Tema 2

INSTRUCCIONES

Índice

1. Introducción

- Arquitectura del repertorio de instrucciones (ISA / Instruction Set Architecture):
  - Se trata de la porción del computador visible por el programador o el diseñador de compiladores
- Conjunto de instrucciones de un computador
- Diferentes computadores tienen diferente conjunto de instrucciones
  - Pero muchos aspectos son comunes
- Los primeros computadores tenían un conjunto de instrucciones muy sencillo
  - Implementaciones simplificadas
- Muchos computadores modernos también tienes conjuntos de instrucciones simples

Factores de diseño del juego de instrucciones

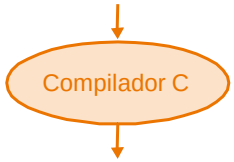
- El juego de instrucciones es la interface entre los programas y la ruta de datos
- Las instrucciones son el producto de la compilación
  - Si una instrucción del juego no la gastan los compiladores, es inútil
- La experiencia con la compilación hace pensar que
  - Los programas pueden ser muy complejas, pero la mayoría son muy sencillos
- Las instrucciones simples pueden ejecutarse rápidamente sobre una ruta de datos simple
  - Las instrucciones complejas necesitan rutas de datos complejas
  - Con instrucciones simples$\rightarrow$ bajan *CPI* y *T*.

Abstracción

- El ahondar en los niveles de profundidad nos revela más información
- La abstracción omite detalles innecesarios que nos ayudan a abordar la complejidad

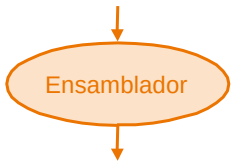
Programa en lenguaje de alto nivel (en C)

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



Programa En lenguaje Ensamblador (para MIPS)

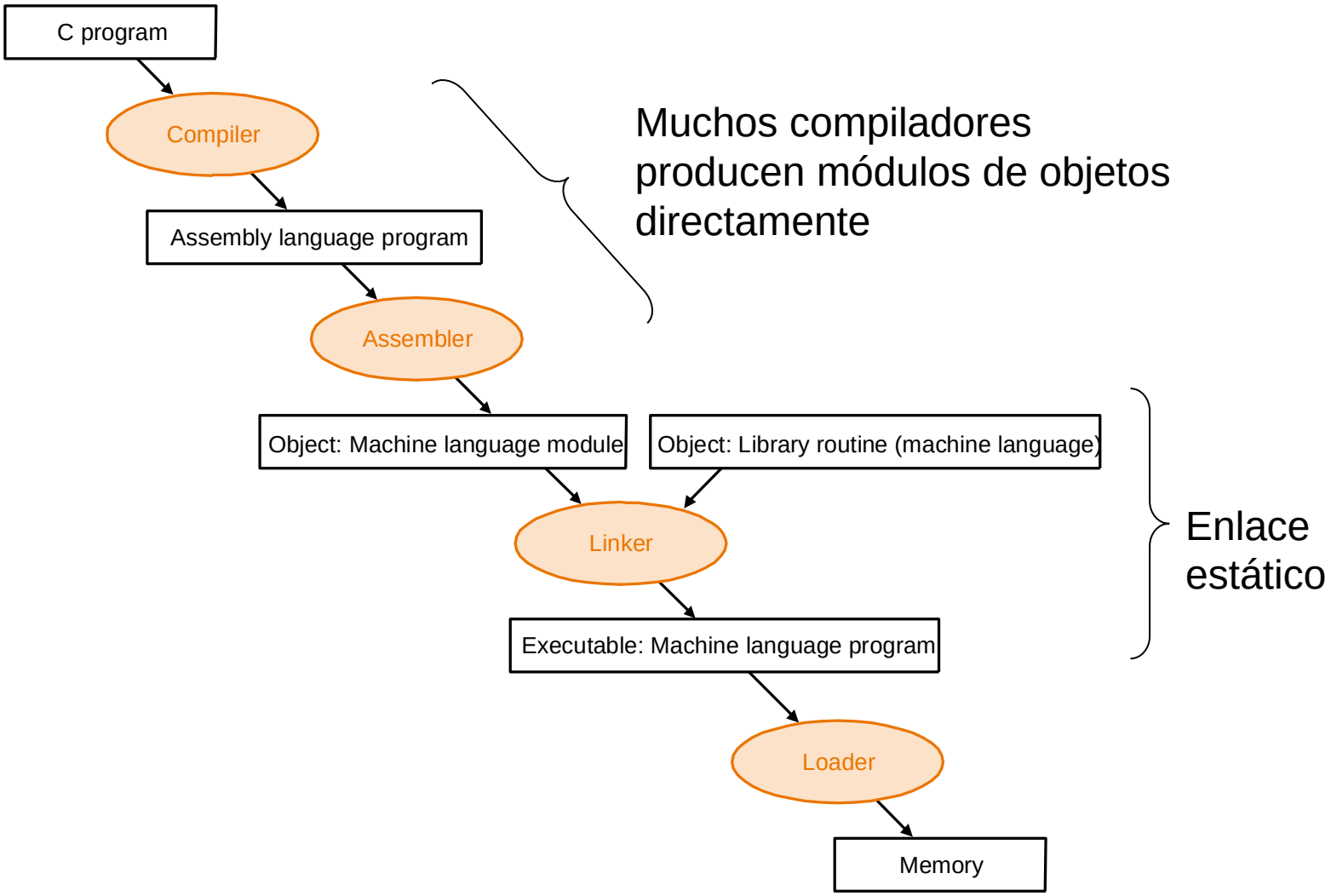
```
swap:
     muli $2, $5,4
     add  $2, $4,$2
     lw   $15, 0($2)
     lw   $16, 4($2)
     sw   $16, 0($2)
     sw   $15, 4($2)
     jr   $31
```



Programa En lenguaje Máquina (para MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Jerarquía de la traducción



Conjunto de instrucciones del MIPS

- Típico de muchas ISAs modernas
- Utilizado como ejemplo en este curso
- ISAs similares tienen una gran cuota de mercado entre los empotrados básicos
  - Aplicaciones en electrónica de consumo, equipos de red/almacenamiento, cámaras, impresoras,…

2. Operaciones Aritméticas y operandos

- Todas las instrucciones aritméticas tienen tres operandos (dos fuentes y un destino)
- El orden de los operandos es fijo (primero el destino)
- Ejemplo:

código C: A = B + C

código MIPS: add $s0, $s1, $s2

(la asociación de variables → realizada por el compilador)

- ***Principio de diseño 1***: La simplicidad favorece la regularidad
  - La regularidad hace que la implementación sea más sencilla
  - La simplicidad permite rendimientos más grandes a bajo coste

Operandos Registros

- Las instrucciones aritméticas usan registros como operandos
- MIPS tiene un banco de registros de 32 registros de 32 bits
  - Se usan para datos accedidos frecuentemente
  - Están numerados de 0 al 31
  - A un dato de 32 bits se le llama *palabra*
- Nombres de los registros en ensamblador
  - $t0, $t1, …, $t9 para almacenar valores temporales
  - $s0, $s1, …, $s7 para almacenar variables

- ***Principio de diseño 2***: Más pequeño es más rápido
  - Comparar con la memoria principal que tiene millones de posiciones

Ejemplo

- Código C

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

  - Las variables f, …, j en $s0, …, $s4

- Código MIPS compilado

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

Organización de la memoria

- La memoria principal se usa para datos compuestos
  - Arrays, estructuras, datos dinámicos…
- Para poder realizar una operación aritmética
  - Previamente: cargar los valores desde la memoria a registros
  - Después: almacenar el resultado desde registro a la memoria
- La memoria se ve como un gran array unidimensional en el que la dirección actúa como índice del array

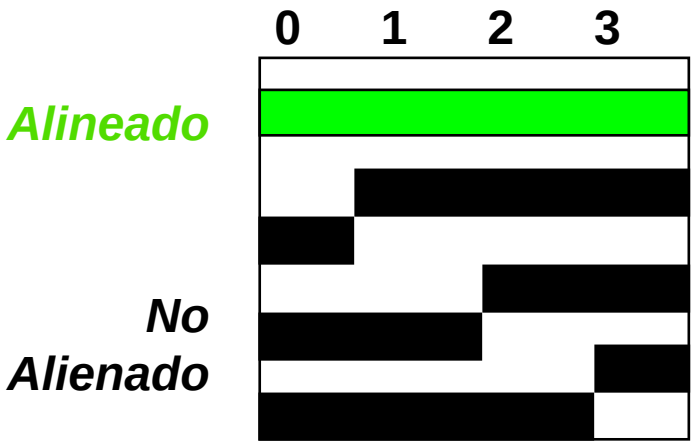
La memoria es direccionable por byte: cada dirección identifica un byte en la memoria

Organización de la memoria

- Una palabra es de 32 bits (4 bytes)

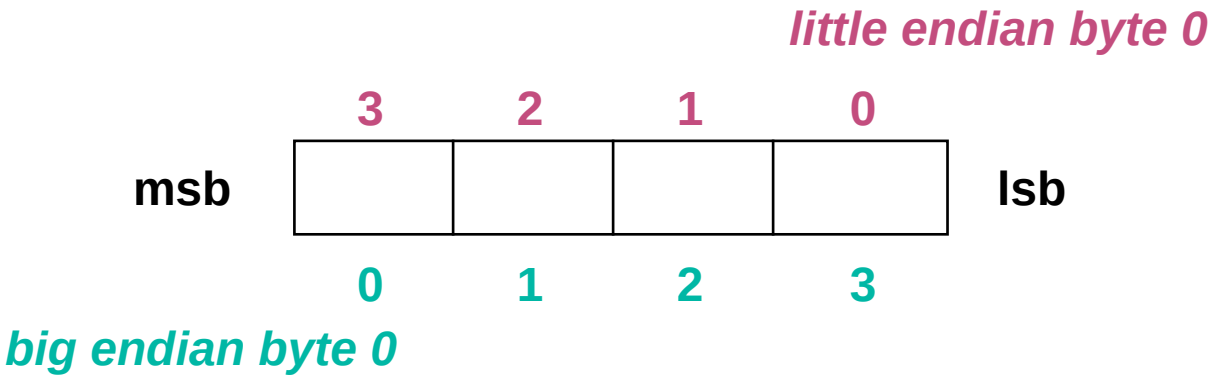
0	32 bits de datos
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

- Las palabras están alineadas en la memoria
  - Las direcciones deben ser múltiplos de 4



Organización de la memoria

- MIPS es Big Endian
  - **Big Endian**: dirección del byte más significativo = dirección de la palabra
  - **Little Endian**: dirección del byte menos significativo = dirección de la palabra



Ejemplo 1 de operandos en memoria

- Código C

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

  - g en $s1, h en $s2, dirección base de A en $s3

- Código MIPS compilado

  - El índice 8 requiere un desplazamiento de 32 (4 bytes por palabra)

```
lw $t0,  32($s3)      # $t0 ← A[8]
add $s1, $s2, $t0     #$s1 ← h + $t0
```

Desplazamiento

Registro base

Ejemplo 2 de operandos en memoria

- Código C

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

  - h en $s2, dirección base de A en $s3

- Código MIPS compilado

  - El índice 8 requiere un desplazamiento de 32

```
lw $t0, 32($s3)      # $t0 ← A[8]
add $t1, $s2, $t0    #$t0 ← h + $t0
sw $t0,  48($s3)     # A[12]← $t0
```

Registros versus Memoria

- Los accesos a registros son más rápidos que a la memoria
- Operar con datos en memoria requiere cargas (*load*) y almacenamientos (*store*)
  - Se necesitan ejecutar más instrucciones
- Tanto como sea posible, el compilador debe utilizar registros para ubicar variables
  - Sólo llevar a memoria las variables que se usan menos frecuentemente
  - La optimización de los registros es importante

Operandos inmediatos

- Datos constantes que se especifican en la propia instrucción

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

- No existe la resta inmediata
  - Se ha de utilizar una constante negativa

```
addi $s3, $s1, -1     # $s3 ← $s1 - 1
```

- ***Principio de diseño 3.*** Hacer rápido el caso común
  - Las constantes pequeñas son muy comunes
  - Los operandos inmediatos evitan las instrucciones de carga

La constante Cero

- El registro 0 ($zero) del MIPS es la constante cero.
  - No se puede sobrescribir
- Útil para operaciones comunes
  - Ejemplo: movimiento entre registros

```
add $t2, $s1, $zero      # $t2 ← $s1
```

Valores binarios sin signo

- Dado un número de n bits

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Rango: 0 a 2^n - 1
- Ejemplo
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Usando 32 bits
  - 0 a 4,294,967,295

Enteros en complemento a 2

- Dado un número de n bits

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Rango: -2^{n-1} a $+2^{n-1} - 1$
- Ejemplo
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Usando 32 bits
  - –2,147,483,648 a +2,147,483,647

Enteros en complemento a 2

- El bit 31 es el bit de signo
  - 1 para números negativos
  - 0 para números no negativos
- El valor $-(-2^{n-1})$ no se puede representar
- Los números no negativos tienen la misma representación sin signo y en complemento a 2
- Algunos ejemplos
  - 0: 0000 0000…0000
  - -1: 1111 1111…1111
  - El más negativo: 1000 0000…0000
  - El más positivo: 0111 1111…1111

3. Representación de las instrucciones

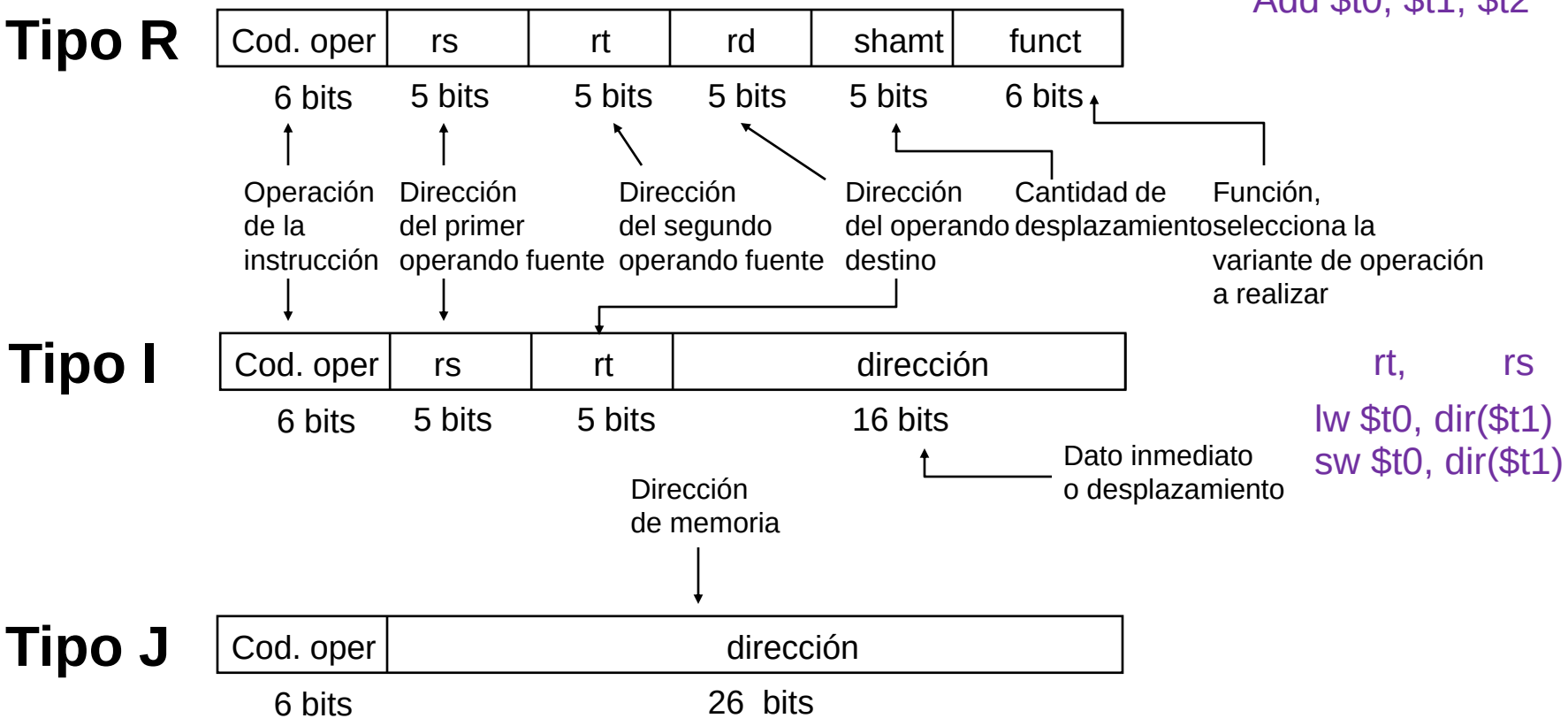
- Las instrucciones se codifican en binario
  - Se llama código máquina
- Instrucciones MIPS
  - Codificadas en palabras de 32 bits
  - Un pequeño número de formatos que codifican el código de operación, números de registro, etc.
  - Regularidad
- Números de registros
  - $t0 – $t7 son los registros 8 al 15
  - $t8 – $t9 son los registros 24 al 25
  - $s0 – $s7 son los registros 16 al 23

Formatos de instrucción

- Toda instrucción MIPS necesita 32 bits (igual que una palabra de datos y un registro)

rd, rs, rt

Add $t0, $t1, $t2

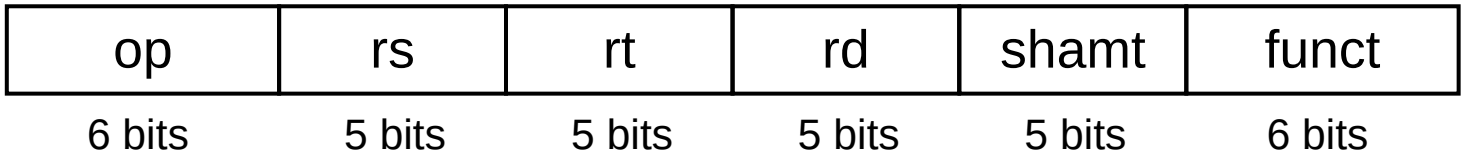


rt, rs

lw $t0, dir($t1)

sw $t0, dir($t1)

Formato de instrucción **R**



- Campos de la instrucción
  - op: código de operación
  - rs: número del primer registro fuente
  - rt: número del segundo registro fuente
  - rd: número del registro destino
  - shamt: cantidad de bits a desplazar (00000 por el momento)
  - funct: código de función (extensión para el código de función)

Ejemplo de Formato tipo **R**

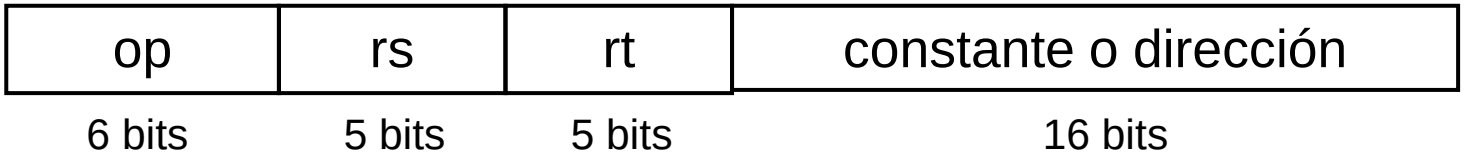
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

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

$$00000010001100100100000000100000_2 = 02324020_{16}$$

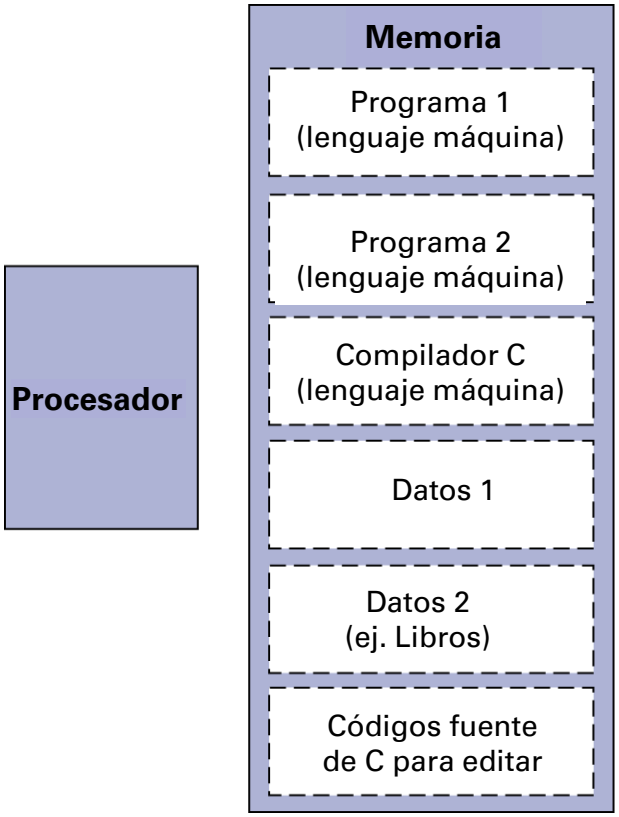
Formato de instrucción I



- Instrucciones load/store y aritméticas inmediatas
  - rt: número del registro fuente o destino
  - Constante: : -2^{15} a $+2^{15} - 1$ (constante entera en complemento a 2)
  - Dirección: desplazamiento añadido a la dirección base en rs

- ***Principio de diseño 4.*** Los buenos diseños requieren de buenos compromisos
  - Los diferentes formatos complican la decodificación, pero permiten instrucciones de 32 bits de manera uniforme
  - Hay que mantener los formatos lo más similares posible

Computadores de programa almacenado



- Las instrucciones como los datos se representan en binario
- Las instrucciones y los datos se almacenan en memoria
- Los programas pueden operar sobre programas
  - Ej. Compiladores, enlazadores…
- La compatibilidad binaria permite que programas compilados trabajen en diferentes máquinas
  - Repertorios de instrucciones estandarizados

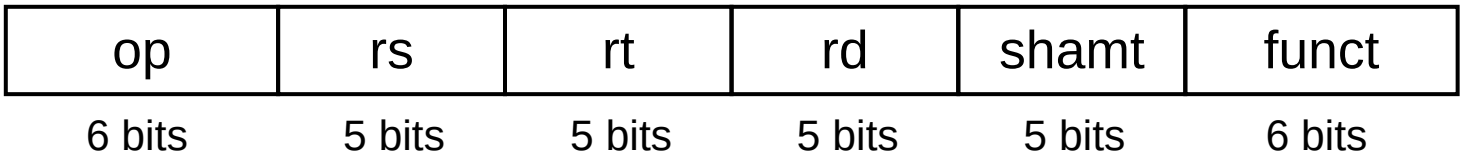
4. Operaciones lógicas y desplazamiento

- Instrucciones para manipulación de bits

Operación	C	Java	MIPS
Desplazamiento a izquierda	<<	<<	`sll`
Desplazamiento a derecha	>>	>>>	`srl`
AND bit a bit	&	&	`and, andi`
OR bit a bit	|	|	`or, ori`
NOT bit a bit	~	~	`nor`

- Útiles para extraer e insertar grupos de bits en una palabra

Operaciones de desplazamiento

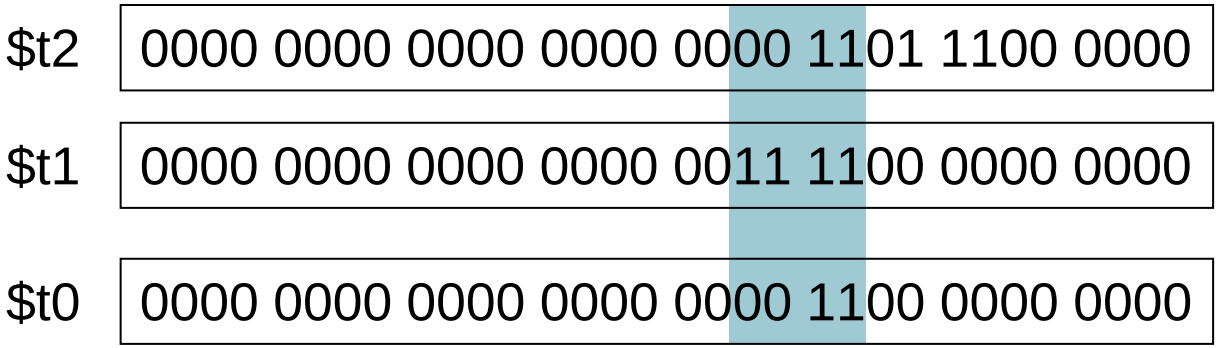


- Shamt: número de bits a desplazar
- Desplazamiento lógico a la izquierda
  - Desplazar a la izquierda y rellenar con bits a 0
  - sll por *i* bits multiplica por 2^i
- Desplazamiento lógico a la derecha
  - Desplaza a la derecha y rellena con bits a 0
  - srl por *i* bits divide por 2^i (solo sin signo)

Operaciones lógicas

- AND: útil en máscaras de bits en una palabra
  - Selecciona bits, el resto a cero

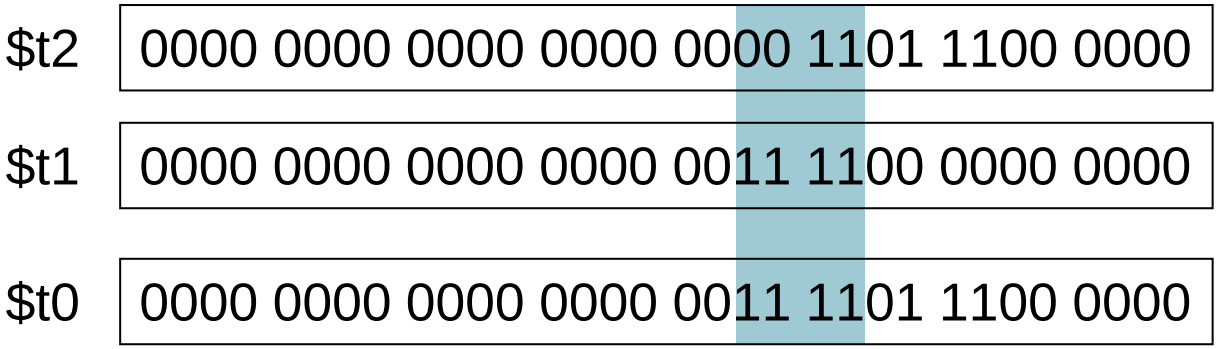
```
and $t0, $t1, $t2
```



Operaciones lógicas

- OR: útil para incluir bits en una palabra
  - Algunos bits a 1, el resto no cambia

```
or $t0, $t1, $t2
```



Operaciones lógicas

- NOT: útil para invertir bits en una palabra
  - MIPS tiene la instrucción NOR

    (a NOR b == NOT (a OR b))

    nor $t0, $t1, $zero

    Registro 0: siempre vale 0

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

5. Operaciones condicionales

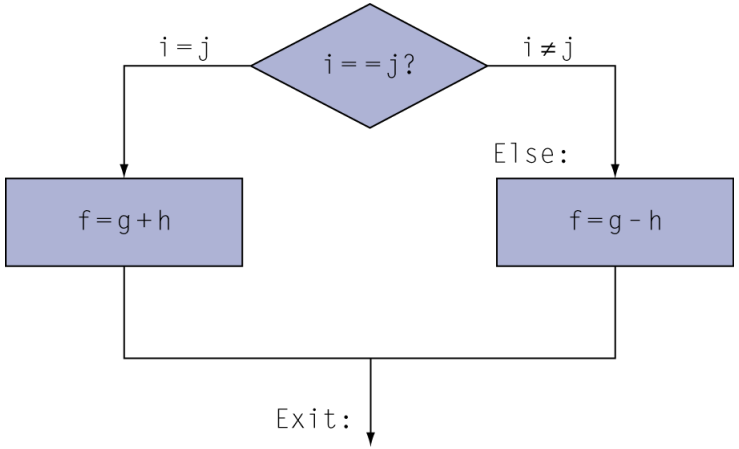
- Saltar a una instrucción etiquetada si la condición es cierta
  - En otro caso continuar secuencialmente
- `beq rs, rt, L1`
  - Si (rs = rt) saltar a la instrucción con etiqueta L1
- `bne rs, rt, L1`
  - Si (rs ≠ rt) saltar a la instrucción con etiqueta L1
- `j L1`
  - Salto incondicional a la instrucción con etiqueta L1

Ejemplo de sentencia if compilada

- Código C

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

  - f, g, … en $s0, $s1,…



- Código MIPS compilado

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   …
```

Direcciones calculadas por el ensambalador

Ejemplo de sentencia loop compilada

- Código C

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

  - i en $s3, k en $s5, dirección de *save* en $s6

- Código MIPS compilado

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

Otras operaciones condicionales

- Poner el resultado a 1 si una condición es cierta
  - En otro caso poner a 0
- `slt rd, rs, rt hile`
  - si (rs < rt) rd = 1; en otro caso rd = 0
- `slti rt, rs, constante`
  - si (rs < constante) rt = 1; en otro caso rt = 0
- Se usa en combinación con beq, bne

```
slt $t0, $s1, $s2        # si ($s1 < $s2)
bne $t0, $zero, L        #  saltar a L
```

Diseño de las instrucciones de salto

- ¿Por qué no hay *blt*, *bge*, etc?
- El hardware para las operaciones <, ≥,… más lento que =, ≠
  - Combinarlas con saltos implica más trabajo por instrucción, lo que requiere un ciclo de reloj más lento
  - Se penaliza a todas las instrucciones
- La mayor parte de las comparaciones son test de igualdad o desigualdad → beq y bne son el caso común
- Se trata de un buen compromiso de diseño

6. Datos de tipo carácter

- Juego de caracteres codificados en byte
  - ASCII: 128 caracteres
    - 95 gráficos (imprimibles), 33 de control
  - Latin-1: 256 caracteres
    - ASCII, +96 caracteres gráficos más
- Unicode: juego de caracteres de 32 bits
  - Usados en Java, caracteres amplios de C++,…
  - La mayoría de los alfabetos del mundo más símbolos
  - UTF-8, UTF16: codificaciones de longitud variable

Operaciones de byte y media palabra

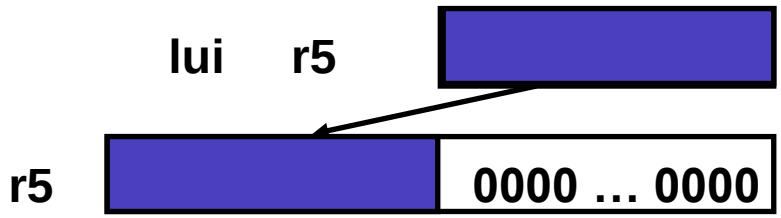
- Pueden utilizar operaciones bit a bit
- El MIPS dispone de load/store de byte y media palabra
  - Un caso común son las operaciones sobre cadenas de caracteres
- `lb rt, offset(rs)` `lh rt, offset(rs)`
  - Signo extendido a 32 bits en rt
- `lbu rt, offset(rs)` `lhu rt, offset(rs)`
  - Extensión con ceros a 32 bits en rt
- `sb rt, offset(rs)` `sh rt, offset(rs)`
  - Se almacena solo el byte/media palabra más a la derecha

7. Direccionamiento del MIPS

- La mayoría de las constantes son pequeñas
  - Es suficiente con un inmediato de 16 bits
- Para las constantes de 32 bits ocasionales

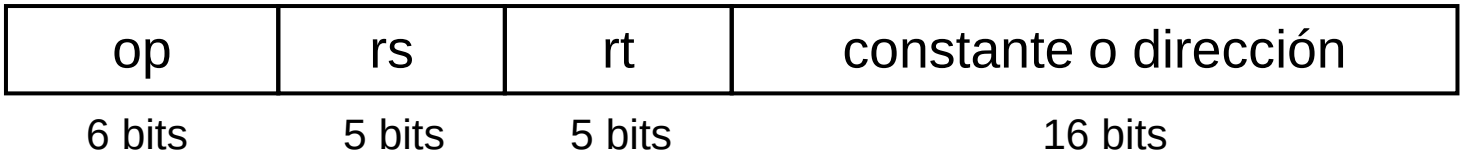
```
Lui rt, constante
```

  - Copia la constante de 16 bits en los 16 bits de la izquierda de rt
  - Pone a cero los 16 bits de la derecha de rt



Direccionamiento en salto condicional

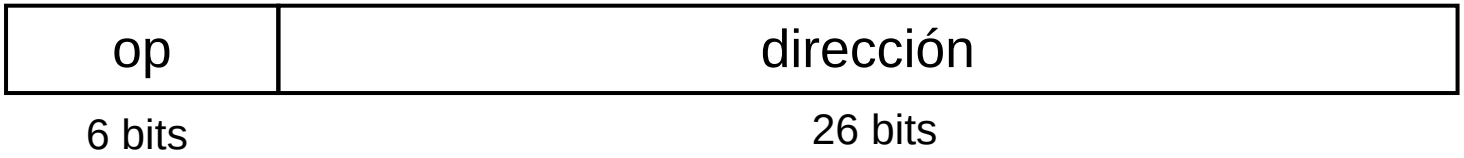
- Las instrucciones de salto especifican:
  - Código de operación, 2 registros, la dirección destino
- La mayoría de los destinos en los saltos están cerca
  - Hacia adelante o hacia atrás



- Direccionamiento relativo al PC
  - Dirección destino = PC + desplazamiento x 4
  - El PC ya se encuentra incrementado en 4

Direccionamiento en salto incondicional

- El destino de *jump* (j y jal) puede estar en cualquier lugar del segmento de texto
  - Código de operación, 2 registros, la dirección destino
- La mayoría de los destinos en los saltos están cerca
  - Se codifica la dirección completa en la instrucción

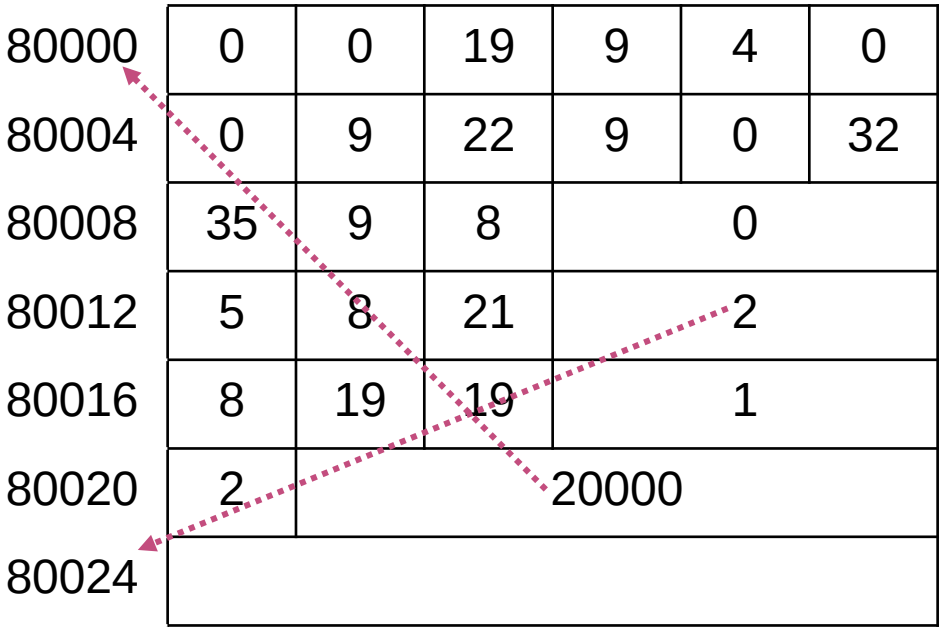


- Direccionamiento de salto (pseudo)directo
  - Dirección destino = $PC_{31...28}$ & dirección x 4

Ejemplo de dirección destino

- Supongamos el código ejemplo anterior del loop
  - Supongamos que *loop* se encuentra en la ubicación 8000

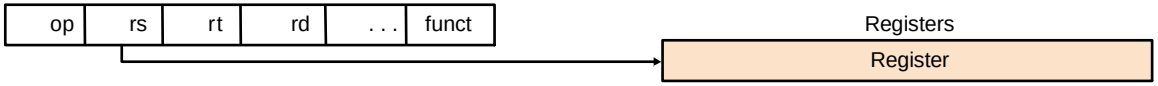
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



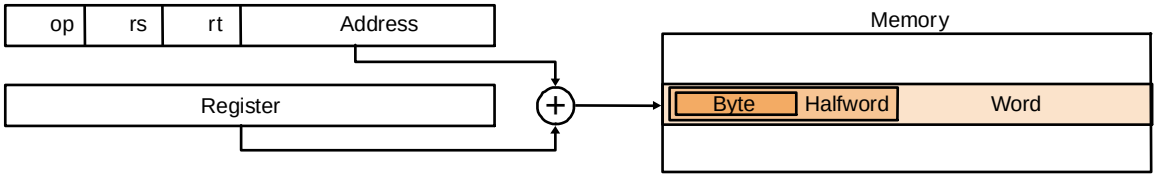
Modos de direccionamiento

1. Direccionamiento inmediato

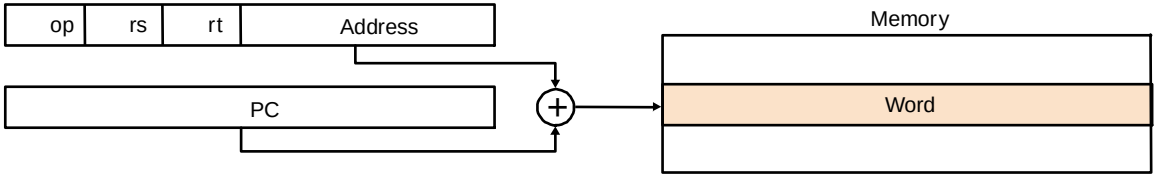
2. Direccionamiento directo a registro



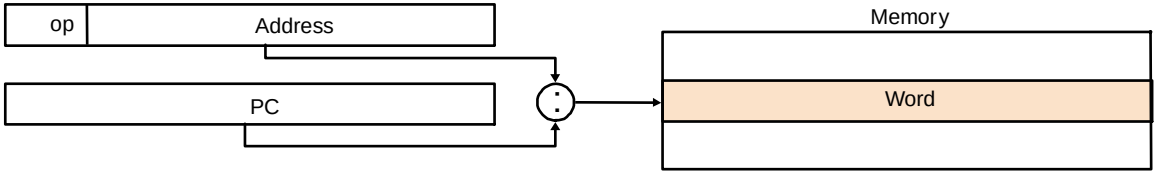
3. Direccionamiento Base



4. Direccionamiento relativo al PC



5. Direccionamiento pseudodirecto



8. Arquitecturas alternativas

- Diseño alternativo:
  - Proveer de operaciones más potentes
  - El objetivo es reducir el número de instrucciones ejecutadas
  - El peligro es un ciclo de reloj más grande y/o un CPI más grande
- Algunas veces se referencia como "RISC vs. CISC"
  - Virtualmente todos los nuevos conjuntos de instrucción desde 1982 han sido RISC
  - *VAX: minimiza tamaño del código haciendo el lenguaje ensamblador fácil*

    instrucciones desde 1 a 54 bytes de longitud!
- Comentaremos brevemente el 80x86

El Intel x86

- Evolución con compatibilidad con versiones anteriores
  - 8080 (1978): microprocesador de 8bits
    - Acumulador, más 3 pares de registros índice
  - 8086 (1978): extension de 16 bit del 8080 (8 bits)
    - Complex instruction set (CISC)
  - 8087 (1980): coprocesador de coma flotante
    - Se añaden instrucciones CF y registro de pila
  - 80286 (1982): incrementa el espacio de direcciones a 24 bits, MMU (Unidad de gestión de memoria)
    - Mapeo y protección de memoria segmentada
  - 80386 (1985): extensión a 32-bit (ahora IA-32)
    - Operaciones y modos de direccionamiento adicionales
    - Mapeo de memoria paginada y con segmentos paginados

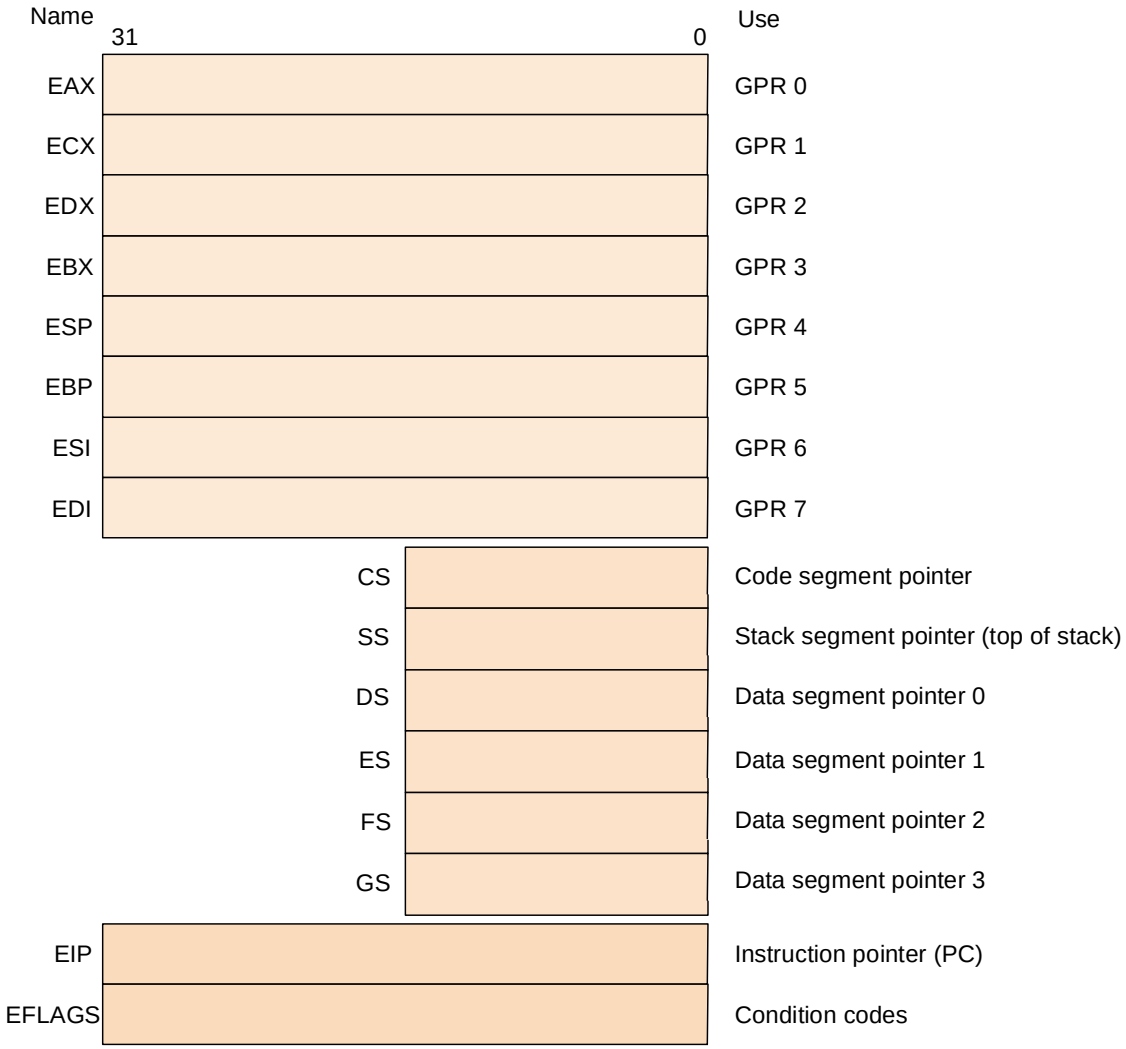
El Intel x86

- Posteriores evoluciones….
  - i486 (1989): segmentación, cachés on-chip y FPU
    - Compatible con competidores: AMD, Cyrix, …
  - Pentium (1993): superescalar, ruta de datos de 64bits
    - Las versiones posteriores agregaron instrucciones MMX (Multi-Media eXtension)
    - El infame error FDIV
  - Pentium Pro (1995), Pentium II (1997)
    - Nueva microarquitectura
  - Pentium III (1999)
    - Se añade SSE (Streaming SIMD Extensions) y registros asociados
  - Pentium 4 (2001)
    - Nueva microarquitectura
    - E añaden instrucciones SSE2

El Intel x86

- l más posterior ….
  - AMD64 (2003): arquitectura extendida a 64 bits
  - EM64T – Tecnología de memoria extendida 64 (2004)
    - AMD64 adoptada por Intel (con refinamientos)
    - Se añaden instrucciones SSE3
  - Intel Core (2006)
    - Se añaden instrucciones SSE4, soporte de máquina virtual
  - AMD64 (anunciada en 2007): instrucciones SSE5
    - Intel se negó a seguirlo, en cambio…
  - Extensión Vectorial Avanzada (anunciado en 2008)
    - Registros SSE más largos, más instrucciones
- Si Intel no ampliara la compatibilidad, ¡sus competidores lo harían!
  - Elegancia técnica ≠ éxito en el mercado

Conjunto de registros básico del x86



Modos de direccionamiento básicos del x86

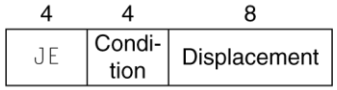
- Dos operandos por instrucción

Fuente/destino operando	Segundo operando fuente
Registro	Registro
Registro	Inmediato
Registro	Memoria
Memoria	Registro
Memoria	Inmediato

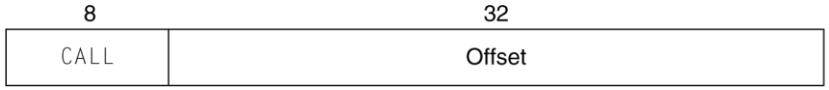
- Modos de direccionamiento a la memoria
  - Direccionamiento en registro
  - Dirección = R_{base} + desplazamiento
  - Dirección = R_{base} + $2e^{scala}$ × $R_{índice}$ (escala = 0, 1, 2, o 3)
  - Dirección = R_{base} + $2e^{scala}$ × $R_{índice}$ + desplazamiento

Formatos de instrucción típicos del x86

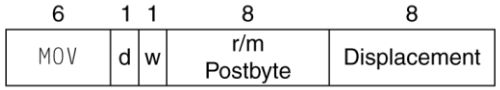
a. JE EIP + displacement



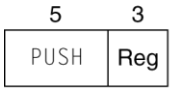
b. CALL



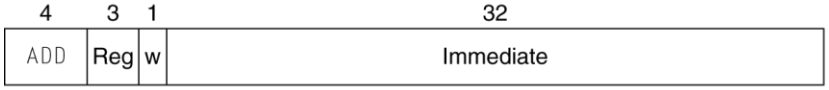
c. MOV EBX, [EDI + 45]



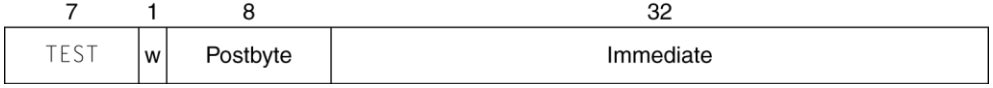
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Formatos de longitud variable (pueden variar de 1 a 17 bytes de longitud)
  - Los bytes de postfijo especifican el modo de direccionamiento
  - Los bytes de prefijo modifican la operación
    - Longitud del operando, repetición, bloqueo…

9. Conclusiones

- Principios de diseño del repertorio de instrucciones
  1. La sencillez favorece la regularidad
  2. Lo pequeño es más rápido
  3. Hacer rápido el caso común
  4. Un buen diseño exige buenos compromisos
- Niveles de software/hardware
  - Compilador, ensamblador, hardware
- MIPS: ejemplo típico de conjunto de instrucciones RISC
  - Comparar con x86

Conclusiones

- Medidas de las ejecuciones de instrucciones MIPS en programas benchmark
  - Ha de considerarse la posibilidad de hacer rápido el caso común
  - Plantearse compromisos

Clases de instrucciones	Ejemplos MIPS	SPEC2006 Int	SPEC2006 FP
Aritméticas	`add, sub, addi`	16%	48%
Transferencia de datos	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Lógicas	`and, or, nor, andi, ori, sll, srl`	12%	4%
Salto condicional	`beq, bne, slt, slti, sltiu`	34%	8%
Salto incondicional	`j, jr, jal`	2%	0%