Dept. de Teoría de la Señal y Comunicaciones y Sistemas Telemáticos y Computación
Área de Telemática (GSyC)

Instrucciones: El lenguaje del computador

Katia Leal Algara
katia.leal@urjc.es
http://gsyc.escet.urjc.es/~katia/

El lenguaje del computador

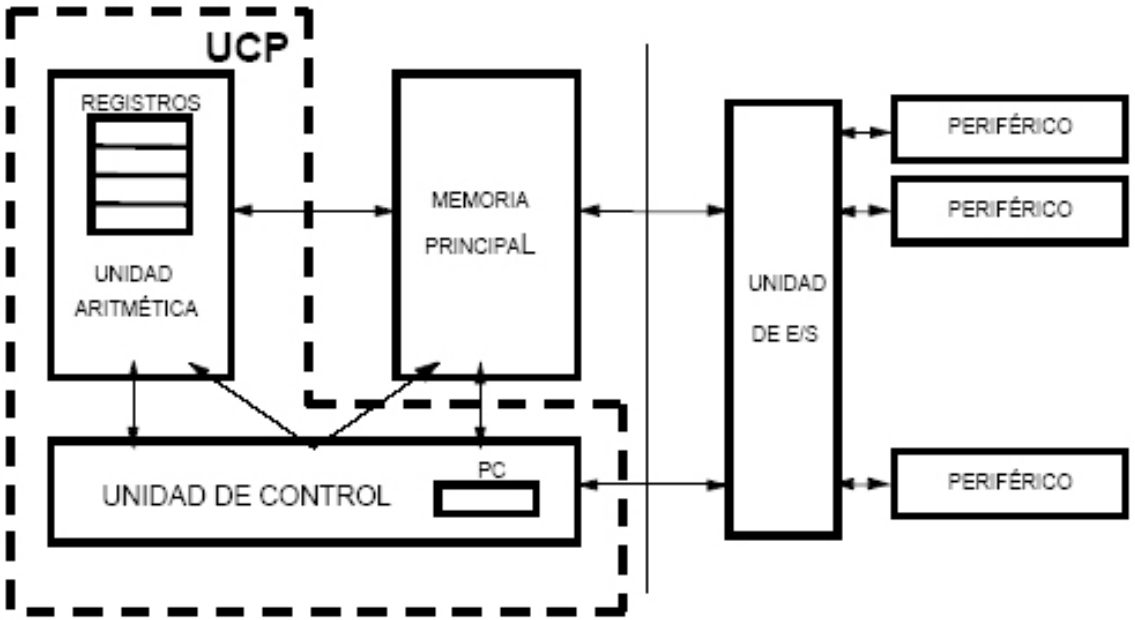
- **Instrucciones**: son las palabras que entiende el ordenador
- **Juego de instrucciones**: vocabulario de comandos entendido por una determinada arquitectura
- Los lenguajes que hablan los distintos computadores son muy similares
- El principal objetivo de un diseñador de computadores es: encontrar un leguaje cuyo hardware y compilador sean sencillos de construir y que al mismo tiempo maximice el rendimiento y minimice el coste y la potencia

Objetivos

- Concepto de **programa almacenado**: instrucciones y datos de diferentes tipos se pueden almacenar en memoria como números
- Escribir programas en el lenguaje del computador y ejecutarlos en un simulador
- Impacto de los lenguajes de programación y de la optimización del compilador en el rendimiento

Operaciones aritméticas

- Von Neumann explicó que una **ALU** es un requisito fundamental para una computadora porque está garantizado que ésta tendrá que efectuar operaciones matemáticas básicas, incluyendo adición, sustracción, multiplicación, y división



Unidad artimético-lógica

- Opera sobre los datos de una instrucción
- Tipo de operaciones: desplazamiento, lógicas y aritméticas
- En la mayoría de los casos, es un **simple sumador-restador**
- Entonces, ¿cómo pueden realizar tantas operaciones diferentes?
  - Descomposición en pasos elementales
  - Ejecutar rápidamente esos pasos

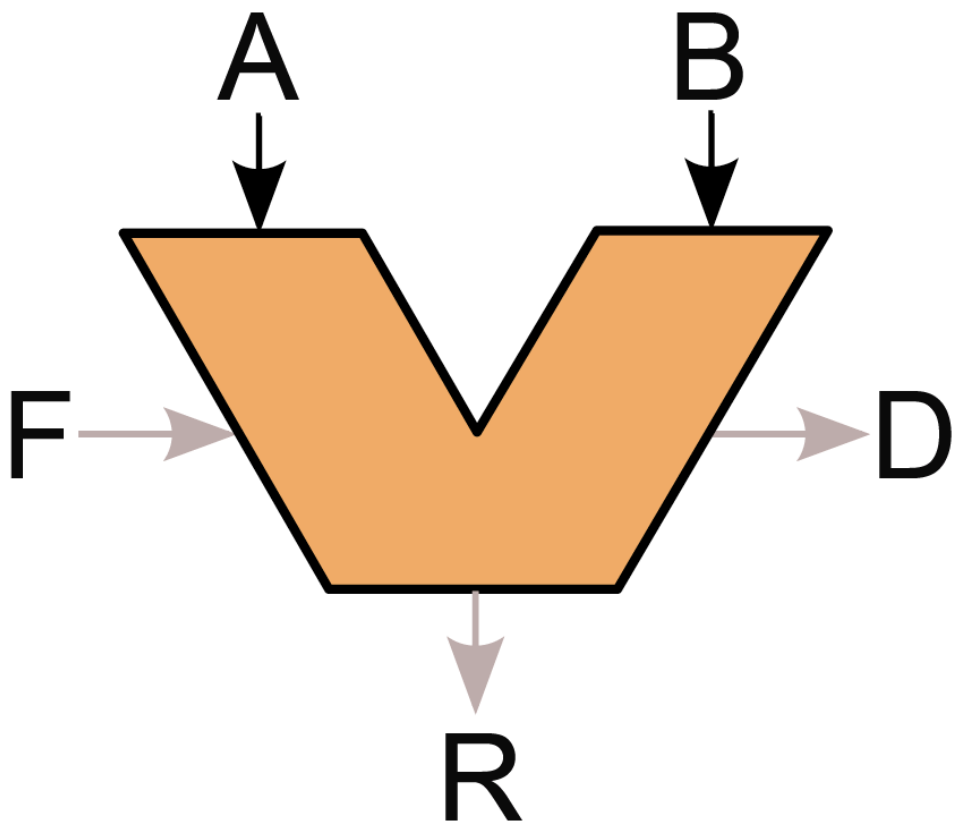
Sistemas numéricos

- Una ALU debe procesar números usando el mismo formato que el resto del circuito digital
- Las primeras computadoras usaron una amplia variedad de sistemas de numeración, incluyendo complemento a uno, signo-magnitud, e incluso verdaderos sistemas decimales
- Las ALUs para cada uno de estos sistemas numéricos mostraban diferentes diseños, y esto influyó en la **preferencia actual por el complemento a dos**, debido a que ésta es la representación más simple, para el circuito electrónico de la ALU, para calcular adiciones y sustracciones, etc

Tipos de operadores

- Circuito capaz de realizar:
  - **Operaciones aritméticas de números enteros**: sumas, restas, multiplicación y división
  - **Operaciones lógicas de bits**: AND, NOT, OR, XOR, NOR
  - **Operaciones de desplazamiento de bits**: desplazar o rotar, a izquierda o a derecha, con o sin extensión de signo

Aspecto de un operador

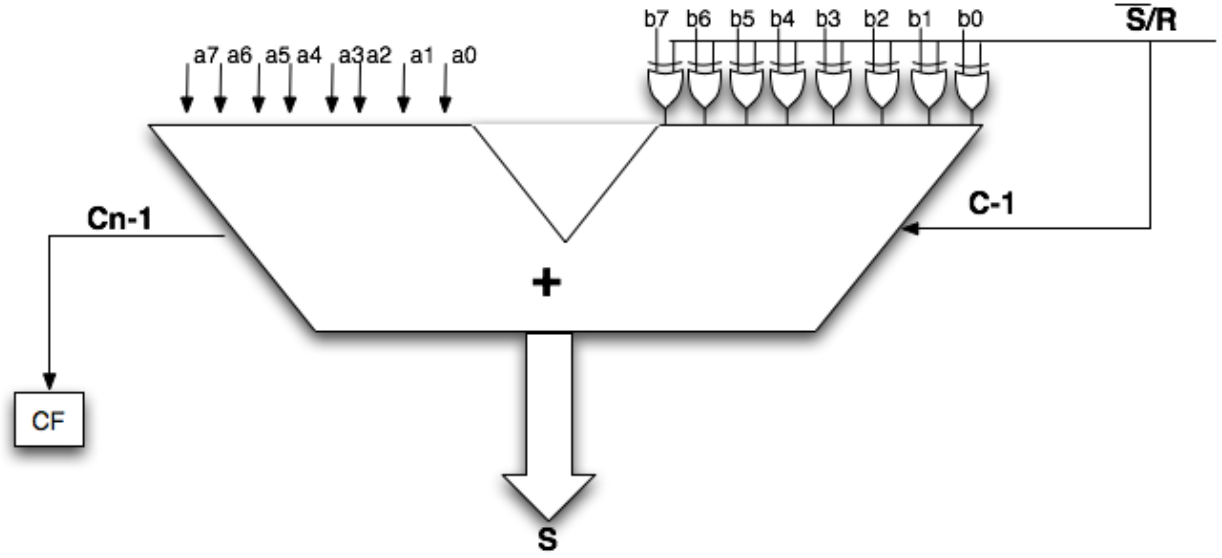


Ámbito de aplicación del operador

- Uno **general** realiza diferentes operaciones
  - Los computadores suelen tener solo uno general
- Uno **especializado** realiza una operación de cierta complejidad
  - Un **coprocesador matemático** es un ejemplo de operador especializado

Sumador-Restador paralelo en Ca2

- Pueda sumar y restar seleccionando la operación mediante la señal de control $\overline{S}/R$
- A-B consiste en cambiar el signo de B y luego sumar los dos operandos
- El cambio de signo dependerá del formato de representación
  - El Complemento a 2 de B es igual a $\overline{B} + 1$
  - ¿Valor de S/R para realizar una suma? ¿y una resta?



Operación de multiplicación

- Sumador-restador + algoritmo
- Sólo máquinas muy potentes cuentan con un operador específico

Algoritmos

- Multiplicación binaria **sin signo**
  - Algoritmo de suma-desplazamiento
  - Algoritmo de sumas y restas
- Multiplicación binaria **con signo**
  - Algoritmo de Booth

- Multiplicadores combinacionales:
  - Más rápidos pero más complicados
  - Sólo en máquinas para cálculo matemático intensivo

Operación de división

- Más compleja que la multiplicación
- Operador sumador-restador + algoritmo
- Es raro que un computador disponga de un divisor combinacional

Ejemplo operación aritmética

```
add a, b, c
```

- MIPS emplea una notación rígida:
  - Cada instrucción aritmética realiza una única operación
  - Siempre tiene tres *variables*
- Si queremos sumar b, c, d y e en a:

```
add a, b, c #Esto es un comentario
add a, a, d
add a, a, e
```

- Cada línea de este lenguaje contiene una única instrucción

Ojo: estas instrucciones todavía no están en el ensamblador del MIPS!

Principios de diseño del hardware

```
add a, b, c
```

- El hardware para un número variable de operandos es más complicado que el hardware para un número fijo
- Cuatro principios de diseño del hardware

Principio de diseño I: La simplicidad favorece la estabilidad

Ojo: estas instrucciones todavía no están en el ensamblador del MIPS!

Compilando dos sentencias de C en MIPS

```
a = b + c;
d = a - e;
```

- La traducción del lenguaje C al lenguaje ensamblador de MIPS la realiza el **compilador**

```
add a, b, c
sub d, a, e
```

Ojo: estas instrucciones todavía no están en el ensamblador del MIPS!

Compilando una sentencia compleja de C en MIPS

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

- ¿Qué genera el compilador de C?

```
add t0, g, h   # t0?
add t1, i, j   # t1?
sub f, t0, t1  # f?
```

Ojo: estas instrucciones todavía no están en el ensamblador del MIPS!

Pregunta

- Dada una determinada función, ¿en qué lenguaje de programación contendrá más líneas de código?
  - **Java**
  - **C**
  - **MIPS**

Operandos y registros

- Los **operandos** de las instrucciones aritméticas son limitados y se corresponden físicamente con una parte del hardware del computador denominada **registros**
- Los registros son una parte del hardware visible a los programadores
- En MIPS el tamaño de un registro es de **32 bits**
- En MIPS a un grupo de 32 bits se le denomina ***palabra***

Operandos y registros

- A diferencia de las variables de un programa, los registros tienen un número limitado
- En MIPS hay 32 registros de 32 bits
- Esta limitación en el número de registro viene impuesta por el:

Principio de diseño II: Más pequeño es más rápido

- Un mayor número de registros supone incrementar el ciclo de reloj puesto que a las señales electrónicas les lleva más tiempo recorrer distancias más largas

Operandos y registros

- En lugar de números, la convención MIPS utiliza el símbolo dólar seguido de dos caracteres para representar un registro
- Por ejemplo:
  - **`$s0`**, `$s1`, … para registros que almacenan variables de programas en C o Java, *registros estáticos*
  - **`$t0`**, `$t1`, … son *registros temporales* necesarios para compilar un programa en instrucciones MIPS

Compilar una asignación en C utilizando registros

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

- Las variables `f, g, h, i` y `j` se asignan respectivamente a los registros `$s0, $s1, $s2, $s3` y `$s4`

```
add $t0, $s1, $s2  # t0 = g + h
add $t1, $s3, $s4  # t1 = i + j
sub $s0, $t0, $t1 # f = t0 - t1
```

Operandos en memoria

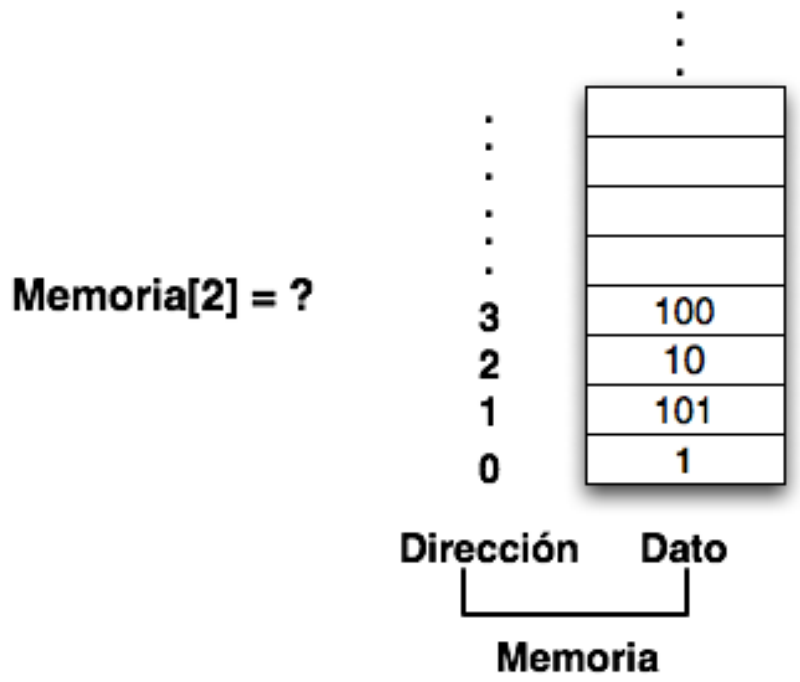
- Hasta ahora, variables sencillas
- Sin embargo, existen **estructuras de datos complejas**:
  - Arrays y estructuras
- Estructuras de datos grandes que contienen más elementos que los que se pueden almacenar en los registros
- ¿Cómo puede acceder y representar el computador estas estructuras de datos?

Operandos en memoria

- … las estructuras de datos se almacenan en memoria
- Sin embargo, en MIPS las operaciones aritméticas se realizan entre registros
- Por lo tanto, se necesitan instrucciones para transferir datos entre los registros y la memoria:
  - **Instrucciones de transferencia de datos**

Operandos en memoria

- Para acceder a una palabra en memoria se necesita la **dirección de memoria** de la misma
- La memoria es como un array unidimensional en el que la dirección es el índice de dicho array



Asignación con operando en memoria: *load*

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

- `A` es un array de 100 palabras. `g` y `h` se asignan a los registros `$s1` y `$s2`. La dirección base del array se almacena en `$s3`

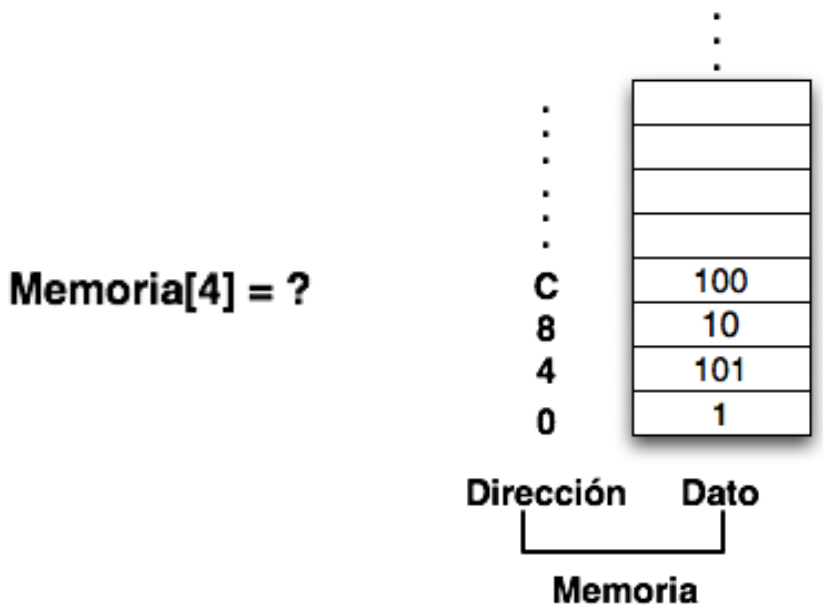
```
lw $t0,8($s3)    # t0 = A[8]
add $s1,$s2,$t0 # s1 = h + A[8]
```

8 => *Offset*, $s3 => *Registro base*

Ojo: el desplazamiento no es correcto!

Operandos en memoria

- El compilador asocia estructuras de datos con direcciones de memoria
- La unidad de direccionamiento es el byte
- Una palabra son 4 bytes …



Asignación con operando en memoria: *load* y *store*

```
A[12] = h + A[8];
```

- `A` es un array de 100 palabras. `h` se asocia al registro `$s2`. La dirección base del array se almacena en `$s3`

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

Offsets?

Diseño de un repertorio de instrucciones

- El repertorio de instrucciones influye directamente en:
  1. El número de instrucciones necesarias para realizar una determinada tarea
  2. El diseño del compilador
  3. El diseño de la ruta de datos y la unidad de control
- Hasta los 80 todo era CISC. Después se impusieron los RISC
- **CISC**, gran número de instrucciones complejas
  - Gran variedad de tipos de datos, de modos de direccionamiento y de operaciones
  - Permite implementar instrucciones de alto nivel directamente o con un número pequeño de instrucciones ensamblador
  - HW más complejo
- **RISC**, pocas instrucciones y muy básicas
  - Pocos tipos de datos y de modos de direccionamiento
  - Técnicas de optimización, tanto a nivel de HW como del compilador, más sencillas de implementar
  - Necesitan más instrucciones para realizar la misma tarea

Aspectos más importantes a tener en cuenta

- **Tipo de almacenamiento de los operandos**
- **Modos de direccionamiento** soportados

Tipo de almacenamiento de los operandos

- Los distintos repertorios se diferencian en el tipo de almacenamiento interno que utilizan:
  - **Pila**
  - **Acumulador**
  - **Registros de propósito general** (*General Purpose Registers*, GPR)

Tipo de almacenamiento de los operandos

- **Pila**: los operandos son implícitos, siempre en la parte superior de la pila (*Top of Stack*, TOS)
  - No es necesario indicar dónde se encuentran los operandos
- Registro **acumulador**: uno de los operandos es implícito
  - El otro se debe especificar de forma explícita

Tipo de almacenamiento de los operandos

- **GPR**: los operandos se especifican de forma explícita
  - Pueden ser dos o tres operandos, ¿por qué?
  - Alguno o todos los operandos pueden estar en memoria. Por lo tanto, se pueden diseñar …

Tipo de almacenamiento de los operandos: **GPR**

- **Registro-Registro** de 3 operandos, todos deben estar en registros
  - Se utilizan instrucciones de carga (load) y almacenamiento (store)
- **Registro-Memoria** de 2 operandos, al menos uno de los operandos debe estar en registro
- **Memoria-Memoria** de 2 o 3 operandos, todos ellos en memoria

Tipo de almacenamiento de los operandos

- Casi todas las arquitecturas se basan en GPR
- **Los registros son más rápidos**
- Son utilizados de manera mucho más eficiente por los compiladores
  - Almacenamiento temporal de variables
- Diferentes alternativas para diseñar un repertorio basado en GPR
  - En los repertorios registro-registro todas las instrucciones tienen la misma longitud y se ejecutan en un número similar de ciclos

Tipo de almacenamiento de los operandos

- En **arquitecturas registro-registro**:
  - La codificación es sencilla, siempre hay que especificar el identificador de tres registros
  - Pero los programas ocupan más, ¿por qué?
- En **arquitecturas memoria-memoria**:
  - Código más compacto
  - La memoria, un cuello de botella, ¿por qué?
  - Grandes diferencias entre la longitud de las instrucciones y entre su duración
  - Se complica la codificación de las instrucciones y puede variar mucho el CPI (Ciclos Por Instrucción) entre instrucciones

Ejemplo

- Vamos a evaluar dos alternativas:
  1. Registro-Registro de 3 operandos con 8 registros de propósito general
  2. Memoria-Memoria de 3 operandos
- Se desea realizar una secuencia de operaciones lógicas

```
R = X AND Y
Z = X OR Y
Y = R AND X
```

- El código de operación ocupa **1Byte**
- Las direcciones de memoria y los operandos ocupan **4 Bytes**
- Los operandos están almacenados inicialmente en memoria

Diseño de un repertorio de instrucciones

	Tráfico con la memoria de instrucciones	**Tráfico con la memoria de datos**
load r1,X	opcode+dir+registro=6B	1 operando = 4B
load r2,Y	opcode+dir+registro=6B	1 operando = 4B
and r3,r1,r2	opcode+3 registro=3B	-
or r4,r1,r2	opcode+3 registro=3B	-
and r2,r3,r1	opcode+3 registro=3B	-
store r3,R	opcode+dir+registro=6B	1 operando = 4B
store r4,Z	opcode+dir+registro=6B	1 operando = 4B
store r2,Y	opcode+dir+registro=6B	1 operando = 4B
	Total, **39B**	Total, **20B**

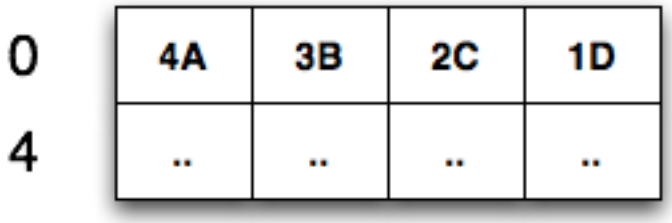
	Tráfico con la memoria de instrucciones	**Tráfico con la memoria de datos**
and R,X,Y	opcode+3 direcciones=13B	3 operandos = 12B
or Z,X,Y	opcode+3 direcciones=13B	3 operandos = 12B
and Y,R,X	opcode+3 direcciones=13B	3 operandos = 12B
	Total, **39B**	Total, **36B**

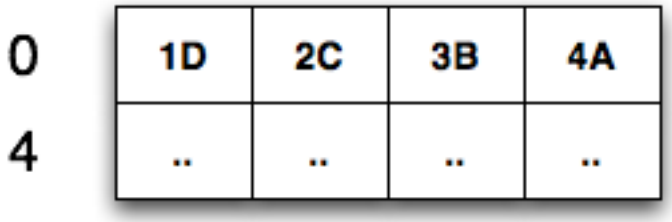
Interpretación de las direcciones de memoria

- La mayoría de las máquinas están direccionadas por bytes
  - Proporcionan acceso a bytes (8 bits), medias palabras (16 bits), palabras (32 bits) y dobles palabras (64 bits)
- Convenios para clasificar los bytes de una palabra
- ***Little Endian***, *"little-end-in"*, de comienzo por el extremo pequeño
  - Coloca el byte menos significativo en la posición más significativa de la palabra
  - La **dirección** de un dato es la del byte menos significativo
- ***Big Endian***, *"big-end-in"*, de comienzo por el extremo grande
  - Coloca el byte menos significativo en la posición menos significativa de la palabra
  - La **dirección** de un dato es la del byte más significativo
- ***Middle Endian***, arquitectura capaz de trabajar con ambas ordenaciones, como por ejemplo los procesadores MIPS o Power PC

Interpretación de las direcciones de memoria

- **Ejemplo**: el valor hexadecimal 0x4A3B2C1D se codificaría en memoria en la secuencia:
  - {4A, 3B, 2C, 1D} en big-endian
  - {1D, 2C, 3B, 4A} en little-endian





Interpretación de las direcciones de memoria

- **Ejemplo**: explica cómo puede ser que tras la ejecución del mismo código, en unas máquinas se imprima un mensaje y en otras otro. Debes indicar la ordenación de los datos en memoria teniendo en cuenta que un entero ocupa 16 bits y un char 8 bits. Además, debes indicar el valor de p[0] y p[1] en cada caso

```
#include <stdio.h>

int main(void){
   int i = 1;
   char *p = (char *) &i;
   if ( p[0] == 1 )
        printf("Little Endian\n");
   else
        printf("Big Endian\n");
   return 0;}
```

Restricciones de alineamiento

- En MIPS, las palabras deben comenzar en direcciones múltiplo de 4
- Estos accesos deben estar **alineados**, es decir, un acceso a una información de s bytes en la dirección del byte B está alineado si

  $$B \text{ módulo } s = 0$$

- Estas restricciones de alineamiento se deben a que las memorias, físicamente, están diseñadas para hacer accesos alineados
- Un acceso **no alineado** o **mal alineado**, supone varios accesos alineados a la memoria

Operandos en memoria Vs operandos en registros

- Los registros tienen un menor tiempo de acceso y un mayor *throughput* que la memoria:
  - Se **accede más rápidamente** y de forma más sencilla a los datos almacenados en registros
- Se necesita **menos energía** para acceder a los registros que a la memoria
- Para alcanzar un mejor rendimiento y conservar energía, los compiladores deben hacer un uso eficiente de los registros

Operandos inmediatos

- Con las instrucciones que hemos visto hasta ahora ...

```
la   $s0,AddrCons4 # s0 = AddrCons4
lw   $t0,0($s0)    # t0 = 4
add $s3,$s3,$t0    # s3 = s3 + t0
```

- ... en su lugar existe una versión de todas las instrucciones aritméticas en la que uno de los operandos es una constante

```
addi $s3,$s3,4 # s3 = s3 + 4
```

Operandos y registros

- Las instrucciones inmediatas ilustran el tercer principio de diseño

Principio de diseño III: Haz que el caso más común sea rápido

- La constante cero tiene el rol de simplificar el juego de instrucciones ofreciendo variaciones útiles
- MIPS tiene un registro cableado a cero, se trata del registro `$zero` que se corresponde con el registro número 0

Representación de las instrucciones en el computador

- Convención para mapear nombres de registro en números:

Alias	**Número**	**Uso**
zero	0	Valor cableado cero
at	1	Reservado para el ensamblador
v0, v1	2, 3	Evaluación de expresión y valor de retorno de subrutina
a0, …, a3	4, …, 7	Argumentos de la subrutina
t0, …, t7	8, …, 15	Variables temporales o auxiliares de corta duración
s0, …, s7	16, …, 23	Variables de larga duración
t8, t9	24, 25	Variables temporales o auxiliares de corta duración
k0, k1	26, 27	Reservado para el kernel del SO
gp	28	Puntero a zona de variables estáticas *(global pointer)*
sp	29	Puntero de pila *(stack pointer)*
fp	30	Puntero de marco *(frame pointer)*
ra	31	Dirección de retorno *(return address)*

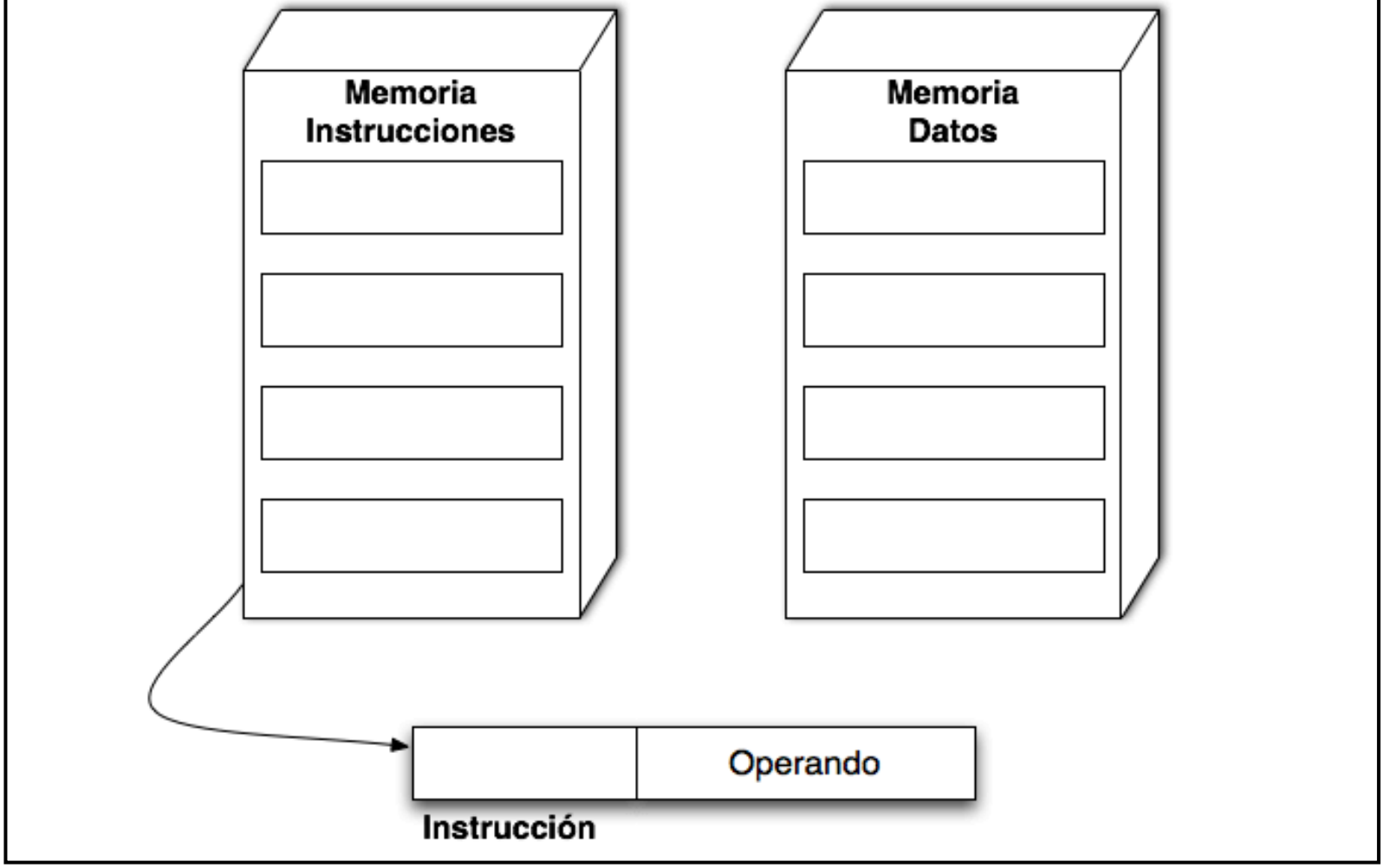
Modos de direccionamiento

- ¿Dónde podemos encontrar un operando?
  - **En la propia instrucción**
  - **En un registro**
  - **En memoria principal**

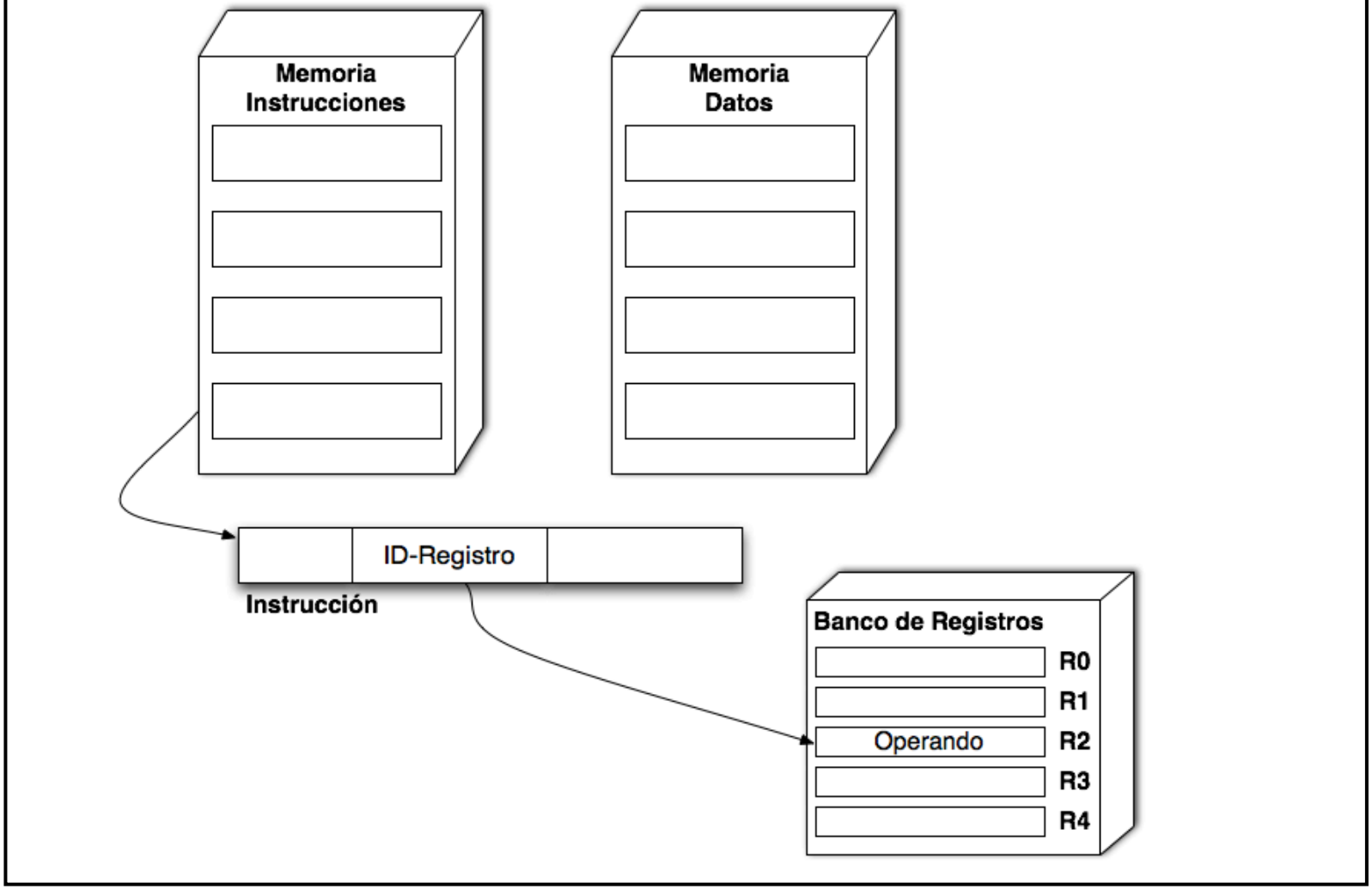
Modos de direccionamiento básicos

- **Inmediato**: el operando se codifica dentro de la instrucción
- **Registro**: se incluye el identificador del registro que amacena el operando
- **Directo**: se incluye la dirección de memoria en la que está almacenado el operando
- **Indirecto**: se indica el registro que almacena la dirección de memoria en la que se encuentra el operando
- **Indirecto con desplazamiento**: se suma un operando inmediato al contenido del registro para obtener la dirección de memoria en la que se encuentra el operando

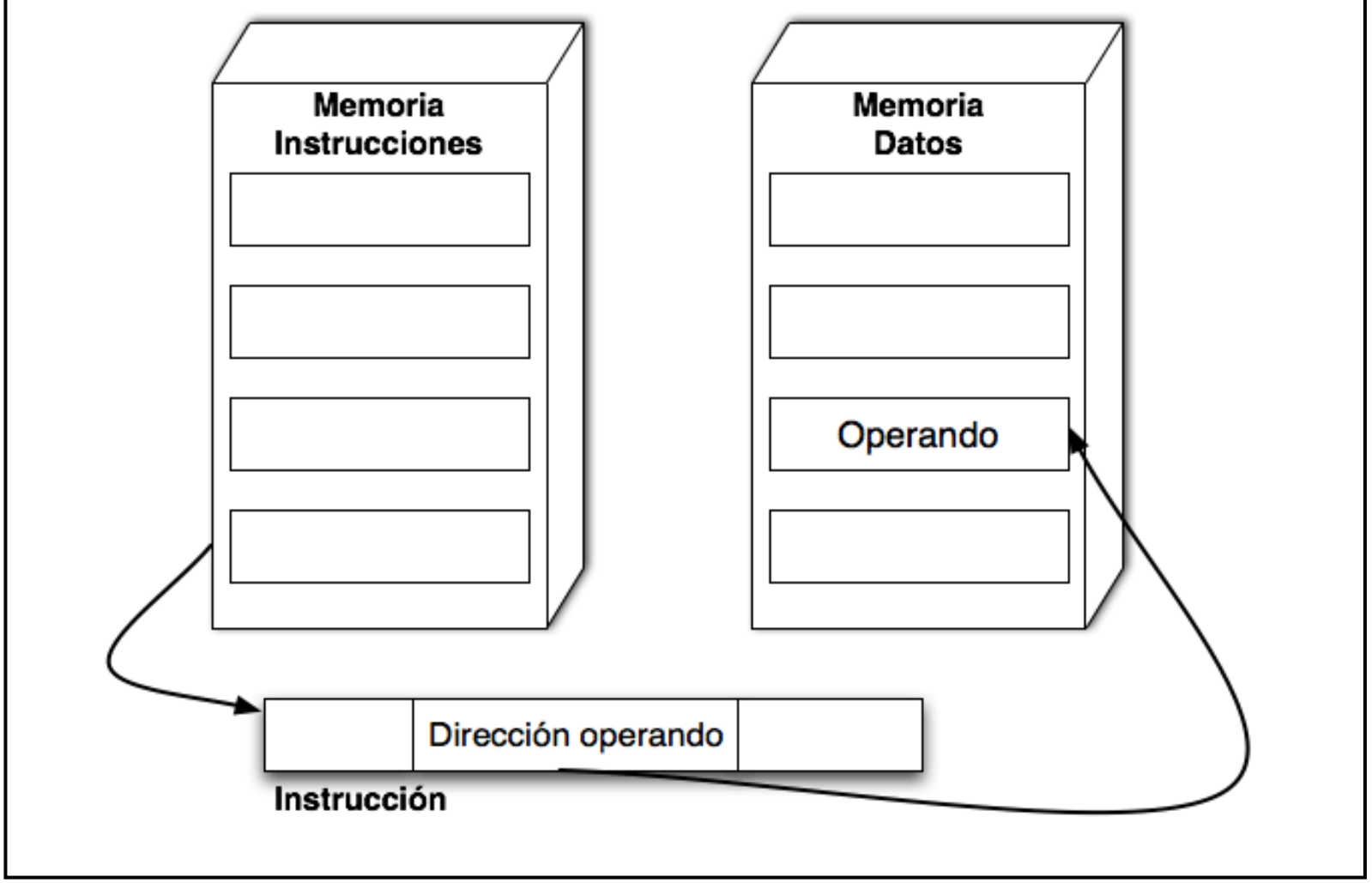
Direcccionamiento inmediato



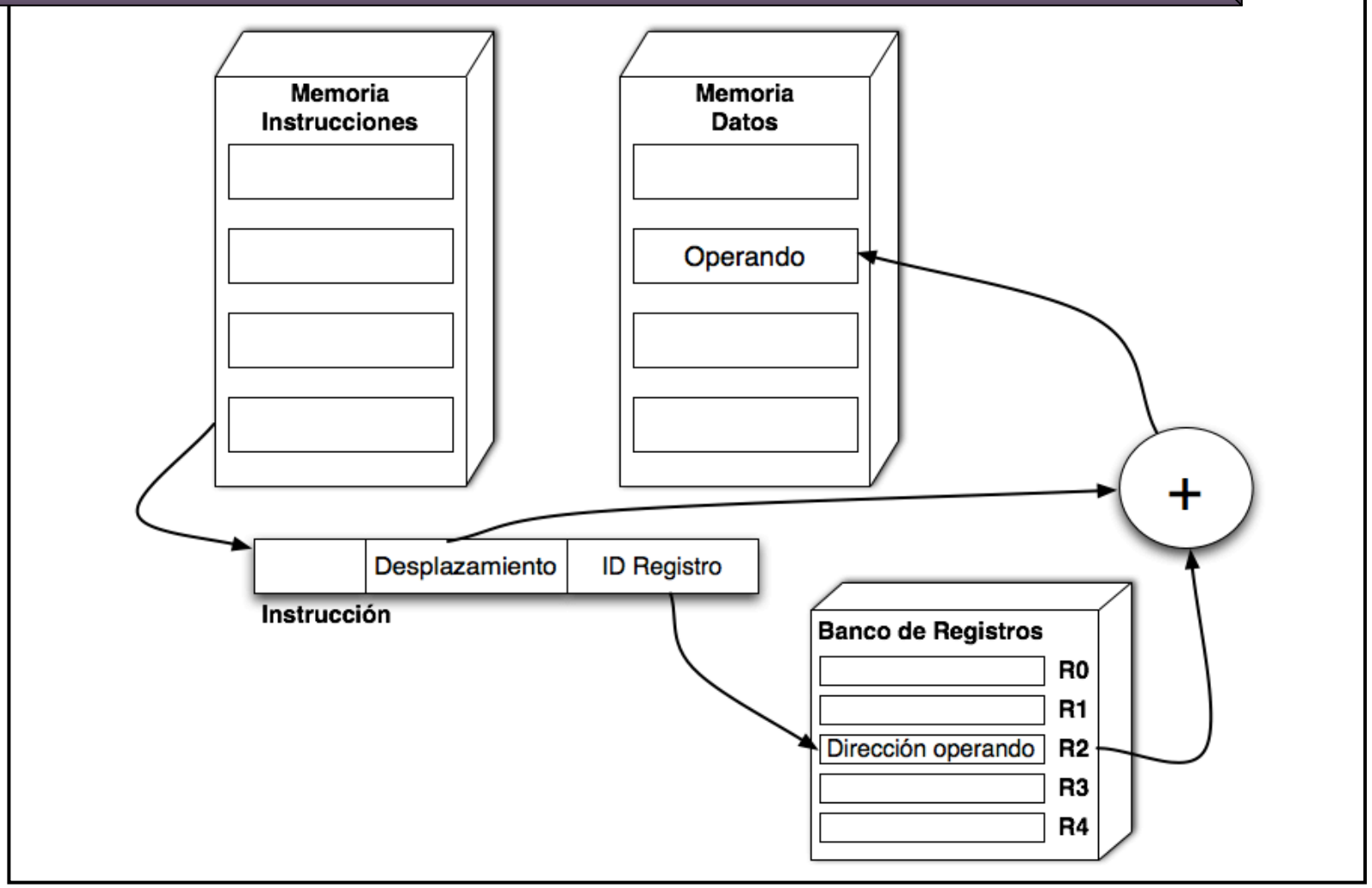
Direcccionamiento de registro



Direcccionamiento directo



Direcccionamiento indirecto con desplazamiento



Modos de direccionamiento en un repertorio RISC

- Los repertorios RISC incluyen como mínimo direccionamiento inmediato e indirecto con desplazamiento
- **Direccionamiento inmediato**, a la hora de diseñar el repertorio hay que decidir sí:
  1. Todas las instrucciones deben soportar este modo o sólo un subconjunto
  2. El rango de valores del operando inmediato, ¿en qué influye esto?
- **Direccionamiento indirecto con desplazamiento**: la decisión más importante consiste en determinar el rango de valores que puede tomar el desplazamiento

Otras consideraciones

- **Tipo y tamaño de los operandos**:
  - ¿Qué tipo de datos se soportan? ¿Con qué tamaños? Carácter, entero, coma flotante, etc
  - El código de operación, *opcode*, indicará el tipo de los operandos implicados en la ejecución de la instrucción
  - … también lo pueden indicar los operandos mediante etiquetas, ¿inconvenientes?
- **Conjunto de operaciones soportadas**:
  - ¿Qué tipo de operaciones van a realizar las instrucciones del repertorio?
  - Un conjunto sencillo: aritmético-logicas, de acceso a memoria, de control de flujo (saltos) y llamadas al sistema operativo
  - Dependiendo de los tipos de datos, instrucciones para manejo de caracteres, coma flotante, etc
- **Tratamiento de las instrucciones de control de flujo**: modifican el flujo de control de un código

Tratamiento de las instrucciones de control de flujo

- **Saltos condicionales**: ¿Cómo se especifica la condición? ¿Cómo se indica la dirección destino de salto?
- **Saltos incondicionales**: ¿Cómo se indica el destino?
- Dos alternativas para indicar el destino de salto
- **Direccionamiento relativo al PC**
  - Se conoce el destino de salto en tiempo de compilación
  - Los destinos de los saltos están cercanos al salto
  - Código reubicable
  - ¿Cuántos bits se necesitan para el desplazamiento?
- **Direccionamiento indirecto con registro**
  - No se conoce la dirección de salto o su valor excede del que se puede indicar con el desplazamiento
  - Se indica el identificador del registro que contiene la dirección destino de salto

Tratamiento de las instrucciones de control de flujo

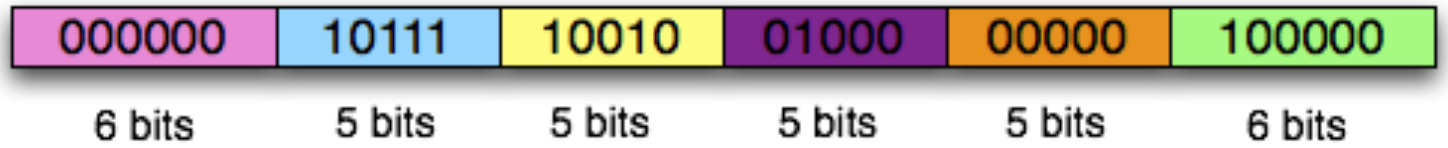
- La condición de salto normalmente suele ir referida a comparaciones con uno o varios registros

Campos de una instrucción MIPS

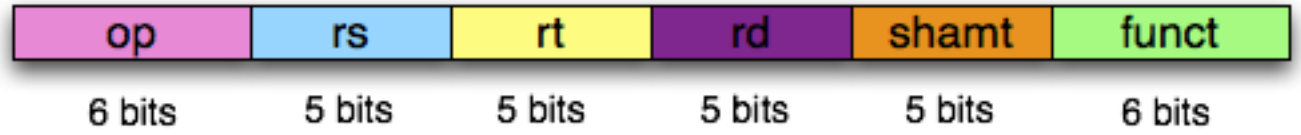
```
add $t0, $s1, $s2
```

Representación decimal

Representación binaria



Campos de una instrucción MIPS



- *op*: operación básica de la instrucción, tradicionalmente denominada opcode
- *rs*: registro del primer operando fuente
- *rt*: registro del segundo operando fuente
- *rd*: registro del operando destino
- *shamt*: desplazamiento
- *funct*: función, sirve para seleccionar la operación específica a realizar

Campos de una instrucción MIPS

- *¿Qué pasa si una instrucción necesita campos más largos*? Conflicto entre el deseo de que todas las instrucciones tengan la misma longitud y el deseo de tener un único formato de instrucción

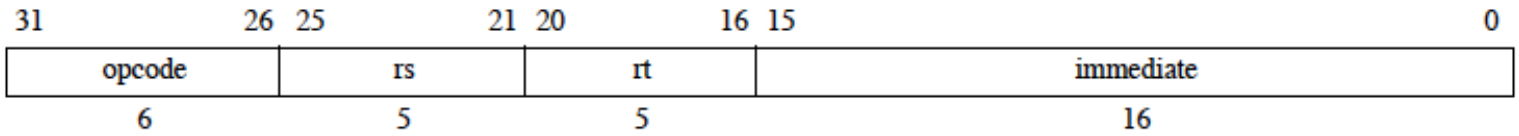
***Principio de diseño IV**:* Un buen diseño demanda buenos compromisos

- El compromiso elegido por los diseñadores del MIPS es mantener todas las instrucciones con la misma longitud, pero requiere de distintos formatos de instrucción para distintos tipos de instrucciones

Codificación del repertorio de instrucciones

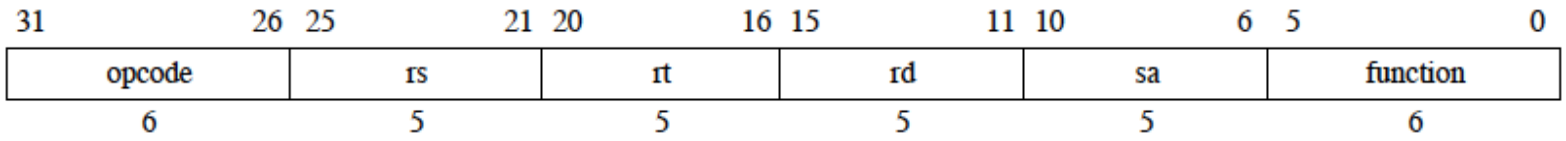
- **Longitud variable**
  - Soporta cualquier número de operandos y cualquier combinación instrucción/modo de direccionamiento
  - ***Etiquetas*** que indican el modo
  - Se añaden tantos campos como sean necesarios + las etiquetas que permiten su interpretación
- **Longitud fija**
  - El **código de operación** especifica el modo de direccionamiento
  - Sólo se permiten unas combinaciones determinadas de operaciones+modos
  - Los campos de la instrucción son siempre los mismos
- **Híbrida**
  - Sólo se permiten unos determinados formatos de instrucción, que incluyen un número variable de modos y operandos

Instrucciones tipo I (Inmmediate)



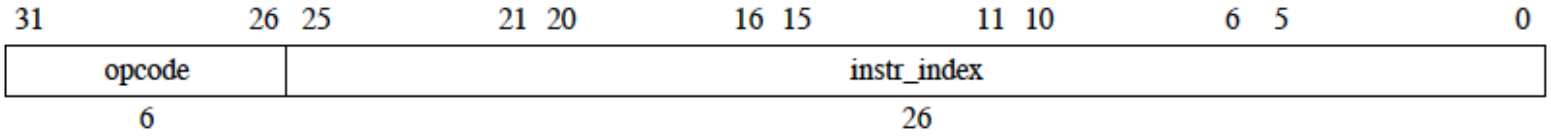
- **Load/Store**
  - **RS (registro fuente)**: registro base para el acceso a memoria
  - **RT (registro destino)**: registro para los datos
  - **Inmediato**: desplazamiento para el cálculo de la dirección de memoria a la que hay que acceder
- **Aritmético-lógicas** con direccionamiento inmediato
  - **RS (registro fuente)**: operando 1
  - **RT (registro destino)**: registro destino de la operación
  - **Inmediato**: operando 2, directamente su valor
- **Saltos codicionales/incodicionales**
  - **RS (registro fuente)**: registro de condición (para la comparación)/ Registro que contiene la dirección destino del salto
  - **RT (registro destino)**: registro de condición (para la comparación)/No se utiliza
  - **Inmediato**: desplazamiento respecto del PC/0

Instrucciones tipo R (Register)



- **Aritmético-lógicas** registro-registro
  - **RS (registro fuente)**: operando 1
  - **RT (registro destino)**: operando 2
  - **RD**: registro destino
  - **sa (Shift Amount)**: indica el desplazamiento para las instrucciones de tipo Shift
  - **function**: junto con el OpCode indica el tipo de operación que se debe realizar

Instrucciones tipo J (Jump)



- **Salto incondicional** y **retorno de procedimiento** que utilizan direccionamiento con desplazamiento relativo al PC
  - **opCode**: Código de la operación
  - **Instr_index**: offset relativo el PC