Estructura de computadores

Grau Enginyeria Informàtica
Q2 2020/2021
FIB

Administración del curso

- Profesor: PhD Octavio Castillo Reyes
- octavio.castillo@upc.edu
- URL: http://docencia.ac.upc.edu/FIB/grau/EC/
- Libro: Computer Organization and Design: The Hardware/Software Interface,
- 5th, Ed. Morgan Kaufmann , 2013.
- Consultas:

  Quedar conmigo en clase o por email (horario a convenir)

Horarios

- Martes 08:00 – 10:00
- Jueves 08:00 – 10:00

Laboratorio:

Consultar horario en http://docencia.ac.upc.edu/FIB/grau/EC/

Estructura del curso

- Temario:
  - Tema 1: Introducción- Rendimiento de un procesador
  - Tema 2: Introducción al MIPS- Lenguaje ensamblador, tipos de datos básicos
  - Tema 3: Traducción de programas
  - Tema 4: Matrices
  - Tema 5: Aritmética de enteros y coma flotante
  - Tema 6: Memoria caché
  - Tema 7: Memoria virtual
  - Tema 8: Excepciones e interrupciones
- Prácticas:
  - Práctica 0: Ensamblador
  - Práctica 1: Punteros
  - Práctica 2: Subrutinas
  - Práctica 3: Matrices
  - Práctica 4: Coma flotante
  - Práctica 5: Memoria caché

Evaluación del curso

- NOTA = 0,20*max(EP, EF) + 0'60*EF + 0,20*(EL*0'85 + AC*0'15)
  EP = Examen Parcial
  EL = Examen de Laboratori
  EF = Examen Final
  AC = Avaluació Continuada de Laboratori
- Les dates dels 3 exàmens EP, EL, EF, i de les sessions de laboratori apareixen al Calendari
- L'Avaluació Continuada del laboratori (AC) es realitza durant cada sessió, i es compon de:
  - Estudi Previ, individual: resposta dels Exercicis del Quadern de Laboratori, fet a casa i lliurat en format electrònic al Racó abans del començament de cada sessió.
  - Activitats, individual: realització amb l'ordinador de les Activitats del Quadern de Laboratori, al llarg de la sessió presencial.

Tema 1

Introducción-
Rendimiento de un procesador

T1: Tecnología computadores

- 1.1 Performance
- 1.2 Multiprocesadores
- 1.3 Ley de Ahmdal

Tiempo de respuesta y rendimiento

- Tiempo de respuesta
  - Cuanto tiempo se necesita para hacer una tarea
- Rendimiento
  - Total trabajo hecho por unidad de tiempo
    - ejemplo: tareas/transacciones/… por hora
- Como afectan al tiempo de respuesta y al rendimiento
  - Reemplazar el procesador por una versión más rápida?
  - Añadir más procesadores?
- Por ahora, en el tiempo de respuesta…

Performance relativo

- Definimos: Performance = 1/Tiempo ejecución
  - "X es *n* veces más rápido que Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Tiempo ejecución}_Y / \text{Tiempo ejecución}_X = n$$

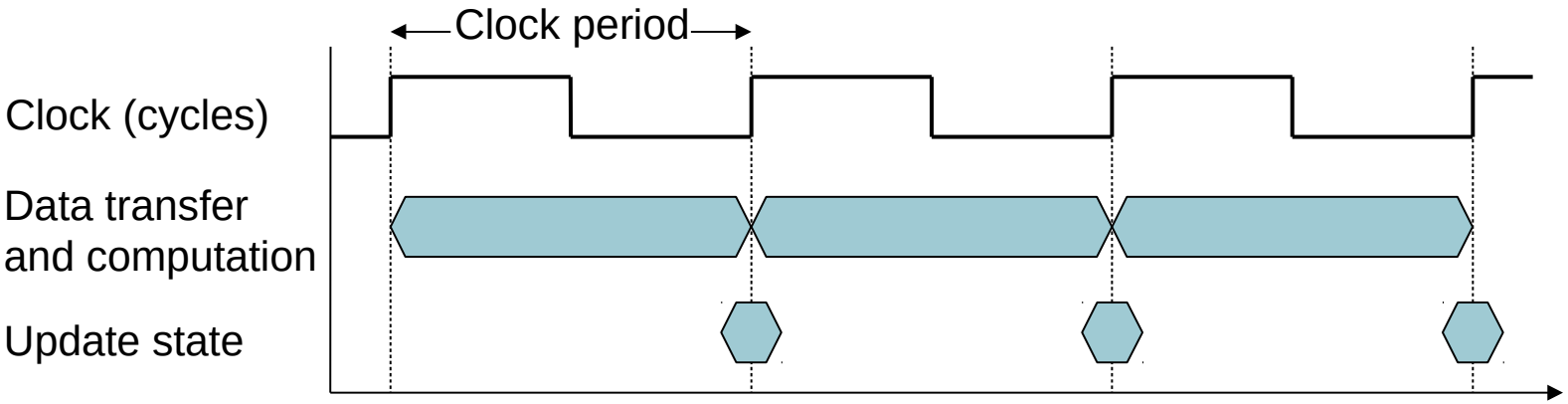
- Ejemplo: tiempo para ejecutar un programa
  - 10s en A, 15s en B
  - $\text{Tiempo ejecución}_B$ / $\text{Tiempo ejecución}_A$
    = 15s / 10s = 1.5
  - A es 1.5 veces más rápido que B

Medida del Tiempo de ejecución

- “Elapsed” time
  - Tiempo total de respuesta, incluyendo todos los aspectos
    - Procesado, E/S, SO overhead, idle time
  - Determina el performance del sistema
- Tiempo CPU
  - Tiempo requerido en el procesado de un trabajo
    - Excluye el tiempo E/S, compartición de trabajos
  - Comprende el tiempo de CPU de usuario y sistema
  - Differentes programas son distintamente impactados por el performance de CPU y sistema

CPU Clocking

- La operativa del hardware está gobernada por un clock a frecuencia constante



  - Período Clock: duración del ciclo de reloj
    - e.g., 250ps = 0.25ns = 250×10^{-12}s
  - Frecuencia Clock (rate): ciclos por segundo
    - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

Tiempo CPU

$$\text{Tiempo CPU} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Mejorar performance (tiempo CPU --)
  - Reducir número de clock cycles
  - Incrementar clock rate
  - Compromiso de diseño entre ambos valores

Tiempo CPU (Ejemplo)

- Computador A: 2GHz clock, 10s tiempo CPU
- Queremos diseñar Computador B con 2 objetivos:
  - 6s tiempo CPU
  - Clock más rápido, 1.2 × clock cycles
- Cuan rápido debe ser el clock del Computador B?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Número de instrucciones (IC) y CPI

- Número medio de ciclos de clock por instrucción (CPI)
- Instrucciones necesarias para realizar un programa (IC)

$$\text{CPU Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instrucciones por programa
  - Determinadas por el programa, ISA y compilador
- Media de ciclos por instrucción
  - Determinados por el hardaware de la CPU
  - Si diferentes instrucciones tienen distintos CPI…
    - La media de CPI estará afectada por la mezcla de instrucciones

ISA = Instruction Set Architecture

CPI Ejemplo

- Computador A: Cycle Time = 250ps, CPI = 2.0
- Computador B: Cycle Time = 500ps, CPI = 1.2
- Mismo ISA
- Cual es el más rápido y por cuánto?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A es más rápido..

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…por esto

I = Número de instrucciones por CPU para el programa constante

CPI en más detalle

- Si diferentes clases de instrucciones requieren diferentes numéros de ciclos

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Valor medio CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Frecuencia relativa}} \right)$$

CPI Ejemplo

- Opciones de secuencias de código compilado usando instrucciones en clases A, B y C, para 2 secuencias de código

Clase	A	B	C
CPI para clase	1	2	3
IC en secuencia 1	2	1	2
IC en secuencia 2	4	1	1

- Secuencia 1: IC = 5
  - Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0
- Secuencia 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9 !!!!!! Más ràpido
  - Avg. CPI = 9/6 = 1.5
- Sec 2 és más rápida aún con más instrucciones, so CPI menor

Performance: Resumen

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depende de:
  - Algoritmo: afecta IC, y posiblemente a CPI
  - Lenguaje de programación: afecta IC, CPI
  - Compilador: afecta IC, CPI
  - ISA: afecta IC, CPI, T_c (tiempo de clock)

Multiprocesadores

- Microprocesadores multinúcleo
  - Más de un procesador por chip
- Requiere explícitamente paralelización (*explicitly parallel programming*)
  - Comparado con paralelismo a nivel de instrucciones → Segmentación
    - Hardware ejecuta múltiples instrucciones a la vez
    - Oculto al programador
  - Difícil de implementar (el paralelismo)
    - Implica programar parelelamente para optimizar el performance
      - Debe solucionar el problema, ser correcto, tener un interfaz útil, y además……ser rápido
    - Distribución de carga uniformemente
    - Optimizar la comunicación y sincronización

SPEC CPU Benchmark

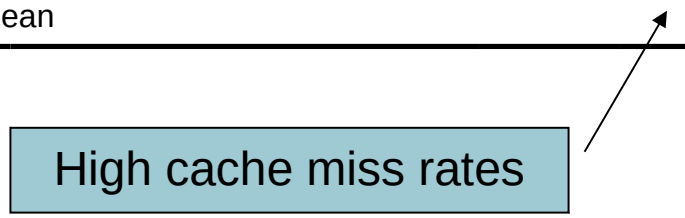
- Programas usados para medir performance
  - Supuestamente reales con carga real
- Standard Performance Evaluation Corp (SPEC)
  - Desarrolla benchmarks para CPU, I/O, Web, …
- SPEC CPU2006
  - Elapsed time para ejecutar una selección de programas
    - Negligible I/O, o sea se centra en CPU performance
  - Normalizado
  - Resume en una media geométrica de ratio de performance (12 CINT2006 (integer) and 17 CFP2006 (floating-point)

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

SPEC = System Performance Evaluation Cooperative

CINT2006 para Opteron X4 2356

Name	Description	IC×10^9	CPI	Tc (ns)	Exec time	Ref time	SPECratio
perl	Interpreted string processing	2,118	0.75	0.40	637	9,777	15.3
bzip2	Block-sorting compression	2,389	0.85	0.40	817	9,650	11.8
gcc	GNU C Compiler	1,050	1.72	0.47	24	8,050	11.1
mcf	Combinatorial optimization	336	10.00	0.40	1,345	9,120	6.8
go	Go game (AI)	1,658	1.09	0.40	721	10,490	14.6
hmmer	Search gene sequence	2,783	0.80	0.40	890	9,330	10.5
sjeng	Chess game (AI)	2,176	0.96	0.48	37	12,100	14.5
libquantum	Quantum computer simulation	1,623	1.61	0.40	1,047	20,720	19.8
h264avc	Video compression	3,102	0.80	0.40	993	22,130	22.3
omnetpp	Discrete event simulation	587	2.94	0.40	690	6,250	9.1
astar	Games/path finding	1,082	1.79	0.40	773	7,020	9.1
xalancbmk	XML parsing	1,058	2.70	0.40	1,143	6,900	6.0
Geometric mean							11.7



Ley de Amdahl

- Mejorar un aspecto de un computador y esperar que repercuta en una mejora proporcional en el performance global

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

- Ejemplo: Programa que tarda 100 s de los cuales 80 son multiplicaciones
  - Cuánto debo multiplicar la velocidad de la multiplicación si quiero que mi programa corra 5 veces más rápido?

$$20 = \frac{80}{n} + 20$$

  - Imposible

- Corolario: hacer más rápido lo que más suceda

MIPS, una medida de performance

- MIPS: Millones de Instrucccciones Por Segundo
  - No representa
    - Diferencias en ISAs entre computadores
    - Diferencias en complejidad entre instrucciones

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$= \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

  - Si un programa ejecuta más instrucciones que otro, pero las instrucciones son más rápidas, MIPS variará independientemente del performance

Conclusión

- Cost/performance ratio está mejorando
  - Debido al desarrollo de las tecnologías subyacentes
- Niveles de abstracción jerárquicos
  - Tanto en hardware como en software
- Instruction set architecture (ISA)
  - El hardware/software interface
- Tiempo de ejecución: la mejor medida de performance

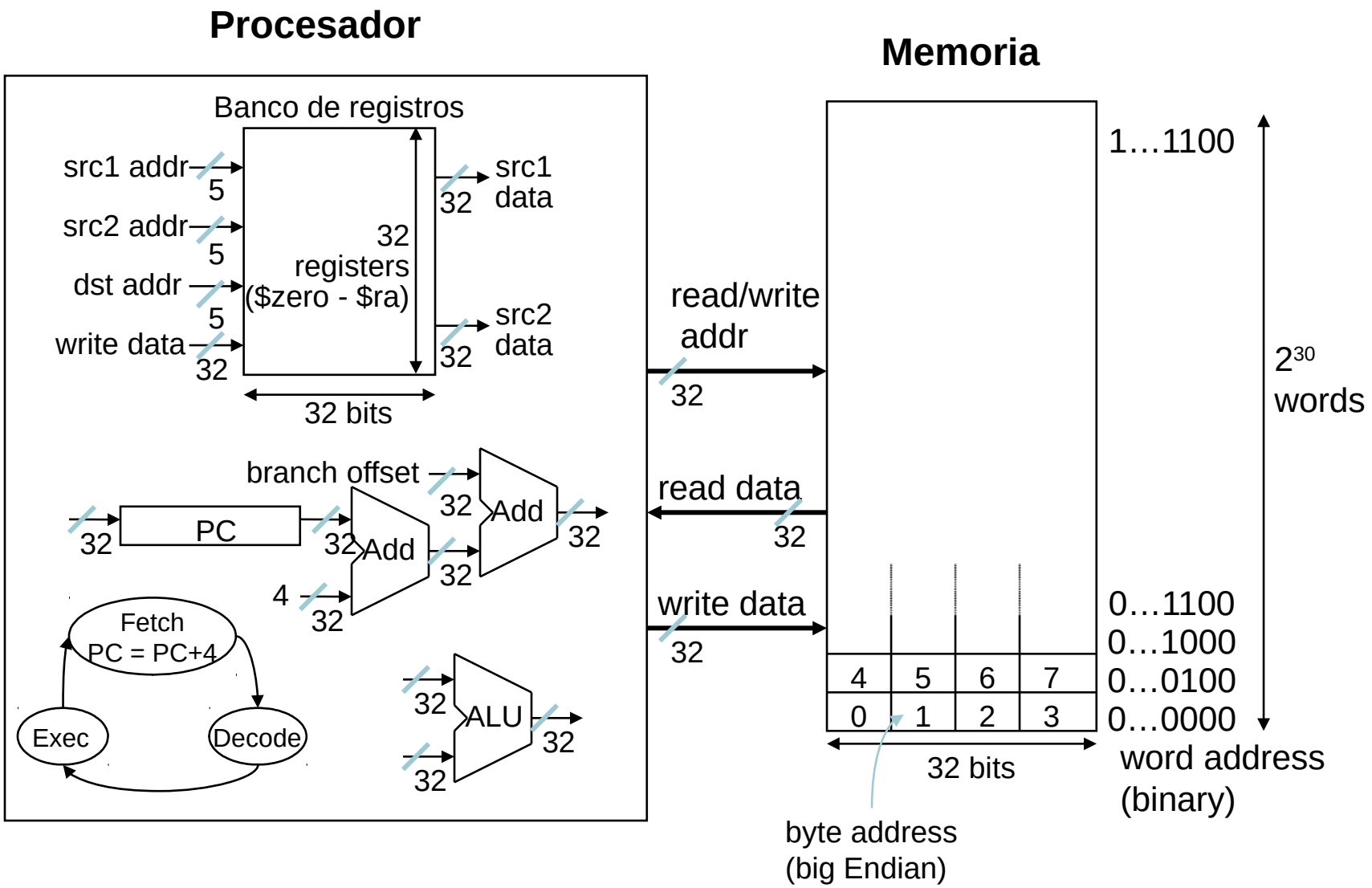
Introducción al MIPS-Lenguaje ensamblador

PhD Octavio Castillo Reyes
(octavio.castillo@upc.edu)

MIPS (RISC) Principios de diseño

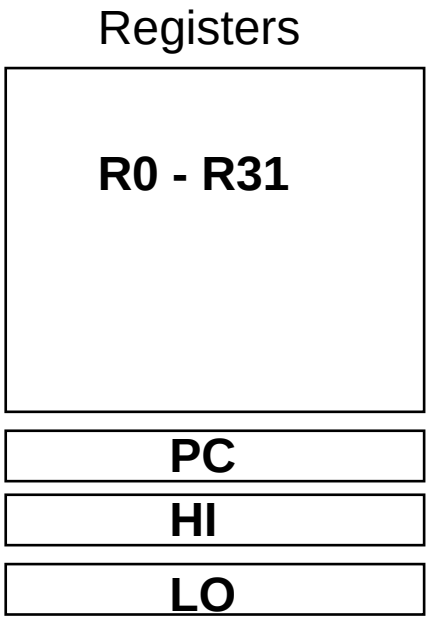
- La simplicidad favorece la regularidad
  - Instrucciones de tamaño fijo
  - Reducido número de formatos de instrucción (3)
  - opcode siempre los 6 primeros bits
- Cuanto más pequeño más rápido
  - Set limitado de instrucciones
  - Limitado número de registros
  - Limitado número de modos de direccionamiento
- Hacer rápido lo más frecuente
  - En las operaciones aritméticas registro hay que hacer primero el load (2 instrucciones)→ Las instrucciones con operandos inmediatos son más ‘rápidas’
- Un buen diseño requiere buenos compromisos
  - Tres formatos de instrucción

Organización del MIPs

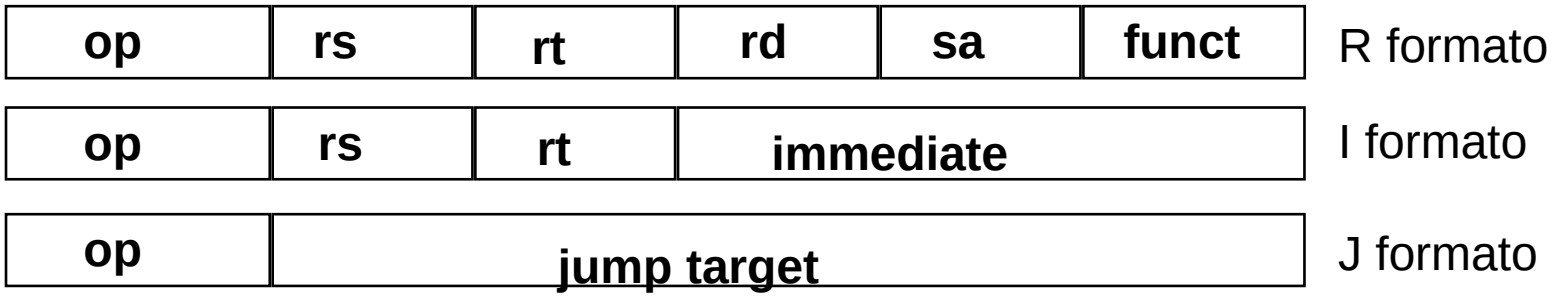


MIPS-32

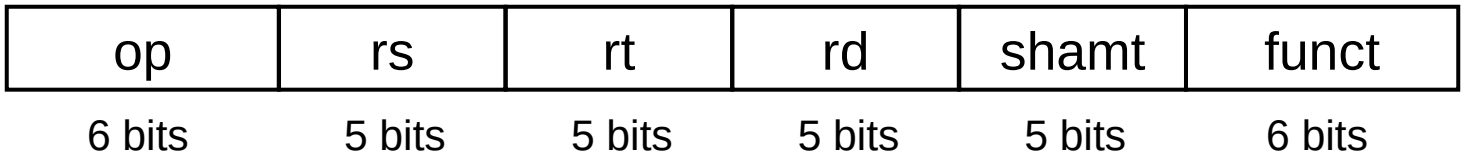
- Microprocessor without Interlocked Pipeline Stages
- Categoría de instrucciones
  - Computacionales
  - Load/Store
  - Jump y Branch
  - Punto flotante
    - coprocessor
  - Manejo de la memoria
  - Especiales



3 Formatos de instrucción: todos de 32 bits



Instrucciones MIPS R-format



- Campos de instrucción
  - op: código de operación (opcode)
  - rs: Número del 1er registro fuente
  - rt: Número del 2º registro fuente
  - rd: Número del registro destino
  - shamt: cantidad a desplazar (00000 por ahora)
  - funct: código de función (extiende el opcode)

R-format ejemplo

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

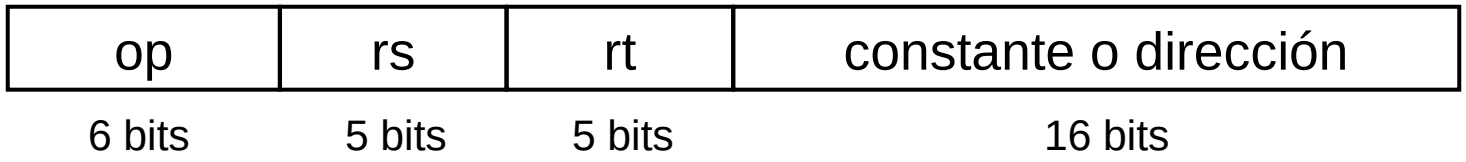
add $t0, $s1, $s2

$t0 - $t7 ␣ 8:15
$s0 - $s7 ␣ 16:23

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Instrucciones MIPS I-format



- Aritméticas inmediatas e instrucciones de load/store
  - rs/rt: Número de registro fuente o destino
  - Constante (Ca2 de 16 bits): -2^{15} to $+2^{15} - 1$
  - Dirección: offset añadido a dirección base en rs
- *Principio de diseño 4:* Los buenos diseños requieren buenos compromisos
  - Diferentes formatos complican la decodificación pero permiten tener todas las instrucciones 32-bits
  - Mantener los formatos lo más similares posibles

MIPS Instrucciones Aritméticas

- Ensamblador MIPS: convenio

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

- Cada operación aritmética se realiza en una instrucción (igual que SISA-F)
- Cada una especifica exactamente 3 operandos que se encuentran en el banco de registros (`$t0,$s1,$s2`)

destination ← source1 op source2

- Formato de Instrucción (R formato)

MIPS Instrucciones Aritméticas

- Ensambaldor MIPS convenio aritmético

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

- Cada operación aritmética se realiza en una instrucción
- Cada una especifica exactamente 3 operandos que se encuentran en el banco de registros (`$t0,$s1,$s2`)

destination ← source1 op source2

- Formato de Instrucción (R formato)

0	17	18	8	0	0x22

MIPS Instrucciones Aritméticas

- Ensambaldor MIPS convenio aritmético

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

- Cada operación aritmética se realiza en una instrucción
- Cada una especifica exactamente 3 operandos que se encuentran en el banco de registros (`$t0,$s1,$s2`)

destination ← source1 op source2

- Formato de Instrucción (R formato)

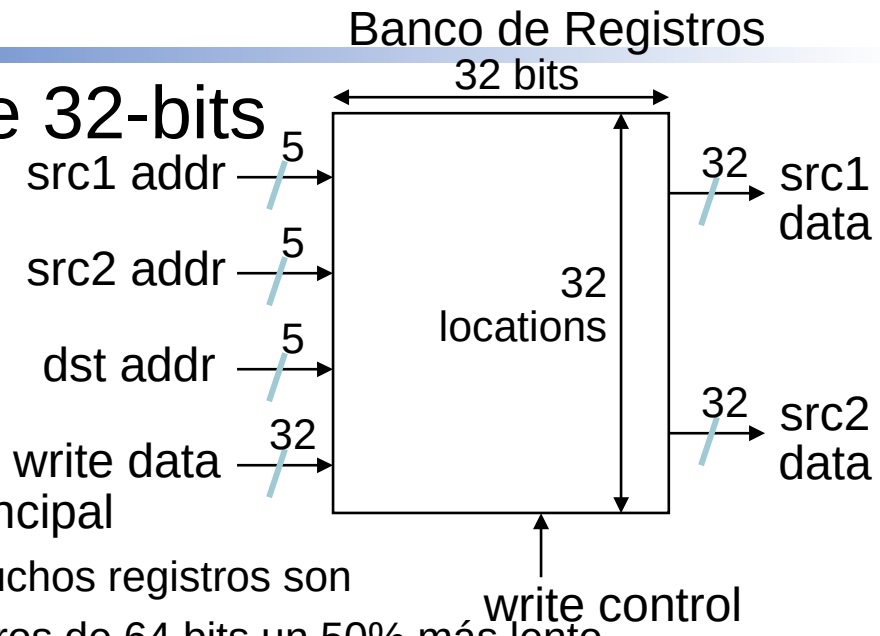
MIPS: Campos de la instrucción

- MIPS se dan nombres a los campos de la instrucción para facilitar el entenderlos

op	6-bits	opcode que especifica la operación
rs	5-bits	registro fuente 1
rt	5-bits	registro fuente 2
rd	5-bits	registro destino
shamt	5-bits	bits a desplazar (para instrucciones de shift)
funct	6-bits	código de función para completar el op

MIPS: Banco de Registros

- Contiene 32 registros de 32-bits
  - Tiene 2 puertos de lectura
  - 1 puerto de escritura



- Los registros son
  - Más rápidos que la memoria principal
    - Pero banco de registros con muchos registros son más lentos (ejem. banco de registros de 64 bits un 50% más lento que de 32 bits)
    - El número de puertos de lectura escritura tiene un impacto cuadrático en la velocidad.
  - Pueden tener variables
    - Por lo tanto la densidad del código mejora (ya que los registros pueden ser nombrados con menos bits que una posición de memoria)

Registros MIPS: Convención

Nombre	Número de Registro	Uso	Preservar en una llamada?
$zero	0	constante 0 (hardware)	n.a.
$at	1	reservado para ensamblador	n.a.
$v0 - $v1	2-3	Valores retornados (subrutina)	no
$a0 - $a3	4-7	Argumentos (subrutina)	si
$t0 - $t7	8-15	temporales	no
$s0 - $s7	16-23	Valores guardados	si
$t8 - $t9	24-25	temporales	no
$gp	28	Puntero global	si
$sp	29	Puntero de pila	si
$fp	30	Puntero de trama	si
$ra	31	Dirección de retorno (subrutina)	si

Ejemplo:Compilación de una asignación en C usando registros

f = (g+h) - (i+j); *ejemplo pág. 81*

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

```
add $t0, $s1, $s2    # el registro $t0 contiene g+h
add $t1, $s3, $s4    # el registro $t1 contiene i+j
sub $s0, $t0, $t1    # f se carga con $t0-$t1
```

Ejercicios

Las variables a, b, c, d, e, f, g, h, se asignan a los registros $s0 : $s7, respectivamente:

h = (f - a + d) - b + c + (d - e + g)

a = (b + c + d + e) - (f - g - h) + (f - c + g - d)

e = (a + b + c + d) - (f + d - c + h + g)

a = (a + b) - (b - c) + (c + d) - (d - e) + (f + g) - (g - h)

MIPS: Instrucciones de acceso a memoria

- MIPS tiene 2 instrucciones de transferencia de datos para acceder a memoria

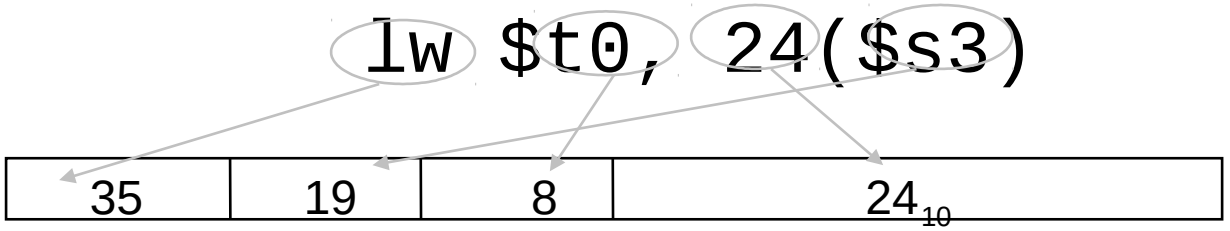
```
lw $t0, 4($s3) #traer word desde memoria
sw $t0, 8($s3) #guardar word a memoria
```

- El dato es cargado (lw) o guardado (sw) desde un registro del banco de registros – una dirección de 5 bits

- La dirección de memoria– de 32 bits – está formada sumando el contenido del registro dirección base al valor de offset
- El offset está limitado a un campo de 16 bits lo que limita las direcciones de memoria dentro de una región de $\pm 2^{13}$ o 8,192 words ($\pm 2^{15}$ o 32,768 bytes) desde la dirección del registro dirección base

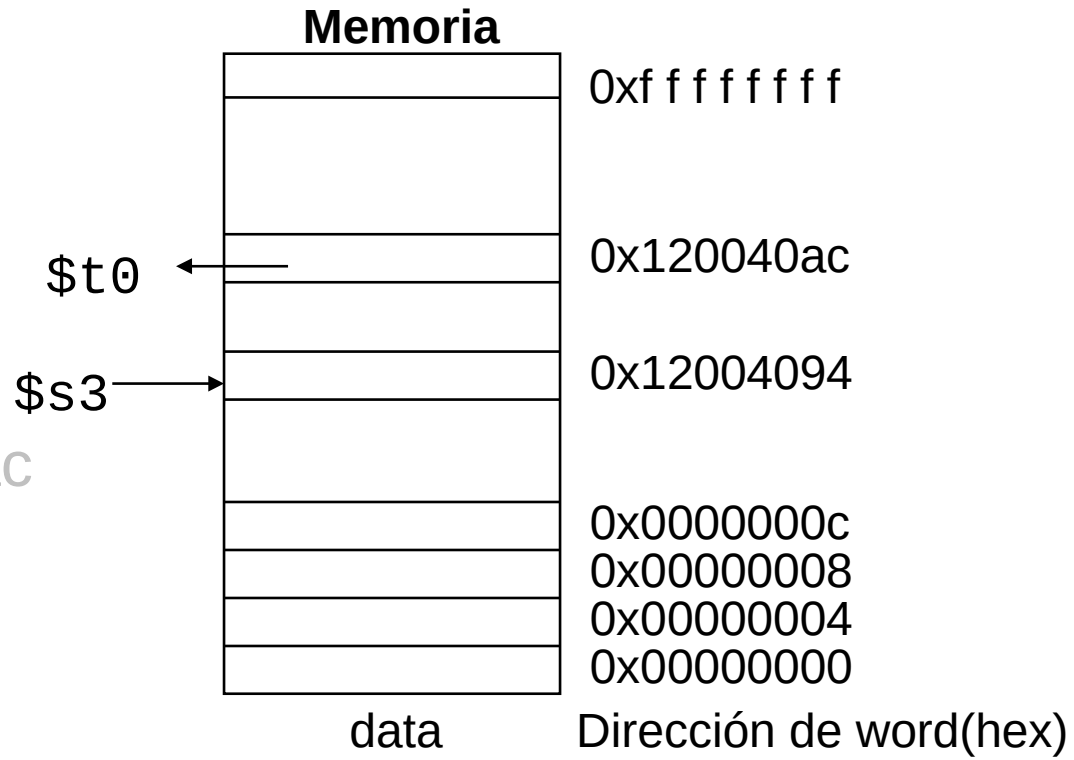
Lenguaje Máquina- Instrucción de Load

- Formato de Instrucciones Load/Store (I format):



24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Direcciones de byte

- Dado que los byte (8-bits) son muy utilizados, muchas arquitecturas permiten direccionar bytes individuales en memoria
  - Restricción de alineamiento – la dirección de memoria de un word (4 bytes) debe ser un multiplo de 4 en MIPS-32
- Big Endian: IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
- Ejemplo: Word en @ 0x48 y valor 0x12345678
  - En Little Endian se almacenaría:
    - 0x48 78
      0x49 56
      0x4A 34
      0x4B 12

      para acceder al 3er byte a la dirección inicial del word (0x48) le sumamos 2 y con 0x48+2=0x4A leyendo el dato 34
  - En Big Endian se almacenaría:
    - 0x48 12
      0x49 34
      0x4A 56
      0x4B 78

      para acceder al 3er byte a la dirección inicial del word (0x48) le sumamos (4-3)=1 con 0x49 leemos el dato 34

Load y Store de Bytes

- MIPS proporciona instrucciones especiales para mover bytes

```
lb  $t0, 1($s3)   #load byte from memory
sb  $t0, 6($s3)   #store byte to  memory
```

- Qué 8 bits son cargados y guardados?
  - load de un byte pone el byte de memoria en los 8 bits más a la derecha del registro destino
    - Qué pasa con los otros bits del registro? Se extiende el signo del byte en el caso de ***lb*** pero no de ***lbu*** (load byte unsigned)
  - store de un byte toma el byte de los 8 bits más a la derecha del registro y lo escribe en un byte de memoria
    - Qué pasa con los otros bits del word de memoria? Nada

Mapa de códigos de las operaciones en MIPS

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Mapa de códigos de las operaciones en MIPS

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Used to add constants

Mapa de códigos de las operaciones en MIPS

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

Mapa de códigos de las operaciones en MIPS

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant

Mapa de códigos de las operaciones en MIPS

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned

Mapa de códigos de las operaciones en MIPS

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Traducción de lenguaje MIPS a lenguaje máquina

```
A[300]= h + A[300];        # ejemplo página 98
```

Supongamos que $t1 tiene la base de la tabla A y $s2 se corresponde con h.

```
lw $t0, 1200($t1)          #en $t0 A[300]
add $t0,$s2,$t0            #en $t0 h+A[300]
sw $t0, 1200($t1)          #h+A[300] se almacena en A[300]
```

Traducción de lenguaje MIPS a lenguaje máquina

```
A[300]= h + A[300];        # ejemplo página 98
```

Supongamos que $t1 tiene la base de la tabla A y $s2 se corresponde con h.

```
lw $t0, 1200($t1)     #en $t0 A[300]
add $t0,$s2,$t0       #en $t0 h+A[300]
sw $t0, 1200($t1)        #h+A[300] se almacena en A[300]
```

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Traducción de lenguaje MIPS a lenguaje máquina

```
A[300]= h + A[300];        # ejemplo página 98
```

Supongamos que $t1 tiene la base de la tabla A y $s2 se corresponde con h.

```
lw $t0, 1200($t1)     # en $t0 A[300]
add $t0,$s2,$t0       # en $t0 h+A[300]
sw $t0, 1200($t1)     # h+A[300] se almacena en A[300]
```

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Instrucciones en lenguaje máquina en decimal:

op	*rs*	*rt*	*rd*	*@/shamt*	*funct*
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

Traducción de lenguaje MIPS a lenguaje máquina

```
A[300]= h + A[300];        # ejemplo página 98
```

Supongamos que $t1 tiene la base de la tabla A y $s2 se corresponde con h.

```
lw $t0, 1200($t1)     #en $t0 A[300]
add $t0,$s2,$t0       #en $t0 h+A[300]
sw $t0, 1200($t1)        #h+A[300] se almacena en A[300]
```

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Instrucciones en lenguaje máquina en binario:

op	*rs*	*rt*	*rd*	*@/shamt*	*funct*
100011	01001	01000	000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	000 0100 1011 0000		

[1]Mirar tabla de registros de página B-24 del libro o transp 37

[2]Mirar mapa de códigos de transp anterior o pág. B-50 del libro

Traducción de lenguaje máquina a ensamblador MIPS

¿Cuál es la instrucción en ensamblador que corresponde a la siguiente instrucción en lenguaje máquina? (Ejemplo página 134 libro)

`00af8020hex`

En binario:

```
31      2726    23      19      15      11      7  5 4  3 2 10
0000 0000 1010 1111 1000 0000 0010 0000

  op        rs      rt      rd     shamt    funct
000000 00101 01111 10000 00000 100000
```

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

Traducción de lenguaje máquina a ensamblador MIPS

¿Cuál es la instrucción en ensamblador que corresponde a la siguiente instrucción en lenguaje máquina? (Ejemplo página 134 libro)

`00af8020hex`

En binario:

```
31      2726    23      19      15      11      7  5 4  3 2 10
0000 0000 1010 1111 1000 0000 0010 0000

  op        rs      rt      rd     shamt    funct
000000 00101 01111 10000 00000 100000
```

Instrucción R-format y como op=0 hay que mirar funct=100000->*add*

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Traducción de lenguaje máquina a ensamblador MIPS

¿Cuál es la instrucción en ensamblador que corresponde a la siguiente instrucción en lenguaje máquina? (Ejemplo página 134 libro)

```
00af8020hex
```

En binario:

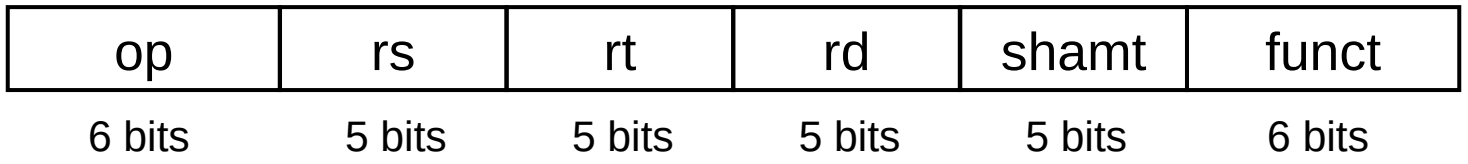
```
31     2726    23     19      15     11      7  5 4  3 2 10
0000 0000 1010 1111 1000 0000 0010 0000

 op        rs     rt      rd    shamt   funct
000000 00101 01111 10000 00000 100000
```

Instrucción R-format y como op=0 hay que mirar funct=100000->*add*

`rs=5($a1), rt=15($t7) y rd=16($s0), el shamt no se usa en add`

```
add $s0,$a1,$t7
```

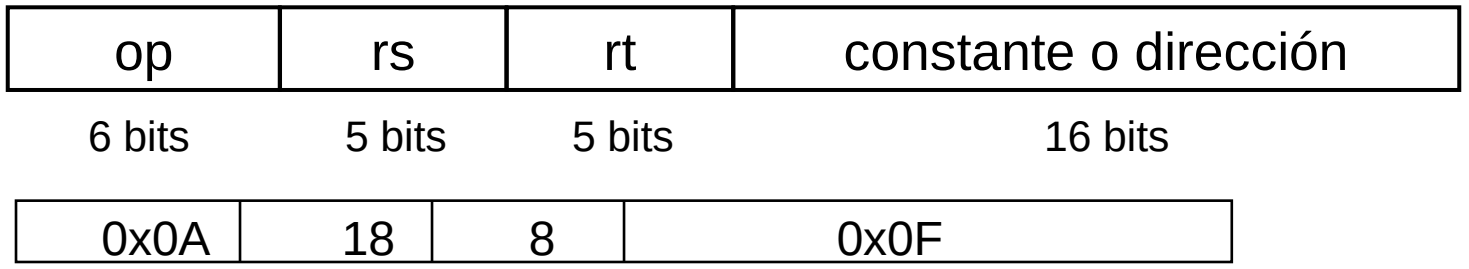


MIPS: Instrucciones con inmediatos

- Constantes se usan a menudo en un código típico
- Posibles aproximaciones?
  - poner “constantes” típicas en memoria y hacer load
  - crear registros hardware (como $zero) para constantes como 1
  - tener instrucciones especiales que contienen constantes!

```
addi $sp, $sp, 4  #$sp = $sp + 4
slti $t0, $s2, 15   #$t0 = 1 if $s2<15
```

- Formato máquina (I format):

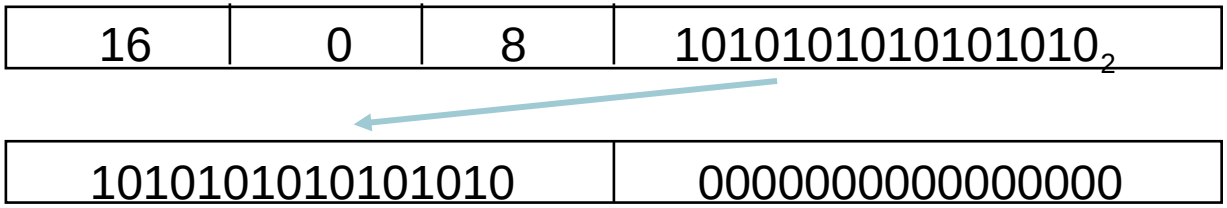


- La constante está en la instrucción misma!
  - El I format limita los valores del inmediato al rango $+2^{15}-1$ to -2^{15} (Ca2)

Qué pasa con constantes mayores?

- Queremos ser capaces de cargar una constante de 32 bits en un registro, para ello necesitamos 2 instrucciones
- Una nueva instrucción "load upper immediate"

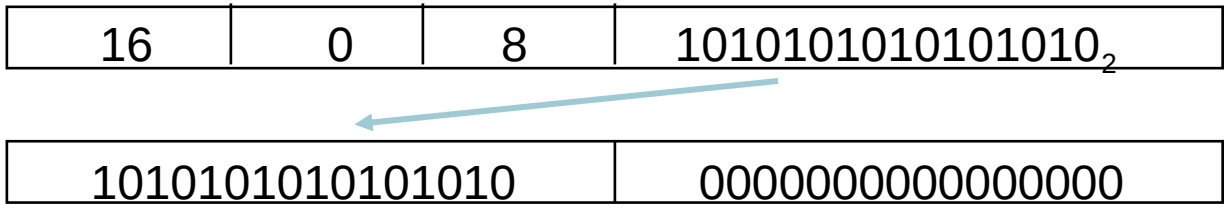
```
lui $t0, 1010101010101010
```



Qué pasa con constantes mayores?

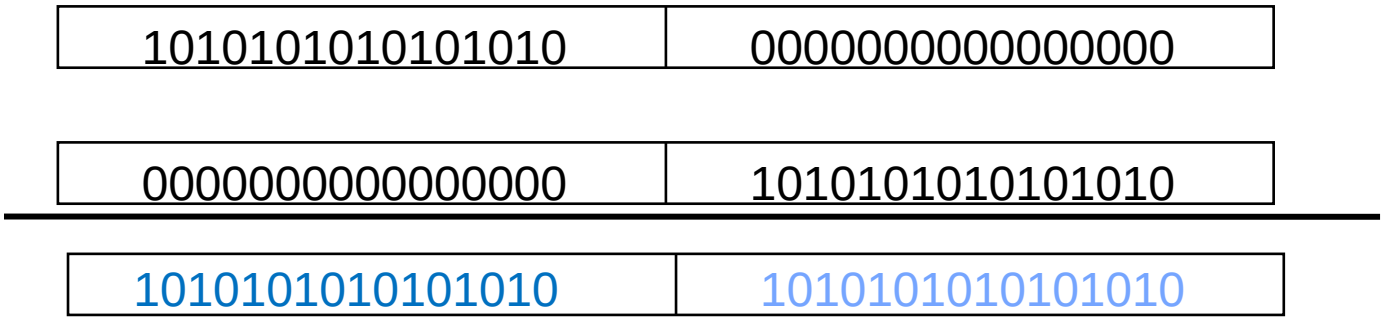
- Queremos ser capaces de cargar una constante de 32 bits en un registro, para ello necesitamos 2 instrucciones
- Una nueva instrucción "load upper immediate"

```
lui $t0, 1010101010101010
```

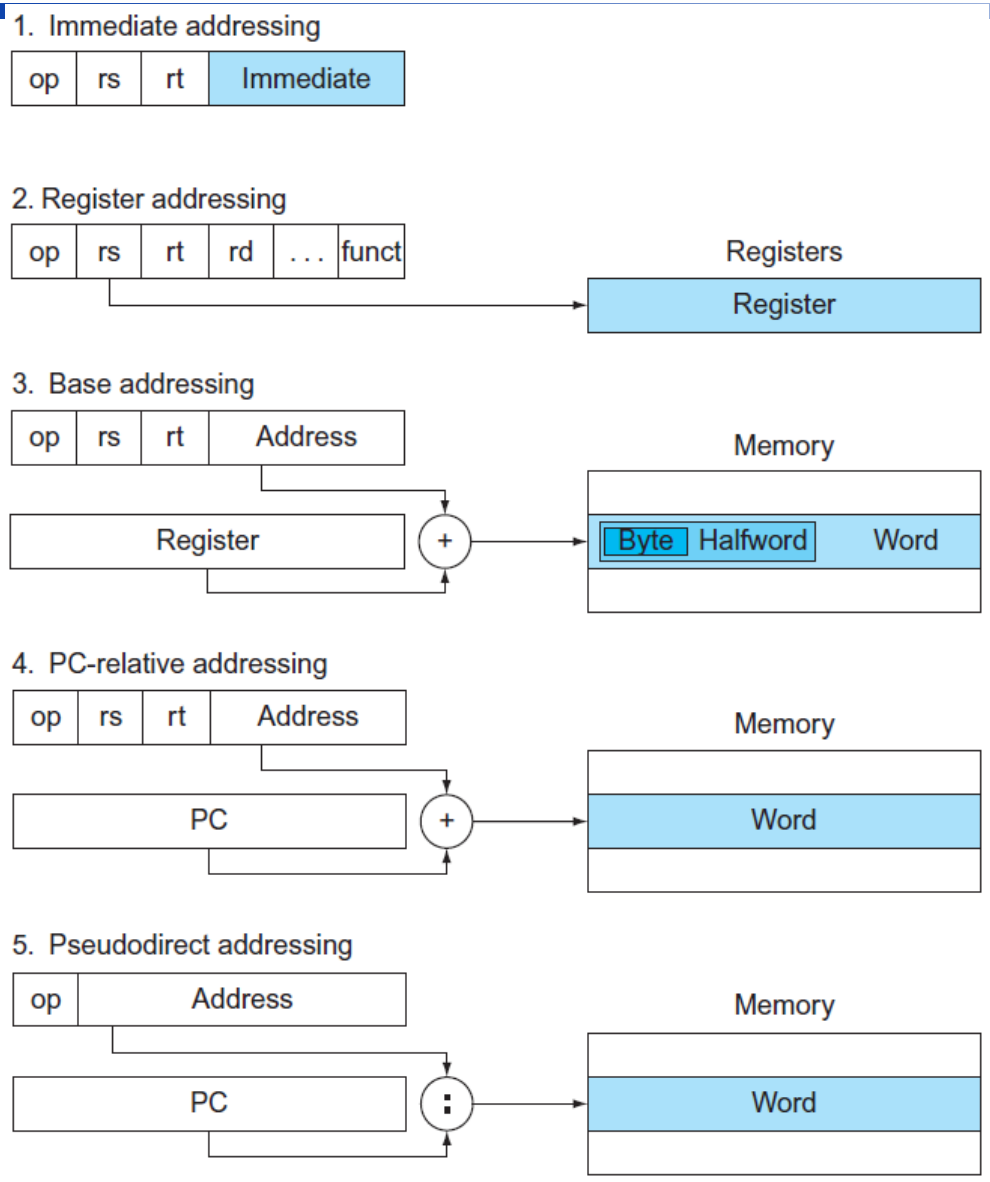


- Para cargar los 16 bits bajos, usamos

```
ori $t0, $t0, 1010101010101010
```



Modos de direccionamiento



addi $t1, $t0, 1

sub $t1, $t2, $s6

lw $t0, 6($t1)

bne/beq $s0, $s1, 2

j label_1

Repaso: Binarios ‘unsigned’ representación

Hex	Binario	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	$2^{32} - 4$
0xFFFFFFFD	1…1101	$2^{32} - 3$
0xFFFFFFFE	1…1110	$2^{32} - 2$
0xFFFFFFFF	1…1111	$2^{32} - 1$

2^{31} 2^{30} 2^{29} . . . 2^3 2^2 2^1 2^0 peso del bit

31 30 29 . . . 3 2 1 0 posición del bit

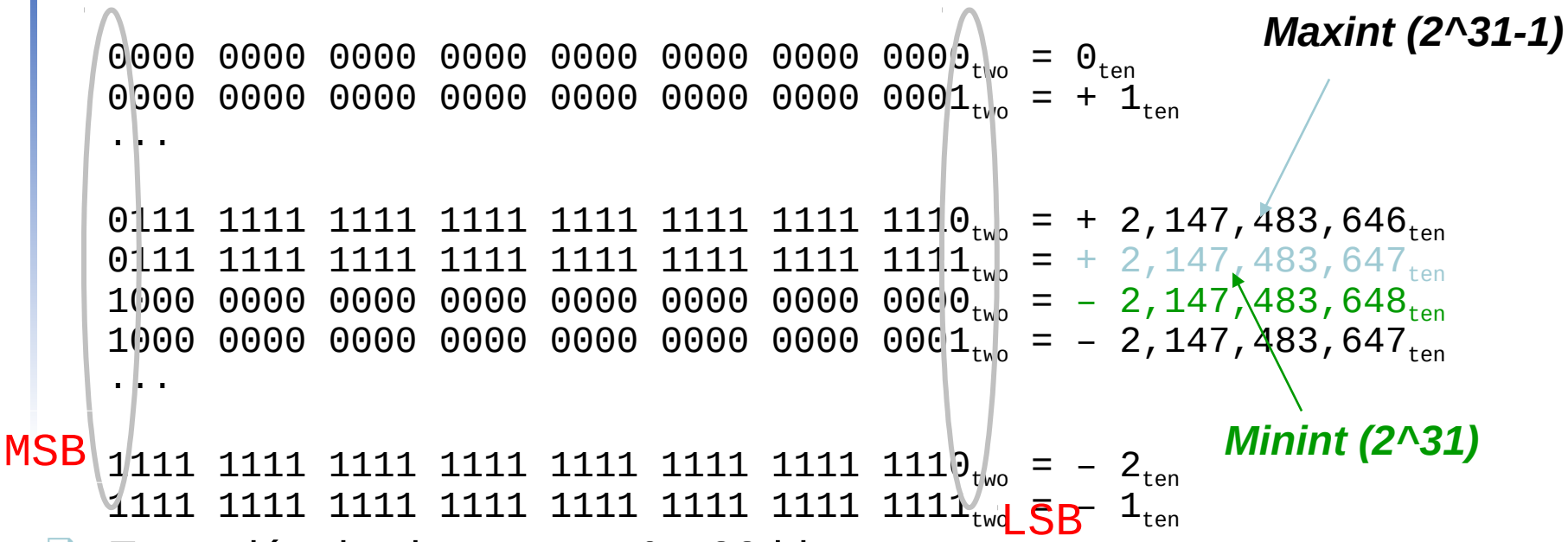
1 1 1 . . . 1 1 1 1 bit

1 0 0 0 . . . 0 0 0 0 - 1

$2^{32} - 1$

Representación de números

- Números con signo de 32 bits (Ca2):



- Extensión de signo para nºs<32 bits
  - Replicar el bit más significativo (bit de signo) en los bits 'vacíos'

```
0010  -> 0000 0010 (ej: extensión de 4 a 8 bits)
1010  -> 1111 1010
```

    - Extensión de signo versus zero extend (añadir 0s): `lb` vs. `lbu`

Repaso:Binarios con signo representación (Ca2)

	Ca2	decimal
-2^3 =	1000	-8
$-(2^3 - 1)$ =	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
$2^3 - 1$ =	0111	7

Complementar todos los bits

0101

y sumar 1

1011

y sumar 1

0110

1010

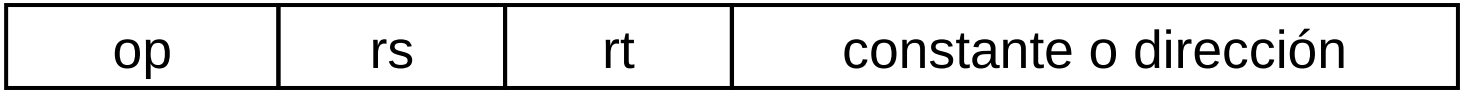
Complementar todos los bits

Ejercicio

Completar los campos para cada instrucción

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits



6 bits 5 bits 5 bits 16 bits

```
add $s7, $s0, $t6
sub $s6, $t5, $t2
addi $s1, $s2, -45
lw $s6, 100($s4)
sw $t6, -5($s3)
```

MIPS: Operaciones de Shift

- Son necesarias operaciones para ‘meter’ y ‘sacar’ carácteres de 8-bits en words de 32-bits
- Shifts mueve todos los bits del word a la izquierda o derecha

```
sll $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- Formato de instrucción (R format)

MIPS: Operaciones de Shift

- Son necesarias operaciones para ‘meter’ y ‘sacar’ carácteres de 8-bits en words de 32-bits
- Shifts mueve todos los bits del word a la izquierda o derecha

```
sll $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- Formato de instrucción (R format)

0		16	10	8	0x00

- Estos shifts (desplazamientos) son lógicos porque llenan los espacios con ceros
  - Notad que con un campo de 5-bit, shamt, es suficiente para desplazar el word de 32 bits (2^5-1) 31 posiciones

MIPS: Operaciones Logicas

- Hay ciertas operaciones lógicas bit a bit en MIPS

```
and $t0, $t1, $t2 #$t0 = $t1 & $t2

or  $t0, $t1, $t2 #$t0 = $t1 | $t2

nor $t0, $t1, $t2 #$t0 = not($t1 | $t2)
```

- Formato de instrucción (R format)

0	9	10	8	0	0x24

$t1 $t2 $t0

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

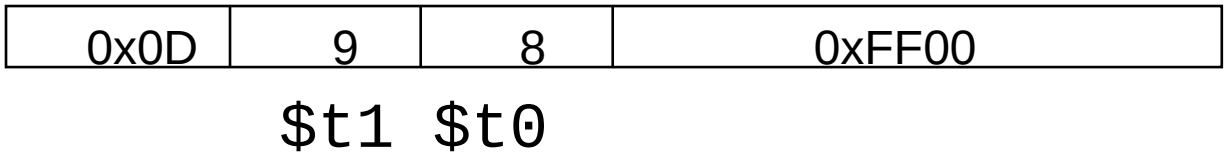
MIPS: Operaciones Logicas

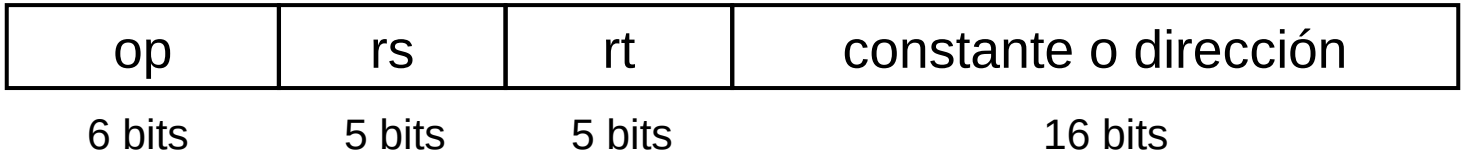
- Hay ciertas operaciones lógicas bit a bit en MIPS

```
andi $t0, $t1, 0xFF00   #$t0 = $t1 & ff00

ori  $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

- Formato de instrucción (I format)





MIPS: Instrucciones de control de flujo

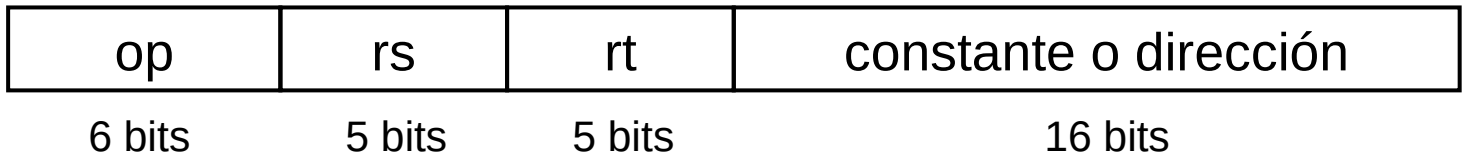
- MIPS, instrucciones de salto condicional:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

  - Ex: `if (i==j) h = i + j;`

```
      bne $s0, $s1, Lbl1
      add $s3, $s0, $s1
Lbl1: ...
```

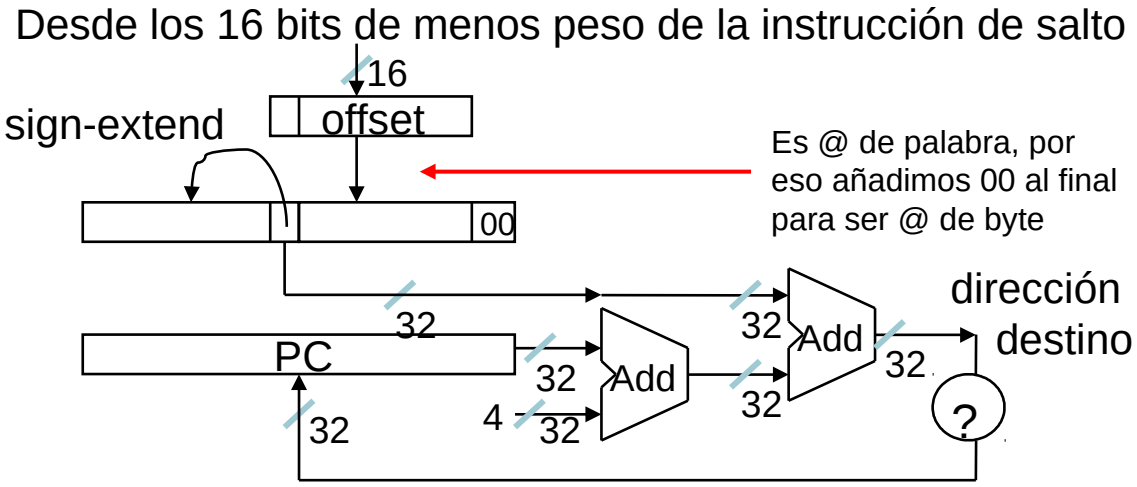
- Formato de instrucción (I format):



- Cómo se especifica la dirección destino de salto?

Especificando destinos de salto

- Usar un registro (como en lw y sw) añadido a los 16-bits de offset
  - Cuál registro? Instruction Address Register (el PC)
    - El uso del PC está implícito en la instrucción
    - PC queda actualizado (PC+4) durante el ciclo de fetch por lo tanto contiene la dirección de la siguiente instrucción
  - Esto limita la distancia de salto desde -2^{15} a $+2^{15}-1$ (words) instrucciones desde la (instrucción de después) de salto, pero muchos saltos son locales



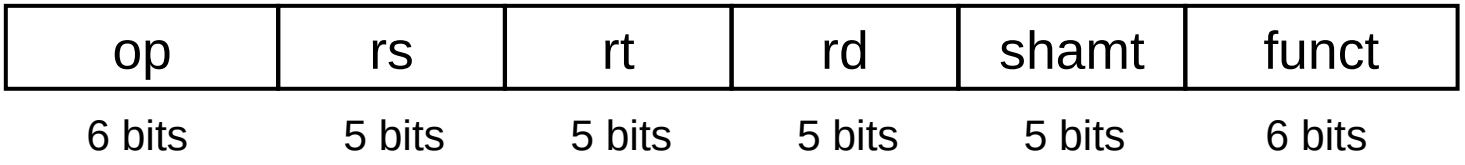
Soporte a las instrucciones de salto

- Tenemos `beq`, `bne`, pero qué hay sobre otros tipos de salto (ej., branch-if-less-than)? Para eso necesitamos otra instrucción, `slt`
- Instrucción Set on less than (pon 1 si es menor que):

```
slt $t0, $s0, $s1     # if $s0 < $s1 then
                          # $t0 = 1
                      # else
                          # $t0 = 0
```

- Instruction format (R format):

0	16	17	8		0x24



Soporte a las instrucciones de salto

- Tenemos `beq`, `bne`, pero qué hay sobre otros tipos de salto (ej., branch-if-less-than)? Para eso necesitamos otra instrucción, `slt`
- Instrucción Set on less than (pon 1 si es menor que):

```
slt $t0, $s0, $s1     # if $s0 < $s1 then
                          # $t0 = 1
                      # else
                          # $t0 = 0
```

- Instruction format (R format):

0	16	17	8		0x24

- Alternate versions of `slt`

```
slti $t0, $s0, 25  # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1 # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25 # if $s0 < 25 then $t0=1 ...
```

Más Instrucciones de Salto

- Podemos usar `slt`, `beq`, `bne`, y el valor fijo de 0 en el registro `$zero` para crear otras condiciones
  - Menor que `blt $s1, $s2, Label`

```
slt  $at, $s1, $s2  #$at set to 1 if
bne  $at, $zero, Label #$s1 < $s2
```

  - Menor o igual que `ble $s1, $s2, Label`
  - Mayor que `bgt $s1, $s2, Label`
  - Mayor o igual que `bge $s1, $s2, Label`

- Estos saltos están incluidos en el set de instrucciones como pseudo instrucciones – reconocidos (y expandidos) por el ensamblador
  - Es por lo que el ensamblador necesita un registro reservado (`$at`)

Truco para comprobar límites

- Tratar números con signo como si fueran sin signo nos da una forma con bajo coste de comprobar si 0 ≤ x < (índice fuera de límites en ‘array’)

```
sltu $t0, $s1, $t2 # $t0 = 0 if
                   # $s1 > $t2 (max)
                   # or $s1 < 0 (min)
beq $t0, $zero, IOOB # go to IOOB if
        # $t0 = 0
```

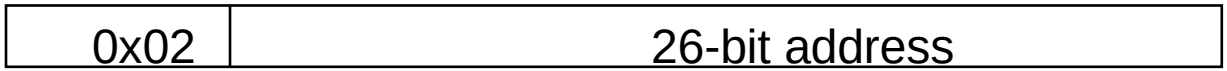
- La clave es que los números negativos en Ca2 parecen números grandes si los consideramos sin signo. Por lo tanto, una comparación sin signo de x < y también comprueba si x es negativo así como si x es menor que y.

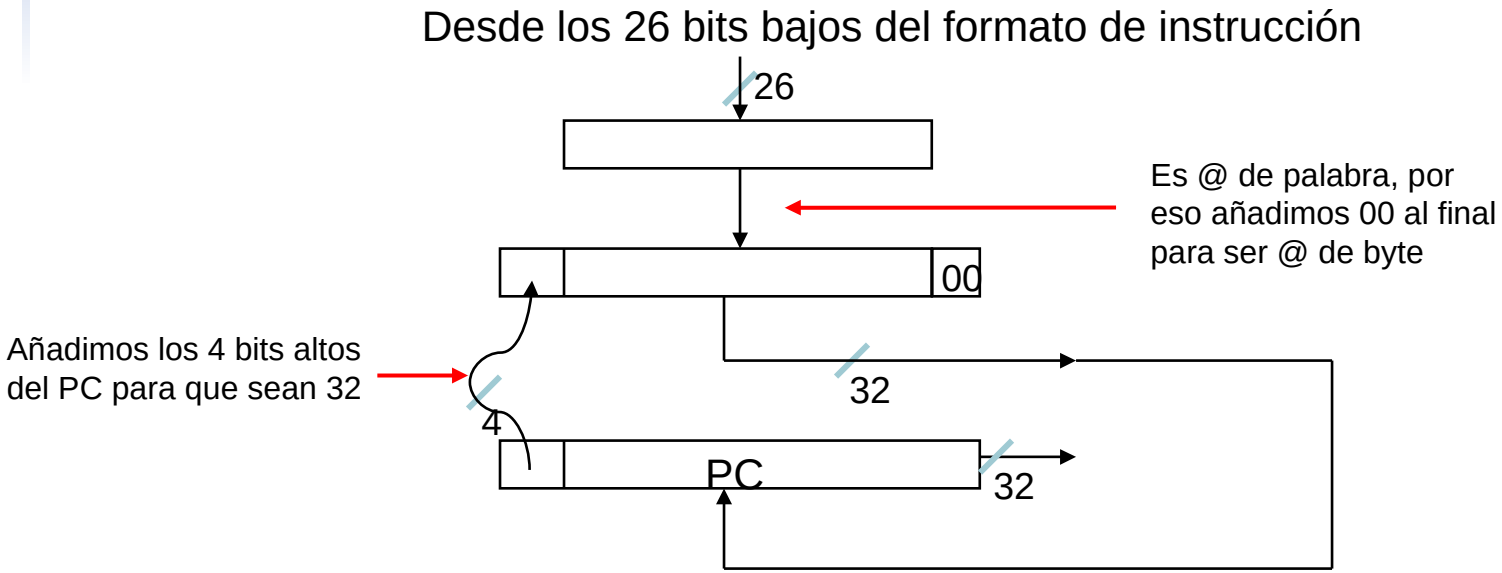
Otras instrucciones de control de flujo

- MIPS también tiene un salto incondicional o instrucción jump:

```
j  label    #go to label
```

- Formato de instrucción(J Format):





Ejemplo de dirección destino

- $s3 = i, $s5= k, $s6 = base de array

```
While (array[i]==k)
  i += 1;
```

```
Loop: sll  $t1, $s3, 2      # reg temp $t1 = 4 * I
      add  $t1, $t1, $s6    # $t1 = address array[i]
      lw   $t0, 0($t1)      # $t0 = array[i]
      bne  $t0, $s5, Exit   # ir a Exit
      addi $s3, $s3, 1      # i = i + 1
      j    Loop             # ir a Loop
Exit: …
```

Ejemplo de dirección destino

- Asumamos un bucle en @ 80000

  Loop:

¿Cómo lucen las instrucciones ensambladas y las direcciones?

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2^1		
80016	8	19	19	1		
80020	2	20000				
80024						

[1]Sólo sumamos 2 pues el PC está ya en la instr. addi $s3, $s3, 1

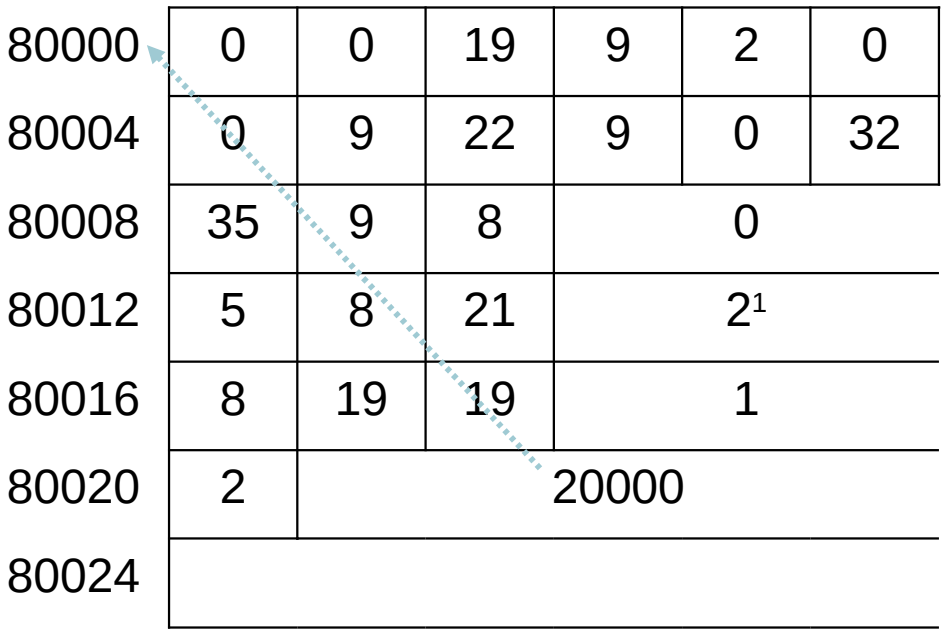
Ejemplo de dirección destino

- Asumamos un bucle en @ 80000

  Loop:

¿Cómo lucen las instrucciones ensambladas y las direcciones?

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: ...
```



Saltos más lejos

- ¿Qué pasa si el salto es más lejos de lo que se puede representar con 16 bits?
- El ensamblador viene al rescate – inserta un salto incondicional después del salto condicional e invierte la condición

```
beq  $s0, $s1, L1
```

16 bits

se convierte en

```
    bne  $s0, $s1, L2
    j  L1
L2:
```

26 bits

Síntasis del Ensamblador de MIPS

- Estructura:

```
.data
#declaración de las variables, por ejemplo
.align 0
str: .asciiz “Frase ejemplo%d\n”
.text
#Aquí iría el programa, por ejemplo
.globl main
main:
.
.
.
rutina1:
.
.
.
rutina2:
```

Síntasis del Ensamblador de MIPS

- Directivas:

.text<addr> lo siguiente que escribamos serán instrucciones (o .word) aquí se escribirá en el text segment. Si el argumento opcional addr está, se guardará a partir de esa dirección

.data<addr> lo siguiente que escribamos se guardará en el segmento de datos. Si el argumento opcional addr está, se guardará a partir de esa dirección

.globl sym declara que la etiqueta sym es global,y puede referenciarse desde otros archivos

.extern sym size declara que datos guardados en sym tienen una medida size y que es una etiqueta global. Permite al ensamblador poner datos a los que puede accederse con el registro $gp

Síntasis del Ensamblador de MIPS

- Directivas:

.align n alinea datos a múltiplos de 2^n

.ascii str almacena el string str sin acabar con 0

.asciiz str almacena el string str acabando con 0

.byte b1,…,bn almacena n bytes en memoria

.double d1,…,dn almacena n datos en punto flotante con doble precisión (64 bits)

.float f1,…,fn almacena n datos en punto flotante con precisión simple (32 bits)

.half h1,…, hn almacena n datos 'media palabra', halfword (16 bits)

.word w1,…,w2 almacena n palabras, word, de 32 bits

.set noat desactiva el 'warning' sobre las siguientes instrucciones que usan $at

.set at activa el 'warning' sobre las siguientes instrucciones que usan $at

.space n reserva n byte en el segmento actual (que debe ser el text segment en MIPS)

Compilación de un *while* en MIPS

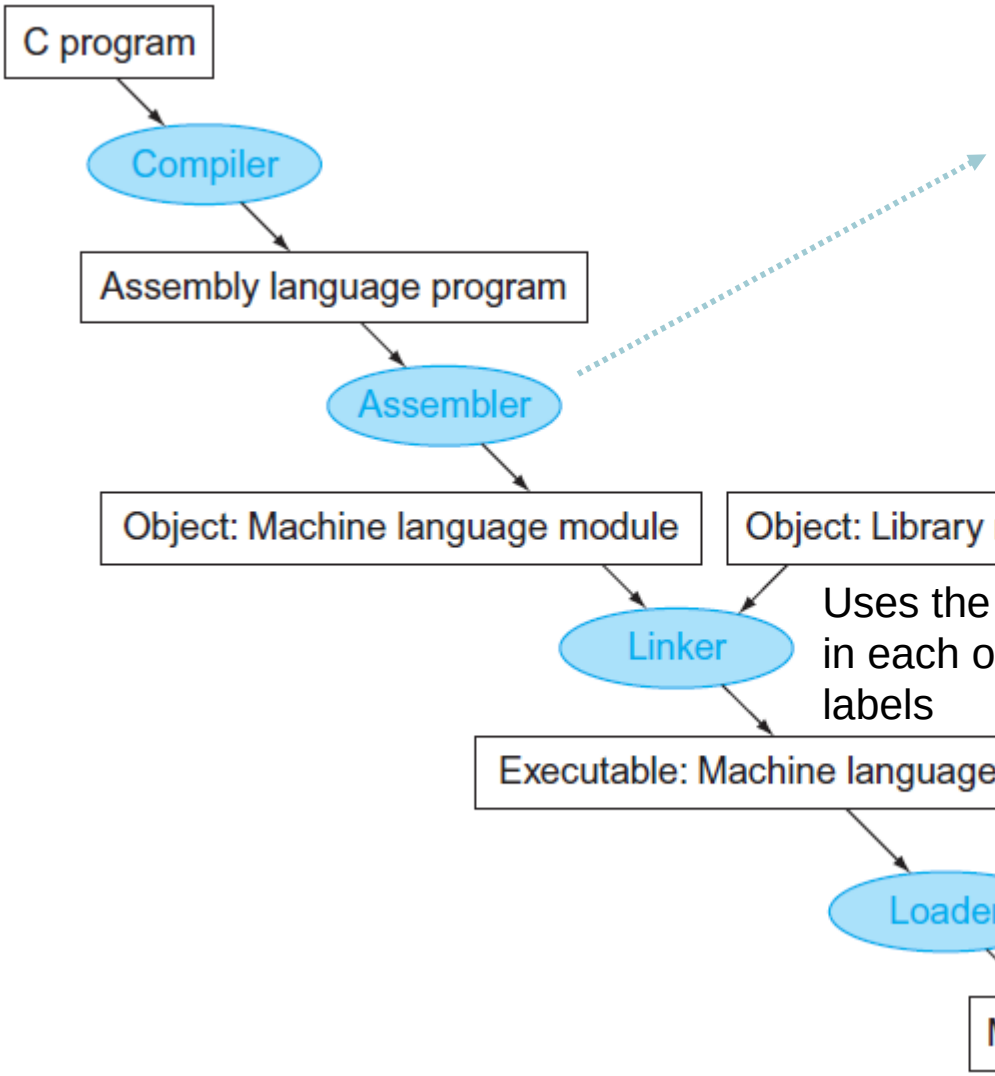
Ejemplo página 107 libro:

```
while(guardar[i]==k)
    i+=1;
```

Supongamos que *i* y *k* están en los registros $s3 y $s5 y la dirección base de la tabla guardar está en $s6

```
Loop:  sll $t1,$s3,2      #sll es despl lógico a izqda. $t1=4*$s3
       add $t1,$s3,$s6    #$t1=dirección de guardar[i]
       lw $t0,0($t1)      #$t0=guardar[i]
       bne $t0,$s5,Exit   #si guardar[i]≠k ir a Exit
       addi $s3,$s3,1     # i=i+1
       j Loop          #volvemos a Loop
Exit:
```

Jerarquía de traducción de código en C



Pseudo-instructions

```
move $t0,$t1
add $t0,$zero,$t1
```

Symbol table

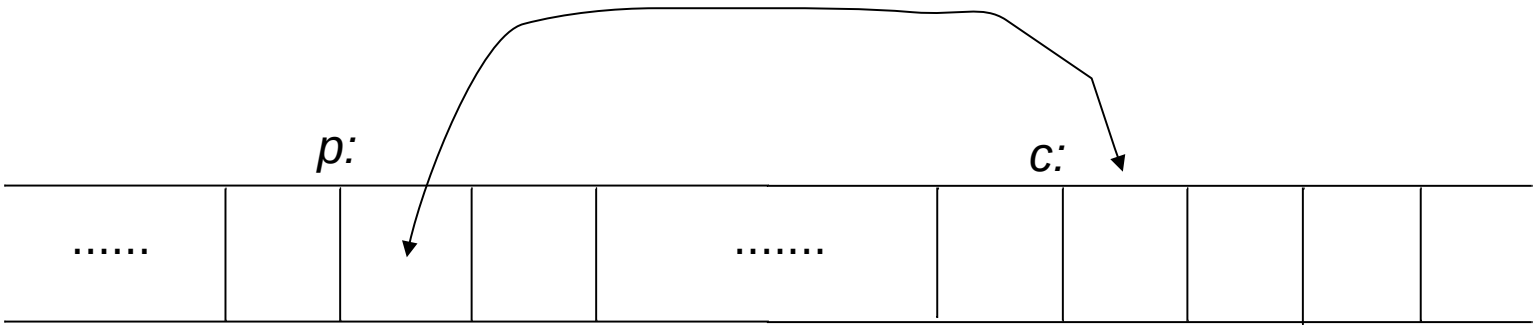
Uses the relocation information and symbol table in each object module to resolve all undefined labels

Punteros

- Tipo de datos más temidos, pero también de los más versátiles
- Su uso permite el paso de argumentos por referencia, construcción de estructuras dinámicas de datos, entre otros.
- Su mal uso produce errores graves, difíciles de detectar y de reproducir
- ¿Qué es un puntero? Variable que contiene una dirección de memoria.
  - Recordad que todo elemento del programa se sitúa en memoria principal.
  - Cada objeto tiene una posición de memoria

Punteros (operador &)

- Puntero es una variable cuyo contenido es la dirección de una variable (apunta a una variable)
- Sea ***p*** una variable de tipo puntero (int) y ***c*** una variable de tipo int:

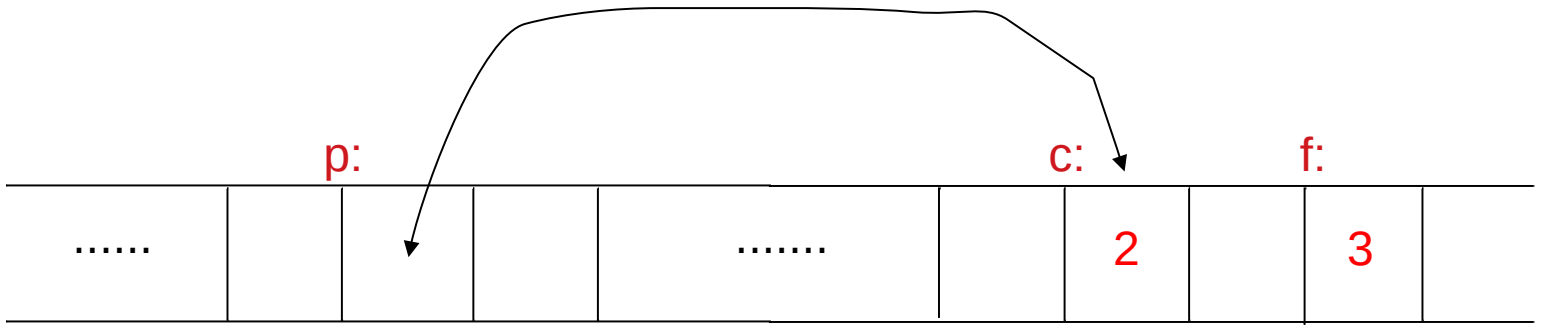


p = &c;

- El operador unario & entrega la dirección del objeto que le sucede

Punteros (operador *)

- El operador unario *****, permite el acceso al contenido de la dirección apuntada por la variable puntero que le sucede
- Ofrece un nombre alternativo para la variable apuntada
- Sean **c** y **f** dos variables enteras, sea **p** una variable puntero que "apunta a **c**"



**p = 2*;

*f =*p + 1*;

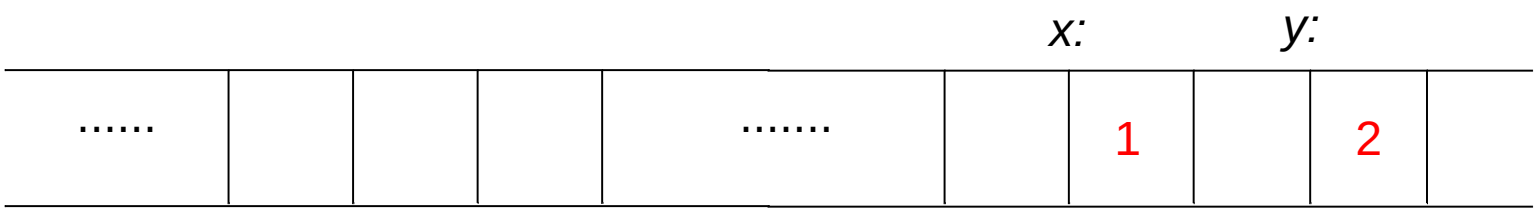
Es lo mismo que: c = 2;
f = c + 1;

Punteros

```
int x = 1, y = 2;
int *ip; /* ip es puntero a entero */
ip = &x; /* ip apunta al entero x  */
y = *ip; /* y es ahora 1           */
*ip = 0; /* x es ahora 0           */
```

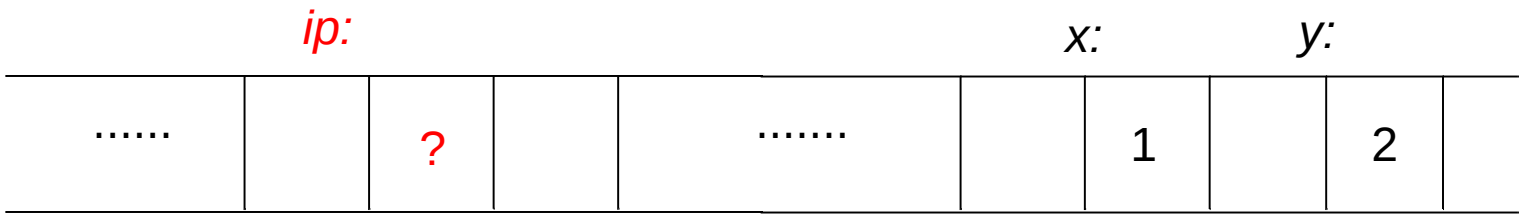
Punteros

```
int x = 1, y = 2;
int *ip; /* ip es puntero a entero */
ip = &x; /* ip apunta al entero x  */
y = *ip; /* y es ahora 1           */
*ip = 0; /* x es ahora 0           */
```



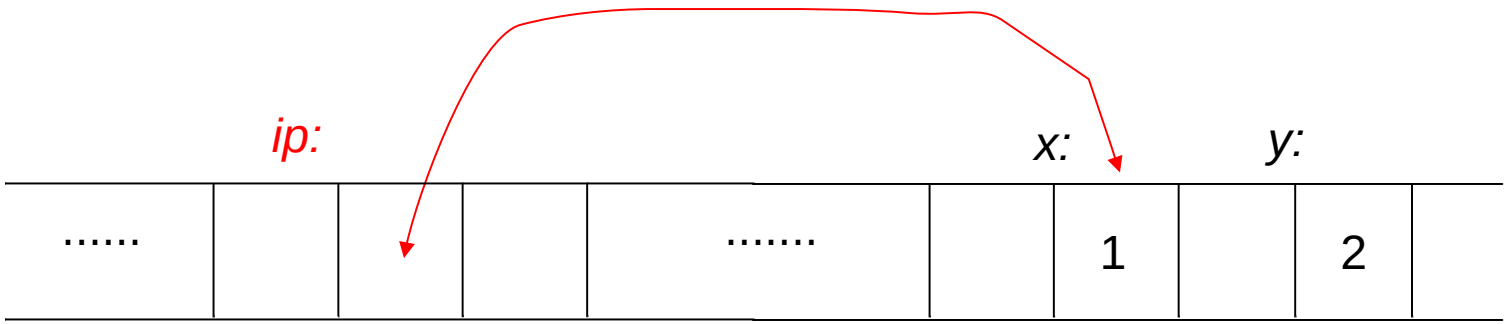
Punteros

```
int x = 1, y = 2;
int *ip; /* ip es puntero a entero */
ip = &x; /* ip apunta al entero x  */
y = *ip; /* y es ahora 1           */
*ip = 0; /* x es ahora 0           */
```



Punteros

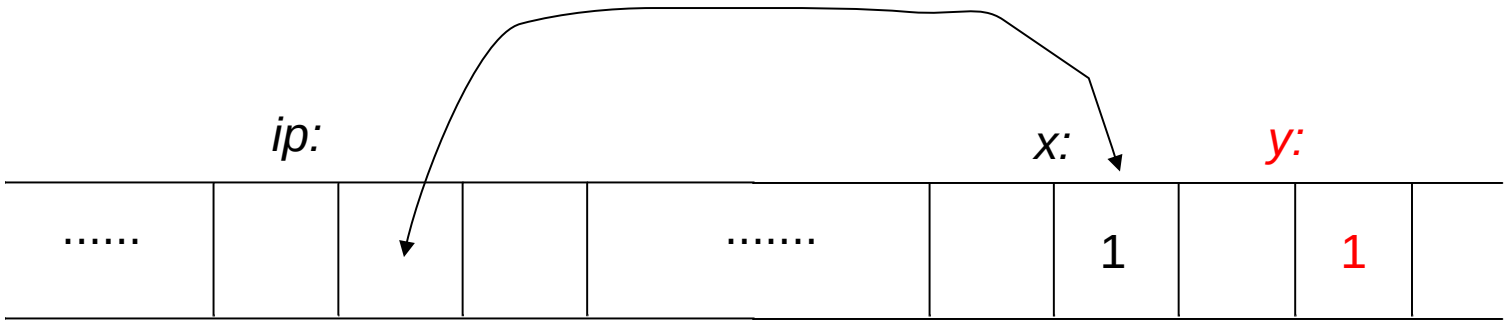
```
int x = 1, y = 2;
int *ip; /* ip es puntero a entero */
ip = &x; /* ip apunta al entero x  */
y = *ip; /* y es ahora 1           */
*ip = 0; /* x es ahora 0           */
```



Punteros

```
int x = 1, y = 2;
int *ip; /* ip es puntero a entero */
ip = &x; /* ip apunta al entero x  */
y = *ip; /* y es ahora 1           */
*ip = 0; /* x es ahora 0           */
```

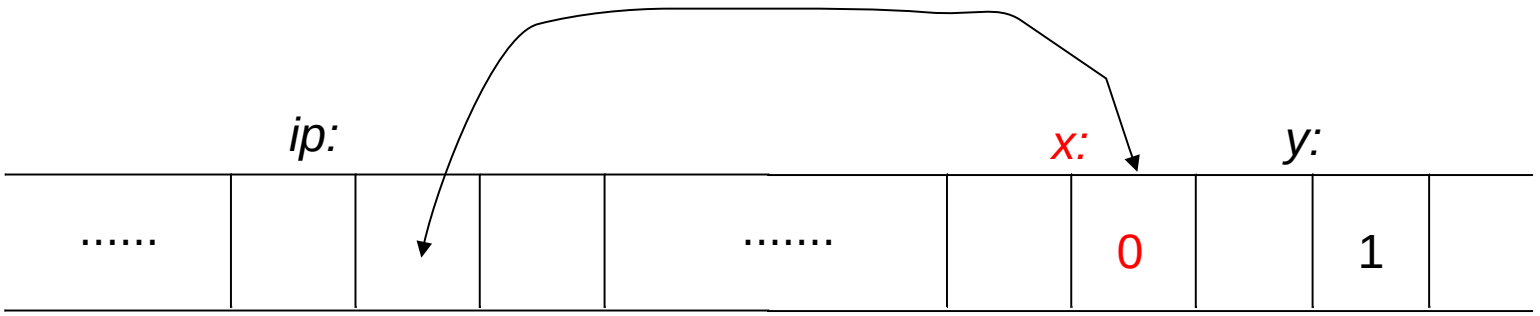
- El operador unario * es el operador de indirección



Punteros

```
int x = 1, y = 2;
int *ip; /* ip es puntero a entero */
ip = &x; /* ip apunta al entero x  */
y = *ip; /* y es ahora 1           */
*ip = 0; /* x es ahora 0           */
```

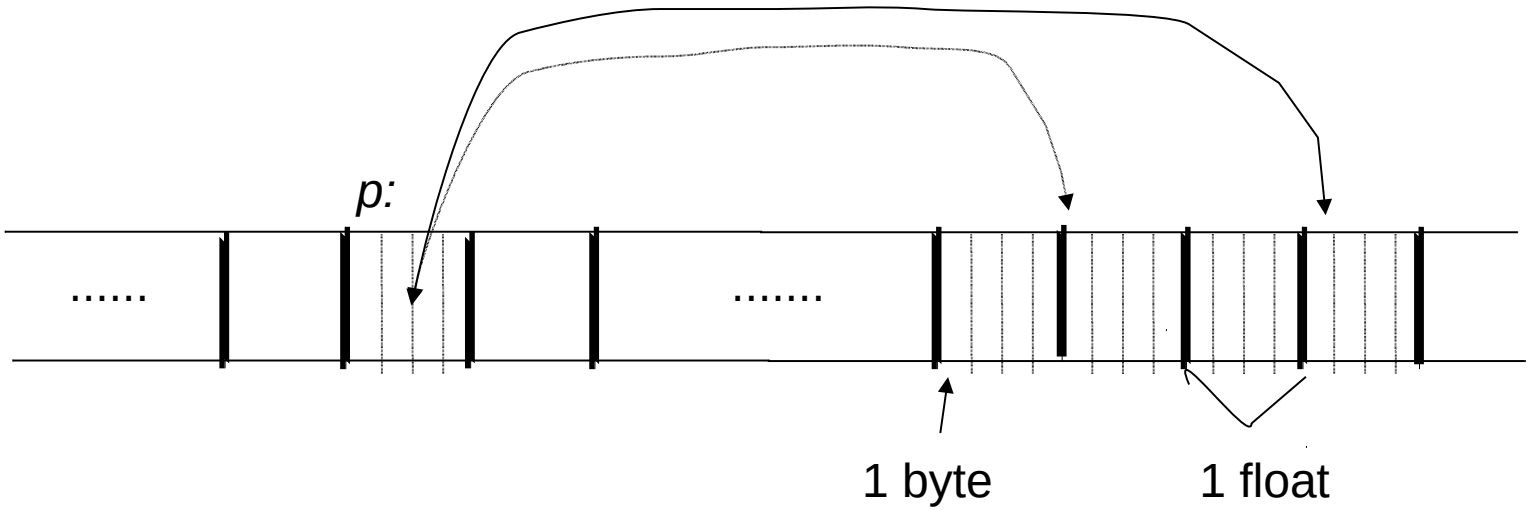
- El operador unario * es el operador de indirección



Aritmética de punteros

- El contenido de un puntero puede ser modificado realizando operaciones aritméticas enteras simples

```
float *p;    / * un float tiene 4 bytes  */
p = p  + 2;  /* p apunta ahora a una variable 2*4
bytes de distancia del la anterior */
```

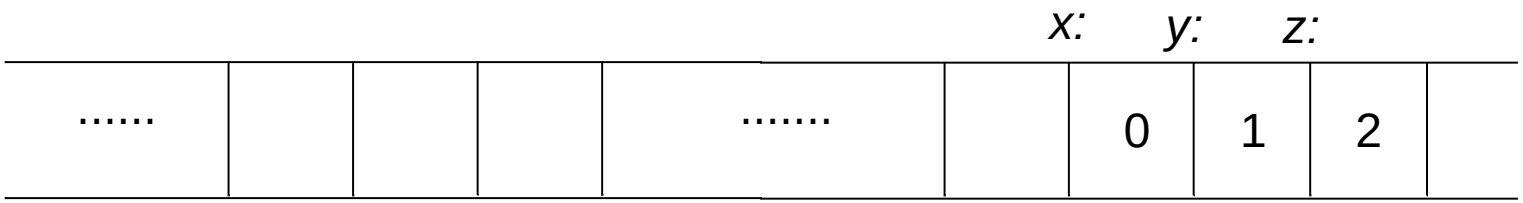


Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```

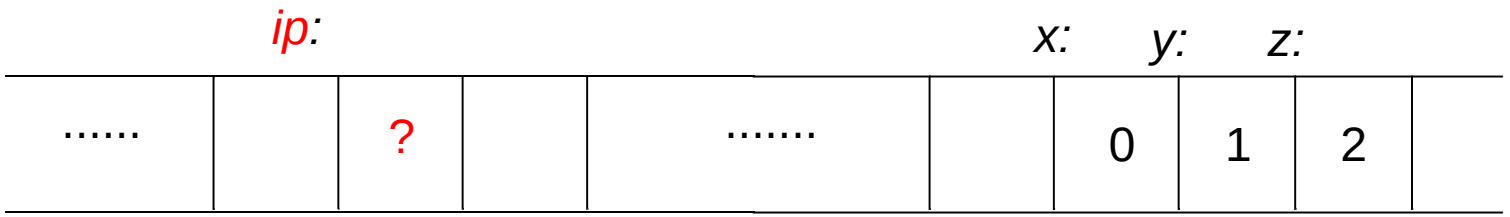
Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```



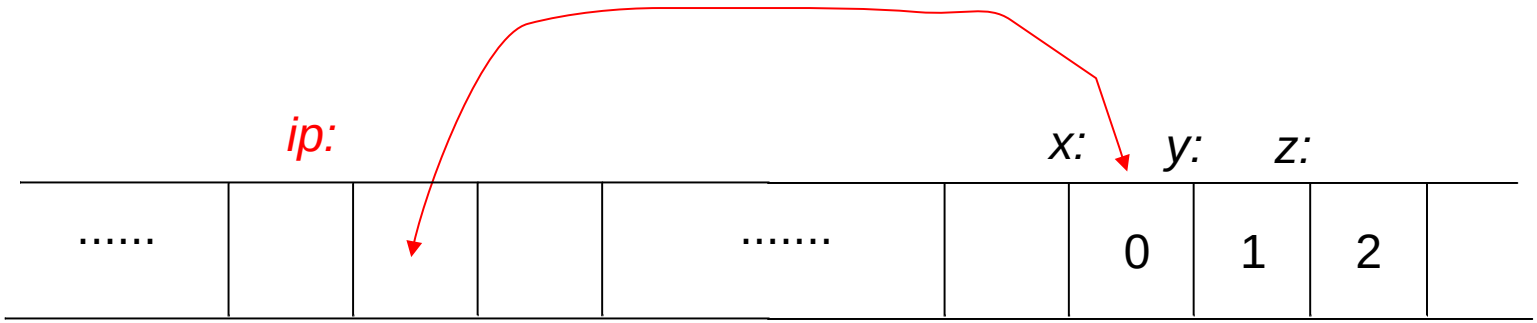
Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;      /* ip es puntero a entero */
ip = &x;     /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```



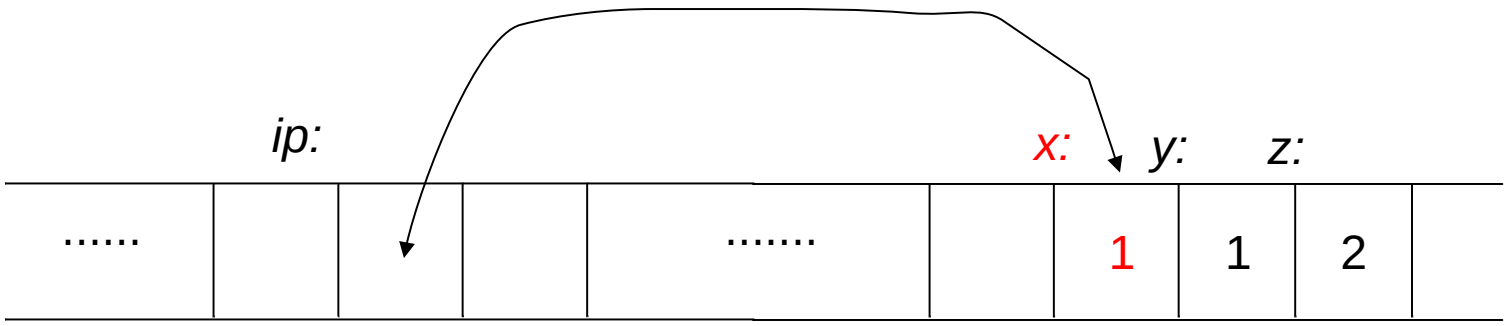
Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```



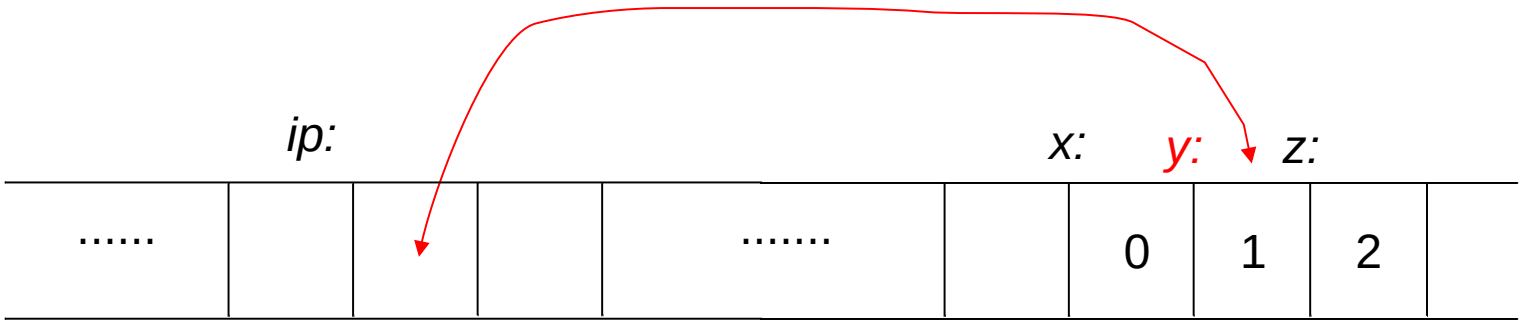
Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```



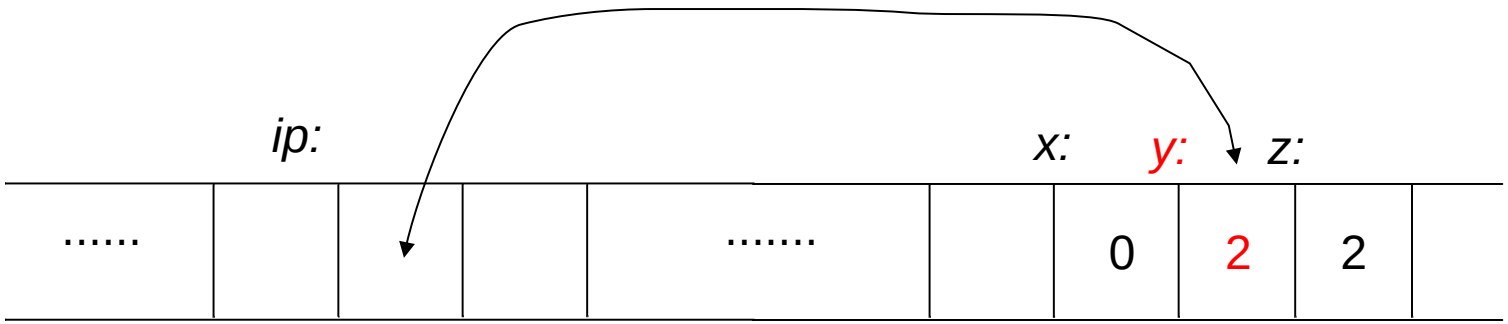
Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```



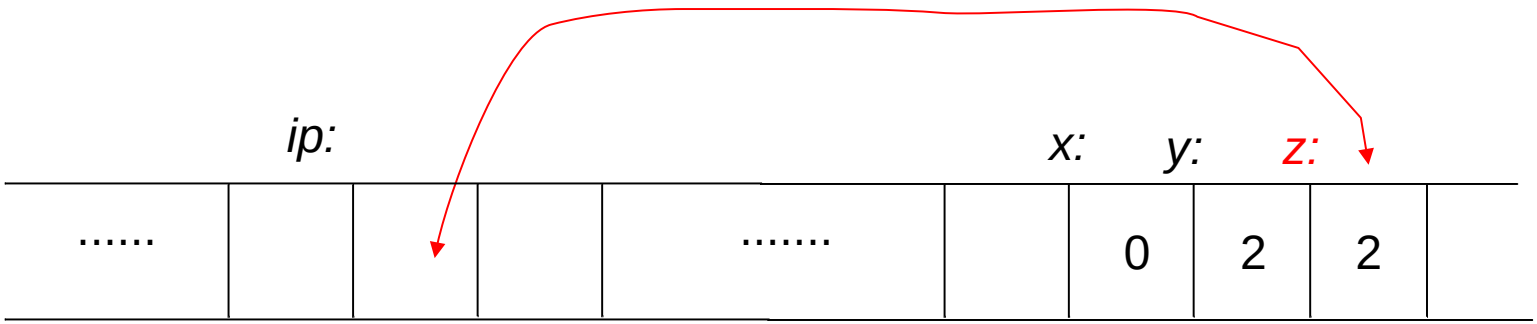
Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```



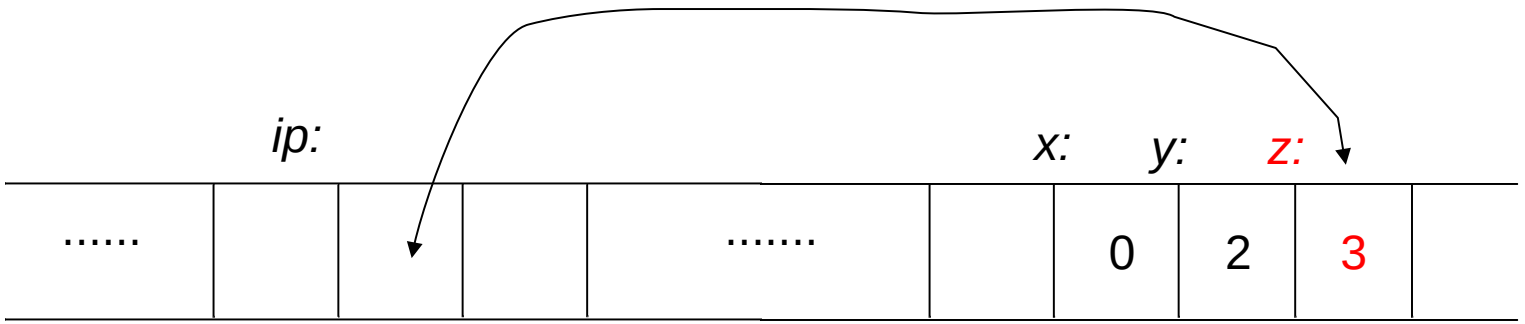
Aritmética de punteros

```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```

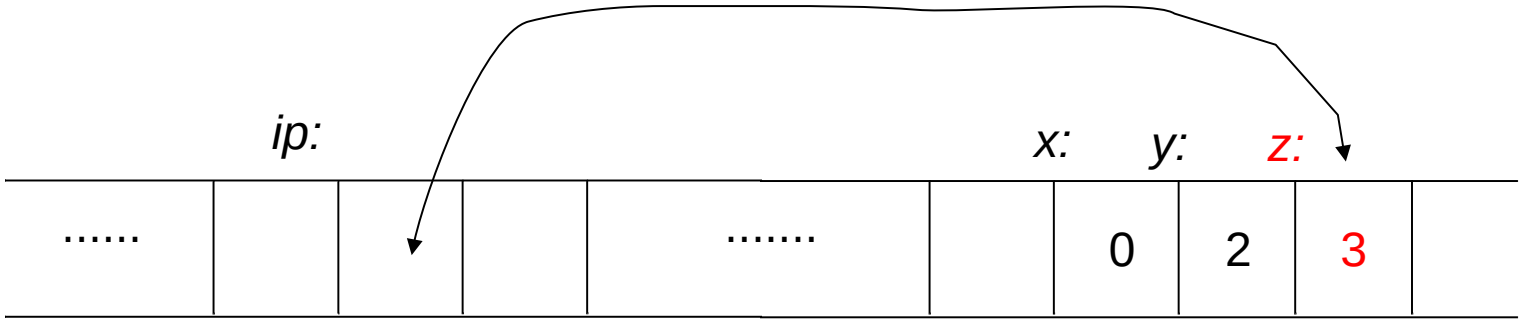


Aritmética de punteros

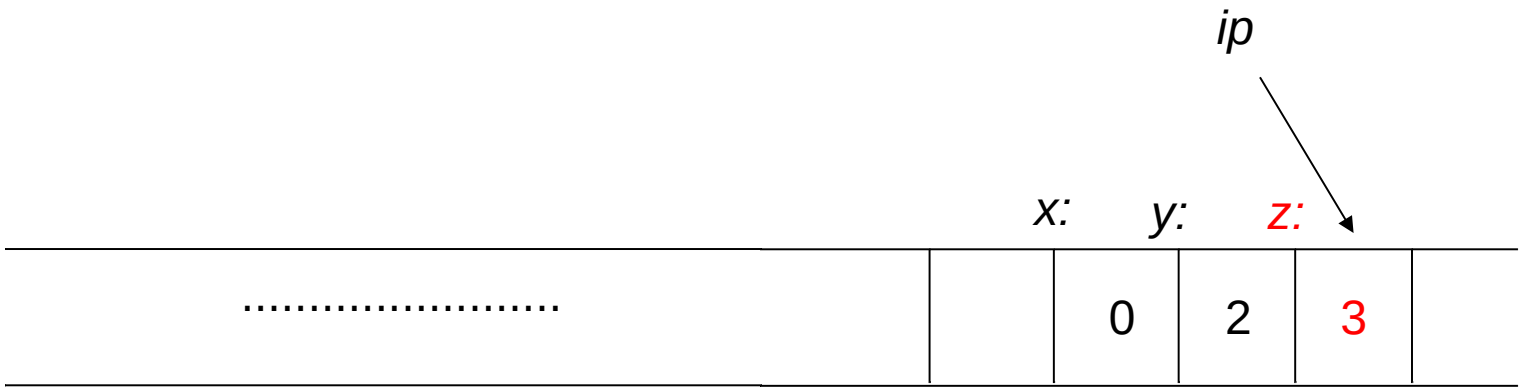
```
int x = 0, y = 1, z = 2;/* globales    */
int *ip;     /* ip es puntero a entero */
ip = &x;    /* ip apunta al entero x  */
*ip = *ip + 1; /* x es ahora 1         */
ip = ip + 1;   /* ip apunta ahora a y */
*ip = *ip + 1; /* y es ahora 2         */
ip = ip + 1;   /* ip apunta ahora a z */
*ip = *ip + 1; /* z es ahora 3         *
```



Punteros: otra imagen



Punteros: otra imagen



Arrays vs. Punteros

- Los Array implican indexación
  - Hay que multiplicar el índice por el tamaño del elemento
  - Y sumarlo a la dirección base del array
- Los Punteros corresponden directamente a direcciones de memoria
  - Pueden eliminar la complejidad de la indexación

Ejemplo: ‘Clear’ un array

clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
  array[i] = 0;
}
En MIPS pasamos en $a0 la @ de array y en $a1 el tamaño (en número de datos)
	clear2(int *array, int size) {
	  int *p;
	  for (p = &array[0]; p < &array[size];
	  p = p + 1)
	  *p = 0;
	}
	En MIPS pasamos en $a0 la @ de array y en $a1 el tamaño (en número de datos)

move $t0,$zero # i = 0
loop1: sll $t1,$t0,2 # $t1 = i * 4
add $t2,$a0,$t1 # $t2 = &array[i]
sw $zero, 0($t2) # array[i] = 0
addi $t0,$t0,1 # i = i + 1
slt $t3,$t0,$a1 # $t3 = (i < size)
bne $t3,$zero,loop1 # if (…) goto loop1
	move $t0,$a0 # p = & array[0]
	sll $t1,$a1,2 # $t1 = size * 4(en bytes)
	add $t2,$a0,$t1 # $t2 = &array[size]
	loop2: sw $zero,0($t0) # Memory[p] = 0
	addi $t0,$t0,4 # p = p + 4
	slt $t3,$t0,$t2 # $t3 = (p<&array[size])
	bne $t3,$zero,loop2 # if (…)
	# goto loop2

- La versión del Array requiere el shift dentro del bucle
  - Es parte del cálculo del índice para cada incremento de i
  - En la versión con punteros sólo incrementamos el puntero
- El compilador logra el mismo efecto que el uso manual de los punteros
  - Eliminación de los cálculos dentro del bucle
  - Multiplicación substituida por shift
  - Mejor para hacer el programa más claro y seguro