



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Departament d'Arquitectura de Computadors

# Estructura de Computadores

## Tema 2: Instrucciones y Tipos Básicos de Datos

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



# Introducción

- ❑ **Introducción Computadores (IC)**
  - Estudio de un Computador Simple (SISP)
  - Diseño a nivel de Circuitos Lógicos Digitales
  - Relación del Lenguaje Máquina con el Hardware
- ❑ **Programación 1 (PRO1)**
  - Conceptos básicos de Programación
  - Lenguaje de Programación de: C++
- ❑ En **Estructura de Computadores (EC)** estudiaremos la relación entre el Lenguaje Máquina (Ensamblador) y un Lenguaje de Programación de Alto Nivel.
  - Lenguaje Máquina: MIPS de 32 bits
  - Lenguaje de Alto Nivel: C

# Descripción Jerárquica del Computador



---

**Aplicación Final:** Menús, Iconos, Botones, Cut&Paste , ...

PRO1

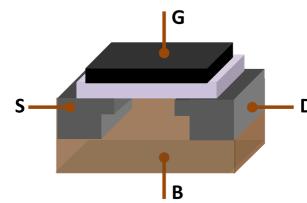
**Lenguaje de Alto Nivel:** C/C++, Java, Fortran, Python, PHP, ...

EC

**Lenguaje Máquina:** MIPS, x86, ARM, RISC-V, ...

IC

**Hardware:** Puertas lógicas, Multiplexores, Biestables, ...



# Interfaz entre Niveles

- Cada nivel establece una interfaz con lo que el nivel superior puede interactuar
  - **ABI:** Application Binary Interface
  - **ISA:** Instruction Set Architecture

Usuario



# Instruction Set Architecture (ISA)

La ISA (Instruction Set Architecture) es una especificación que describe los aspectos del procesador visibles al programador de Lenguaje Máquina (o ensamblador): Instrucciones, Registros, Modos de Direcciónamiento, Modelo de Memoria, Entrada/Salida, excepciones, etc.

- ❑ La misma ISA puede ser implementada de diferentes maneras (Intel vs AMD)
- ❑ Compatibilidad a nivel hardware: un programa compilado para una ISA (y SO) se podrá ejecutar en cualquier hardware que la implemente
- ❑ Ejemplos: MIPS, x86, ARM, RISC-V

# Instruction Set Architecture (ISA)

MOV EAX, 78  
MOV EBX, BUFFER  
XOR ECX, ECX  
MOV ECX, 1000  
MUL ECX  
MOV EBX, EAX  
DIV ECX  
ADD EAX, EBX  
MOV [START\_TIME], EAX

x86

MOVWF CONTADOR  
MOVLW NUM\_SEC  
SUBWF CONTADOR, W  
BTFS C STATUS, C  
CALL NUMERO  
CALL LCD  
GOTO BLOCK  
MOVF CONTADOR, W  
ANDLW B'00001111'

pic16

BNE David3  
STA David2  
LSR A  
LDX #3  
STX BLPTR+1  
STX BLN+1  
STX EXCN+1  
DEX  
JSR OSBYTE

6502

MOV X2, X0  
MOV X0, #1  
ADRP X1, B@PAGE  
ADD X1, X1, B@PAGE  
MOV X16, #4  
SVC #0X80  
MOV X0, #0  
MOV X16, #1  
SVC #0X80

ARM

LD HL, MEMORY  
LD D, 0  
LD E, C  
ADD HL, DE  
LD A, [HL]  
OR A  
LD A, [MEMORY\_ID]  
SET 7, A  
LD [MEMORY\_ID], A

z80

LA \$A0, V  
LI \$A2, 4  
LW \$A3, 0(\$T0)  
JAL BINARYSEARCH  
LA \$T0, RESULT  
SW \$V0, 0(\$T0)  
LW \$RA, 0(\$SP)  
ADDIU \$SP, \$SP, 4  
JR \$RA

MIPS

DCR H  
JNZ AFTERCH2  
LDA SETCH2VOL+2  
CPI 0D3H  
MVI A, 0D3H  
JNZ \$+5  
MVI A, 0DBH  
STA SETCH2VOL+2  
MOV H, B

8080

Existen numerosos ISAs. Todos diferentes, aunque ...

# Todos los ISAs tienen:

## Instrucciones aritmético/lógicas:

- $OP_d \leftarrow OP_1 \text{ (?) } OP_2$

## Instrucciones de Acceso a Memoria:

- $OP_d \leftarrow MEM[dir]$
- $MEM[dir] \leftarrow OP_f$

## Instrucciones de ruptura de secuencia:

- if ( $OP_1$  cond  $OP_2$ ) goto L

- Los operandos ( $OP_x$ ) pueden ser registros del procesador, memoria, o constantes.

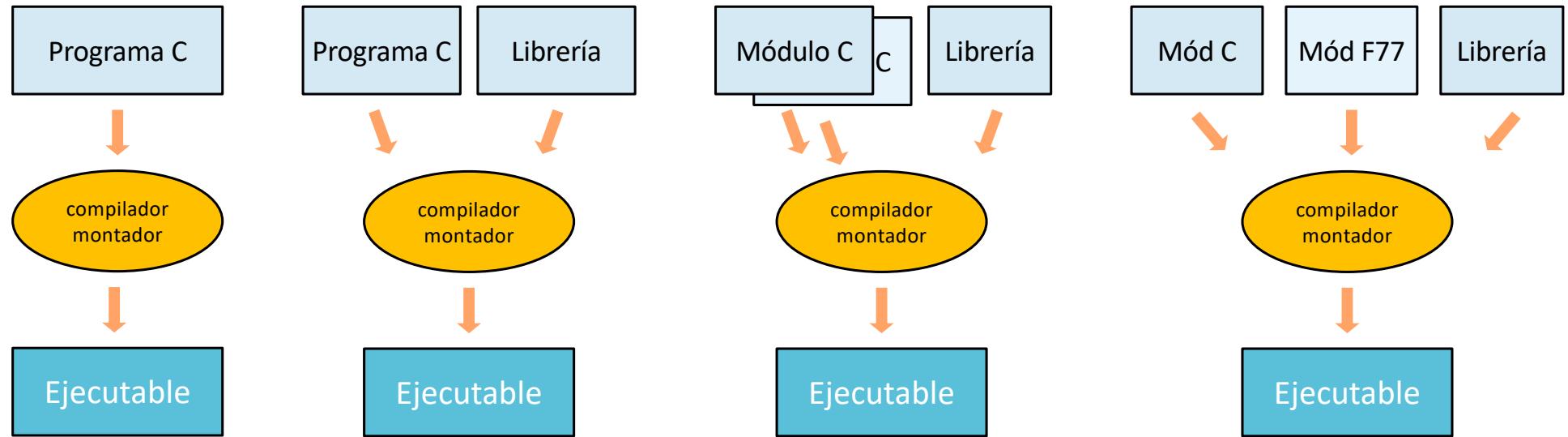
- Los operadores son los que podemos encontrar en cualquier LAN:

- ✓ **aritméticos:** +, -, \*, /, %
- ✓ **bit-wise:** ~, |, &, ^, >>, <<
- ✓ **lógicos:** !, ||, &&
- ✓ **relacionales:** !=, ==, >, <, >=, <=

- La dirección con la que accedemos a memoria puede ser algo muy simple (un registro), o requerir un cálculo más complejo (modos de direccionamiento)

- Las condiciones de salto han de permitir implementar las sentencias condicionales e iterativas de un LAN convencional.

# Application Binary Interface (ABI)



CPU  
MIPS

Para construir el ejecutable, los diferentes módulos y librerías han seguir un convenio: ABI

# Application Binary Interface (ABI)

Una Application Binary Interface (ABI) es una especificación que describe la interfaz de bajo nivel (p.e. convenios para llamar a una función, retornar resultado, ...) entre los módulos de un programa. En particular, con las librerías del Sistema Operativo.

- La ABI define la interfaz entre módulos y librerías.

Dos módulos que siguen la misma ABI comparten:

- Una ISA en concreto
- Los tamaños y alineaciones de los diferentes tipos de datos
- La forma de llamar a las funciones, paso de parámetros, recogida de resultados
- Cómo se hacen las llamadas al Sistema Operativo
- Formato de los binarios

Normalmente, todo esto es función del compilador/montador

# Compilación vs Interpretación

- ❑ ¿Cómo ejecutamos un programa escrito en C en una CPU MIPS?
- ❑ En una CPU MIPS sólo se puede ejecutar una código escrito en Lenguaje Máquina MIPS.
- ❑ Necesitamos algún mecanismo para pasar de C a MIPS.

```
void swap(int v[], int k) {  
    int tmp;  
    tmp = v[k];  
    v[k] = tmp;  
    v[k+1] = v[k];  
}
```

C

```
swap: sll $v0, $a1, 2  
       addu $v0, $a0, $v0  
       lw   $t0, 0($v0)  
       lw   $t1, 4($v0)  
       sw   $t1, 0($v0)  
       sw   $t0, 4($v0)  
       jr   $ra
```

Ensamblador  
MIPS

```
0000 1000 1001 1110 1000 0010 0000 0100  
...
```

Lenguaje  
Máquina MIPS

- ❑ El Lenguaje Ensamblador y Máquina son “básicamente” LO MISMO.

# Compilación

- ❑ Compilar es traducir un programa escrito en un Lenguaje X a un programa equivalente escrito en un lenguaje Y.
- ❑ Normalmente un compilador traduce un programa escrito en un Lenguaje de Alto Nivel a un programa equivalente escrito en un Lenguaje Máquina específico.
- ❑ Ejemplos de Lenguajes que siempre se compilan: C/C++, Fortran, Pascal, Cobol, ...
- ❑ Características
  - Sólo se ha de compilar 1 vez, se puede ejecutar muchas veces.
  - Si cambiamos de ISA (y/o ABI) se ha de recompilar.
  - Un compilador es una aplicación **MUY COMPLEJA** (100.000's de líneas de código).
  - El código generado por un compilador puede ser **MUY EFICIENTE**.

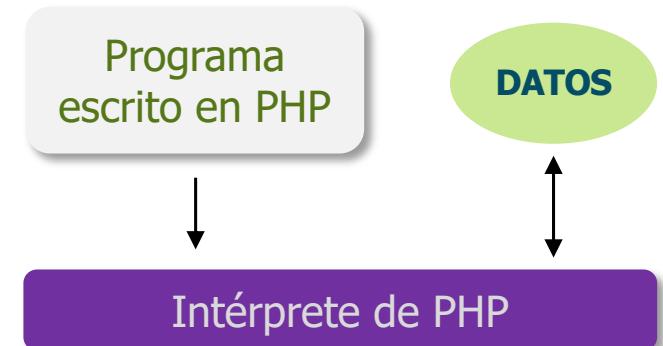
Programa escrito en C++

↓  
Compilador de C++ → MIPS

↓  
Programa ejecutable en MIPS

# Interpretación

- ❑ Un intérprete tiene como entrada un programa escrito en un Lenguaje de Alto Nivel y lo ejecuta directamente.
- ❑ Primero comprueba que no haya errores sintácticos, igual que un compilador, y luego ejecuta el código.
- ❑ Es muy común: Lenguajes de Alto Nivel: BASIC, PHP, Phyton, Perl, ...; el Shell de Linux; Máquina Virtual de Java; una CPU es un intérprete de Lenguaje Máquina.
- ❑ Características
  - Se ha de “traducir” en cada ejecución.
  - Un mismo código puede ejecutarse en múltiples plataformas sin cambios.
  - Un intérprete es una aplicación más simple que un compilador.
  - La ejecución es más **INEFICIENTE**.
  - Es más fácil/rápido depurar programas interpretados.



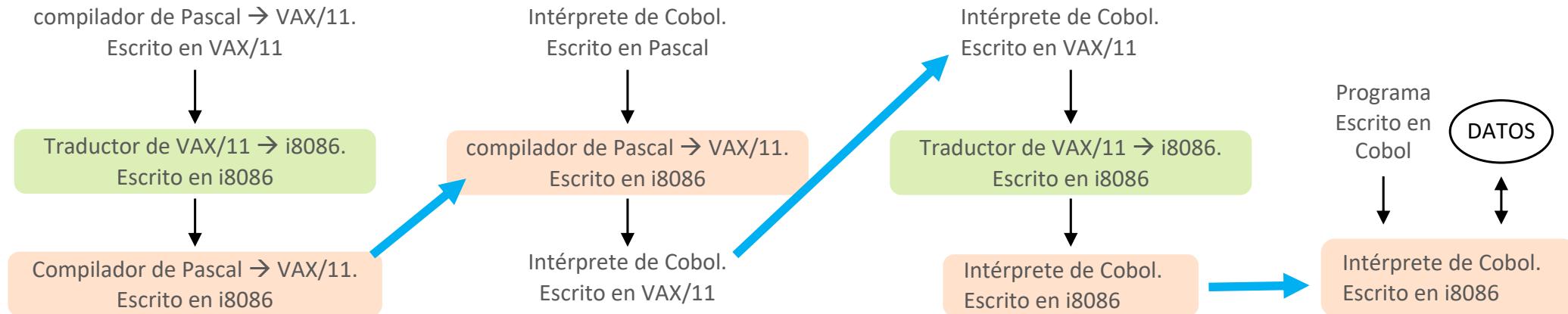
# Problema

Tenemos:

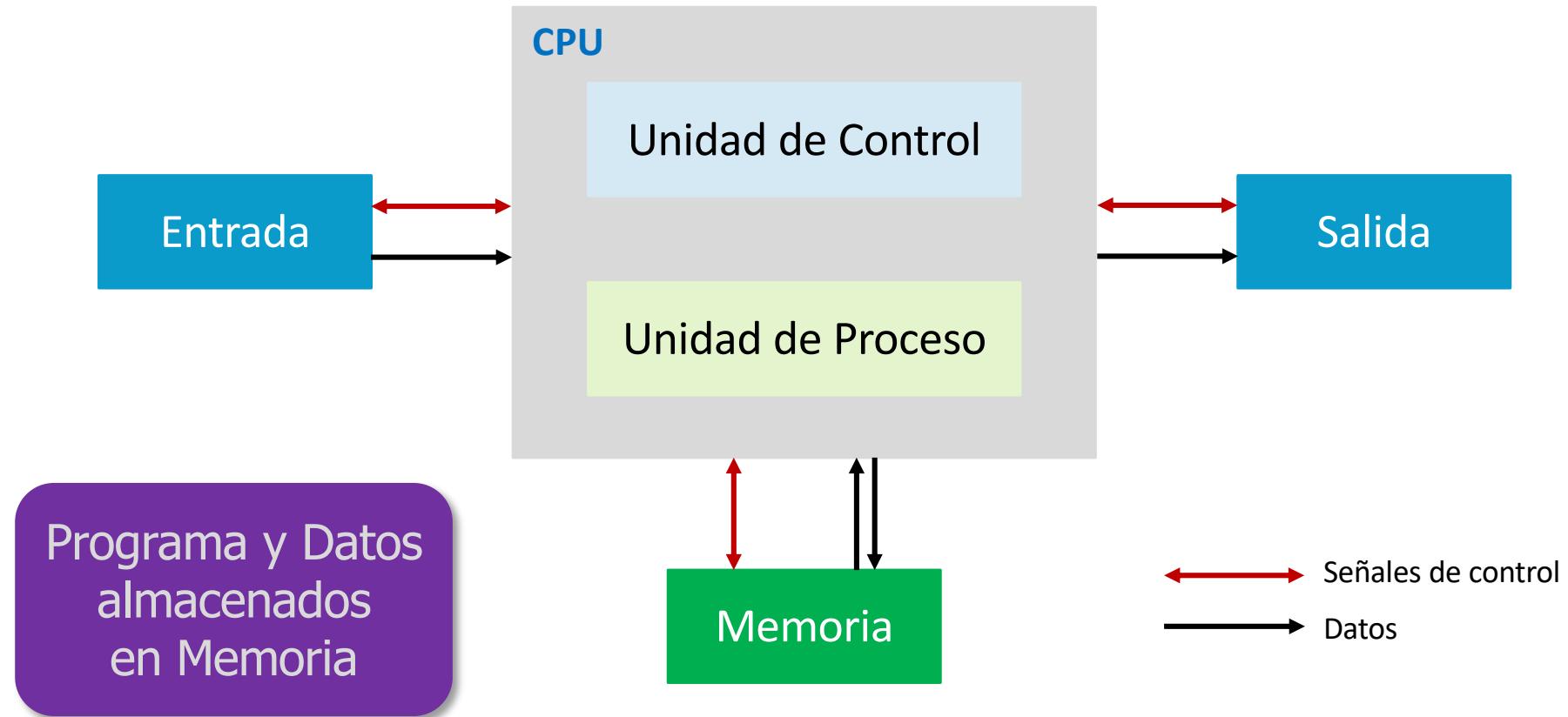
- ❑ Una ordenador con una CPU i8086.
- ❑ Un compilador de Pascal → VAX/11. Escrito en VAX/11
- ❑ Un traductor de VAX/11 → i8086. Escrito en i8086
- ❑ Un interprete de Cobol. Escrito en Pascal.

Examen EC  
abril 1988

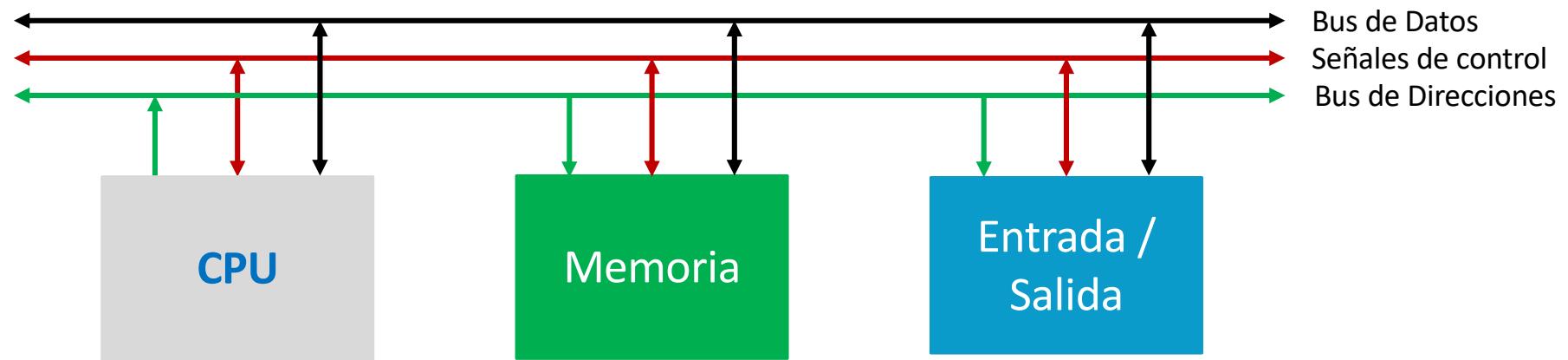
¿Qué pasos hay que seguir para ejecutar un programa escrito en Cobol.



# Arquitectura von Neumann



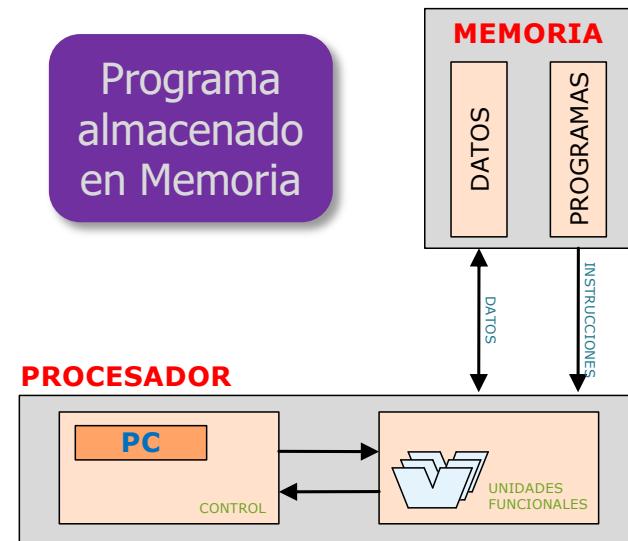
# Arquitectura von Neumann



Programa y Datos  
almacenados  
en Memoria

**Esquema simplificado.**  
La arquitectura del PC original de IBM de 1981  
tenía una arquitectura muy similar a ésta.

# Máquina (von Neumann) de Programa Almacenado en Memoria



**John von Neumann** (matemático húngaro) "First Draft of a report on the EDVAC" contract n. W-670-DRD-492 Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, **June 1945**

Sólo aparece un autor aunque el reporte es resultado de múltiples horas de discusión del grupo que diseñó el ENIAC, en el que von Neumann era sólo un visitante. Prosper Eckert y John Mauchly (diseñadores principales del ENIAC) abandonaron la Moore School por este hecho.

- ❑ Memoria accesible por dirección.
- ❑ Las posiciones de memoria se pueden leer/escribir las veces que sean necesarias.
- ❑ **Tanto los datos como las instrucciones se almacenan en memoria.**
- ❑ No existe ninguna señal para diferenciar en memoria datos de instrucciones.
- ❑ Las instrucciones se ejecutan en secuencia.
- ❑ Existe un registro (**PC**) que apunta siempre a la instrucción en ejecución.
- ❑ Existen instrucciones explícitas para romper el secuenciamiento.
- ❑ Las instrucciones son **imperativas**, especifican **cómo** obtener los operandos, **qué** operación hay que realizar y **dónde** dejar el resultado.

# Introducción a la ISA de MIPS

La Instruction Set Architecture (ISA) es una especificación que describe: Instrucciones, Registros, Modos de Direccionamiento, Modelo de Memoria, Entrada/Salida, excepciones, etc.

- ❑ MIPS es un ISA de tipo RISC (fue el primer RISC de la historia)
- ❑ MIPS son las siglas de *Microprocessor without Interlock Pipeline Stages*.
- ❑ Dr. John Hennessy de la Universidad de Stanford
  - En 1981, inicio el proyecto MIPS RISC Architecture
  - Doctor Honoris Causa por la UPC en 2002
- ❑ Primer procesador: MIPS R2000 (MIPS-I de 32 bits)
- ❑ Distintas implementaciones comerciales: Routers de Cisco y Linksys; Modems ADSL; controladores de impresora; Playstation (PSX, PS2 y PSP); Nintedo 64
- ❑ Nosotros estudiaremos la versión MIPS32 (teoría y laboratorio)
  - Registros de **32 bits**, direcciones de **32 bits**, instrucciones de **32 bits**

# RISC vs CISC

## Complex Instruction Set Computer (CISC)

- ❑ Juego de instrucciones grande y complejo.
- ❑ Instrucciones de longitud variable
- ❑ Modos de direccionamiento complejos
- ❑ Cualquier instrucción puede acceder a memoria.
- ❑ Cada instrucción se convierte en 1..n microoperaciones (inst. RISC?)
- ❑ Ejemplos: x86

## Reduced Instruction Set Computer (RISC)

- ❑ Juego de instrucciones pequeño y sencillo.
- ❑ Instrucciones de longitud fija
- ❑ Formatos de instrucción y modos de direccionamiento sencillos
- ❑ Acceso a memoria con instrucciones específicas: load y store
- ❑ Ejecutadas directamente por hardware
- ❑ Ejemplos: MIPS, ARM, RISC-V

# RISC, es una idea del pasado?

Un porcentaje elevadísimo de los computadores que están funcionando hoy utilizan x86 (arquitectura CISC).

- ❑ Sin embargo, en EC enseñamos una arquitectura RISC
- ❑ La controversia entre CISC y RISC viene desde mediados de los 80's.
- ❑ En AC, os enseñarán la arquitectura x86
  
- ❑ ¿RISC es una idea antigua?
  - **Fugaku**. El supercomputador más potente del mundo (nov 2021), está basado en el A64FX que utiliza el ISA de ARMv8 (RISC).
  - **Apple M1**. Los nuevos Macbooks utilizarán ARMv8-A (RISC)
  - Prácticamente **TODOS** los smartphones utilizan arquitectura ARM (RISC)
  - **RISC-V**. Es una nueva Open ISA (sin licencia). La idea es animar a la innovación en hardware a través de una ISA extensible. **European Processor Initiative**.

# Instrucciones básicas: Operaciones

## □ Operadores en C (o cualquier otro LAN)

- Aritméticos: +, -, \*, /, %
- Lógicos: !, ||, &&
- bit-wise: ~, |, &, ^, >>, <<
- relacionales: !=, ==, >, <, >=, <=
- Ejemplos:

- ✓  $c = a+b;$
- ✓  $x = x/5;$
- ✓  $d = a \& 0x7;$

Cualquier expresión en C puede convertirse en secuencias del tipo:

- $c = a \text{ op } b; // c = a+b;$
- $c = a \text{ op } inm; // c = a-45;$

Usaremos 2 “tipos” de instrucciones aritmético/lógicas:

- $Rd \leftarrow R1 \text{ op } R2$
- $Rd \leftarrow R1 \text{ op } inm$

**SIEMPRE con REGISTROS**



# Instrucciones básicas: Memoria

- ❑ En C (o cualquier LAN) necesitamos acceder a estructuras de datos complejas

- Vectores, Matrices y Tuplas
  - Ejemplos:

- ✓ `c = M[6][i];`
    - ✓ `x = V[i];`
    - ✓ `d = G[i].s;`

- ❑ Podemos ver la memoria como un vector de bytes de tamaño TAM:

- `unsigned char MEM[TAM];`
  - Con 2 operaciones básicas:
    - ✓ LEER(load): `dat = M[i];`
    - ✓ ESCRIBIR (store): `M[i] = dat;`

Las estructuras de datos han de almacenarse en MEMORIA

Usaremos 2 instrucciones de acceso a Memoria:

- `Rd ← M[R1+cte] // load`
- `M[R1+cte] ← R2 // store`

# Instrucciones básicas: Ruptura de secuencia

- En C (o cualquier LAN) necesitamos utilizar sentencias condicionales e iterativas

- if, while, for

- Ejemplos:

- ✓ `if (a>b) c = 7;`

- ✓ `for (i=0; i<N; i++) ...`

- Todas las instrucciones pueden estar etiquetadas, las instrucciones de salto hacen referencia a esas etiquetas.

- Los saltos, los calcula el ensamblador

Las instrucciones se ejecutan en secuencia.

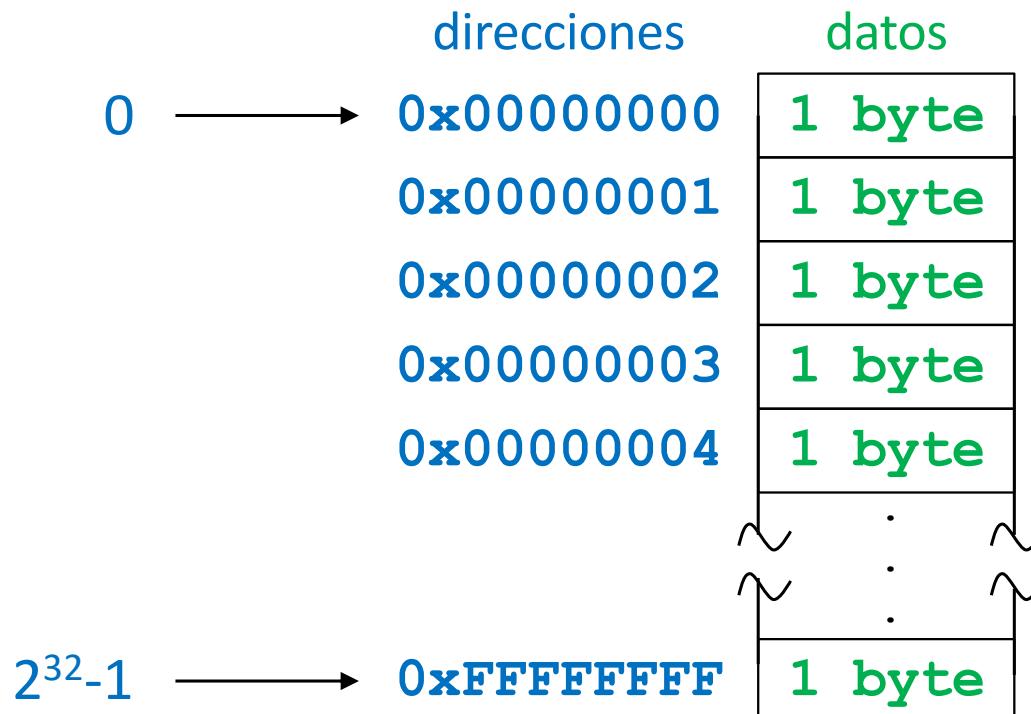
Para implementar CONDICIONALES y BUCLES, necesitamos instrucciones que rompan esa secuencia.

Las instrucciones de salto, evalúan una condición y saltan si se cumple:

- `if (R1 cond R2) goto L`

# Memoria

- La podemos ver como un vector de bytes
- Cada byte de identifica con una dirección. Las direcciones van de 0 a  $2^{32}-1$ .



# Palabra (Word)

- ❑ Dato que tiene el tamaño nativo de la arquitectura
- ❑ Normalmente, el tamaño de la palabra es igual al tamaño de registro
- ❑ MIPS: 32 bits (4 bytes)
- ❑ Ejemplos: **0x234AB6708, 0x00F30001**
- ❑ Si almacenamos **0x00F30001** en la dirección **0x234AB6708**, significa que estamos ocupando los bytes de memoria: **0x234AB6708, 0x234AB6709, 0x234AB670A y 0x234AB670B**
  
- ❑ Pero, ¿Cómo se almacenan los bytes en una palabra?

# Endianness (ordenación de bytes)

0x00F30001



❑ **Little-endian**: byte de menor peso en la dirección más baja

0x234AB6708	01
0x234AB6709	00
0x234AB670A	F3
0x234AB670B	00

❑ **Big-endian**: byte de mayor peso en la dirección más baja

0x234AB6708	00
0x234AB6709	F3
0x234AB670A	00
0x234AB670B	01

# Endianness (ordenación de bytes)

- ❑ MIPS no define una ordenación específica (puede ser little o big endian). En los procesadores MIPS se configura por hardware.
- ❑ En EC utilizaremos LITTLE-ENDIAN.
- ❑ ¡Atención! En el libro de Patterson & Hennesy utilizan big endian.
- ❑ Es muy importante saber en que tipo de máquina estamos trabajando.
  - IBM usa big endian
  - Intel usa Little endian

```
#include <stdio.h>
#include <stdint.h>
void main(void) {
    int16_t i = 5;
    int8_t *p = (int8_t *) &i;

    if (p[0] == 5) printf("Little Endian\n");
    else           printf("Big Endian\n");
}
```

Código C para averiguar el endianness



# Registros

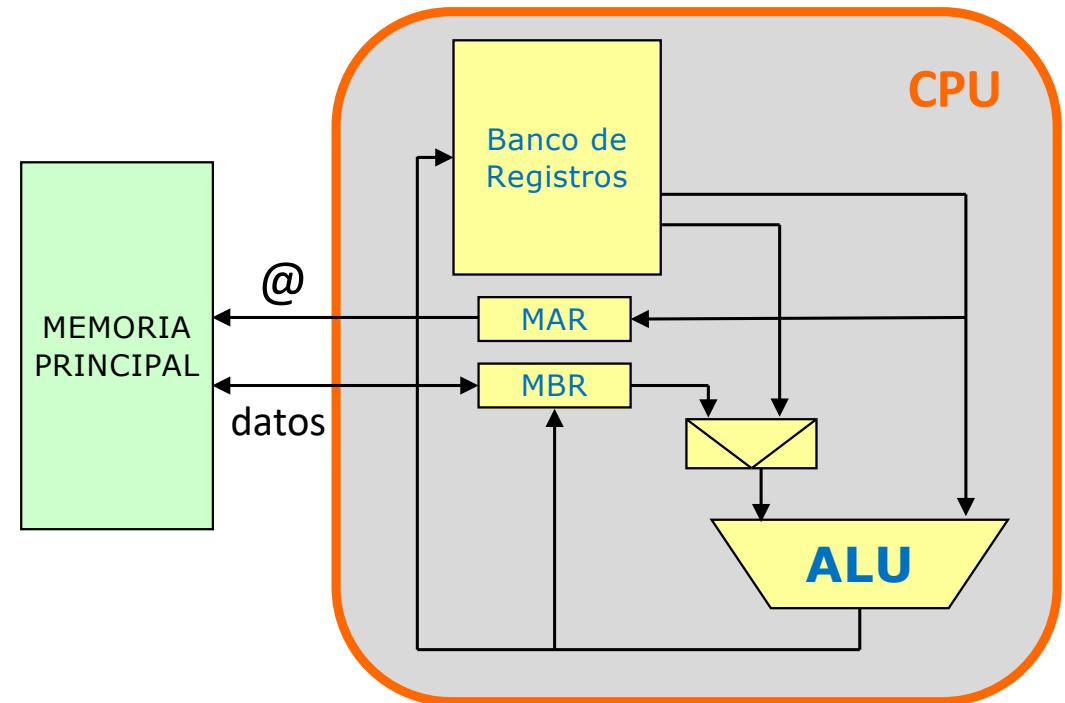
- ❑ Cuando programamos utilizamos variables.
- ❑ Las variables pueden ser:
  - Básicas: enteros, caracteres, ...
  - Estructuradas: vectores, matrices, tuplas, ...
- ❑ Cualquier variable se puede almacenar en memoria.
- ❑ Pero si queremos ir rápido, usaremos **REGISTROS**.
- ❑ Los **REGISTROS** son variables hardware en las que se puede guardar un dato de 32 bits.
- ❑ En MIPS tenemos 32 registros de 32 bits.
- ❑ Podemos referenciar cada registro por su número o por su nombre.
  - Nosotros usaremos siempre el nombre

# Registros

Número	Nombre	Descripción
\$0	\$zero	Valor 0, read only
\$1	\$at	Reservado
\$2, \$3	\$v0, \$v1	Retornos de Subrutinas
\$4, \$5, \$6, \$7	\$a0, \$a1, \$a2, \$a3	Argumentos de Subrutinas
\$8, \$9, \$10, \$11, \$12, \$13, \$14, \$15	\$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7	Valores Temporales
\$16, \$17, \$18, \$19, \$20, \$21, \$22, \$23	\$s0, \$s1, \$s2, \$s3, \$s4, \$s5, \$s6, \$s7	Variables Locales
\$24, \$25	\$t8, \$t9	Valores Temporales
\$26, \$27	\$k0, \$k1	Reservado SO
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$fp	Frame Pointer
\$31	\$ra	Return Address

# Registros vs Memoria

- ❑ Acceder a los **registros** del procesador es **MUY RÁPIDO**.
- ❑ MIPS es capaz de leer 2 registros y de escribir en 1 registro en 1 ciclo de CPU.
- ❑ Acceder a **Memoria Principal** es **MUY LENTO**.
- ❑ Una acceso a Memoria Principal (lectura/escritura), puede costar 100's de ciclos.



Siempre que podamos utilizaremos registros del procesador para guardar datos.

# Un ejemplo de traducción C → MIPS

```
int V[10] = {-1, -2, -3, -4, -5, -6, -7, -8, -9, -10};  
void main() {  
    int suma = 0;  
    int i = 0;  
  
    while (i < 10) {  
        suma = suma + V[i];  
        i++;  
    }  
}
```

Código C que suma los elementos de un vector de enteros

- El vector **V**, ha de almacenarse en Memoria.
- Las variables **suma** e **i**, se almacenarán en registros.

# Un ejemplo de traducción C → MIPS

```
.data
V: .word -1, -2, -3, -4, -5, -6, -7, -8, -9, -10 # vector V
.text
.globl main
main: li $t0, 0          # $t0 = 0 (suma = 0)
      li $t1, 0          # $t1 = 0 (i = 0)
      li $t2, 10         # $t2 = 10
loop: slt $t3, $t1, $t2    # $t3 = $t1 < $t2
      beq $t3, $zero, end   # 10 iteraciones
      la $t3, V           # $t3 = @inicial de V
      sll $t4, $t1, 2       # $t4 = 4*i (cada elemento ocupa 4 bytes)
      addu $t3, $t3, $t4     # @V[i] = @inicio V + i*4
      lw $t3, 0($t3)        # $t3 = V[i]
      addu $t0, $t0, $t3     # suma = suma + V[i]
      addiu $t1, $t1, 1       # i++
      b loop                # salto incondicional
end: jr $ra
```

Código MIPS que suma los elementos de un vector de enteros



# Variables Locales vs Variables Globales

## □ Variables Globales

- Se declaran fuera de cualquier función y pueden ser accedidas desde cualquier sitio
- Se mantienen en memoria durante toda la ejecución del programa
- Se almacenan en posiciones fijas de memoria

## □ Variables Locales

- Sólo se pueden acceder dentro del bloque donde se declaran
- Se crean al inicio de la ejecución del bloque y dejan de existir cuando finaliza
- Se reserva espacio de almacenamiento de forma dinámica (memoria o registro)

```
int V[10] = {-1, -2, ...};  
void main() {  
    int suma = 0;  
    int i = 0;  
    ...  
}
```

V es una variable Global

suma e i son variables Locales

```
.data  
V: .word -1, -2, ...  
.text  
...  
li $t0, 0 # $t0 = 0 (suma = 0)  
li $t1, 0 # $t1 = 0 (i = 0)  
...
```

# Variables Globales Ensamblador

Tipo C	MIPS	Tamaño
(unsigned) <b>char</b>	.byte	1 byte
(unsigned) <b>short</b>	.half	2 bytes
(unsigned) <b>int / long</b>	.word	4 bytes (1 word)
(unsigned) <b>long long</b>	.dword	8 bytes (2 words)

```
unsigned char a;
short x = 13;
char b = -1, c = 10;
int y = 0x13457AB2;
long long d = 0x12038034a3f00001;
```

Declaraciones en C

```
.data
a: .byte 0
x: .half 13
b: .byte -1
c: .byte 10
y: .word 0x13457AB2
d: .dword 0x12038034a3f00001
```

Declaraciones en MIPS

# Alineación de Memoria

- ❑ Existen instrucciones específicas para **leer/escribir** (**load/store**) de Memoria
  - Datos de 4 bytes (**word**): **lw** y **sw**
  - Datos de 2 bytes (**half**): **lh**, **lhu** y **sh**
  - Datos de 1 byte (**byte**): **lb**, **lbu** y **sb**
- ❑ En MIPS los accesos a Memoria han de estar alineados
  - Las instrucciones **lw** y **sw** sólo pueden acceder a direcciones múltiplo de 4.
  - Las instrucciones **lh**, **lhu** y **sh** sólo pueden acceder a direcciones múltiplo de 2.
  - Las instrucciones **lb**, **lbu** y **sb** no tienen restricciones

```
.data  
y: .word 0x13457AB2  
...  
.text  
...  
la $t0, y      #t0=@y  
lw $t1, 0($t0)  
lw $t1, 1($t0)
```

La primera instrucción **lw** funciona perfectamente. La segunda **lw** provoca un error de alineamiento que hará que el programa aborte.

# Variables Globales Ensamblador

```
unsigned char a;
short x = 13;
char b = -1, c = 10;
int y = 0x13457AB2;
long long d = 0x12038034a3f00001;
```

Declaraciones en C

```
.data
a: .byte 0
x: .half 13
b: .byte -1
c: .byte 10
y: .word 0x13457AB2
d: .dword 0x12038034a3f00001
```

Declaraciones en MIPS

Las directivas .half, .word y .dword fuerzan la alineación.

a:	00	-
x:	-	-
b:	0D	-
c:	00	-
y:	FF	d:
	0A	01
	-	00
	-	F0
	B2	A3
	7A	34
	45	80
	13	03
		12

@ alineada a 8

# Accediendo a Memoria Principal (loads)

0x1000	00
0x1001	36
0x1002	0D
0x1003	00
0x1004	FF
0x1005	0A
0x1006	A1
0x1007	32
0x1008	B2
0x1009	7A
0x100A	45
0x100B	13
0x100C	A3
0x100D	34
0x100E	03
0x100F	83

```
li $t0, 0x1000
lw $t1, 4($t0)      # t1=0x32A10AFF
lbu $t2, 11($t0)    # t1=0x00000013
lb $t3, 11($t0)     # t1=0x00000013
lb $t4, 12($t0)     # t1=0xFFFFFFF3
lhu $t4, 8($t0)      # t1=0x00007AB2
lh $t5, 8($t0)       # t1=0x00007AB2
lh $t6, 14($t0)     # t1=0xFFFF8303
lw $t1, 6($t0)       # ERROR ALINEACIÓN
lh $t1, 3($t0)       # ERROR ALINEACIÓN
```

# Accediendo a Memoria Principal (stores)

0x1000	00
0x1001	36
0x1002	0D
0x1003	00
0x1004	FF
0x1005	0A
0x1006	A1
0x1007	32
0x1008	B2
0x1009	7A
0x100A	45
0x100B	13
0x100C	A3
0x100D	34
0x100E	03
0x100F	83

```
li $t0,0x1000  
li $t1,0x11223344  
  
sw $t1,0($t0)  
  
sh $t1,6($t0)  
sh $t0,10($t0)  
  
sb $t1,0xF($t0)  
  
sw $t1,1($t0) #ERROR
```

0x1000	44
0x1001	33
0x1002	22
0x1003	11
0x1004	FF
0x1005	0A
0x1006	44
0x1007	33
0x1008	B2
0x1009	7A
0x100A	00
0x100B	01
0x100C	A3
0x100D	34
0x100E	03
0x100F	44

# Declaración y Alineación de Variables Globales

- ❑ **Alineación Automática.** Por defecto las variables globales se ubican en direcciones múltiplo de su tamaño.
- ❑ Las directivas **.half**, **.word** y **.dword** fuerzan la alineación correcta.

## Directivas .byte .half .word .dword

	Directivas .byte .half .word .dword
<b>.byte n</b>	Reserva 1 byte y lo inicializa a n
<b>.half n</b>	Reserva 1 halfword (2 bytes) alineado a una dirección múltiplo de 2 y lo inicializa a n.
<b>.word n</b>	Reserva 1 word (4 bytes) alineado a una dirección múltiplo de 4 y lo inicializa a n.
<b>.dword n</b>	Reserva 1 doubleword (8 bytes) alineado a una dirección múltiplo de 8 y lo inicializa a n.

# Declaración y Alineación de Variables Globales

- ¿Qué pasa con los vectores?

```
int v[5] = {1, 2, 3, 4, 5};
```

Declaración en C

```
v: .word 1, 2, 3, 4, 5
```

Declaración en MIPS

- Y ¿si el vector no está inicializado?

- La directiva **.space n** reserva n bytes en memoria inicializados a cero.

```
char F[450];
int V[100];
```

Declaración en C

```
F: .space 450
V: .space 100*4
```

Declaración en MIPS

iPROBLEMA! La directiva **.space** no puede alinear.



# Declaración y Alineación de Variables Globales

- Para alinear de forma explícita podemos utilizar la directiva `.align n`

Directivas <code>.align .space</code>	
<code>.align n</code>	Ubica el próximo dato en una dirección múltiplo de $2^n$
<code>.space m</code>	Reserva m bytes y los inicializa a 0

```
short x;  
int v[100];
```

Declaración en C

```
x: .half 0      # alineado a 2 bytes  
.align 2  
v: .space 100*4 # alineado a 4 bytes
```

Declaración en MIPS

# Problema 2.6

- 2.6. Indica quin és el contingut de memòria a partir de l'adreça 0x10010000 i a nivell de byte si tenim la següent declaració de variables globals.

	.data	0x10010000		10	0x10010010	D:	42	0x10010020	F:	43
	.byte 0x10	0x10010001		XX	0x10010011		00	0x10010021		XX
A:	.half -5	0x10010002	A:	FB	0x10010012		43	0x10010022	G:	66
B:	.word -1, 67	0x10010003		FF	0x10010013		00	0x10010023		00
C:	.byte -4, '5', 6	0x10010004	B:	FF	0x10010014		XX	0x10010024		XX
D:	.half 66, 67	0x10010005		FF	0x10010015		XX	0x10010025		XX
E:	.dword 0x5799	0x10010006		FF	0x10010016		XX	0x10010026		XX
F:	.byte 'C'	0x10010007		FF	0x10010017		XX	0x10010027		XX
G:	.half 0x66	0x10010008		43	0x10010018	E:	99	0x10010028	H:	43
H:	.dword 579	0x10010009		00	0x10010019		57	0x10010029		02
		0x1001000A		00	0x1001001A		00	0x1001002A		00
		0x1001000B		00	0x1001001B		00	0x1001002B		00
		0x1001000C	C:	FC	0x1001001C		00	0x1001002C		00
		0x1001000D		35	0x1001001D		00	0x1001002D		00
		0x1001000E		06	0x1001001E		00	0x1001002E		00
		0x1001000F		XX	0x1001001F		00	0x1001002F		00

¿Cómo se representa  
-5, 579, '5', 'C' en  
binario?

# Representación de Números Naturales en Binario

□ Para representar naturales usamos un sistema convencional en base r:

- $X = (x_{n-1}x_{n-2} \dots x_2x_1x_0)$
- $x_i \in \{0, 1, 2, \dots, r - 1\}$
- $r > 1$
- $X_u = \sum_{i=0}^{n-1} x_i \cdot r^i$

□ Como trabajamos en **BINARIO**, entonces:

- $x_i \in \{0, 1\}$
- $r = 2$
- $n = 8, 16, 32, 64$

□ Si tenemos  $X=10011101$  (0x9D)

- $X_u = 1 \cdot 2^7 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = 128 + 16 + 8 + 4 + 1 = 157$

# Representación de Números Naturales en Binario

- Un punto fundamental, es que el número de bits que tenemos para representar un número está limitado
  - $n = 8, 16, 32, 64$
- Eso significa, que para un valor determinado de  $n$ , tenemos un rango de valores representables con ese número de bits.

El rango de números naturales de  $n$  bits es  $X_u \in [0..2^n-1]$

<b><math>n</math></b>	<b>Mínimo</b>	<b>Máximo (<math>2^n-1</math>)</b>
8	0	255
16	0	65.535
32	0	4.294.967.295
64	0	18.446.744.073.709.551.615

# Representación de Números Naturales en Binario

- Operación típica, representar un Número Natural  $X_u$  en binario con n bits:

iter	$X_u$	$x_i$
0	$Q_0 = X_u / 2$	$x_0 = X_u \% 2$
1	$Q_1 = Q_0 / 2$	$x_1 = Q_0 \% 2$
2	$Q_2 = Q_1 / 2$	$x_2 = Q_1 \% 2$
...	...	...
$n-1$	$Q_{n-1} = Q_{n-2} / 2$	$x_{n-1} = Q_{n-2} \% 2$

¿Qué ocurre si  $Q_{n-1}$  no vale 0?

$X_u$  no se puede representar con n bits

iter	$X_u = 32$	$x_i$
0	$32 / 2 = 17$	$x_0 = 32 \% 2 = 0$
1	$17 / 2 = 8$	$x_1 = 17 \% 2 = 1$
2	$8 / 2 = 4$	$x_2 = 8 \% 2 = 0$
3	$4 / 2 = 2$	$x_3 = 4 \% 2 = 0$
4	$2 / 2 = 1$	$x_4 = 2 \% 2 = 0$
5	$1 / 2 = 0$	$x_5 = 1 \% 2 = 1$
6	$0 / 2 = 0$	$x_6 = 0 \% 2 = 0$
7	$0 / 2 = 0$	$x_7 = 0 \% 2 = 0$

$X_u = 32 \quad X = 00100010 = 0x22$

# Representación de Números Naturales en Binario

- Operación típica, pasar un número natural de n a m bits (m>n) :

Extender la representación es poner 0's a la izquierda

n	X (binario)	X (hex)
8	0010 0010	0x22
16	0000 0000 0010 0010	0x0022
32	0000 0000 0000 0000 0000 0000 0010 0010	0x00000022
64	0000 0000 0000 ... 0000 0000 0000 0000 0010 0010	0x0000000000000022

# Representación de Números Enteros en ca2

## □ Representación de enteros en complemento a 2 (ca2):

- X, valor entero que representamos (valor implícito)
- $X_u$ , valor natural asociado (valor explícito)
- Representación

$X = (x_{n-1}x_{n-2} \dots x_2x_1x_0)$ , vector de bits,  $x_i \in \{0, 1\}$

- Función de interpretación:

$$X = X_u - x_{n-1} \cdot 2^{n-1}$$

- Función de representación:

$$X_u = \begin{cases} x & \text{si } x \geq 0 \\ x + 2^n & \text{si } x < 0 \end{cases}$$

Los valores enteros están  
“desordenados”

El bit  $x_{n-1}$  indica el signo

Se utiliza el mismo +/- que con  
números naturales

$x_3x_2x_1x_0$	$X_u$	X
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

# Representación de Números Enteros en ca2

- El rango de representación de enteros, de n bits, en **complemento a 2** (ca2) es:

$$X \in [-2^{n-1} .. 2^{n-1}-1]$$

- Hay un valor más en negativo que en positivo

n	Mínimo (-2 <sup>n-1</sup> )	Máximo (2 <sup>n-1</sup> -1)
8	-128	127
16	-32.768	32.767
32	-2.147.483.648	2.147.483.647
64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

# Representación de Números Enteros en ca2

- Extensión de la representación, el mismo valor pero con más bits

Hay que extender el bit de signo ( $x_{n-1}$ ) a la izquierda

$x_3x_2x_1x_0$	$X_u$	$X$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

$x_3x_2x_1x_0$	$X_u$	$X$
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

$x_4x_3x_2x_1x_0$	$X_u$	$X$
00000	0	0
00001	1	1
00010	2	2
00011	3	3
00100	4	4
00101	5	5
00110	6	6
00111	7	7
01000	8	8
01001	9	9
01010	10	10
01011	11	11
01100	12	12
01101	13	13
01110	14	14
01111	15	15
10000	16	-16
10001	17	-15
10010	18	-14
10011	19	-13
10100	20	-12
10101	21	-11
10110	22	-10
10111	23	-9
11000	24	-8
11001	25	-7
11010	26	-6
11011	27	-5
11100	28	-4
11101	29	-3
11110	30	-2
11111	31	-1

# Problema

- Representar en complemento a 2 (n=8 bits) los siguientes números enteros:

x	binario	Hexadecimal
36	0010 0100	0x24
-97	1001 1111	0x9F
109	0110 1101	0x6D
132	NO representable	-
-128	1000 0000	0x80
43	0010 1011	0x2B
-34	1101 1110	0xDE
-131	NO representable	-
-1	1111 1111	0xFF

# Representación Números Enteros en Complemento a 1

$x_4x_3x_2x_1x_0$	$x_u$	$x$	$x_4x_3x_2x_1x_0$	$x_u$	$x$
00000	0	0	10000	16	-15
00001	1	1	10001	17	-14
00010	2	2	10010	18	-13
00011	3	3	10011	19	-12
00100	4	4	10100	20	-11
00101	5	5	10101	21	-10
00110	6	6	10110	22	-9
00111	7	7	10111	23	-8
01000	8	8	11000	24	-7
01001	9	9	11001	25	-6
01010	10	10	11010	26	-5
01011	11	11	11011	27	-4
01100	12	12	11100	28	-3
01101	13	13	11101	29	-2
01110	14	14	11110	30	-1
01111	15	15	11111	31	0

Los valores están “desordenados”

El bit  $x_{n-1}$  indica el signo

Hay 2 ceros

NO se utiliza

Se utiliza el mismo +/- que con números naturales

# Representación Números Enteros en Signo y Magnitud

$x_4x_3x_2x_1x_0$	$x_u$	$x$
00000	0	0
00001	1	1
00010	2	2
00011	3	3
00100	4	4
00101	5	5
00110	6	6
00111	7	7
01000	8	8
01001	9	9
01010	10	10
01011	11	11
01100	12	12
01101	13	13
01110	14	14
01111	15	15

$x_4x_3x_2x_1x_0$	$x_u$	$x$
10000	16	0
10001	17	-1
10010	18	-2
10011	19	-3
10100	20	-4
10101	21	-5
10110	22	-6
10111	23	-7
11000	24	-8
11001	25	-9
11010	26	-10
11011	27	-11
11100	28	-12
11101	29	-13
11110	30	-14
11111	31	-15



Los valores están “ordenados”

El bit  $x_{n-1}$  ES el signo

Hay 2 ceros

¿Se utiliza?

El +/- es muy ineficiente

# Representación de Números Enteros en exceso a $2^{n-1}-1$

## □ Representación de enteros en **exceso a $2^{n-1}-1$** :

- Función de interpretación:

$$X = X_u - (2^{n-1} - 1)$$

- Función de representación:

$$X_u = X + (2^{n-1} - 1)$$

Los valores están “ordenados”

El número más pequeño (-15) corresponde con el 0...000, y el número más grande (16) corresponde con el 1...111.

Exceso a 15

$X_4X_3X_2X_1X_0$	$x_u$	$x$	$X_4X_3X_2X_1X_0$	$x_u$	$x$
00000	0	-15	10000	16	1
00001	1	-14	10001	17	2
00010	2	-13	10010	18	3
00011	3	-12	10011	19	4
00100	4	-11	10100	20	5
00101	5	-10	10101	21	6
00110	6	-9	10110	22	7
00111	7	-8	10111	23	8
01000	8	-7	11000	24	9
01001	9	-6	11001	25	10
01010	10	-5	11010	26	11
01011	11	-4	11011	27	12
01100	12	-3	11100	28	13
01101	13	-2	11101	29	14
01110	14	-1	11110	30	15
01111	15	0	11111	31	16

# Representación de Números Enteros en exceso a $2^{n-1}-1$

- El rango de representación de enteros en **exceso a  $2^{n-1}-1$**  es

$$X \in [-(2^{n-1}-1) \dots 2^{n-1}]$$

- Hay un valor más en positivo que en negativo

n		Mínimo $-(2^{n-1}-1)$	Máximo $(-2^{n-1}+1)$
8	exceso a 127	-127	128
16	exceso a $2^{15}-1$	-32.767	32.768
32	exceso a $2^{31}-1$	-2.147.483.647	2.147.483.648
64	exceso a $2^{63}-1$	-9.223.372.036.854.775.807	9.223.372.036.854.775.808

# Problema 2.11 y 2.12

N	S y M	ca1	ca2	exceso a 7
8	N.R.	N.R.	N.R.	1111
7	0111	0111	0111	1110
6	0110	0110	0110	1101
5	0101	0101	0101	1100
4	0100	0100	0100	1011
3	0011	0011	0011	1010
2	0010	0010	0010	1001
1	0001	0001	0001	1000
0	0000 y 1000	0000 y 1111	0000	0111
-1	1001	1110	1111	0110
-2	1010	1101	1110	0101
-3	1011	1100	1101	0100
-4	1100	1011	1100	0011
-5	1101	1010	1011	0010
-6	1110	1001	1010	0001
-7	1111	1000	1001	0000
-8	N.R.	N.R.	1000	N.R.

## Ca2

Función de interpretación:  $x = x_u - X_{n-1} \cdot 2^n$

Función de representación:  $x_u = \begin{cases} x & \text{si } x \geq 0 \\ x + 2^n & \text{si } x < 0 \end{cases}$

## Exceso a $2^{n-1}-1$

Función de interpretación:  $x = x_u - (2^{n-1}-1)$

Función de representación:  $x_u = x + (2^{n-1}-1)$

	Valor Minímo	Valor Máximo
SyM	$-2^{n-1}-1$	$2^{n-1}-1$
ca1	$-2^{n-1}-1$	$2^{n-1}-1$
ca2	$-2^{n-1}$	$2^{n-1}-1$
Exc a $2^{n-1}-1$	$-2^{n-1}-1$	$2^{n-1}$



# Repaso de Representaciones Numéricas de Enteros

- ❑ Para representar Enteros utilizaremos siempre complemento a 2:
  - El sumador/restador en complemento a 2, es el mismo que usamos con Naturales.
- ❑ La representación en complemento a 1 y signo y magnitud no se usan para representar enteros:
  - Tienen 2 ceros.
  - El sumador/restador en signo y magnitud es "muy complejo".
- ❑ En la representación de números reales se utiliza notación científica ( $13,45 \cdot 10^{-5}$ ):
  - Para representar la mantisa (13,45) se utiliza signo y magnitud.
  - Para representar el exponente (-5) se usa exceso a  $2^{n-1}-1$ .

# Problema

- Traducid de C a MIPS, está porción de código (variables globales). Mostrad cómo quedan almacenadas en Memoria.

```
unsigned char a;  
short x = -1;  
char b = -109, c = 47;  
int y = 356;  
short s = 119;  
long long d = -519;
```

C

```
.data  
a: .byte 0  
x: .half -1      # 0xFFFF  
b: .byte -109    # 0x93  
c: .byte 47       # 0x2F  
y: .word 356      # 0x00000164  
s: .half 119      # 0x0077  
d: .dword -519    # 0xFFFFFFFFFFFFFD9
```

MIPS

a:	00
	-
x:	FF
	FF
b:	93
c:	2F
	-
	-
y:	64
	01
	00
	00

s:	77
	00
	-
	-
d:	F9
	FD
	FF

# Instrucciones

- La mayoría de instrucciones tienen 3 operandos:

```
INST op1,op2,op3      # op1 ← op2 (INST) op3
```

- Principio de Diseño 1: **Simplicity favors Regularity**
- A la forma en cómo se especifica un operando lo llamamos **MODO de DIRECCIONAMIENTO**
- Empezaremos trabajando con 3 Modos de Direcciónamiento
  - **Registro**, el Operando está en uno de los 32 registros del procesador
    - ✓ **MUY RÁPIDO**, lo usaremos siempre que podamos
  - **Inmediato**, el Operando es una constante que se almacena en la propia instrucción
  - **Memoria**, el Operando es una posición de memoria que identificamos con su dirección

# Registros

Número	Nombre	Descripción
\$0	\$zero	Valor 0, read only
\$1	\$at	Reservado
\$2, \$3	\$v0, \$v1	Retornos de Subrutinas
\$4, \$5, \$6, \$7	\$a0, \$a1, \$a2, \$a3	Argumentos de Subrutinas
\$8, \$9, \$10, \$11, \$12, \$13, \$14, \$15	\$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7	Valores Temporales
\$16, \$17, \$18, \$19, \$20, \$21, \$22, \$23	\$s0, \$s1, \$s2, \$s3, \$s4, \$s5, \$s6, \$s7	Variables Locales
\$24, \$25	\$t8, \$t9	Valores Temporales
\$26, \$27	\$k0, \$k1	Reservado SO
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$fp	Frame Pointer
\$31	\$ra	Return Address

# Las Primeras Instrucciones

```
addu $t1, $t2, $t3 # $t1 = $t2 + $t3
```

- En **addu**, los operandos sólo pueden ser registros

```
addiu $t1, $t2, 29 # $t1 = $t2 + 29
```

- El inmediato (29) es un número entero ca2 de 16 bits.

Instrucciones de suma y resta		
<b>addu rd, rs, rt</b>	$rd = rs + rt$	
<b>addiu rd, rs, imm16</b>	$rd = rs + \text{SignExt}(imm16)$	Inmediato de 16 bits en ca2
<b>subu rd, rs, rt</b>	$rd = rs - rt$	

# Un ejemplo de uso

```
void main() {  
    int f, g, h, i, j;  
    . . .  
    f = (g + h) - (i + j) - 45;  
    . . .  
}
```

C

- Suponiendo que f, g, h, i, j están en los registros \$t0, \$t1, \$t2, \$t3 y \$t4 respectivamente

```
addu $t0, $t1, $t2      # $t0 = g + h  
addu $t5, $t3, $t4      # $t5 = i + j  
subu $t0, $t0, $t5      # f = (g+h) - (i+j)  
addiu $t0, $t0, -45     # f = (g+h) - (i+j) - 45
```

MIPS

# Operación típica: Copia entre Registros

```
addiu $t1, $t2, 0      # $t1 = $t2
```

- ❑ Alternativa:

```
addu $t1, $zero, $t2    # $t1 = $t2
```

- ❑ Alternativa (pseudoinstrucción o macro):

```
move $t1, $t2    # equivale a addu $t1, $t2, $zero
```

- Las macros se expanden en 1 o más instrucciones MIPS
- Facilitan la lectura y Depuración

move (pseudoinstrucción)

move rdest, rsrc

rdest = rsrc

addu rdest, rsrc, \$zero

# Operación típica: Inicializar un Registro con un literal

- Algunas instrucciones admiten operandos inmediatos de 16 bits (ca2).

Permite asignar un inmediato a un registro:

```
addiu $t1, $zero, 100      # $t1 = 100
```

```
addiu $t1, $zero, -45      # $t1 = -45
```

- Pero, si el operando es de 32 bits, hemos de hacer otra cosa: **usar la macro li**

```
li $t1, 0x44332211      # lui $at, 0x4433  
                        # ori $t1, $at, 0x2211
```

- La macro li, también se puede usar con operandos de 16 bits:

```
li $t1, 0x2211      # addiu $t1, $zero, 0x2211
```

# Operación típica: Inicializar un Registro con un literal

li (pseudoinstrucción)		
<b>li rdest, imm16</b>	rdest = sigext(imm16)	<b>addiu rdest, \$zero, imm16</b>
<b>li rdest, imm32</b>	rdest=imm32	<b>lui \$at, hi(imm32) ori rdest. \$at, lo(imm32)</b>

- ¡Mucho cuidado con las macros! MARS (esimulador de MIPS que usaremos en el Laboratorio) siempre trabaja con literales de 32 bits

```
li $t1, 0xFFFF # Mars lee 0x0000FFFF
```

# Cómo gestionamos las constantes

- En C es muy habitual definir constantes:

```
#define N 10  
#define M -67
```

C

- También podemos usar constantes en MIPS:

```
.eqv N 10  
.eqv M -67  
.data  
a: .word 0  
.text  
.globl main  
main: li $t0, N          # $t0 = N  
      addiu $t0, $t0, M    # $t0 = N + M
```

MIPS



# Y ¿Cómo accedemos a Memoria?

- Cuando accedemos a memoria podemos:
  - LEER de Memoria (load): **dat**  $\leftarrow$  **M[dir]**
  - ESCRIBIR en Memoria (store): **M[dir]**  $\leftarrow$  **dat**
- Lo primero que necesitamos es la dirección, usaremos la macro **la** (load address)

```
.data  
num: .word 0  
...  
.text  
...  
la $t0, num      # t0  $\leftarrow$  &num = 0x010010000
```

MIPS

## la (pseudoinstrucción)

la rdest, etiq

rdest =  
@etiq

lui \$at, hi(@etiq)  
ori rdest, \$at, lo(@etiq)

# ¿Cómo accedemos a Memoria? Instrucciones load y store

```
int a;  
void main() {  
    int b;          // b es $t1  
    b = a;  
    a = 2*b  
}
```

C

```
.data  
a: .word 0  
.text  
.globl main  
main: la $t0,a           # t0 ← &a  
      lw $t1,0($t0)       # t1 ← M[$t0+0]  
      addu $t1,$t1,$t1     # t1 ← 2*b  
      sw $t1,0($t0)       # M[t0+0] ← t1
```

MIPS



# Operandos en Memoria. Instrucciones load y store

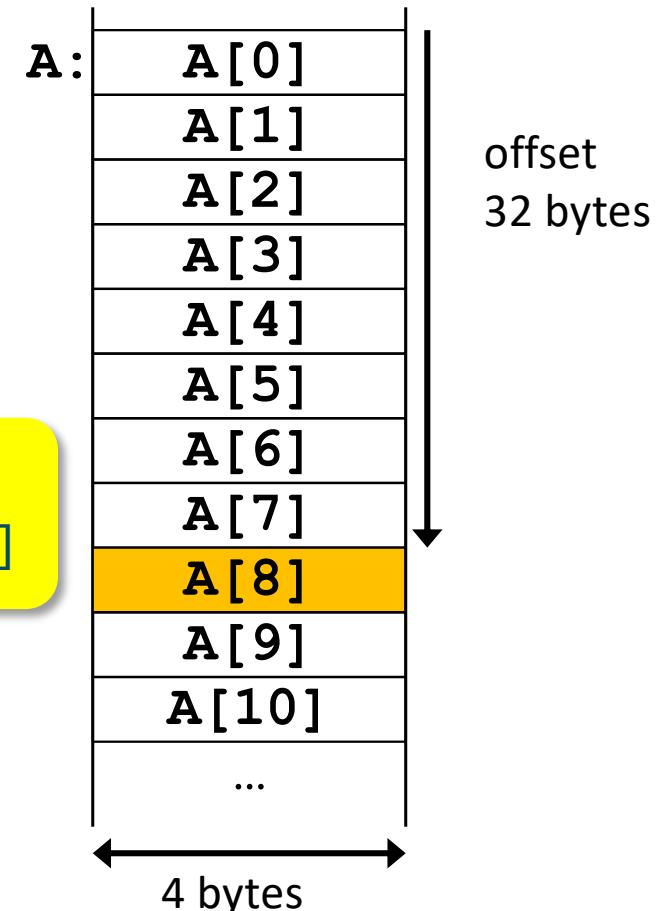
```
int A[100];
void main() {
    int g, h; // en $t1 y $t2
    ...
    g = h + A[8];
}
```

C

```
.align 2
A: .space 400
...
la $t3, A          # t3 = @A[0]
lw $t0, 32($t3)    # t0 = Mem[@A+32]
addu $t1, $t2, $t0 # g = h + A[8]
```

MIPS

Si @A es la dirección de A[0],  
@A+32 es la dirección de A[8]



# Operandos en Memoria. Instrucciones load y store

```
int A[100];
void main() {
    int h; // en $t2
    ...
    A[12] = h + A[8];
}
```

C

Si @A es la dirección de A[0],  
@A+32 es la dirección de A[8],  
@A+48 es la dirección de A[12]

```
.align 2
A: .space 400
...
la $t3, A           # t3 = @A[0]
lw $t0, 32($t3)    # t0 = Mem[@A+32]
addu $t0, $t2, $t0   # t0 = h + A[8]
sw $t0, 48($t3)    # A[12] = t0
```

MIPS



# Operandos en Memoria. Instrucciones load y store

```
char v[100];  
void main() {  
    char c; // en $t1  
    ...  
    c = v[8] op v[6];  
}
```

C

Si @v es la dirección de v[0],  
@v+8 es la dirección de v[8],  
cada char ocupa 1 byte

PROBLEMA!, \$t2 ocupa 4 bytes, y v[8] ocupa 1 byte

```
la $t2,v  
lb $t1, 8($t2) # $t1 = 0x00000011  
lb $t2, 6($t2) # $t1 = 0xFFFFFCC  
lbu $t1, 8($t2) # $t1 = 0x00000011  
lbu $t2, 6($t2) # $t1 = 0x000000CC
```

MIPS

lb, extiende signo  
lbu, extiende ceros

Trabajo para el  
PROGRAMADOR



# Operandos en Memoria. Instrucciones load y store

Instrucciones load / store		
<b>lw rt off16(rs)</b>	$rt = M_w[rs + \text{SignExt}(off16)]$	Load word
<b>lh rt off16(rs)</b>	$rt = \text{SignExt}(M_h[rs + \text{SignExt}(off16)])$	Load half (extiende signo)
<b>lhu rt off16(rs)</b>	$rt = M_h[rs + \text{SignExt}(off16)]$	Load half (extiende ceros)
<b>lb rt off16(rs)</b>	$rt = \text{SignExt}(M_b[rs + \text{SignExt}(off16)])$	Load byte (extiende signo)
<b>lbu rt off16(rs)</b>	$rt = M_b[rs + \text{SignExt}(off16)]$	Load byte (extiende ceros)
<b>sw rt off16(rs)</b>	$M_w[rs + \text{SignExt}(off16)] = rt$	Store word
<b>sh rt off16(rs)</b>	$M_h[rs + \text{SignExt}(off16)] = rt_{15:0}$	Store Half
<b>sb rt off16(rs)</b>	$M_b[rs + \text{SignExt}(off16)] = rt_{7:0}$	Store Byte

# Operandos en Memoria. Instrucciones load y store

- **ALINEACIÓN.** Las **DIRECCIONES** utilizadas en las instrucciones de leer/escribir en **MEMORIA** han de ser **MÚLTIPLO** del **NÚMERO de BYTES** que se quiera leer/escribir.
  - En las instrucciones **lw** y **sw**, las direcciones han de ser **múltiplo de 4**.
  - En las instrucciones **lh**, **lhu** y **sh**, las direcciones han de ser **múltiplo de 2**.
  - En las instrucciones **lb**, **lbu** y **sb**, no hay ninguna restricción.

```
.data  
x: .word -11          # x está alineado a 4B  
.text  
la $t0,x  
lw $t1,1($t0) # accede a @x+1
```

MIPS

Provoca una excepción por acceso NO ALINEADO

# Representación de Caracteres

- ❑ Todos los Lenguajes de Alto Nivel incluyen el tipo carácter para trabajar con textos.

```
char v[100];  
char a = 'A';
```

- ❑ Obviamente han de tener algún tipo de codificación.
- ❑ **Nosotros utilizaremos ASCII**, la primera versión del Código ASCII es de 1963 (telegrafía).
- ❑ No es un problema trivial.
  - Hay muchos alfabetos diferentes: cirílico, hebreo, árabe, chino, coreano, japonés, ...
  - Una operación típica con textos (palabras) es la ordenación alfabética
    - ✓ Á À Ä á à ä, son la misma letra pero las codificaciones son diferentes [**iPROBLEMA!**]
- ❑ Actualmente se utiliza **UNICODE**:
  - Todos los caracteres del mundo, en todos los idiomas, tienen asignado un número. Ese número es el UNICODE.
  - El UNICODE hay que codificarlo. Diferentes codificaciones:UTF-8, UTF-16, UTF-32

# Representación de Caracteres ASCII

Caracteres ASCII de control			Caracteres ASCII imprimibles								ASCII extendido							
00	NULL	(carácter nulo)	32	espacio	64	@	96	`	128	Ç	160	á	192	ł	224	ó		
01	SOH	(inicio encabezado)	33	!	65	A	97	a	129	ü	161	í	193	ł	225	þ		
02	STX	(inicio texto)	34	"	66	B	98	b	130	é	162	ó	194	ł	226	ô		
03	ETX	(fin de texto)	35	#	67	C	99	c	131	â	163	ú	195	ł	227	ò		
04	EOT	(fin transmisión)	36	\$	68	D	100	d	132	ã	164	ñ	196	—	228	ö		
05	ENQ	(consulta)	37	%	69	E	101	e	133	à	165	Ñ	197	+	229	ö		
06	ACK	(reconocimiento)	38	&	70	F	102	f	134	á	166	º	198	ä	230	µ		
07	BEL	(timbre)	39	'	71	G	103	g	135	ç	167	º	199	À	231	þ		
08	BS	(retroceso)	40	(	72	H	104	h	136	è	168	¿	200	ł	232	þ		
09	HT	(tab horizontal)	41	)	73	I	105	i	137	ë	169	®	201	ł	233	ú		
10	LF	(nueva línea)	42	*	74	J	106	j	138	è	170	¬	202	ł	234	û		
11	VT	(tab vertical)	43	+	75	K	107	k	139	í	171	½	203	ł	235	ú		
12	FF	(nueva página)	44	,	76	L	108	l	140	î	172	¼	204	ł	236	ý		
13	CR	(retorno de carro)	45	-	77	M	109	m	141	í	173	i	205	=	237	ÿ		
14	SO	(desplaza afuera)	46	.	78	N	110	n	142	Ä	174	«	206	‡	238	—		
15	SI	(desplaza adentro)	47	/	79	O	111	o	143	Á	175	»	207	¤	239	·		
16	DLE	(esc. vínculo datos)	48	0	80	P	112	p	144	É	176	…	208	ð	240	≡		
17	DC1	(control disp. 1)	49	1	81	Q	113	q	145	æ	177	„	209	đ	241	±		
18	DC2	(control disp. 2)	50	2	82	R	114	r	146	Æ	178	„	210	é	242	—		
19	DC3	(control disp. 3)	51	3	83	S	115	s	147	ô	179	—	211	é	243	¾		
20	DC4	(control disp. 4)	52	4	84	T	116	t	148	ö	180	—	212	é	244	¶		
21	NAK	(conf. negativa)	53	5	85	U	117	u	149	ò	181	á	213	í	245	§		
22	SYN	(inactividad sinc)	54	6	86	V	118	v	150	û	182	â	214	í	246	÷		
23	ETB	(fin bloque trans)	55	7	87	W	119	w	151	ù	183	à	215	í	247	·		
24	CAN	(cancelar)	56	8	88	X	120	x	152	ÿ	184	©	216	í	248	·		
25	EM	(fin del medio)	57	9	89	Y	121	y	153	Ö	185	—	217	í	249	·		
26	SUB	(sustitución)	58	:	90	Z	122	z	154	Ü	186	—	218	í	250	·		
27	ESC	(escape)	59	:	91	[	123	{	155	ø	187	—	219	■	251	·		
28	FS	(sep. archivos)	60	<	92	\	124		156	£	188	—	220	■	252	·		
29	GS	(sep. grupos)	61	=	93	]	125	}	157	Ø	189	¢	221	—	253	·		
30	RS	(sep. registros)	62	>	94	^	126	~	158	×	190	¥	222	■	254	■		
31	US	(sep. unidades)	63	?	95	—			159	f	191	—	223	■	255	nnbsp		
127	DEL	(suprimir)																

ASCII 7 bits

ASCII 8 bits

# Representación de Caracteres ASCII

## □ Usaremos ASCII 7 bits

- 128 caracteres (33 de control)
- Almacenado en 1 byte, bit de mayor peso = 0.
- Faltan: vocales con acentos, ñ, ç, ., € (por eso se creó el ASCII extendido)

## □ Propiedades de la codificación ASCII

- Símbolos decimales: '0' al '9', ordenados a partir del 48 (0x30) ['0'+1='1']
- Mayúsculas: 'A' a 'Z' ordenadas a partir del 65 (0x41) ['A'+1='B']
- Minúsculas: 'a' a 'z' ordenadas a partir del 97 (0x61) ['a'+1='b']
- Carácter null (0 o '\0' en C) es el 0x00
- Espacio en blanco '' es el 0x20
- Salto de línea '\n' es el 0x0A

# Operaciones típicas con Caracteres

## □ Conversión Mayúsculas/Minúsculas

- $C_{MIN} \leftarrow C_{MAY} + 32;$

## □ Conversión Dígito ASCII / valor Numérico

- $CaracterDigito \leftarrow Valor + 48;$

```
char A, b, n; int x;
```

```
A = b + ('A' - 'a'); // MAY ← MIN
b = A + ('a' - 'A'); // MIN ← MAY
x = n - '0';          // VAL ← DIG
n = x + '0';          // DIG ← VAL
```

C

No es buena idea utilizar los valores de la codificación directamente.

No es necesario conocer los códigos ASCII

Algo equivalente se puede hacer en MIPS

# Operaciones típicas con Caracteres

## □ Declaración de variables de tipo carácter

```
char c = 'R';
```

C

```
c: .byte 'R'
```

MIPS

- Como el bit de mayor peso es cero no cambia nada si lo declaramos unsigned.

```
.data  
c1: .byte 'A'  
c2: .byte 0  
.text  
.globl main  
main:  
    la $t0,c1  
    lb $t1,0($t0)  
    addiu $t1,$t1,'a'-'A'  
    ...
```

MIPS

```
...  
li $t1,7  
addiu $t1,$t1,'0'  
la $t0,c2  
sb $t1,0($t0)  
...
```

```
c1 = minusc(c1);  
c2 = digit($t1);
```

C

# Formato de instrucciones

## ❑ **Stored-Program (Programa Almacenado en memoria).**

Igual que representamos los datos, hemos de guardar las instrucciones en memoria, pero con una codificación especial.

- ❑ El formato de instrucción determina la representación de las instrucciones.
  - ❑ Idealmente queremos tener el menor número de formatos diferentes.
  - ❑ **Good design demands good compromises.**
- 
- ❑ En una máquina RISC:
    - tenemos pocos formatos de instrucciones,
    - son instrucciones de tamaño fijo,
    - simplifica mucho la implementación del procesador.

# Formato de instrucciones MIPS

- 3 formatos: R (register), I (immediate) y J (jumps)

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	opcode	rs	rt	rd	shamt	func
I	opcode	rs	rt		imm16	
J	opcode			target		

- Ejemplos:

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
<b>addu rd, rs, rt</b>	R	0x00	rs	rt	rd	0x00
<b>sra rd, rt, shamt</b>	R	0x00	rs	rt	rd	shamt
<b>addiu rt, rs, imm16</b>	I	0x08	rs	rt		imm16
<b>lui rt imm16</b>	I	0x0f	rs	rt		imm16
<b>lw rt, offset16(rs)</b>	I	0x23	rs	rt		offset16
<b>j target</b>	J	0x02		target		

# Formato de instrucciones MIPS

- Codificar en binario

**addu \$t0, \$s2, \$zero # \$t0 = \$s2**

- Opcode = 0x00 = 000000
- rs = \$s2 = \$18 = 10010 (\$s0..\$s7 → \$16..\$23)
- rt = zero = \$0 = 00000
- rd = \$t0 = \$8 = 01000 (\$t0..\$t7 → \$8..\$15)
- shamt = 00000
- funct = 0x21 = 100001

000000|10010|00000|01000|00000|100001 → 00000010010000000100000000100001

0000 0010 0100 0000 0100 0000 0010 0001 → **0x02404021**

# Vectores

- Un vector es una agrupación unidimensional de datos del mismo tipo que se almacenan en posiciones consecutivas de memoria y se identifica con un índice.

```
...
int v1[100];
...
int v2[5] = {0, 1, 2, 3, 4}
```

C

El vector v1 está definido del índice 0 al 99.  
El vector v2 está definido del índice 0 al 4.

```
.data
...
.align 2
v1:.space 400 # cada entero ocupa 4B
...
v2:.word 0, 1, 2, 3, 4
```

MIPS

Atención con  
la alineación

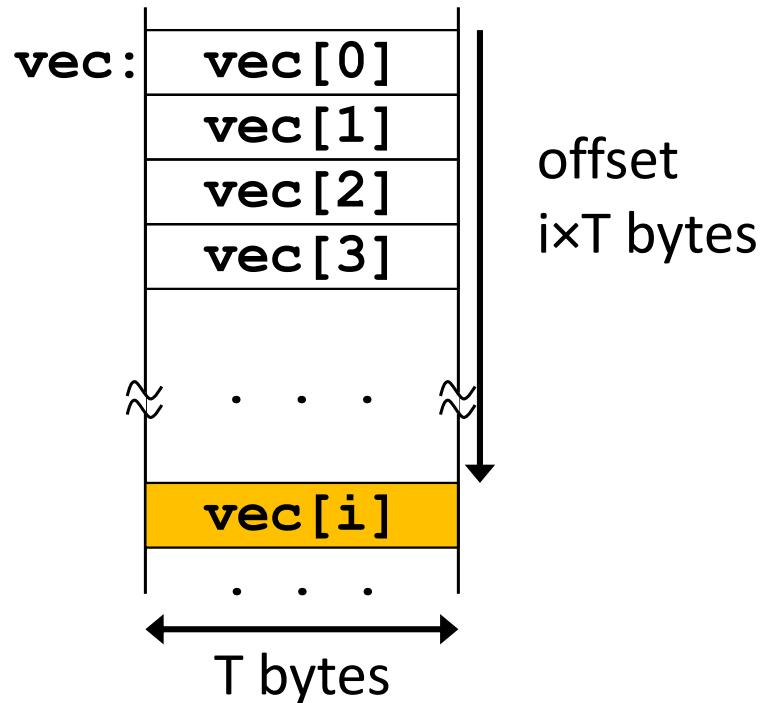


# Vectores, acceso aleatorio

- Para acceder al elemento `vec[i]` hemos de calcular su dirección

$$\text{@vec}[i] = \text{@vec}[0] + i * \text{tam}$$

siendo `tam`, el tamaño en bytes de los elementos del vector `vec`.



```
int vec[100];  
...  
x = vec[3]; // x en $t1
```

C

```
la $t0,vec  
lw $t1,12($t0)  
  
la $t0,vec+12  
lw $t1,0($t0)
```

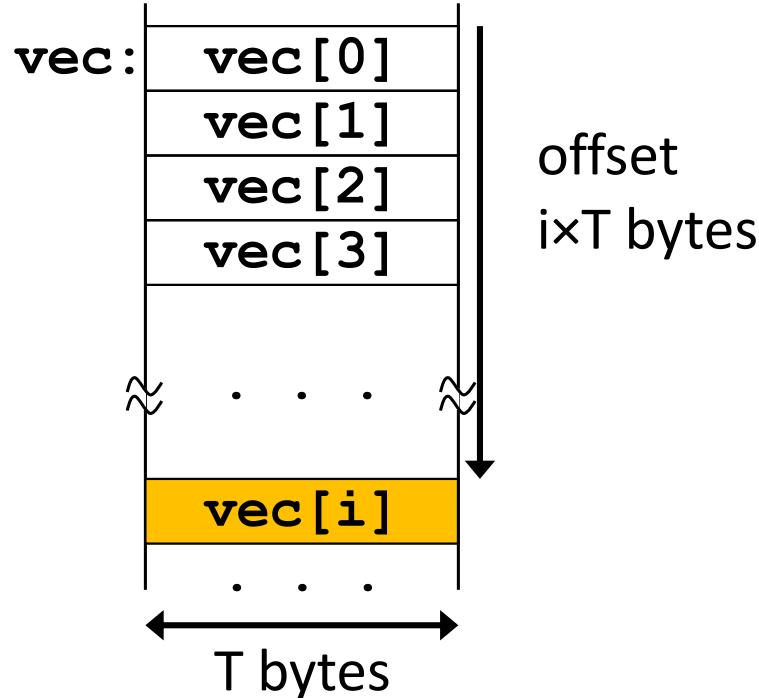
MIPS

# Vectores, acceso aleatorio

- Para acceder al elemento `vec[i]` hemos de calcular su dirección

$$\text{@vec}[i] = \text{@vec}[0] + i * \text{tam}$$

siendo `tam`, el tamaño en bytes de los elementos del vector `vec`.



```
int vec[100];  
...  
x = vec[i]; // x,i en $t1,$t2
```

C

```
la $t0,vec          # @vec      MIPS  
sll $t3,$t2,2       # i*4  
addu $t0,$t0,$t3    # @vec+i*4  
lw $t1,0($t0)       # x=vec[i]
```

MIPS

# Strings

- ❑ Un string es un vector de caracteres de tamaño variable.
- ❑ Se puede implementar de muchas formas:
  - En Java, con una tupla que tiene un entero con el tamaño y un vector de caracteres
  - **En C, un vector de caracteres con un centinela (el carácter '\0' = 0)**

```
char cadena[20] = "Una frase";  
char cadena[20] = { 'U', 'n', 'a', ' ', 'f',  
                   'r', 'a', 's', 'e', '\0' };
```

C

Declaraciones  
Equivalentes

```
cadena: .ascii "Una frase" # sin centinela  
        .space 11          # el centinela y 10 0's  
cadena: .asciiz "Una frase" # incluye el centinela  
        .space 10
```

MIPS



# Strings, ejemplo de uso

- Contar cuantos caracteres tiene un string.

```
char text[100];
int num = 0;
...
while (text[num] != '\0') num++;
```

C

```
main: ...
      move $t0,$zero          # num=0
      la $t1, text             # t1 = @text[0]
      while: addu $t3,$t0,$t1  # $t3 = @text[num]
              lb $t2,0($t3)       # $t2 = text[num]
              beq $t2,$zero,end    # ¿$t2 == '\0'?
              addiu $t0,$t0,1        # num++
              b while
end:
```

MIPS



# Punteros

## ❑ ¿Qué es un puntero?

Un puntero es una variable que contiene una dirección de memoria.

- Los punteros en MIPS ocupan 32 bits.
  - Si el puntero p contiene la dirección de la variable v, decimos que p apunta a v.
- 
- ❑ En C, como cualquier variable, puede ser global o local

```
int *p1;          // variable global
void main() {
    int *p2;      // variable local
    ...
}
```

C



# Punteros

- Si el punter es una **variable global**:

```
int *p1  
short *p2;  
char *p3;
```

C

- Se traduce a MIPS como:

```
.data  
p1: .word 0  
p2: .word 0  
p3: .word 0
```

MIPS

Un puntero es una dirección.  
Siempre ocupa 32 bits (1 word),  
independientemente de que sea un  
puntero a un int, un char o un short.

- Si el puntero es una **variable local**, no hay que reservar espacio, se almacenará en un registro.

# Punteros

- ❑ Uno de los errores más típicos en C es usar un puntero que no está inicializado.

Para inicializar un puntero utilizamos el operador de C: **&**. Este operador, delante de una variable, nos devuelve la dirección de esa variable.

- ❑ La inicialización del puntero puede estar en la declaración:

```
int v[100];
char a = 'E';
char *q = &a;
int *p = &v[12];
```

C

```
.data
v: .space 400
a: .byte 'E'
q: .word a      # q = &a
p: .word v+48  # p = &v[0]+12*4
```

MIPS

# Punteros

- ❑ Inicialización del puntero dentro de una sentencia

```
int *p;  
int b;  
void f() {  
    int *q; // en $t0  
    p = &b;  
    ...  
    q = &b;  
}
```

C

```
.data  
p: .word 0  
b: .word 0  
...  
f: la $t1,b      # $t1 = &b  
    la $t2,p      # $t2 = &p  
    sw $t1,0($t2) # p = &b  
    ...  
    move $t0, $t1  # q = &b
```

MIPS

# Punteros

- ❑ ¿Cómo puedo acceder al dato apuntado por un puntero?

En C, el operador \* delante de un puntero, devuelve el valor de la variable a la que apunta el puntero. Operando indirección (desreferència en català)

```
int *pgl;          C
void f() {
    int tmp; // en $t0
    tmp = *pgl;
    ...
    *pgl = 17;
}
```

```
.data
pgl: .word 0
...
f: la $t1, pgl      # $t1 = &pgl
    lw $t1, 0($t1)  # $t1 = pgl
    lw $t0, 0($t1)  # $t0 = *pgl
    ...
    li $t2, 17       # $t2 = 17
    sw $t2, 0($t1)  # *pgl = 17
```

# Punteros

- ❑ ¡Atención! No hay que confundir el **operando de declaración \*** con el **operando de indirección \***, aunque sean el mismo símbolo.

```
int *p1;
void g() {
    int *p2; // en $t0
    *p1 = *p2;
    ...
    *p2 = 37;
}
```

C

```
.data
p1: .word 0
...
g: lw $t2, 0($t0) # $t2 = *p2
    la $t1, p1      # $t1 = &p1
    lw $t1, 0($t1) # $t1 = p1
    sw $t2, 0($t1) # *p1 = *p2
    ...
    li $t2, 37      # $t2 = 37
    sw $t2, 0($t0) # *p2 = 37
```

MIPS

# Punteros

## □ Aritmética de punteros

Sumar un entero N, a un puntero p, que apunta a variables de tamaño T bytes, da como resultado un nuevo puntero que apunta a  $(p + N \cdot T)$

```
char *p1;  
short *p2;  
int *p3;  
long long *p4;  
  
...  
p1 += 3;  
p2 = p2 + 3;  
p3 += 3;  
p4 = p4 + 3;
```

C

```
.data  
p1: .word 0    # en $t1  
p2: .word 0    # en $t2  
p3: .word 0    # en $t3  
p4: .word 0    # en $t4  
  
...  
addiu $t1, $t1, 3  
addiu $t2, $t2, 3*2  
addiu $t3, $t3, 3*4  
addiu $t4, $t4, 3*8
```

MIPS

# Relación entre punteros y vectores en C

En C, un vector es un puntero que apunta al primer elemento del vector.

```
int vec[100];
int *p;
main() {
    p = vec
    ...
    p[8] = 10;
    ...
    *(p+i) = 0;
    p[i] = 0;
}
```

C

Expresión correcta

Acceso al elemento 8 del vector vec

Expresiones Equivalentes.  
Aritmética de punteros



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Departament d'Arquitectura de Computadors

# Estructura de Computadores

## Tema 2: Instrucciones y Tipos Básicos de Datos

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

