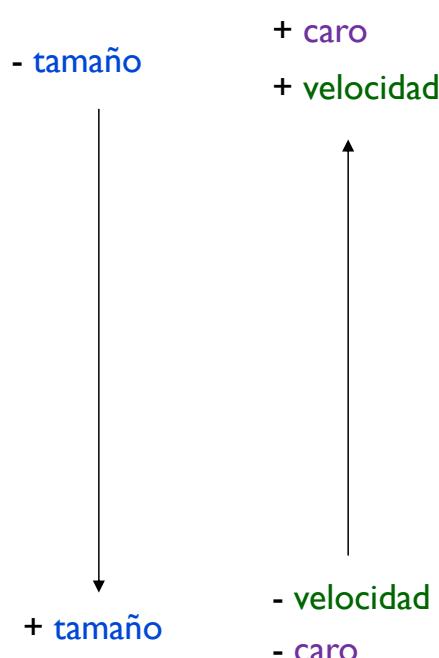
1. Tipos e memoria
2. Jerarquía de memoria
3. Memoria principal
4. Memoria caché
5. Memoria virtual

## ¿Cómo sería el sistema de memoria ideal?



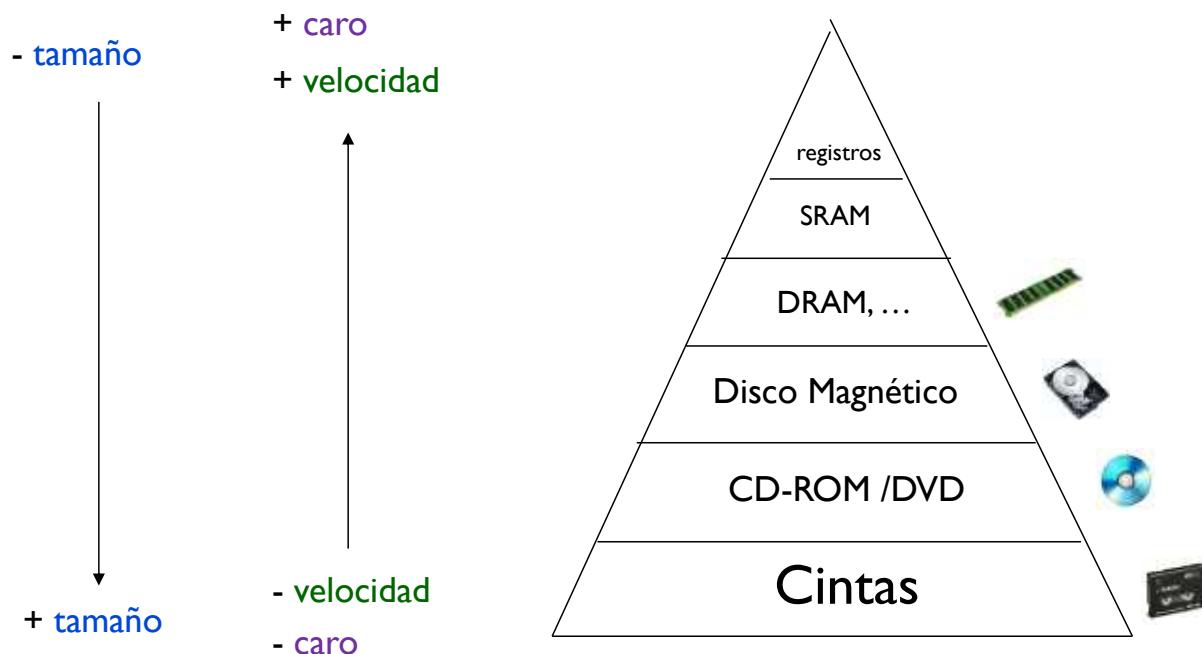
- ▶ Minimiza **tiempo de acceso**
- ▶ Maximiza la **capacidad**
- ▶ Minimiza el **coste**

# Realidad



- ▶ Objetivos **incompatibles entre si:**
  - ▶ + velocidad → - tamaño
- ▶ Se usan distintos tipos de memoria:
  - ▶ DRAM, Disco Duro, ...
- ▶ Se organizan los distintos tipos de memoria por velocidad de acceso:
  - ▶ **Jerarquía de memoria**

# Jerarquía de memoria



## Uso de la jerarquía de memoria: diferentes tiempos de acceso

- ▶ T. acceso a registro                            La biblioteca de la UC3M...
  - ▶ ~1 ns
- ▶ T. acceso a SRAM                            La biblioteca de la UPC...
  - ▶ ~2-5 ns
- ▶ T. acceso a DRAM                            Una biblioteca en Florida...
  - ▶ ~70-100 ns

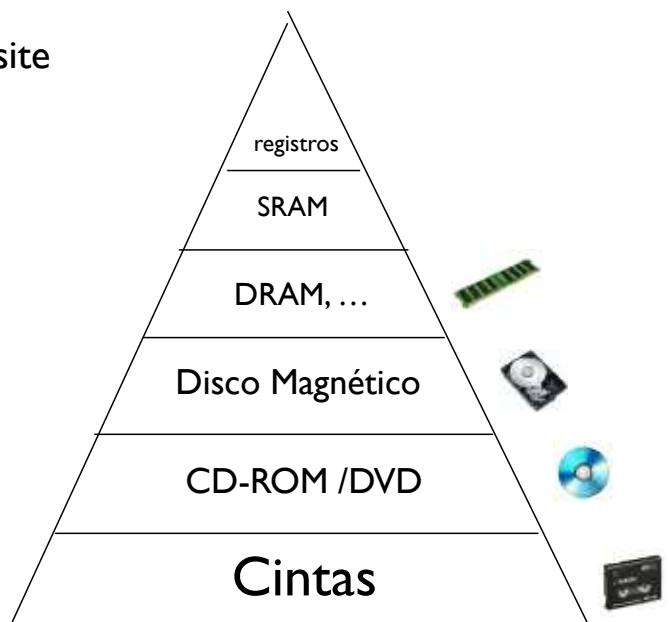
# Comparación

Technology	Bytes per Access (typ.)	Latency per Access	Cost per Megabyte <sup>a</sup>	Energy per Access
On-chip Cache	10	100 of picoseconds	\$1–100	1 nJ
Off-chip Cache	100	Nanoseconds	\$1–10	10–100 nJ
DRAM	1000 (internally fetched)	10–100 nanoseconds	\$0.1	1–100 nJ (per device)
Disk	1000	Milliseconds	\$0.001	100–1000 mJ

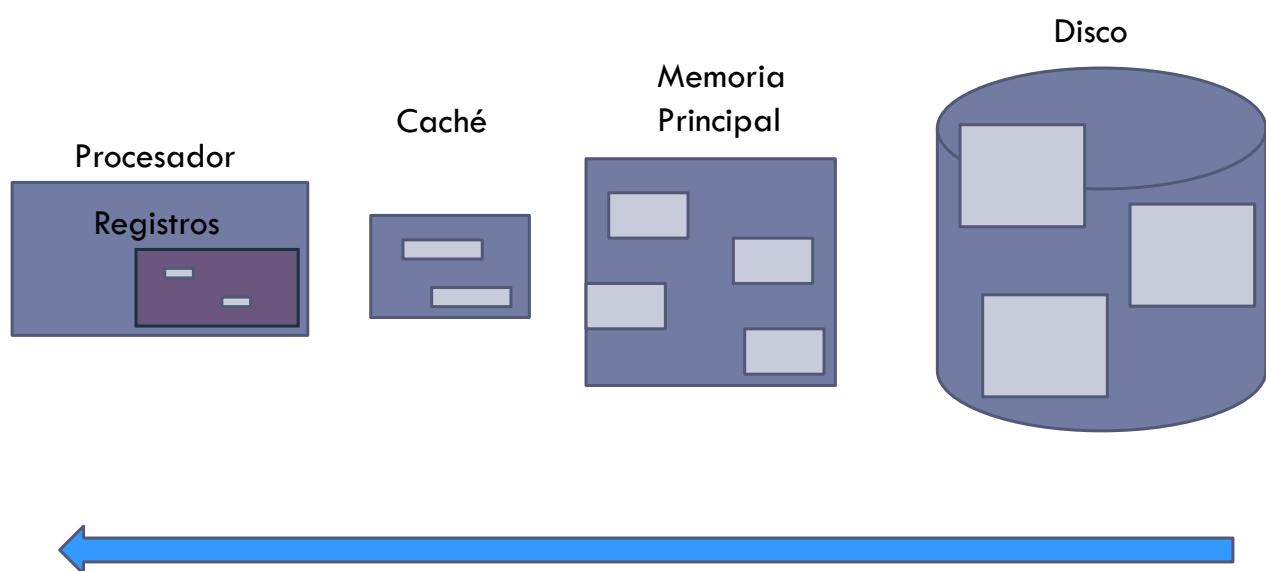
Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David Wang  
Elsevier

# Uso de la jerarquía de memoria

- ▶ Solo en memoria lo que se necesite en un instante dado.
- ▶ Si no está, se copia de un nivel a otro la porción necesaria:
  - ▶ Ej.: cargar un programa en RAM
- ▶ Cuando no se necesite, se borra la copia realizada.
- ▶ El comportamiento de los accesos lo favorece:
  - ▶ **Proximidad de referencias**



# Idea de la jerarquía de memoria



## Diseño de la jerarquía de memoria

- ▶ El diseño de la jerarquía de memoria es crucial en procesadores multicore
- ▶ El ancho de banda crece con el número de cores
  - ▶ Un Intel Core i7 genera dos accesos a memoria por core y por ciclo de reloj
  - ▶ Con 4 cores y 3.2 GHz de frecuencia de reloj
    - ▶ 25.6 billones de accesos a datos de 64 bit/segundo +
    - ▶ 12.8 billones de accesos de 128 bits para instrucciones =
    - ▶ = 409.6 GB/s
  - ▶ Una memoria DRAM solo ofrece un 6% (25GB/s)
  - ▶ Se requiere:
    - ▶ Memorias multi puerto
    - ▶ Niveles de memoria caché

# Memorias de semiconductores

- ▶ **Memoria de solo lectura (ROM)**
  - ▶ No necesita alimentación
  - ▶ Persistente
  - ▶ Ejemplo de uso: BIOS
- ▶ **Memoria de lectura/escritura (RAM)**
  - ▶ Necesita alimentación
  - ▶ No persistente
  - ▶ Más rápida que la ROM
  - ▶ Ejemplo de uso: memoria principal

# Matriz de memoria semiconductor

- ▶ Cada **celda** almacena un **1** o un **0**

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

(a) Matriz  $8 \times 8$

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

(b) Matriz  $16 \times 4$

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

(c) Matriz  $64 \times 1$

Fundamenros de Sistemas Digitales  
Thomas L. Floyd

# Direcciones y capacidad

- ▶ Dirección: posición de una unidad de datos en la matriz de memoria

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

(a) La dirección del bit gris claro es fila 5, columna 4.

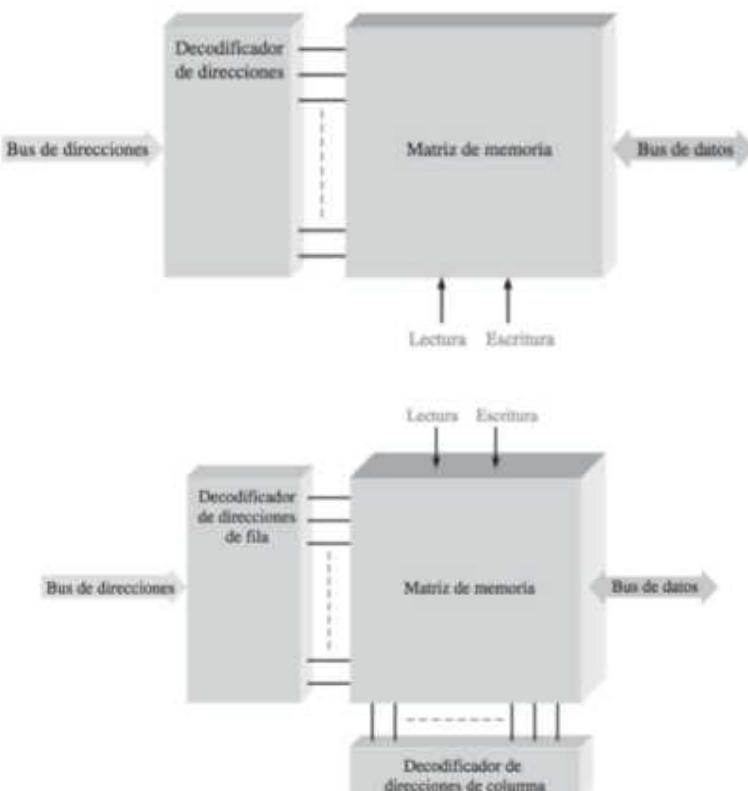
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

(b) La dirección del byte gris claro es la fila 3.

Fundamentos de Sistemas Digitales  
Thomas L. Floyd

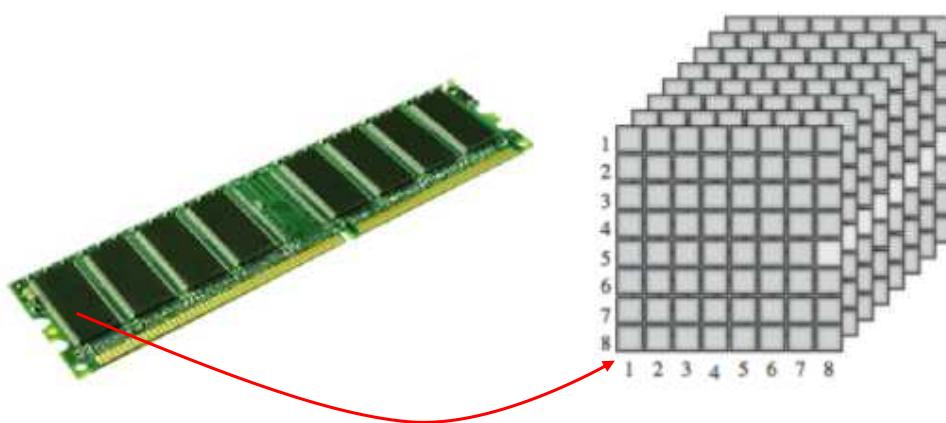
- ▶ Capacidad: número total de unidades de datos que se pueden almacenar

# Tipos de direccionamientos



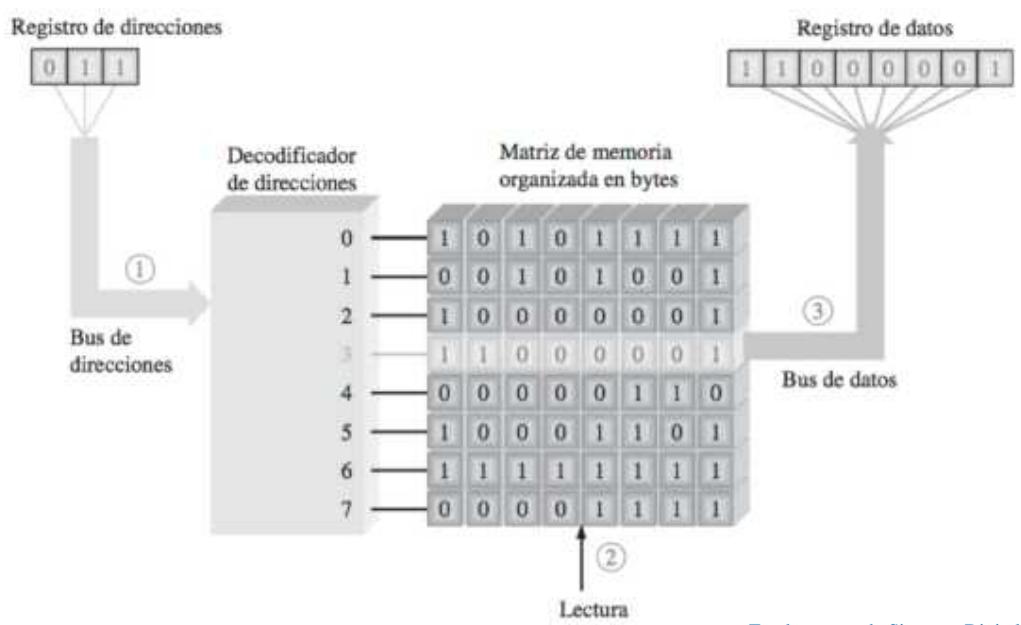
Fundamentos de Sistemas Digitales  
Thomas L. Floyd

## Ejemplo de organización



La dirección del byte en color gris claro corresponde a la fila 5, columna 8.

# Operación de lectura

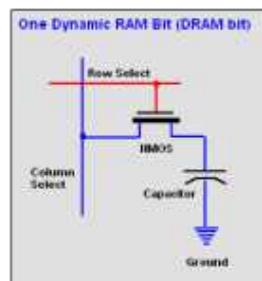


# Memoria RAM (memorias de acceso aleatorio)

## ▶ RAM dinámica (DRAM)

- ▶ Almacena bits como carga en condensadores.
- ▶ Tiende a descargarse: necesita refrescos periódicos.
  - ▶ Ventaja: construcción más simple, **más almacenamiento**, más económica
  - ▶ Inconveniente: necesita circuitería de refresco, **más lenta**.
    - 2%-3% de los ciclos de reloj consume el refresco
- ▶ Utilizada en memorias principales

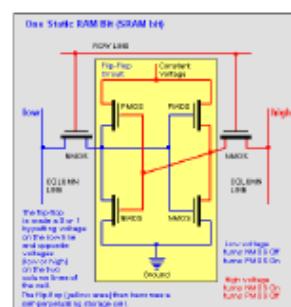
From Computer Desktop Encyclopedia  
© 2005 The Computer Language Co., Inc.



## ▶ RAM estática (SRAM)

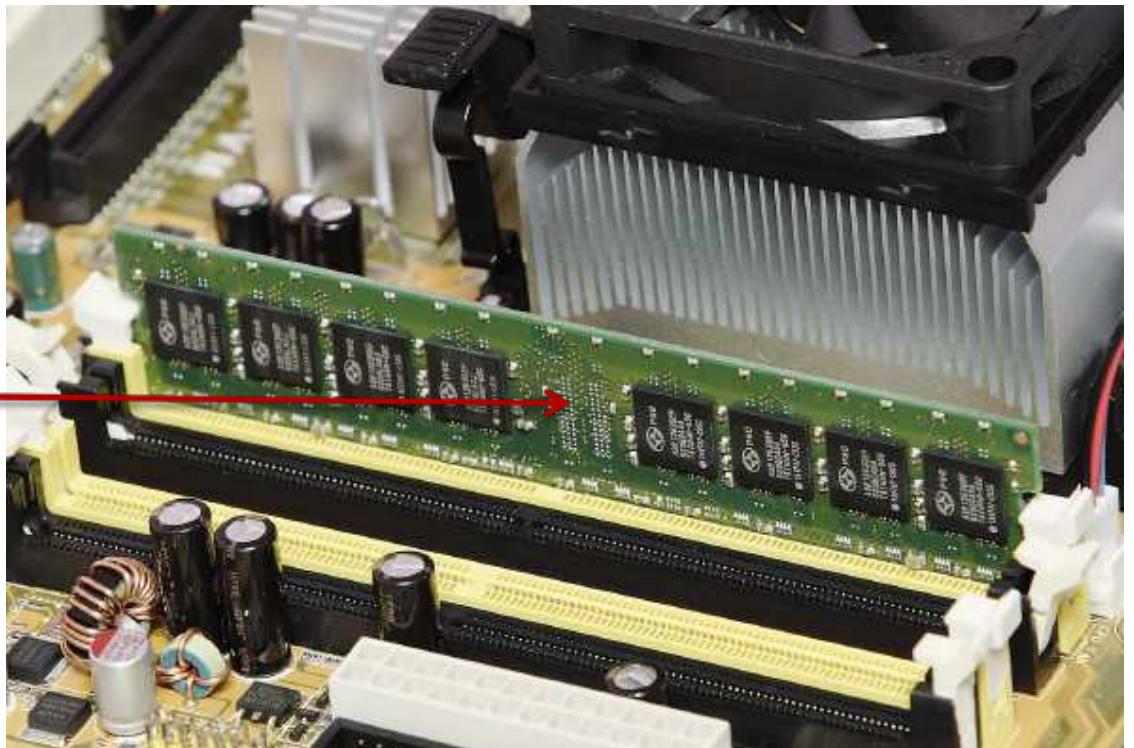
- ▶ Almacena bits como interruptores en **on** y **off**.
- ▶ Tiende a no descargarse: **no** necesita refresco.
  - ▶ Ventaja: No necesita circuitería de refresco, **más rápida**.
  - ▶ Inconveniente: Construcción compleja, **menos almacenamiento**, más cara.
- ▶ Utilizada en memorias cachés

From Computer Desktop Encyclopedia  
© 2005 The Computer Language Co., Inc.

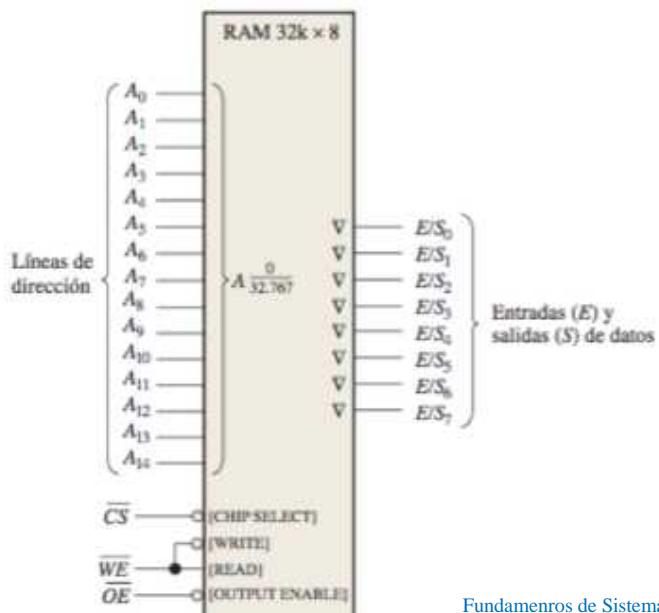


# ¿Dónde se encuentra la memoria DRAM?

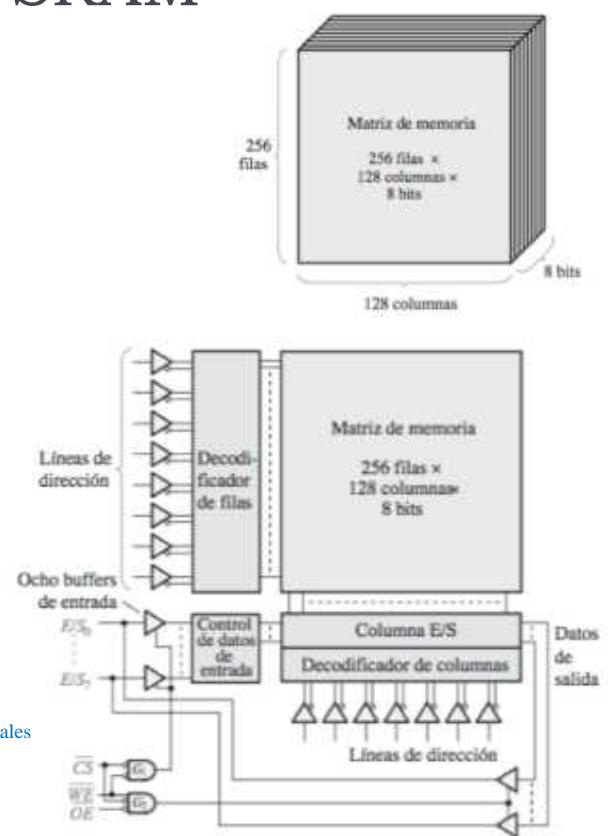
Memoria  
DRAM



# Ejemplo de memoria SRAM

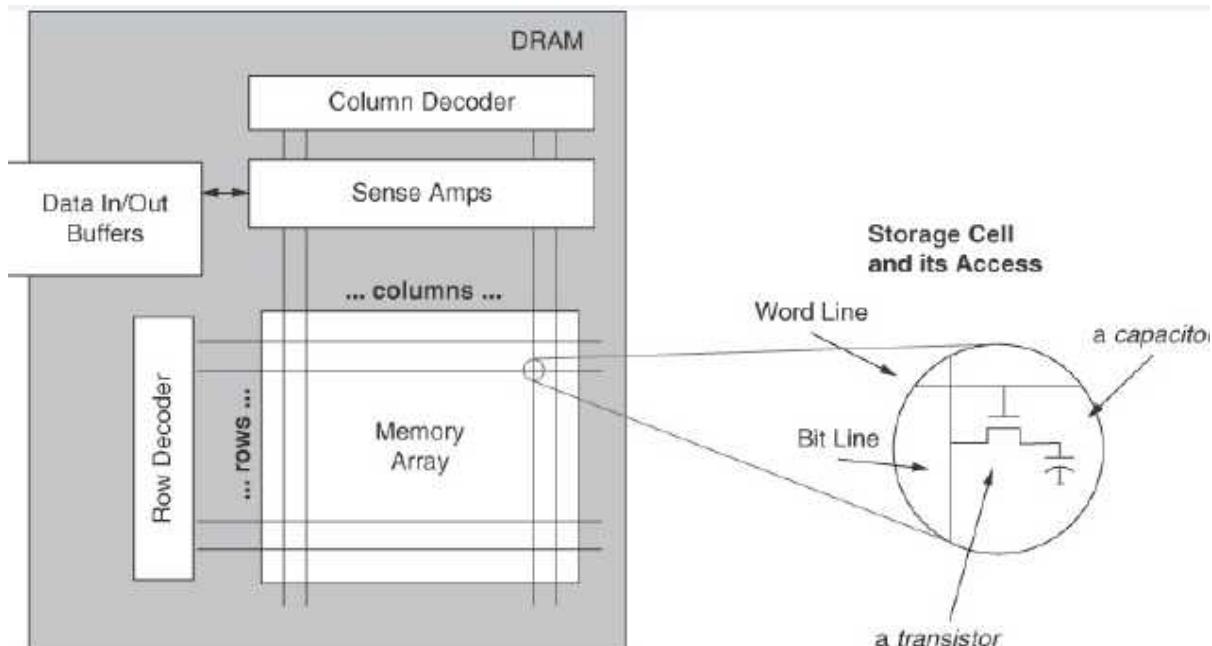


Organización lógica



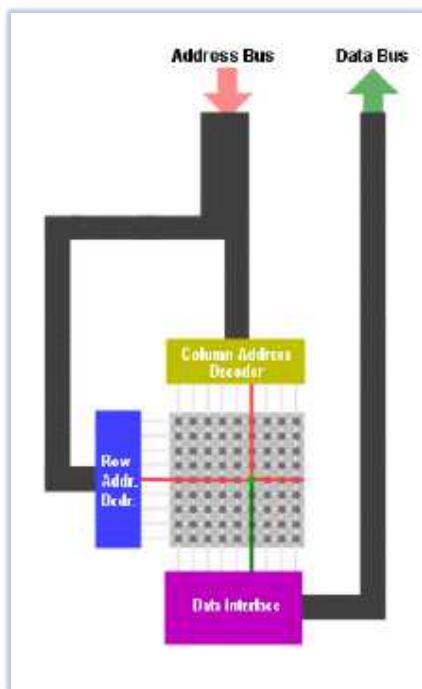
Organización física      Ocho buffers de salida

# Estructura de una memoria DRAM

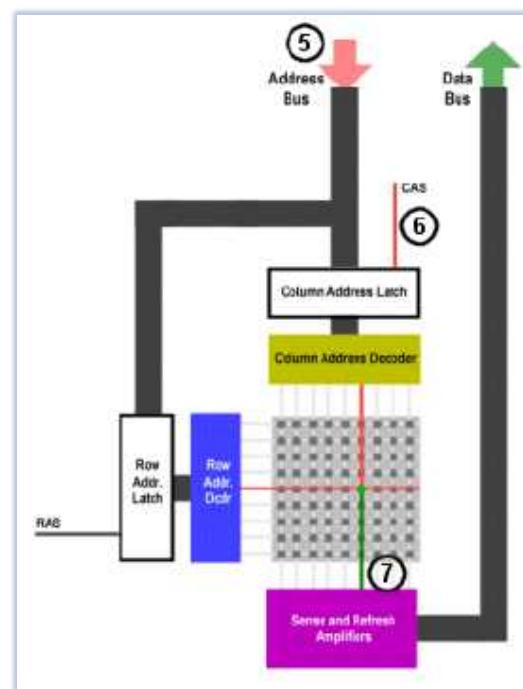


Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David  
Wang  
Elsevier

# Multiplexión de direcciones en DRAM

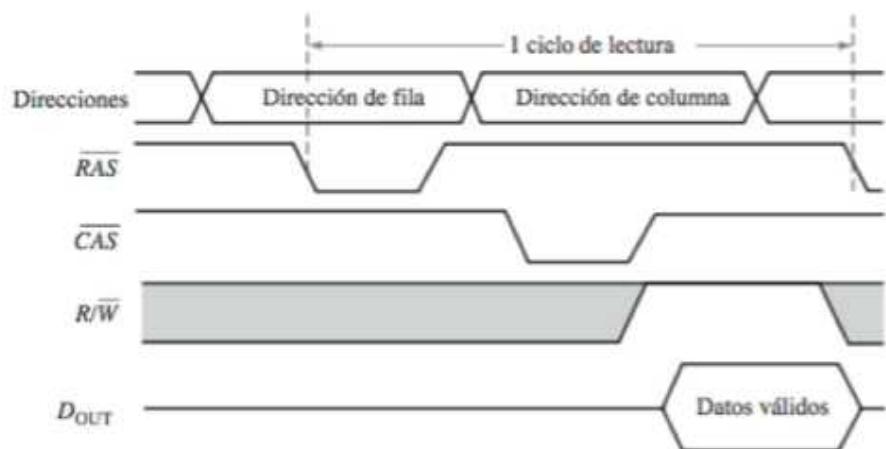


Direccionamiento  
por fila/columna



Direccionamiento  
por fila/columna con CAS/RAS

# Operación de lectura con CAS/RAS



## Ciclos de refresco

- ▶ Una DRAM almacena un bit en un condensador
- ▶ Esta carga se degrada con el tiempo y la temperatura
- ▶ Necesario refrescar cada bit
- ▶ Tipicamente una DRAM se debe refrescar cada pocos milisegundos
- ▶ Una operación de lectura refresca toda las direcciones de una fila
- ▶ Una DRAM utiliza ciclos de refresco

# Velocidad de las memorias DRAM

Production year	Chip size	DRAM Type	Slowest DRAM (ns)	Fastest DRAM (ns)	Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

Figure 2.13 Times of fast and slow DRAMs vary with each generation. (Cycle time is defined on page 95.) Perfor-

Patterson y Hennesy

# Tipos de memoria RAM

## ▶ DRAM

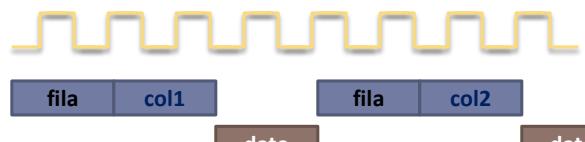
▶ FPM  
(Fast Page mode)

▶ EDO  
(Extended Data Output)

reloj

dir.

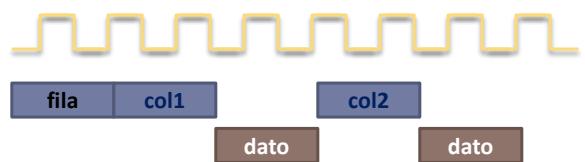
datos



reloj

dir.

datos



## ▶ SDRAM

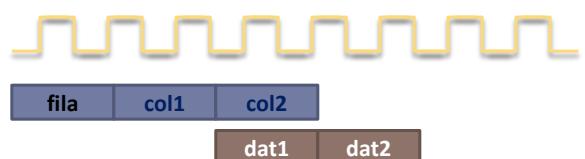
▶ DDR

▶ DDR2

reloj

dir.

datos



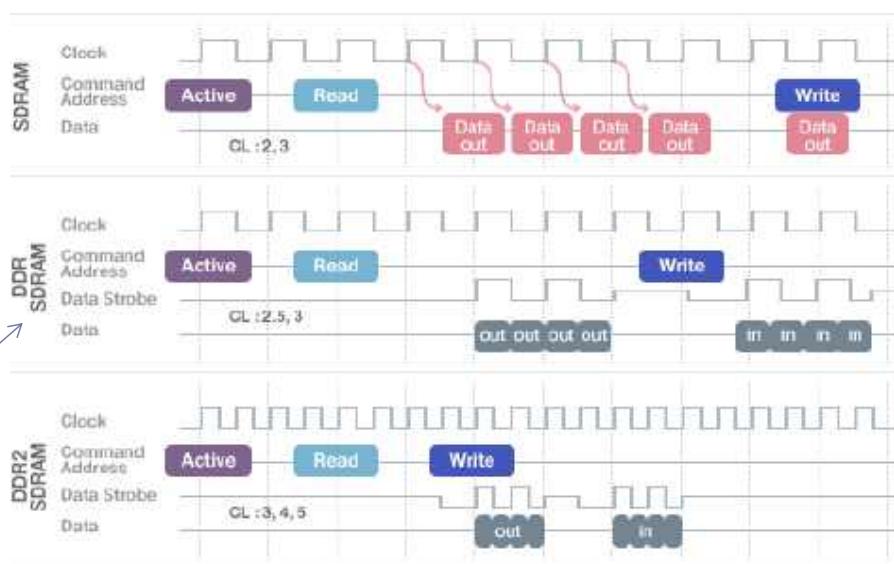
# Tipos de memoria RAM

## ▶ DRAM

- ▶ EDO
- ▶ FPM

## ▶ SDRAM

- ▶ DDR
- ▶ DDR2  
(double data rate)



SDRAM (Synchronous DRAM): sincronizadas con el reloj del sistema

# Tipos de memoria DDR

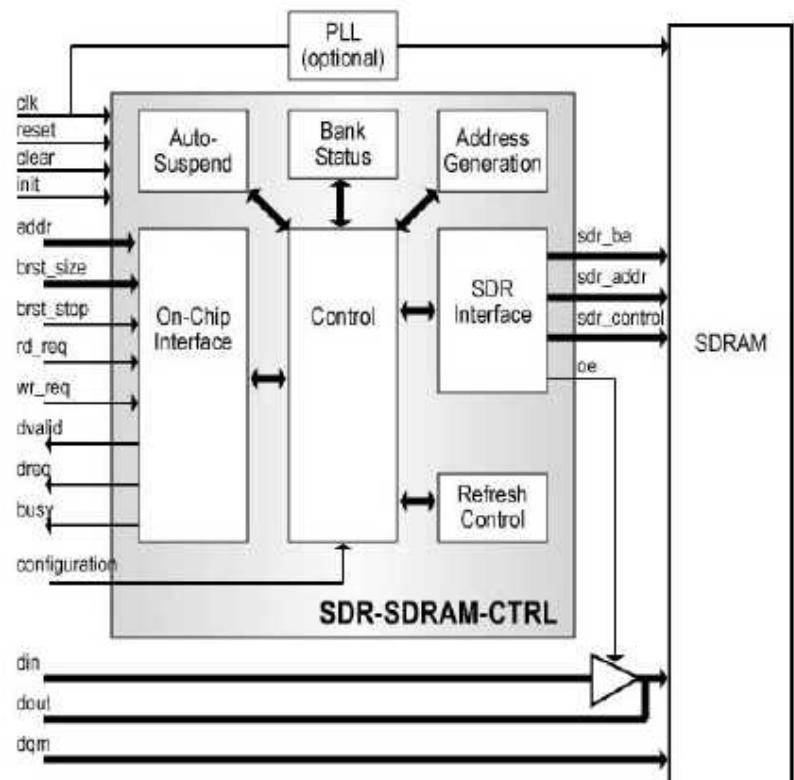
Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMs and DIMMs in 2010. Note the numerical relation-

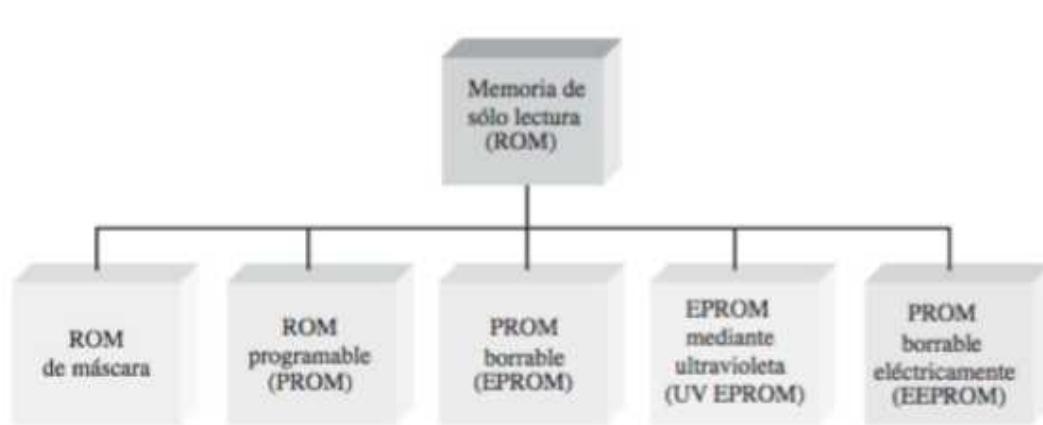
Patterson y Hennesy

# Controlador de memoria DRAM

- ▶ Controlador se encarga del refresco y particularidades de la DRAM
- ▶ Oculta todo esto al procesador y le ofrece una interfaz simple
  - ▶ Procesador **no** dependiente de la tecnología de la memoria



# Memorias ROM



Fundamentos de Sistemas Digitales  
Thomas L. Floyd

Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 5 (II) Jerarquía de Memoria



# Contenidos

1. Tipos de memoria
2. Jerarquía de memoria
3. Memoria principal
4. Memoria caché

# Característica de la memoria principal

- ▶ Se premia el acceso a posiciones consecutivas de memoria
  - ▶ Ejemplo 1: acceder a 5 posiciones de memoria **individuales**



- ▶ Ejemplo 2: acceder a 5 posiciones de memoria **consecutivas**



# Característica de los accesos a memoria

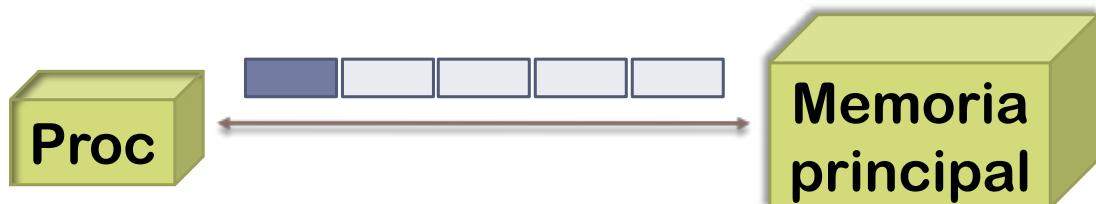
- ▶ “Principio de proximidad o localidad de referencias”:  
**Durante la ejecución de un programa,  
las referencias (direcciones) a memoria tienden a estar  
agrupadas por:**
- ▶ **proximidad espacial**
  - ▶ Secuencia de instrucciones
  - ▶ Acceso secuencial a arrays
- ▶ **proximidad temporal**
  - ▶ Bucles

```
.data
vector: .space 4*1024

.text
main: li $t0 0
      la $t1 vector
      b2: bge $t0 1024 finb2
           mult $t2 $t0 4
           add $t2 $t1 $t2
           sw $t0 ($t2)
           add $t0 $t0 1
           b b2
finb2: jr $ra
```

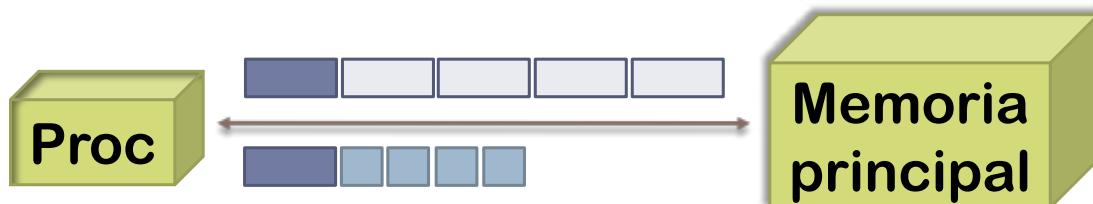
Objetivo de la memoria caché: aprovechar los accesos contiguos

- ▶ Si cuando se accede a una posición de memoria solo se transfieren los datos de esa posición, no se aprovecha los posibles accesos a datos contiguos.



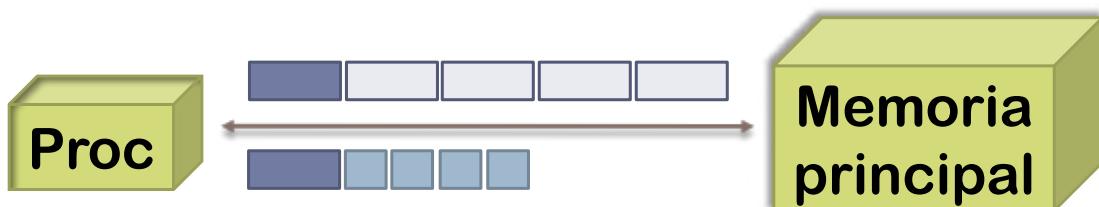
Objetivo de la memoria caché: aprovechar los accesos contiguos

- ▶ Si cuando se accede a una posición de memoria se transfiere esos datos y los contiguos, sí se aprovecha el acceso a datos contiguos



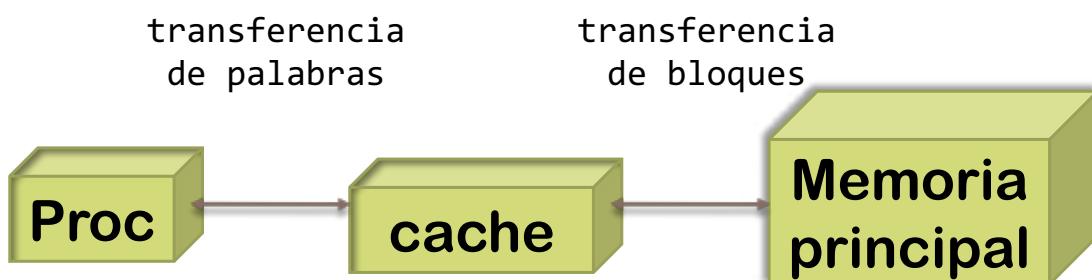
## Objetivo de la memoria caché: aprovechar los accesos contiguos

- ▶ Si cuando se accede a una posición de memoria se transfiere esos datos y los contiguos, sí se aprovecha el acceso a datos contiguos
  - ▶ Transfiero de la memoria principal un bloque de palabras
  - ▶ **¿Dónde se almacenan las palabras del bloque?**

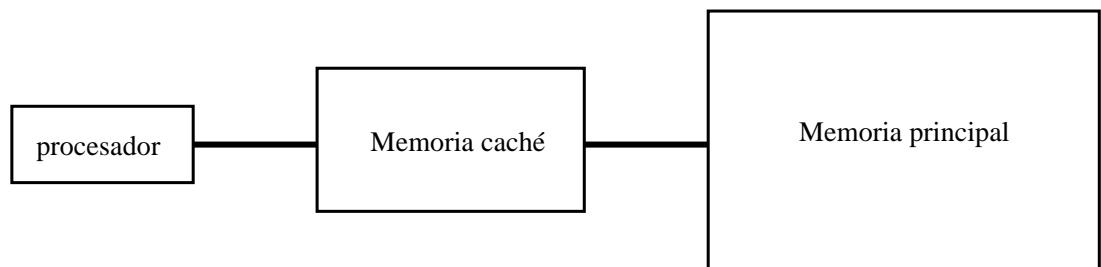


## Memoria Cache

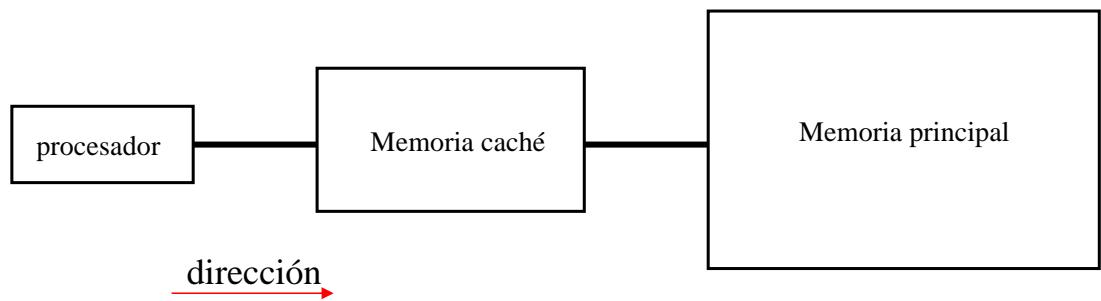
- ▶ Cantidad pequeña de memoria rápida SRAM
  - ▶ Integrada en el mismo procesador
  - ▶ Más rápida y cara que la memoria principal DRAM
- ▶ Está entre la memoria principal y el procesador
- ▶ Almacena una **copia** de partes de la memoria principal



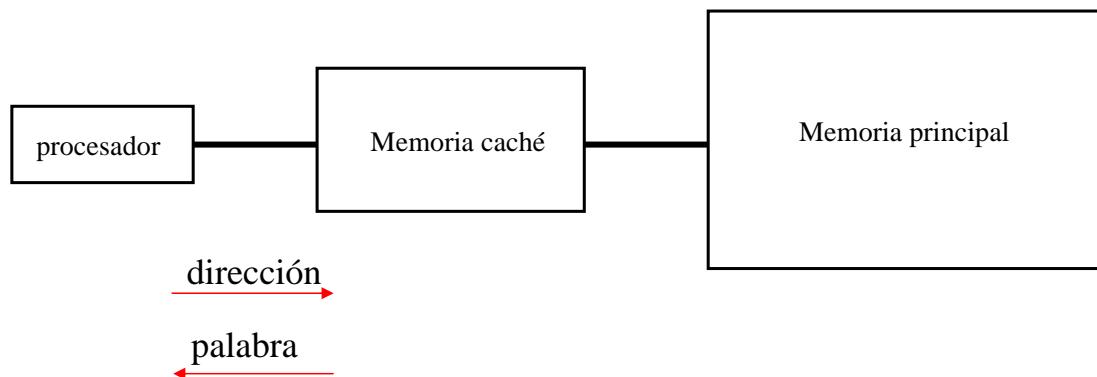
# Funcionamiento de la memoria caché



# Funcionamiento de la memoria caché

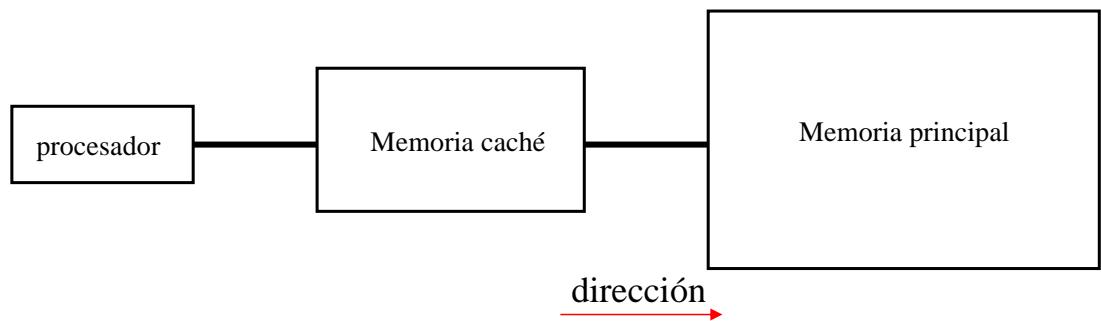


# Funcionamiento de la memoria caché



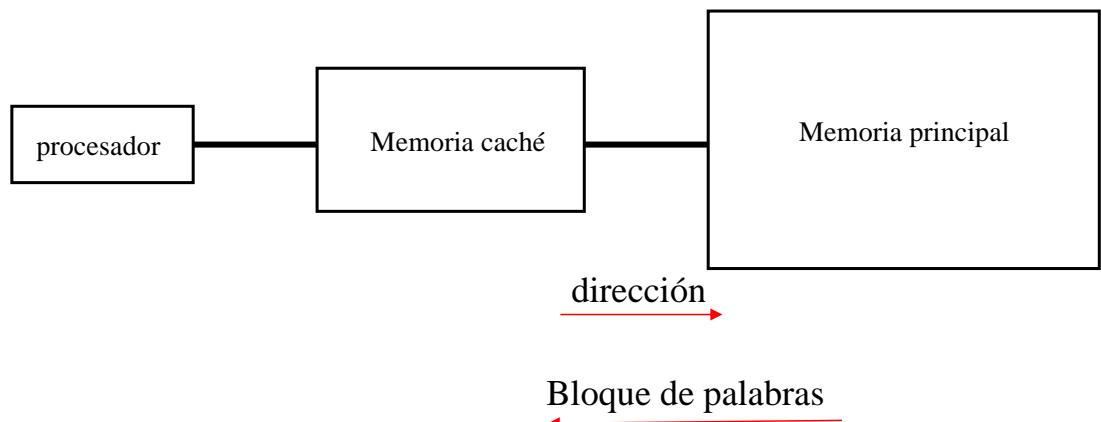
ACIERTO

# Funcionamiento de la memoria caché

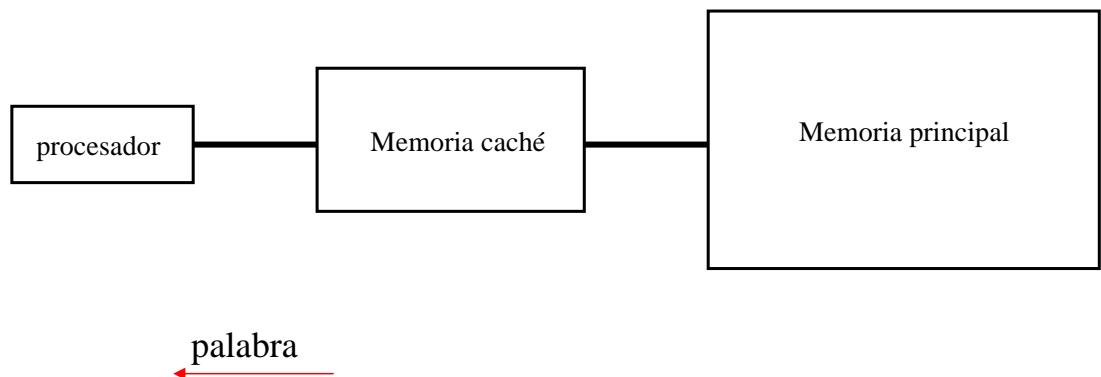


FALLO

# Funcionamiento de la memoria caché

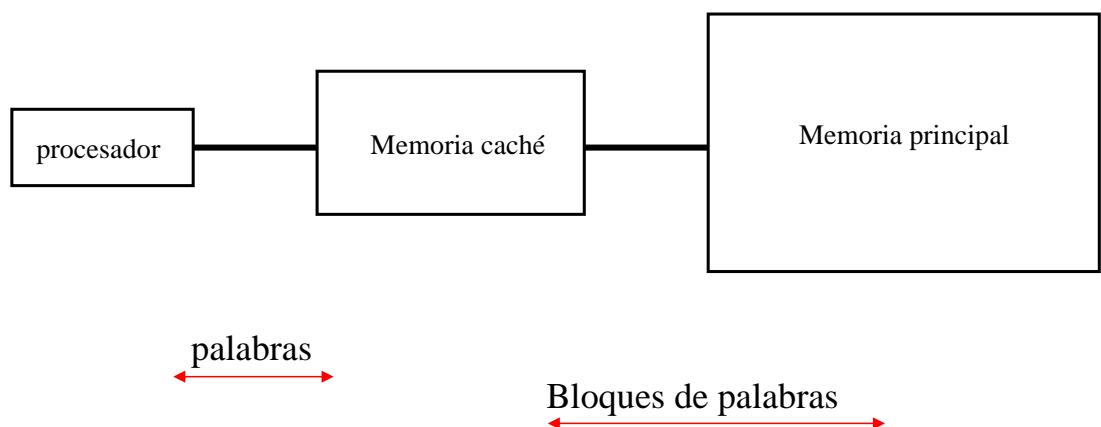


# Funcionamiento de la memoria caché

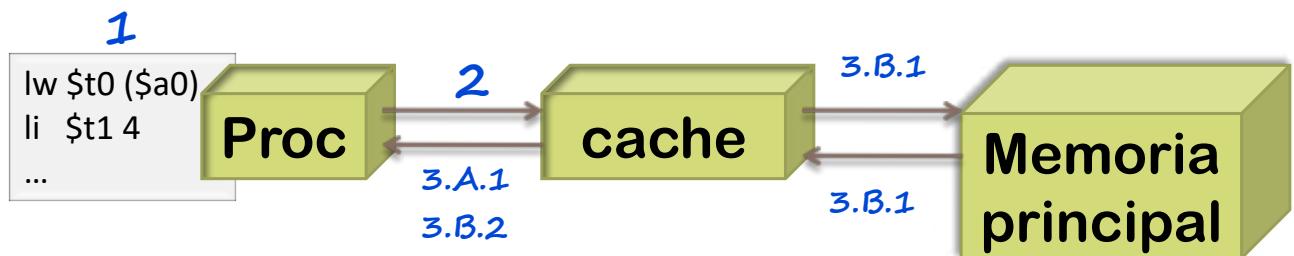


# Funcionamiento de la memoria caché

## Resumen



# Funcionamiento general



1. El procesador solicita contenidos de una posición de memoria.
2. La cache comprueba si ya están los datos de esta posición:
  - ▶ Si está (**ACIERTO**)
    - 3.A.1 Se la sirve al procesador desde la cache (rápidamente).
  - ▶ Si no está (**FALLO**),
    - 3.B.1 La cache transfiere de memoria principal el bloque asociado a la posición.
    - 3.B.2 Después, la cache entrega los datos pedidos al procesador

# Ejemplo de funcioamiento

```
int i;                                li    $t0, 0      // s
int s = 0;                            li    $t1, 0      // i
for (i=0; i < 1000; i++)          bucle: li    $t2, 1000
                                         bge   $t1, $t2, fin
                                         add   $t0, $t0, $t1
                                         addi  $t1, $t1, 1
                                         b     bucle
                                         fin:  ...
```

- ▶ Ejemplo:

- ▶ Acceso a caché: 2 ns
- ▶ Acceso a MP: 120 ns
- ▶ Bloque de caché: 4 palabras
- ▶ Transferencia de un bloque entre memoria principal y caché: 200 ns

# Ejemplo de funcionamiento

```
int i;                                li    $t0, 0      // s
int s = 0;                            li    $t1, 0      // i
for (i=0; i < 1000; i++)          bucle: li    $t2, 1000
                                         bge   $t1, $t2, fin
                                         add   $t0, $t0, $t1
                                         addi  $t1, $t1, 1
                                         b     bucle
                                         fin: ...
```

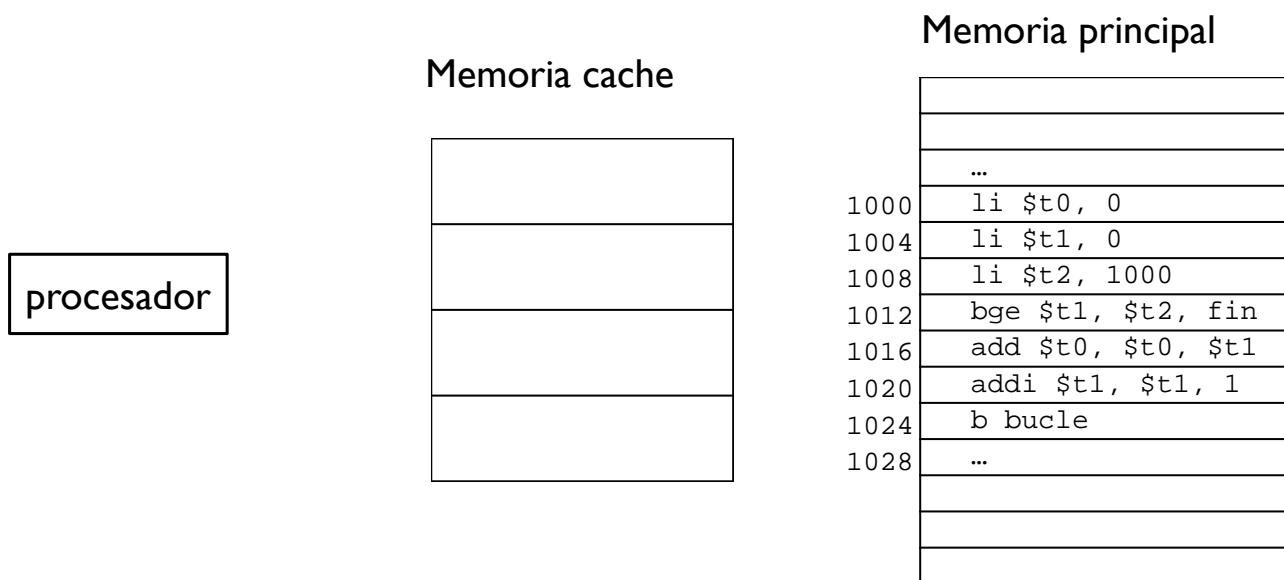
- ▶ **Sin memoria caché:**
  - ▶ Número de accesos a memoria =  $3 + 4 \times 1000 + 1 = 4004$  accesos
  - ▶ Tiempo de acceso a memoria =  $4004 \times 120 = 480480$  ns = **480.480 ms**

# Ejemplo de funcionamiento

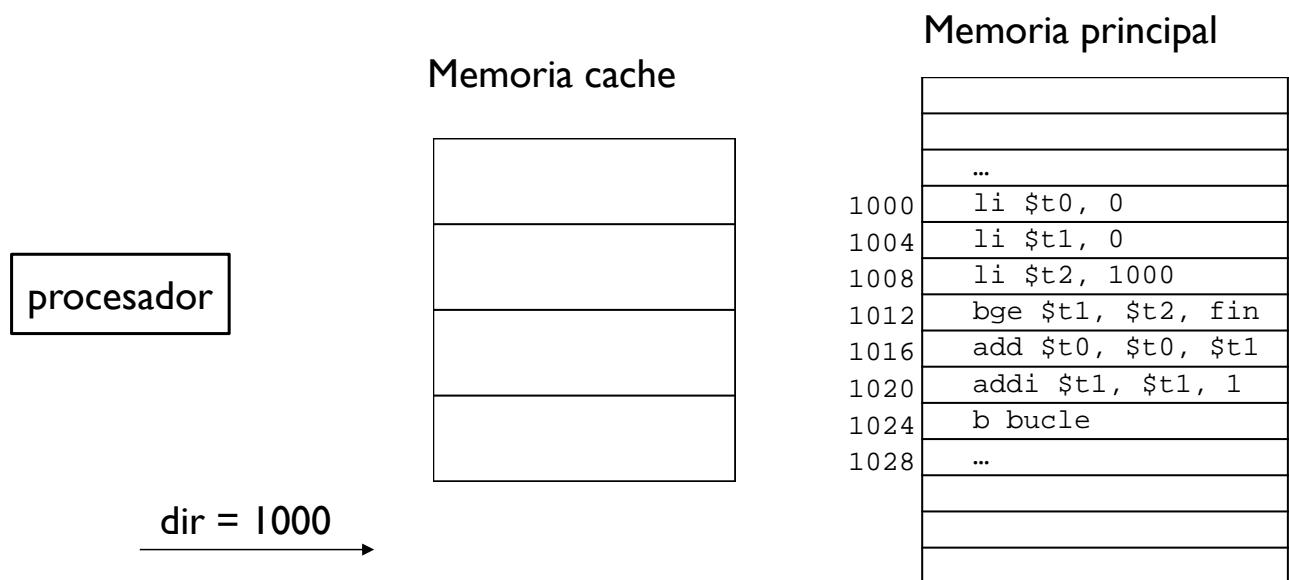
```
int i;                                li    $t0, 0      // s
int s = 0;                            li    $t1, 0      // i
for (i=0; i < 1000; i++)          bucle: li    $t2, 1000
                                         bge   $t1, $t2, fin
                                         add   $t0, $t0, $t1
                                         addi  $t1, $t1, 1
                                         b     bucle
                                         fin: ...
```

- ▶ Con memoria caché (bloque de 4 palabras):

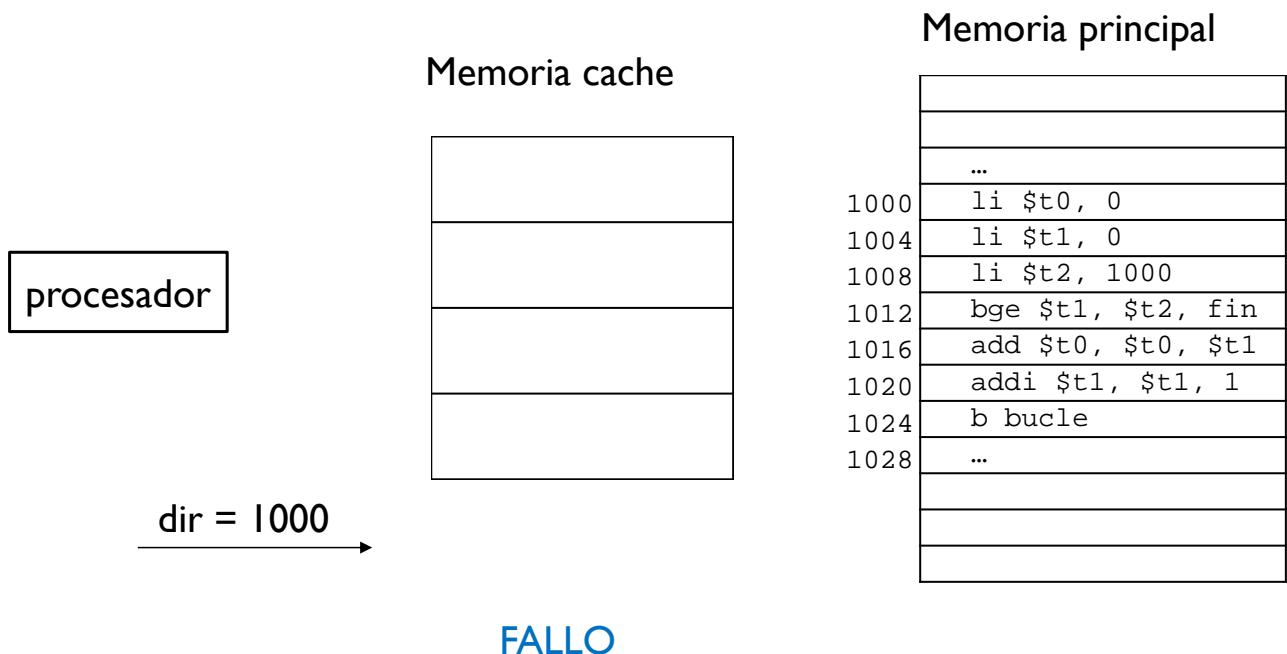
# Ejemplo de funcionamiento



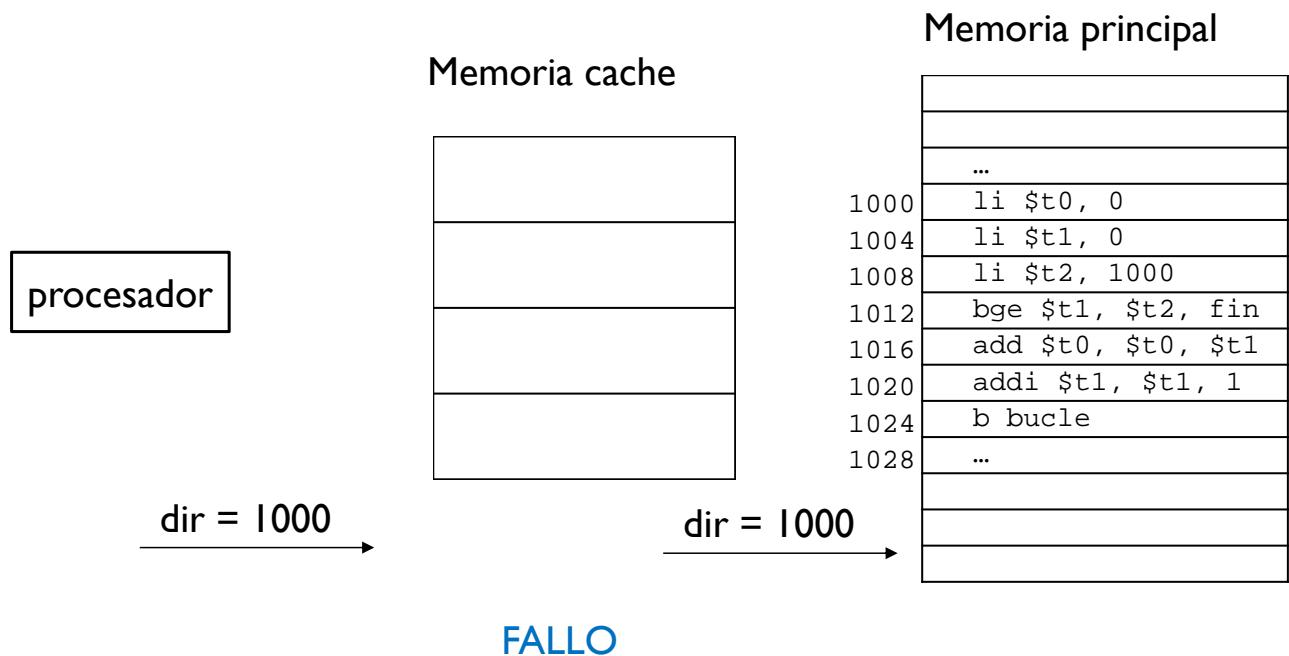
# Ejemplo de funcionamiento



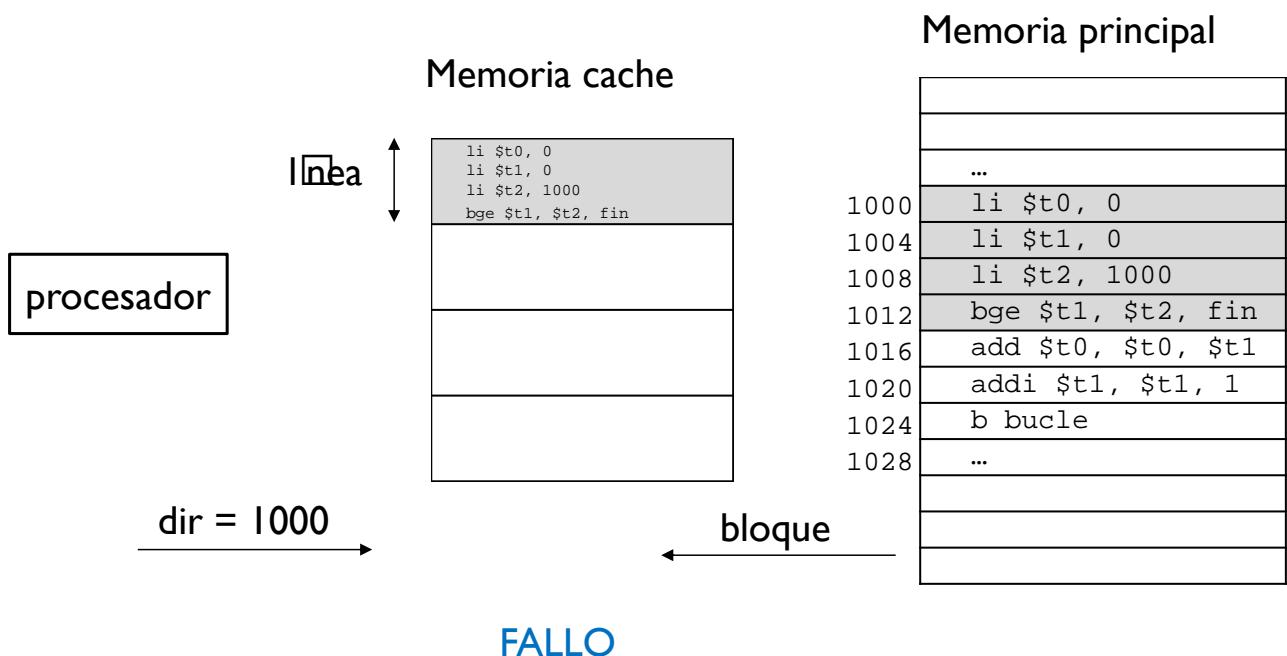
# Ejemplo de funcionamiento



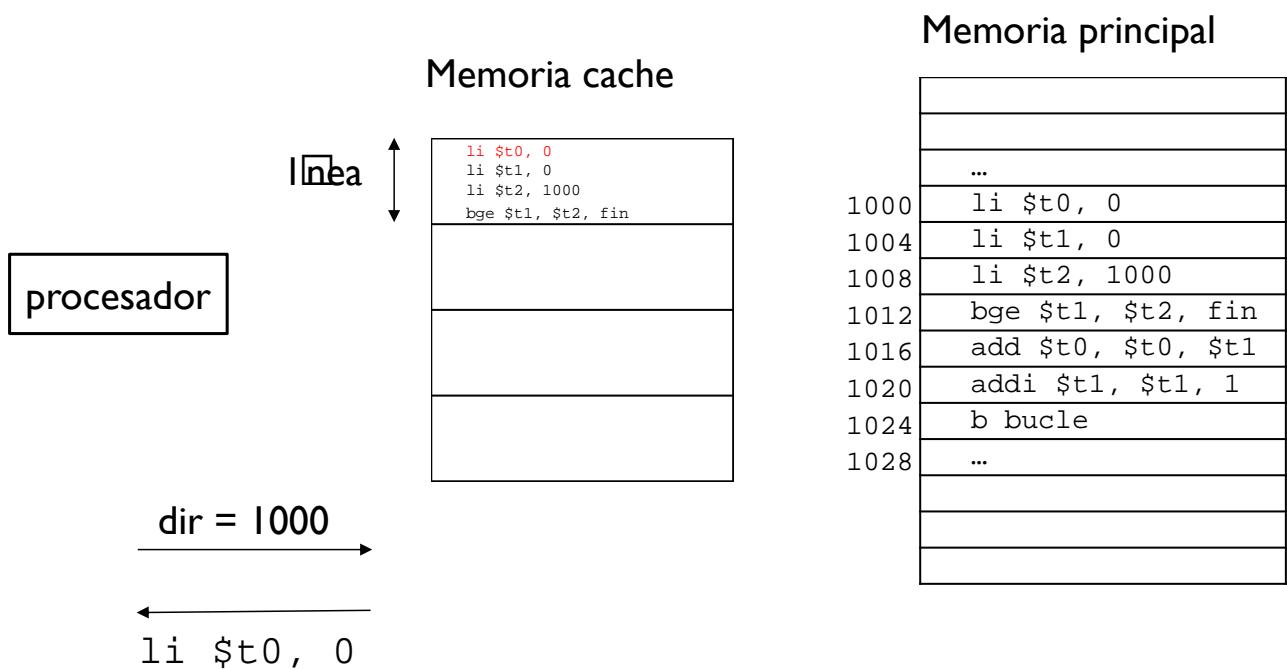
# Ejemplo de funcionamiento



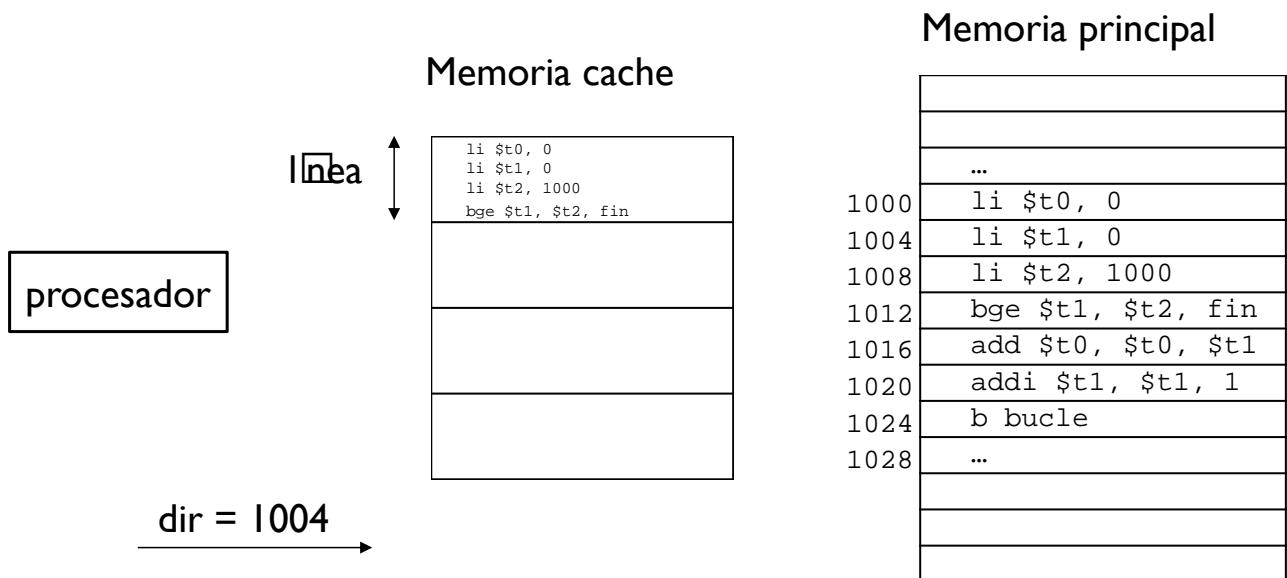
# Ejemplo de funcionamiento



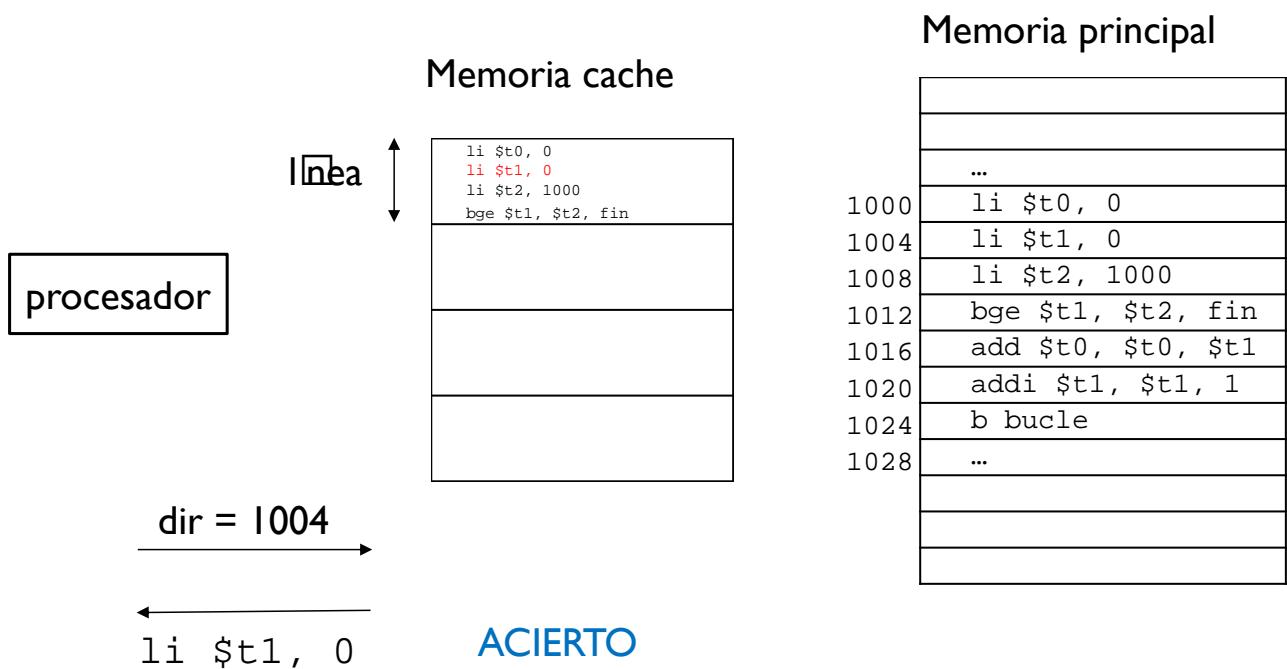
# Ejemplo de funcionamiento



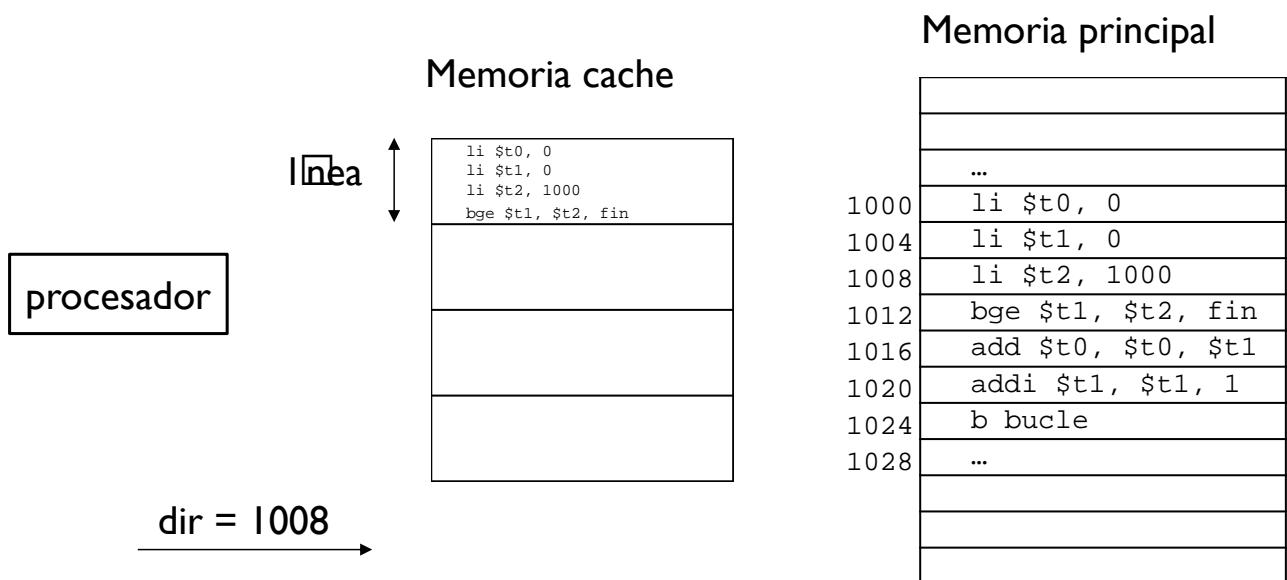
# Ejemplo de funcionamiento



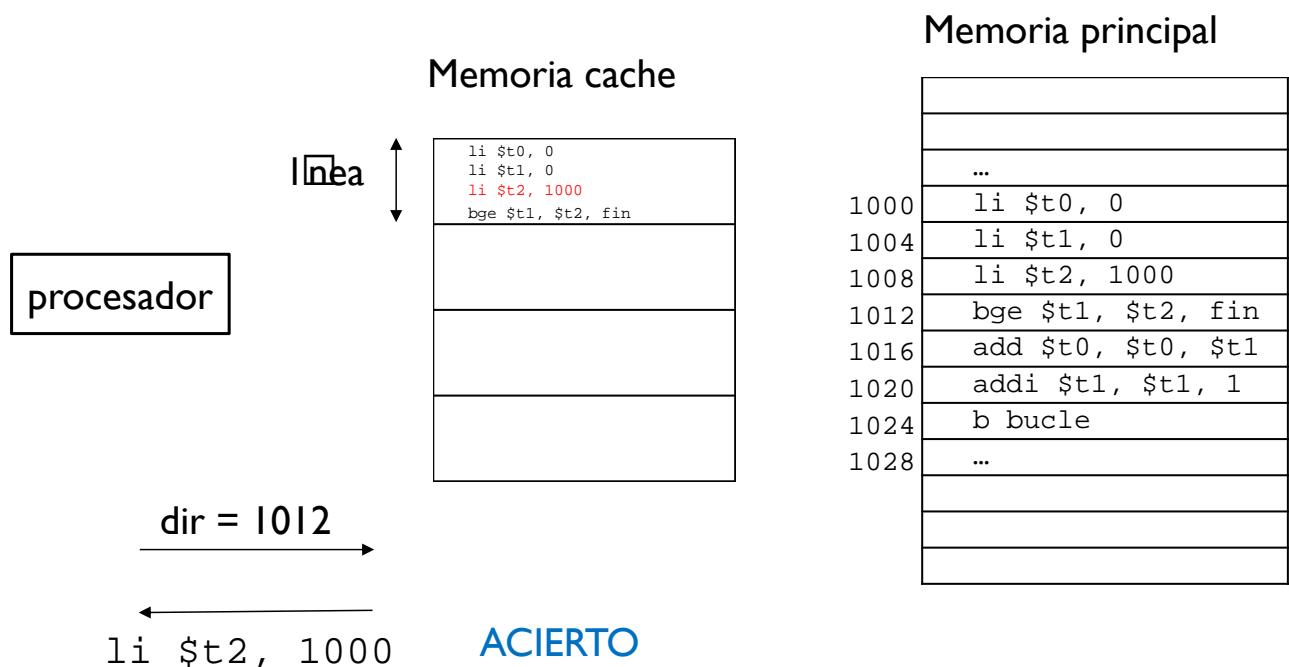
# Ejemplo de funcionamiento



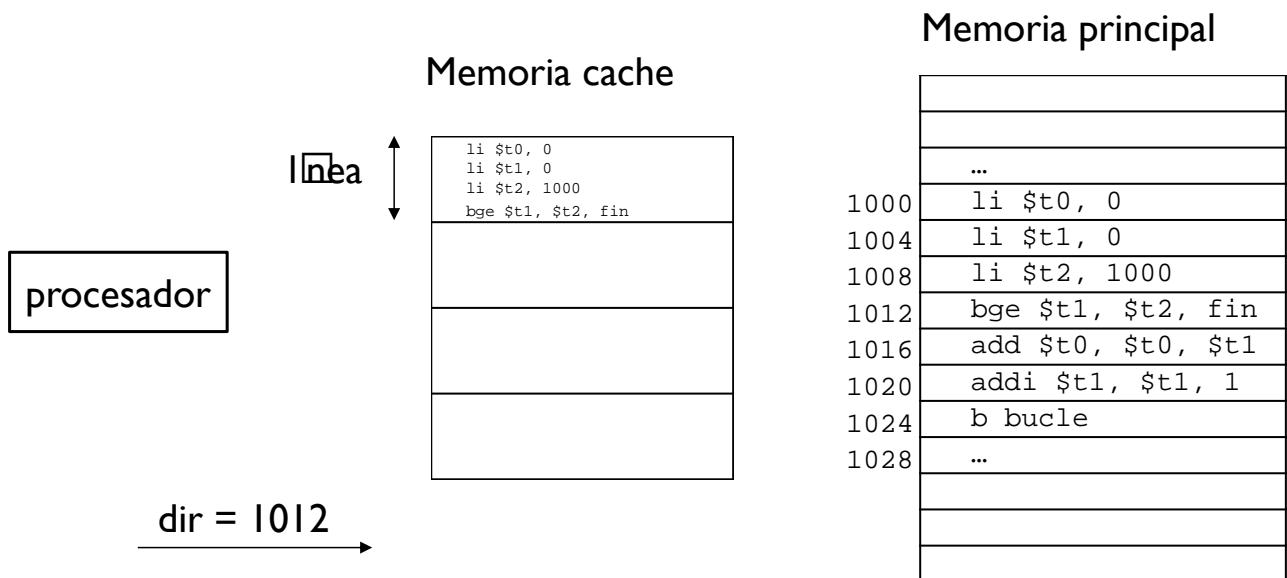
# Ejemplo de funcionamiento



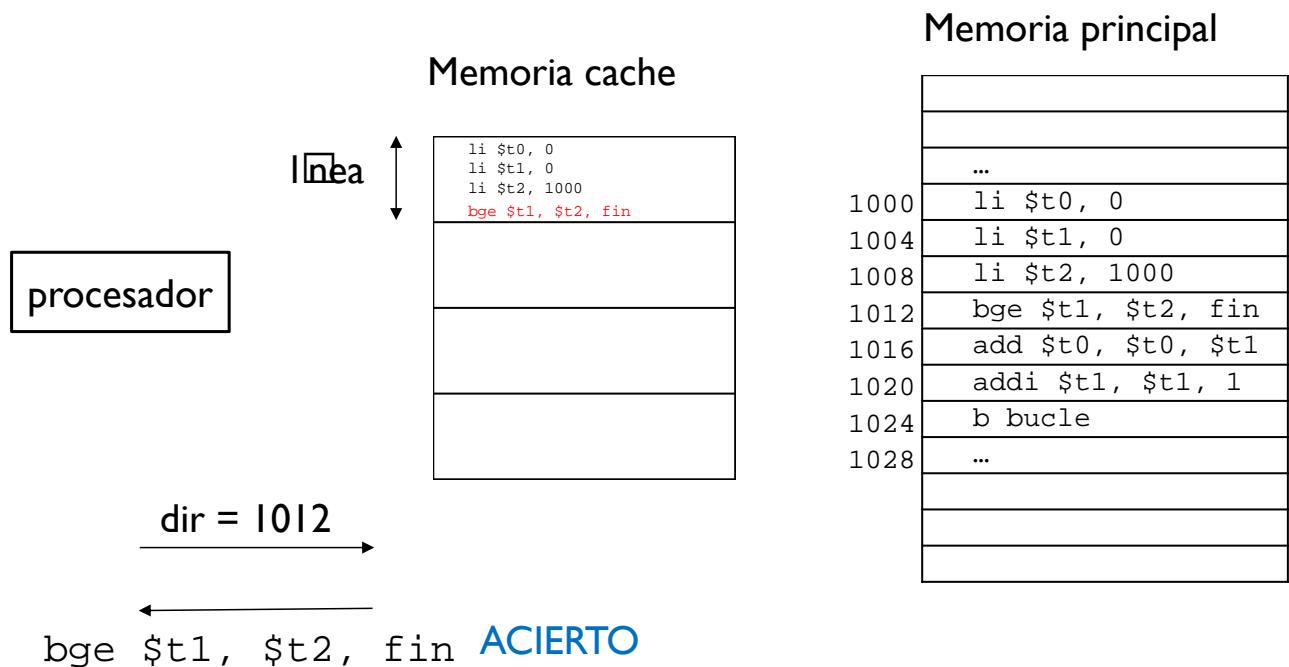
# Ejemplo de funcionamiento



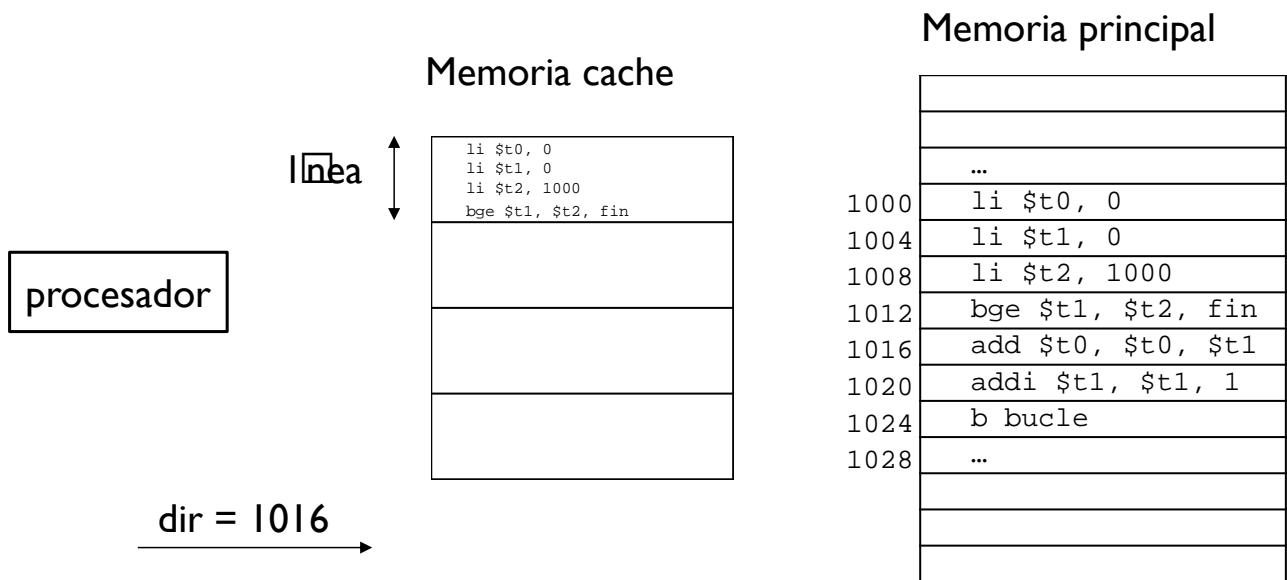
# Ejemplo de funcionamiento



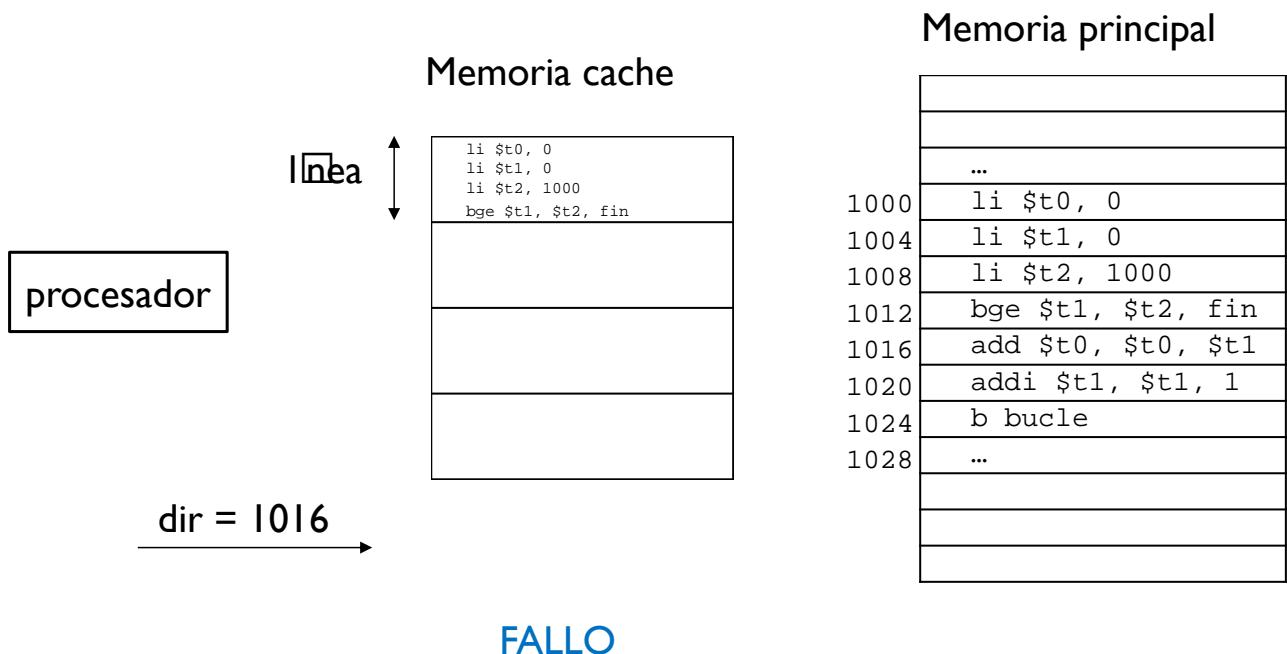
# Ejemplo de funcionamiento



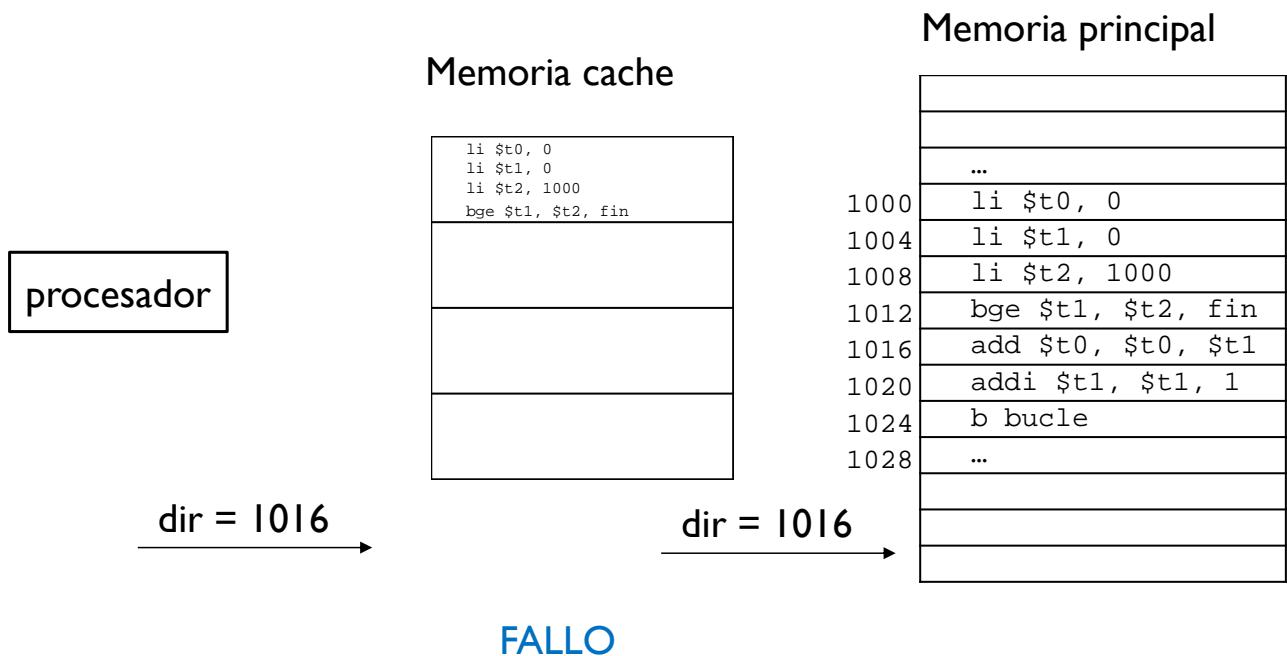
# Ejemplo de funcionamiento



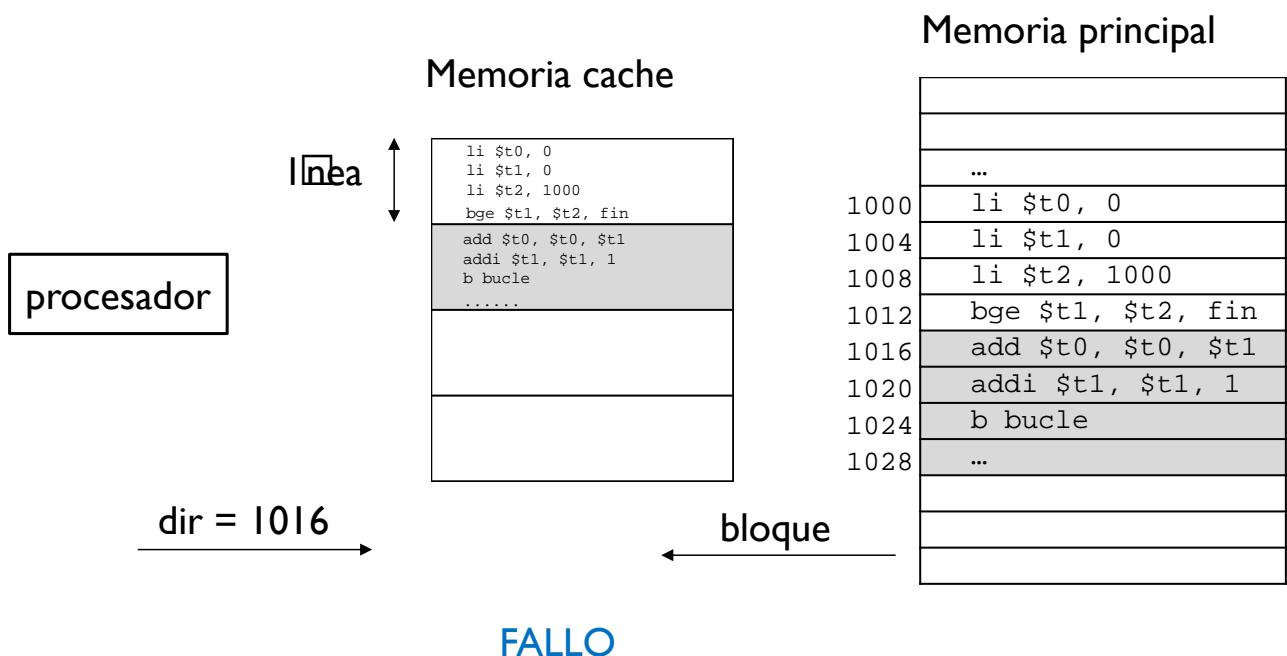
# Ejemplo de funcionamiento



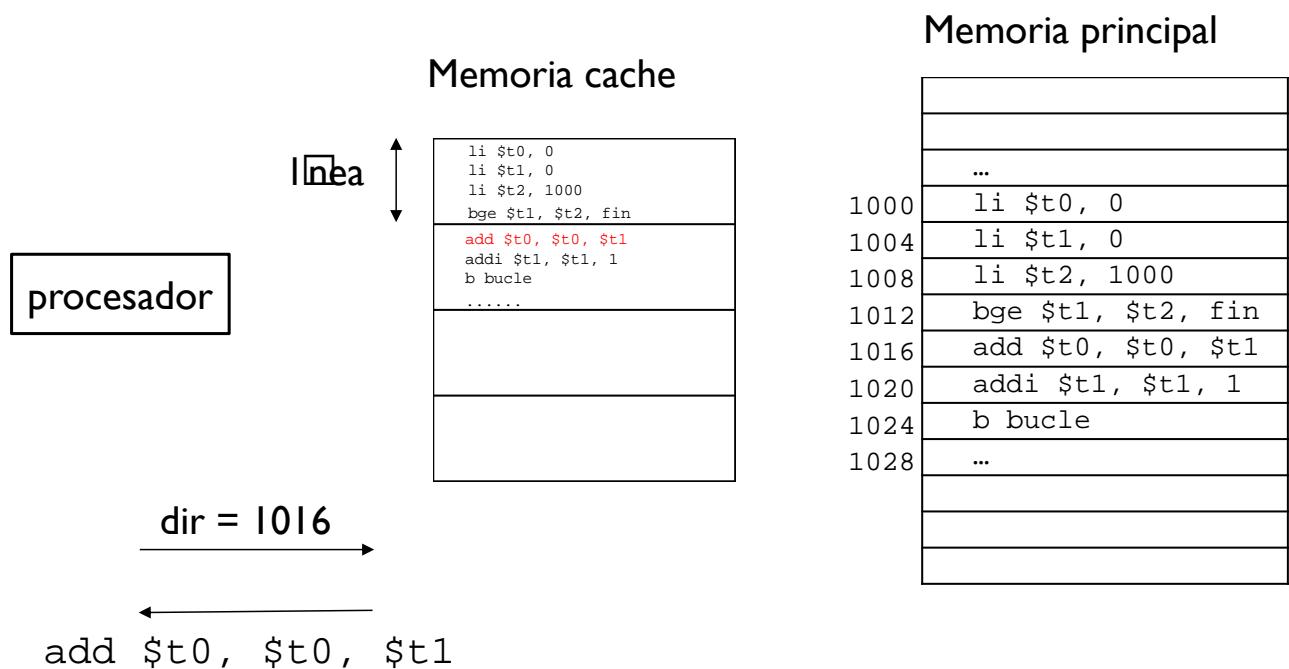
# Ejemplo de funcionamiento



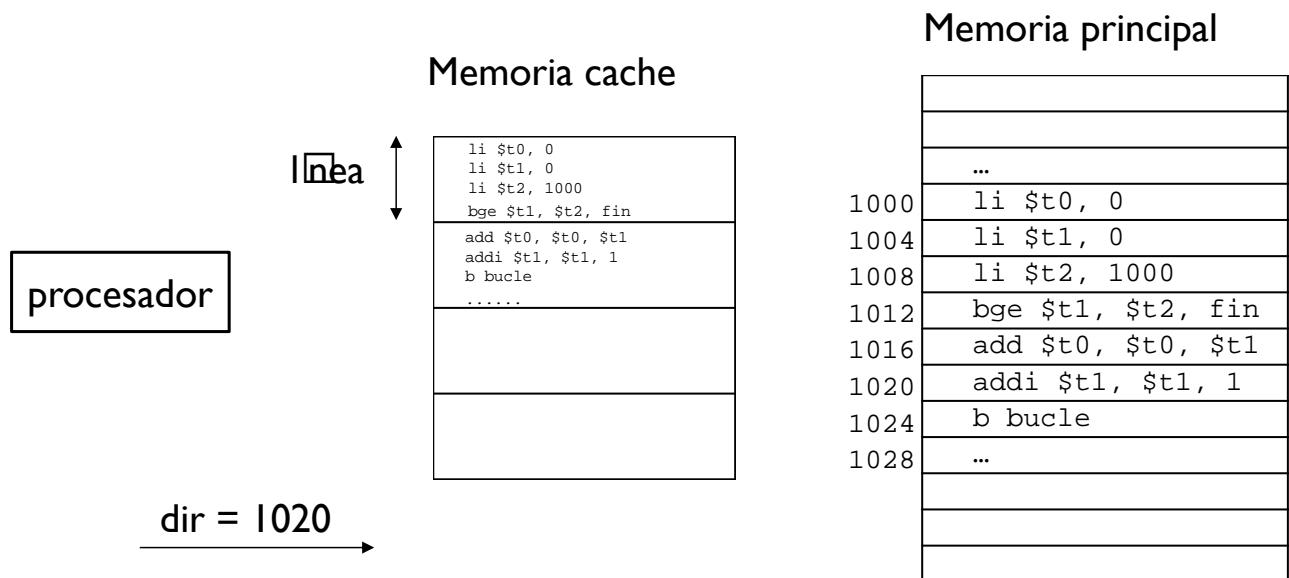
# Ejemplo de funcionamiento



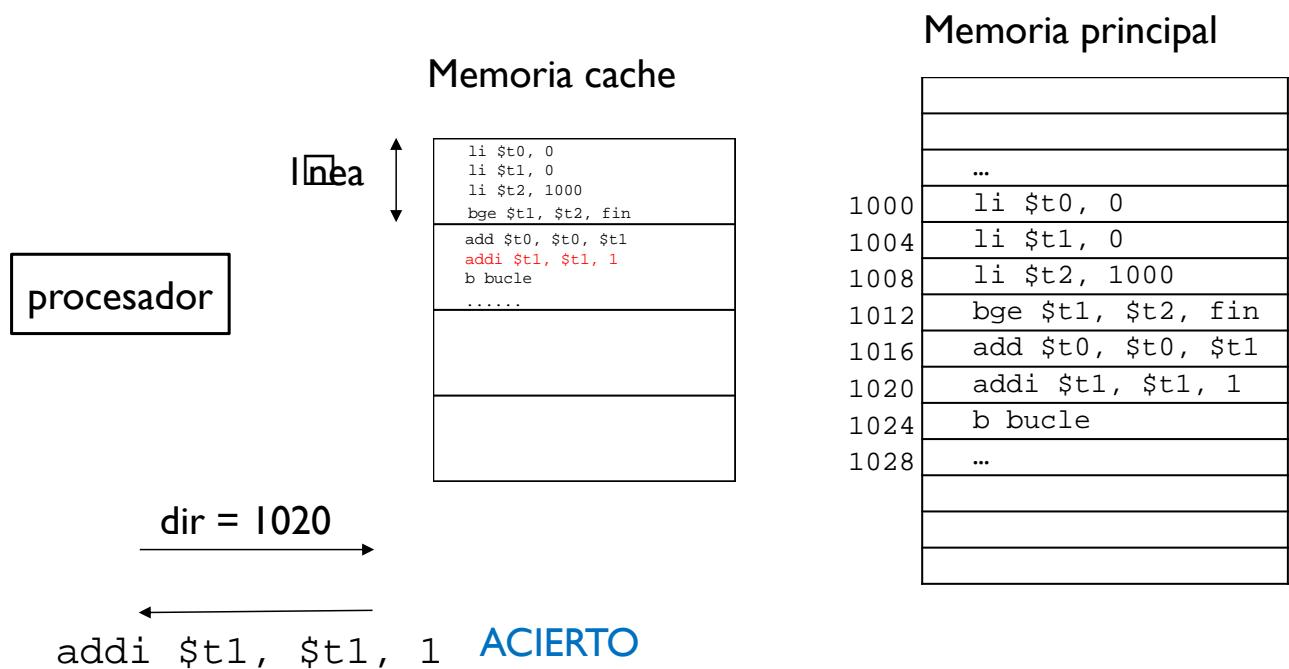
# Ejemplo de funcionamiento



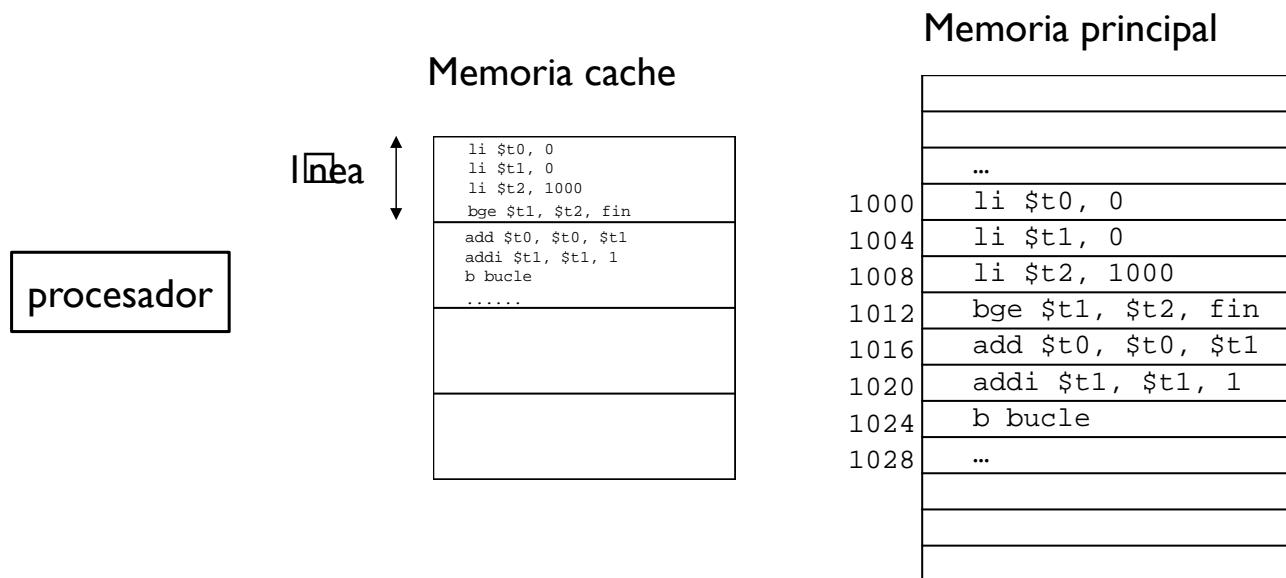
# Ejemplo de funcionamiento



# Ejemplo de funcionamiento



# Ejemplo de funcionamiento



El resto de accesos son ACIERTOS

# Ejemplo de funcionamiento

```
int i;  
int s = 0;  
for (i=0; i < 1000; i++)  
    s = s + i;
```

```
bucle:  
    li $t0, 0      // s  
    li $t1, 0      // i  
    li $t2, 1000  
    bge $t1, $t2, fin  
    add $t0, $t0, $t1  
    addi $t1, $t1, 1  
    b bucle  
fin:  
    ...
```

- ▶ Con memoria caché:
  - ▶ Número de bloques = 2

# Ejemplo de funcionamiento

```
int i;
int s = 0;
for (i=0; i < 1000; i++)
    s = s + i;
```

li \$t0, 0 // s  
li \$t1, 0 // i  
li \$t2, 1000  
bucle: bge \$t1, \$t2, fin  
add \$t0, \$t0, \$t1  
addi \$t1, \$t1, 1  
b bucle

fin: ...

▶ Con memoria caché:

- ▶ Número de bloques = 2
- ▶ Número de fallos = 2
- ▶ Tiempo para transferir 2 bloques =  $200 \times 2 = 400$  ns
- ▶ Tiempo de acceso a la caché =  $4004 \times 2 = 8008$  ns
- ▶ Tiempo total = 8408 ns

- ▶ Acceso a caché: 2 ns
- ▶ Acceso a MP: 120 ns
- ▶ Bloque de caché: 4 palabras
- ▶ Transferencia de un bloque entre memoria principal y caché: 200 ns

# Ejemplo de funcionamiento

```
int i;                                li    $t0, 0      // s
int s = 0;                            li    $t1, 0      // i
for (i=0; i < 1000; i++)          bucle: li    $t2, 1000
                                         bge   $t1, $t2, fin
                                         add   $t0, $t0, $t1
                                         addi  $t1, $t1, 1
                                         b     bucle
```

- ▶ Sin memoria caché: 480480 ns
- ▶ Con memoria caché = 8408 ns
- ▶ Tasa de aciertos a la caché = 4002 / 4004 => 99,95 %

# Ejercicio

## Calcular tasa de aciertos

```
int v[1000]; // global      .data:  
                           v: .space 4000  
  
int i;  
int s;                      .text:  
for (i=0; i < 1000; i++)  
    s = s + v[i];  
  
▶ Ejemplo:  
▶ Acceso a caché: 2 ns  
▶ Acceso a MP: 120 ns  
▶ Bloque de MP: 4 palabras  
▶ Transferencia de un bloque  
entre memoria principal y  
caché : 200 ns
```

```
          li    $t0, 0      // i  
          li    $t1, 0      // i de v  
          li    $t2, 1000   // componentes  
          li    $t3, 0      // s  
bucle: bge  $t0, $t2, fin  
       lw    $t4, v($t1)  
       addi $t3, $t3, $t4  
       addi $t0, $t0, 1  
       addi $t1, $t1, 4  
       b     bucle  
fin:    ...
```

## ¿Por qué funciona?

- ▶ Tiempo de acceso a caché mucho menor que tiempo de acceso a memoria principal.
- ▶ A la memoria principal se accede por bloques.
- ▶ Cuando un programa accede a una dirección, es probable que vuelva a acceder a ella en el futuro cercano.
  - ▶ Proximidad o Localidad temporal.
- ▶ Cuando un programa accede a una dirección, es probable que en el futuro cercano acceda a posiciones cercanas.
  - ▶ Proximidad o Localidad espacial.
- ▶ Tasa de aciertos: probabilidad de que un dato accedido esté en la caché

Tasa de aciertos elevada

# Ejemplos de tiempos de acceso

- ▶ **Memoria principal.**
  - ▶ Tecnología DRAM o similar.
  - ▶ Tiempo de acceso: 20-50 ns.
- ▶ **Memoria caché.**
  - ▶ Tecnología SRAM o similar.
  - ▶ Tiempo de acceso: <1-2.5 ns.

## Tiempo medio de acceso a caché

- ▶ Tiempo medio de acceso a un sistema de memoria con dos niveles

$$\begin{aligned} T_m &= h \cdot T_a + (1-h) \cdot (T_a + T_f) \\ &= T_a + (1-h) \cdot T_f \end{aligned}$$

- ▶  $T_a$ : tiempo de acceso a la caché
- ▶  $T_f$ : tiempo en tratar el fallo, tiempo en remplazar un bloque y traerlo de memoria principal a caché
- ▶  $h$ : tasa de aciertos

# Rendimiento general

$$\begin{aligned} T_m &= h \cdot T_a + (1-h) \cdot (T_a + T_f) \\ &= T_a + (1-h) \cdot T_f \end{aligned}$$

1. El procesador realiza un acceso a memoria caché.
2. La cache comprueba si ya están los datos de esta posición:
  - ▶ Si está (**ACIERTO**),
    - 3.A.1 Se la sirve al procesador desde la cache (rápidamente):  $T_a$
  - ▶ Si no está (**FALLO**),
    - 3.B.1 La cache transfiere de memoria principal el bloque asociado a la posición:  $T_f$
    - 3.B.2 Después, la cache entrega los datos pedidos al procesador :  $T_a$

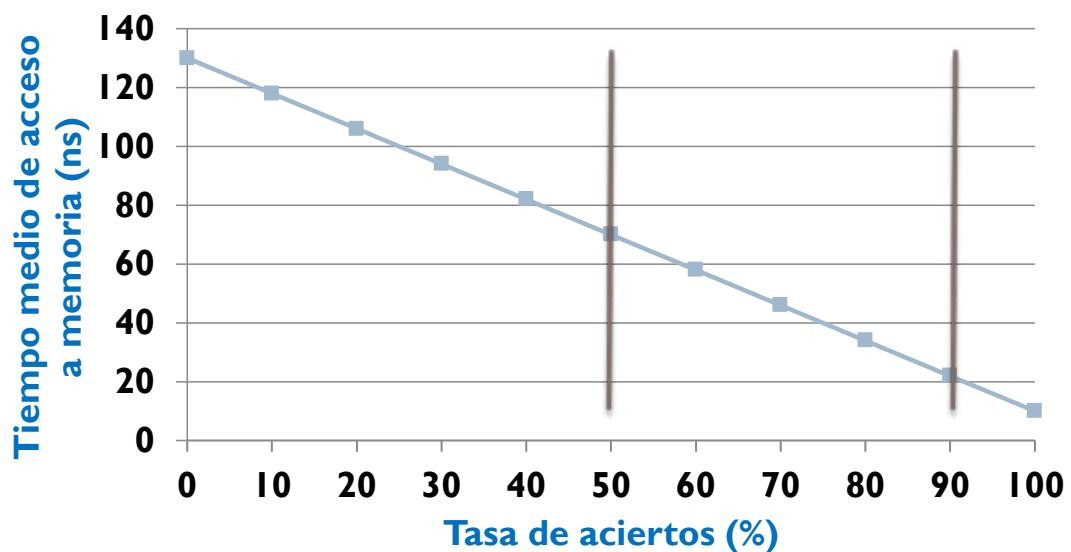
# Ejemplo

$$\begin{aligned} T_m &= h \cdot T_a + (1-h) \cdot (T_a + T_f) \\ &= T_a + (1-h) \cdot T_f \end{aligned}$$

1.  $T_a$ : Tiempo de acceso a caché = **10 ns**
2.  $T_f$ : Tiempo de acceso a memoria principal = **120 ns**
3.  $h$ : tasa de aciertos -> **X = 0.1, 0.2, ..., 0.9, 1.0**  
**10%, 20%, ..., 90%, 100%**

## Ejemplo

$$\begin{aligned} T_m &= h \cdot T_a + (1-h) \cdot (T_a + T_f) \\ &= T_a + (1-h) \cdot T_f \end{aligned}$$

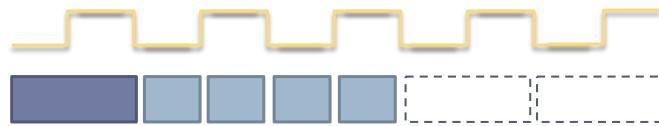


# Acceso por bloques

- ▶ Se premia el acceso a posiciones consecutivas de memoria
  - ▶ Ejemplo 1:  
acceder a 5 posiciones de memoria **individuales** (no consecutivas)

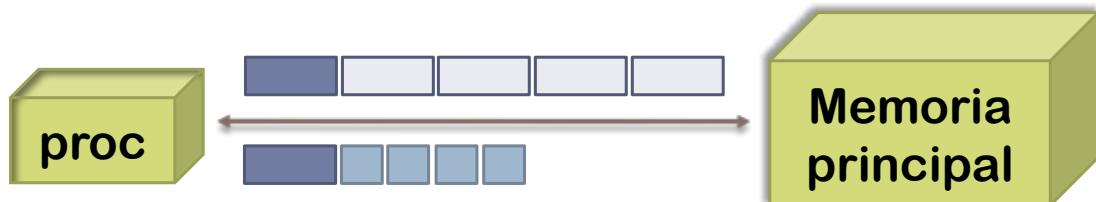


- ▶ Ejemplo 2:  
acceder a 5 posiciones de memoria **consecutivas**



## Acceso por bloques

- ▶ Si se accede a un conjunto consecutivo de posiciones de memoria, los accesos siguientes al primero tienen un coste menor.



## Niveles de memoria caché

- ▶ Es habitual encontrar tres niveles:

- ▶ L1 o nivel 1:

- ▶ Caché interna: la más cercana al procesador
    - ▶ Tamaño pequeño (8KB-128KB) y máxima velocidad
    - ▶ Pueden separarse para instrucciones y datos

- ▶ L2 o nivel 2:

- ▶ Caché interna
    - ▶ Entre L1 y L3 (o entre L1 y M.Principal)
    - ▶ Tamaño mediano (256KB – 4MB) y menor velocidad que L1

- ▶ L3 o nivel 3:

- ▶ Tipicamente último nivel antes de M. Principal
    - ▶ Tamaño mayor y menor velocidad que L2
    - ▶ Interna o externa al procesador

# Ejercicio

- ▶ **Computador:**
  - ▶ Tiempo de acceso a caché: 4 ns
  - ▶ Tiempo de acceso a bloque de MP: 120 ns.
- ▶ Si se tiene una tasa de aciertos del 90%. ¿Cuál es el tiempo medio de acceso?
- ▶ Tasas de acierto necesarias para que el tiempo medio de acceso sea menor de 10 ns y 5 ns.

## Ejercicio (solución)

- ▶ **Computador:**
  - ▶ Tiempo de acceso a caché: 4 ns
  - ▶ Tiempo de acceso a bloque de MP: 120 ns.
- ▶ Si se tiene una tasa de aciertos del 90%. ¿Cuál es el tiempo medio de acceso?

$$T_m = 4 \times 0.9 + (120 + 4) \times 0.1 = 16 \text{ ns}$$

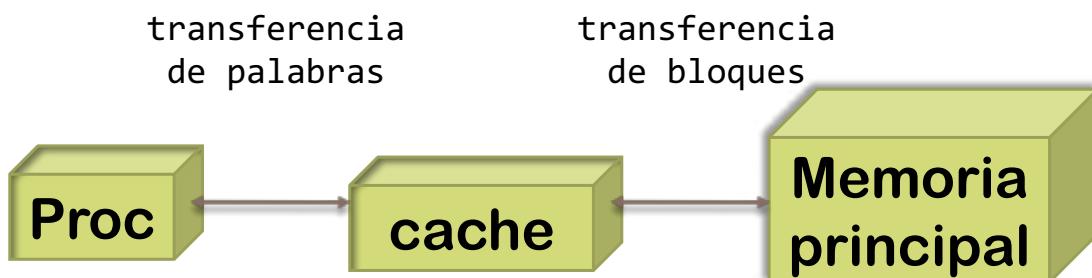
- ▶ Tasas de acierto necesarias para que el tiempo medio de acceso sea menor de 5 ns.

$$\begin{aligned} 5 &= 4 \times h + (120 + 4) \times (1 - h) = 16 \text{ ns} \Rightarrow \\ &\Rightarrow h > 0.9916 \end{aligned}$$

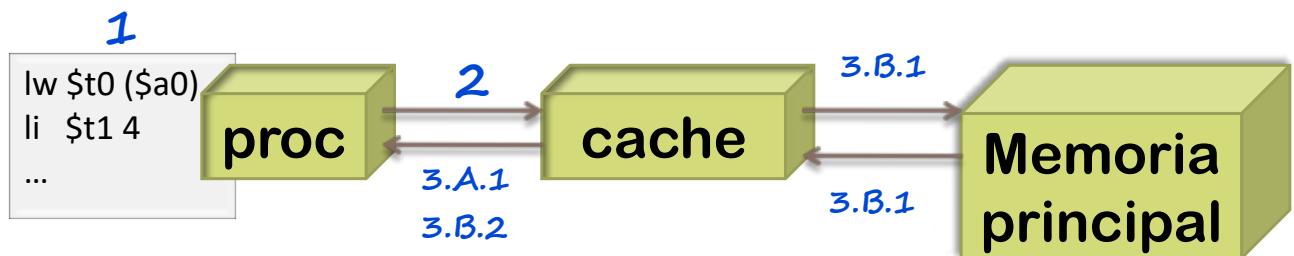
# Memoria caché

## Resumen

- ▶ Se construye con tecnología SRAM
  - ▶ Integrada en el mismo procesador.
  - ▶ Más rápida y más cara que la memoria DRAM.
- ▶ Mantiene una **copia** de partes de la memoria principal.

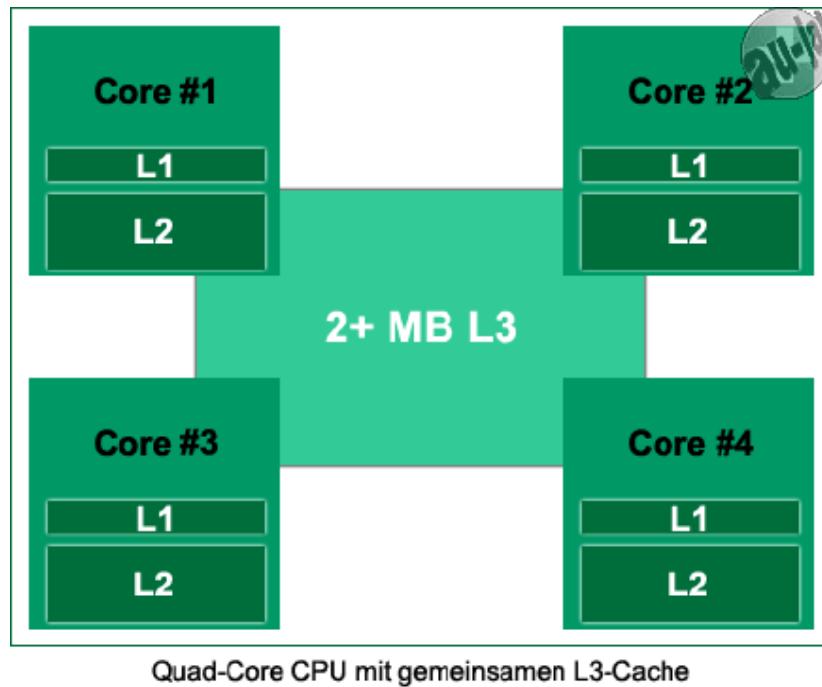


# Funcionamiento general

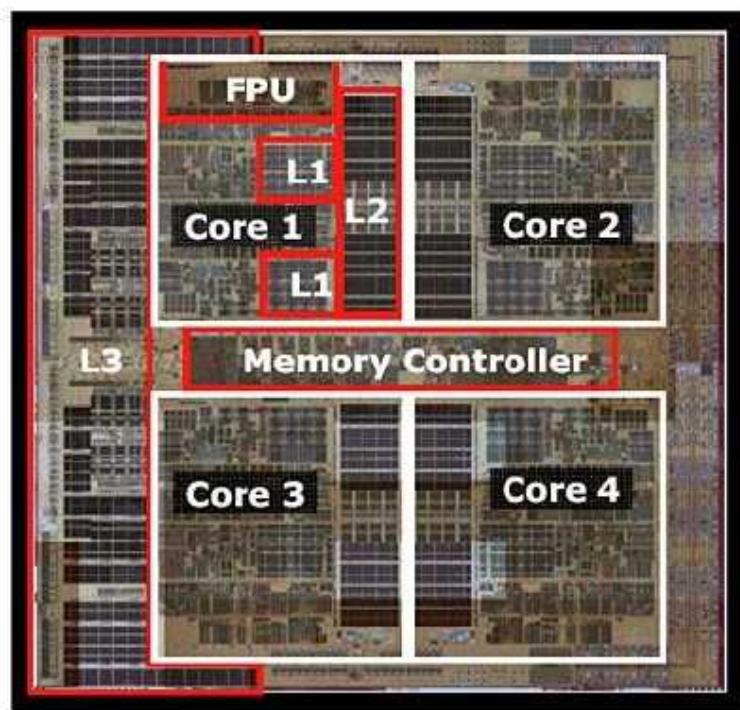


1. El procesador solicita contenidos de una posición (dirección) de memoria.
2. La cache comprueba si ya están los datos de esta posición:
  - Si está (AICERTO),**
    - 3.A.1** Se la sirve al procesador desde la cache (rápidamente).
  - Si no está (FALLO),**
    - 3.B.1** La cache transfiere de memoria principal el bloque de palabras asociado a la posición.
    - 3.B.2** Después, la cache entrega los datos pedidos a la procesador.

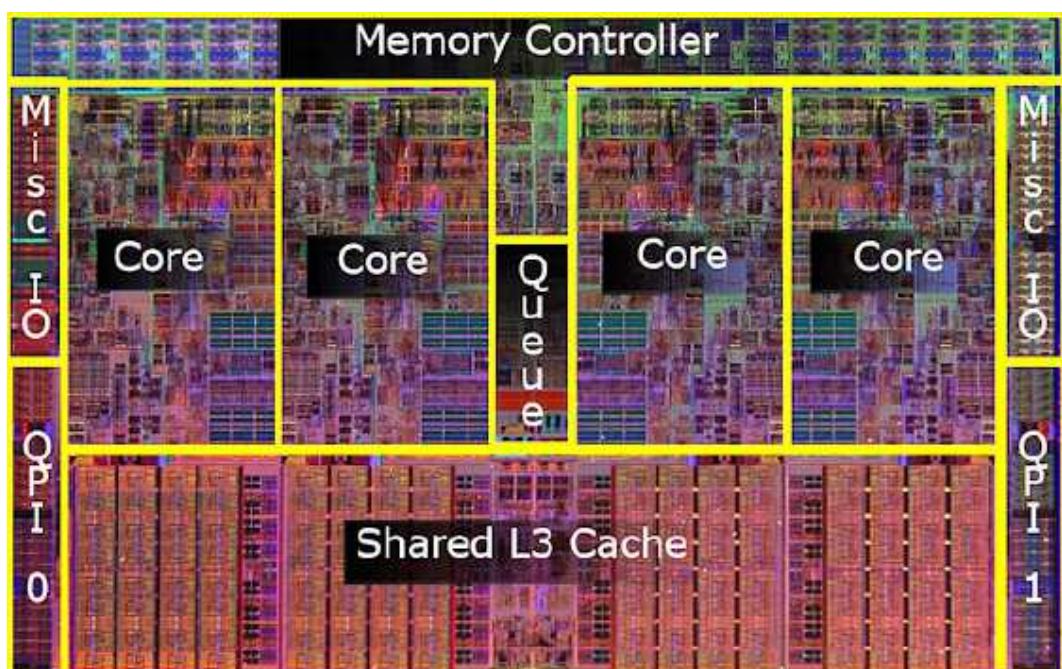
## Ejemplo: AMD quad-core



## Ejemplo: AMD quad-core



## Ejemplo: Intel Core i7

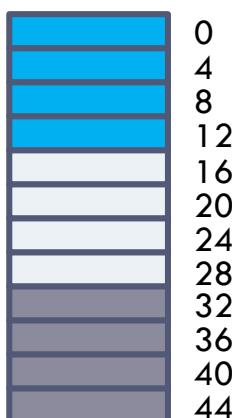


# Diseño y organización de la memoria caché

- I. Estructura de la memoria caché**
- 2. Diseño de la memoria cache**

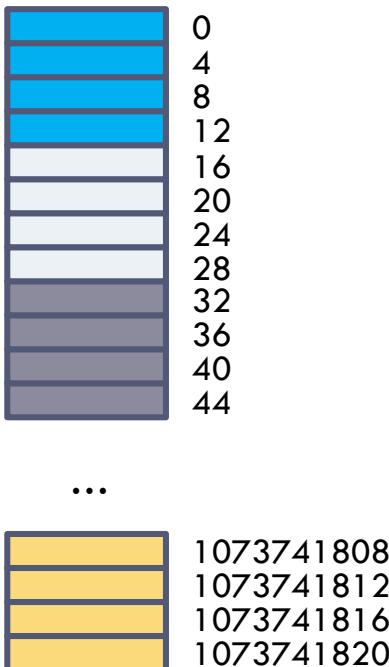
- ▶ Tamaño
- ▶ Función de correspondencia
- ▶ Algoritmo de sustitución
- ▶ Política de escritura

# Organización de la memoria principal



- ▶ La memoria principal *lógicamente* se divide en bloques de igual tamaño.
  - ▶ 1 bloque = k palabras.
- ▶ Ejercicio: ¿Cuantos bloques de 4 palabras hay en una memoria de 1 GB?

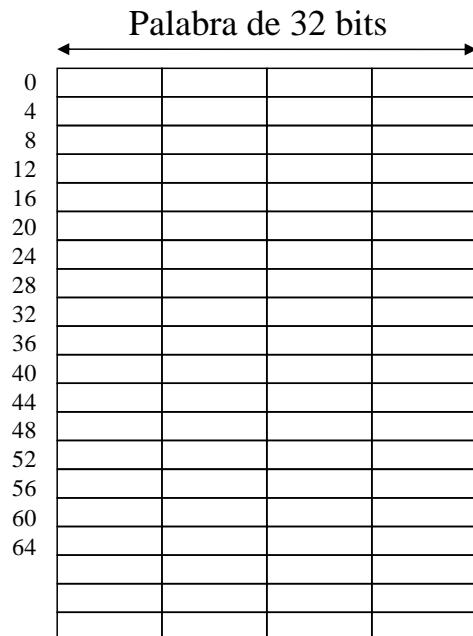
# Organización de la memoria principal



- ▶ La memoria principal *lógicamente* se divide en bloques de igual tamaño.
  - ▶ 1 bloque = k palabras.
- ▶ Ejercicio: ¿Cuantos bloques de 4 palabras hay en una memoria de 1 GB?
- ▶ Solución:  
$$2^{30} \text{ B} / 16 \text{ B} = 2^{30-4} \text{ B} = 2^{26} \text{ B} = 64 \text{ megablockes} \approx 64 \text{ millones}$$

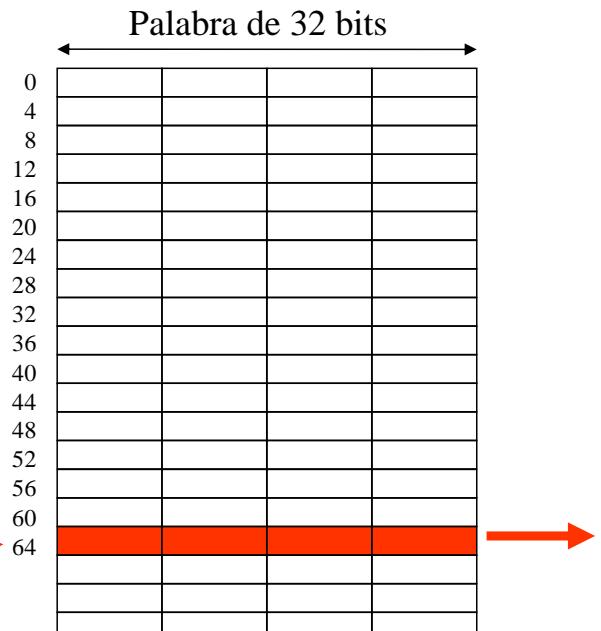
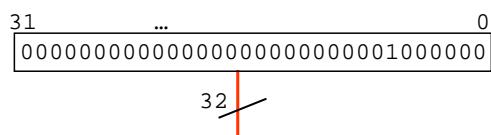
# Acceso a una palabra en memoria principal

- ▶ Ejemplo:
  - ▶ Computador de 32 bits
  - ▶ Memoria direccionada por bytes
  - ▶ Acceso a memorial principal por palabras
  - ▶ Acceso a la palabra con dirección  
**0x00000064**



# Acceso a una palabra en memoria principal

- ▶ Ejemplo:
  - ▶ Computador de 32 bits
  - ▶ Memoria direccionada por bytes
  - ▶ Acceso a memoria principal por palabras
  - ▶ Acceso a la palabra con dirección  
**0x00000064**



## Organización de la memoria caché

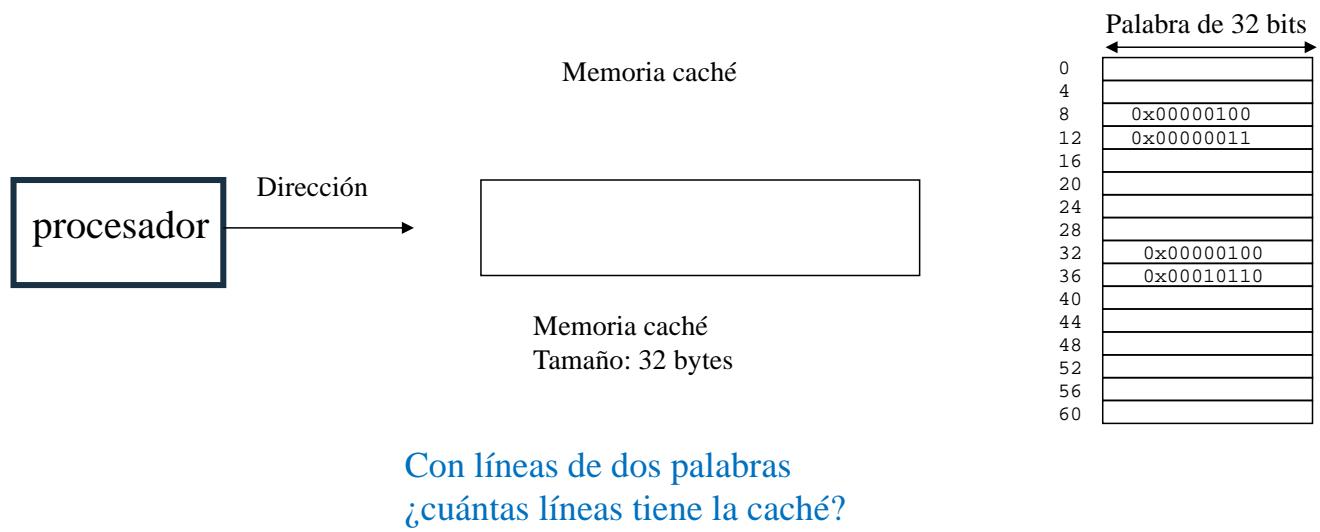
- ▶ La memoria caché está organizada en bloques (**I<sub>n</sub>ea**)
  - ▶ Al bloque de memoria caché se le llama **I<sub>n</sub>ea** de caché
- ▶ El tamaño del bloque de M. principal es igual al tamaño de la **I<sub>n</sub>ea**.
  - ▶ Pero el tamaño de la memoria caché es mucho menor.
  - ▶ El número de bloques que cabe en la memoria caché es muy pequeño.
- ▶ ¿Cuántos bloques de 4 palabras cabe en una memoria caché de 32 KB?

## Organización de la memoria caché

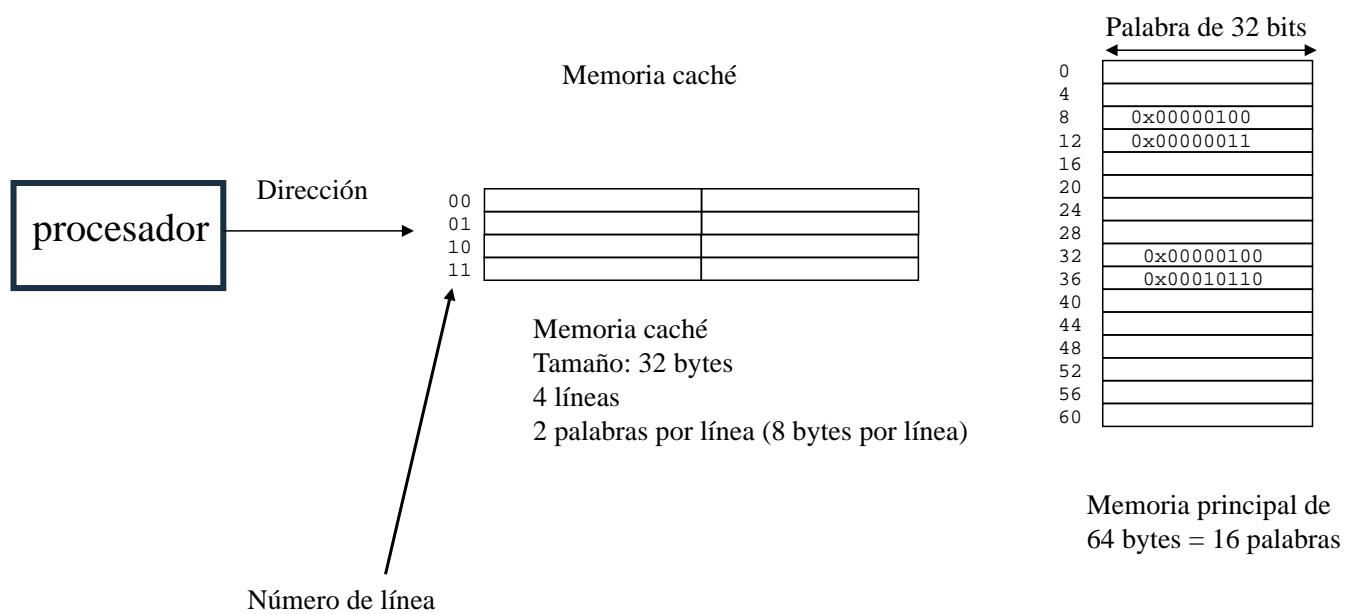
- ▶ La memoria caché está organizada en bloques (**I<sub>n</sub>eas**)
  - ▶ Al bloque de memoria caché se le llama **I<sub>n</sub>ea** de caché
- ▶ El tamaño del bloque de M. principal es igual al tamaño de la **I<sub>n</sub>ea**.
  - ▶ Pero el tamaño de la memoria caché es mucho menor.
  - ▶ El número de bloques que cabe en la memoria caché es muy pequeño.
- ▶ ¿Cuántos bloques de 4 palabras cabe en una memoria caché de 32 KB?
  - ▶ Solución:  
$$2^5 \cdot 2^{10} \text{ B} / 2^4 \text{ B} = 2^{11} \text{ bloques} = 2048 \text{ bloques} = 2048 \text{ I}_n\text{eas}$$

# ¿Cómo buscar una palabra en caché?

Ejemplo:



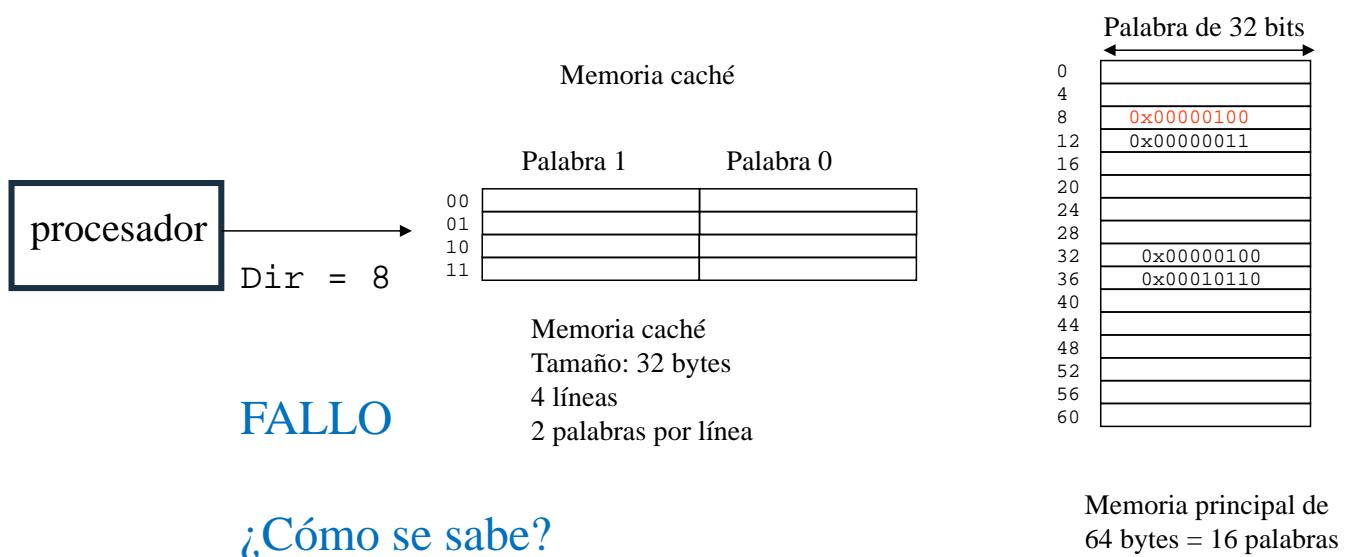
# ¿Cómo buscar una palabra en la cache?



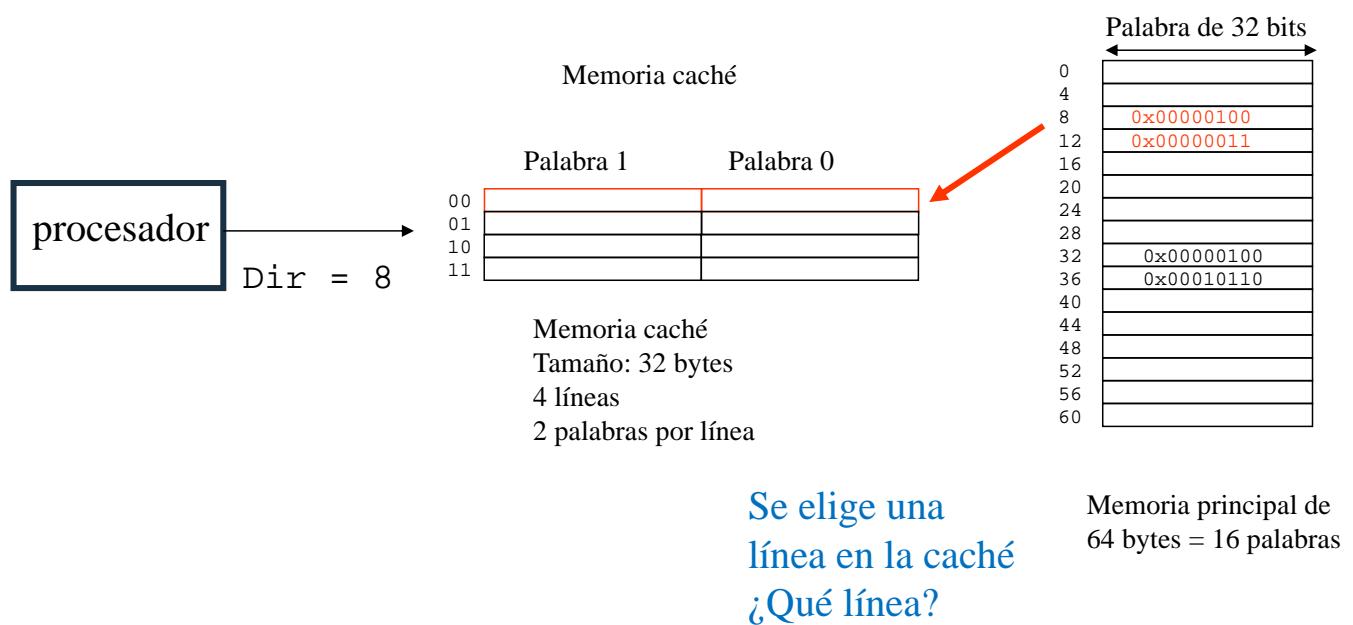
# ¿Cómo buscar una palabra en caché?



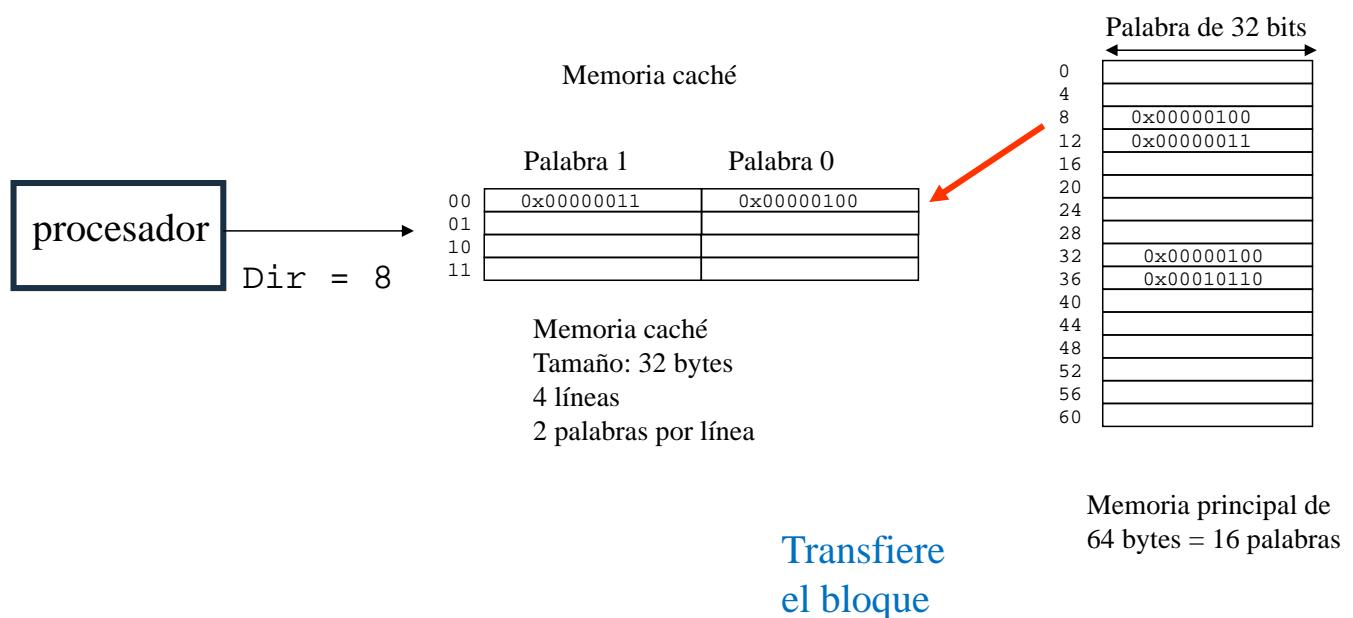
# ¿Cómo buscar una palabra en caché?



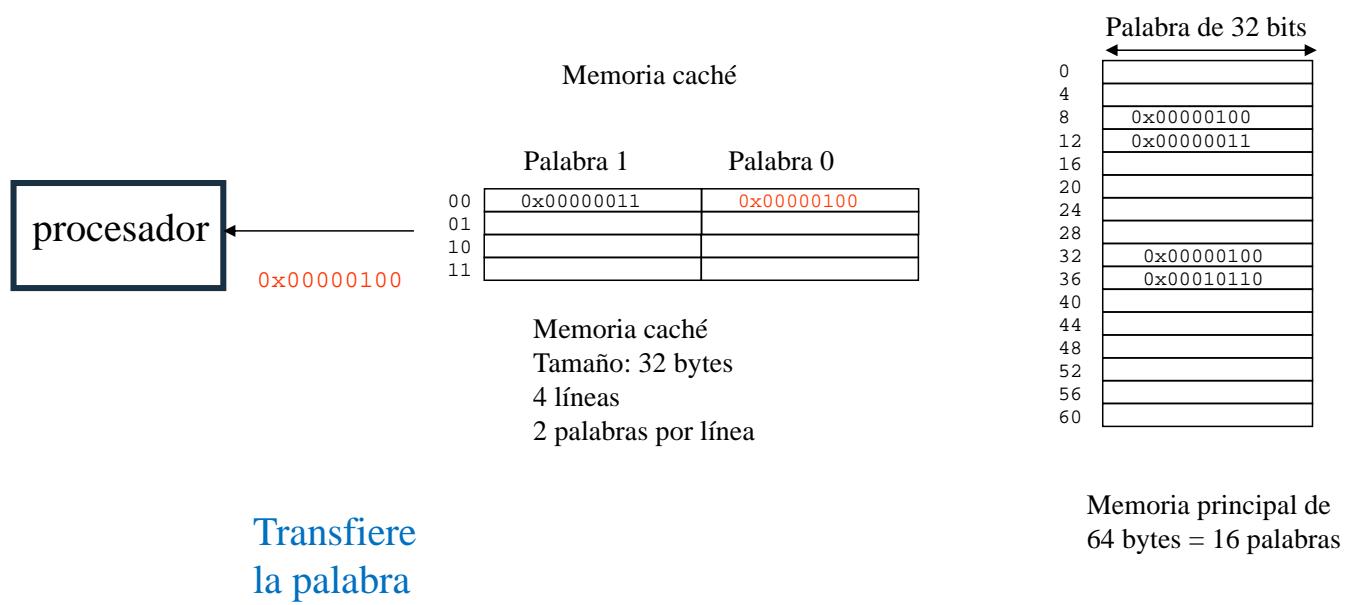
# ¿Cómo buscar una palabra en caché?



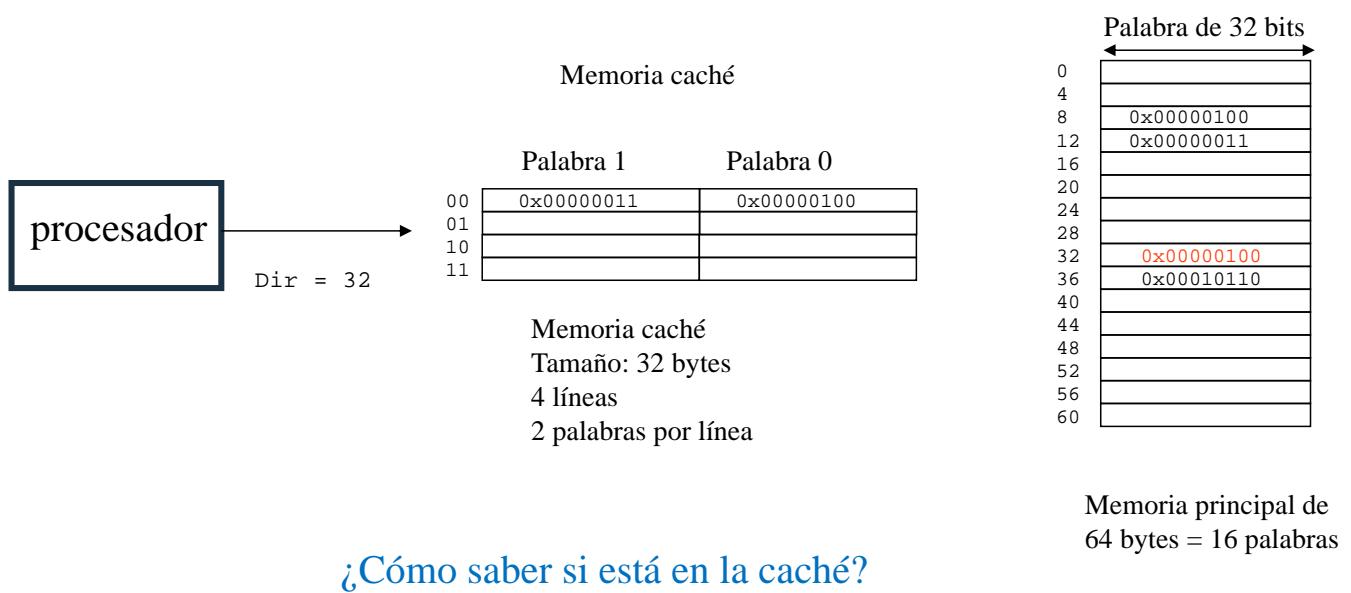
# ¿Cómo buscar una palabra en caché?



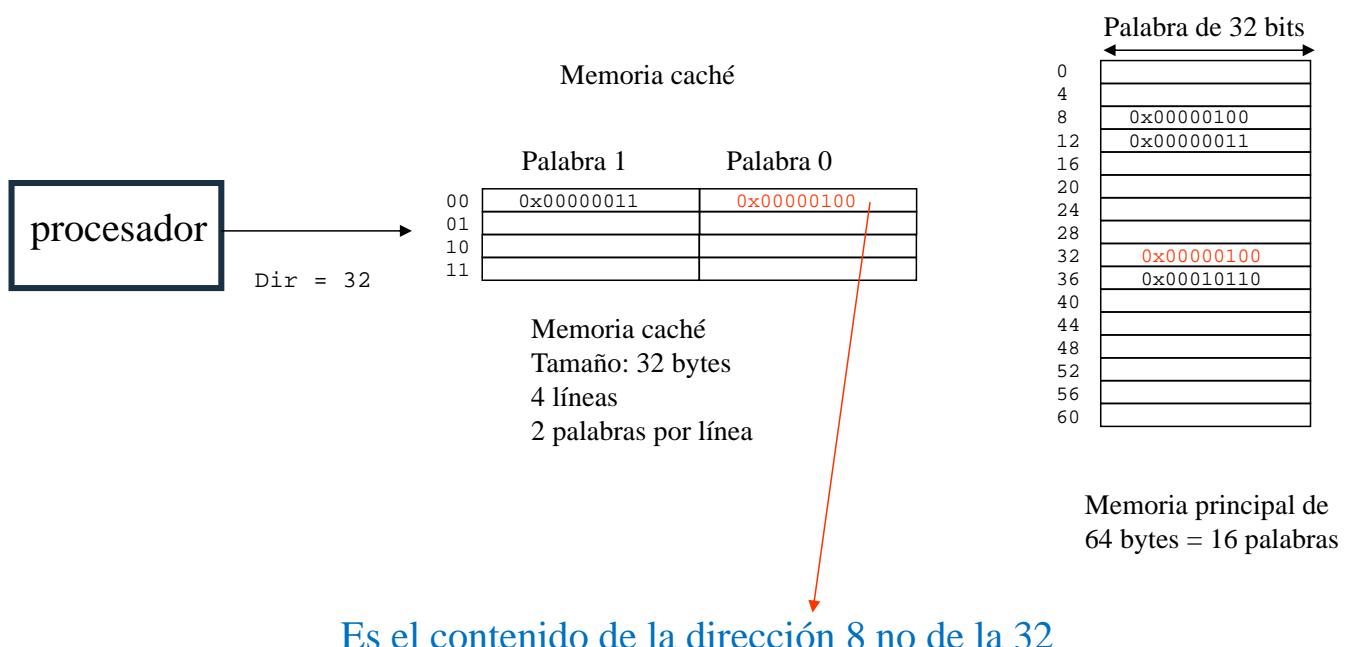
# ¿Cómo buscar una palabra en caché?



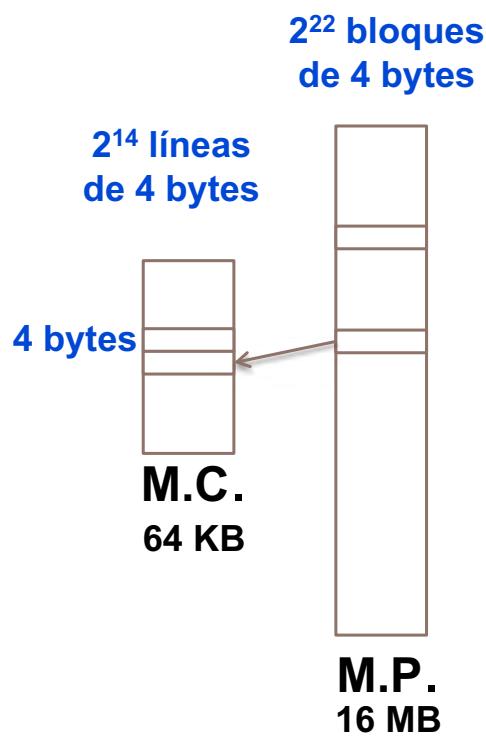
# ¿Cómo buscar una palabra en caché?



# ¿Cómo buscar una palabra en caché?

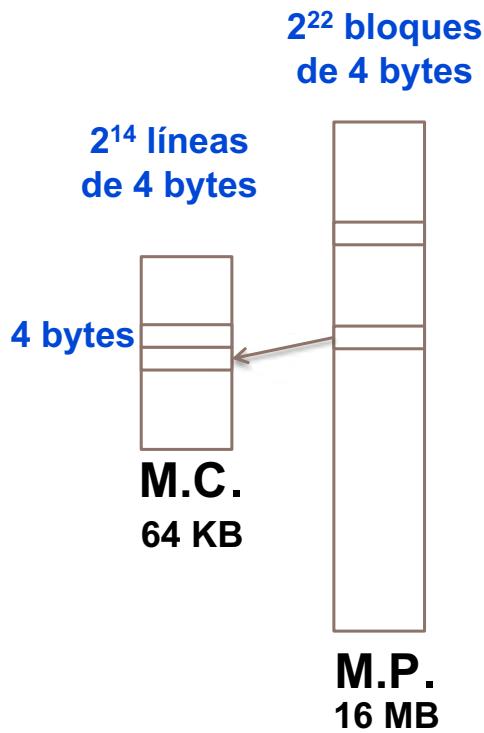


# Estructura y diseño de la cache



- ▶ Se divide la M.P. y la M.C. en bloques de igual tamaño
- ▶ A cada bloque de M.P. le corresponderá una Línea de M.C. (bloque en caché)

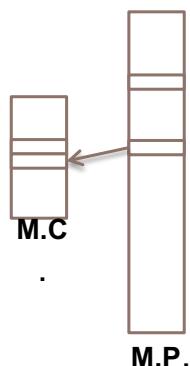
# Estructura y diseño de la cache



- ▶ Se divide la M.P. y la M.C. en bloques de igual tamaño
- ▶ A cada bloque de M.P. le corresponderá una línea de M.C. (bloque en caché)
- ▶ En el diseño se determina:
  - ▶ Tamaño
  - ▶ Función de correspondencia
  - ▶ Algoritmo de sustitución
  - ▶ Política de escritura
- ▶ Es habitual distintos diseños para L1, L2, ...

## Tamaño de caché

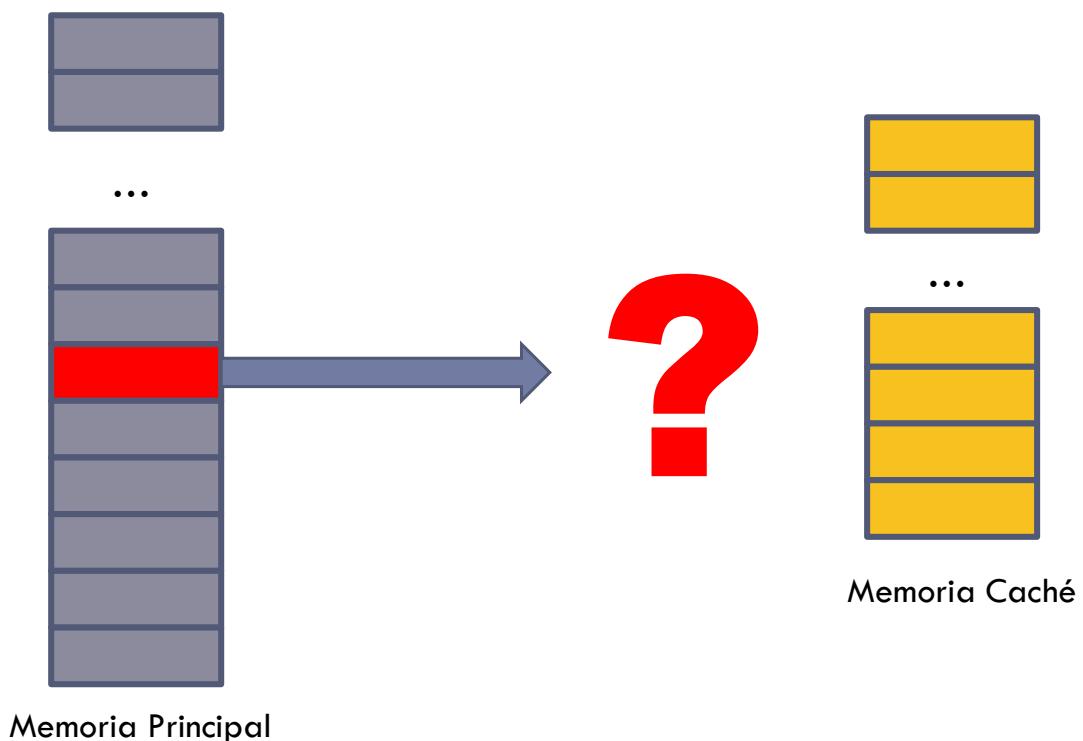
- ▶ El tamaño total y de las líneas en los que se organiza
- ▶ Se determina por estudios sobre códigos muy usados



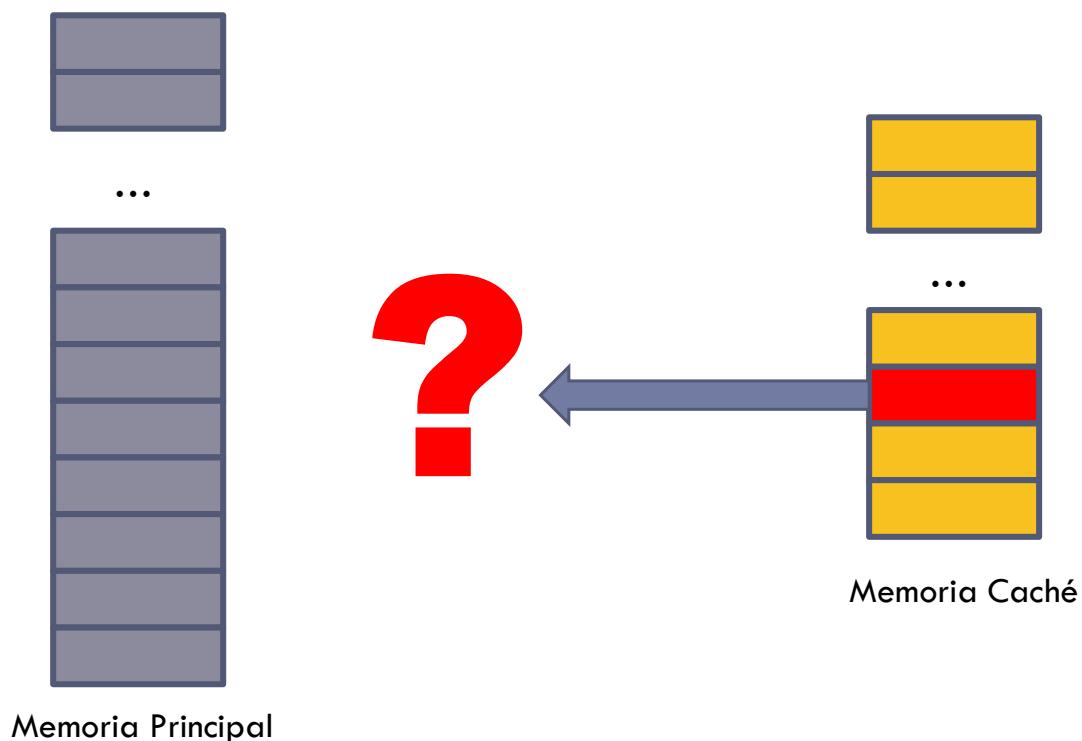
## Función de correspondencia

- ▶ Algoritmo que determina en qué lugares de la memoria caché se puede almacenar un bloque concreto de la memoria principal
- ▶ Un mecanismo que permita saber qué bloque concreto de memoria principal está en una lÍnea de la memoria caché
  - ▶ Se asocian a las lÍneas etiquetas

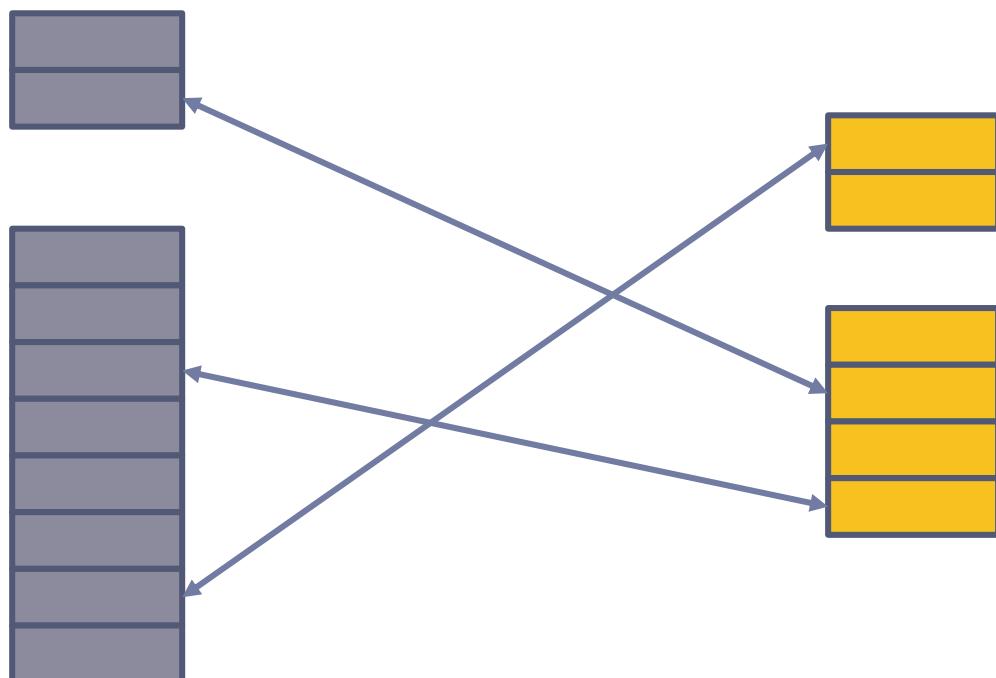
# Ubicación en caché



# Ubicación en memoria principal



## Función de correspondencia

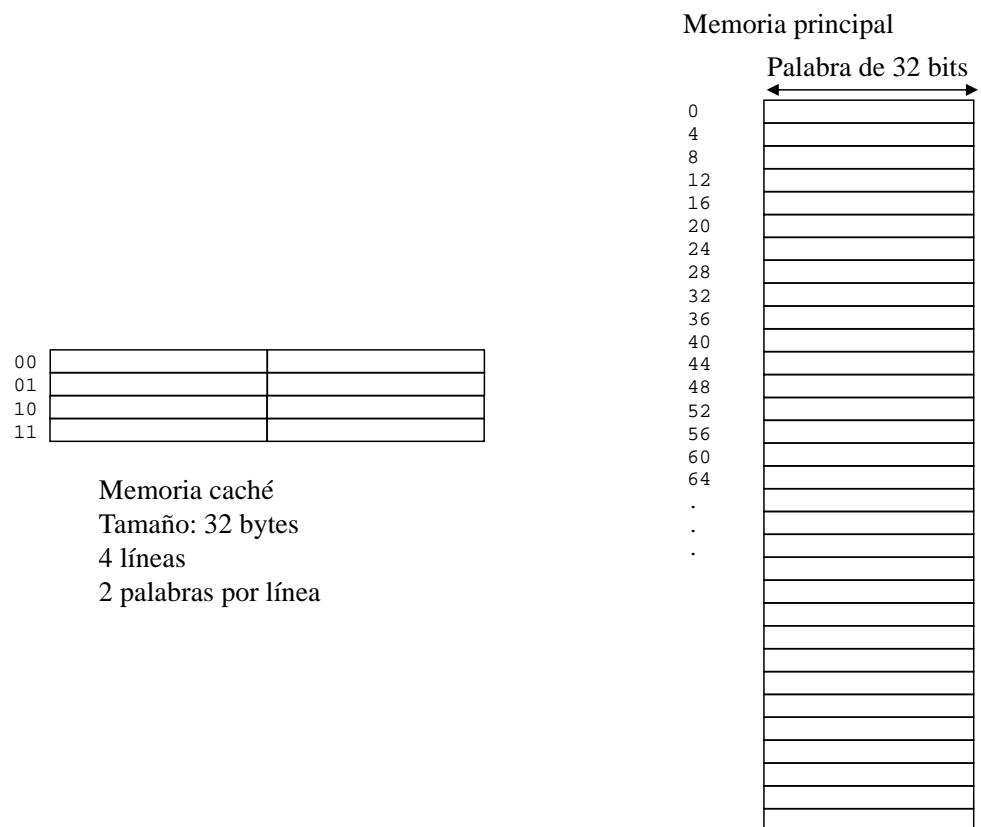


¿Dónde están en caché los datos de una posición de M.P.?

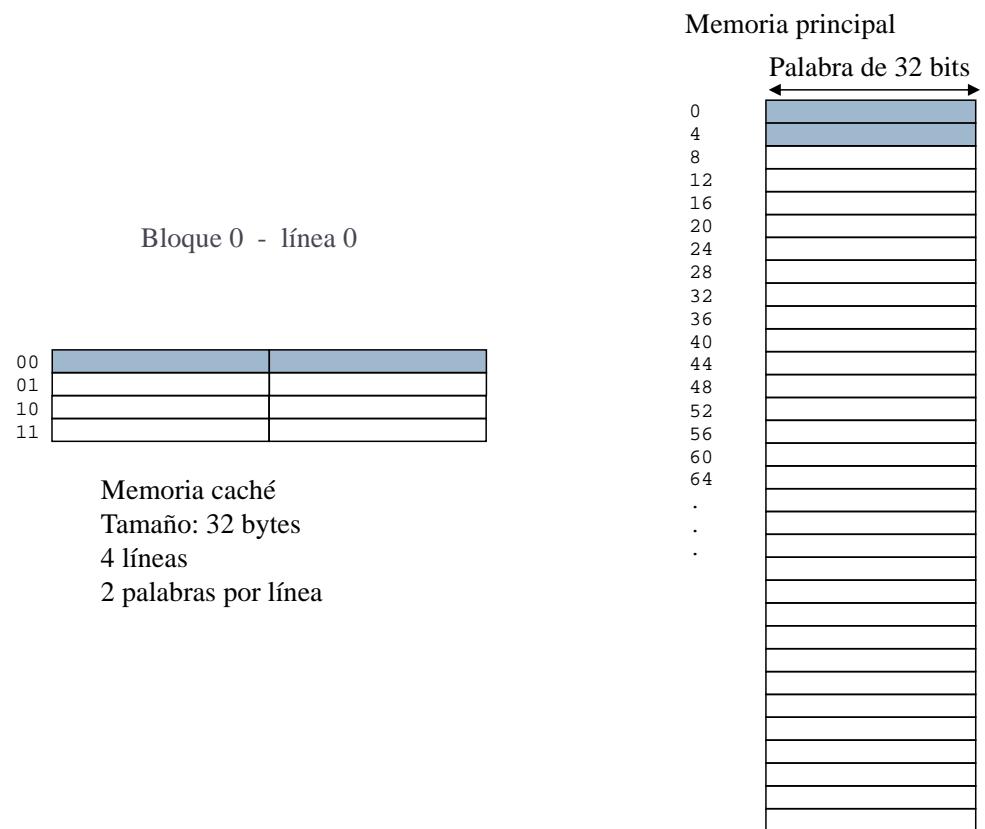
# Funciones de correspondencia

- ▶ Función de correspondencia directa
- ▶ Función de correspondencia asociativa
- ▶ Función de correspondencia asociativa por conjuntos

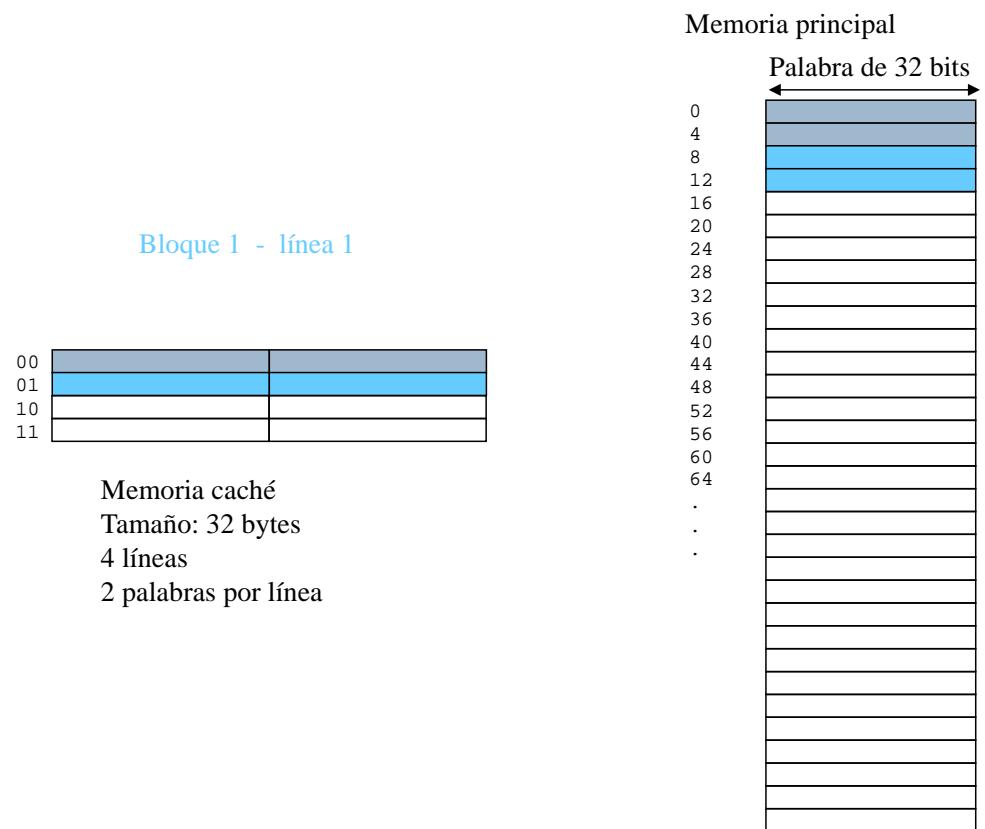
# Correspondencia directa



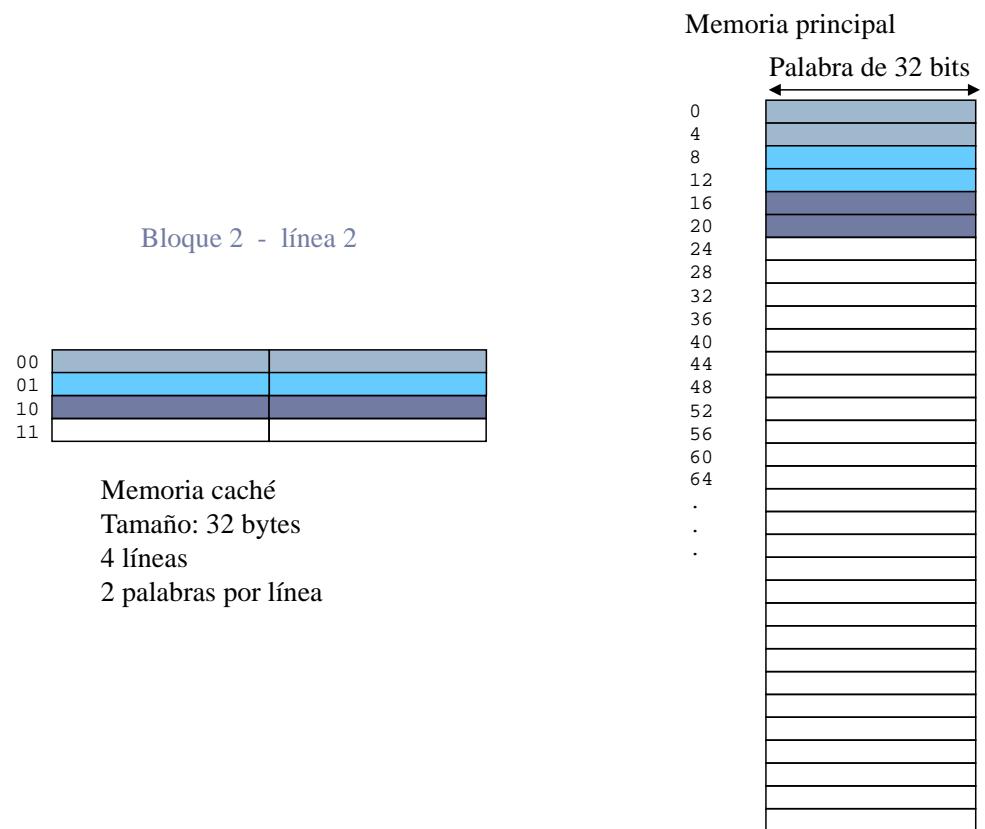
# Correspondencia directa



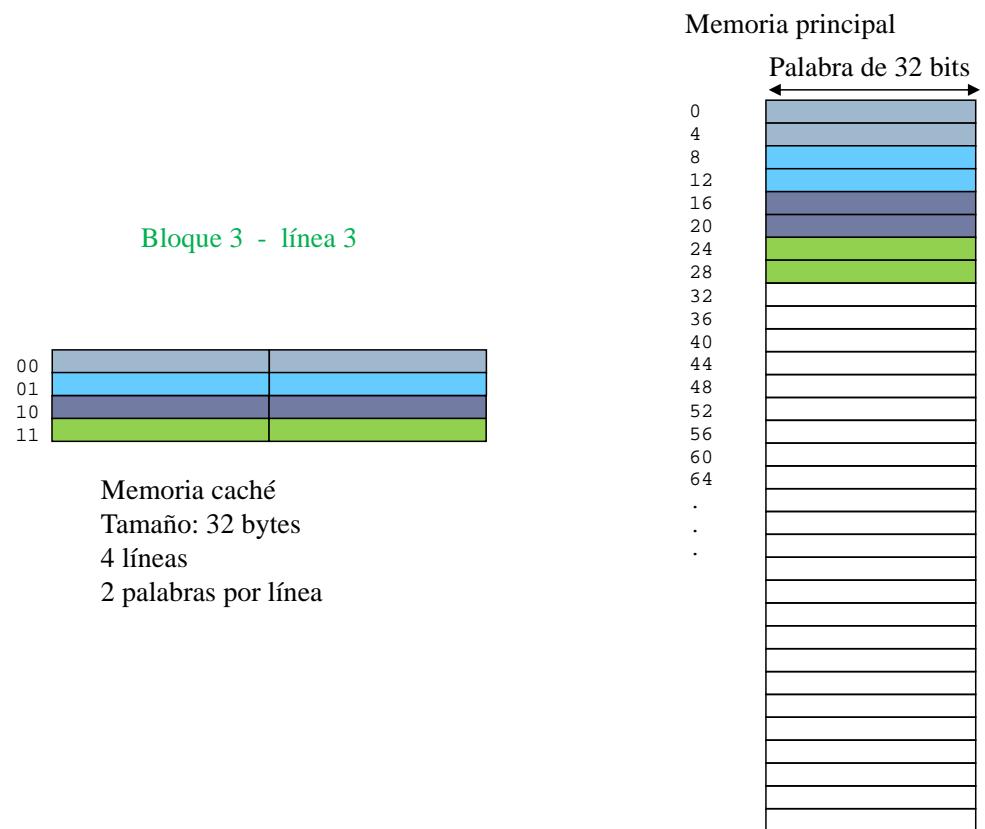
# Correspondencia directa



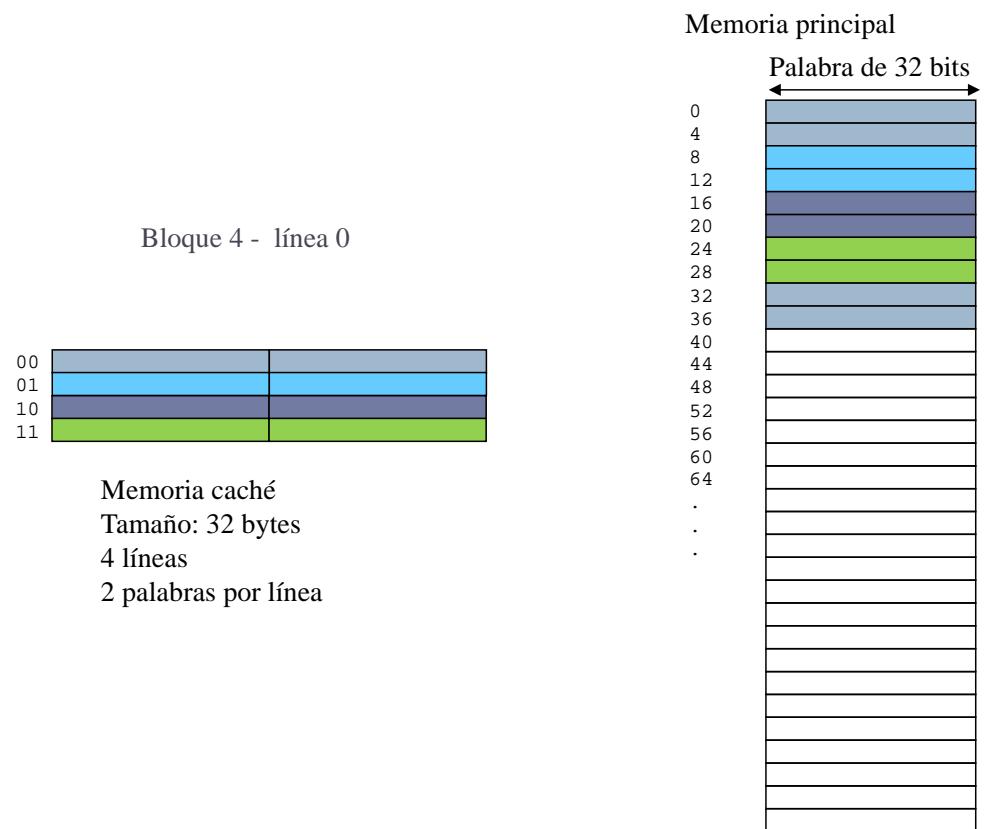
# Correspondencia directa



# Correspondencia directa



# Correspondencia directa



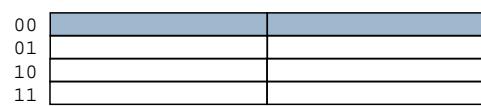
# Correspondencia directa

## ▶ En general:

- ▶ El bloque de memoria  $K$  se almacena en la línea:

$K \bmod \text{número de líneas}$

# Correspondencia directa



Memoria caché

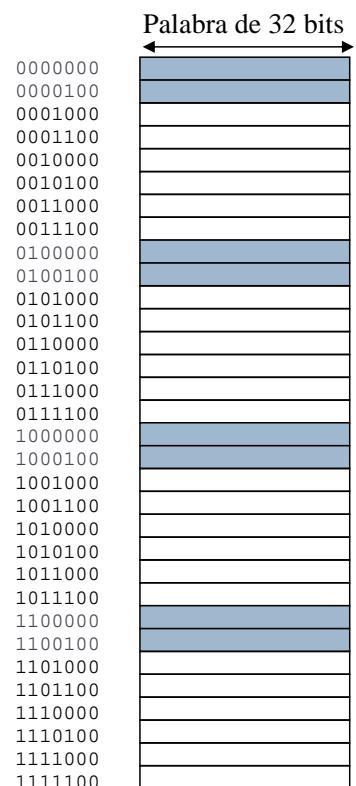
Tamaño: 32 bytes

4 líneas

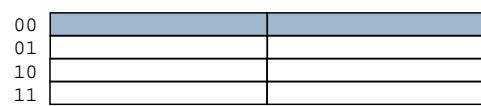
2 palabras por línea

Varios bloques en la misma línea

Memoria principal



# Correspondencia directa



Memoria caché

Tamaño: 32 bytes

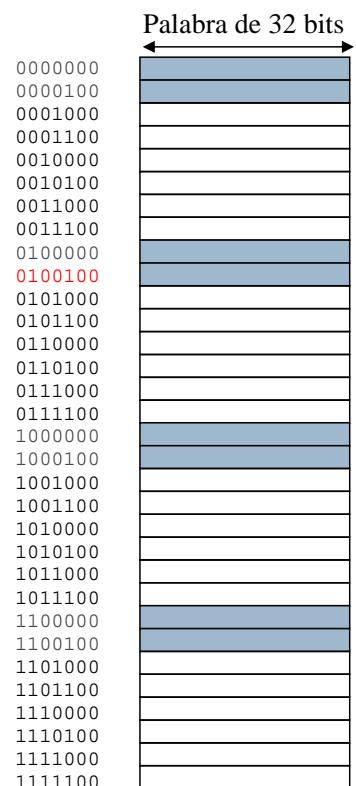
4 líneas

2 palabras por línea

¿Cómo se sabe qué bloque de memoria se encuentra una determinada línea?

Ejemplo: la dirección 0100100

Memoria principal



# Correspondencia directa

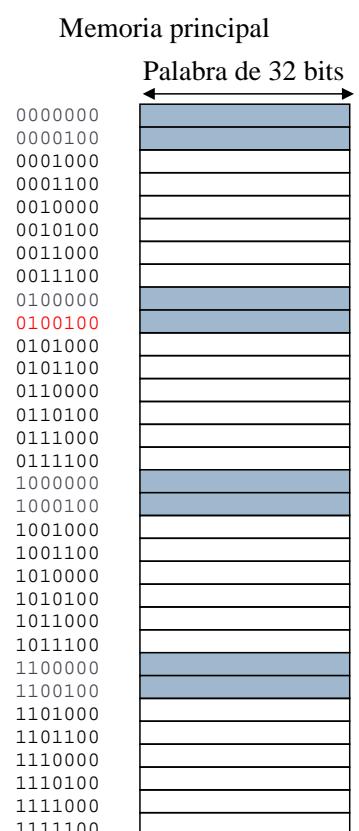
00	
01	
10	
11	

Memoria caché  
Tamaño: 32 bytes  
4 líneas  
2 palabras por línea

¿Cómo se sabe qué bloque de memoria se encuentra una determinada línea?

Ejemplo: la dirección 0100100

Se añade a cada línea una etiqueta



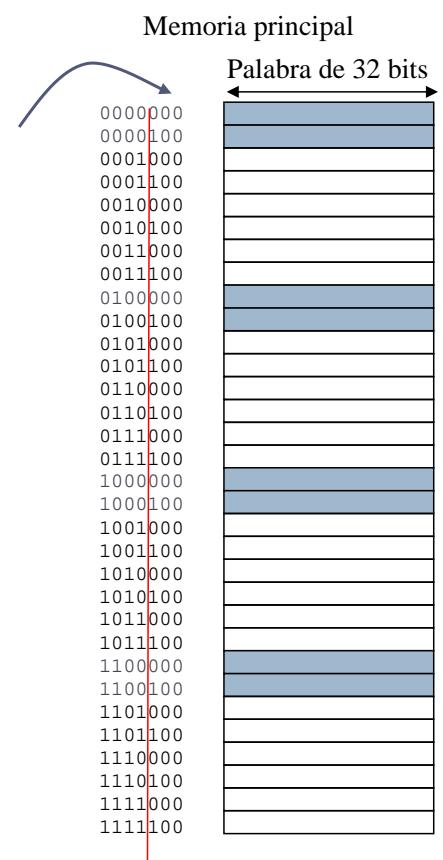
# Correspondencia directa

## Idea

¿qué byte dentro de la línea?  
Líneas de 8 bytes

00	
01	
10	
11	

Memoria caché  
Tamaño: 32 bytes  
4 líneas  
2 palabras por línea



# Correspondencia directa

## Idea

¿a qué línea?

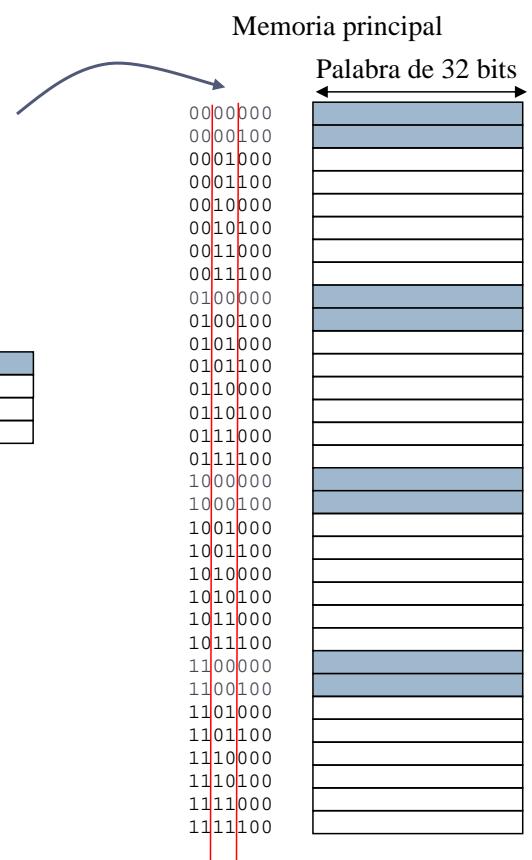
00	
01	
10	
11	

Memoria caché

Tamaño: 32 bytes

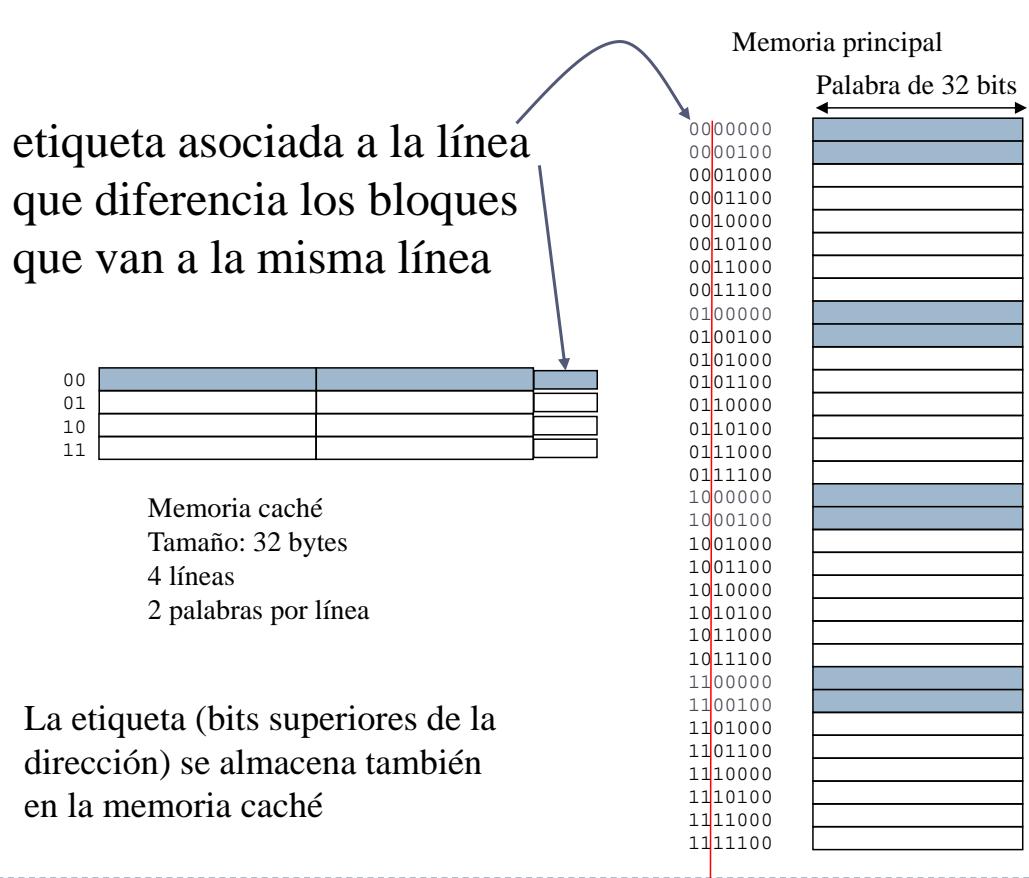
4 líneas

2 palabras por línea

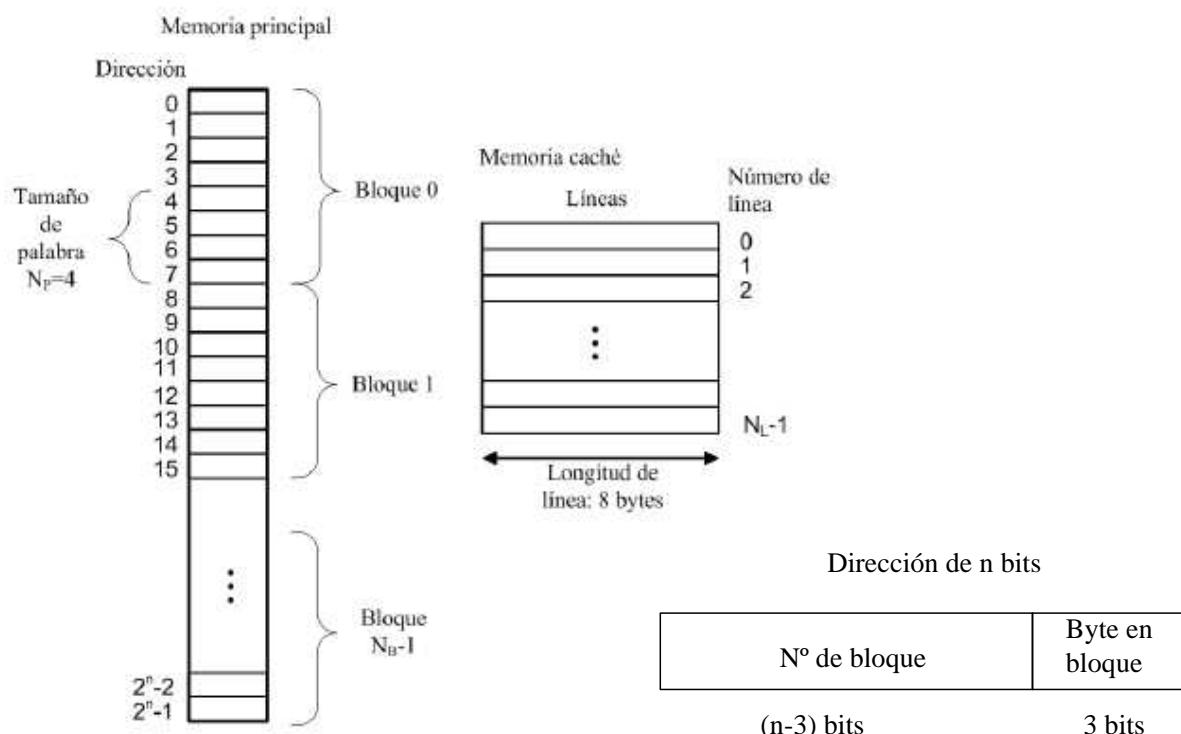


# Correspondencia directa

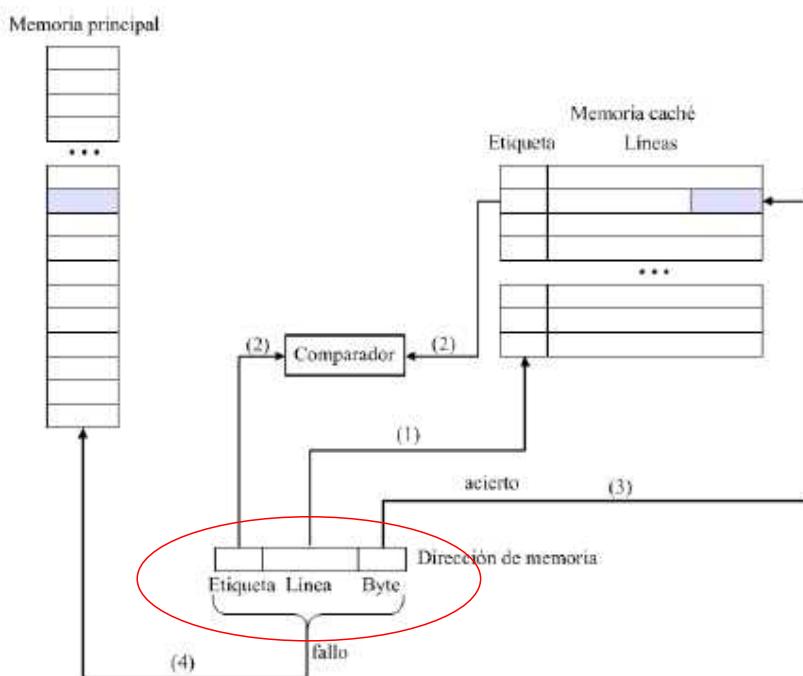
## Idea



## Ejemplo de organización de la memoria caché



# Organización de una memoria caché con correspondencia directa

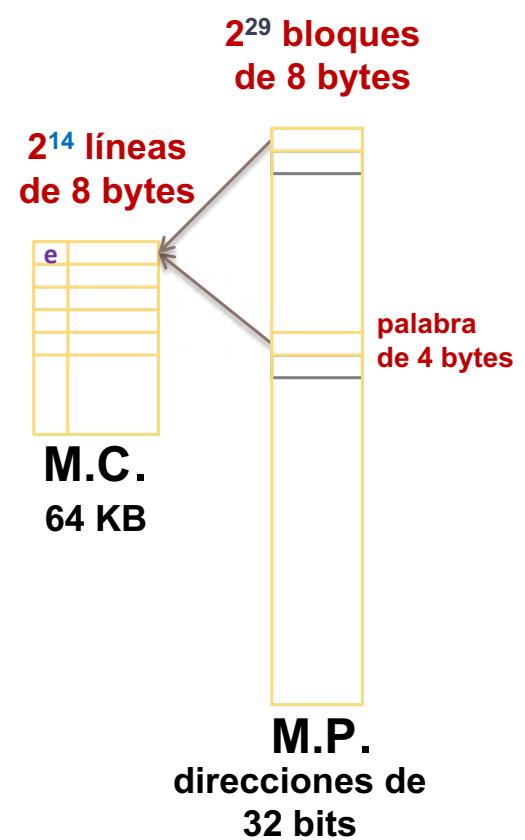


## Función de correspondencia directa Ejemplo

- ▶ Cada bloque de M.P. le corresponde una sola línea de caché (siempre la misma)
- ▶ La dirección de M.P. la determina:

**32-17    14    3**  
etiqueta    línea    byte

- ▶ Si en 'línea' está 'etiqueta', entonces está el bloque en caché
- ▶ Simple, poco costosa, pero puede provocar muchos fallos dependiendo del patrón de accesos

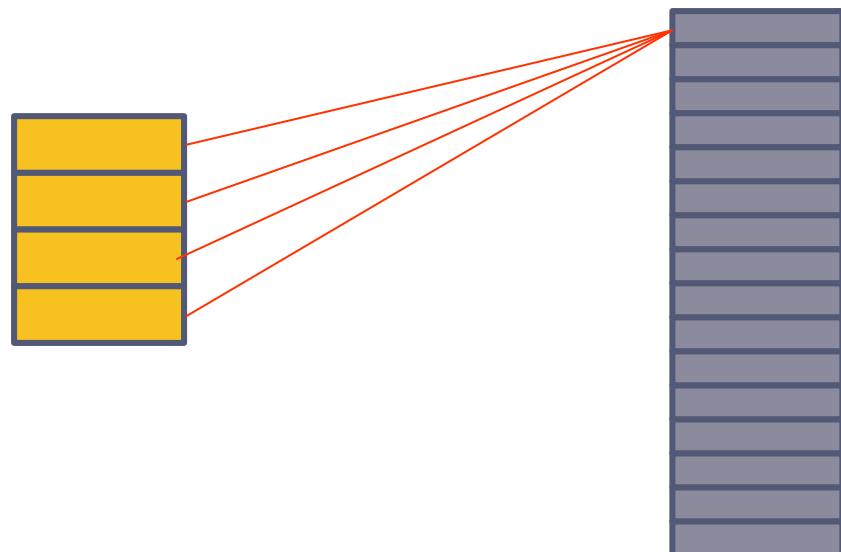


# Ejercicio

- ▶ Dado un computador de 32 bits con una memoria caché de 64 KB y bloques de 32 bytes. Si se utiliza correspondencia directa
  - ▶ ¿En qué lÍnea de la memoria caché se almacena la palabra de la dirección 0x0000408A?
  - ▶ ¿Cómo se puede obtener rápidamente?
  - ▶ ¿En qué lÍnea de la memoria caché se almacena la palabra de la dirección 0x1000408A?
  - ▶ ¿Cómo sabe la caché si la palabra almacenada en esa lÍnea corresponde a la palabra de la dirección 0x0000408A o a la palabra de la dirección 0x1000408A?

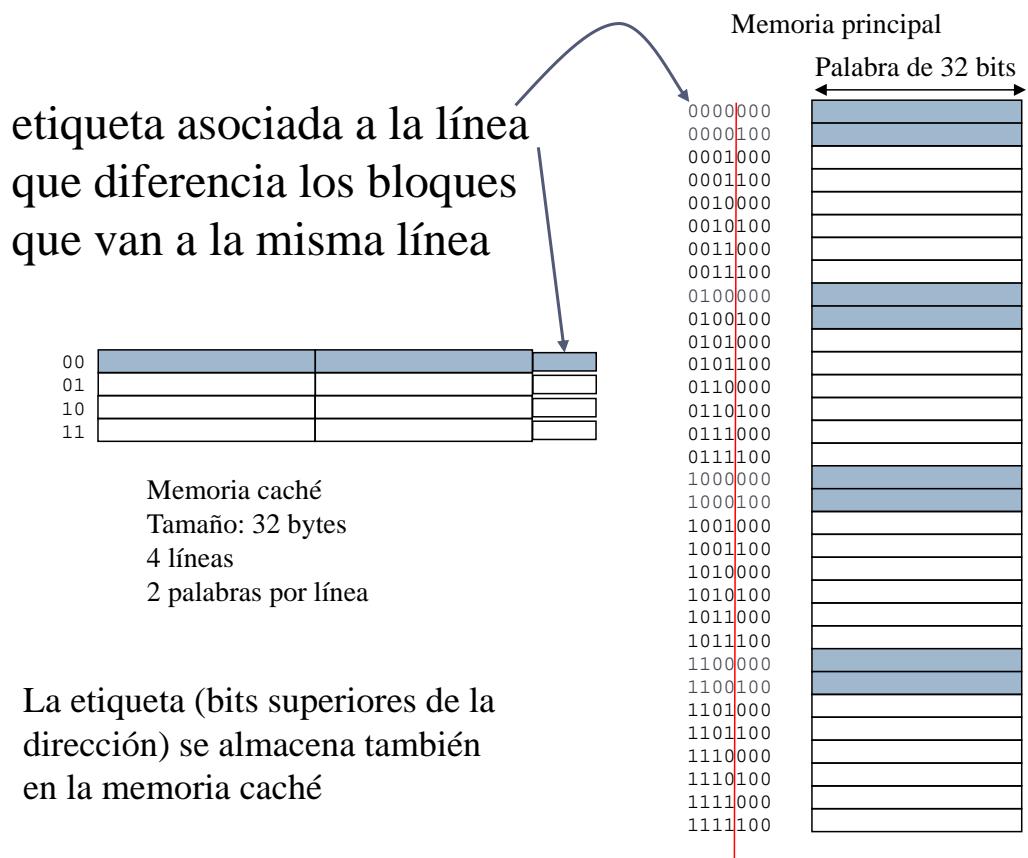
## Correspondencia asociativa

- ▶ Cada bloque de MP puede almacenarse en cualquier línea de la caché

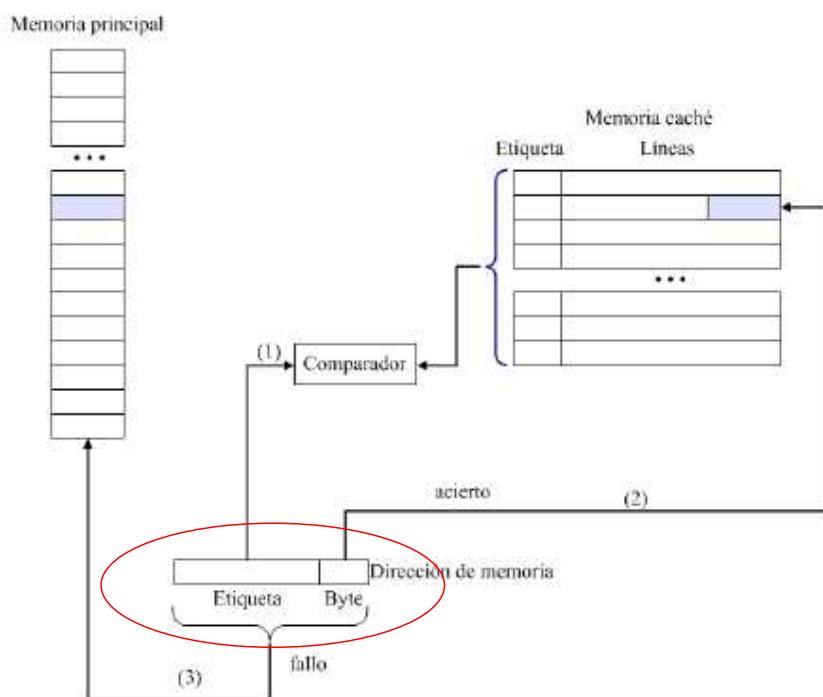


# Correspondencia asociativa

## Idea



# Organización de una memoria caché con correspondencia asociativa



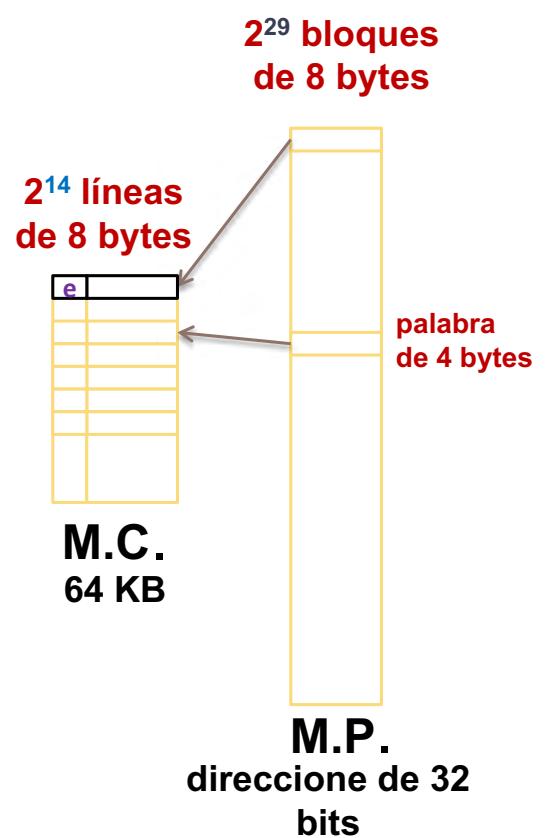
## Función de correspondencia asociativa

### Ejemplo

- ▶ Un bloque de M.P. puede cargarse en cualquier línea de caché
- ▶ La dir. de M.P. se interpreta como:

32-3	3
etiqueta	byte

- ▶ Si hay un línea con ‘etiqueta’ en la caché, está allí el bloque
- ▶ Independiente del patrón de acceso, búsqueda costosa
- ▶ Etiquetas más grandes: cachés más grandes



## Correspondencia asociativa por conjuntos

- ▶ La memoria se organiza en conjuntos de  $I$  líneas
- ▶ Una memoria caché asociativa por conjunto de  $K$  vías:
  - ▶ Cada conjunto almacena  $K$  líneas
- ▶ Cada bloque siempre se almacena en el mismo conjunto
  - ▶ El bloque  $B$  se almacena en el conjunto:
    - ▶  $B \bmod$  número de conjuntos
- ▶ Dentro de un conjunto el bloque se puede almacenar en cualquiera de las  $I$  líneas de ese conjunto

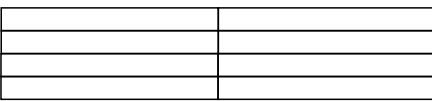
## Correspondencia asociativa por conjuntos

Número de conjunto

Conjunto 0  
Conjunto 1

Número de línea

Memoria caché  
Tamaño: 32 bytes  
Asociativa por conjunto de 2 vías  
2 líneas por conjunto  
2 palabras por línea

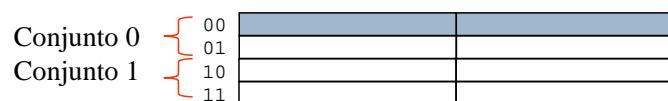


Memoria principal

Palabra de 32 bits

0000000
0000100
0001000
0001100
0010000
0010100
0011000
0011100
0100000
0100100
0101000
0101100
0110000
0110100
0111000
0111100
1000000
1000100
1001000
1001100
1010000
1010100
1011000
1011100
1100000
1100100
1101000
1101100
1110000
1110100
1111000
1111100

# Correspondencia asociativa por conjuntos



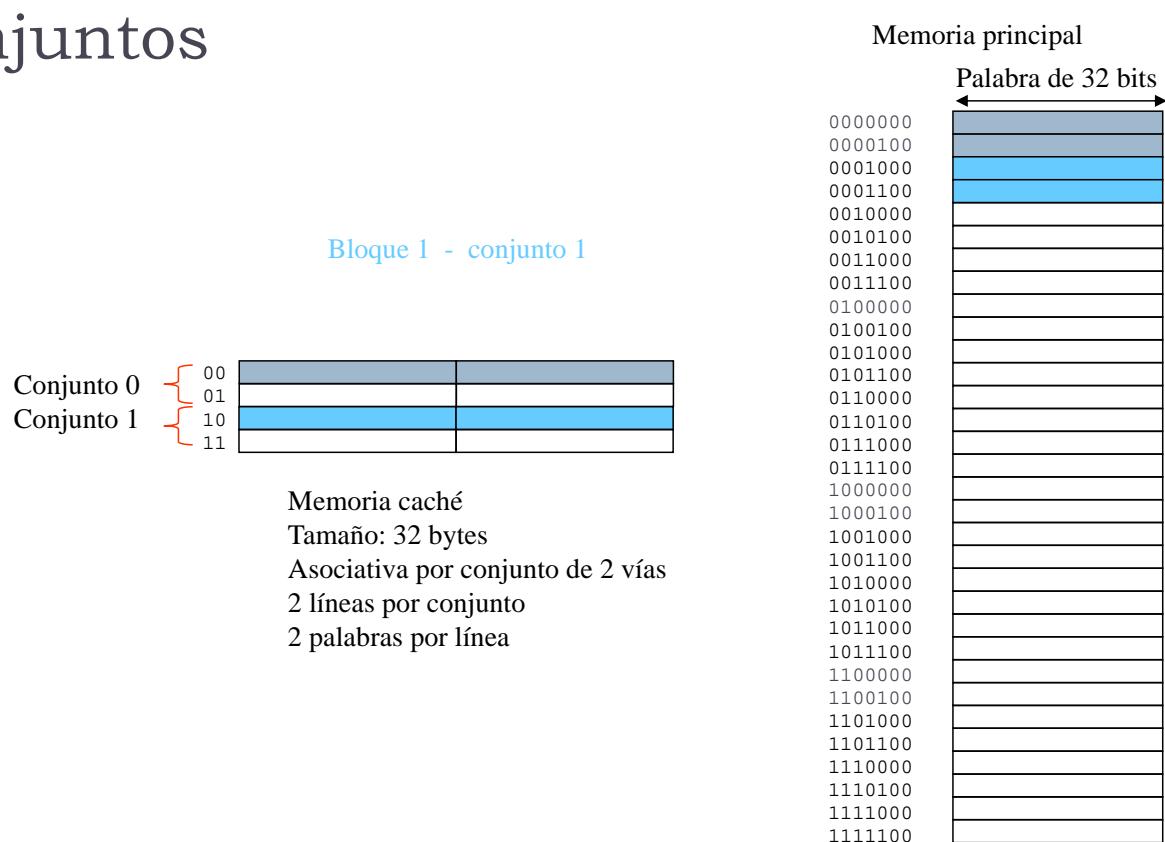
Bloque 0 - Conjunto 0

Memoria caché  
Tamaño: 32 bytes  
Asociativa por conjunto de 2 vías  
2 líneas por conjunto  
2 palabras por línea

Memoria principal

Palabra de 32 bits
0000000
0000100
0001000
0001100
0010000
0010100
0011000
0011100
0100000
0100100
0101000
0101100
0110000
0110100
0111000
0111100
1000000
1000100
1001000
1001100
1010000
1010100
1011000
1011100
1100000
1100100
1101000
1101100
1110000
1110100
1111000
1111100

# Correspondencia asociativa por conjuntos



# Correspondencia asociativa por conjuntos



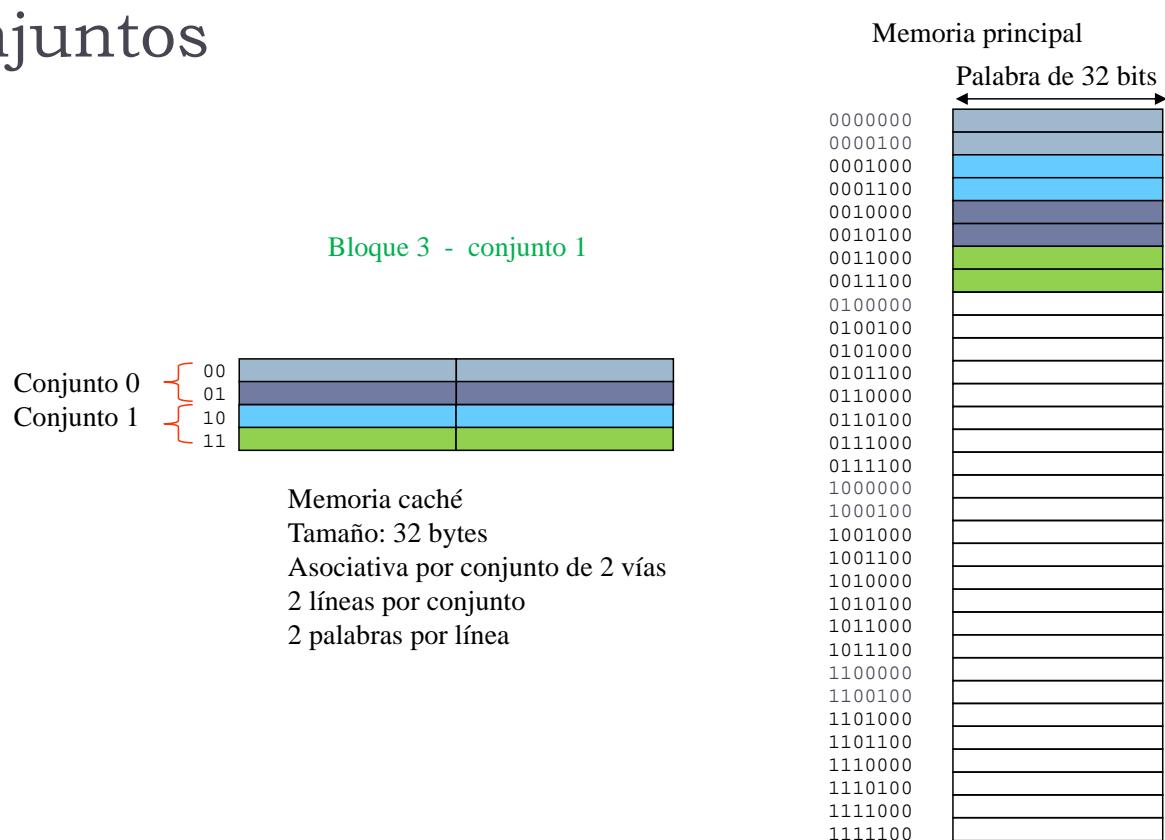
Bloque 2 - conjunto 0

Memoria caché  
Tamaño: 32 bytes  
Asociativa por conjunto de 2 vías  
2 líneas por conjunto  
2 palabras por línea

Memoria principal

Palabra de 32 bits
0000000
0000100
0001000
0001100
0010000
0010100
0011000
0011100
0100000
0100100
0101000
0101100
0110000
0110100
0111000
0111100
1000000
1000100
1001000
1001100
1010000
1010100
1011000
1011100
1100000
1100100
1101000
1101100
1110000
1110100
1111000
1111100

# Correspondencia asociativa por conjuntos

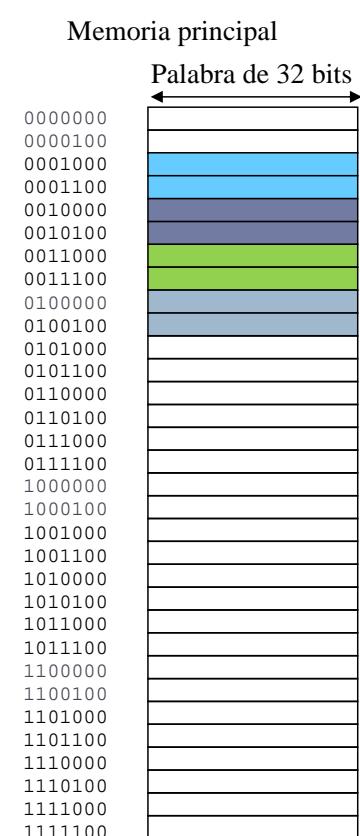


# Correspondencia asociativa por conjuntos



Bloque 4 - conjunto 0

Memoria caché  
Tamaño: 32 bytes  
Asociativa por conjunto de 2 vías  
2 líneas por conjunto  
2 palabras por línea



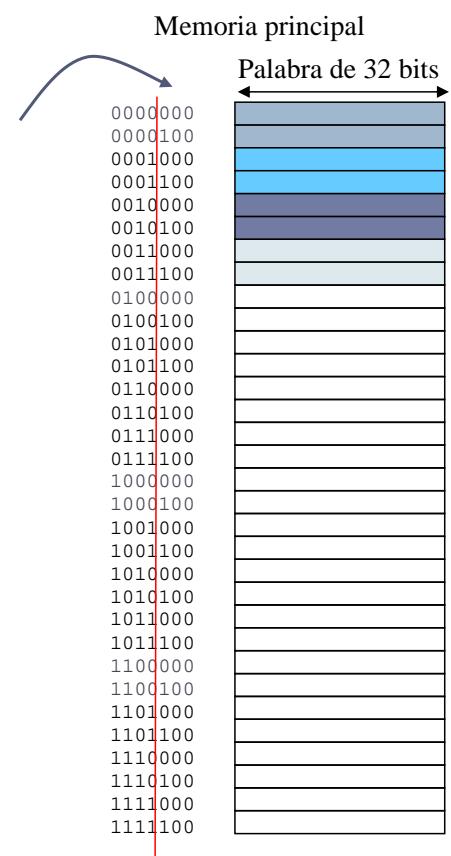
Habría que eliminar la línea que estaba antes

# Correspondencia asociativa por conjuntos

¿qué byte dentro de la línea?  
Líneas de 8 bytes



Memoria caché  
Tamaño: 32 bytes  
Asociativa por conjunto de 2 vías  
2 líneas por conjunto  
2 palabras por línea

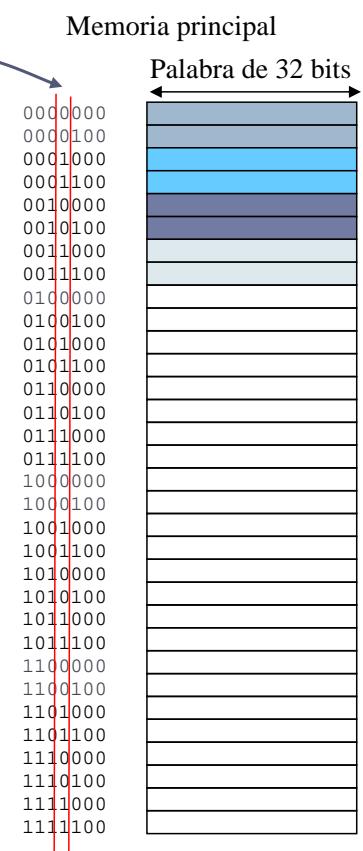


# Correspondencia asociativa por conjuntos

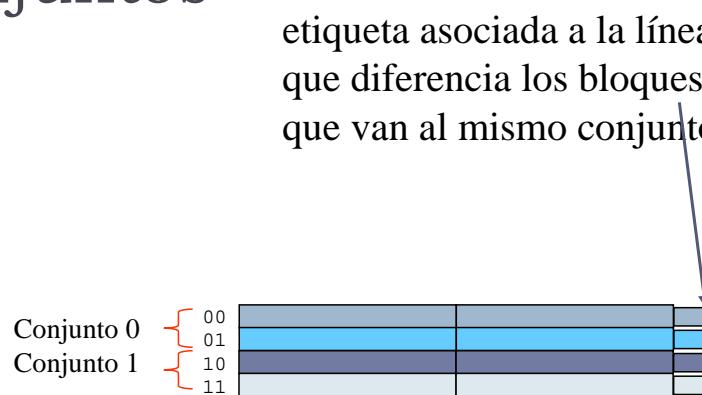
¿qué conjunto?  
dentro del conjunto  
a cualquier línea



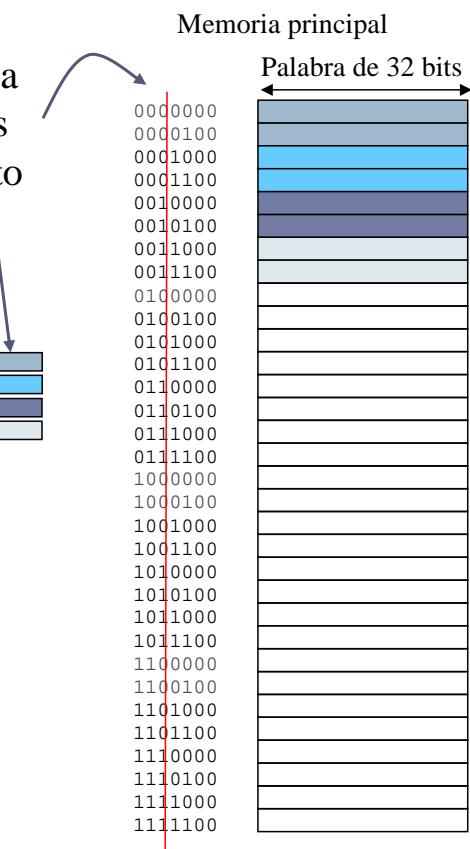
Memoria caché  
Tamaño: 32 bytes  
Asociativa por conjunto de 2 vías  
2 líneas por conjunto  
2 palabras por línea



# Correspondencia asociativa por conjuntos



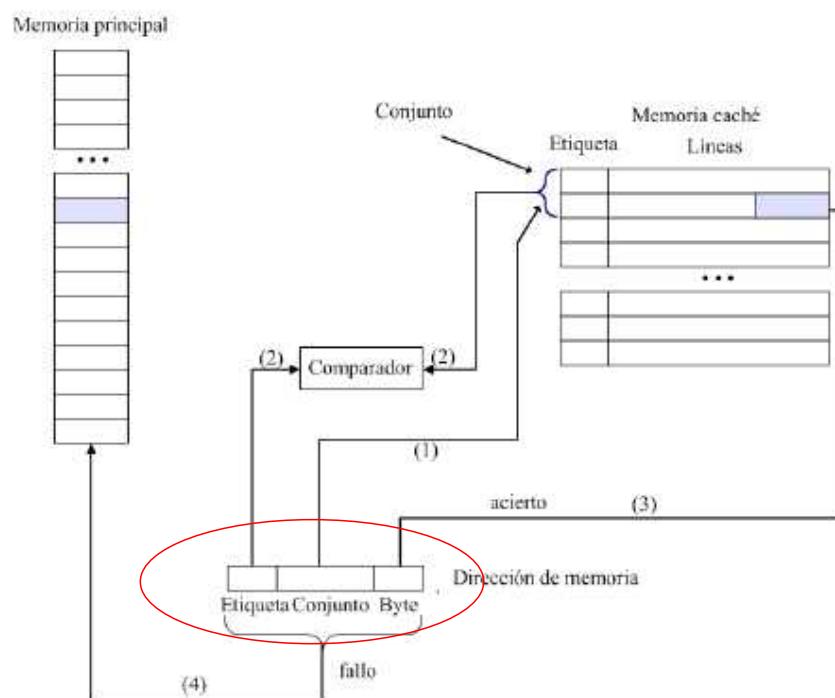
Memoria caché  
Tamaño: 32 bytes  
Asociativa por conjunto de 2 vías  
2 líneas por conjunto  
2 palabras por línea



## Correspondencia asociativa por conjuntos

- ▶ **Establece un compromiso entre flexibilidad y coste.**
  - ▶ Es más flexible que la correspondencia directa.
  - ▶ Es menos costosa que la correspondencia asociativa.

# Organización de una memoria caché asociativa por conjuntos

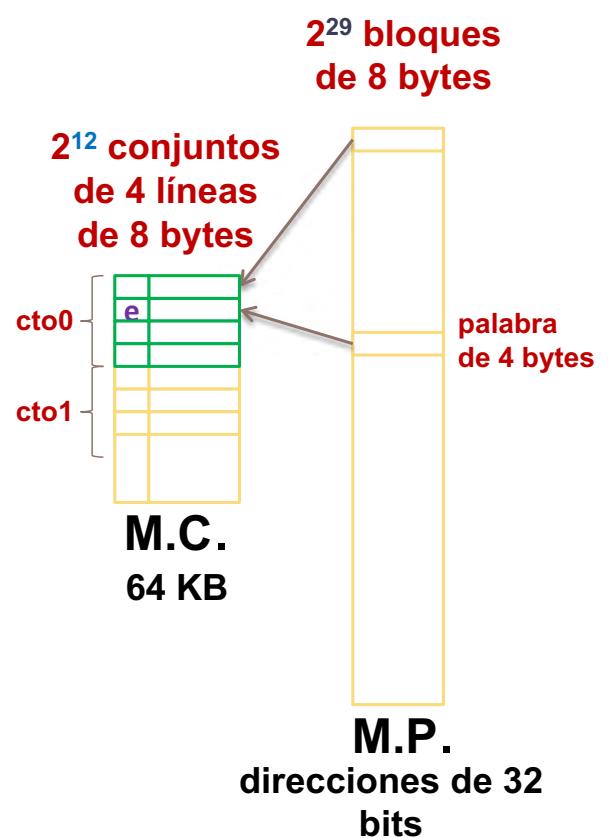


## Función de correspondencia asociativa por conjuntos. Ejemplo

### ▶ Asociativa por conjuntos:

- ▶ Un bloque de M.P. puede cargarse en cualquier línea de caché (vía) de un conjunto determinado
- ▶ La dir. de M.P. se interpreta como:

17	12	3
etiqueta	conjunto	palabra
- ▶ Si hay un línea con 'etiqueta' en el conjunto 'conjunto', está allí el bloque en caché
- ▶ [V] lo mejor de directa y asociativa  
[!] búsqueda menos costosa



## Sustitución de bloques

- ▶ Cuando todas las entradas de la caché contienen bloques de memoria principal:
  - ▶ Hace falta seleccionar una línea que hay que dejar libre para traer un bloque de la MP.
    - ▶ Directa: no hay posible elección
    - ▶ Asociativa: seleccionar una línea de la caché.
    - ▶ Asociativa por conjuntos: seleccionar una línea del conjunto seleccionado.
  - ▶ Existen diversos algoritmos para seleccionar la línea de la caché que hay que liberar.

# Algoritmos de sustitución

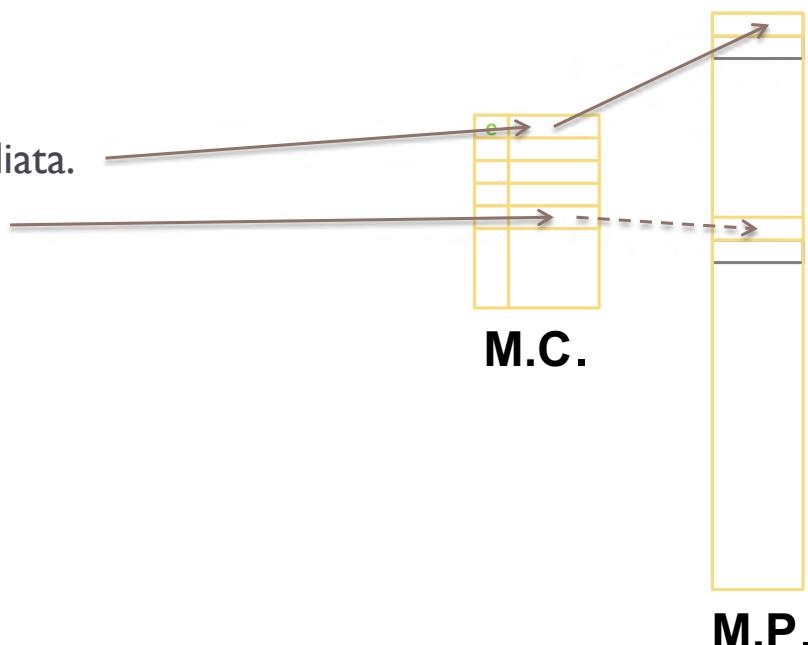
- ▶ **FIFO**
  - ▶ *First-in-first-out*
  - ▶ Sustituye la lnea que lleva más tiempo en la caché.
- ▶ **LRU:**
  - ▶ *Least Recently Used*
  - ▶ Sustituye la lnea que lleva más tiempo sin usarse.
- ▶ **LFU:**
  - ▶ *Least Frequently Used*
  - ▶ Sustituye la lnea que se ha usado menos veces.

## Políticas de escritura

- ▶ Cuando se modifica un dato en memoria caché, hay que actualizar en algún momento la memoria principal

- ▶ Alternativas:

- ▶ Escritura inmediata.
- ▶ Post-escritura.



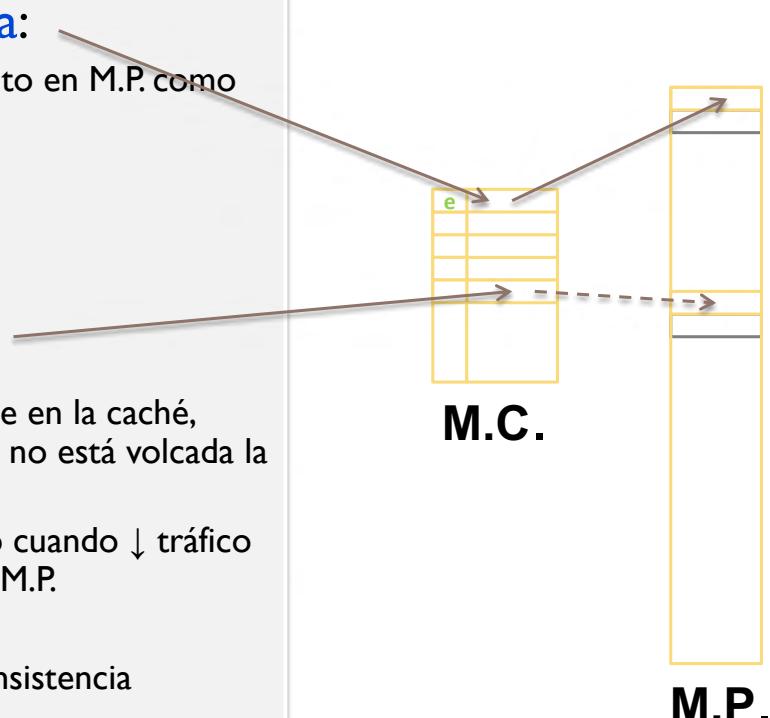
# Política de escritura

## ▶ Escritura inmediata:

- ▶ La escritura se hace tanto en M.P. como en cache
- ▶ [V] Coherencia
- ▶ [I] Mucho tráfico
- ▶ [I] Escrituras lentas

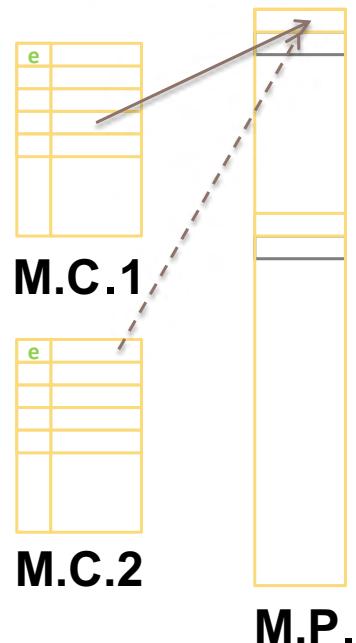
## ▶ Escritura diferida:

- ▶ La escritura solo se hace en la caché, indicando en un bit que no está volcada la línea en M.P.
- ▶ Al sustituir el bloque (o cuando ↓ tráfico con M.P.) se escribe en M.P.
- ▶ [V] Velocidad
- ▶ [I] Coherencia + inconsistencia

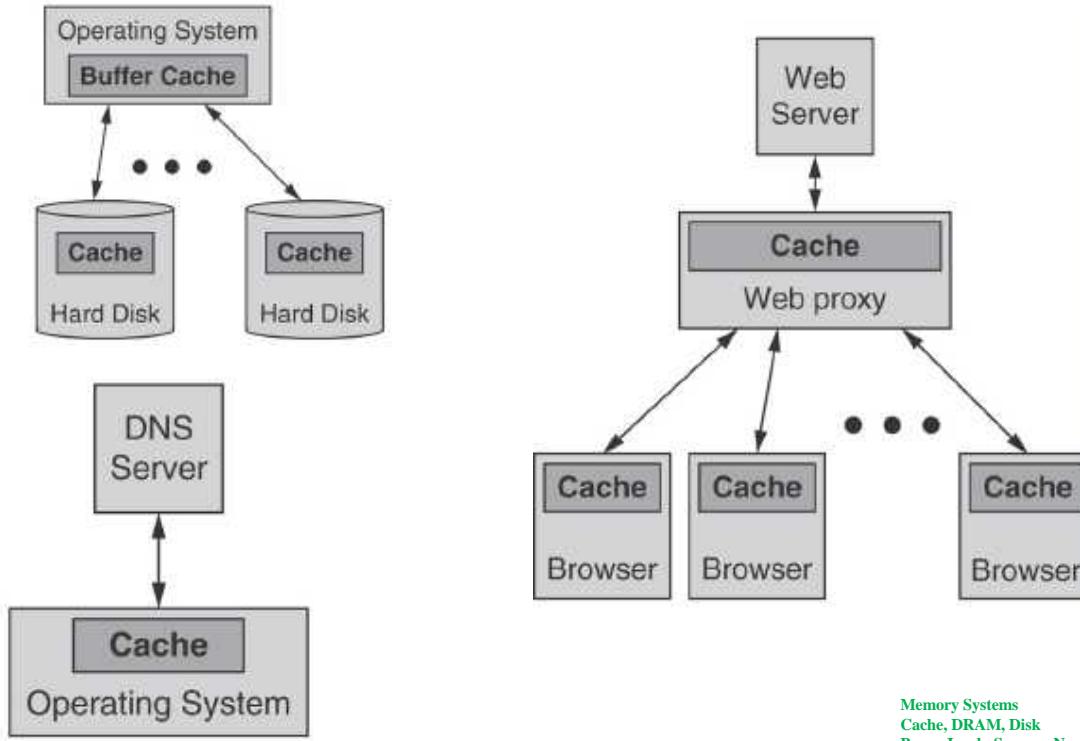


# Política de escritura

- ▶ Ej: CPU multicore con caché por core
  - ▶ Las escrituras en caché solo son vistas por un core
  - ▶ Si cada core escribe sobre una misma palabra, ¿cuál es el resultado final?
- ▶ Ej: módulo de E/S con acceso directo a M.P.
  - ▶ Actualización por DMA puede no ser coherente
- ▶ Porcentaje de referencias a memoria para escritura del orden del 15%.



# Ejemplos de cachés en otros sistemas



Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David Wang  
Elsevier

# Ejercicio

- ▶ Considere un computador de 32 bits con las siguientes características:
- Memoria física instalada de 256 MB con un tiempo de acceso de 70 ns.
  - Direccionamiento de la memoria por bytes.
  - Tamaño de la memoria caché de 64 KB.
  - Tamaño de la línea 64 bytes.
  - La caché es asociativa por conjuntos de 4 vías.
  - El tiempo de acceso a la caché es de 5 ns y el tiempo de penalización en caso de fallo es de 100 ns.

# Ejercicio

## ► Se pide:

- a) ¿Cuántos bloques tiene la memoria principal?
- b) ¿Cuántos conjuntos tiene la memoria caché?
- c) Dada una dirección de memoria, indique qué partes de la dirección se utilizan para identificar la etiqueta, el conjunto y el byte dentro de la linea.  
Indique también el número de bits de cada parte.
- d) Dada la siguiente dirección de memoria  
0000 0011 1100 0011 0000 0000 1111 1000.  
En caso de encontrarse en la memoria caché  
¿en qué conjunto se almacenará?
- e) Si el tiempo medio de acceso al sistema de memoria es de 8 ns  
¿cuál es tasa de acierto necesaria para lograr este tiempo?

## Ejercicio (solución)

- a) La memoria tiene un tamaño de línea de 64 bytes =  $2^6$  bytes.  
Por tanto, el número de bloques de memoria principal será

$$\begin{aligned} \text{nbloques} &= \text{tamaño memoria} / \text{tamaño de línea} = \\ &256 \text{ MB} / 64 = \\ &256 \times 2^{20} / 64 = 256 * 2^{14} \text{ bloques} \end{aligned}$$

- b) El número de líneas de memoria caché es

$$\begin{aligned} \text{nlineas} &= \text{tamaño memoria} / \text{tamaño de línea} = \\ &64 \text{ KB} / 64 \text{ bytes} = \\ &2^{16} / 2^6 = 2^{10} = 1024 \text{ líneas} \end{aligned}$$

$$\text{Conjuntos} = \text{Nº líneas} / \text{Nº vías} = 1024 / 4 = 256 \text{ conjuntos.}$$

## Ejercicio (solución)

- c) La dirección de una caché asociativa por conjuntos se divide en tres partes: etiqueta, conjunto y byte dentro de la línea.
- ▶ Byte: el tamaño de la línea es 64 bytes =  $2^6$ bytes.  
Se necesitan, por tanto 6 bits para identificar el byte dentro de la línea.
  - ▶ Conjunto: Hay 256 conjuntos =  $2^8$ ,  
por lo que se necesitan 8 bits para identificar un conjunto
  - ▶ Etiqueta: para la etiqueta se emplean el resto de los bits de la dirección =  $32 - 6 - 8 = 18$

La dirección quedará:

Etiqueta (18 bits)	Conjunto (8 bits)	Byte (6 bits)
--------------------	-------------------	---------------

## Ejercicio (solución)

- d) Utilizamos el formato de la dirección del apartado anterior:

Etiqueta (18 bits)	Conjunto (8 bits)	Byte (6 bits)
--------------------	-------------------	---------------

El byte asociado a esta dirección se encontrará en el **conjunto 3**

- e) El cálculo del tiempo medio de acceso a memoria se hace con la siguiente fórmula:

- ▶  $T_{\text{medio}} = tc + (l-h) * tfallo$
- ▶  $8 = 5 + (l-h) * 100$

Despejando h, se tiene  $h = 97/100 = 0,97$  (tanto por uno)

Es decir, **una tasa de acierto del 97 %**

## Ejercicio

- ▶ Sea un computador con una memoria caché y principal con las siguientes características:
  - ▶ Tiempo de acceso a memoria caché de 4 ns
  - ▶ Tiempo de acceso a memoria principal de 80 ns
  - ▶ Tiempo para servir un fallo de caché de 120 ns
  - ▶ Política de escritura inmediata

En este computador se ha observado que la tasa de aciertos a la memoria caché es del 95 % y que cada 100 accesos, 90 son de lectura. Calcular el tiempo medio de acceso a memoria.

## Ejercicio

- ▶ **Sea un computador dotado de una memoria cache con las siguientes características:**
  - ▶ Tamaño: 16 KB con bloques de 32 bytes (8 palabras)
  - ▶ Tiempo de acceso: 10ns
  - ▶ Esta memoria está conectada a través de un bus de 32 bits a una memoria principal que es capaz de transferir un bloque de 8 palabras en 120 ns
  - ▶ Política de escritura: post-escritura o escritura diferida.
  - ▶ Se pide:
- ▶ **Calcular la tasa de aciertos que es necesaria para que el tiempo medio de acceso al sistema de memoria sea de 20 ns.**

## Ejercicio

- ▶ Se dispone de un computador con una memoria caché con un tamaño de 64 KB. El tamaño de la línea es de 64 bytes. La caché tiene un tiempo de acceso de 20 ns y un tiempo de penalización por fallo de 120 ns. La caché es asociativa por conjuntos de dos vías. Se pide:
  - ▶ Indique el número total de líneas de caché
  - ▶ Indique el número de conjuntos que tiene la caché.
  - ▶ Indique el número de líneas por conjunto
  - ▶ Haga un dibujo con la estructura de la caché
  - ▶ Diga cuánto tiempo tardaríamos en obtener un dato si se produce un fallo en la caché.

# Ejercicio

- ▶ Sea un computador de 32 bits con el juego de instrucciones del MIPS, que ejecuta el siguiente fragmento de código cargado a partir de la dirección 0x00000000

```
    li      $t0, 1000
    li      $t1, 0
    li      $t2, 0
bucle: addi   $t1, $t1, 1
        addi   $t2, $t2, 4
        beq    $t1, $t0, bucle
```

- ▶ Este computador dispone de una memoria caché asociativa por conjunto de 4 vías, de 32 KB y 16 líneas de 16 bytes. Calcule de forma razonada el número de fallos de caché y la tasa de aciertos que produce el fragmento de código anterior, asumiendo que se ejecuta sin ninguna interrupción y que la memoria caché está inicialmente vacía

# Ejercicio

Se dispone de un computador con direcciones de memoria de 32 bits, que direcciona la memoria por bytes. El computador dispone de una memoria caché asociativa por conjuntos de 4 ~~vías~~, con un tamaño de ~~lnea~~ de 64 bytes. Dicha caché tiene un tamaño de 128 KB. El tiempo de acceso a la memoria caché es de 2 ns y el tiempo necesario para tratar un fallo de caché es de 80 ns. Considere el siguiente fragmento de programa.

```
float v1[10000];
float v2[10000];

for (i = 0; i < 10000; i = i + 1)
    v1[i] = v1[i] + v2[i];
```

Indique de forma razonada:

- a) El tamaño en MB de la memoria que se puede direccionar en este computador.
- b) El número de palabras que se pueden almacenar en la memoria caché de este computador.
- c) El número de ~~lneas~~ de la caché y número de conjuntos de la caché.
- d) Indique la tasa de aciertos necesaria para que el tiempo medio de acceso al sistema de memoria de este computador sea de 10 ns.
- e) Indique de forma razonada la tasa de aciertos a la caché para el fragmento de código anterior teniendo en cuenta solo los accesos a datos (considere que la variable *i* se almacena en un registro y que la caché está inicialmente vacía).

## **Parte VII**

### **Tema 6. Sistemas de Entrada Salida**



Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 6 E/S y dispositivos periféricos

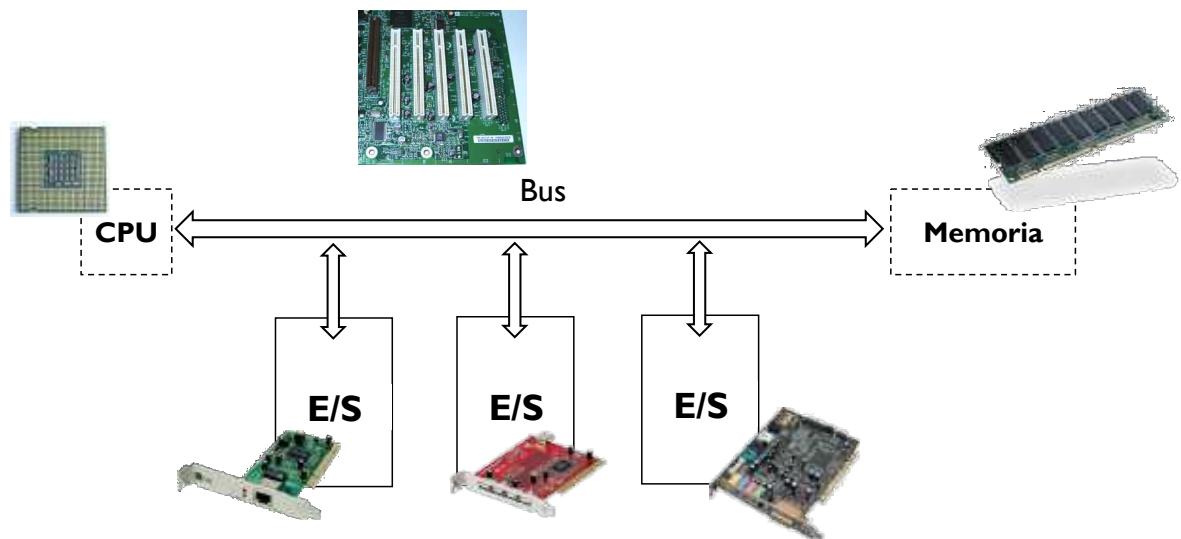
Estructura de Computadores  
Grado en Ingeniería Informática



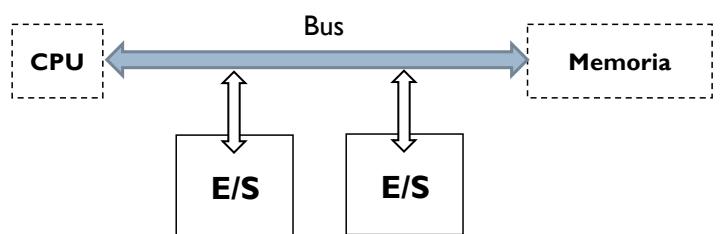
# Contenidos

1. Introducción
2. Periféricos
  - Concepto de periférico
  - Clasificación y tipos de periféricos
  - Estructura general de un periférico
    - Caso de estudio: disco duro, discos de estado sólido
3. Buses
  - Estructura y funcionamiento
  - Jerarquía de buses
4. Módulos de E/S
  - Estructura
  - Características
  - Técnicas de E/S

# Introducción

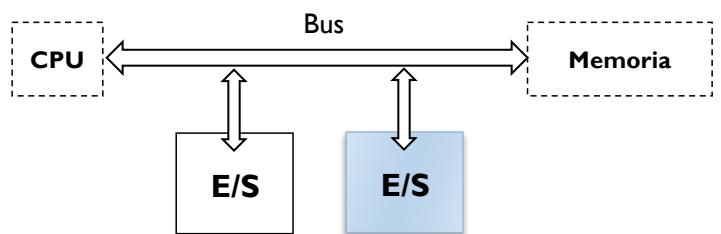


# Introducción



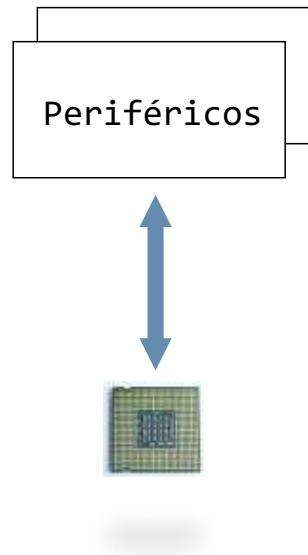
- ▶ Qué es un bus de interconexión

# Introducción



- ▶ Qué es un periférico
- ▶ Qué es un módulo de entrada/salida
- ▶ Cómo se acceden a los datos de los periféricos

# Concepto de periférico



## ► Periférico:

- ▶ Todo aquel dispositivo externo que se conecta a un procesador a través de la unidades o **módulos de entrada/salida (E/S)**.
- ▶ Permiten almacenar información o comunicar el computador con el mundo exterior.

# Clasificación de periféricos (por uso)



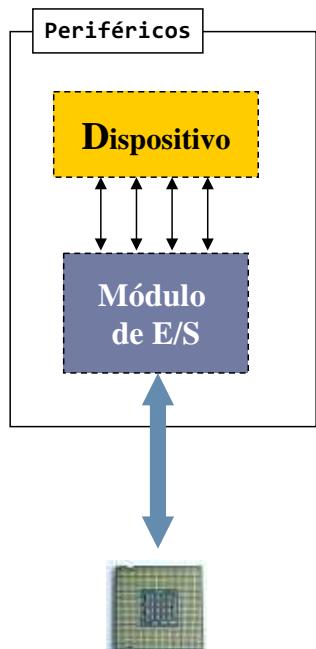
## ▶ Comunicación:

- ▶ Hombre-máquina
  - (Terminal) teclado, ratón, ...
  - (Impresa) plotter, escáner, ...
- ▶ Máquina-máquina (Módem, ...)
- ▶ Medio físico
  - (Lectura/accionamiento) x (análogo/digital)

## ▶ Almacenamiento:

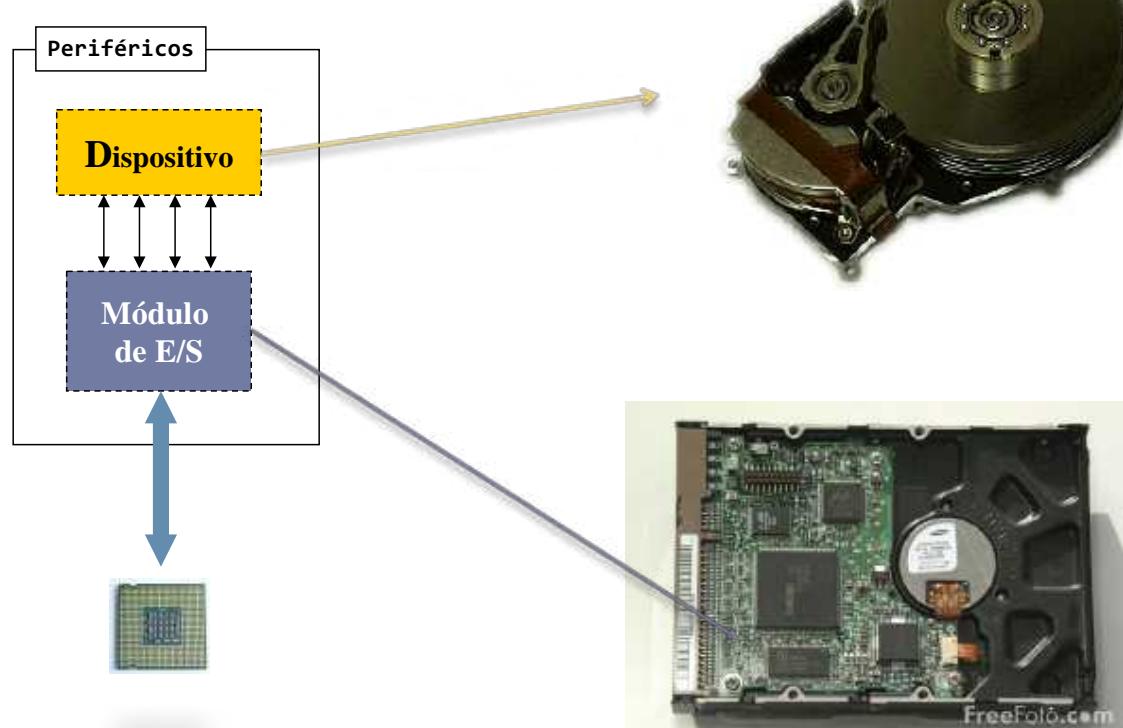
- ▶ Acceso “directo” (Discos, DVD, ...)
- ▶ Acceso secuencial (Cintas)

# Estructura general de un periférico



- ▶ Compuesto de:
  - ▶ **Dispositivo**
    - ▶ Hardware que interactúa con el entorno
  - ▶ **Módulo de Entrada/Salida**
    - ▶ También denominado **controlador**
    - ▶ Interfaz entre dispositivo y el procesador, que le oculta las particularidades de éste

## Ejemplo Disco magnético



## Un poco de historia...



- ▶ El primer disco duro apareció en 1956
  - ▶ Encargo de las fuerzas Aéreas de EEUU
  - ▶ Se le llamó IBM RAMAC 305
    - ▶ 50 discos de aluminio de 61 cm (24") de diámetro
    - ▶ 5 MB de datos
    - ▶ Giraba a 3.600 revoluciones por minuto
    - ▶ Contaba con una velocidad de transferencia de 8,8 Kbps
    - ▶ Alquiler por año 35000 \$
    - ▶ Pesaba cerca de una tonelada

## Un poco de historia...

- ▶ En 1980 aparece el primer disco de 5 1/4”
  - ▶ 5 MB
  - ▶ 10.000 \$



- ▶ En 1997 aparece el primer disco a 15000 RPM

## Un poco de historia...



1956

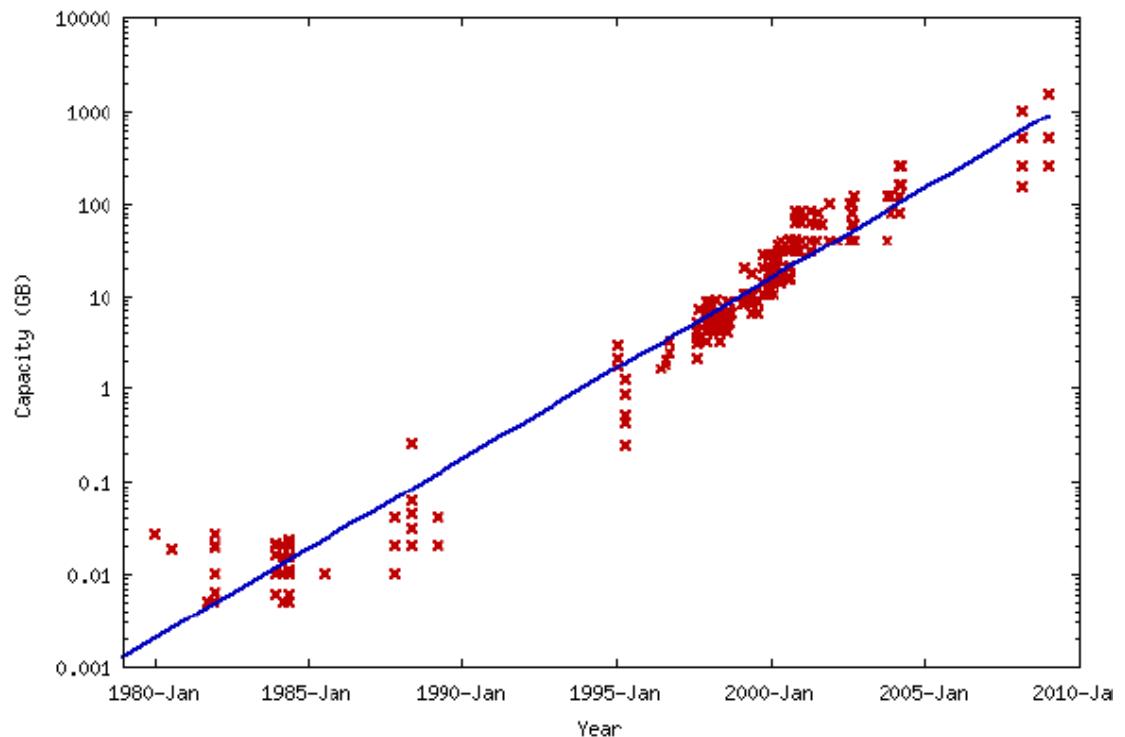


80's



2005

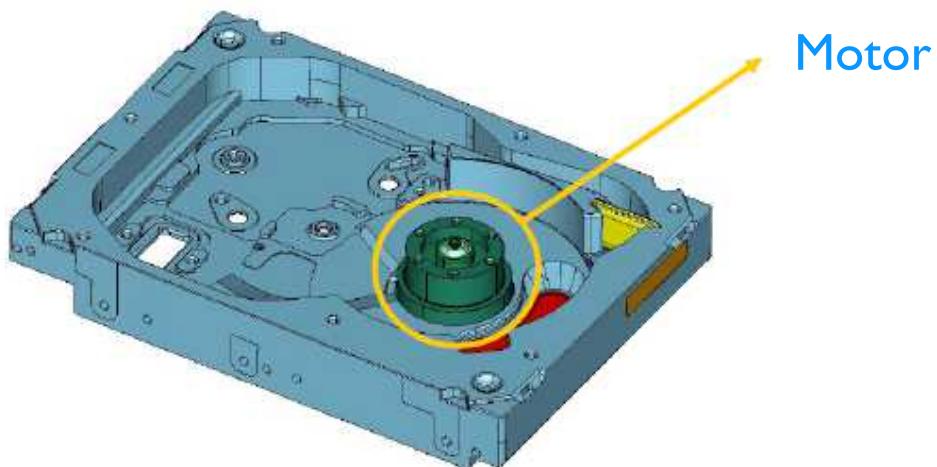
## Un poco de historia...



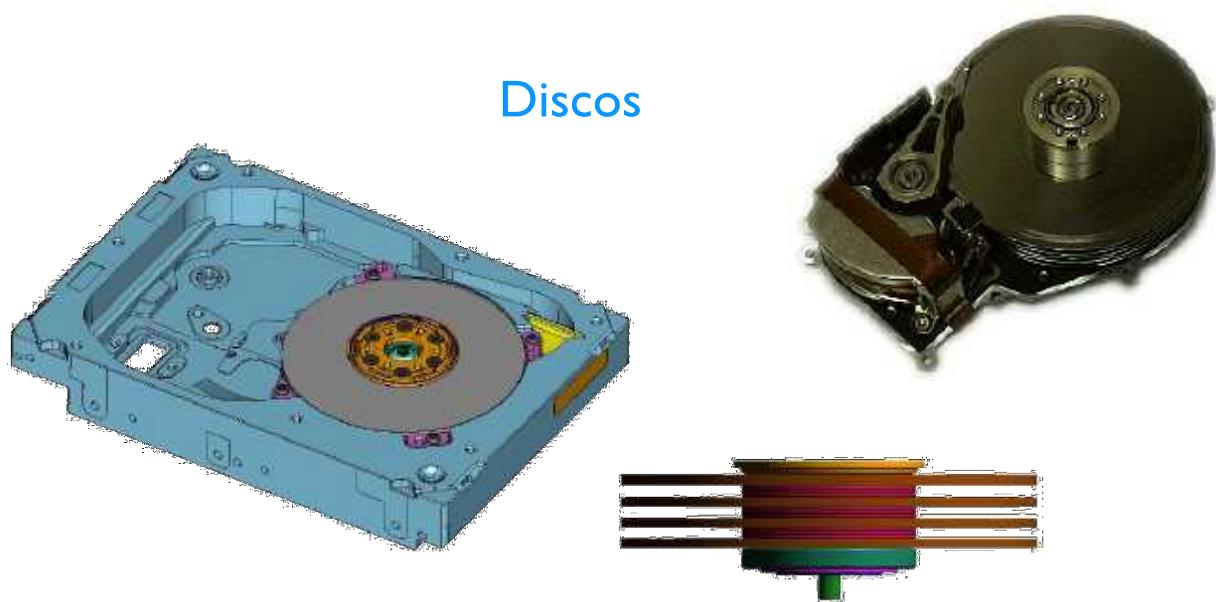
## Un poco de historia...

	<b>Tasa de crecimiento por año</b>
Capacidad	1.93/año
Coste	0.6/año
Prestaciones	0.05/año

# Anatomía de un disco duro



# Anatomía de un disco duro

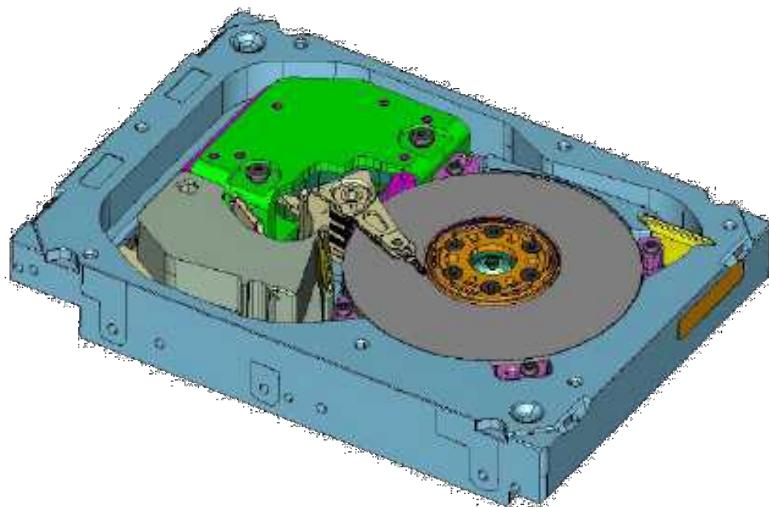


# Anatomía de un disco duro

Cabezas  
lectoras/escritoras



# Anatomía de un disco duro



Módulo de control y  
mecánica

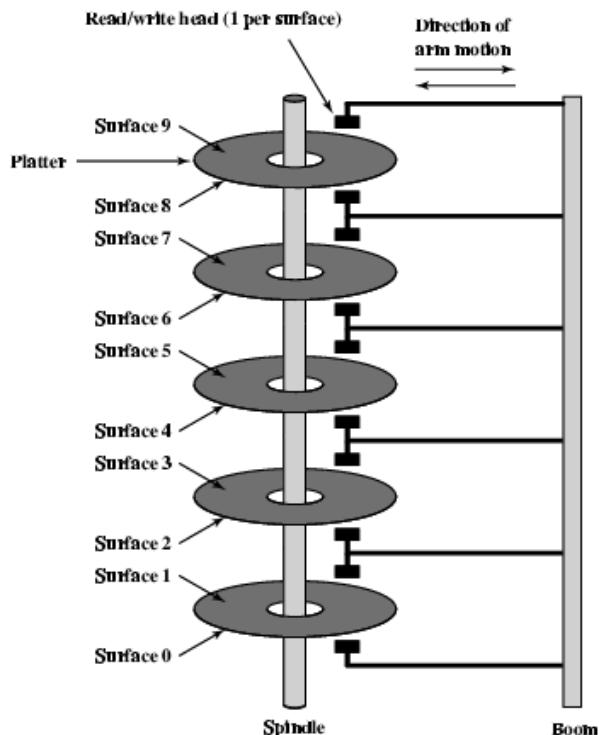
# Anatomía de un disco duro



<http://www.snia.org>

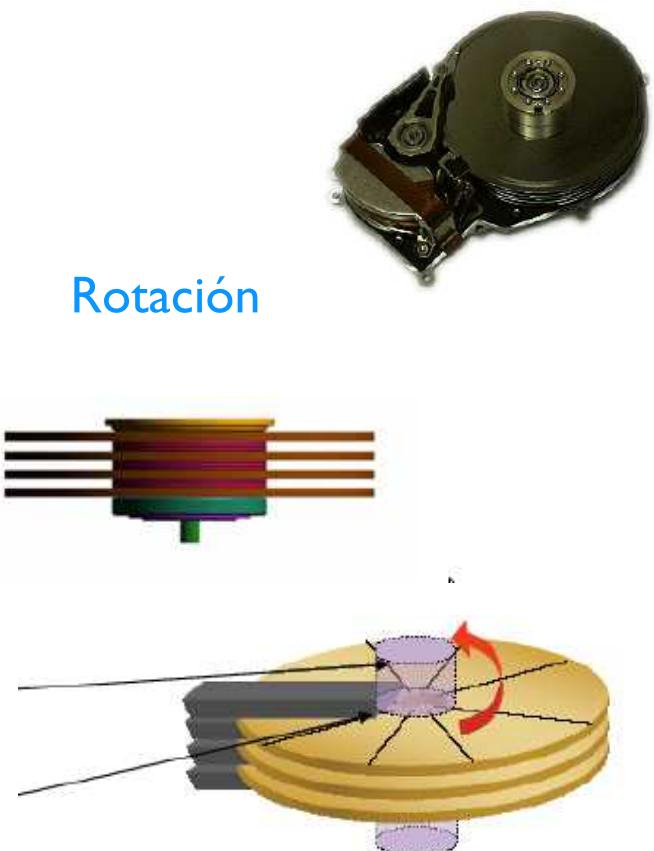
- ▶ **Controlador de disco**
  - ▶ Planificación de comandos
  - ▶ Corrección de errores
  - ▶ Optimización
  - ▶ Comprobación de integridad
  - ▶ Control de las revoluciones por minuto (RPM)
  - ▶ Caché de disco

# Múltiples platos

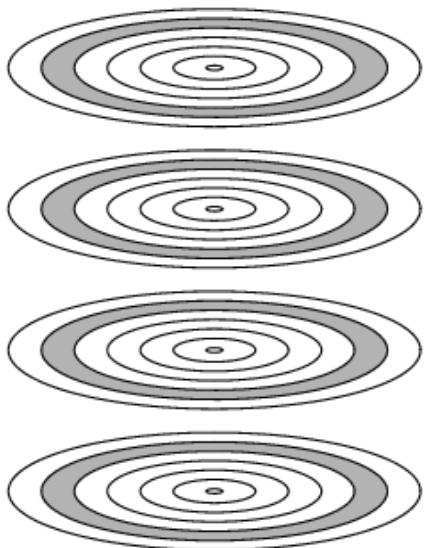


<http://www.snia.org>

Rotación

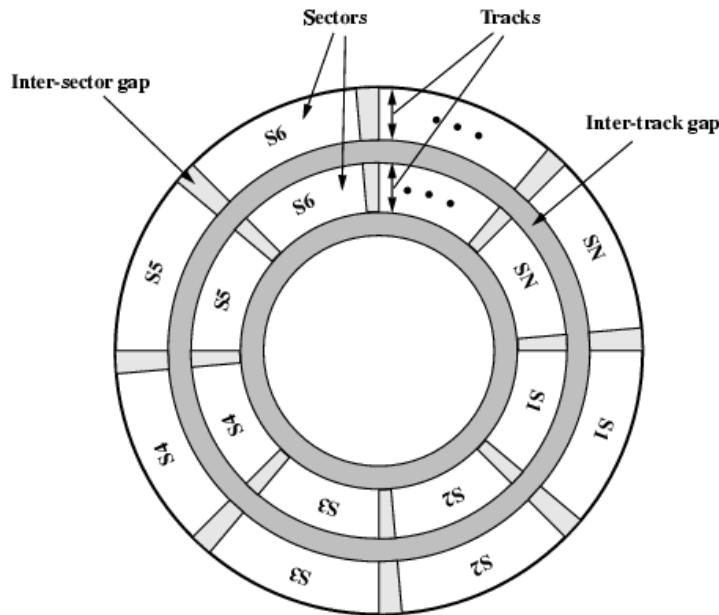


# Cilindros



- ▶ **Cilindro:** información accedida por todas las cabezas en una rotación

# Almacenamiento: pistas y sectores



## Pista:

- ▶ Un anillo del plato

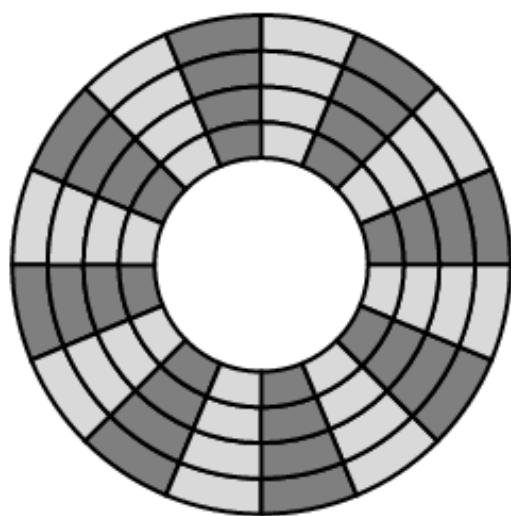
## Sector:

- ▶ División de la superficie del disco realizada en el formateo (típicamente 512 bytes)

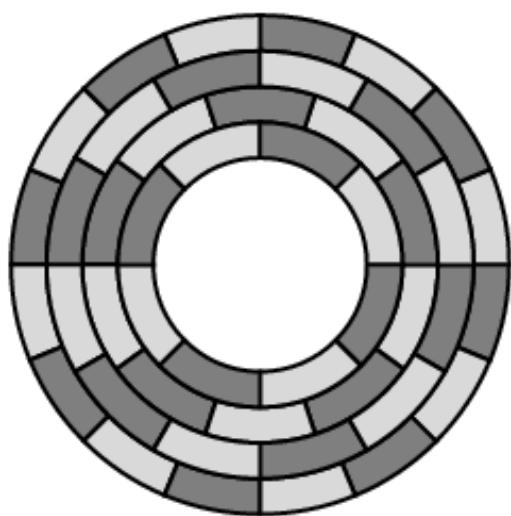
## Bloques:

- ▶ El sistema de ficheros escribe en bloques
- ▶ Grupo de sectores

# Distribución de sectores

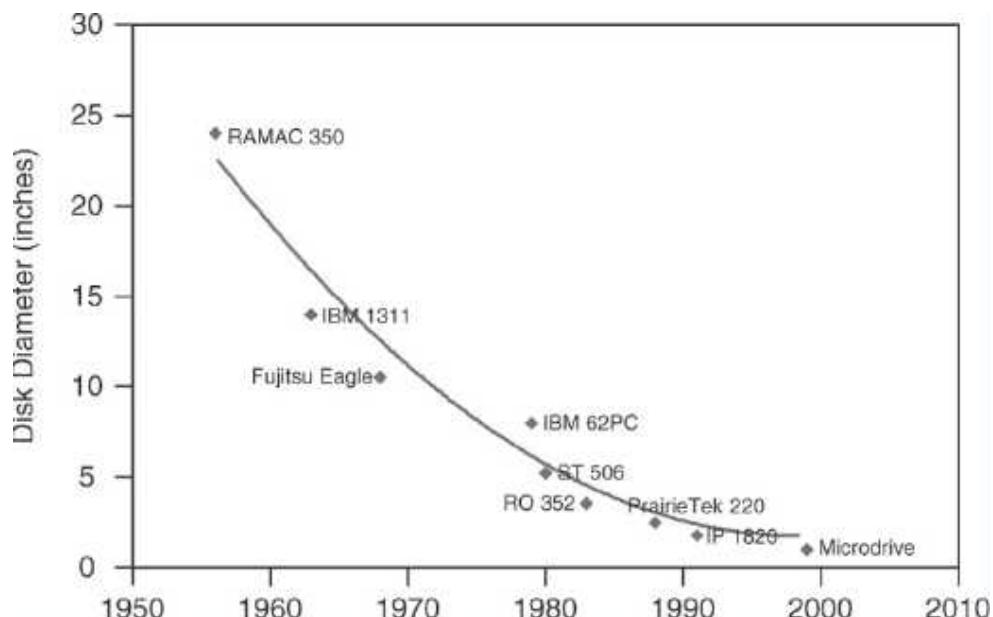


(a) Constant angular velocity



(b) Multiple zoned recording

# Evolución del tamaño de los discos



Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David Wang  
Elsevier

# Medida de la capacidad

- ▶ Bits por pulgada cuadrada
  - ▶ Dependen de la cabeza de lectura/escritura, del medio de grabación, de la rotación del disco y de la velocidad a la que el bus puede aceptar datos.
- ▶ Pistas por pulgada
  - ▶ Dependen de la cabeza de lectura/escritura, el medio de grabación, la precisión con la que la cabeza puede posicionarse y la capacidad del disco para girar en un círculo perfecto

# Capacidad de almacenamiento

## ▶ Para discos con velocidad angular constante

- ▶  $n_s$ : número de superficies
- ▶  $p$ : número de pistas por superficie
- ▶  $s$ : número de sectores por pista
- ▶  $t_s$ : bytes por sector

$$\text{Capacidad} = n_s \times p \times s \times t_s$$

## ▶ Para discos con múltiples zonas

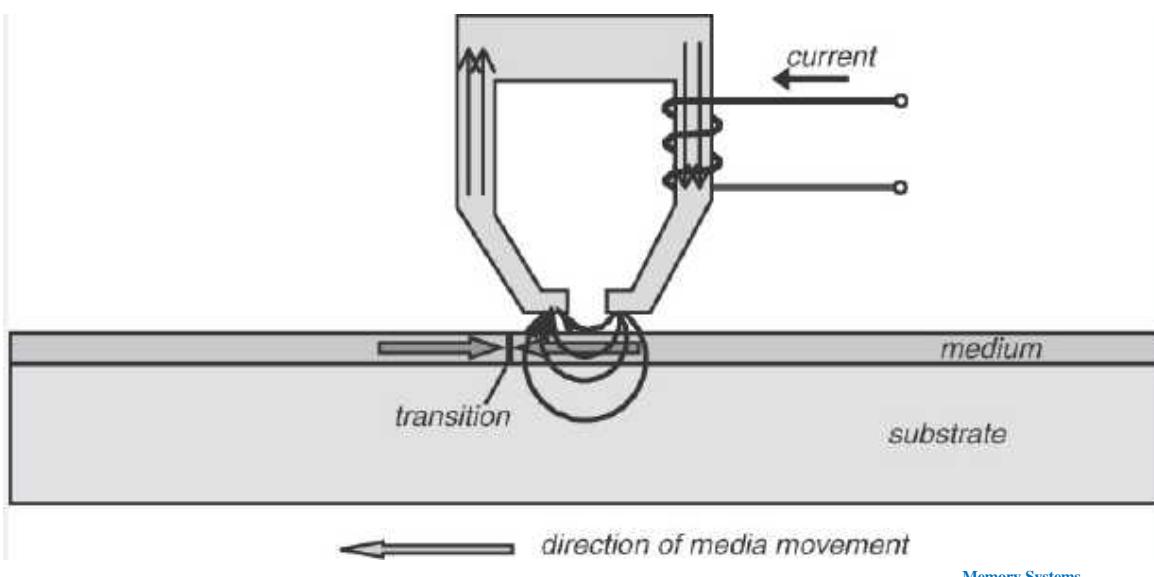
- ▶  $z$ : número de zonas
- ▶  $p_i$ : pistas de la zona  $i$
- ▶  $s_i$ : sectores por pista de la zona  $i$

$$\text{Capacidad} = n_s \times t_s \times \sum_{i=1}^z (p_i \times s_i)$$

# Métodos de grabación

- ▶ En la última década se ha incrementando en un 100% la densidad de grabación
- ▶ Cada bit en una pista consta de múltiples granos magnéticos
- ▶ El tamaño de estos granos no puede reducirse por debajo de los 10 nanómetros debido a:
  - ▶ Efecto superparamagnético
  - ▶ La temperatura ambiente podría afectar a la estabilidad de los granos
- ▶ Técnicas de grabación:
  - ▶ Longitudinal: guardan datos de forma longitudinal sobre un plano horizontal
  - ▶ Perpendicular: los datos se alinean de forma vertical, lo que permite disponer de mayor espacio y guardar más información

# Cabeza lectora/escritora



Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David Wang  
Elsevier

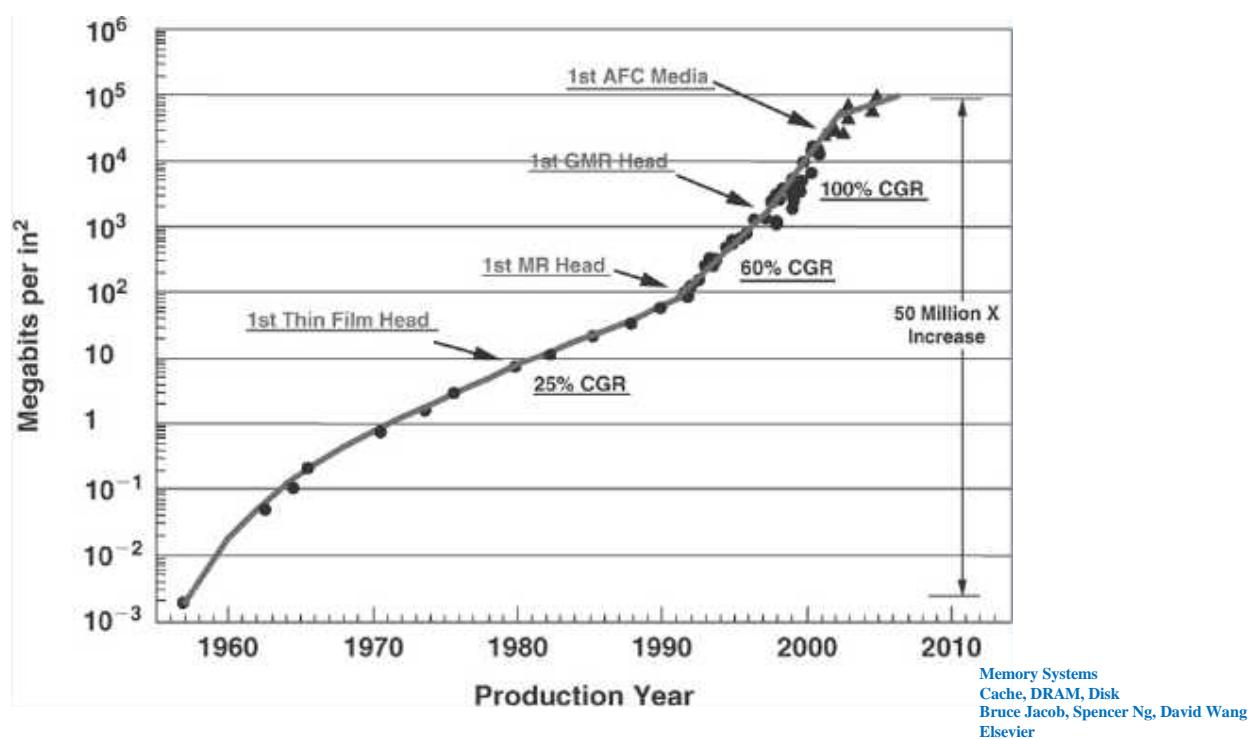
## Densidad superficial

- ▶ Las mejoras en la capacidad de los disco se expresa como mejorar en la **densidad superficial**:

$$\text{Areal density (AD)} = \frac{\text{Tracks}}{\text{Inch}} \text{ on a disk surface} \times \frac{\text{Bits}}{\text{Inch}} \text{ on a track}$$

- ▶ Hasta 1988 la tasa de mejora anual en AD fue del 29%
- ▶ Entre 1988 y 1996 la tasa de mejora fue del 60% al año
- ▶ Entre 1997 y 2003 la tasa de incremento fue del 100% anual
- ▶ Entre 2003-2011 la tasa de incremento ha sido del 30% anual
- ▶ En 2011 la mayor densidad en productos comerciales fue de 400 billones de bits por pulgada cuadrada
- ▶ El coste por bit se ha mejorado en un factor de 1.000.000 entre 1983 y 2011

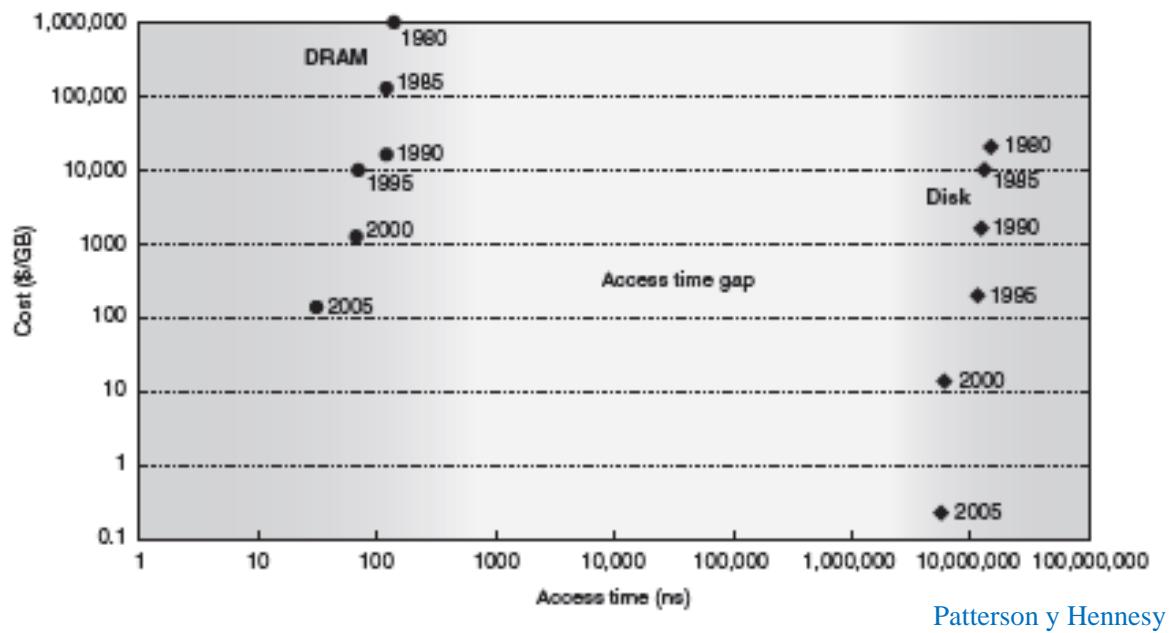
## Evolución de la densidad



## Discos y memoria principal

- ▶ La latencia de una memoria DRAM es 100.000 veces menor que la de un disco
- ▶ El coste por GB para una memoria DRAM está entre 30 y 150 veces el coste por GB de un disco duro
- ▶ En 2015:
  - ▶ Un disco de 8 TB transfiere 190 MB/s con un coste 250 \$
  - ▶ Un módulo de 8 GB de DDR4 transfiere 25 GB/s y cuesta aprox. 70\$

# Discos y memoria principal

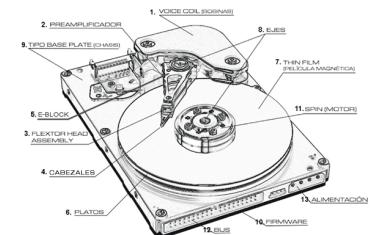


# Direccionamiento

- ▶ Tipos de direccionamiento
  - ▶ **Direccionamiento físico**: cilindro-pista-sector. Un sector queda determinado por estos tres valores
  - ▶ **Direccionamiento de bloques lógicos (LBN)**
    - ▶ Cada sector se identifica con un bloque lógico y la correspondencia la hace el propio disco
- ▶ Los controladores de disco actuales realizan la traducción de LBN a dirección física

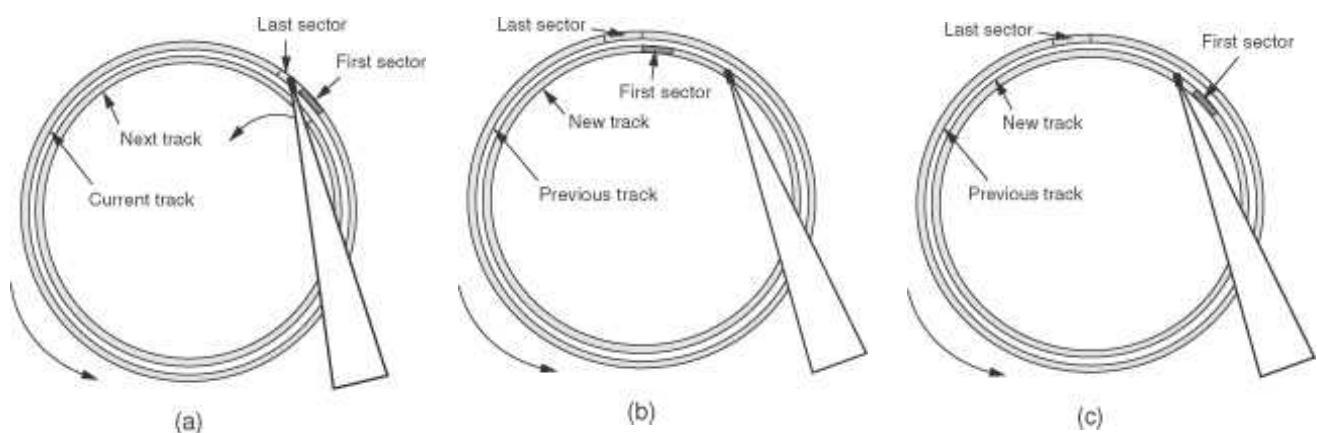
# Tiempo de acceso

$$\triangleright T_{\text{acceso}} = T_{\text{búsqueda}} + T_{\text{latencia}} + T_{\text{transferencia}}$$



- ▶ **Tiempo de búsqueda ( $T_{\text{búsqueda}}$ )**: tiempo necesario para mover la cabeza desde el cilindro actual al cilindro sobre el que se quiere operar
- ▶ **Latencia de rotación ( $T_{\text{latencia}}$ )**: tiempo que pasa hasta que el sector deseado pasa por debajo de la cabeza de lectura/escritura
  - ▶  $T_{\text{latencia}} = \text{Tiempo medio para recorrer media pista}$
- ▶ **Tiempo de transferencia ( $T_{\text{transferencia}}$ )**: tiempo necesario para recorrer un sector y transferir los datos de él

# Posicionamiento y rotación



Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David Wang  
Elsevier

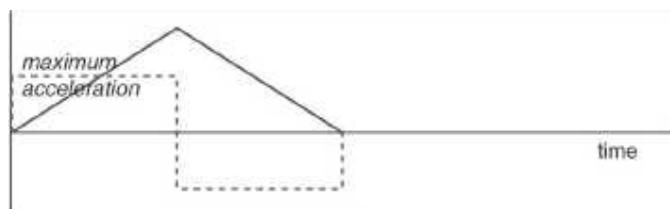
# Tiempo de posicionamiento

- ▶ Elementos que incluye:
  - ▶ Tiempo de aceleración de la cabeza ( $T_{acc}$ )
  - ▶ Tiempo con velocidad constante ( $T_c$ ) para distancias largas
  - ▶ Tiempo de deceleración ( $T_{dec}$ )
  - ▶ Tiempo de colocación ( $T_s$ ). Este tipo domina al resto de tiempos
  - ▶ El tiempo de posicionamiento medio se toma como el tiempo de posicionamiento entre dos pitas aleatorias

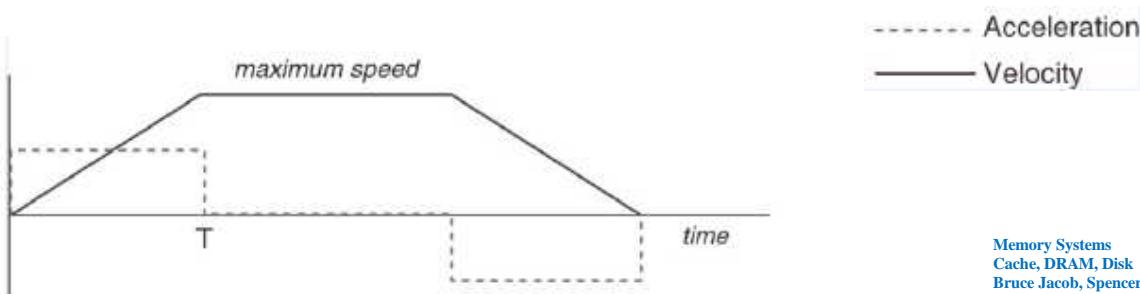
$$D_{average} = \frac{1}{2} \times a_{acc} \times t_{acc}^2 + a_{dec} \times t_{dec}^2$$

# Tiempo de posicionamiento

## ▶ Posicionamiento entre pistas cercanas



## ▶ Posicionamiento entre pistas lejanas



Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David Wang  
Elsevier

## Tiempo de posicionamiento

- ▶ Cuando la aceleración y deceleración son iguales:

$$t_{acc} = t_{dec} = \sqrt{\frac{D_{average}}{a}}$$

$$t_{seek} = 2 \times \sqrt{\frac{D_{average}}{a}} + t_s$$

## Tiempo de posicionamiento

- ▶ El aumento en la densidad de grabación da lugar a cilindros con mayor capacidad
  - ▶ Incrementa la probabilidad de reducir el número de posicionamientos de la cabeza
  - ▶ Aumenta la probabilidad de que los siguientes datos a solicitar estén en el mismo cilindro lo que reduce los posicionamientos

## Latencia rotacional

- ▶ La latencia media es el tiempo en dar media vuelta

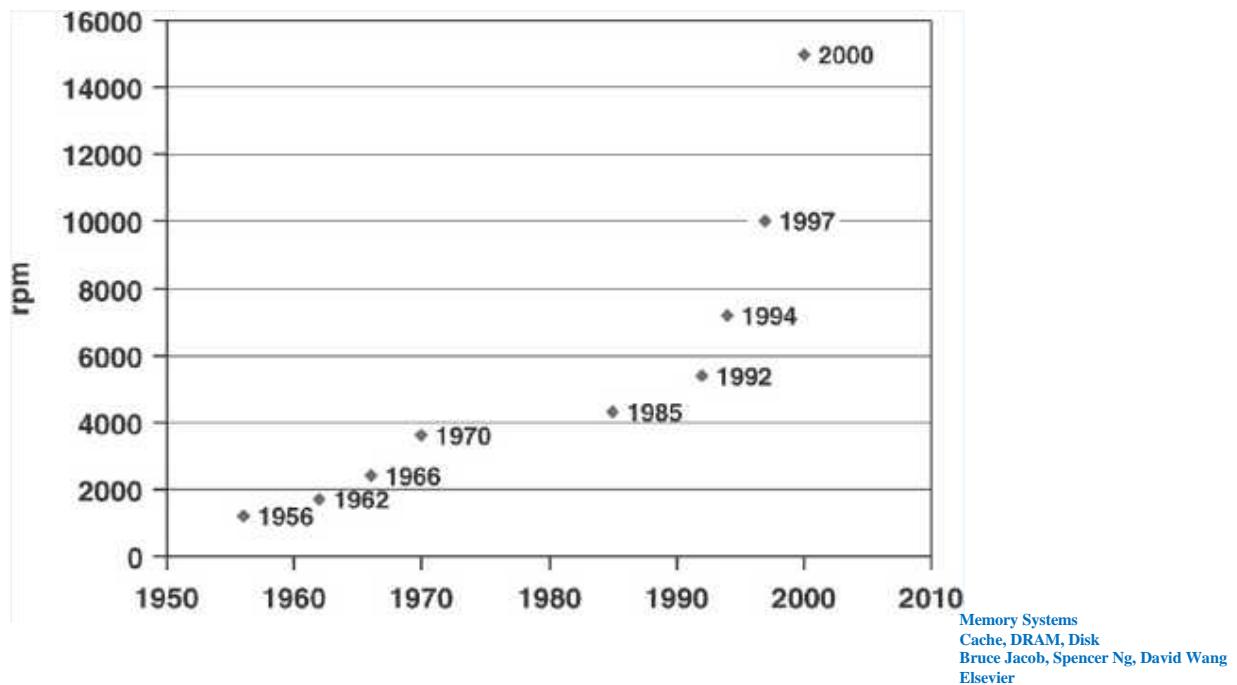
$$T_{rotate} = \frac{1}{2} \times \frac{60 \times 10^3}{RPM}$$

- Disco con latencia de acceso cero (*Zero-latency access*)
  - ▶ Transfieren los datos tan pronto como la cabeza se encuentra en la pista deseada a un bufer donde los bloques son después reordenados

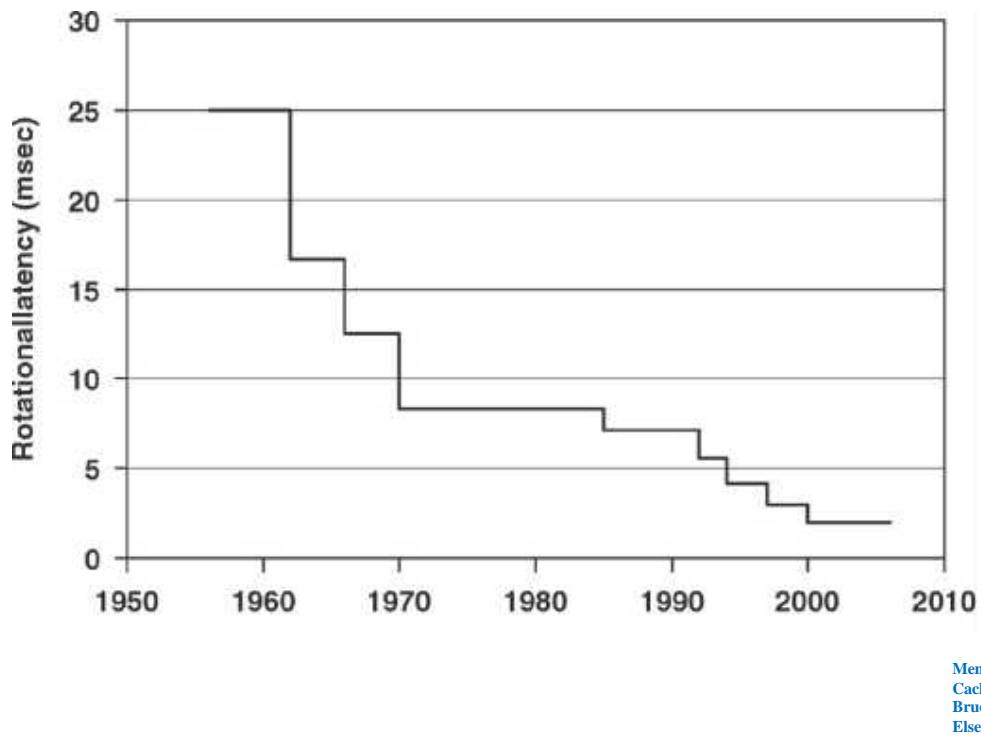
# Velocidad y latencia

<b>Rotational Speed</b>	<b>Rotational Latency (in ms)</b>
5400	5.6
7200	4.2
10000	3.0
12000	2.5
15000	2.0

# Evolución de las RPM



## Evolución de la latencia rotacional



# Tiempo de transferencia

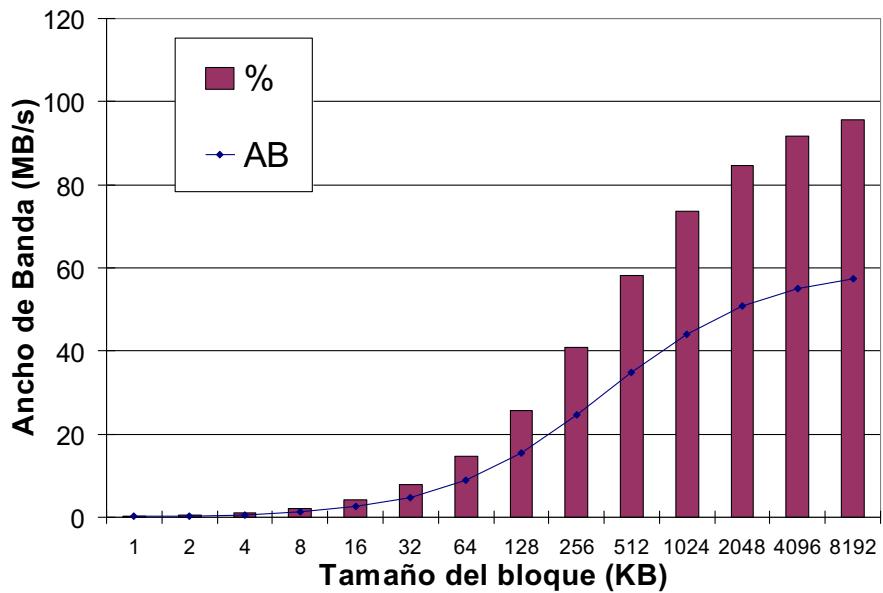
- ▶ Consta de dos partes:
  - ▶ Tasa de transferencia externa entre la memoria y la caché del disco
  - ▶ Tasa de transferencia interna entre la caché de disco y el medio de almacenamiento
- ▶ La tasa de transferencia interna se mide como:

$$T_{transfer} = \frac{N_{request}}{N_{track}} \times \frac{60}{RPM}$$

- ▶ Donde  $N_{track}$  es el número de sectores en una pista y  $N_{request}$  la cantidad de datos solicitados en número de sectores.
- ▶ En los discos con pistas externas más densas la relación de sectores de las pistas más externas a las más internas suele estar entre 1.43 y 1.58.

## Efecto del tamaño de la petición

- ▶ Efecto del tamaño de bloque ( $ta=6$  ms y  $AB = 60MB/s$ )



# Controlador de disco

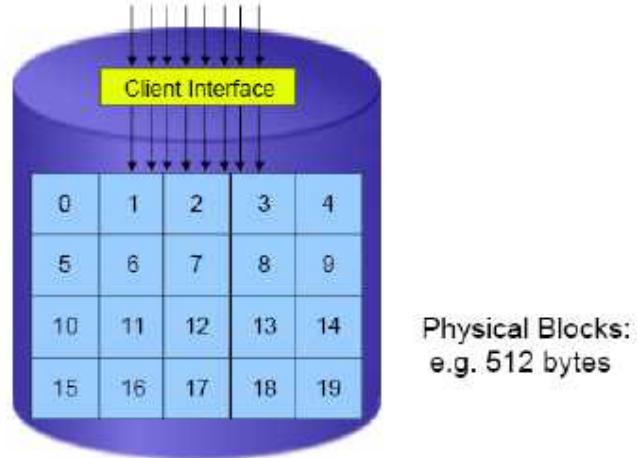
- ▶ Circuitos y componentes responsables de controlar el disco:
  - ▶ Interfaz de almacenamiento
  - ▶ Secuenciador de disco
  - ▶ Códigos correctores de errores (ECC)
  - ▶ Control del motor
  - ▶ Microprocesador
  - ▶ Controlador del bufer
  - ▶ Caché de disco

## Interfaz de almacenamiento

- ▶ Ofrece un protocolo estándar para comunicar el disco con el sistema.
  - ▶ IDE, SCSI, FC, SATA
- ▶ Suelen ofrecer la capacidad del disco como un array lineal de bloques de tamaño fijo
- ▶ El acceso se especifica con un LBN y el número de bloques a transferir
  - ▶ El controlador del disco traduce el LBN a dirección física (Cilindro, pista, sector)
- ▶ Existen otros tipos de dispositivos: *Object-based Storage Devices* (OSD) que crean objetos en el dispositivo mediante interfaces más potentes y amplias

# Acceso a un disco

SCSI, SAS, FCP, SRP, iSCSI, iSER, ATA, SATA



## Caché de disco

- ▶ Explota la localidad
- ▶ Reduce los accesos físicos al disco
  - ▶ Reduce la disipación de calor
  - ▶ Mejora el rendimiento
- ▶ No suele exhibir localidad temporal debido a las cachés de datos de los SSOO
- ▶ Normalmente implementa lectura adelantada (*prefetch*) para mejorar la localidad espacial
- ▶ Las cachés de disco normalmente se dividen en segmentos independientes correspondientes a flujos secuenciales de datos

# Caché de disco

- ▶ Algoritmos de reemplazo
  - ▶ Random Replacement
  - ▶ Least Frequently UsEd (LFU)
  - ▶ Least Recently Used (LRU)
- ▶ Su tamaño debería estar entre e. 0.1 y el 0.3 % del tamaño del disco

## Planificador de disco

- ▶ Los dispositivos actuales incluyen un planificador de las peticiones a disco cuyo objetivo es mejorar el tiempo de respuesta y minimizar el movimiento de las cabezas
- ▶ Políticas:
  - ▶ First Come First Served (FCFS)
  - ▶ Shortest Seek Time First (SSTF)
  - ▶ SCAN
  - ▶ C-SCAN
  - ▶ LOOK

## Otros elementos del controlador

- ▶ **Secuenciador de disco:** gestiona las transferencias de datos entre la interfaz de almacenamiento y el buffer de datos
- ▶ **ECC:** comprueba y corrige errores mediante códigos de corrección
- ▶ **Control del motor:** detecta la posición actual de la cabeza y controla el avance a la pista futura
- ▶ **Microprocesador:** controla el funcionamiento global del disco
- ▶ **Controlador del buffer:** controla las señales para el buffer de memoria

## Ejercicio

- ▶ ¿Cuántos bytes almacena un disco duro de 250 GB ?

# Recordatorio

- ▶ 1 KB = 1024 bytes, pero en el SI es 1000 bytes
- ▶ Los fabricantes de disco duros y en telecomunicaciones emplea el SI.
  - ▶ Un disco duro de 30 GB almacena  $30 \times 10^9$  bytes
  - ▶ Una red de 1 Mbit/s transfiere  $10^6$  bps.

Nombre	Abr	Factor	SI
Kilo	K	$2^{10} = 1,024$	$10^3 = 1,000$
Mega	M	$2^{20} = 1,048,576$	$10^6 = 1,000,000$
Giga	G	$2^{30} = 1,073,741,824$	$10^9 = 1,000,000,000$
Tera	T	$2^{40} = 1,099,511,627,776$	$10^{12} = 1,000,000,000,000$
Peta	P	$2^{50} = 1,125,899,906,842,624$	$10^{15} = 1,000,000,000,000,000$
Exa	E	$2^{60} = 1,152,921,504,606,846,976$	$10^{18} = 1,000,000,000,000,000,000$
Zetta	Z	$2^{70} = 1,180,591,620,717,411,303,424$	$10^{21} = 1,000,000,000,000,000,000,000$
Yotta	Y	$2^{80} = 1,208,925,819,614,629,174,706,176$	$10^{24} = 1,000,000,000,000,000,000,000,000$

## Ejercicio

- ▶ Un disco duro tiene una velocidad de rotación de 7200 rpm y una densidad constante de 604 sectores por pista. El tiempo medio de búsqueda es de 4 ms.
- ▶ Calcular el tiempo de acceso a un sector

# Ejercicio

- ▶ Sea un disco con un solo plato con las siguientes características:
  - ▶ Velocidad de rotación: 7200 rpm
  - ▶ Platos: 5, con 2 superficies por plato
  - ▶ Número de pistas de una cara del plato: 30000
  - ▶ Sectores por pista: 600 (de 512 bytes)
  - ▶ Tiempo de búsqueda: 1 ms por cada 100 pistas atravesadas
- ▶ Suponiendo que la cabeza está en la pista 0 y se solicita un sector de la pista 600, calcular:
  - ▶ Capacidad del disco duro
  - ▶ La latencia de rotación
  - ▶ Tiempo de transferencia de un sector
  - ▶ Tiempo de búsqueda del sector pedido

## Ejercicio (solución)

- ▶ Capacidad:
  - ▶  $5 \text{ platos} * 2 \text{ caras/plato} * 30.000 \text{ pistas/cara} * 600 \text{ sectores/pista} * 512 \text{ bytes/sector} = 85,8 \text{ GB}$
- ▶ Latencia de rotación:
  - ▶  $L_r = \text{Tiempo de media vuelta a una pista}$
  - ▶  $7.200 \text{ vueltas/minuto} \rightarrow 120 \text{ vuelta/segundo}$   
->  $0,0083 \text{ segundos/vuelta} \rightarrow 4,2 \text{ milisegundos}$  (media vuelta)
- ▶ Tiempo de transferencia de un sector:
  - ▶ Hay 600 sectores por pista y la pista se lee en 8,3 milisegundos
  - ▶  $8,3 / 600 \rightarrow 0,014 \text{ milisegundos}$
- ▶ Tiempo de búsqueda:
  - ▶ Cada 100 pistas 1 ms, y hay que ir a la pista 600
  - ▶  $600 / 100 = 6 \text{ milisegundos}$

## Ejercicio

- ▶ Sea un disco duro con tiempo medio de búsqueda de 4 ms, una velocidad de rotación de 15000 rpm y sectores de 512 bytes con 500 sectores por pista. Se quiere leer un fichero que consta de 2500 sectores con un total de 1,22 MB. Estimar el tiempo necesario para leer este fichero en dos escenarios:
  - ▶ El fichero está almacenado de forma secuencial, es decir, el fichero ocupa los sectores de 5 pistas adyacentes
  - ▶ Los sectores del fichero están distribuidos de forma aleatoria por el disco

## Fiabilidad

- ▶ Sea TMF el tiempo medio hasta el fallo
- ▶ Sea TMR el tiempo medio de reparación
- ▶ Se define la disponibilidad de un sistema como:

$$\text{disponibilidad} = \frac{\text{TMF}}{\text{TMF} + \text{TMR}}$$

- ▶ ¿Qué significa una disponibilidad del 99%?
  - ▶ En 365 días funciona correctamente  $99*365/100 = 361.3$  días
  - ▶ Está sin servicio 3.65 días
- ▶ Los fallos en los discos duros contribuyen al 20-55% de los fallos en el sistema de almacenamiento

# Consumo de energía

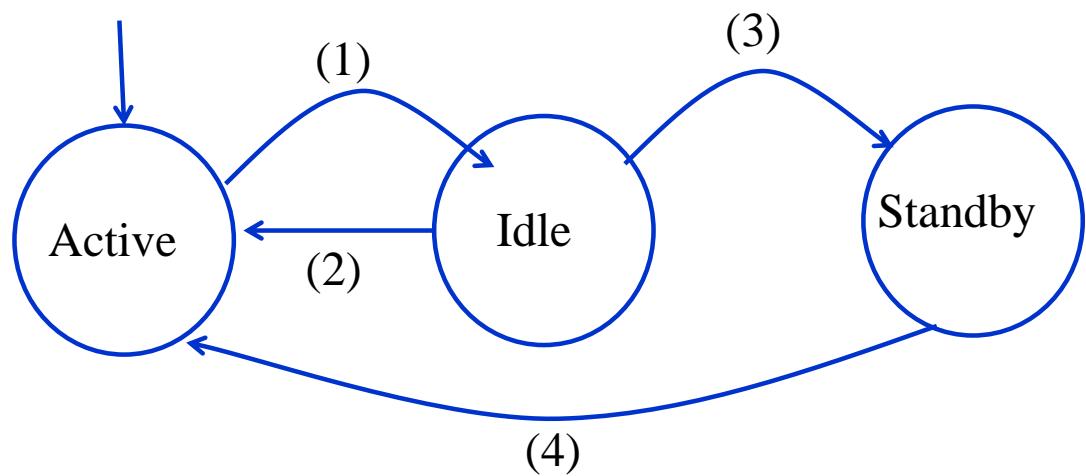
- ▶ Un típico disco ATA del 2011 consume:
  - ▶ 9 w cuando está inactivo
  - ▶ 11 w cuando está leyendo o escribiendo
  - ▶ 13 w en una operación de posicionamiento
- ▶ Estimación del consumo en un disco

$$Power = N_{platter} \times D_{platter}^{4.6} \times RPM^{2.8}$$

- ▶ Donde  $N_{platter}$  es el número de discos y  $D_{platter}$  el diámetro de los platos
- ▶ La temperatura es el factor que más afecta a la fiabilidad de los discos
  - ▶ Cada  $10^\circ$  de incremento por encima de los  $21^\circ$  reduce la fiabilidad de la electrónica en un 50%

## Diagrama de transición de estados de consumo de energía

R/W requests



## Transición de estados de consumo de energía

- ▶ (1): No hay peticiones pendientes pero los platos siguen girando, las cabezas se aparcán
- ▶ (2): El disco recibe una petición en el estado *idle*
- ▶ (3): Para reducir la energía el disco se para y las cabezas se sitúan fuera del disco
- ▶ (4): Nuevas peticiones, el disco tiene que activarse

## Estrategias de ahorro de energía

- ▶ Basada en *timeouts*. Cuando el disco está en estado *idle* durante un cierto tiempo pasa a *standby*
- ▶ Predicción dinámica. Basada en el funcionamiento de las aplicaciones
- ▶ Mecanismos estocásticos.
- ▶ Gestión de la energía guiado por la aplicación
  - ▶ La aplicación informa sobre el patrón de acceso ( en el código o generados por el compilador)

## Problemas del apagado de los discos

- ▶ Se incrementa el consumo cuando los platos tienen que volver a girar
- ▶ Reduce la fiabilidad de los discos. Los fabricantes suelen indicar el número de ciclos de start/stop que puede soportar un disco. Por encima de este valor la probabilidad de fallos se incrementa en un 50%
- ▶ Los métodos de ahorro de energía se suelen aplicar a dispositivos portátiles y no se aplican a servidores por las cargas de datos intensivas.

## Discos de estado sólido SSD (Solid State Drive)

- ▶ Dispositivo de almacenamiento de bloques basado en semiconductores que actúa como una unidad de disco
  - ▶ Basados en memorias Flash
    - ▶ Almacenamiento no volátil
  - ▶ Basados en memorias DDR
    - ▶ Requiere baterías y backup en disco para conseguir almacenamiento no volátil

HDD  
con partes móviles



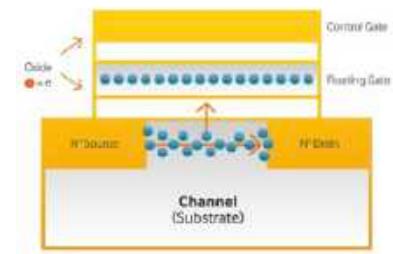
SDD  
sin partes móviles

# Memorias Flash

- ▶ Memorias no volátiles que se pueden borrar y grabar eléctricamente
- ▶ Tipos:
  - ▶ Flash NOR
    - ▶ Basadas en puertas NOR
    - ▶ Permite acceder a datos de 1 byte
    - ▶ Buenas para accesos aleatorios de alta velocidad
    - ▶ Usada en memorias BIOS (función de arranque)
    - ▶ Operaciones de lectura más rápidas
  - ▶ Flash NAND
    - ▶ Basadas en puertas NAND
    - ▶ Lee y escribe a alta velocidad en modo secuencial en tamaños de bloques
    - ▶ No puede acceder a bytes individuales
    - ▶ Mayor densidad y más baratas. Usada en SSD
    - ▶ Más duraderas, menos caras, más densas, operaciones de escritura/borrado más rápidas

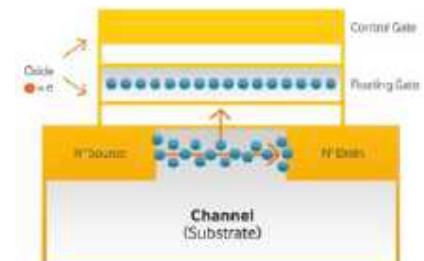
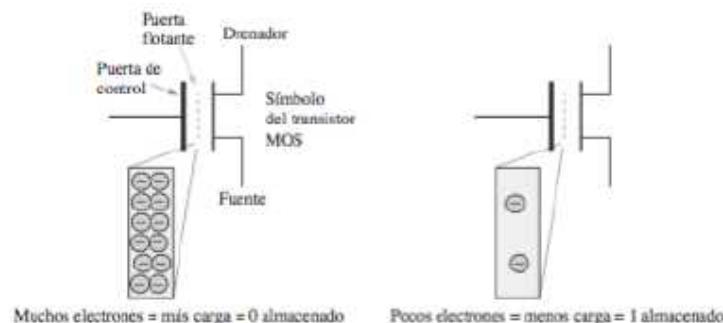
## Celdas de memoria

- ▶ Cada celda de almacenamiento está compuesta por un transistor MOS de puerta flotante
- ▶ Hay dos puertas aisladas por una capa de óxido
  - ▶ Puerta de control
  - ▶ Puerta flotante
- ▶ Los electrones fluyen libremente entre ambas puertas
- ▶ La puerta flotante está eléctricamente aislada atrapando los electrones



# Celdas de memoria

- ▶ El bit de datos se almacena como una carga o ausencia de carga en la puerta flotante, dependiendo de si se desea almacenar un 0 o un 1

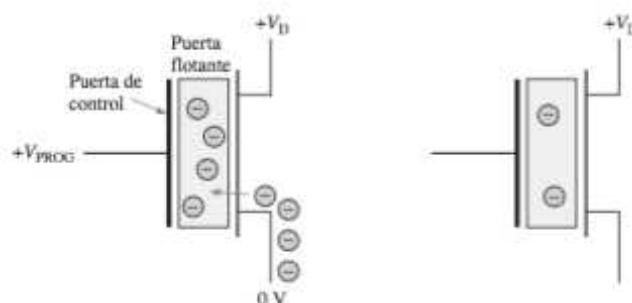


Fundamentos de Sistemas Digitales  
Thomas L. Floyd

# Operaciones básicas de una celda de memoria Flash

- ▶ **Programación**
  - ▶ Inicialmente todas las celdas están en el estado 1, porque la carga está eliminada
- ▶ **Operación de lectura**
- ▶ **Operación de borrado**

# Programación de una celda de memoria Flash

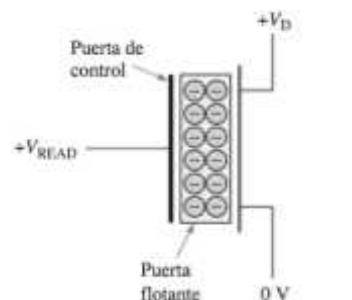


Fundamentos de Sistemas Digitales  
Thomas L. Floyd

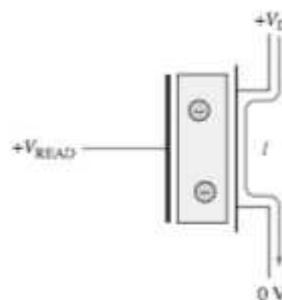
- ▶ Para almacenar un 0, se aplica a la puerta de control una tensión suficientemente positiva con respecto a la fuente, para añadir carga a la puerta flotante durante la programación (atrae los electrones)
- ▶ En el proceso de escritura se añaden electrones a aquellas puertas que deban almacenar un 0 y no se añade a las que deben almacenar un 1.
- ▶ Solo se escriben “0”
- ▶ Para almacenar un 1, no se añade ninguna carga, dejándose la celda en el estado de borrado

# Lectura de una celda de memoria Flash

- ▶ Se aplica una tensión positiva a la puerta de control



Cuando se lee un 0, el transistor permanece desactivado, porque la carga de la puerta flotante impide a la tensión de lectura exceder el umbral de activación.

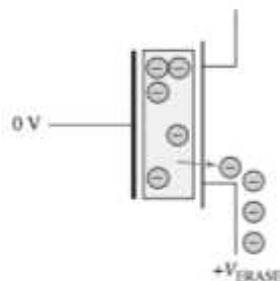


Cuando se lee un 1, el transistor se activa, porque la ausencia de carga en la puerta flotante permite que la tensión de lectura exceda el umbral de activación.

Fundamentos de Sistemas Digitales  
Thomas L. Floyd

## Borrado

- ▶ Durante el borrado se elimina la carga de todas las celdas de memoria. Se aplica un voltaje en sentido contrario para eliminar los electrones



Para borrar una célula, se aplica a la fuente una tensión suficientemente positiva con respecto a la puerta de control, con el fin de extraer la carga de la puerta flotante durante la operación de borrado.

Fundamentos de Sistemas Digitales  
Thomas L. Floyd

## Tipos de memorias Flash NAND

- ▶ Flash de celdas de un nivel (SLC)
  - ▶ Almacenan 1 bit por celda
  - ▶ 50000 - 100000 escrituras por celda
  - ▶ Usado fundamentalmente en aplicaciones militares e industriales
- ▶ Flash de celdas de múltiples niveles (MLC)
  - ▶ Almacenan varios bits por celda, en función del número de electrones almacenados en la celda
  - ▶ Ofrecen más capacidad pero menos duración (se desgastan más)
  - ▶ < 10000 escrituras por celda
  - ▶ Usado en electrónica de consumo
  - ▶ Menor coste
  - ▶ Mitad de prestaciones que las SLC

## Nivelación del desgaste

- ▶ Una memoria flash NAND solo puede escribir un determinado número de veces en cada bloque (o celda)
- ▶ Cuando se supera el límite, la celda se desgasta (su capa de óxido) y ya no almacena correctamente los electrones
- ▶ Nivelación del desgaste: proceso que usa el controlador de una unidad SSD para maximizar la duración de la memoria Flash
- ▶ Esta técnica nivela el desgaste en todos los bloques distribuyendo la escritura de datos por todos los bloques
  - ▶ Cuando se va a modificar un bloque se escribe en uno nuevo

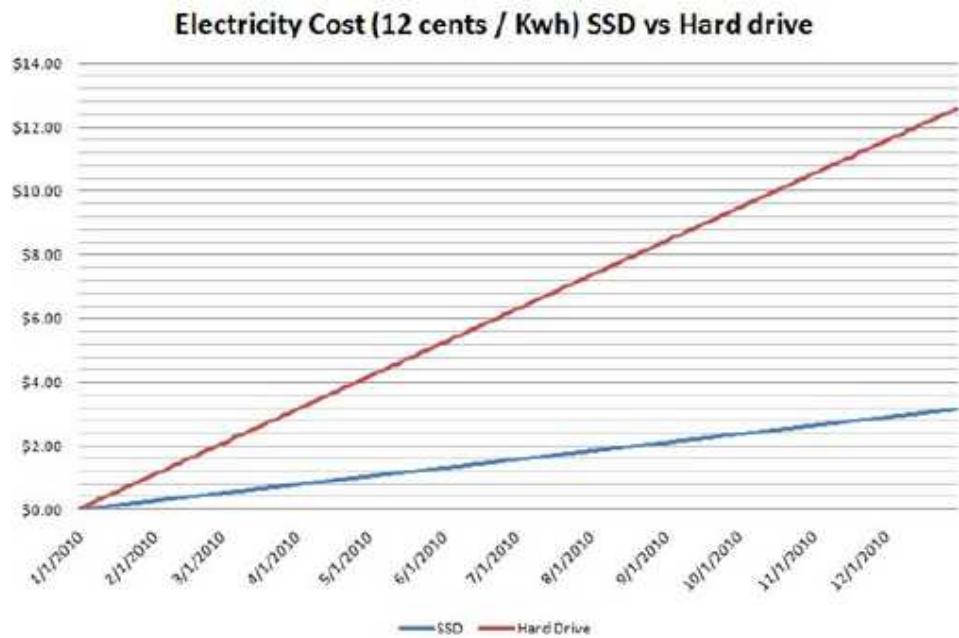
## Estructura de una Flash Nand

- ▶ Las flash NAND se dividen en bloques de 128 KB que se subdividen en páginas de 2KB
- ▶ El borrado se hace de un bloque completo
- ▶ La programación y lectura de una página

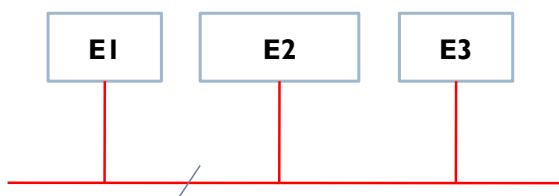
# Comparativa SDD vs HDD

	<b>SDD</b>	<b>HDD</b>
Tiempo de acceso	0.1 ms	5-8 ms
Operaciones de E/S por segundo	6000 io/s	400 io/s
consumo	2-5 warios	6-15 warios

# Consumo energético

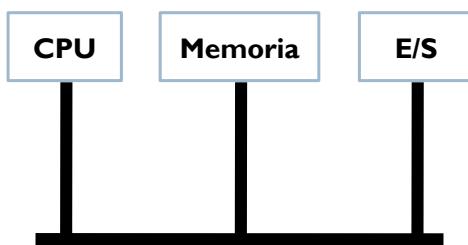


# Bus



- ▶ Un bus es un **camino de comunicación entre dos o más dispositivos**.
- ▶ Constituido por **varias líneas** de transmisión de bit.
- ▶ **Medio compartido, unívoco.**
- ▶ Permite transmitir varios bits entre dos elementos conectados a él

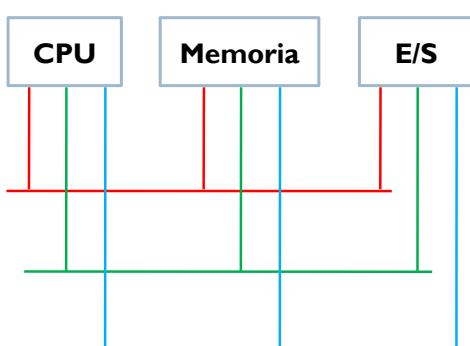
# Bus del sistema



## ▶ **Bus del sistema**

- ▶ Conecta los principales componentes del computador
- ▶ Representa la unión de tres buses:
  - ▶ Control
  - ▶ Direcciones
  - ▶ Datos

# Buses



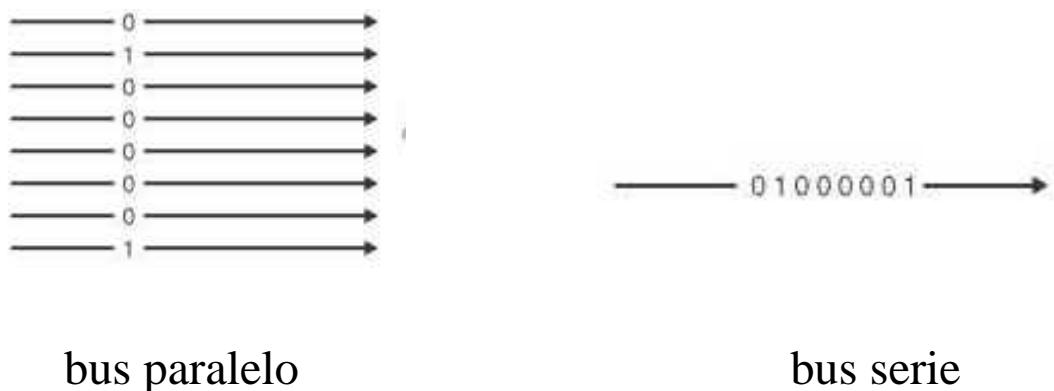
- ▶ **Bus de datos**
  - ▶ Transmite datos
  - ▶ Su anchura y velocidad influye mucho en las prestaciones
- ▶ **Bus de direcciones**
  - ▶ Direcciones de memoria y dispositivos de E/S
  - ▶ Su anchura determina la máxima capacidad de memoria
- ▶ **Bus de control**
  - ▶ Señales de control y temporización

## Características de un bus

- ▶ Ancho el bus
- ▶ Frecuencia
- ▶ Velocidad de transferencia
- ▶ Ancho de banda

## Características de un bus

- ▶ **Ancho del bus:** determina el número de bits que pueden transmitirse simultáneamente



## Características de un bus

- ▶ **Frecuencia:** frecuencia de reloj con la que puede operar
- ▶ **Velocidad de transferencia:** número de bytes por ciclo de reloj
- ▶ **Ancho de banda (tasa de transferencia):** bytes transmitidos por segundo
  - ▶ Velocidad de transferencia × frecuencia

## Ejercicio

- ▶ Calcular el ancho de banda de un bus de 32 bits y una frecuencia de 66 MHz

## Ejercicio (solución)

- ▶ Calcular el ancho de banda en MBps de un bus de 32 bits y una frecuencia de 66 MHz

$$\text{Ancho de banda} = \frac{32 \text{ bits} \times 66 \text{ MHz}}{8 \text{ bits por byte}} = \frac{32 \times 66 \cdot 10^6}{8} = 264 \text{ MBps}$$

## Método de arbitraje (protocolo del bus)

- ▶ Determina qué elemento de los que están conectados al bus puede acceder al bus
  - ▶ Esquema **centralizado**: un controlador del bus concede el uso del bus
    - ▶ Cuando un elemento quiere acceder al bus solicita permiso al controlador a través de las líneas de control (BUSRQ)
    - ▶ Cuando el bus está libre el controlador concede el uso (BUSACK)
  - ▶ Esquema **distribuido**: cada elemento conectado al bus incluye una lógica de control de acceso que permite usar de forma conjunta el bus (protocolo de acceso)

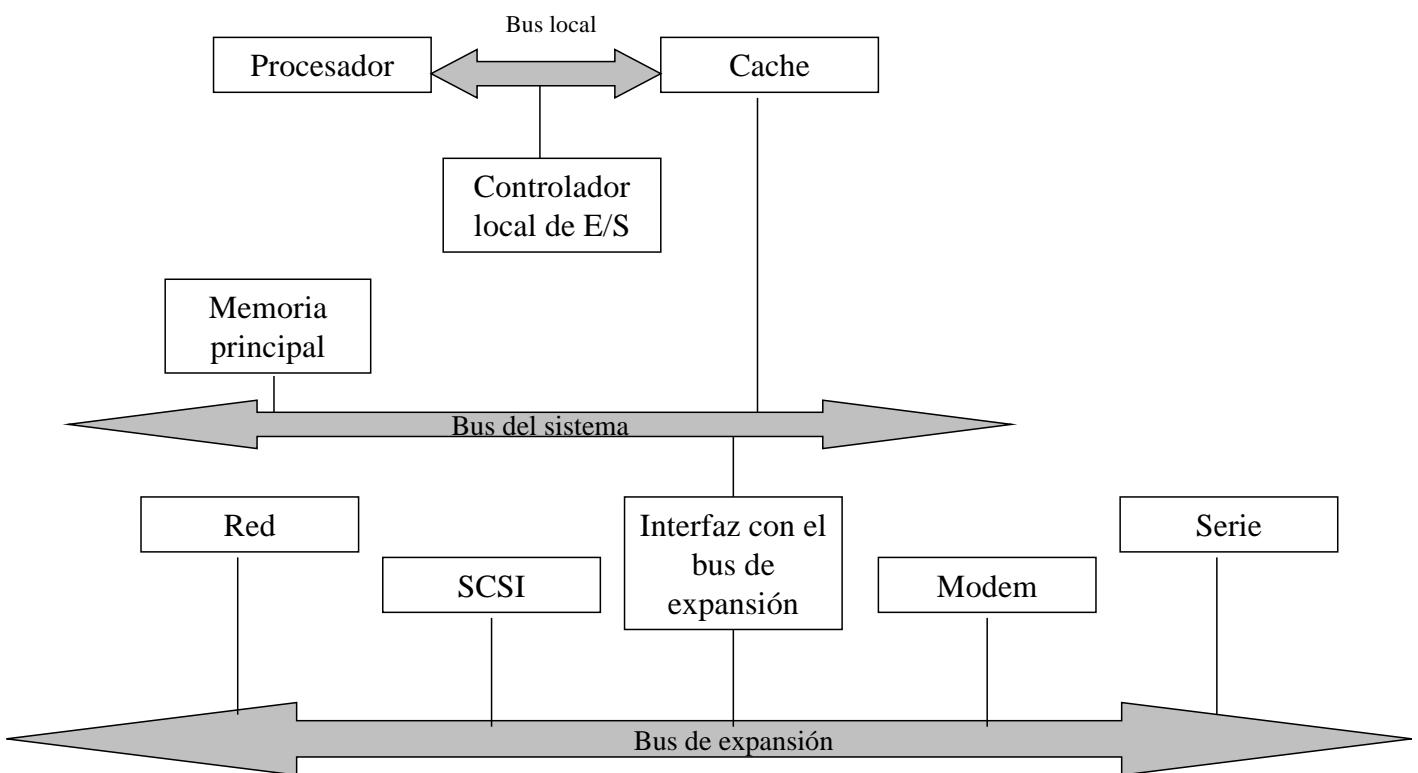
## Buses síncronos y asíncronos

- ▶ Un bus **síncrono** está gobernado por una señal de reloj y un protocolo de comunicación ajustado al funcionamiento del reloj
  - ▶ Rápido
  - ▶ Todos los dispositivos conectados a él deben operar a la misma frecuencia de reloj
- ▶ Un bus **asíncrono** no utiliza un reloj, la comunicación se realiza mediante el envío de órdenes a través de las líneas de control del bus

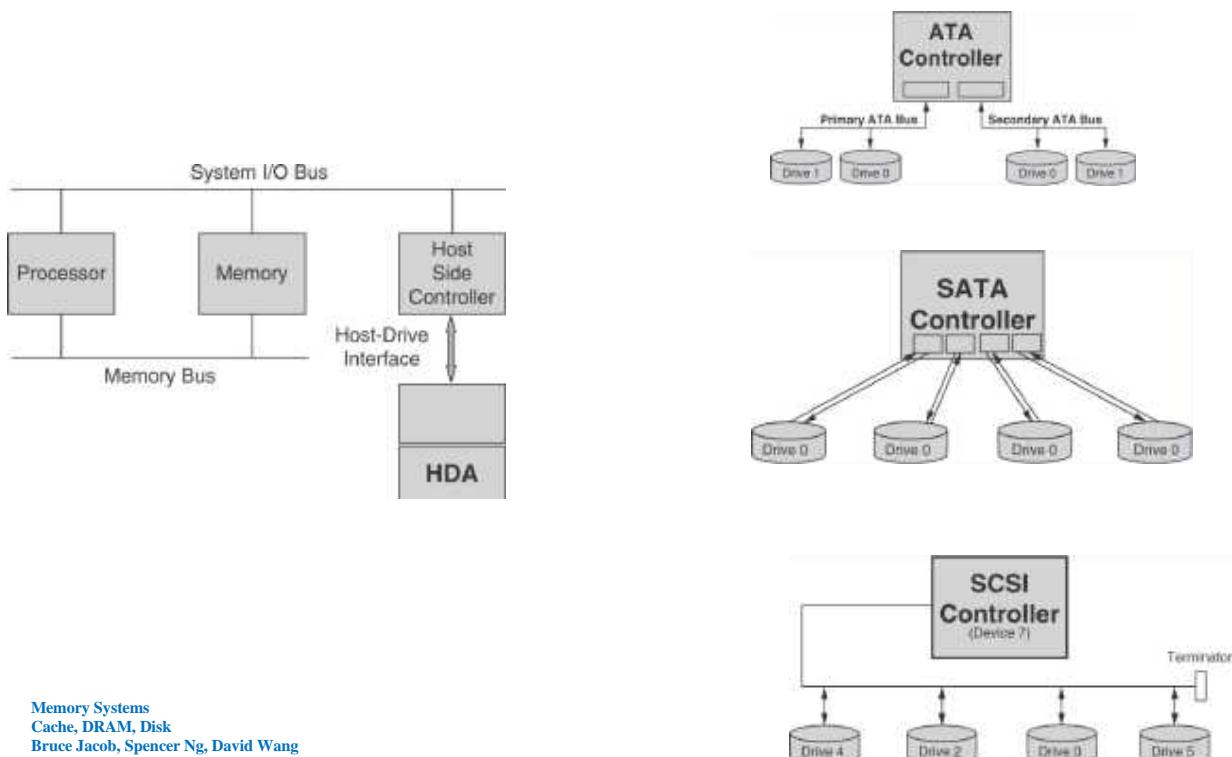
## Jerarquías de buses

- ▶ A más dispositivos conectados al bus, mayor es el retardo de propagación.
- ▶ A medida que aumenta el número de peticiones de transferencia, se puede producir un cuello de botella.
- ▶ Soluciones:
  - ▶ Aumentar la velocidad de transmisión de datos con buses más anchos.
  - ▶ Utilizar más buses de datos, organizados jerárquicamente.

# Jerarquías de buses

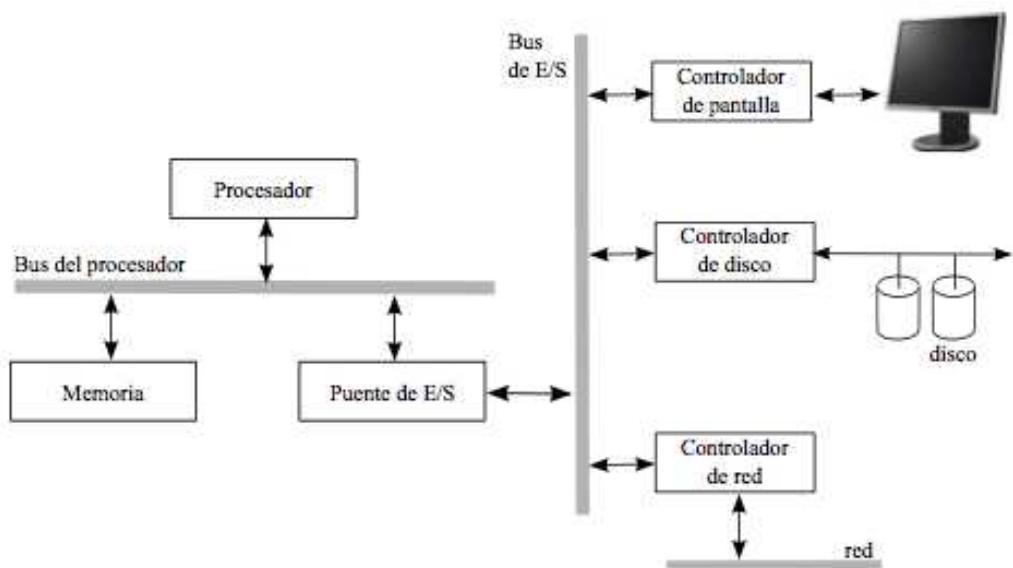


# Controladores de disco



Memory Systems  
Cache, DRAM, Disk  
Bruce Jacob, Spencer Ng, David Wang  
Elsevier

## Esquema de buses en un sistema informático típico



## Curiosidades: Familia USB



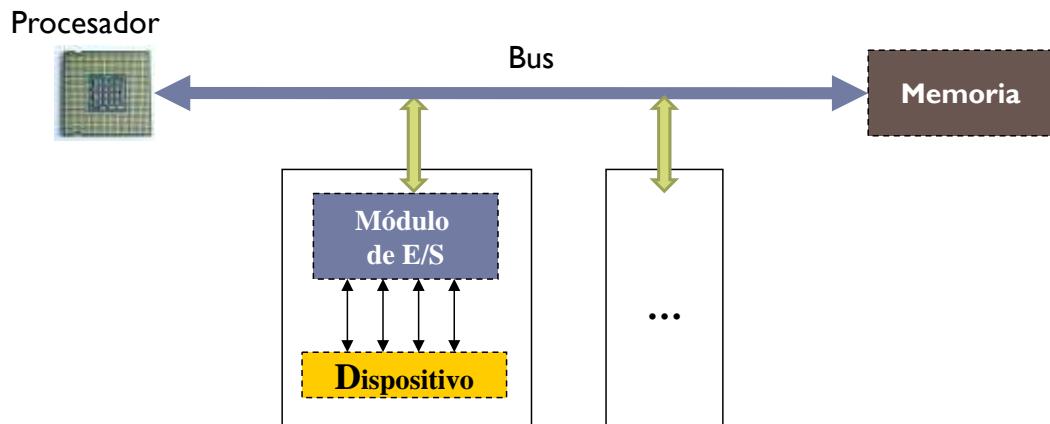
	Transfer. (por seg.)	Aparición
USB 3.0	600 MB/s	2010
USB 2.0	60 MB/s	2000
USB 1.0	1.5 MB/s y 187 KB/s	1996

	Song / Pic	256 Flash	USB Flash	SD-Movie	USB Flash	HD-Movie
	4 MB	256 MB	1 GB	6 GB	16 GB	25 GB
USB 1.0	5.3 sec	5.7 min	22 min	2.2 hr	5.9 hr	9.3 hr
USB 2.0	0.1 sec	8.5 sec	33 sec	3.3 min	8.9 min	13.9 min
USB 3.0	0.01 sec	0.8 sec	3.3 sec	20 sec	53.3 sec	70 sec

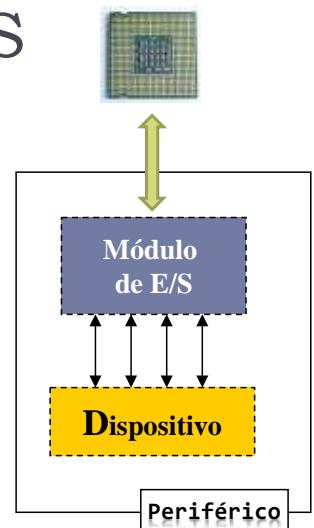
<http://www.unp.co.in/f140/comparison-of-usb-3-0-port-with-usb-2-0-and-usb-1-0-a-70063/>

## Módulo de E/S

- ▶ Las **unidades o módulos de E/S** realizan la conexión del procesador con los dispositivos periféricos.



## Necesidad de los módulo de E/S

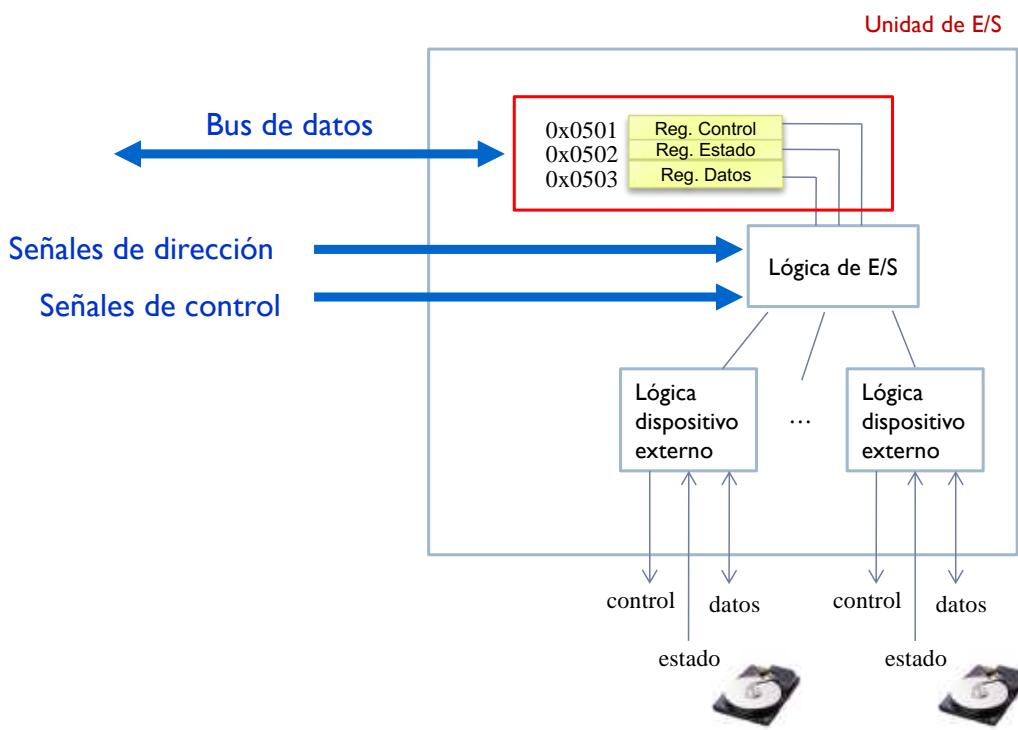


- ▶ Son necesarios debido a:
  - ▶ Gran variedad de periféricos.
    - ▶ Los periféricos son 'raros'
  - ▶ La velocidad de transferencia de datos de los periféricos es mucho menor que la de la memoria o el procesador.
    - ▶ Los periféricos son 'muy lentos'
  - ▶ Formatos y tamaños de palabra de los periféricos distintos a los del computador al que se conectan.

# Necesidades de los módulos de E/S

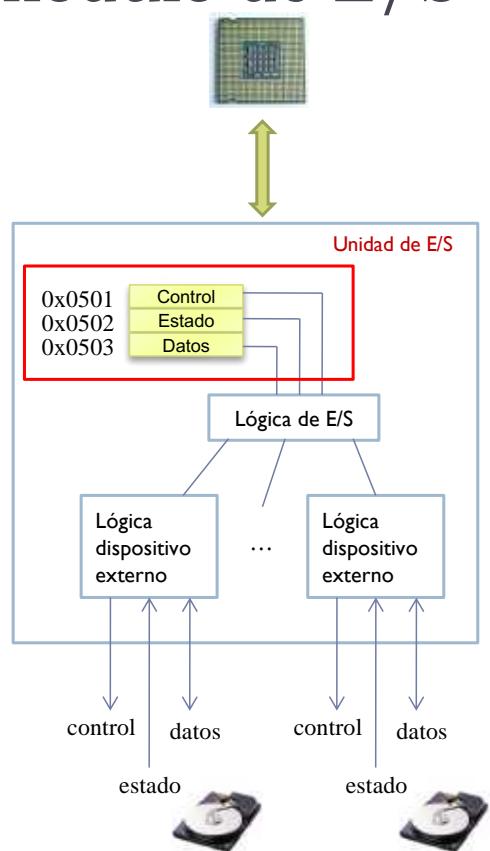
- ▶ Control y temporización
- ▶ Comunicación con el procesador o la memoria
- ▶ Comunicación con el periférico
- ▶ *Buffering* o almacenamiento intermedio
- ▶ Detección de errores

## Modelo simplificado de módulo de E/S



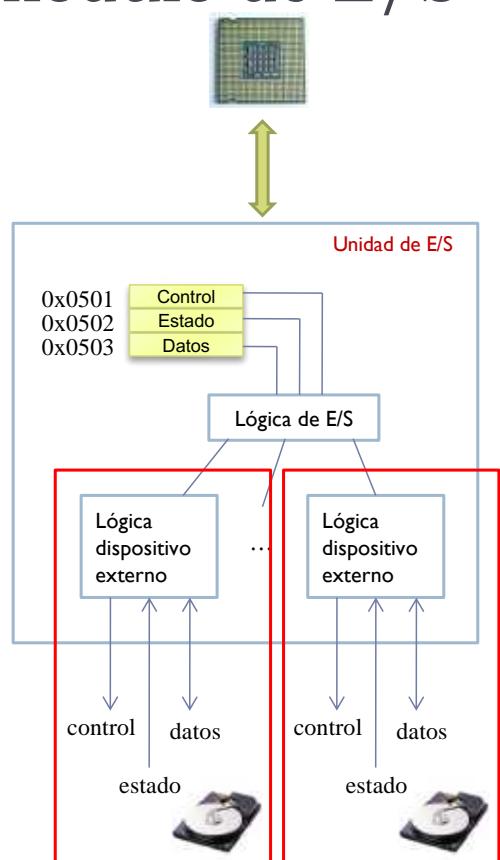
# Modelo simplificado de módulo de E/S

- ▶ Interacción entre procesador y Unidad de E/S a través de 3 registros:
  - ▶ Registro de **control**
    - ▶ Órdenes para el periférico
  - ▶ Registro de **estado**
    - ▶ Estado desde de la última orden
  - ▶ Registro de **datos**
    - ▶ Datos intercambiados Procesador/Perif.



# Modelo simplificado de módulo de E/S

- ▶ Interacción periférico/unidad de E/S:
  - ▶ **Líneas de datos:** transferencia de información
  - ▶ **Señales de estado:** diagnóstico del periférico
    - ▶ Ejemplos:
      - Nuevo dato disponible
      - Periférico encendido/apagado
      - Periférico ocupado
      - Periférico operativo o no
      - Error de operación
      - ...
  - ▶ **Señales de control:** accionamiento del periférico
    - ▶ Ejemplos:
      - Encender o apagar
      - Saltar página en impresoras
      - Posicionar el brazo de un disco
      - ...



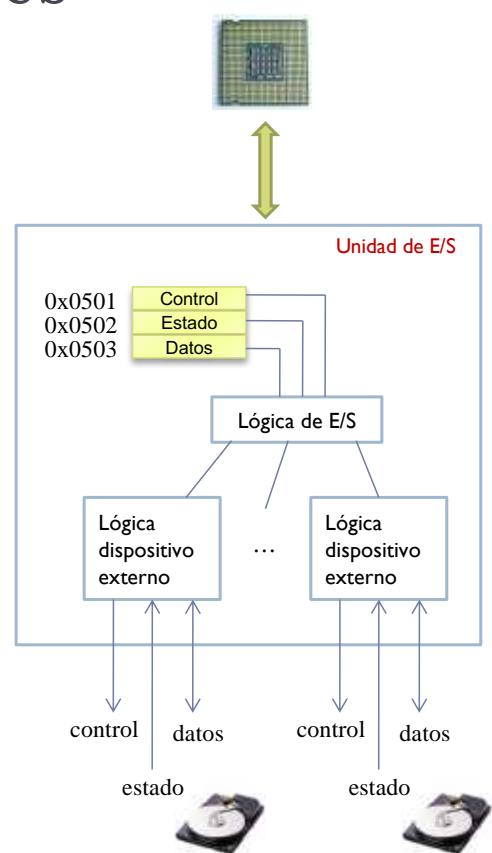
# Módulo de E/S: funciones

## ▶ Atender al procesador:

- ▶ Decodificación de órdenes
- ▶ Información de estado
- ▶ Control y temporización
  - ▶ Ej.: datos a M.P.

## ▶ Controlar periférico(s):

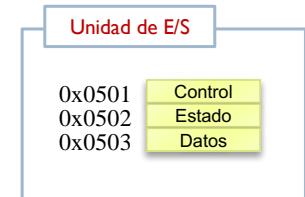
- ▶ Comunicación con dispositivos
- ▶ Detección de errores
- ▶ Almacenamiento temporal de datos
  - ▶ periférico->procesador



# Módulo de E/S: tipos

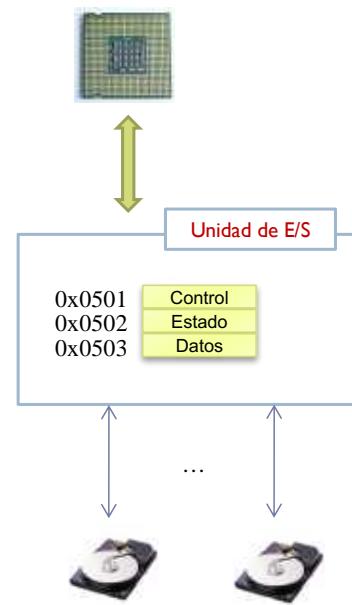
## ▶ Tipos de módulos por complejidad:

- ▶ **Canal de E/S o procesador de E/S:**  
se encarga de la mayoría de los detalles del procesamiento.
- ▶ **Controlador de E/S o controlador de dispositivo:**  
módulo más simple, que requiere que el procesador tenga un control detallado del dispositivo.



# Módulo de E/S: características

- ▶ Características fundamentales:
  - ▶ Unidad de transferencia
  - ▶ Direccionamiento
  - ▶ Técnicas de Entrada/Salida



# Características (1 / 3)

## ▶ Unidad de transferencia:

### ▶ **Dispositivos de bloque:**

- ▶ Unidad: **bloque** de bytes
- ▶ Acceso **secuencial** o **directo** a bloques
- ▶ Operaciones: leer, escribir, situarse, ...
- ▶ Ejemplos: discos y “cintas”

### ▶ **Dispositivos de carácter:**

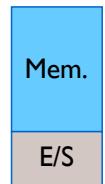
- ▶ Unidad: **caracteres** (ASCII, Unicode, etc)
- ▶ Acceso **secuencial** a caracteres
- ▶ Operaciones: get, put, ....
- ▶ Ejemplo: terminales, impresoras, tarjetas de red

## Características (2/3)

### ▶ Direccionamiento de E/S:

#### ▶ **Espacio de memoria conjunto**

- ▶ Los registros del ‘controlador’ se proyectan en memoria y usando un conjunto de direcciones de memoria se acceden a dichos registros.
- ▶ Ej: `sw $a0 etiqueta_discoA`

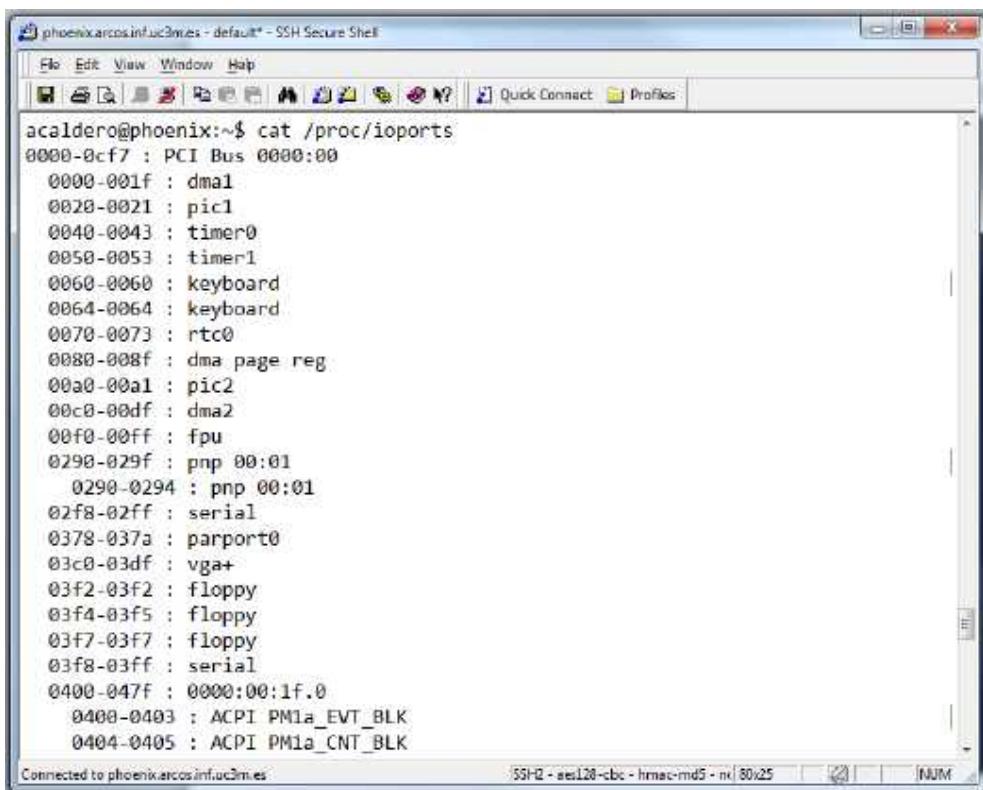


#### ▶ **Espacio de memoria separado (puertos):**

- ▶ Con instrucciones ensamblador especiales (In/Out) se acceden a las direcciones de E/S (denominadas puertos) que representan los registros del ‘controlador’.



# Direccionamiento Linux



A screenshot of a Windows terminal window titled "phoenix.arcos.inf.uc3m.es - default - SSH Secure Shell". The window shows the command "cat /proc/ioports" being run, displaying a list of I/O port ranges and their assigned devices. The output is as follows:

```
acaldero@phoenix:~$ cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
    0000-001f : dma1
    0020-0021 : pic1
    0040-0043 : timer0
    0050-0053 : timer1
    0060-0060 : keyboard
    0064-0064 : keyboard
    0070-0073 : rtc0
    0080-008f : dma page reg
    00a0-00a1 : pic2
    00c0-00df : dma2
    00f0-00ff : fpu
    0290-029f : pnp 00:01
        0290-0294 : pnp 00:01
    02f8-02ff : serial
    0378-037a : parport0
    03c0-03df : vga+
    03f2-03f2 : floppy
    03f4-03f5 : floppy
    03f7-03f7 : floppy
    03f8-03ff : serial
    0400-047f : 0000:00:1f.0
        0400-0403 : ACPI PM1a_EVT_BLK
        0404-0405 : ACPI PM1a_CNT_BLK
```

The bottom status bar indicates the connection is "Connected to phoenix.arcos.inf.uc3m.es" via "SSH2 - aes128-cbc - hmac-md5 - mt 80x25".

# Direccionamiento Windows

Información del sistema			
Resumen del sistema	Recurso:	Dispositivo	Estado:
Recursos de hardware	0x00000000-0x0000000F	Controladora de acceso directo a memoria	OK
Conflictos/uso compartido	0x00000000-0x0000000F	Bus PCI	OK
DMA	0x00000010-0x0000001F	Recursos de la placa base	OK
Hardware forzado	0x00000020-0x000000...	Controladora programable de interrupciones	OK
IRQs	0x00000022-0x0000003F	Recursos de la placa base	OK
Memoria	0x00000040-0x000000...	Cronómetro del sistema	OK
Componentes	0x00000044-0x0000005F	Recursos de la placa base	OK
Entorno de software	0x00000060-0x000000...	Recursos de la placa base	OK
	0x00000061-0x000000...	Altavoz del sistema	OK
	0x00000062-0x000000...	Recursos de la placa base	OK
	0x00000064-0x000000...	Recursos de la placa base	OK
	0x00000065-0x0000006F	Recursos de la placa base	OK
	0x00000070-0x000000...	Sistema CMOS/reloj en tiempo real	OK
	0x00000072-0x0000007F	Recursos de la placa base	OK
	0x00000080-0x000000...	Recursos de la placa base	OK
	0x00000081-0x000000...	Controladora de acceso directo a memoria	OK
	0x00000084-0x000000...	Recursos de la placa base	OK
	0x00000087-0x000000...	Controladora de acceso directo a memoria	OK
	0x00000088-0x000000...	Recursos de la placa base	OK
	0x00000089-0x000000...	Controladora de acceso directo a memoria	OK
	0x0000008C-0x000000...	Recursos de la placa base	OK

## Características (3/3)

- ▶ **Técnicas de E/S:** Interacción Procesador-Controlador
  - ▶ **E/S programada**
  - ▶ **E/S por interrupciones**
  - ▶ **E/S por DMA (acceso directo a memoria)**

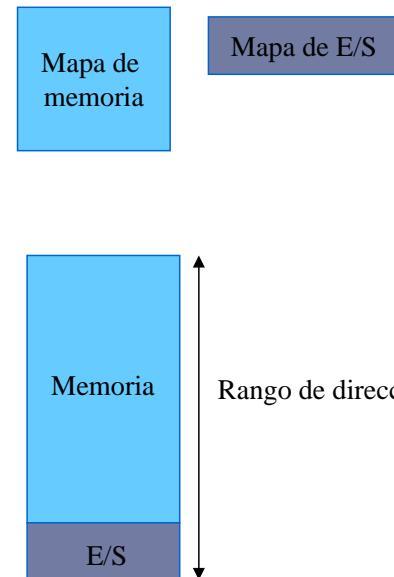
# E/S programada

- ▶ Toda la transferencia entre el procesador (o memoria) y la unidad de E/S se realiza bajo el control del procesador mediante la ejecución de **instrucciones máquina de E/S**
- ▶ Instrucciones de E/S:
  - ▶ Instrucciones máquina especiales (similares a LW y SW)
  - ▶ Son instrucciones privilegiadas
- ▶ Ejemplo de hipotética instrucción de E/S
  - ▶ **IN Reg, dirección**
    - ▶ Carga en el registro del procesador Reg el dato que se encuentra en el registro del módulo de E/S cuya dirección es dirección
  - ▶ **OUT Reg, dirección**
    - ▶ Realiza la escritura en el módulo de E/S

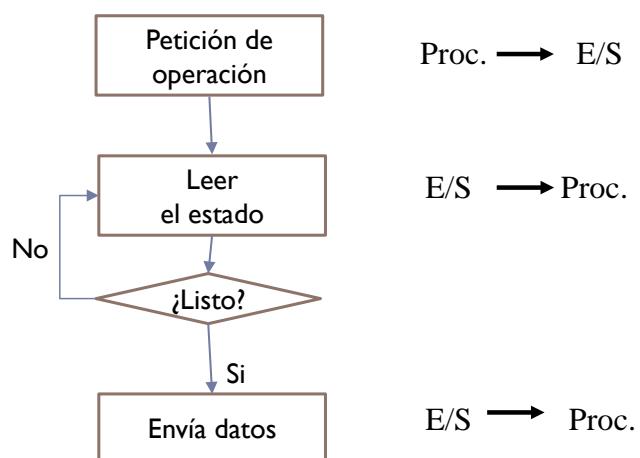
# Mapa de E/S

- ▶ Conjunto de todas las direcciones de E/S
  - ▶ Con  $p$  bits,  $2^p$  direcciones posibles
- ▶ Tipos:
  - ▶ **Mapa de E/S separado**
    - ▶ Incluye instrucciones especiales de E/S (IN, OUT)

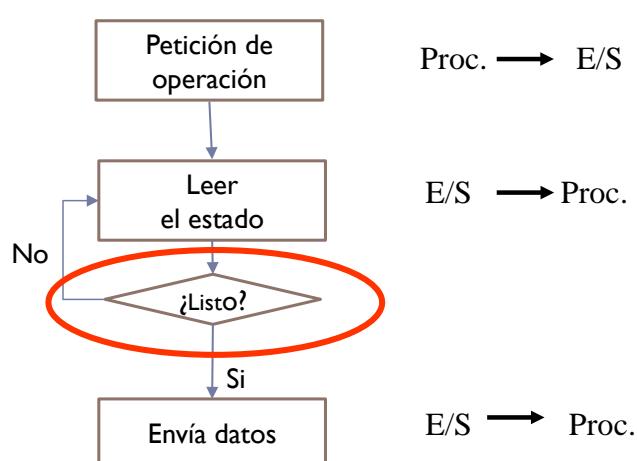
- ▶ **Mapa común**
  - ▶ El acceso a los módulos de E/S se realiza con las mismas instrucciones que se utilizan para acceder a memoria (LOAD, STORE)



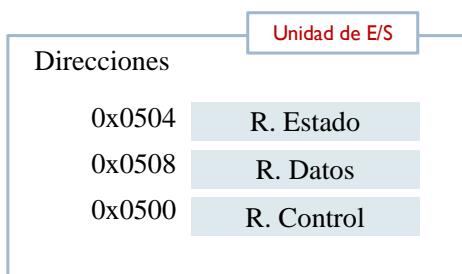
## Interacción mediante E/S programada



## Interacción mediante E/S programada

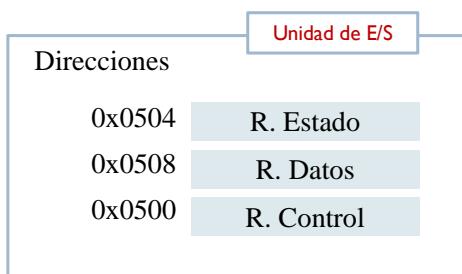


# Ejemplo



- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw del MIPS

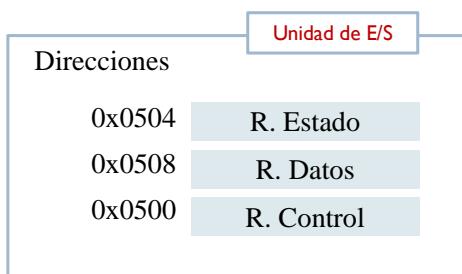
# Ejemplo



- ▶ ¿Instrucciones para escribir un **I** en el registro 0x0508 (de datos)?

- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones **lw** y **sw** del MIPS

# Ejemplo

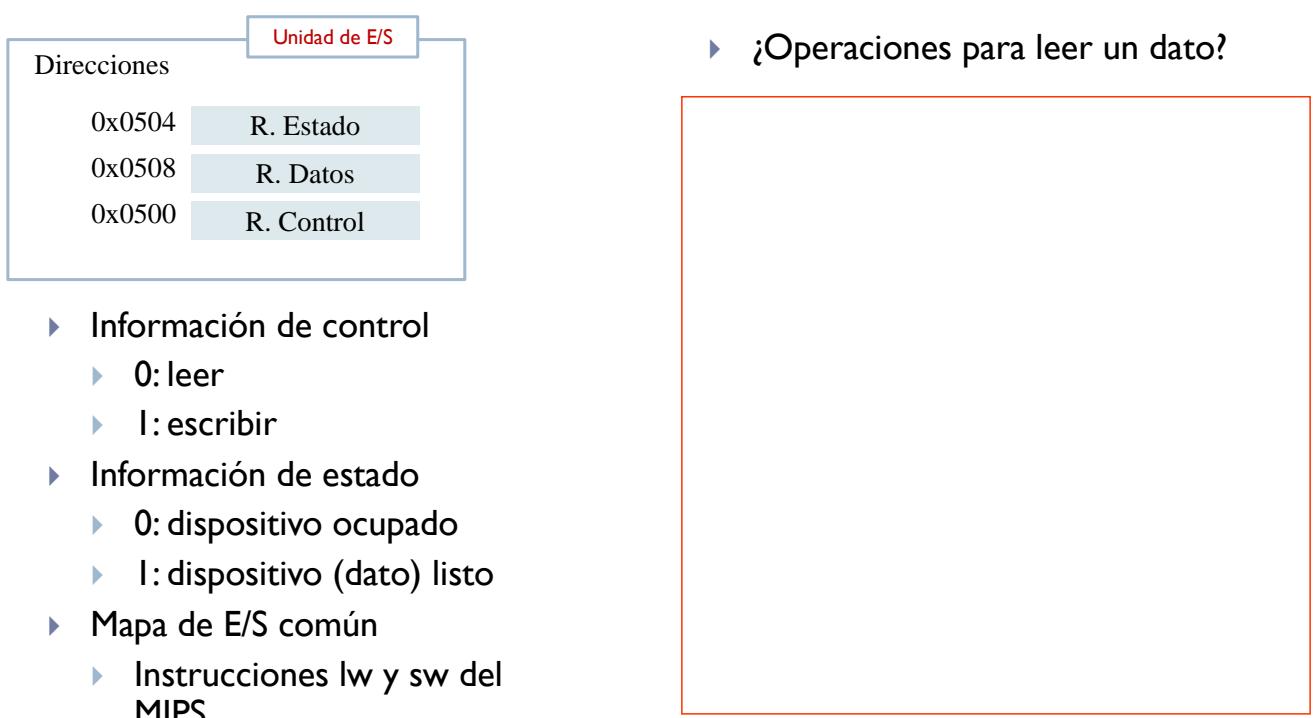


- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw del MIPS

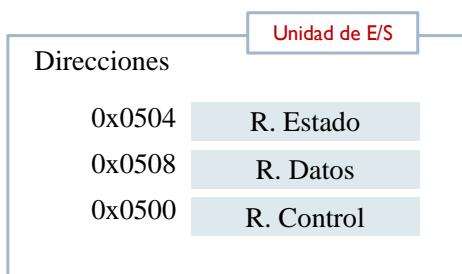
- ▶ ¿Instrucciones para escribir un 1 en el registro 0x0508 (de datos)?

```
li $t0, 1  
sw $t0, 0x0508
```

# Ejemplo



# Ejemplo



- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw del MIPS

## ▶ ¿Operaciones para leer un dato?

1. Enviar la orden

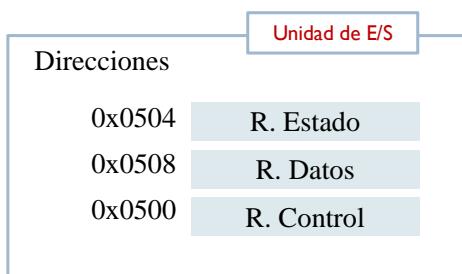
```
li $t0, 0
sw $t0, 0x0500
```
2. Leer el estado  
bucle: 

```
lw $t0, 0x0504
```
3. Comprobar el estado  

```
beqz $t0, bucle
```
4. Leer el dato  

```
lw $t0, 0x0508
```

# Ejemplo



- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw del MIPS

## ▶ ¿Operaciones para escribir un dato?

1. Enviar el dato

```
li $t0, 123  
sw $t0, 0x0508
```

2. Enviar la orden

```
li $t0, 1  
sw $t0, 0x0500
```

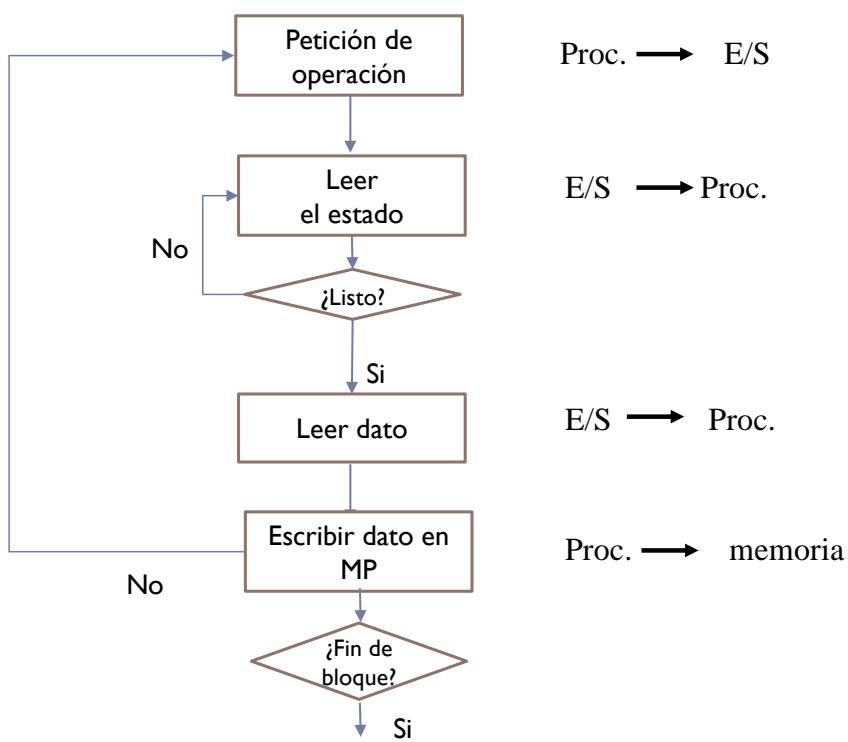
3. leer estado

```
bucle: lw $t0, 0x0504
```

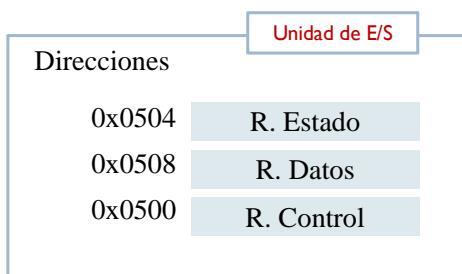
4. comprobar estado

```
beqz $t0, bucle
```

# Lectura de un bloque de datos



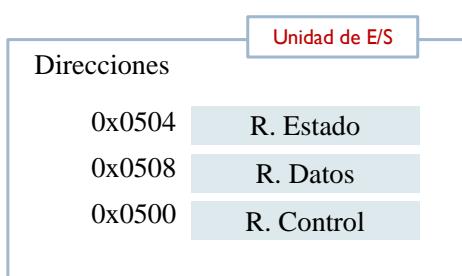
# Ejercicio



- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw

Codifique un programa en ensamblador que lee 100 datos usando la unidad de E/S descrita, y los almacena en la dirección de memoria principal dada por la etiqueta ‘datos’.

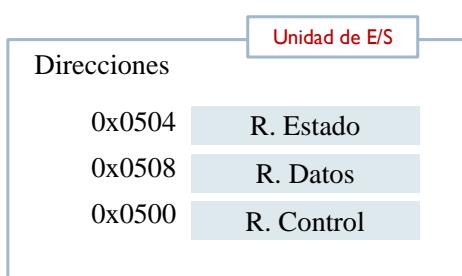
# Ejercicio (solución)



- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw

```
.data
    datos: .space 400
.text
.globl main
main:    li $t3 0
bucle1:  li $t0 0
          sw $t0 0x500
bucle2:  lw $t1 0x504
          beqz $t1 bucle2
          lw $t2 0x508
          sw $t2 datos($t3)
          add $t3 $t3 4
          bne $t3 400 bucle1
```

# Ejercicio (solución)

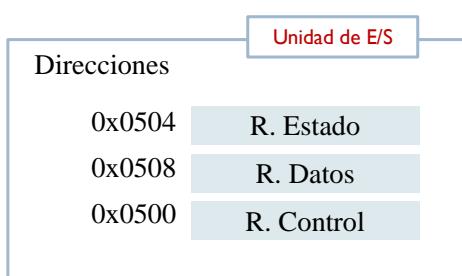


- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw

```
.data
    datos: .space 400
.text
.globl main
main:    li $t3 0
        bucle1: li $t0 0
                  sw $t0 0x500
        bucle2: lw $t1 0x504
                  beqz $t1 bucle2
                  lw $t2 0x508
                  sw $t2 datos($t3)
                  add $t3 $t3 4
                  bne $t3 400 bucle1
```

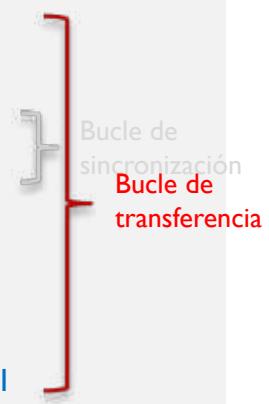
} Bucle de sincronización

# Ejercicio (solución)



- ▶ Información de control
  - ▶ 0: leer
  - ▶ 1: escribir
- ▶ Información de estado
  - ▶ 0: dispositivo ocupado
  - ▶ 1: dispositivo (dato) listo
- ▶ Mapa de E/S común
  - ▶ Instrucciones lw y sw

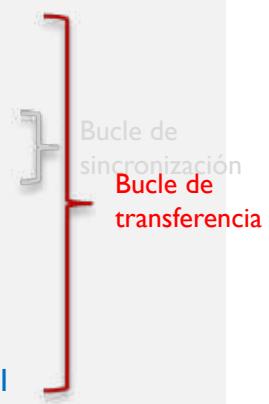
```
.data
    datos: .space 400
.text
.globl main
main:    li $t3 0
        bucle1: li $t0 0
                  sw $t0 0x500
        bucle2: lw $t1 0x504
                  beqz $t1 bucle2
                  lw $t2 0x508
                  sw $t2 datos($t3)
                  add $t3 $t3 4
                  bne $t3 400 bucle1
```



# Ejercicio

- ▶ Sea un computador con la capacidad de ejecutar 200 millones de instrucciones por segundo (200 MIPS)
- ▶ Se conecta el módulo de E/S anteriormente descrito siendo el tiempo medio de espera de lectura de 5 ms
- ▶ Calcule cuantas instrucciones se ejecutan en el bucle de sincronización y en el bucle de transferencia para el programa mostrado

```
.data  
    datos: .space 400  
.text  
.globl main  
main:      li $t3 0  
bucle1:   li $t0 0  
           sw $t0 0x500  
bucle2:   lw $t1 0x504  
           beqz $t1 bucle2  
           lw $t2 0x508  
           sw $t2 datos($t3)  
           add $t3 $t3 4  
           bne $t3 400 bucle1
```



Bucle de sincronización  
Bucle de transferencia

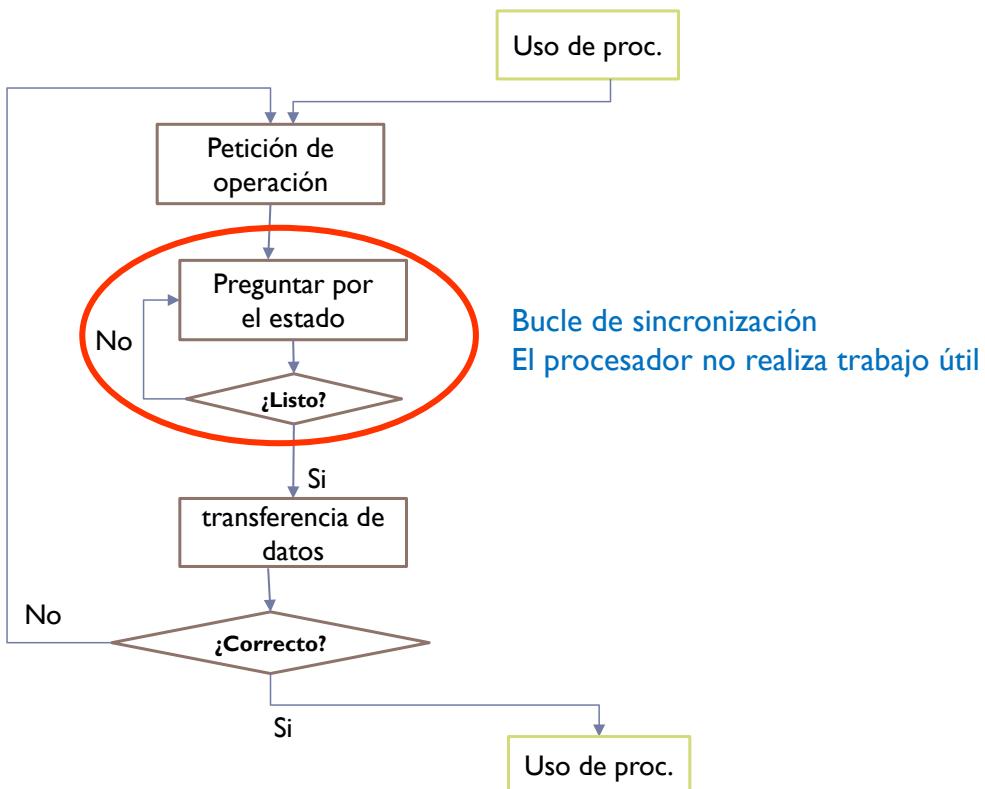
# Ejercicio (solución)

- ▶ Bucle de sincronización:
  - ▶ En media dura 5 ms
  - ▶ Se ejecuta 200 MIPS en media
  - ▶  $I_{bs} = 200*10^6 * 5*10^{-3} = 10^6$
- ▶ Bucle de transferencia:
  - ▶  $I_{(li \$t3 0)} + 6 * 100 + 10^6 (I_{bs})$
- ▶ Como puede comprobarse, en el bucle se ejecuta 1.000.601 instrucciones, de las cuales 1.000.000 corresponden al bucle de espera (el 99,9%)
  - ▶ Es un desperdicio de ciclos del procesador
  - ▶ El procesador no realiza trabajo útil

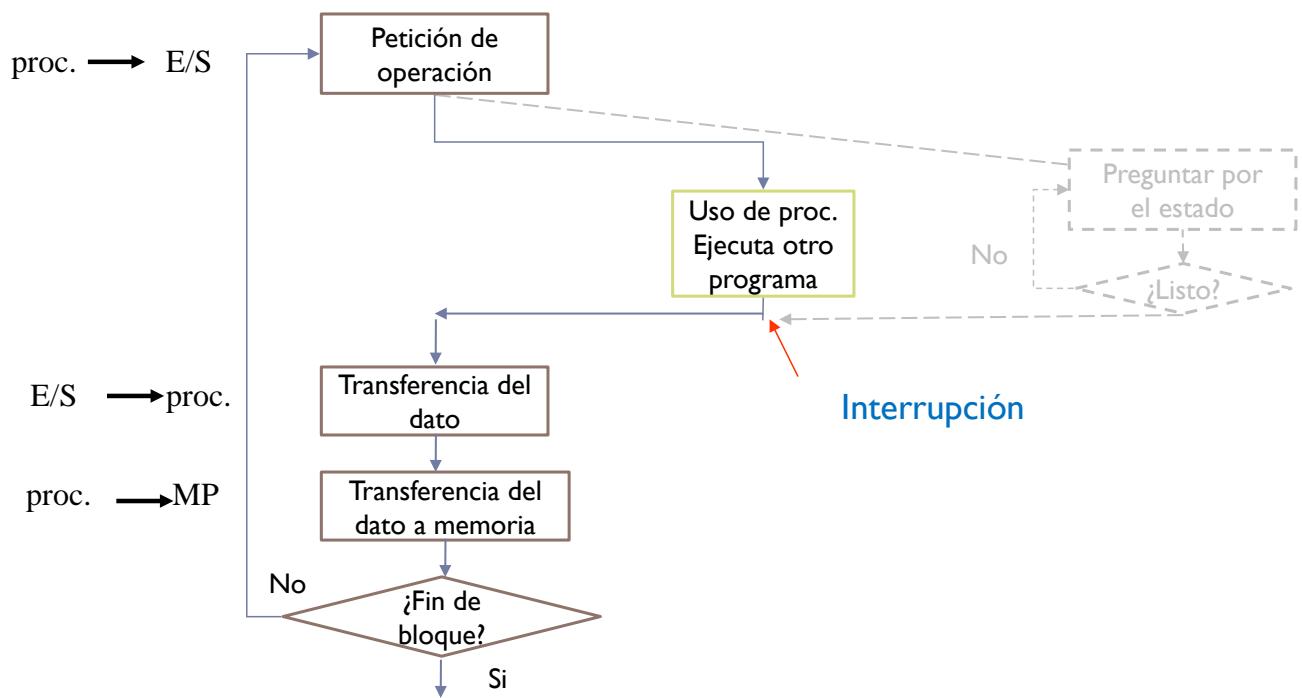
```
.data
    datos: .space 400
.text
.globl main
main:    li $t3 0
bucle1:  li $t0 0
          sw $t0 0x500
bucle2:  lw $t1 0x504
          beqz $t1 bucle2
          lw $t2 0x508
          sw $t2 datos($t3)
          add $t3 $t3 4
          bne $t3 400 bucle1
```



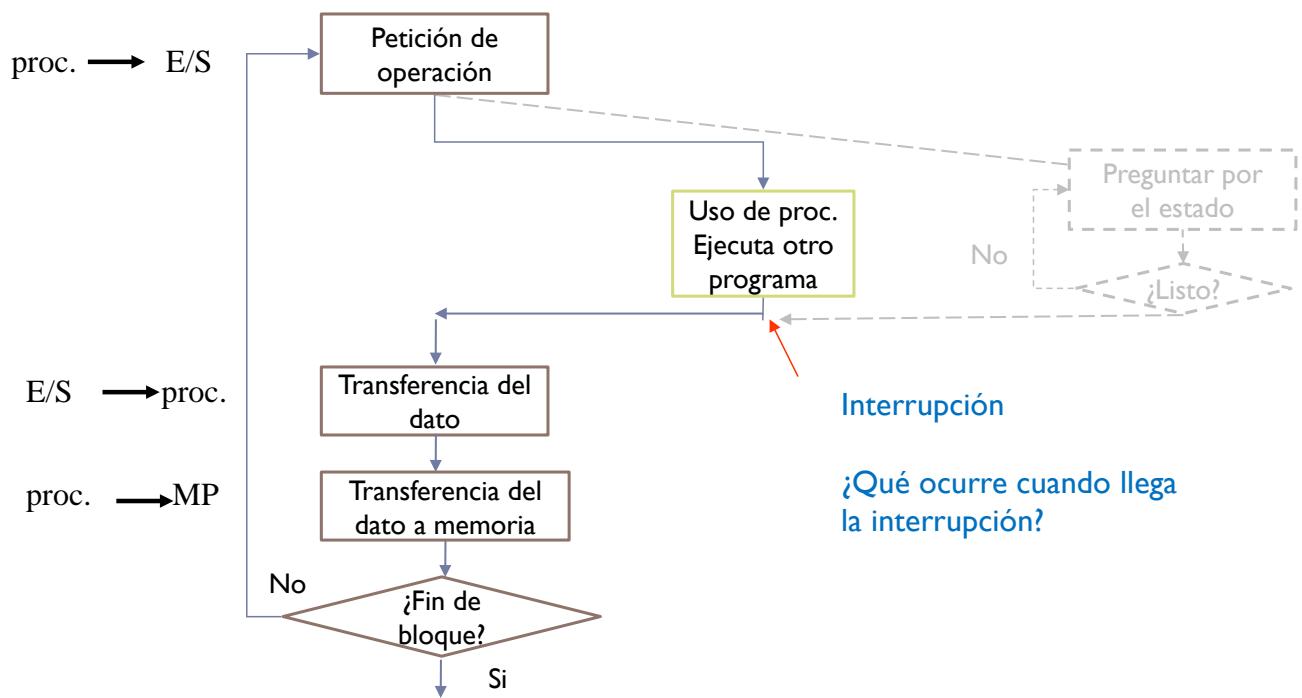
## Problema de la E/S programada



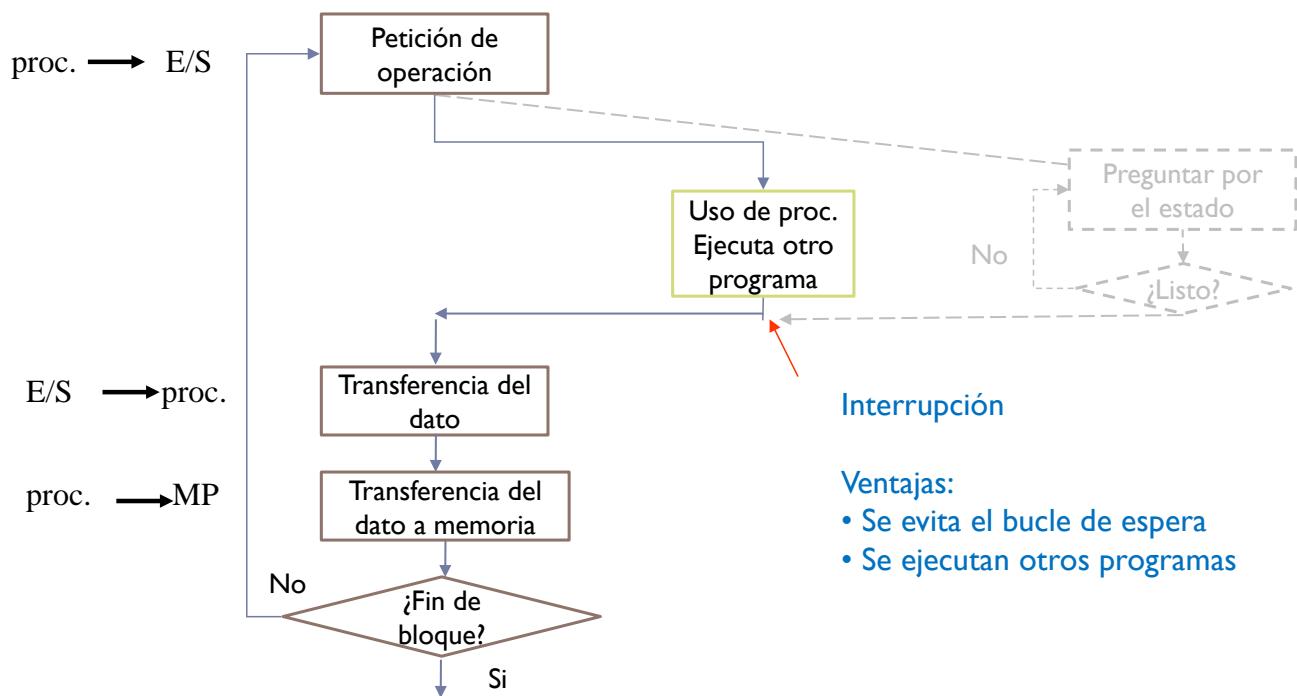
## E/S mediante interrupciones



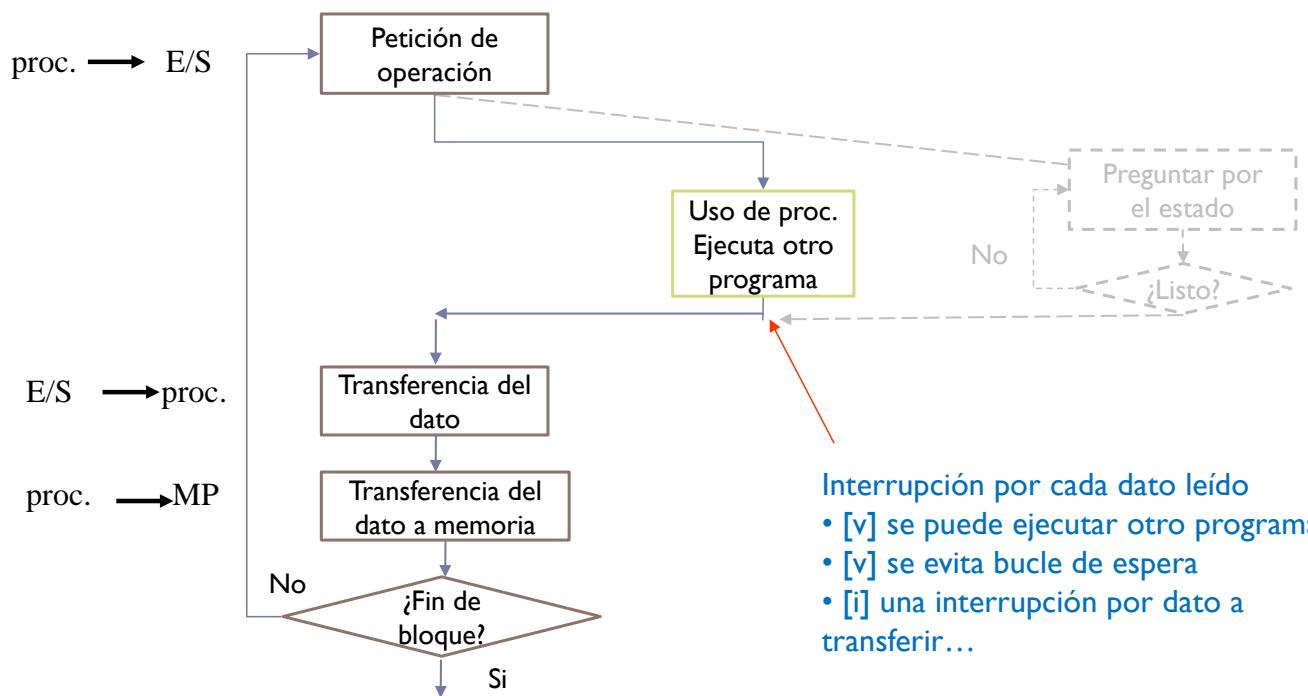
## E/S mediante interrupciones



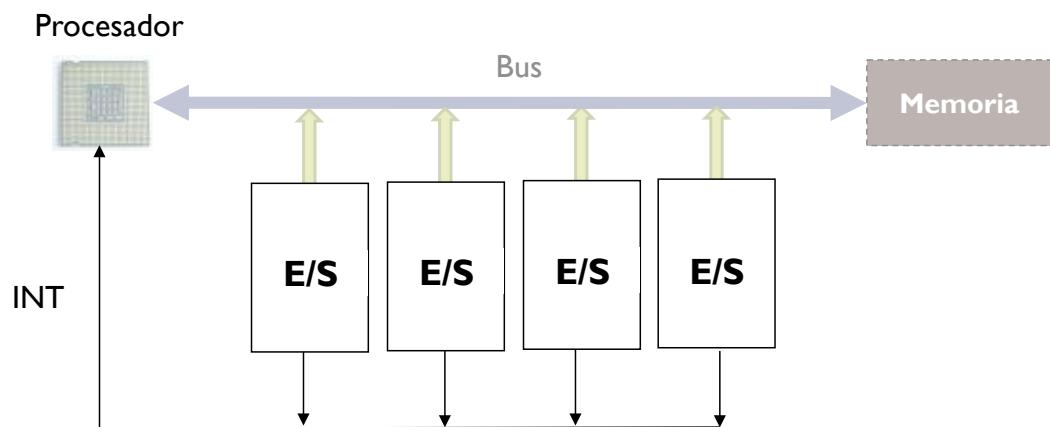
## E/S mediante interrupciones



# Interacción mediante interrupciones

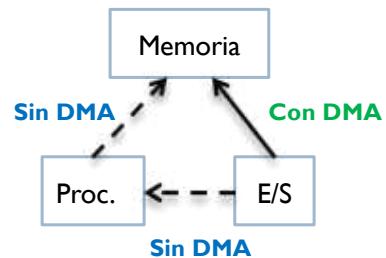


## E/S mediante interrupciones

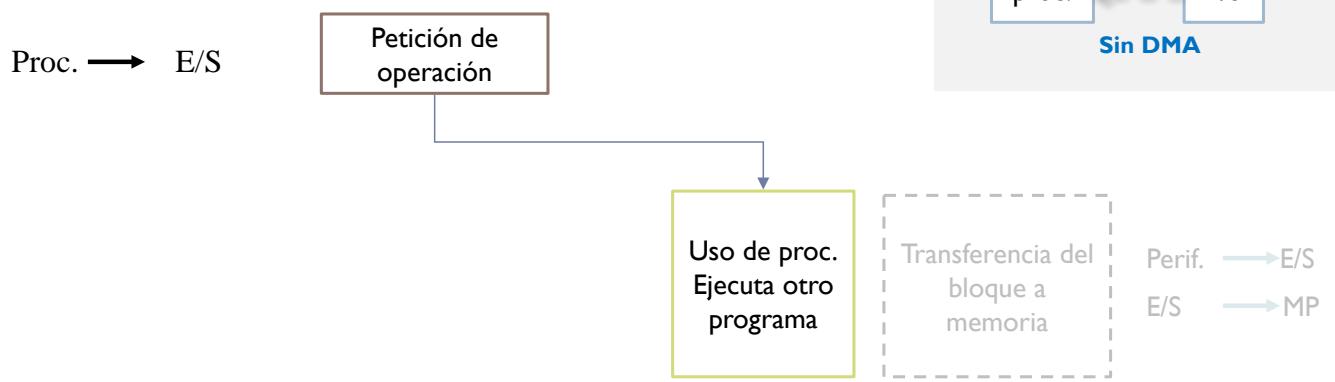


## E/S mediante DMA

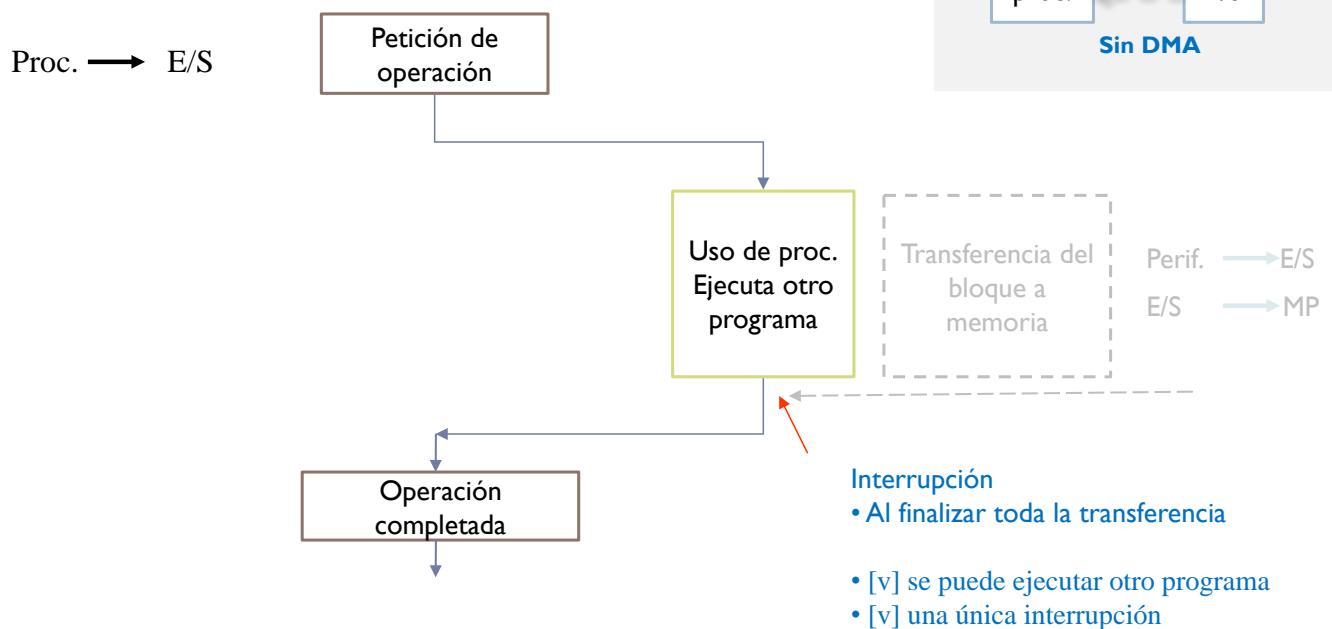
- ▶ DMA (Direct Memory Access): Acceso directo a memoria
- ▶ El procesador no realiza la transferencia entre la unidad de E/S y la memoria
  - ▶ Con interrupciones se evita el bucle de espera pero la transferencia la lleva a cabo el procesador
  - ▶ Para un bloque de  $N$  bytes, se generan  $N$  interrupciones
- ▶ Con DMA toda la transferencia la realiza la unidad de E/S
  - ▶ Solo una interrupción al final



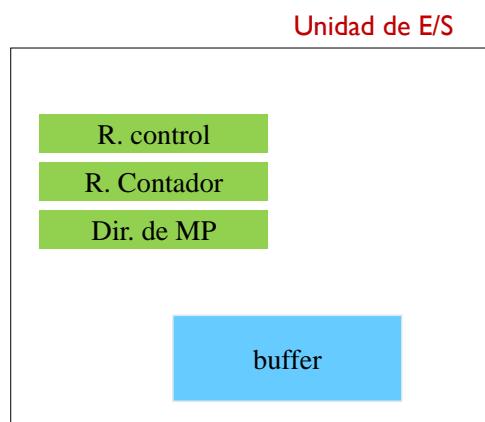
## Transferencia de un bloque mediante DMA



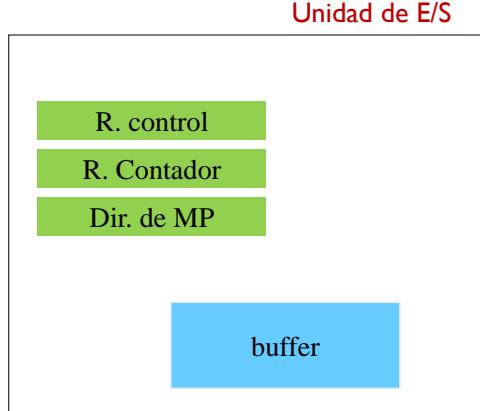
# Transferencia de un bloque mediante DMA



# Estructura simplificada de un módulo de E/S para DMA

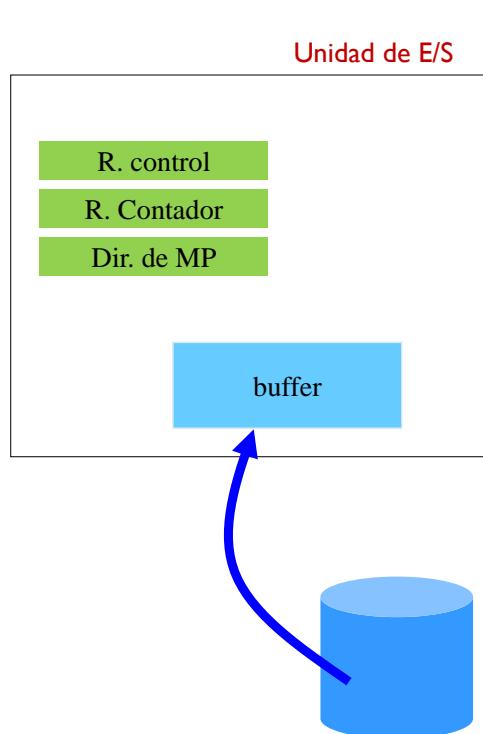


# Transferencia con DMA



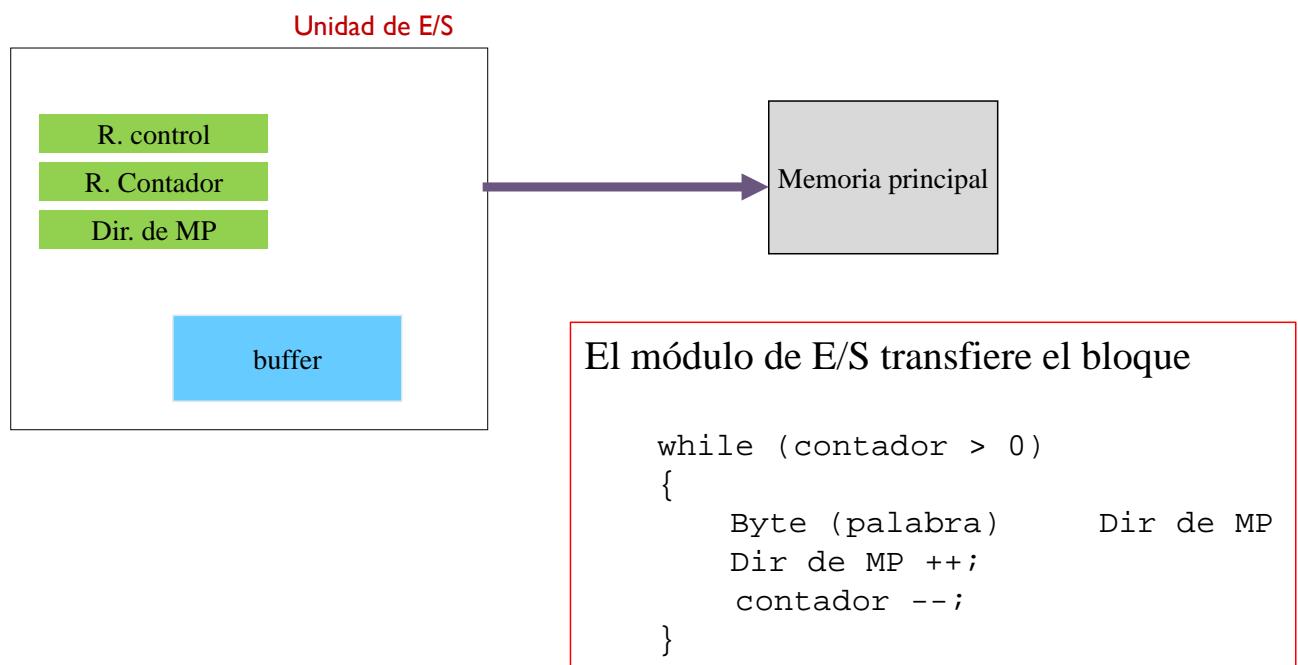
- ▶ El procesador escribe en los registros (con instrucciones de E/S)
  - ▶ La operación (**R. control**)
    - ▶ Lectura, escritura
  - ▶ El número de bytes a transferir (**R. contador**)
  - ▶ La **dirección de memoria principal** donde
    - ▶ Se almacenan los datos (escritura al periférico)
    - ▶ Almacenar los datos (lectura del periférico)

# Transferencia con DMA

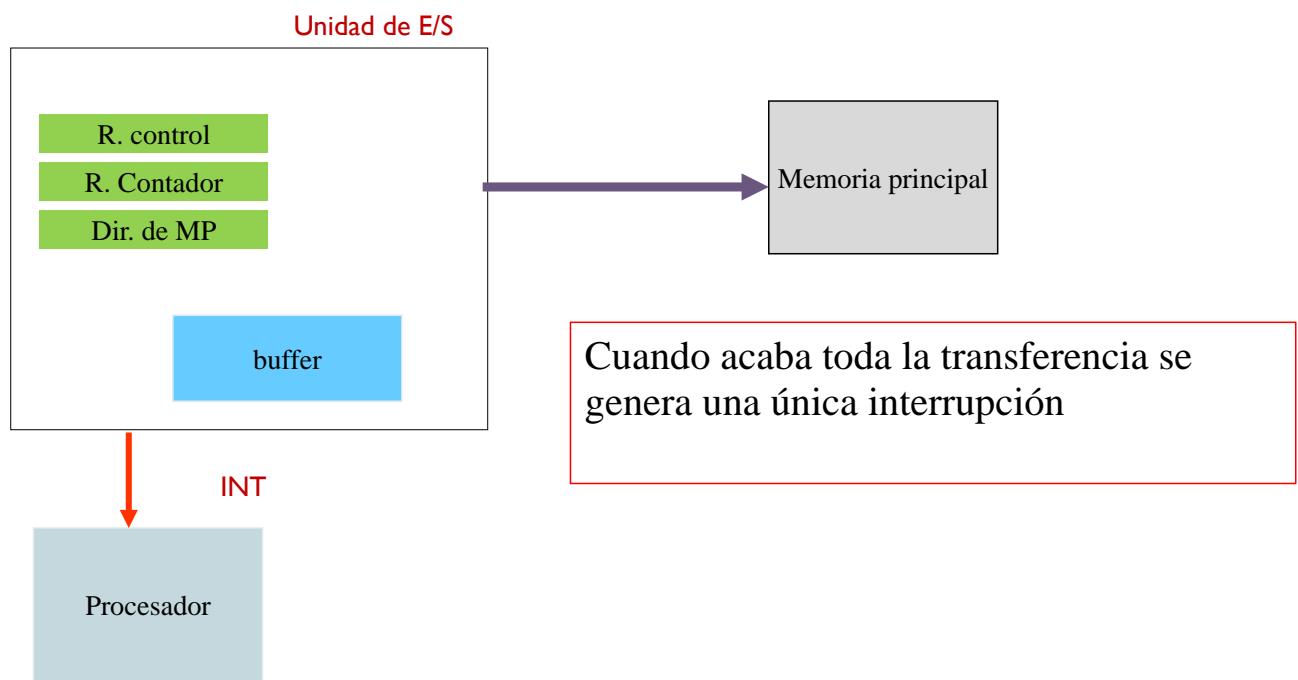


- ▶ La unidad de E/S transfiere todo el bloque de datos del periférico al buffer interno de la unidad de E/S (para lectura)

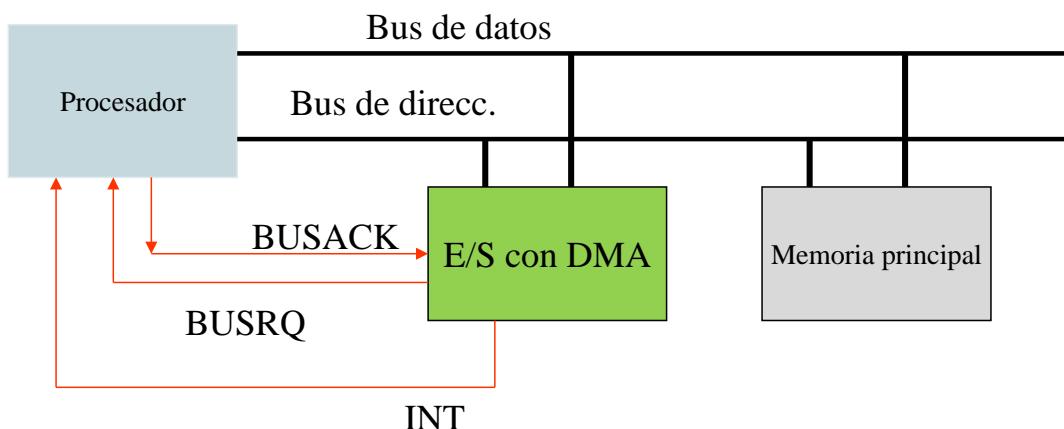
# Transferencia con DMA



# Transferencia con DMA



## Acceso del módulo de E/S a MP



- ▶ Hay que coordinar el acceso a memoria entre el procesador y el módulo de E/S

## Acceso del módulo de E/S a MP

### Robo de ciclo

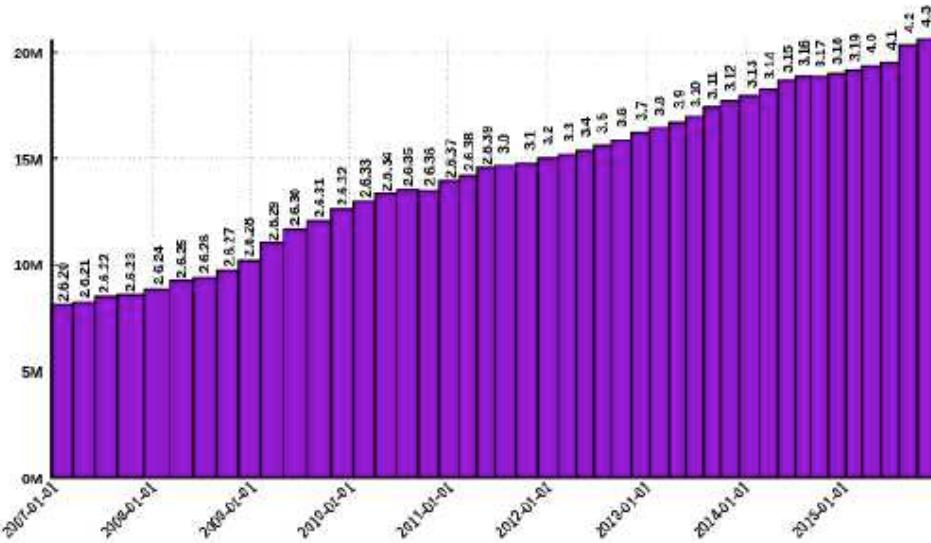


- ▶ Cuando la unidad de E/S está lista para transferir un dato:
  - ▶ Activa la señal **BUSRQ** para solicitar el acceso a los buses
  - ▶ Al final de cada fase de una instrucción el procesador comprueba la señal **BUSRQ**. Si está activa libera los buses y activa **BUSACK**
  - ▶ La unidad de E/S accede a MP y a continuación desactiva **BUSRQ**
  - ▶ El procesador continúa
  - ▶ La interrupción se genera al final

# Curiosidades: Importancia de los controladores

- ▶ **Estadísticas del kernel de Linux (2007-2008):**
  - ▶ 9,2 millones de líneas de código.
  - ▶ Se incrementa un 10% cada año:
    - ▶ En media cada día se:
      - Añaden 4.500 líneas,
      - Borran 1.800 líneas
      - Modifican 1.500 líneas
  - ▶ La mayor parte del código es de los drivers:
    - ▶ **El 55% del código son los controladores de dispositivo (o drivers)**
      - Software parte del sistema operativo que la CPU ejecuta para trabajar con el dispositivo asociado
    - ▶ El núcleo del kernel ocupa un 5% y el resto (40%) se reparte entre soporte para las distintas arquitectura, el código de red, etc.

# Importancia de los controladores Kernel de Linux



Líneas de  
código del  
kernel de  
Linux

- ▶ El 70% del código de linux son los controladores de dispositivo (o drivers)

## Ejercicio 1

- ▶ Sea un disco con 600 sectores por pista, con una velocidad de rotación de 7200 rpm y un tiempo medio de búsqueda de 2ms. Calcule el tiempo medio de acceso a un sector.

## Ejercicio 2

- ▶ Se desea desarrollar un controlador para un semáforo. El controlador dispone de una CPU de 32 bits, mapa de E/S común y juego de instrucciones del MIPS 32. A esta CPU se le conectan dos módulos de E/S. El primero es un cronómetro y el segundo es el módulo de E/S que controla el funcionamiento del semáforo. El módulo cronómetro dispone de los tres registros siguientes:
  - ▶ Registro con dirección 1000. En este registro se carga el valor correspondiente a la cuenta atrás en segundos.
  - ▶ Registro con dirección 1004. En este registro se carga un 1 cuando se quiere comenzar la cuenta atrás.
  - ▶ Registro con dirección 1008. Cuando la cuenta atrás llega a 0, en este registro se carga un 1. Mientras se está realizando la cuenta atrás el valor de este registro es 0.
- ▶ El módulo de E/S que controla el semáforo dispone de tres registros:
  - ▶ Registro con dirección 1012. En este registro se codifica el valor correspondiente al color del semáforo: 100 para el rojo, 010 para el amarillo y 001 para el verde.
- ▶ Se pide:
  - ▶ Escriba el programa ensamblador que controla el funcionamiento de este semáforo. El semáforo siempre comienza su funcionamiento en rojo. La duración del semáforo en rojo y verde es de 90 segundos y en amarillo de 20 segundos.

## Ejercicio 3

- ▶ Un computador tiene conectado un ratón que debe consultarse al menos 30 veces por segundo para poder actualizar su posición en la pantalla. La rutina que consulta su posición y vuelve a dibujar el puntero en la pantalla requiere 2000 ciclos para su ejecución. Si el computador tiene una frecuencia de 2.7 GHz, ¿qué sobrecarga supone la mencionada rutina de tratamiento de interrupciones, es decir, qué porcentaje de tiempo dedica el computador a ejecutar esta rutina.

## Ejercicio 4

- ▶ Se dispone de un computador que tiene conectados un sensor que mide la temperatura de un horno y una alarma. El sensor de temperatura se conecta al computador a través de un módulo de E/S que dispone de los tres siguientes registros:
  - ▶ **Registro de control** (dirección: ST\_REG\_CONTROL). Este registro se utiliza para indicar la operación a realizar sobre el sensor. Se pueden indicar dos operaciones:
    - ▶ Inicialización del dispositivo. El dispositivo se inicializa mediante el valor 0
    - ▶ Lectura de la temperatura. Esta operación se indica con el valor 1.
  - ▶ **Registro de datos** (dirección: ST\_REG\_DATOS). En este registro se almacena el valor de la temperatura tomado por el sensor.
  - ▶ **Registro de estado** (dirección: ST\_REG\_ESTADO). Este registro puede almacenar dos valores:
    - ▶ Listo (valor 1): cuando el dispositivo se ha inicializado o se dispone de una medida de temperatura válida.
    - ▶ Ocupado (valor 0): cuando el dispositivo está inicializándose o realizando una toma de medida.

## Ejercicio 4 (cont)

- ▶ Por su parte, la alarma se conecta a un módulo de E/S que dispone de dos registros:
  - ▶ **Registro de control** (dirección:A\_REG\_CONTROL). Este registro se utiliza para activar o desactivar la alarma. Para activar la alarma se escribe en este registro el valor 1. Para desactivarla se escribe el valor 0.
  - ▶ **Registro de estado** (dirección:A\_REG\_ESTADO). Este registro permite conocer el estado de la alarma. Si el valor de este registro es 0, la alarma se encuentra desactivada. Si el valor almacenado es 1, la alarma está activada.
- ▶ El computador dispone de un mapa de E/S separado y dos instrucciones de E/S:
  - ▶ in RegProcesador, RegE/S que carga en el registro del procesador RegProcesador el dato almacenado en el registro de un módulo de E/S con dirección RegE/S.
  - ▶ out RegProcesador, RegE/S que carga en el registro del módulo de E/S con dirección regE/S el dato almacenado en el registro del procesador RegProcesador.
- ▶ Todos los registros de los módulos de E/S son registros de 32 bits. Escriba utilizando el ensamblador del MIPS un programa que lea continuamente la temperatura del horno. Si la temperatura supera los 100° C, se debe activar la alarma y dejarla activada mientras la temperatura se encuentre por encima de los 100° C. Cuando la temperatura caiga por debajo de los 100° se debe desactivar la alarma.

# Ejercicio 5

- ▶ Sea un computador de 32 bits con direccionamiento de bytes y mapas de direcciones separados para memoria y entrada/salida. Su arquitectura ofrece el juego de instrucciones del MIPS R2000 más las instrucciones in y out, que permiten leer y, respectivamente, escribir en los registros de los módulos de E/S:
  - ▶ in rdest, dirección
  - ▶ out rsrcc, dirección
- ▶ A este computador se encuentra conectado un sensor de nivel de aguas Px con, entre otras, las siguientes características:
  - ▶ Su módulo o unidad de E/S dispone de los siguientes registros de 32 bits:
    - ▶ Reg. de Control (R\_CONTROL\_Px)
    - ▶ Reg. de Estado (R\_ESTADO\_Px)
    - ▶ Reg. de Datos (R\_DATOS\_Px)
  - ▶ Sólo dispone de dos mandatos o ‘comandos’, ON (activa el sensor) y OFF (desactiva el sensor).
  - ▶ El Reg. de Estado tiene tres valores posibles: MIDIENDO, NUEVO y ERROR.
  - ▶ El modo de operación del sensor es el siguiente: cuando se activa pone su estado inicialmente a MIDIENDO, a partir de ese momento devolverá (en su Reg. de Datos) una nueva medida (32 bits) del nivel del agua cada vez que detecte una variación en el nivel mayor de un determinado umbral. Cuando suministra un nueva medida en el Reg. de Estado aparece el valor NUEVO, y cuando detecta que se ha leído éste (del Reg. de Datos) cambia su valor a MIDIENDO, hasta la llegada de una nueva medida. Además es posible que se produzca alguna anomalía en el sensor, dejando de estar operativo, circunstancia que indica con el valor ERROR en el Reg. de Estado.

## Ejercicio 5 (cont.)

- ▶ Se pide:
  - ▶ Escribir un programa (*driver*) para manejar este periférico mediante *E/S Programada* con el siguiente funcionamiento:
    1. Se activa el sensor.
    2. Hasta que se complete un total de 100 medidas, cada vez que llega una nueva se deberá almacenar en la siguiente posición de la zona de Memoria asignada, que comienza en la dirección **M\_ALMACEN**.
    3. Si en cualquier momento se detecta que se ha producido un error de funcionamiento del periférico, se deberá interrumpir la lectura de las 100 medidas y escribir en la posición de Memoria **M\_CODIGO** el valor **PROBLEMAS**.
    4. Si se completan las 100 medidas se escribirá en la posición de Memoria **M\_CODIGO** el valor **TODO\_OK**. Indicar qué inconvenientes tiene el uso aquí de esta técnica de *E/S Programada* o *Directa* y cómo se podría utilizar en su lugar la de *E/S por Interrupciones*.
- ▶ Indique brevemente el beneficio que supondría su uso en este caso.

NOTA: todos los identificadores (nombres simbólicos) que aparecen en el enunciado tendrán un determinado valor (no especificado aquí) y corresponden siempre a datos de 32 bits.