



Estructura de computadores

Grado en ingeniería informática

PhD Octavio Castillo-Reyes
octavio.castillo@upc.edu

Q2 2022-2023

Acerca del curso

- Conocer la jerarquía de niveles de un computador
- Conocer la arquitectura **ISA** de un procesador **RISC**
- Saber representar y operar con **números reales y enteros**
- Saber cómo se almacenan y acceden **datos estructurados**
- Saber traducir programas de alto nivel a **lenguaje ensamblador**

Acerca del curso

- Conocer la estructura y el funcionamiento de la **memoria cache**
- Entender el funcionamiento básico de la **memoria virtual**
- Conocer los conceptos de **excepción e interrupción**
- Adquirir competencias transversales (**sostenibilidad**)

Acerca del curso

Coordinador del curso

Jordi Tubella Murgadas (jordit@ac.upc.edu)

Horas semanales

Teoría	Problemas	Laboratorio	Aprendizaje dirigido	Aprendizaje autónomo
2,7	1,3	1	0,5	7

Si no dispones de este tiempo para dedicarte a la asignatura es mejor que lo dejes para cuando lo tengas. Pero si haces lo que te proponemos es **casi seguro que aprobarás la asignatura.**

Asesorías (horario a convenir)

octavio.castillo@upc.edu

Sala C6-122

Acerca del curso

Contenido

Tema 1: Introducción

Tema 2: Ensamblador MIPS y tipos de datos básicos

Tema 3: Traducción de programas

Tema 4: Matrices

Tema 5: Aritmética de enteros y coma flotante

Tema 6: Memoria cache

Tema 7: Memoria virtual

Tema 8: Excepciones e interrupciones

Recursos

Bibliografía

D. Patterson and J. L. Hennessy. “Estructura y Diseño de Computadores: La Interfaz Hardware/Software”, 2011.

Online:

<http://docencia.ac.upc.edu/FIB/grau/EC/>

<https://raco.fib.upc.edu/>

Problemas, apuntes, prácticas, exámenes de años anteriores...

Organización del curso

27 sesiones de teoría y problemas

Lunes y Jueves de 12:00 a 14:00 hrs (A5E01)

6 sesiones de laboratorio

Sesión 0: Introducción

Sesión 1: Ensamblador MIPS y tipos de datos básicos

Sesión 2: Traducción de programas

Sesión 3: Tipos de datos estructurados

Sesión 4: Codificación en coma flotante

Sesión 5: Memoria cache

Exámenes

Parcial **24/04 (8:00hrs)**

Final **20/06 (8:00hrs)**

Laboratorio Depende del grupo (revisar calendario en sitio web)

Evaluación

$$0.2 * \max(\text{EP}, \text{EF}) + 0.6 * \text{EF} + 0.2 * (\text{EL} * 0.85 + \text{EC} * 0.15)$$

EP = Examen parcial

EF = Examen final

EL = Examen de laboratorio

EC = Evaluación continua de laboratorio

Evaluación continua en cada sesión de laboratorio

Estudio previo e individual

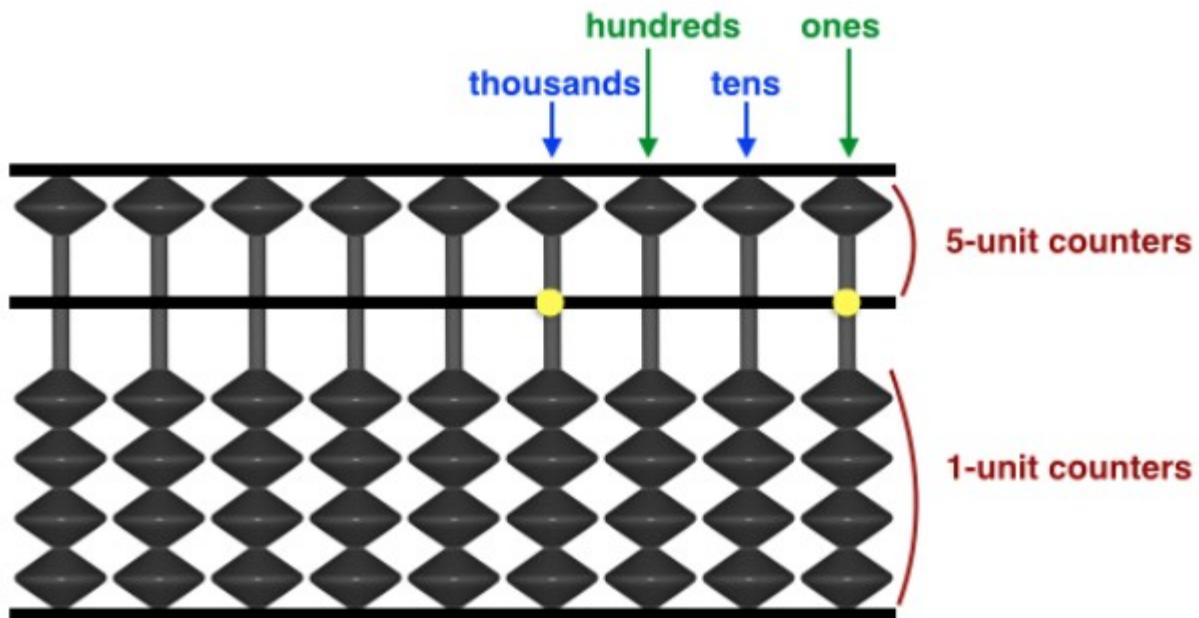
Actividades por parejas durante la sesión presencial



T1: Introducción

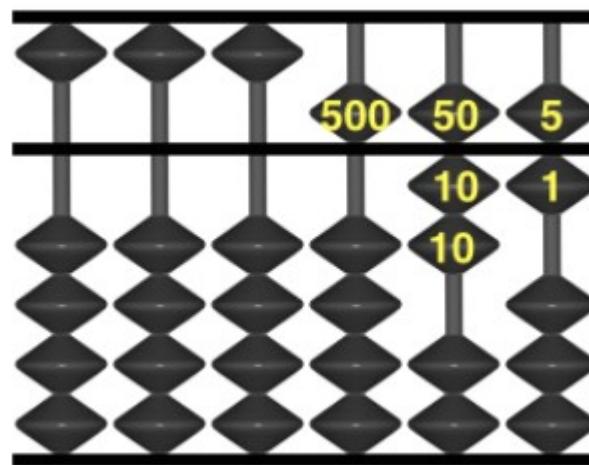
PhD Octavio Castillo-Reyes
octavio.castillo@upc.edu

El computador



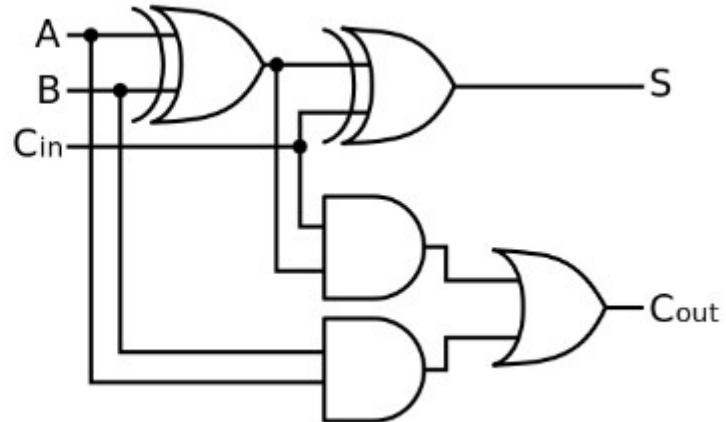
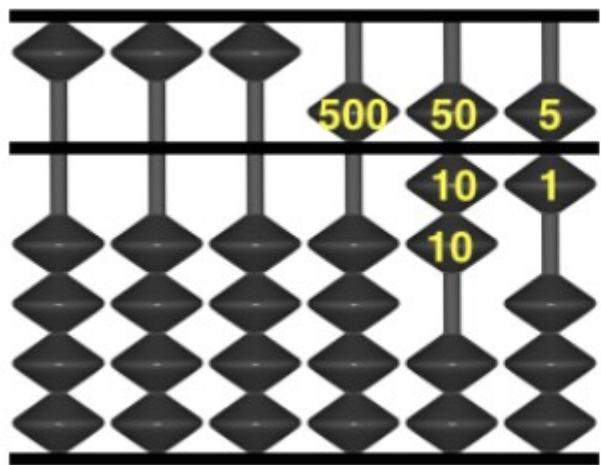
El computador

Ábaco: movimiento de fichas



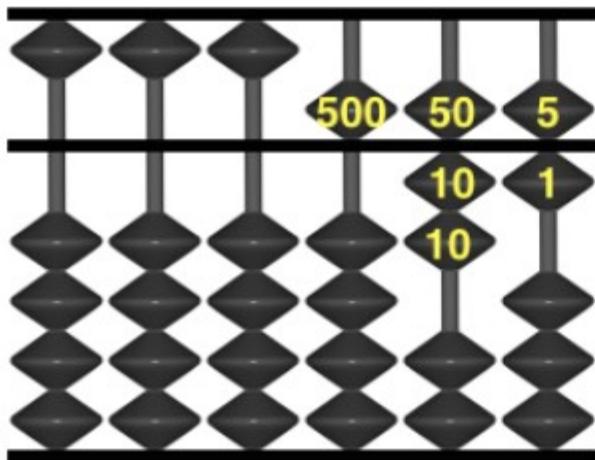
El computador

Ábaco: movimiento de fichas

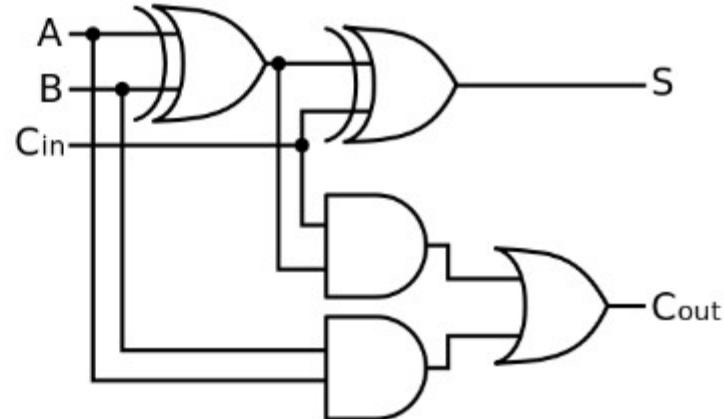


Computador digital: movimiento de electrones

El computador



Mecánico
Manual
Almacenamiento reducido
Lento



Electrónico
Automático/**Programable**
Gran almacenamiento
Muy rápido (PFLOPS)

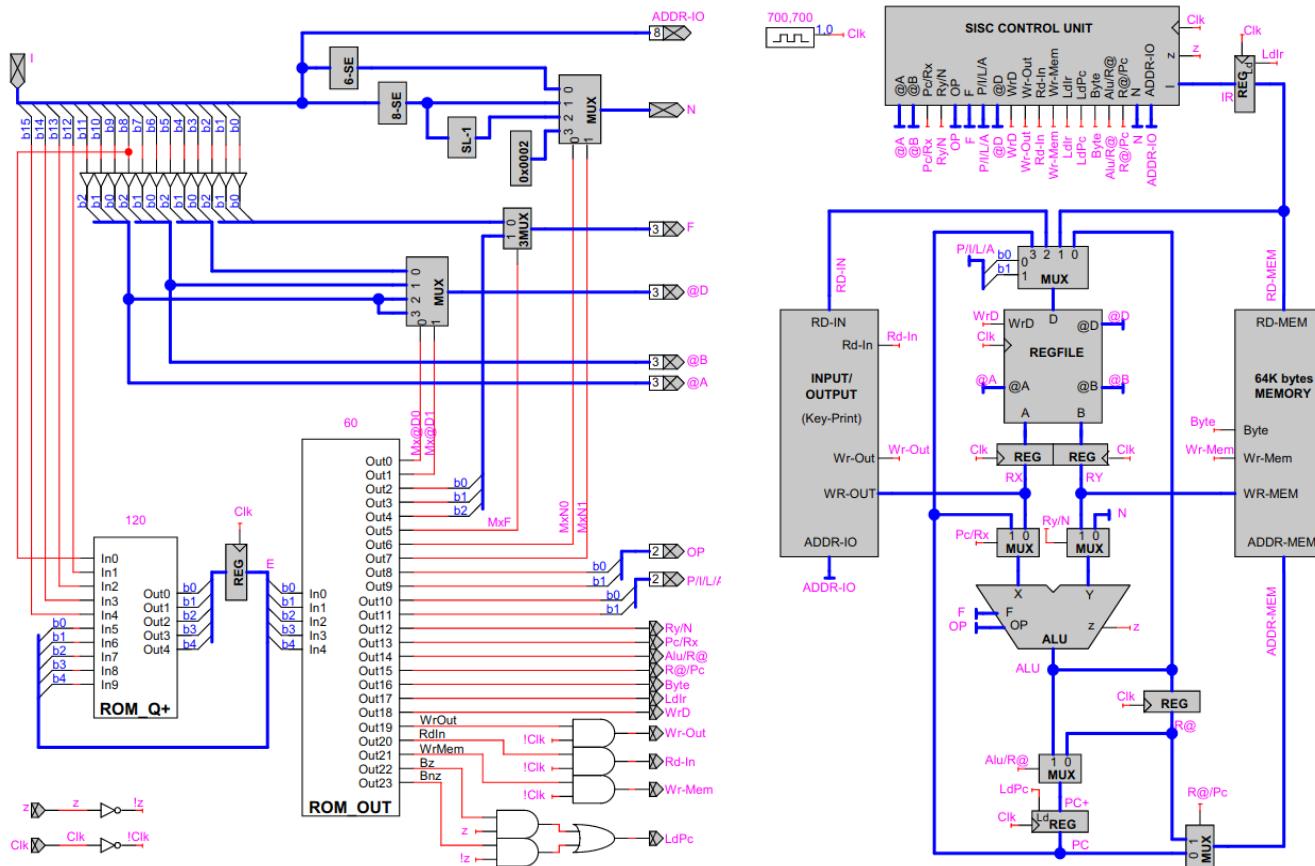
El computador

Computador digital: Dispositivo electrónico capaz de almacenar y procesar información automática



El computador

Computador digital: Dispositivo electrónico capaz de almacenar y procesar información automática



El usuario no puede interactuar con los electrones de forma directa

Niveles de abstracción



Interfaz de usuario

Iconos, menus, botones, ...

Lenguaje de alto nivel

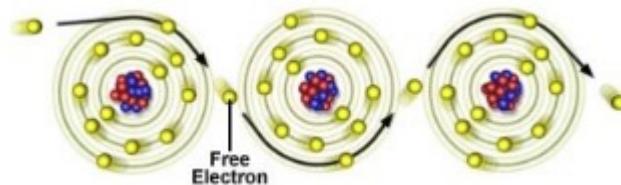
C, C++, Fortran, Java, Python, R, Matlab (PRO2)

Lenguaje máquina

MIPS, x86, ARM, RISC-C (**EC**)

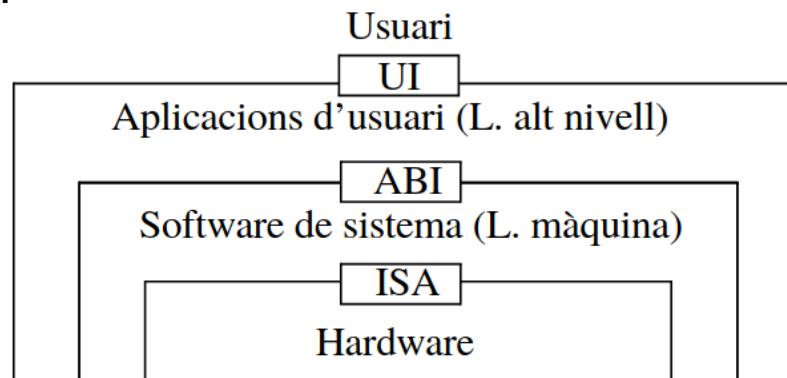
Hardware

Puertas lógicas, multiplexores, registros, CLC, CLS (IC)



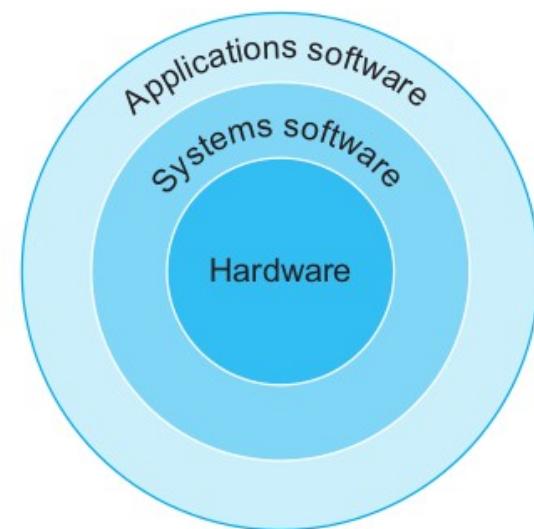
Interfaces entre niveles

- Application Binary Interface (ABI)
 - Funciones del Sistema Operativo (llamadas al sistema)
- Instruction Set Architecture (ISA)
 - Interfaz entre el hardware y el software
 - Describe los aspectos del procesador visibles al programador en lenguaje máquina
 - Conjunto de instrucciones, modos de direccionamiento, excepciones, modelo de memoria
 - MIPS, x86, ARM



Interfaces entre niveles

- Application Binary Interface (ABI)
 - Funciones del Sistema Operativo (llamadas al sistema)
- Instruction Set Architecture (ISA)
 - Interfaz entre el hardware y el software
 - Describe los aspectos del procesador visibles al programador en lenguaje máquina
 - Conjunto de instrucciones, modos de direccionamiento, excepciones, modelo de memoria
 - MIPS, x86, ARM



Traducción entre niveles

- Lenguaje de alto nivel
 - Abstracto
 - Portable
 - Rápido de escribir
- Lenguaje ensamblador
 - Representación textual de instrucciones
- Lenguaje máquina
 - Representación hardware
 - Dígitos binarios (bits)
 - Datos e instrucciones

High-level
language
program
(in C)

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

Compiler

Assembly
language
program
(for MIPS)

```
swap:
 multi $2, $5,4
 add $2, $4,$2
 lw $15, 0($2)
 lw $16, 4($2)
 sw $16, 0($2)
 sw $15, 4($2)
 jr $31
```

Assembler

Binary machine
language
program
(for MIPS)

```
000000001010001000000000100011000
00000000100000100001000000100001
10001101110001000000000000000000
10001110000100100000000000000000
10101110000100100000000000000000
10101101110001000000000000000000
00000011110000000000000000000000
```

Compilación vs interpretación

- Compilación
 - Traducción del programa entero una sola vez (estático)
 - Programa generado es muy rápido (flags de optimización)
 - Recompilación para cada ISA y/o ABI (no portable)
 - Ejemplos: C/C++, Fortran, Pascal...
- Interpretación
 - Traducción dinámica en tiempo de ejecución
 - La ejecución es más lenta
 - Portabilidad
 - Ejemplos: Java, Python, R, Matlab

Los lenguajes interpretados son mucho más productivos, portables y seguros... pero son demasiado lentos para aplicaciones donde el rendimiento es crítico



T2: Ensamblador MIPS y tipos de datos básicos

PhD Octavio Castillo-Reyes
octavio.castillo@upc.edu

CISC vs RISC

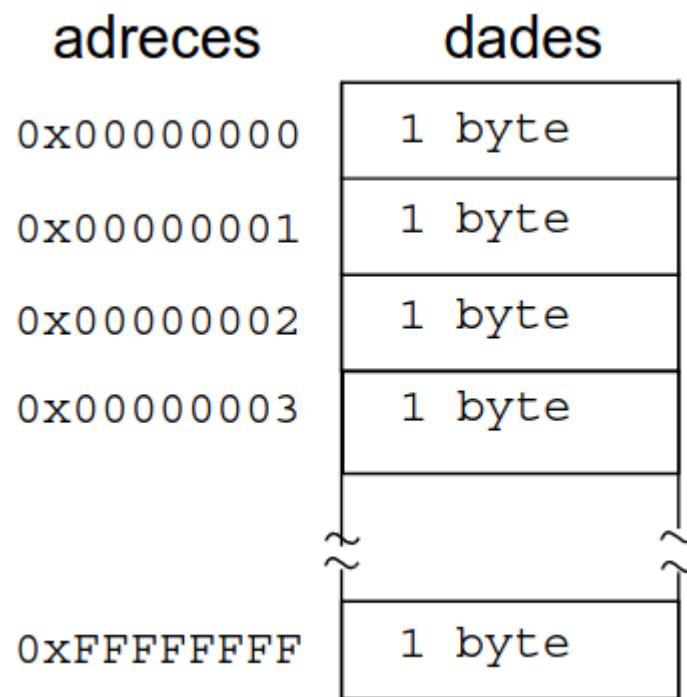
- Complex Instruction Set Computer (CISC)
 - Juego de instrucciones grande y complejo
 - Instrucciones de longitud variable
 - Cada instrucción se decodifica en múltiple microoperaciones
 - Ejemplo: x86
- Reduced Instruction Set Computer (RISC)
 - Juego de instrucciones pequeño y sencillo
 - Instrucciones de longitud fija
 - Formatos de instrucción y modos de direccionamiento sencillos
 - Ejecutadas directamente por hardware
 - Ejemplos: **MIPS**, ARM, RISC-V

MIPS

- Microprocessor without Interlocked Pipeline Stages
 - Diseñado en 1981-1985 por Henessy y Patterson
 - Juego de instrucciones sencillo (RISC)
 - Distintas implementaciones comerciales
 - Routers de Cisco y Linksys
 - Módems ADSL
 - Controladoras de impresora láser
 - Playstation (PSX, PS2 y PSP)
 - Nintendo 64
- Ampliamente utilizado para la docencia
 - Muchos de los conceptos son muy similares en otros ISAs RISC
 - **MIPS32**: ISA utilizado en la asignatura (teoría y laboratorio)

La memoria

- Vector de bytes
 - Cada byte se identifica por una dirección

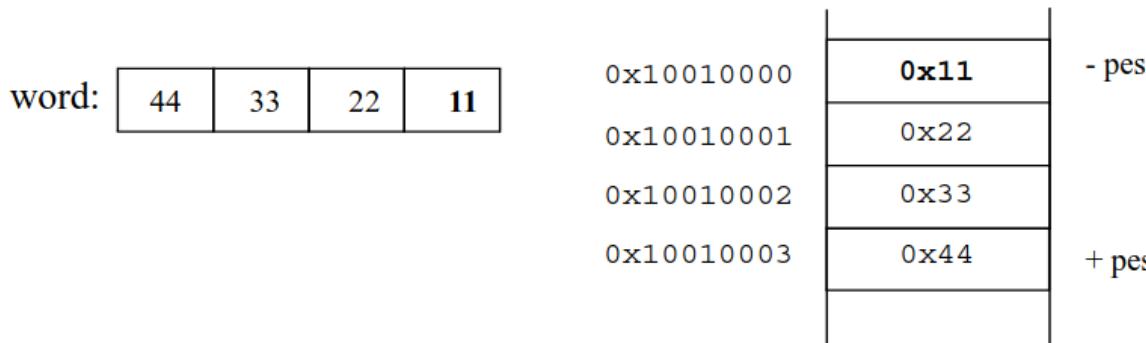


Palabra (word)

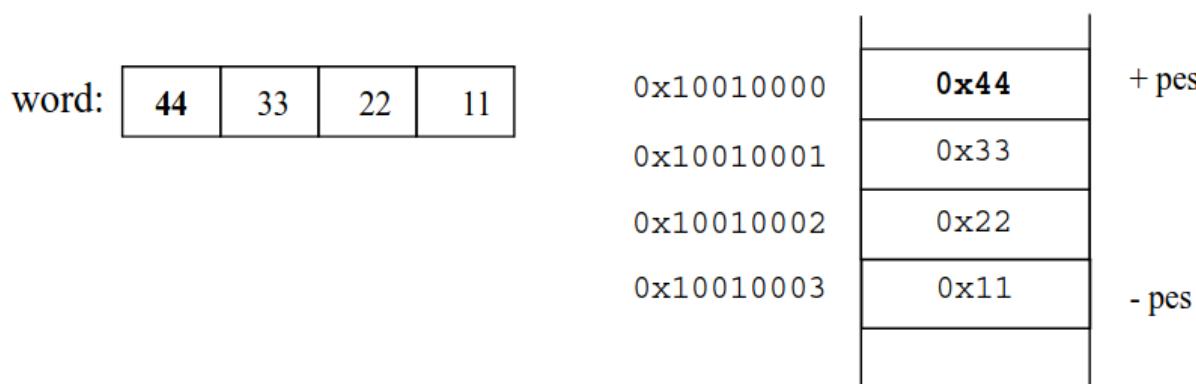
- Dato que tiene el tamaño nativo de la arquitectura
- Normalmente, tamaño de palabra = tamaño de registro
- MIPS32: 32 bits (4 bytes)
- Ejemplos: 0xAABBCCDD, 0x44332211
- **¿Cómo se almacenan los bytes de una palabra en memoria?**

Endianness

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



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



Declaración de variables

- Variables **globales**
 - Pueden ser accedidas desde cualquier función
 - Se mantienen en memoria durante toda la ejecución del programa
 - Se almacenan en una dirección de memoria fija
- Variables **locales**
 - Solo 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)

Declaración de variables

Lenguaje C

```
int a = 0x44332211;  
  
int main(void) {  
    int i = 7;  
}
```

Ensamblador MIPS

```
.data  
a: .word 0x44332211  
  
.text  
main:  
    li $t0, 7
```

Tamaño de variables

Tamaño	Lenguaje C	Ensamblador MIPS
1 byte	char / unsigned char	.byte
2 bytes	short / unsigned short	.half
4 bytes	int / unsigned int	.word
8 bytes	long long / unsigned long long	.dword

Alineación de memoria

```
unsigned char a;          /* 1 byte      */
short b = 13,             /* 2 bytes     */
char c = -1, d = 10;      /* 1+1 bytes   */
int e = 0x10AA00FF;       /* 4 bytes     */
long long f = 0x7766554433221100; /* 8  */
```

```
.data
a: .byte 0
b: .half 13
c: .byte -1
d: .byte 10
e: .word 0x10AA00FF
f: .dword 0x7766554433221100
```

Alineación de memoria

```
unsigned char a;          /* 1 byte    */  
short b = 13,             /* 2 bytes   */  
char c = -1, d = 10;      /* 1+1 bytes */  
int e = 0x10AA00FF;       /* 4 bytes   */  
long long f = 0x7766554433221100; /* 8   */
```

```
.data  
a: .byte 0  
b: .half 13  
c: .byte -1  
d: .byte 10  
e: .word 0x10AA00FF  
f: .dword 0x7766554433221100
```

etiqs.	adreses
a:	00
b:	0D
c:	00
d:	FF
	0A
e:	00
	AA
	FF
	10
f:	00
	11
	22
	33
	44
	55
	66
	77

Alineación de memoria

Traduce a MIPS la siguiente declaración de variables en C e indica el contenido de la memoria

```
char a = 0xFF;
char b = 0xEE;
char c = 0xDD;
unsigned long long d = 0x7766554433221100;
short e = 0xABCD;
unsigned int f = 0x40302010;
```

Declaración de vectores

- Declaración con inicialización

```
short vec[5] = {2, -1, 3, 5, 0};  
  
.data  
vec: .half 2, -1, 3, 5, 0
```

- Declaración sin inicialización (alineación explícita)

```
char a;  
int v[100];  
  
.data  
a: .byte 0  
.align 2    # Alinear a multiplo de 4  
v: .space 400 # Vector de 100 enteros
```



T2: Modos de direccionamiento y enteros

PhD Octavio Castillo-Reyes
octavio.castillo@upc.edu

Modos de direccionamiento

- Forma en la que se especifica un operando en una instrucción
- MIPS soporta cinco modos de direccionamiento
 - Modo inmediato
 - Modo registro
 - Modo memoria
 - Modo relativo al PC
 - Modo pseudodirecto

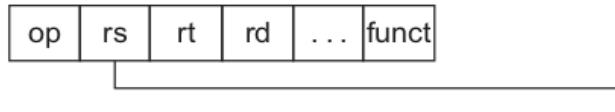
Modos de direccionamiento

1. Immediate addressing



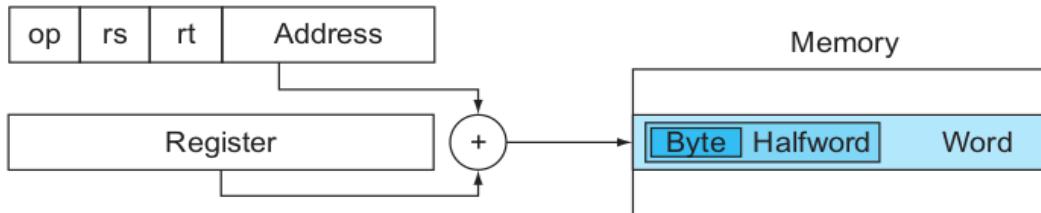
addi \$t1, \$t0, 1

2. Register addressing



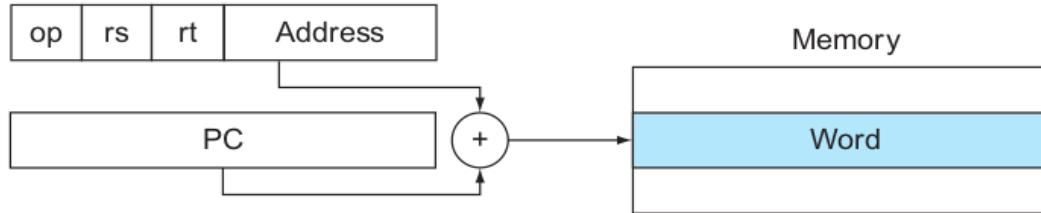
sub \$t1, \$t2, \$s6

3. Base addressing



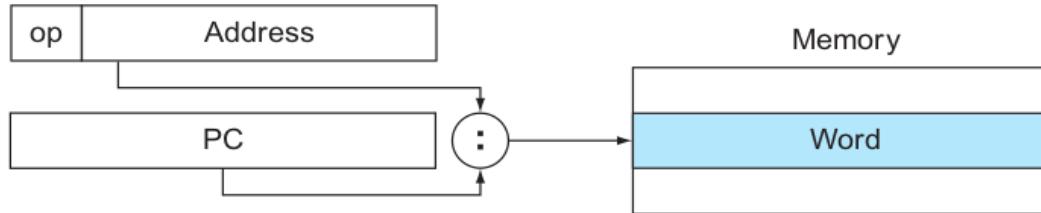
lw \$t0, 6(\$t1)

4. PC-relative addressing



bne/beq \$s0, \$s1, 2

5. Pseudodirect addressing



j label_1

Modo registro

- El operando reside en un registro
- La instrucción especifica el identificador del registro
- Suma: **addu rd, rs, rt**
- Resta: **subu rd, rs, rt**
- MIPS incluye 32 registros de 32 bits

Registros

Número	Nombre	Descripción
\$0	\$zero	Contiene el valor 0 (solo lectura)
\$1	\$at	Registro temporal para pseudoinstrucciones
\$2-\$3	\$v0-\$v1	Resultados de subrutinas
\$4-\$7	\$a0-\$a3	Parametros de una subrutina
\$8-\$15	\$t0-\$t7	Temporales
\$16-\$23	\$s0-\$s7	Seguros (se preservan al llamar a una subrutina)
\$24-\$25	\$t8-\$t9	Temporales
\$26-\$27	\$k0-\$k1	Reservados para el SO
\$28	\$gp	Global pointer
\$29	\$sp	Stack pointer
\$30	\$fp	Frame pointer
\$31	\$ra	Return address

Modo registro: ejercicio

Dada la siguiente sentencia en C:

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

Las variables f, g, h, i, j se asignan a los registros \$s0, \$s1, \$s2, \$s3 y \$s4, respectivamente. ¿cuál será su traducción a lenguaje ensamblador MIPS?

Operандos en modo inmediato

- El operando se codifica en la propia instrucción
- Valor de 16 bits en Ca2 (rango: -2^{15} a $2^{15}-1$)
 - -2^{n-1} a $2^{n-1}-1$
- ¿Cómo se convierte en un valor de 32 bits?
 - Extensión de signo
 - Extensión de ceros

<code>addiu rt, rs, imm16</code>	$rt = rs + \text{SignExt}(imm16)$
<code>lui rt, imm16</code>	$rt_{31..16} = imm16$ $rt_{15..0} = 0x0000$
<code>ori rt, rs, imm16</code>	$rt = rs \text{ OR } \text{ZeroExt}(imm16)$

Operandos en modo memoria

- Solo las operaciones de tipo **load** y **store** admiten un operando que resida en memoria
 - MIPS es una arquitectura load-store
- Hay que cargar los datos de memoria en registros para poder utilizarlos en instrucciones aritmético-lógicas
- Lectura y escritura de palabras en memoria

<code>lw rt, off16(rs)</code>	$rt = M_W[rs + \text{SignExt}(off16)]$	Load word
<code>sw rt, off16(rs)</code>	$M_W[rs + \text{SignExt}(off16)] = rt$	Store word

Modo memoria: ejercicio

- Traducir a MIPS la siguiente asignación de valores enteros (words) en C, suponiendo que \$t1 tiene la base de la tabla A y \$s2 se corresponde con h

$A[300] = h + A[300];$

Modo memoria: ejercicio

- Traducir a MIPS la siguiente asignación de valores enteros (words) en C, suponiendo que h ocupa \$t2 y que la dirección base de A está en \$t3

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

Acceso a halfword (byte): enteros con signo

- Load halfword: **lh rt, off16(rs)**
 - Copia un halfword (2 bytes) de la memoria a los 16 bits de menor peso del registro rt
 - **Realiza extensión de signo** (extiende el bit 15)
- Store halfword: **sh rt, off16(rs)**
 - Copia a memoria los dos bytes de menor peso de rt
- Load byte: **lb rt, off16(rs)**
 - Copia 1 byte de memoria a los 8 bits de menor peso del registro rt
 - **Realiza extensión de signo** (extiende el bit 7)
- Store byte: **sb rt, off16(rs)**
 - Copia a memoria el byte de menor peso del registro rt

Acceso a halfword (byte): enteros con signo

- Suponiendo que \$t2 contiene la dirección 0x10010000 y que el contenido de la memoria es el que se muestra en la parte inferior, indica el resultado de cada instrucción y el contenido final de la memoria:

```
lb    $t1, 1($t2)
lb    $t1, 2($t2)
lh    $t1, 0($t2)
lh    $t1, 2($t2)
sb    $t1, 1($t2)
```

adreça	estat inicial
0x10010000	0x11
0x10010001	0x22
0x10010002	0xCC
0x10010003	0xDD

Acceso a halfword (byte): enteros con signo

- Suponiendo que \$t2 contiene la dirección 0x10010000 y que el contenido de la memoria es el que se muestra en la parte inferior, indica el resultado de cada instrucción y el contenido final de la memoria:

lb	\$t1, 1(\$t2)	# load byte 0x22.	\$t1 = 0x00000022
lb	\$t1, 2(\$t2)	# load byte 0xCC.	\$t1 = 0xFFFFFFFCC
lh	\$t1, 0(\$t2)	# load half 0x2211.	\$t1 = 0x00002211
lh	\$t1, 2(\$t2)	# load half 0xDDCC.	\$t1 = 0xFFFFDDCC
sb	\$t1, 1(\$t2)	# store byte 0xCC a l'adreça 0x10010001	

adreça	estat inicial	estat final
0x10010000	0x11	0x11
0x10010001	0x22	0xCC
0x10010002	0xCC	0xCC
0x10010003	0xDD	0xDD

Acceso a halfword (byte): naturales

- Load halfword unsigned: **Ihu rt, off16(rs)**
 - Copia un halfword (2 bytes) de la memoria a los 16 bits de menor peso del registro rt
 - **Realiza extensión de ceros**
- Load byte unsigned: **Ibu rt, off16(rs)**
 - Copia 1 byte de memoria a los 8 bits de menor peso del registro rt
 - **Realiza extensión de ceros**

Acceso a halfword (byte): naturales

- Suponiendo que \$t2 contiene la dirección 0x10010000 y que el contenido de la memoria es el que se muestra en la parte inferior, indica el resultado de cada instrucción:

```
lbu    $t1, 2($t2)      # load byte 0xCC.      $t1 = 0x000000CC  
lhu    $t1, 2($t2)      # load half 0xDDCC.    $t1 = 0x0000DDCC
```

adreça	estat inicial
0x10010000	0x11
0x10010001	0x22
0x10010002	0xCC
0x10010003	0xDD

Acceso a doble palabra

- ¿Cómo se accede a un dato de 64 bits (long long) en la arquitectura MIPS de 32 bits?

```
. data
x: .dword 0x7766554433221100

.text
main:
# $t2 contiene la direccion de memoria de x
lw $t0, 0($t2)
lw $t1, 4($t2)
```

Restricción de alineación

- Las direcciones utilizadas en las instrucciones lw y sw han de ser **múltiplos de 4**
- Las direcciones utilizadas en las instrucciones lh, lhu y sh han de ser **múltiplos de 2**
- En caso contrario se produce una excepción por dirección no alineada y el programa termina

```
.data  
x:    .word    0xDDCCAABB
```

```
.text  
la    $t0, x+3  
lw    $t1, 0($t0)
```

Pseudoinstrucciones o macros

- Simplifican operaciones comunes para las que no existe una instrucción MIPS
- Facilitan el desarrollo, la lectura y la depuración del código
- En el momento de ser ensamblada se traduce a una o varias instrucciones MIPS

```
move $t1, $t2          # addu $t1, $t2, $zero  
  
li $t1, 100            # addiu $t1, $zero, 100  
  
li $t1, 0x0030D900    # lui $at, 0x0030  
                      # ori $t1, $at, 0xD900
```

Pseudoinstrucciones o macros

- Simplifican operaciones comunes para las que no existe una instrucción MIPS
- Facilitan el desarrollo, la lectura y la depuración del código
- En el momento de ser ensamblada se traduce a una o varias instrucciones MIPS

```
.data  
y: .word 42      # Dirección de y = 0x10010024  
  
.text  
la $t0, y        # lui $at, 0x1001  
                  # ori $t0, $at, 0x0024
```

Formatos de representación de enteros

- En EC estudiaremos 4 formatos
 - Complemento a 2
 - Complemento a 1
 - Signo y magnitud
 - Exceso
- **Regla de representación**
 - Indica cómo codificar un número entero en binario
- **Regla de interpretación**
 - Indica cómo convertir una codificación binaria a un número entero en base 10
- **Rango de representación**
 - Números enteros que se pueden codificar usando N bits

Complemento a 2 (Ca2)

- Regla de representación para n bits:
 - Si es positivo: representar como natural
 - Si es negativo: sumar 2^n y representar como natural
- Regla de interpretación para n bits:
 - Interpretar como natural
 - Si el bit de mayor peso es 1 (negativo): restar 2^{n-1}
- Regla de cambio de signo
 - Complementar bits y sumar 1
 - Rango de representación con n bits: $[-2^{n-1}, 2^{n-1} - 1]$
- Rango NO simétrico
- Mismo circuito sumador para naturales y enteros en Ca2

Complemento a 1 (Ca1)

- Regla de representación para n bits:
 - Si es positivo: representar como natural
 - Si es negativo: sumar 2^{n-1} y representar como natural
- Regla de interpretación para n bits:
 - Interpretar como natural
 - Si el bit de mayor peso es 1 (negativo): restar 2^{n-1}
- Regla de cambio de signo
 - Complementar bits
- Rango de representación con n bits: $[-2^{n-1} + 1, 2^{n-1} - 1]$
- Dos representaciones para el cero
- Requiere un circuito de suma diferente al de los naturales

Signo y magnitud

- Regla de representación para n bits:
 - Codificar el signo en el bit de mayor peso (0 positivo, 1 negativo)
 - Codificar el valor absoluto (magnitud) en los $n - 1$ bits restantes
- Regla de interpretación para n bits:
 - El bit de mayor peso indica el signo (0 positivo, 1 negativo)
 - Los $n - 1$ bits restantes indican el valor absoluto
- Regla de cambio de signo
 - Complementar el bit de mayor peso
- Dos representaciones para el cero
- Circuito específico para la suma

Exceso K

- Regla de representación para n bits:
 - Sumar K y representar el resultado como natural
- Regla de interpretación para n bits:
 - Interpretar como natural y restar K
- Rango de representación con n bits: $[-K, 2^n - K - 1]$
- Normalmente $K=2^{n-1} - 1$ para equilibrar positivos y negativos
- El bit de mayor peso NO indica el signo
- La comparación se puede hacer con el mismo circuito que compara naturales

Ejercicio

$$N_{EX} = N + 2^n - 1$$

N	SM	Ca1	Ca2	Exceso a $2^3 - 1$
8				
7				
6				
5				
4				
3				
2				
1				
0				
-1				
-2				
-3				



T2: ASCII y punteros

PhD Octavio Castillo-Reyes
octavio.castillo@upc.edu

ASCII

- Sistema de codificación de caracteres
 - Asigna un código numérico a cada símbolo
- En EC estudiaremos el ASCII de 7 bits
 - Los caracteres se almacenan utilizando 1 byte (8 bits)
 - El bit de mayor peso siempre vale 0
 - Códigos del 0 al 31 son de control
 - El resto de códigos son símbolos tipográficos (imprimibles)

Código	Símbolo	En C y MIPS
0x09	TAB	'\t'
0x0A	LF	'\n'
0x30	1	'1'
0x41	A	'A'
0x61	a	'a'

ASCII: propiedades

- Orden alfabético
 - 'a' = 97, 'b' = 98, 'c' = 99, etc.
 - 'A' = 65, 'B' = 66, 'C' = 67, etc.
 - '0' = 48, '1' = 49, '2' = 50, etc.
- Conversión minúscula/mayúscula
 - De mayúscula a minúscula sumando 32: 'a' = 'A' + 32
 - De minúscula a mayúscula restando 32: 'B' = 'b' - 32
- Conversión dígito ASCII/valor numérico
 - Caracter '1' representa el dígito 1 y tiene un código ASCII de 49
 - Se puede convertir un dígito (número natural) a su código ASCII sumando 48
 - $1 + 48 = '1'$
 - También se puede convertir sumando el '0'
 - $1 + '0' = '1'$

ASCII: propiedades

- Declaración en C de variables tipo char

```
char letra = 'R'
```

- En MIPS

```
letra: .byte 'R'
```

Formato de instrucciones en MIPS

- Las instrucciones se representan con cadenas de bits y se almacenan en memoria
 - MIPS32: Instrucciones de 32 bits
- Formatos de instrucción en MIPS:
 - R (register), I (immediate) y J (jump)

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

Formato de instrucciones en MIPS

- *opcode*: código de operación
- *funct*: extensión del código de operación
- *rs*, *rt*, *rd*: operandos en modo registro
- *imm16*: operando en modo inmediato de 16 bits
- *shamt*: shift amount, inmediato para desplazamientos
- *target*: dirección de destino de un salto

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

Formato de instrucciones en MIPS

- Ejemplos

		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
addu rd, rs, rt	R	000000	rs	rt	rd	00000	100001
sra rd, rt, shamt	R	000000	rs	rt	rd	shamt	000011
addiu rt, rs, imm16	I	001000	rs	rt		imm16	
lui rt, imm16	I	001111	00000	rt		imm16	
lw rt, offset16(rs)	I	100011	rs	rt		offset16	
j target	J	000010				target	

Formato de instrucciones en MIPS

- Codifica en lenguaje máquina las siguientes instrucciones en lenguaje ensamblador MIPS

addu \$t4, \$t3, \$t5

addiu \$t7, \$t6, 25

lw \$t3, 0(\$t2)

		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
addu rd, rs, rt	R	000000	rs	rt	rd	00000	100001
sra rd, rt, shamt	R	000000	rs	rt	rd	shamt	000011
addiu rt, rs, imm16	I	001000	rs	rt		imm16	
lui rt, imm16	I	001111	00000	rt		imm16	
lw rt, offset16(rs)	I	100011	rs	rt		offset16	
j target	J	000010			target		

Formato de instrucciones en MIPS

- Desensambla las siguientes instrucciones

0xAE0BFFFC

0x8D08FFC0

0x0233a823

		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
addu rd, rs, rt	R	000000	rs	rt	rd	00000	100001
sra rd, rt, shamt	R	000000	rs	rt	rd	shamt	000011
addiu rt, rs, imm16	I	001000	rs	rt		imm16	
lui rt, imm16	I	001111	00000	rt		imm16	
lw rt, offset16(rs)	I	100011	rs	rt		offset16	
j target	J	000010			target		

Punteros

- Variable que contiene una dirección de memoria
 - 32 bits en MIPS32
 - Similar a una variable de tipo entero
- Si p contiene la dirección de memoria de la variable v decimos que p apunta a v
- Declaración de punteros:

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

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

Inicialización de punteros

- Asignando otro puntero del mismo tipo
- Asignando la dirección de una variable (operador &)

```
char a = 'E';
char b = 'K';
char *p = &a;

void f() {
    p = &b;
}
```

```
.data
a: .byte 'E'
b: .byte 'K'
p: .word a

.text
f:
    la $t0 , b
    la $t1 , p
    sw $t0 , 0($t1)
```

Desreferencia (indirección) de punteros

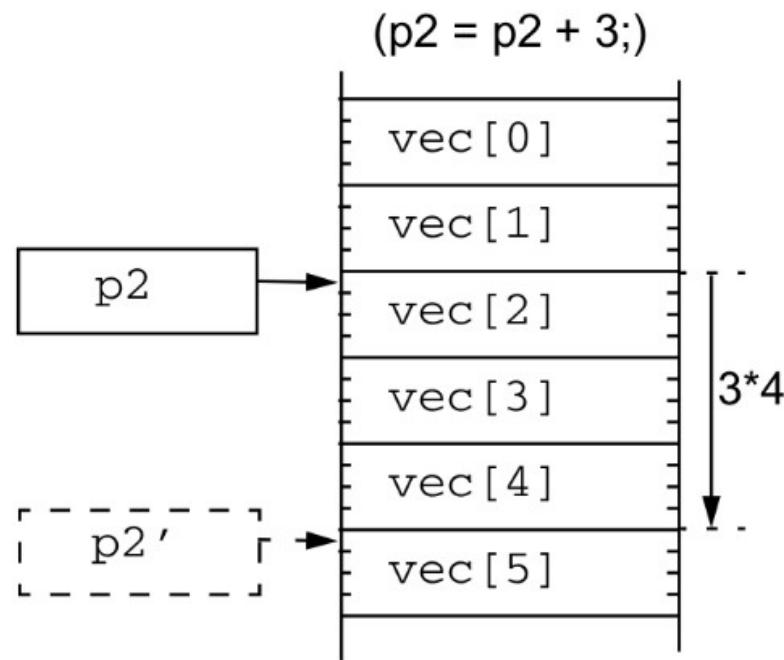
- Los punteros sirven para acceder a las variables apuntadas
- La desreferencia consiste en acceder a la dirección de memoria apuntada por el puntero
- En C se utiliza el operador *
 - $*p$: contenido de la dirección de memoria apuntada por p
 - No confundir con la declaración de punteros

```
char a = 'E';           . data
char *p = &a;           a: .byte 'E'
                        p: .word a

void f() {
    char tmp = *p;     . text
}                           f: la $t0, p
                            lw $t1, 0($t0)
                            lb $t2, 0($t1)
```

Aritmética de punteros

- Suma de un puntero p más un entero N
- Da como resultado otro puntero q del mismo tipo: $q = p + N$
- q apunta a otro entero situado N elementos más adelante



Aritmética de punteros

C	MIPS
char *p1;	p1: .word 0
int *p2;	p2: .word 0
long long *p3;	p3: .word 0
...	...
p1 = p1 + 3;	addiu \$t1 , \$t1 , 3
p2 = p2 + 3;	addiu \$t2 , \$t2 , 12
p3 = p3 + 3;	addiu \$t3 , \$t3 , 24

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
q = q + 1;
```

```
a = *p;
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
q = q + 1;
```

```
addiu $t1, $t1, 4
```

```
a = *p;
```

```
lb $t3, 0($t0)
la $t4, a
sb $t3, 0($t4)
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
h = &c;
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
h = &c;
```

```
la $t2, c
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
b = *(q + b);
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
b = *(q + b);
```

```
la  $t3, b
lw  $t6, 0($t3)
sll $t4, $t6, 2
addu $t5, $t1, $t4
lw  $t5, 0($t5)
sw  $t5, 0($t3)
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
*h = *(h + b);
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
*h = *(h + b);
```

```
la  $t3, b
lw  $t3, 0($t3)
sll $t3, $t3, 3
addu $t4, $t3, $t2
lw  $t5, 0($t4)
sw  $t5, 0($t2)
lw  $t5, 4($t4)
sw  $t5, 4($t2)
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
p[*q + 10] = a;
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
p[*q + 10] = a;
```

```
la    $t3, a
lb    $t3, 0($t3)
lw    $t4, 0($t1)
addiu $t4, $t4, 10
addu $t4, $t4, $t0
sb    $t3, 0($t4)
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
h = &h[*p];
```

Aritmética de punteros

- Dadas las siguientes declaraciones

```
char a;
int b;
long long int c;
main()
{
    char *p;          /* punter guardat en $t0 */
    int *q;           /* punter guardat en $t1 */
    long long int *h; /* punter guardat en $t2 */
    ...
}
```

Traduce a ensamblador las siguientes sentencias en C.

```
h = &h[*p];
```

```
lw    $t3, 0($t0)
sll   $t3, $t3, 3
addiu $t2, $t3, $t2
```



T2: Vectores

PhD Octavio Castillo-Reyes
octavio.castillo@upc.edu

Vectores

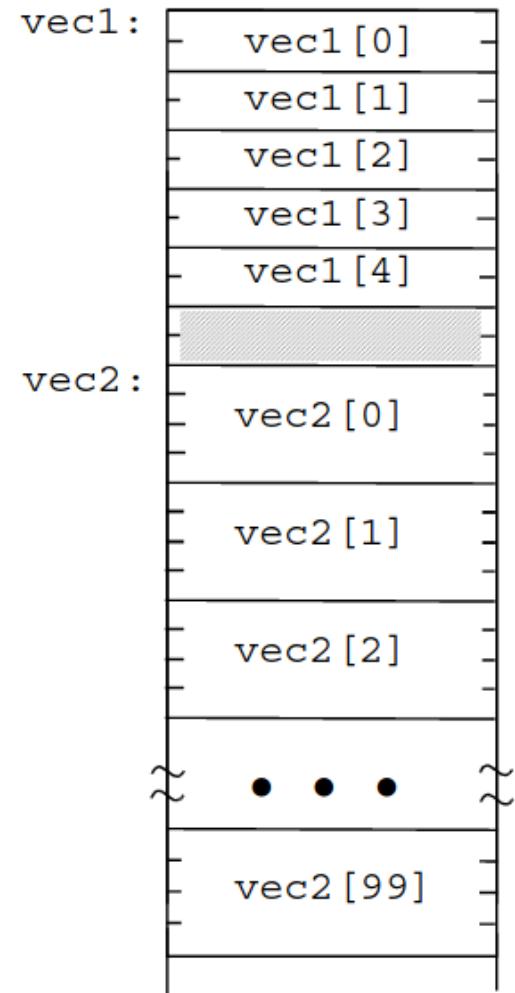
- Agrupaciones unidimensionales de elementos del mismo tipo, los cuales se identifican por un índice
- Los elementos se almacenan en posiciones consecutivas a partir de la dirección inicial del vector
- El primer elemento tiene índice 0
- En MIPS, los elementos han de respetar la **reglas de alineación**
 - Como todos los tipos tienen tamaño múltiplo de 2, si el primer elemento del vector está alineado el resto también
 - No es necesario insertar espacio intermedio entre los elementos

Vectores

```
/* En C */
short vec1[5] = {0, -1, 2, -3, 4};
int vec2[100];
```

```
# En MIPS
.data
vec1: .half 0, -1, 2, -3, 4
      .align 2
vec2: .space 400
```

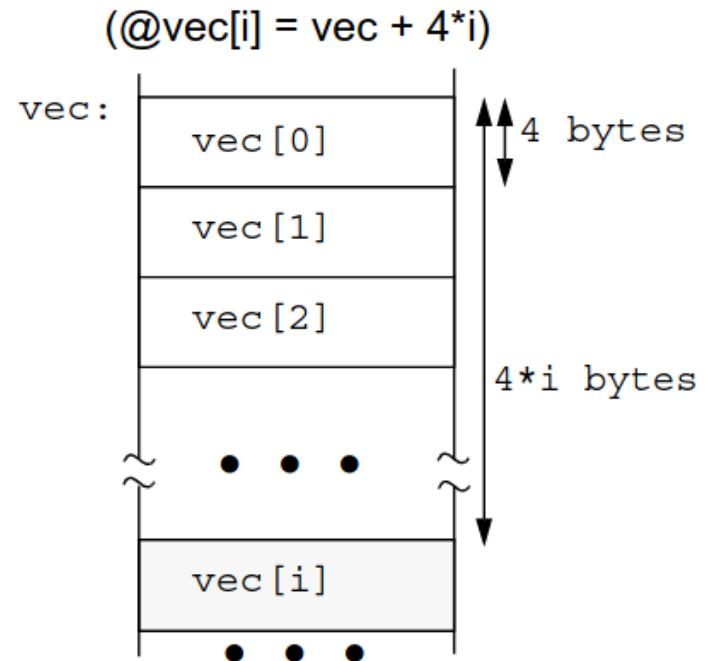
(@vec1 = 0x10010000)



Acceso a un elemento de un vector

- Para acceder a un elemento i de un vector hay que calcular su dirección
- Si los elementos tienen un tamaño T bytes, la dirección del elemento i se obtiene:

$$@vec[i] = @vec[0] + i \cdot T$$



Acceso a un elemento de un vector

- Indica el código en lenguaje ensamblador del MIPS asumiendo que i está en \$t0 y que *vec* es un vector global de enteros de 32 bits

```
vec[i] = 10;
```

Acceso a un elemento de un vector

- Indica el código en lenguaje ensamblador del MIPS asumiendo que i está en \$t0 y que `vec` es un vector global de enteros de 32 bits

```
vec[i] = 10;
```

```
la $t2, vec          # $t2 = @vec[0]
sll $t1, $t0, 2      # $t1 = 4 * i
addu $t2, $t2, $t1   # $t2 = @vec[0] + 4 * i
li $t1, 10
sw $t1, 0($t2)
```

Acceso a un elemento de un vector

- Dadas las siguientes declaraciones globales en C

```
int val[100], vec[100];
int elem;
```

- Traduce las siguientes sentencias en C a lenguaje ensamblador MIPS

```
elem = val[5] + vec[10];
elem = vec[10 + val[5]];
```

Vectores y punteros

- En C, los vectores tienen el mismo tipo que un puntero y siempre apuntan al primer elemento del vector
 - `int vec[100];`
 - `int *p;`
- Podemos inicializar un puntero asignandole un vector pero no al revés
 - `p = vec;`
- Podemos utilizar el operador [] con un puntero
 - `p[8] = 0;`
- Podemos utilizar el operador de indirección * con un vector
 - `*vec = 0;`

Vectores y punteros

- Un puntero se puede considerar como un vector, el primer elemento del cual es el apuntado por p
- La siguiente expresión es válida:
 - $*(p + i) = 0;$
- Y es equivalente a la expresión:
 - $p[i] = 0;$

Cadenas de caracteres (strings)

- Vectores con un número variable de caracteres
- El último carácter de la cadena (centinela) siempre vale 0 ('\0')
- String “Cap” se representa con los caracteres 67, 97, 112, 0

(vec = “Cap”;

vec:	
	'C' = 67
	'a' = 97
	'p' = 112
	'\0' = 0

Declaración de strings

- En C

```
char cadena [] = "Una frase";
```

- El compilador reserva 10 bytes: 9 caracteres más el centinela

- En MIPS

```
.data  
cadena: .ascii "Una frase"  
       .byte 0           # centinela
```

- Alternativa en MIPS

```
.data:  
cadena: .asciiz "Una frase"
```

Acceso a los elementos de un string

- En C los caracteres se codifican en ASCII (1 byte)
- Se acceden utilizando las instrucciones **lb** y **sb**
- Mismo método que se utiliza para acceder a vectores:
 - `char cadena[] = "Una frase";`
 - `@cadena[i] = @cadena[0] + i;`
 - Tamaño de elemento es siempre 1 (byte) para los strings
- Traduce a ensamblador MIPS la siguiente sentencia en C, asumiendo que las variables i y j están en los registros \$t0 y \$t1 respectivamente:
 - `cadena[i] = cadena[j] - 32;`

Ejercicios

- En C los caracteres se codifican en ASCII (1 byte)
- Se acceden utilizando las instrucciones **lb** y **sb**
- Mismo método que se utiliza para acceder a vectores:
 - `char cadena[] = "Una frase";`
 - `@cadena[i] = @cadena[0] + i;`
 - Tamaño de elemento es siempre 1 (byte) para los strings
- Traduce a ensamblador MIPS la siguiente sentencia en C, asumiendo que las variables i y j están en los registros \$t0 y \$t1 respectivamente:
 - `cadena[i] = cadena[j] - 32;`

Ejercicios

```
#define N 4
int V[N] = {0,1,2,3};
void main () {
    int suma=0, i=N;
    while ( i != 0) {
        i--;
        suma = suma + V[i];
    }
}
```

Ejercicios

```
.eqv N, 4                      # define N 4
.data
V: .word 0, 1, 2, 3            # vector d'enters
.text
main:
    move $t0, $zero          # suma = 0
    li   $t1, N              # i = N
    la   $t4, V              # @V[0] = 0x10010004. S'expandeix:
                            #      lui $at,0x10010000
                            #      ori $t4,$at,0x0004
while:
    bne $t1, $zero, fiwhile # salta si i!=0
    addiu $t1, $t1, -1       # i--
    sll  $t3, $t1, 2         # i*4
    addu $t3, $t3, $t4       # @V[0] + i*4
    lw   $t5, 0($t3)        # V[i]
    addu $t0, $t0, $t5       # suma = suma + V[i]
    b    while               # salta
fiwhile:
```

Ejercicios

```
i = 0;  
while ((x[i] = y[i]) != '\0')  
    i++;
```

Ejercicios

```
i = 0;  
while ((x[i] = y[i]) != '\0')  
    i++;
```

```
move $t0, $zero  
la   $t1, x  
la   $t2, y  
while:  
    addu $t3, $t0, $t2    # $t3 = @y[i]  
    lb   $t3, 0($t3)      # $t3 = y[i]  
    addu $t4, $t0, $t1    # $t4 = @x[i]  
    sb   $t3, 0($t4)      # x[i] = y[i]  
    beq  $t3, $zero, out  # Condició  
    addiu $t0, $t0, 1      # i++;  
    b    while             # Bucle  
out:
```

Ejercicios

Dadas las siguientes declaraciones de variables, almacenadas en memoria a partir de la dirección 0x10010000,

```
.data
A:    .word 5, 2
B:    .word 3
C:    .word 4, 0xFFFF, 0x4180
D:    .word 0x10010008, 0
```

Determina cuál es el valor de la variable B después de ejecutar el código

```
.text
la    $t1, C
lw    $t2, 12($t1)
lb    $t1, 8($t2)
la    $t3, B
lb    $t4, 0($t3)
addu $t1, $t1, $t4
sb    $t1, 0($t3)
```

Ejercicios

Dadas las siguientes declaraciones de variables, almacenadas en memoria a partir de la dirección 0x10010000,

```
.data
A:    .word  5, 2
B:    .word  3
C:    .word  4, 0xFFFF, 0x4180
D:    .word  0x10010008, 0
```

Determina cuál es el valor de la variable B después de ejecutar el código

```
.text
la    $t1, C
lw    $t2, 12($t1)
lb    $t1, 8($t2)
la    $t3, B
lb    $t4, 0($t3)
addu $t1, $t1, $t4
sb    $t1, 0($t3)
```

```
.text
la    $t1, C          $$t1 = 0x1001000C
lw    $t2, 12($t1)   $$t2 = 0x10010008
lb    $t1, 8($t2)    $$t1 = 0xFFFFFFFF
la    $t3, B          $$t3 = 0x10010008
lb    $t4, 0($t3)    $$t4 = 0x00000003
addu $t1, $t1, $t4   $$t1 = 0x00000002
sb    $t1, 0($t3)    #M[0x10010008] = 0x02. B val 2
```

Ejercicios

Dadas las siguientes declaraciones de variables, almacenadas en memoria a partir de la dirección 0x10010000,

```
.data
A:    .word  5, 2
B:    .word  3
C:    .word  4, 0xFFFF, 0x4180
D:    .word  0x10010008, 0
```

Explica qué hace el siguiente código

```
.text
la    $t1, A+1
lw    $t1, 0($t1)
```

Ejercicios

Dadas las siguientes declaraciones de variables, almacenadas en memoria a partir de la dirección 0x10010000,

```
.data
A:    .word  5, 2
B:    .word  3
C:    .word  4, 0xFFFF, 0x4180
D:    .word  0x10010008, 0
```

Explica qué hace el siguiente código

```
.text
la      $t1, A+1
lw      $t1, 0($t1)
```