

Lenguaje ensamblador MIPS

MIPS

- MIPS (Microprocessor without Interlocked Pipeline Stages) es una familia de microprocesadores RISC desarrollada por MIPS Technologies desde 1981 y por Wave Computing desde 2017.
- Usados, entre otros, en algunas consolas de videojuegos de Nintendo y Sony y sistemas empotrados como ruteadores Cisco, dispositivos Windows CE, gateways, etc.
- Más información en https://en.wikipedia.org/wiki/MIPS_instruction_set

MIPS

- El lenguaje ensamblador MIPS es sencillo y fácil de aprender.
- Principio de regularidad. Todas las instrucciones ocupan 4 bytes (32 bits).
- Principio de simplicidad. Instrucciones sencillas. Solo hay 3 formatos de instrucción.

Instrucciones

- Clases de instrucciones:
 - Aritméticas. Suma, resta, multiplicación, división.
 - Transferencia de datos. Carga (load) de la memoria a un registro, almacena (store) de un registro a la memoria.
 - Lógicas. AND, OR, corrimiento (shift).
 - Brincos condicionales.
 - Brincos incondicionales.
- Hay instrucciones para números reales y para enteros.



Operandos

- Registros.
- Memoria.

Registros

- Un registro es una memoria integrada en la CPU.
- Tienen poca capacidad, 4 bytes (32 bits) en MIPS, pero de acceso muy rápido.
- 32 registros enteros: \$s0-\$s7, \$t0-\$t9, \$a0-\$a3, \$v0-\$v1, \$zero, \$gp, \$fp, \$sp, \$ra, \$at.
- 32 registros float: \$f0-\$f31.
- Se usan para guardar valores temporales y resultados de operaciones aritméticas.
- En ese curso veremos solo instrucciones y registros enteros.

Registros

Nombre	Número	Uso
\$zero	0	Constante cero
\$at	1	Reservado
\$v0-\$v1	2-3	Valores de retorno de funciones
\$a0-\$a3	4-7	Argumentos de funciones
\$t0-\$t7	8-15	Temporales
\$s0-\$s7	16-23	Temporales salvados
\$t8-\$t9	24-25	Temporales
\$ko-\$k1	26-27	Reservados por el kernel del S.O.
\$gp	28	Apuntador global
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Dirección de retorno

Memoria

- Se usa para guardar variables.
- MIPS usa byte addressing.
- La memoria es un arreglo de bytes.
- La dirección de una palabra es la dirección de su primer byte.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
...	

Memoria

- Las palabras son de 4 bytes (32 bits).
- Los registros son de 4 bytes (32 bits).
- La memoria está alineada. Las palabras deben comenzar en direcciones que son múltiplo de 4.

0	Datos 32 bits
4	Datos 32 bits
8	Datos 32 bits
12	Datos 32 bits

...

Memoria

- MIPS se puede configurar por hardware como big-endian o little-endian.
- Se usan 32 bits para direccionar la memoria.
- La memoria tiene:
- 2^{32} bytes con direcciones desde 0 hasta $2^{32} - 1$.
- 2^{30} palabras con direcciones desde 0 hasta $2^{32} - 4$.

Registros vs memoria

- La mayoría de las CPUs modernas mueven las variables de la memoria principal a los registros, operan sobre ellos y regresan el resultado a la memoria.
- A esto se le conoce como **arquitectura load/store**.
- MIPS emplea arquitectura load/store.

Instrucciones aritméticas

- add – suma de registros con overflow.
- addu – suma de registros sin overflow.
- addi – suma inmediata con overflow.
- addiu – suma inmediata sin overflow.
- sub, subu – restas con/sin overflow.
- div, divu – divisiones con/sin overflow.
- mult, multu – multiplicaciones con/sin overflow.



Instrucciones aritméticas

- Las sumas tienen 3 operandos.
- El orden de los operandos es fija (primero el operando destino y luego los fuentes).
- Hay dos tipos de sumas:
- Suma de registros: los dos operandos fuentes son registros.
- Suma inmediata: el primer operando es un registro y el segundo es una constante.

Instrucciones aritméticas

- Ejemplos de sumas de registros:

`add $r0, $r1, $r2 # $r0 ← $r1 + $r2`

`addu $r0, $r1, $r2 # $r0 ← $r1 + $r2`

\$r0, \$r1 y \$r2 son registros con números de 32 bits con signo.

- La diferencia entre `add` y `addu` es que, si la suma genera overflow, `add` lo indica y `addu` lo ignora.

Instrucciones aritméticas

- Ejemplos de sumas inmediatas.

`addi $r0, $r1, 20` **# \$r0 ← \$r1 + 20**

`addiu $r0, $r1, 0x20` **# \$r0 ← \$r1 + 32**

\$r0 y \$r1 son registros con signo, la constante es de 16 bits con signo en base 8, 10 o 16.

- La diferencia entre `addi` y `addiu` es que, si la suma genera overflow, `addi` lo indica y `addiu` lo ignora.

Instrucciones aritméticas

- Las restas sub y subu funcionan de la misma manera.

sub \$r0, \$r1, \$r2 # \$r0 ← \$r1 – \$r2

subu \$r0, \$r1, \$r2 # \$r0 ← \$r1 – \$r2

\$r0, \$r1 y \$r2 son registros con signo.

- La diferencia entre sub y subu es que, si la resta genera overflow, sub lo indica y subu lo ignora.



Instrucciones aritméticas

- Las divisiones y multiplicaciones tienen dos argumentos.
- Dejan el resultado en dos registros especiales: LO y HI.
- Hay instrucciones para copiar datos de LO y HI a los registros normales.

Instrucciones aritméticas

mult \$r0, \$r1 # \$r0 x \$r1

multu \$r0, \$r1 # \$r0 x \$r1

- \$r0 y \$r1 tienen números con signo.
- LO tiene los 32 bits bajos del producto.
- HI tiene los 32 bits altos del producto.
- La diferencia entre mult y multu es que, si el producto genera overflow, mult lo indica y multu lo ignora.

Instrucciones aritméticas

div \$r0, \$r1 # \$r0 / \$r1

divu \$r0, \$r1 # \$r0 / \$r1

- \$r0 y \$r1 tienen números con signo.
- LO tiene el cociente (\$r0 DIV \$r1).
- HI tiene el residuo o módulo (\$r0 MOD \$r1).
- La diferencia entre div y divu es que, si la división genera overflow, div lo indica y divu lo ignora.

Transferencia desde LO/HI

- mflo – mueve el valor de LO a un registro regular.
- mfhi – mueve el valor de HI a un registro regular.

mflo \$r # \$r ← LO

mfhi \$r # \$r ← HI

Uso de la memoria

- Por convención, los sistemas basados en MIPS dividen la memoria en 3 segmentos:
 1. Segmento de texto (text segment).
 2. Segmento de datos (data segment).
 3. Segmento de pila (stack segment).

Segmento de texto

- El segmento de texto, que comienza a partir de la dirección 400000_{16} , es donde se guarda el código del programa.
- En los programas, el segmento de texto se marca por medio de la directiva `.text`.

Segmento de datos

- El segmento de datos, que comienza en la dirección 10010000_{16} , es donde se guardan los datos. Se indica por medio de la directiva `.data`.
- A su vez, el segmento de datos consta de dos partes:
 - a) Datos estáticos. Contiene variables de tamaño fijo y que necesitan ser accedidos durante todo el programa. Por ejemplo, variables globales.



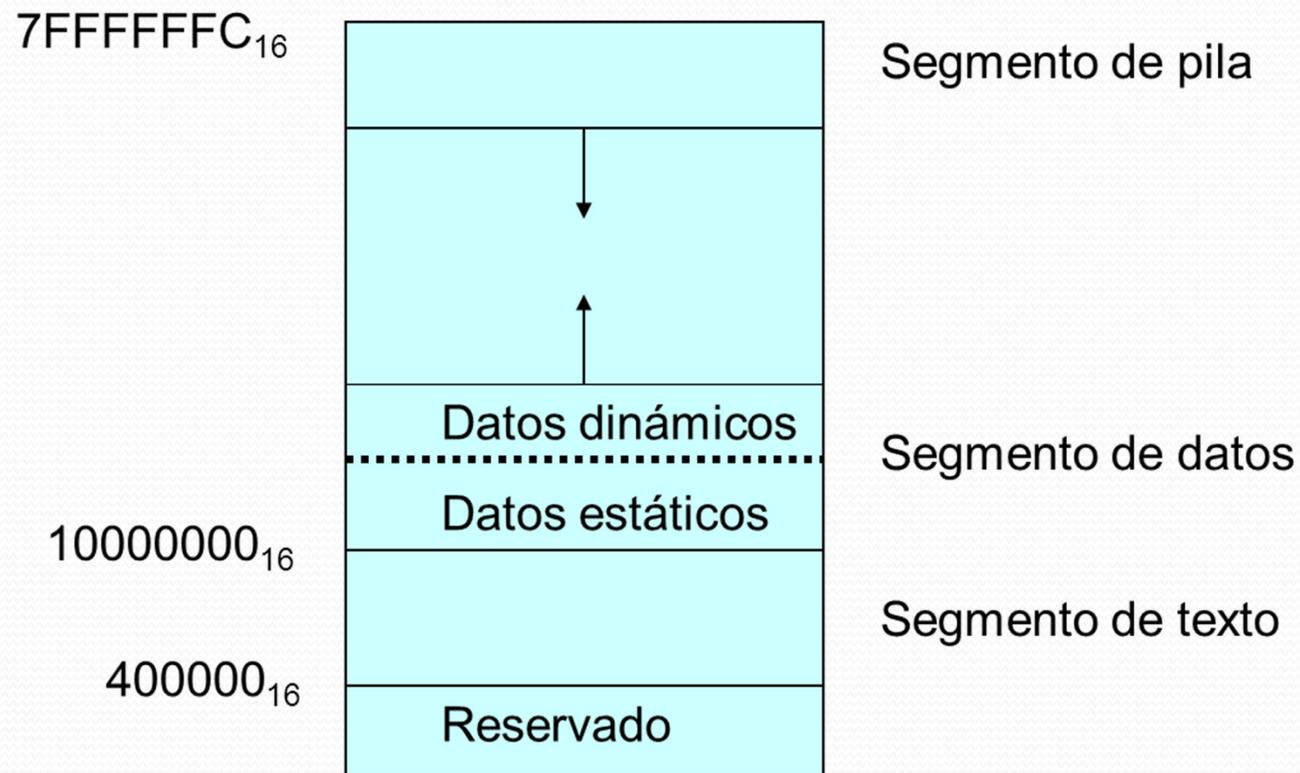
Segmento de datos

- b) Datos dinámicos. Contiene variables que se crean durante el programa. Por ejemplo, espacio asignado con `malloc` en C, u objetos creados en Java.

Segmento de pila

- El segmento de pila, que comienza en la dirección $7FFFFFFC_{16}$, es donde se guardan los stack frames.

Uso de la memoria



Transferencia de datos

- `lw` – mueve (carga) una palabra (4 bytes) de la memoria a un registro.
- `sw` – mueve (almacena) el valor de un registro (4 bytes) a la memoria.

Transferencia de datos

$lw \$r0, C(\$r1) \quad \# \$r0 \leftarrow Mem[\$r1 + C]$

- Copia (carga – load) una palabra (4 bytes) de la memoria a un registro.
- $\$r0$ es el registro destino.
- $\$r1$ es el registro base.
- C es el offset.
- La dirección de la palabra (de su primer byte) en la memoria está dado por $\$r1 + C$.

Transferencia de datos

sw \$r0, C(\$r1) # Mem[\$r1 + C] ← \$r0

- Copia (almacena – store) una palabra (4 bytes) de un registro a la memoria.
- \$r0 es el registro fuente.
- \$r1 es el registro base.
- C es el offset.
- La dirección de la palabra (de su primer byte) en la memoria está dado por $\$r1 + C$.

Variables

- Las variables se guardan en la memoria en el segmento de datos.
- Para hacer operaciones con ellas hay que pasarlas a los registros (arquitectura load/store).
- MIPS maneja varios tipos de variables.
- En ese curso solo se verán enteros, arreglos de enteros y strings.

Variables enteras

- Un entero se declara así:

```
x: .word 17
```

- x es una etiqueta y funciona como nombre de la variable.
- La directiva `.word` está reservando una palabra (4 bytes).
- El número 17 es el valor inicial.
- Otra forma es ver a x como un apuntador al primer byte de una palabra que contiene al 17.

Variables enteras

- Para usar una variable hay que pasarla a un registro:
- Hay al menos dos formas.
- Una:

la \$t0, x # \$t0 tiene la dirección de x

lw \$t1, 0(\$t0) # \$t1 tiene el valor de x

- Otra:

lw \$t1, x # \$t1 tiene el valor de x

Variables enteras

- Para guardar un registro en una variable hay al menos dos formas.

- Una:

la \$t0, x # \$t0 tiene la dirección de x

sw \$t1, 0(\$t0) # Se actualiza la memoria con el
valor de \$t1

- Otra:

sw \$t1, x # Se actualiza la memoria con el
valor de \$t1



Variables enteras

- Los arreglos de enteros se verán más adelante.

Ejemplo

- Generar el código MIPS para:

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

- Actualizando en la memoria la variable f.

Ejemplo

.data

f: .word -1

g: .word 10

h: .word 17

i: .word 12

j: .word -10

Ejemplo

.text

la \$t0, g # **dirección de g**

lw \$s1, 0(\$t0) # **valor de g**

la \$t0, h # **dirección de h**

lw \$s2, 0(\$t0) # **valor de h**

la \$t0, i # **dirección de i**

lw \$s3, 0(\$t0) # **valor de i**

la \$t0, j # **dirección de j**

lw \$s4, 0(\$t0) # **valor de j**

Ejemplo

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

add \$t0, \$s1, \$s2 **# \$t0 $\leftarrow g + h$**

add \$t1, \$s3, \$s4 **# \$t1 $\leftarrow i + j$**

sub \$s0, \$t0, \$t1 **# \$s0 $\leftarrow t0 - t1$**

actualizar f

la \$t0, f

sw \$s0, 0(\$t0)

Strings

- Un string se declara así:

```
mssg: .asciiz "Hola mundo"
```

- `mssg` es una etiqueta y funciona como el nombre del string.
- La directiva `.asciiz` indica que es un string terminado en nulo (ASCII 0).
- "Hola mundo" es el valor inicial.
- Los strings se guardan empacados, 4 caracteres en una palabra.

Strings

- El uso más común de los strings es para imprimir mensajes en la consola.
- Para imprimir un string:
 1. Obtener la dirección.
 la \$t0, mssg
 2. Hacer una llamada al sistema con `syscall`.

Llamadas al sistema

- MIPS no tiene instrucciones de entrada y salida.
- MIPS ofrece llamadas al sistema a través de la instrucción `syscall`.
- Las llamadas permiten leer del teclado, escribir a la consola y manejar archivos del sistema.
- Para hacer una llamada, se carga el número de llamada en `$v0` y los argumentos en `$a0 – $a3`.
- Las llamadas que regresan un valor lo hacen en `$v0`.

Llamadas al sistema

Servicio	Código	Argumentos	Resultado
print_int	1	sa0 = integer	
print_float	2	sf12 = float	
print_double	3	sf12 = float	
print_string	4	sa0 = string	
read_int	5		integer (en svo)
read_float	6		float (en svo)
read_double	7		double (en svo)
read_string	8	sa0 = buffer, sa1 = tamaño	
sbrk	9	sa0 = cantidad	dirección (en svo)
exit	10		
print_char	11	sa0 = char	
read_char	12		char (en svo)
open	13	sa0 = nombre del archivo (string), sa1 = banderas, sa2 = modo	handle (en sa0)
read	14	sa0 = handle, sa1 = buffer, sa2 = tamaño	num. de caracteres leídos (en sa0)
write	15	sa0 = handle, sa1 = buffer, sa2 = tamaño	num. de caracteres escritos (en sa0)
close	16	sa0 = handle	
exit2	17	sa0 = result	

Ejemplo

- Hacer un programa completo en MIPS para el siguiente código en C:

```
int x = 10  
int y = 7  
int z = x + y  
printf("La suma es: %d\n", z);
```

Segmento de datos

```
.data  
x:   .word 10  
y:   .word 7  
z:   .word -1  
m:   .asciiz "La suma es: "  
enter: .asciiz "\n"
```

Segmento de texto

```
.text
```

```
# carga x y y, hace la suma y la guarda en z
```

```
lw $t0, x
```

```
lw $t1, y
```

```
add $t2, $t0, $t1
```

```
sw $t2, z
```

Segmento de texto

imprime el string “La suma es:”

```
addi $v0, $zero, 4  
la $a0, m  
syscall
```

imprime la suma

```
addi $v0, $zero, 1  
add $a0, $zero, $t2  
syscall
```

imprime el enter

```
addi $v0, $zero, 4  
la $a0, enter  
syscall
```

Segmento de texto

exit

addi \$v0, \$zero, 10 **# servicio stop**

syscall



Seudo-instrucciones

- Son instrucciones que un programa ensamblador reconoce pero que no existen en la definición del ISA.
- El ensamblador mapea cada seudo-instrucción a dos o mas instrucciones reales.
- Se usan para comodidad de los programadores.
- Algunas seudo-instrucciones no son standard. Dependen de cada programa ensamblador.

Seudo-instrucciones

- Por ejemplo, la seudo-instrucción `li` (carga inmediata – load immediate) se usa para asignar una constante a un registro:

`li $r, 17`

- Un programa ensamblador la puede traducir a la siguiente instrucción básica:

`addiu $r, $zero, 17`

Seudo-instrucciones

- Otra seudo-instrucción útil es `move`. Sirve para copiar un registro a otro:

```
move $r1, $r2
```

- Un programa ensamblador la puede traducir a la siguiente instrucción básica:

```
addu $r1, $r2, $zero
```

Seudo-instrucciones

- Algunas seudo-instrucciones que veremos:

Nombre	Sintaxis	Significado
Carga inmediata	li \$r, C	$\$r = C$
Copia de registro	move \$r1, \$r2	$\$r1 = \$r2$
Carga de dirección	la \$r, L	$\$r = L$
Variante de carga	lw \$r, L	$\$r = \text{mem}[L]$
Variante de store	sw \$r, L	$\text{mem}[L] = \$r$

Lenguaje máquina

- El lenguaje de máquina es lenguaje binario.
- El hardware solo entiende bits.
- El ensamblador traduce cada instrucción de lenguaje ensamblador a lenguaje de máquina.
- Por ejemplo, la instrucción

add \$t0, \$s1, \$s2

- se traduce en binario a

00000010001100100100100000100000₂

→ 02324820₁₆

Lenguaje máquina

- 00000010001100100100100000100000 se descompone en los siguientes campos:
- 000000 es el código de todas las instrucciones R.
- 10001 es el número del registro \$s1.
- 10010 es el número del registro \$s2.
- 01001 es el número del registro \$t0.
- 00000 es el campo shamt (add no lo usa).
- 100000 es el código de la suma de registros.

Lenguaje máquina

- En general, la instrucción
 `add destino, fuente1, fuente2`
- genera el código máquina
 `000000ffffgggggddddd00000100000`
- donde:
- `000000` es el código de todas las instrucciones R.
- `ffff` es el número del registro fuente 1.
- `ggggg` es el número del registro fuente 2.
- `ddddd` es el número del registro destino.
- `00000` es el `shamt` (`add` no lo usa).
- `100000` es el código de la suma de registros.

Formatos de instrucción

- En MIPS hay 3 formatos de instrucción:
- Formato R. Todos los operandos son registros.
- Formato I. Hay un operando inmediato (número).
- Formato J. La instrucción es un brinco (jump).

Formatos de instrucción

- Ejemplos:

- Formato R.

add \$t0, \$s1, \$s2 # \$t0 = \$s1 + \$s2

- Formato I.

addi \$t0, \$s1, C # \$t0 = \$s1 + C [número de 16

bits]

- Formato J.

j C # brinca a C [dirección de 26 bits]

Formatos de instrucción

Type	-31-	format (bits)				-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Fuente: https://en.wikipedia.org/wiki/MIPS_instruction_set#MIPS_instruction_formats

Ejemplos

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS Instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic Instruction format
I-format	I	op	rs	rt	address			Data transfer format

Fuente: COD 5, p. 85

Operaciones lógicas

- Operan sobre números.
- Principales operaciones lógicas.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

Fuente: COD 5, p. 87

Operaciones lógicas

- sll – corrimiento (shift) a la izquierda.

- Ejemplo:

sll \$t2, \$s0, 4 # en C: $t2 = s0 \ll 4$

- Si \$s0 vale 9:

0000 0000 0000 0000 0000 0000 0000 1001

- \$t2 valdrá 144 (9×2^4)

0000 0000 0000 0000 0000 0000 1001 0000

Operaciones lógicas

- Código binario generado por `sll $t2, $s0, 4`:

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Operaciones lógicas

- Otras operaciones lógicas:
- Corrimiento a la izquierda variable.
sllv \$r0, \$r1, \$r2 # \$r0 ← \$r1 << \$r2
- Corrimiento a la derecha.
srl \$r0, \$r1, C # \$r0 ← \$r1 >> C

Operaciones lógicas

- Corrimiento a la derecha variable.

srlv \$r0, \$r1, \$r2 # \$r0 ← \$r1 >> \$r2

- AND.

and \$r0, \$r1, \$r2 # \$r0 ← \$r1 & \$r2

andi \$r0, \$r1, C # \$r0 ← \$r1 & C

- OR.

or \$r0, \$r1, \$r2 # \$r0 ← \$r1 | \$r2

ori \$r0, \$r1, C # \$r0 ← \$r1 | C

Operaciones lógicas

xor \$r0, \$r1, \$r2 # r0 ← \$r1 ^ \$r2

xori \$r0, \$r1, C # r0 ← \$r1 ^ C

nor \$r0, \$r1, \$r2 # r0 ← \$r1 nor \$r2

- Se puede usar como operador not

nor \$r0, \$r1, \$zero # r0 ← not \$r1

- $(a + 0)' \rightarrow a' . 1 \rightarrow a'$

Operaciones lógicas

- ¿Para qué sirven las operaciones lógicas?
- Los corrimientos a la izquierda (`sll` y `sllv`) son una manera rápida de multiplicar un número por una potencia de 2.
- Los corrimientos a la derecha (`srl` y `srlv`) son una manera rápida de dividir un número entre una potencia de 2.
- El `and`, `or` y los corrimientos se pueden usar para extraer y guardar datos dentro de un número.

Operaciones lógicas

- Ejemplo de extracción de un dato.
- Dado un número binario.

$$x = 11100101$$

- Se desea extraer la información que está en los bits 2 al 5.
- Recordar que los bits se numeran de derecha a izquierda.

Operaciones lógicas

- Dato original.

$$x = 11100101$$

- Se hace un AND con la máscara 00111100:

$$t = x \& 00111100$$

- Ahora t vale 00100100

- Se hace un corrimiento a la derecha:

$$t = t \gg 2$$

- Ahora t vale 00001001 y es lo que se deseaba.

Operaciones lógicas

- Ejemplo de guardar datos en un número.
- Guardar los siguientes datos en un número binario de 16 bits:
- 101 en los bits 1 al 3.
- 0111 en los bits 8 al 11.
- 11 en los bits 14 y 15.
- Dejando los demás en 0.

Operaciones lógicas

- 101 en los bits 1 al 3:

$$t = 0000000000000000101 \ll 1$$

- Ahora t vale 00000000000000001010.

- 0111 en los bits 8 al 11:

- Se hace un OR con la máscara 011100000000

$$t = t | 011100000000$$

- Ahora t vale 0000011100001010

Operaciones lógicas

- 11 en los bits 14 y 15.
- Se hace un OR con la máscara 1100000000000000.
- $t = t | 1100000000000000$
- Ahora t vale 1100011100001010 que es lo que se deseaba.

Operaciones lógicas

- Para hacer máscaras recordar:
 - $x \text{ AND } 0 = 0$
 - $x \text{ AND } 1 = x$
 - $x \text{ OR } 0 = x$
 - $x \text{ OR } 1 = 1$



Instrucciones de control

- Alteran el flujo de control.
- En otras palabras, cambian la siguiente instrucción que será ejecutada.
- Brincos condicionales (branches).
- Brincos incondicionales (jumps).

Brincos condicionales

- En MIPS hay dos instrucciones de brinco condicional:

1. `beq $s, $t, C` **# brinca a la dirección C si $\$s == \t**

2. `bne $s, $t, C` **# brinca a la dirección C si $\$s \neq \t**

Brincos condicionales

- Ejemplo:

C:

```
if (i == j)
    h = i + j;
```

MIPS:

```
bne $t1, $t2, L1      # $t1: valor de i, $t2: valor de j
add $t0, $t1, $t2     # $t0: valor de h
```

L1: ...

Brincos incondicionales

- En MIPS hay tres instrucciones de brinco incondicional:

1. `j C` **# brinca a la dirección C**
2. `jr $r` **# brinca a la dirección guardada en \$r**
3. `jal C` **# llama al procedimiento que comienza en C**
la dirección de regreso se guarda en \$ra
es decir, se regresa con `jr $ra`

Ejemplo de jump

C:

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

MIPS:

```
    bne $s1, $s2, L1      # $s1 = i, $s2 = j
    sub $s0, $s1, $s2    # $s0 = h
    j L2                 # brinca al final
L1:  add $s0, $s1, $s2    # $s0 = h
L2:  ...
```

Ejemplo de jal y jr

C:

```
main()
{
    ...
    foo ();
    ...
}
```

```
void foo ()
{
    ...
}
```

Ejemplo de jal y jr

MIPS:

...

jal foo

brinca a foo, guarda la dirección de

...

regreso en el registro \$ra

foo:

...

...

jr \$ra

regresa al programa principal

Brincos

- MIPS tiene beq (brinca si es igual) y bne (brinca si no es igual).
- ¿Y si la condición es $if (i < j)$?
- Hay dos soluciones:
 - a) Usar una pseudo-instrucción (código más entendible).
 - b) Usar la instrucción básica slt (código no tan entendible).

slt

- Set if less than:

`slt $t0, $t1, $t2`

- `$t0` vale 1 si `$t1` es menor que `$t2` y vale 0 en otro caso.

- Equivale en pseudo C a

`$t0 = ($t1 < $t2)`

- `slt` se usa junto con `beq` o `bne` para condiciones tipo `if (i < j)` o `if (i <= j)`.

Un ejemplo de slt

C:

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

MIPS:

```
slt $s3, $s1, $s2           # $s1 = i, $s2 = j $s3 = i < j
beq $s3, $zero, L1         # $zero siempre vale 0
add $s0, $s1, $s2         # then h = i + j
j L2                       # brinca al final
L1: sub $s0, $s1, $s2     # else h = i - j
L2: ...
```

Otro ejemplo de slt

C:

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

MIPS:

```
                beq $s1, $s2, L3           # brinca si $s1 == $s2
                slt $s3, $s1, $s2         # checa su $s1 < $s2
                beq $s3, $zero, L1         # $zero siempre vale 0
L3:             add $s0, $s1, $s2         # then h = i + j
                j L2                       # brinca al final
L1:             sub $s0, $s1, $s2         # else h = i - j
L2:             ...
```

Seudo-instrucciones de brincos

- Algunas seudo-instrucciones de brincos:

Nombre	Sintaxis	Significado
Brinca si mayor que	bgt \$r, \$t, L	Si $r > t$ goto L
Brinca si menor que	blt \$r, \$t, L	Si $r < t$ goto L
Brinca si mayor que o igual	bge \$r, \$t, L	Si $r \geq t$ goto L
Brinca si menor que o igual	ble \$r, \$t, L	Si $r \leq t$ goto L
Brinca si mayor que sin signo	bgtu \$r, \$t, L	Si $r > t$ goto L
Brinca si mayor que cero	bgtz \$r, L	Si $r > 0$ goto L

Un ejemplo

C:

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

MIPS:

```
blt $s1, $s2, L1          # $s1 = i, $s2 = j
sub $s0, $s1, $s2        # else h = i - j
j L2                     # brinca al final
L1: add $s0, $s1, $s2    # then h = i + j
L2: ...
```

Otro ejemplo

C:

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

MIPS:

```
ble $s1, $s2, L1          # $s1 = i, $s2 = j
sub $s0, $s1, $s2        # else h = i - j
j L2                     # brinca al final
L1: add $s0, $s1, $s2    # then h = i + j
L2: ...
```



Traducción mecánica de ciclos

- Dado un ciclo while, do-while, for en C o Java, traducirlo a MIPS.
- Es una de las primeras tareas al diseñar un compilador.

Ciclo while

```
while (condición) {  
    instrucciones;  
}
```

Seudo MIPS:

```
L1:  if condicion == false goto L2  
    instrucciones  
    goto L1  
L2:  ...
```

Ejemplo de while

```
while (i <= j) {  
    instrucciones;  
}
```

```
L1:    bgt $s0, $s1, L2    # $s0 = i, $s1 = j  
    instrucciones  
    j L1  
L2:    ...
```

Ciclo for

```
for (precondición; condición; postcondición) {  
    instrucciones;  
}
```

Seudo MIPS:

```
precondición  
L1:  if condición == false goto L2  
    instrucciones  
    postcondición  
    j L1  
L2:
```

Ejemplo de for

```
for (int i = 0; i < 10; i++) {  
    instrucciones;  
}
```

```
    li $t0, 0          # $t0 = i  
L1:  bge $t0, 10, L2  # if i >= 10 goto L2  
    instrucciones  
    addi $t0, $t0, 1  # i++  
    j L1  
L2:  ...
```

Ciclo do-while

```
do {  
    instrucciones;  
} while (condición);
```

- Seudo MIPS:

```
L1:  instrucciones  
    if condición == true goto L1  
...
```

Ejemplo de do-while

```
do {  
    instrucciones;  
} while (i == j);
```

```
L1:  instrucciones  
    beq $t0, $t1, L1  # $t0 = i, $t1 = j  
    ...
```

Arreglos de enteros

- Se declaran en el segmento de datos usando la directiva `.word`.
- Ejemplo

a: `.word 7, 2, 1, 9, 10` # arreglo

n: `.word 5` # tamaño

Arreglos de enteros

- Para acceder los elementos:

la $\$t0$, a **# $\$t0$ tiene la dirección de $a[0]$**

lw $\$s0$, 0($\$t0$) **# $\$s0 = a[0]$**

lw $\$s1$, 4($\$t0$) **# $\$s1 = a[1]$**

lw $\$s2$, 8($\$t0$) **# $\$s2 = a[2]$**

- ...
- Notar que el offset debe ser una constante.

El primer ejemplo

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



swap:

```
mulh $v0, $a1, 4
add $v0, $a0, $v0
lw $t7, 0($v0)
lw $s0, 4($v0)
sw $s0, 0($v0)
sw $t7, 4($v0)
jr $ra
```

El primer ejemplo

swap:

```
muli $v0, $a1, 4    # v0 ← a1 x 4, a1 tiene el valor de k
add $v0, $a0, $v0   # v0 ← a0 + v0, a0 apunta a v[0], v0 apunta a v[k]
lw $t7, 0($v0)      # t7 ← mem[v0], t7 tiene el valor de v[k]
lw $s0, 4($v0)      # s0 ← mem[v0 + 4], t7 tiene el valor de v[k + 1]
sw $s0, 0($v0)      # mem[v0] ← s0, s0 se guarda en v[k]
sw $t7, 4($v0)      # mem[v0 + 4] ← t7, t7 se guarda en v[k + 1]
jr $ra              # return
```

El primer ejemplo

- En C

```
swap (int v[], int k)
{
    int t1 = v[k];
    int t2 = v[k + 1];
    v[k] = t2;
    v[k + 1] = t1;
}
```

Ejemplo 1

- Generar el código MIPS para:

$$a[12] = h + a[8]$$

- Suponiendo que el valor de h está en $\$s2$ y que la dirección del arreglo a está en $\$s3$.

Ejemplo 1

$a[12] = h + a[8]$

lw \$t0, 32(\$s3) # $t0 \leftarrow a[8]$

add \$t0, \$s2, \$t0 # $t0 \leftarrow h + a[8]$

sw \$t0, 48(\$s3) # $a[12] \leftarrow t0$

Ejemplo 2

- Traducir a MIPS el siguiente código en C:
while (save[i] == k)
 i++;
- Suponiendo que:
- i está en \$s3, k en \$s5 y la dirección del arreglo save está en \$s6.

Ejemplo 2

Loop:

```
sll $t1, $s3, 2
```

```
# $t1 = i * 4
```

```
add $t1, $t1, $s6
```

```
# $t1 = address of save[i]
```

```
lw $t0, 0($t1)
```

```
# $t0 = save[i]
```

```
bne $t0, $s5, Exit
```

```
# go to Exit if save[i] ≠ k
```

```
addi $s3, $s3, 1
```

```
# i = i + 1
```

```
j Loop
```

```
# go to Loop
```

Exit:

Ejemplo 3

- Traducir a MIPS este código en C:

```
int a[] = {7, 10, 2, 9, 8};  
int i, n = 5, suma = 0;  
for (i = 0; i < n; i++) {  
    suma += a[i];  
}  
printf("La suma es: %d\n", suma);
```

Segmento de datos

```
.data  
a:    .word 7, 10, 5, 9, 8  
n:    .word 5  
suma: .word 0  
mssg: .asciiz "La suma es: "  
enter: .asciiz "\n"
```

Segmento de texto

Parte inicial

```
.text
```

```
li $t0, 0      # i  
lw $t1, n      # t1 = n  
li $t2, 0      # suma  
la $t3, a      # t3 = &a[0]
```

Segmento de texto

Ciclo de suma

```
for:   bge $t0, $t1, fin
        lw $t4, 0($t3)           # t4 = a[t3]
        add $t2, $t2, $t4       # suma
        addi $t0, $t0, 1        # i++
        addi $t3, $t3, 4        # apuntador
j for
```

Segmento de texto

actualiza suma en la memoria

fin: la \$t5, suma

sw \$t2, 0(\$t5)

Segmento de texto

Imprime

li \$v0, 4 **# print mssg**

la \$a0, mssg

syscall

li \$v0, 1 **# print suma**

move \$a0, \$t2

syscall

li \$v0, 4 **# print enter**

la \$a0, enter

syscall

Segmento de texto

stop

li \$v0, 10

syscall



Llamadas a funciones

- En lenguajes de alto nivel (C, Java) las llamadas a procedimiento son transparentes al usuario.
- En ensamblador, el programador debe implementar las llamadas y retorno de procedimiento.
- Las llamadas y regreso de procedimiento involucran manejar bloques de memoria llamados call frames o stack frames.
- Los frames se guardan en el segmento de pila.

Llamadas a funciones

- Hay dos componentes en una llamada:
 - La función que invoca o llamador (caller).
 - La función invocada o llamada (callee).
- Ejemplo

```
void main()      // llamador
{
    foo();
}
void foo()      // llamado
{
}
```

En MIPS

```
.text
```

```
...
```

```
jal foo
```

brinca a foo, guarda la dirección

```
...
```

de regreso en el registro \$ra

```
foo: ...
```

```
...
```

```
jr $ra
```

regresa al programa principal

Llamadas a funciones

- La instrucción `jal` no puede tener argumentos.
- No hay un equivalente a:

`a = foo(x, y, z);`

- Los argumentos se pasan en registros antes de llamar a `jal`.
- De la misma forma no hay un equivalente a:
`return w;`
- La función llamada debe regresar valores en los registros.

Llamadas a funciones

- Por convención los registros \$a0, \$a1, \$a2 y \$a3 se usan para pasar argumentos.
- Y los registros \$v0 y \$v1 para regresar valores al llamador.

Llamadas a funciones

- ¿Una función tiene derecho a que los registros mantengan su valor después de llamar a otra función?

antes de la llamada

```
li $t0, 77
```

llamada

```
jal foo
```

después de la llamada

```
addi $t1, $t0, $zero
```

Registros

- Hay una convención de que registros deben ser guardados por el llamado.
- El llamador tiene derecho a esperar que esos registros mantengan su valor después de hacer una llamada a una función.

Registros

Nombre	Número	Uso	¿Salvado por el llamado?
\$zero	0	Constante cero	No aplica
\$at	1	Reservado	No
\$v0-\$v1	2-3	Valores de retorno de funciones	No
\$a0-\$a3	4-7	Argumentos de funciones	No
\$t0-\$t7	8-15	Temporales	No
\$s0-\$s7	16-23	Temporales salvados	Si
\$t8-\$t9	24-25	Temporales	No
\$ko-\$k1	26-27	Reservados por el kernel del S.O.	No
\$gp	28	Apuntador global	Si
\$sp	29	Stack pointer	Si
\$fp	30	Frame pointer	Si
\$ra	31	Dirección de retorno	No aplica

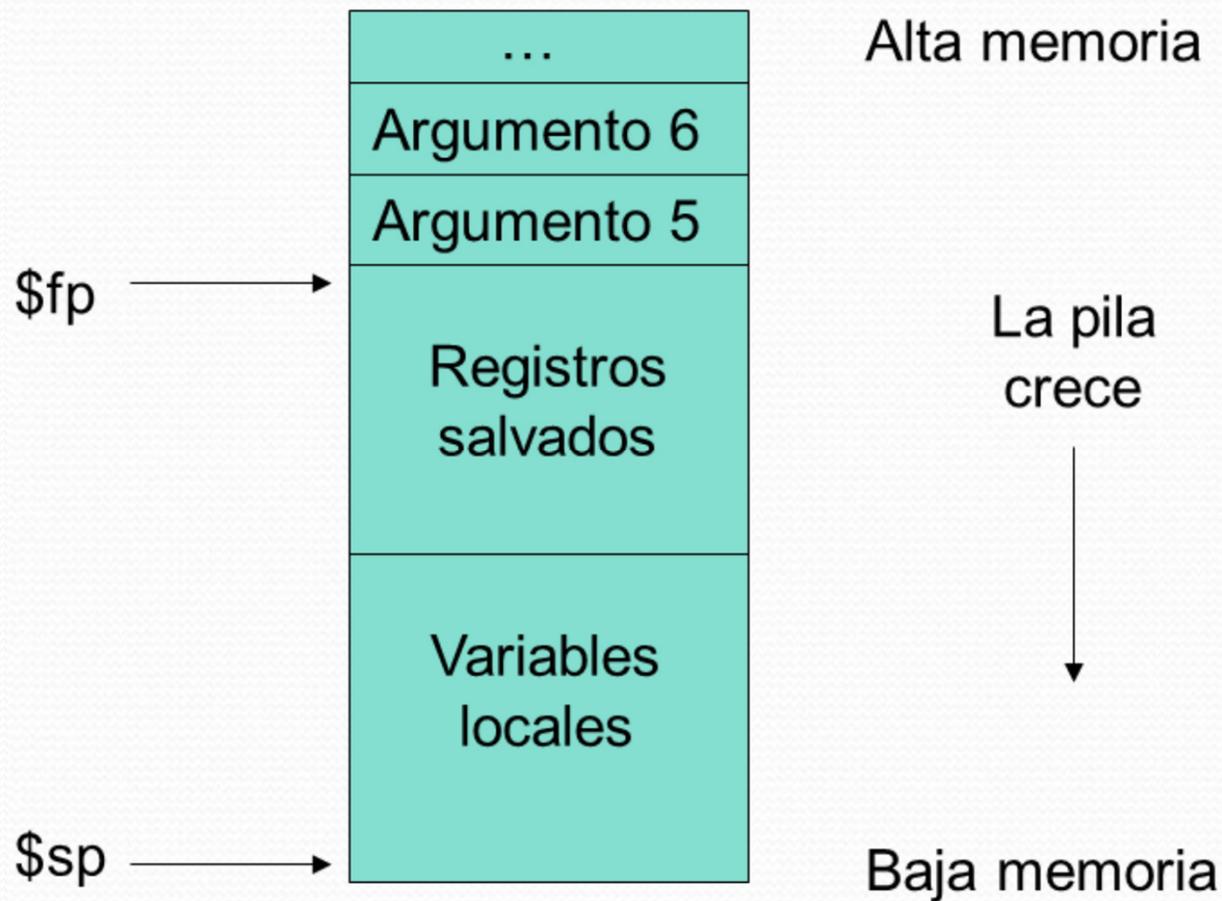
Registros

- El llamador si tiene derecho a esperar que los registros `$s0` a `$s7`, `$gp`, `$sp` y `$fp` mantengan sus valores después de una llamada.
- Si al llamador le interesa preservar el valor de algún otro registro debe guardarlo antes de llamar a la función.
- El llamado debe guardar los registros que vaya a usar y dejarlos como estaban antes de regresar.
- Además, si el llamado a su vez llama a otra función debe guardar el registro `$ra`.

Stack frame

- El llamado utiliza una estructura llamada stack frame para guardar esos registros.
- Los stack frames se guardan en el segmento de pila.
- El stack frame guarda:
 - Argumentos (a partir del quinto) al procedimiento.
 - Los registros que se deben preservar.
 - Las variables locales al procedimiento.
- Se puede usar el frame pointer (`$fp`) para acceder datos en el stack frame.

Stack frame



Antes de la llamada

- El llamador hace lo siguiente:
 1. Pasar argumentos. Por convención, los primeros 4 argumentos van en $\$a0$ a $\$a3$. Los demás argumentos se pasan en la pila.
 2. Guarda los registros. No hay garantía que $\$a0$ a $\$a3$ ni $\$t0$ a $\$t9$ mantengan su valor después de la llamada. Si el llamador espera usar alguno de esos registros, debe salvarlos en la pila.
 3. Ejecuta una instrucción jal . La dirección de regreso se guarda en $\$ra$ automáticamente.

Después de la llamada

- El llamado, antes de correr, hace lo siguiente:
 1. Reservar espacio para el frame, restándole al stack pointer el tamaño del frame.
 2. Salvar los registros $\$s0$ a $\$s7$ si es que se usan en el procedimiento. $\$fp$ debe salvarse, $\$gp$ debe salvarse si se usa y $\$ra$ solo en caso de que la función haga a su vez una llamada.
 3. Dejar a $\$fp$ apuntando al comienzo del frame, sumando al stack pointer el tamaño del frame menos 4.

Antes de regresar

- Antes de regresar, el llamado hace lo siguiente:
 1. Si el llamado regresa un valor, guardar el valor en \$v0.
 2. Restaurar los registros que se salvaron al comienzo.
 3. Eliminar (pop) el stack frame sumándole el tamaño del frame al stack pointer.
 4. Brincar a la dirección almacenada en \$ra.

Ejemplo

- Traducir a MIPS el siguiente código en C:

```
void main()
{
    int valor = foo(7, 10, 5, 4);
    printf(“%d\n”, valor);
}
int foo (int g, int h, int i, int j)
{
    int f = (g + h) - (i + j);
    return f;
}
```

Segmento de datos

.data

valor: .word 0

mssg: .ascii "El valor es: "

Segmento de texto

Guarda los argumentos y hace la llamada

```
.text  
li $a0, 7  
li $a1, 10  
li $a2, 5  
li $a3, 4  
jal foo
```

Segmento de texto

Después de la llamada se actualiza valor y se imprime el mensaje y el valor.

```
    move $t0, $v0    # foo regresa la suma en $v0
    sw $t0, valor    # actualiza valor
    li $v0, 4        # print mssg
    la $a0, mssg
    syscall
    li $v0, 1        # print valor
    move $a0, $t0
    syscall
    li $v0, 10       # stop
    syscall
```

Segmento de texto

Prólogo de la función.

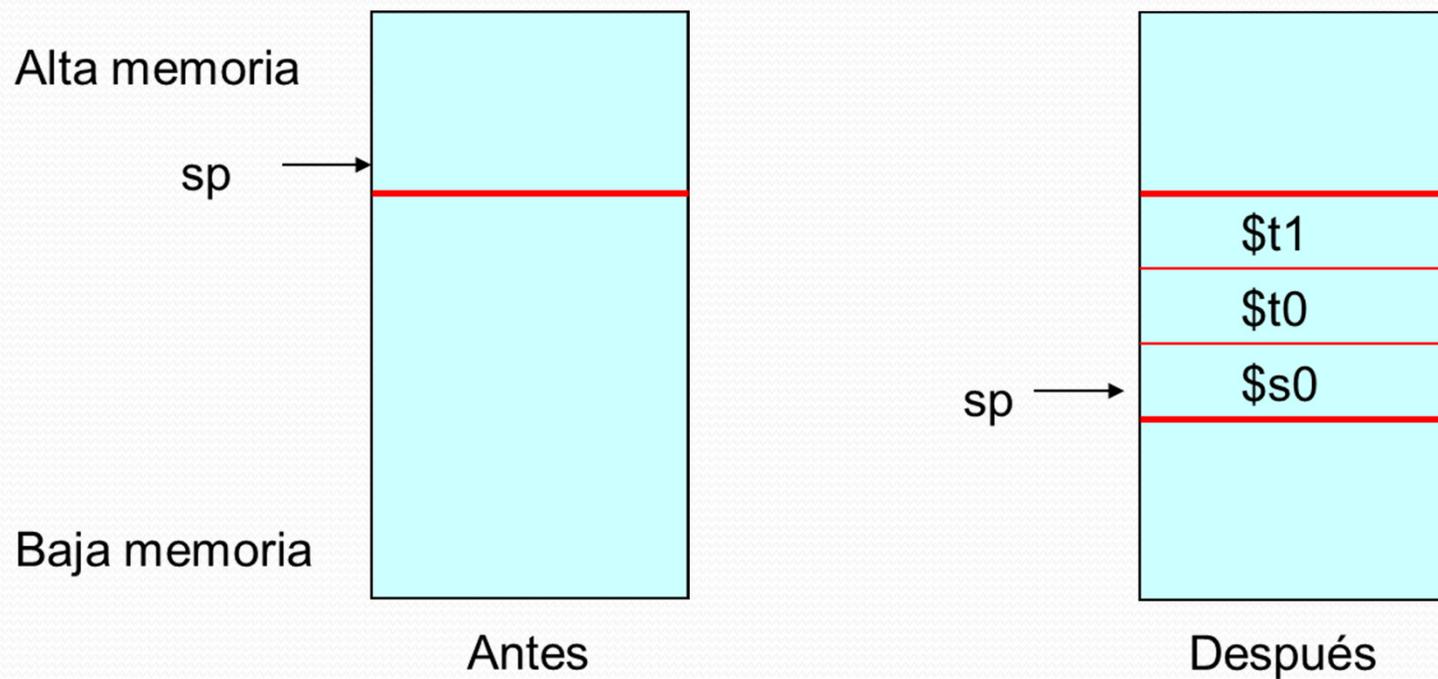
Reserva espacio en la pila para el frame.

Salva los registros que se van a usar: \$s0, \$t0 y \$t1.

```
foo:   addi $sp, $sp, -12    # el frame ocupa 12 bytes
      sw $t1, 8($sp)      # guarda $t1
      sw $t0, 4($sp)      # guarda $t0
      sw $s0, 0($sp)      # guarda $s0
```

Pila

- Pila del sistema antes y después de guardar el stack frame de foo.



Segmento de texto

Código del procedimiento.

Los argumentos g, h, i, y j están en \$a0, \$a1, \$a2 y \$a3, respectivamente.

```
add $t0, $a0, $a1    # $t0 = g + h
add $t1, $a2, $a3    # $t1 = i + j
sub $s0, $t0, $t1    # $s0 = $t0 - $t1
move $v0, $s0        # el valor se regresa en $v0
```

Segmento de texto

Epílogo de la función.

Restaura los valores de \$t0, \$t1 y \$s0 sacándolos de la pila.

Regresa el control al procedimiento llamador.

lw \$s0, 0(\$sp)

restaura \$s0

lw \$t0, 4(\$sp)

restaura \$t0

lw \$t1, 8(\$sp)

restaura \$t1

addi \$sp, \$sp, 12

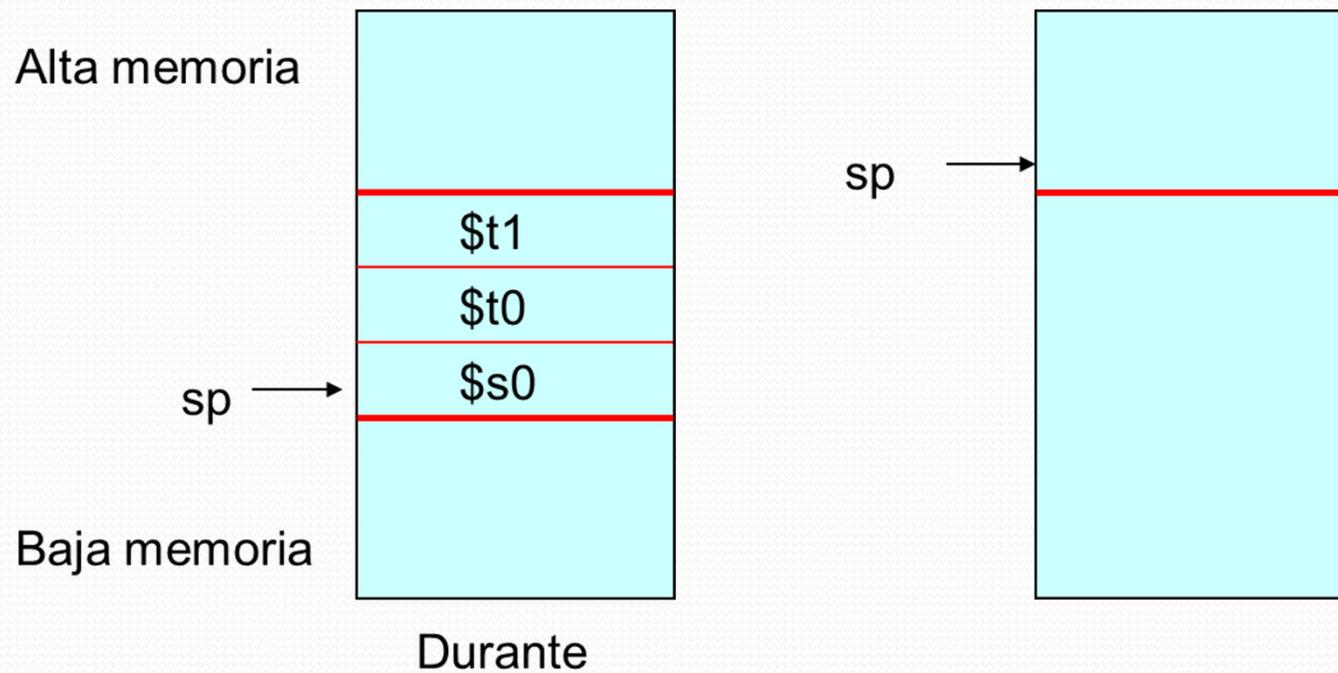
ajusta la pila

jr \$ra

brinca al llamador

Pila

- Pila del sistema antes y después de que foo regrese.



Constantes de 32 bits

- A veces se necesita manejar constantes o direcciones de 32 bits.
- La instrucción `addi` tiene formato I y solo maneja constantes de 16 bits.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

`addi $t0, $zero, 0x1234`

Constantes de 32 bits

- Para cargar constantes de 32 bits se utilizan dos instrucciones:
- lui – load upper immediate.
- ori – OR immediate.

Ejemplo

- Cargar 0000 0000 0011 1101 0000 1001 0000 0000 en \$s0.
- Se escribe en base 16: 0x003d0900
lui \$s0, 0x3d
ori \$s0, \$s0, 0x0900

Addressing mode

- Se puede traducir como “modo de direccionamiento.”
- Especifica cómo calcular la dirección de memoria efectiva de un operando.
- Se utiliza la información contenida en registros y/o constantes contenidas dentro de una instrucción.

Direcciones en jumps

- Los jumps utilizan el formato J: 6 bits para el opcode y 26 bits para la dirección.

- Ejemplo:

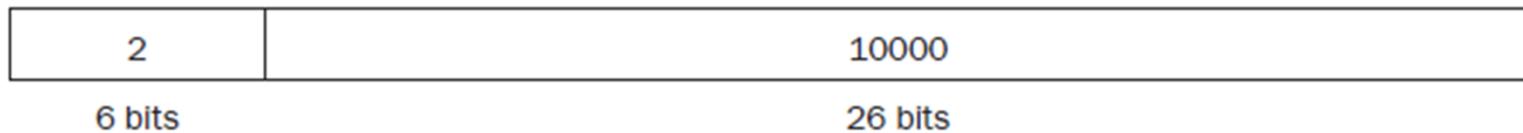
39700 j Label

...

...

40000 Label: ...

- La instrucción j genera el siguiente código:



Direcciones en jumps

- Donde:
- 2 es el opcode de j.
- 10000 es la dirección en palabras.
- A esto se le llama **direccionamiento seudodirecto**.

Direcciones en branches

- Los brincos condicionales (branches) son relativos al contador de programa (PC).
- Ejemplo:

```
1000      beq $s0, $s1, Exit
```

```
...      ...
```

```
1400 Exit:      ...
```

- Genera el siguiente código:

```
000100 10000 10001 0000000001100011
```

```
beq    $s0    $s1        99
```

Direcciones en branches

- Por conveniencia, el PC se incrementa en 4 *antes* de checar la condición.
- El brinco es relativo a la siguiente instrucción (PC + 4).
- La diferencia se calcula en palabras no en bytes.
- Hay 99 palabras entre 1004 y 1400.
- A esto se le llama **direccionamiento relativo al PC**.

Ejemplo

```

Loop: sll $t1,$s3,2      # Temp reg $t1 = 4 * i
      add $t1,$t1,$s6    # $t1 = address of save[i]
      lw  $t0,0($t1)     # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3,$s3,1     # i = i + 1
      j   Loop          # go to Loop
Exit:

```

If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

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

Brincos lejanos

- beq y bne son formato I.
- Solo hay 16 bits para guardar el offset del brinco.
- Para brincos lejanos se puede usar un j.

- Por ejemplo

beq \$s0, \$s1, L1

- Se puede reemplazar por:

bne \$s0, \$s1, L2

j L1

L2: ...

Adressing modes

- En MIPS:
 1. Inmediato. El operando es una constante. Ej: `addi $s0, $s0, 4`
 2. Registro. El operando es un registro. Ej: `jr $ra`
 3. Base. El operando está en memoria con dirección dada por la suma de un registro base y una constante. Ej: `lw $s0, 4($t0)`

Modos de direccionamiento

4. Relativo al PC. La dirección es la suma del PC y una constante. Ej: `beq $s0, $s1, L1`
5. Seudodirecto. La dirección consta de los 26 bits guardados en la instrucción concatenados con la parte alta del PC. Ej: `j Loop`

Resumen

- Instrucciones sencillas, todas de 32 bits.
- Tres formatos de instrucción:
 - R – los operandos son registros.
 - I – un operando es inmediato.
 - J – brincos.

Resumen

- Arquitectura load/store.
- Los datos deben estar en registros para realizar operaciones aritméticas o lógicas.
- Las únicas instrucciones que accesan la memoria son las instrucciones de transferencia de datos.
- Byte addressing con memoria alineada.
- Las palabras comienzan en direcciones múltiplos de 4.
- 2^{32} bytes con direcciones de 0 a $2^{32} - 1$.

Resumen

- El registro `$zero` siempre contiene valor 0.
- Los registros `$at`, `$k0`, `$k1` están reservados y no deben usarse en programas.
- Los registros `$gp`, `$sp`, `$fp` y `$ra` son de propósito especial.
- En particular `$sp`, `$fp` y `$ra` se usan en las llamadas.
- El registro `$gp` es el apuntador global (global pointer). Por default vale `0x10008000`. Se puede usar para apuntar al final de los datos estáticos y comienzo de los datos dinámicos.

Resumen

- El registro $\$sp$ es el stack pointer, que apunta al tope de la pila.
- El registro $\$fp$ es el frame pointer. Se puede usar para apuntar al stack frame.
- $\$ra$ guarda la dirección de retorno de una llamada a función. La instrucción `jal` actualiza $\$ra$ de forma automática.

Resumen

- Los registros \$t0 a \$t9 son para temporales de vida corta.
- Los registros \$s0 a \$s7 son para temporales de vida larga.
- Los registros \$a0 a \$a3 se usan para pasar argumentos.
- Los registros \$v0 y \$v1 se usan para regresar valores.