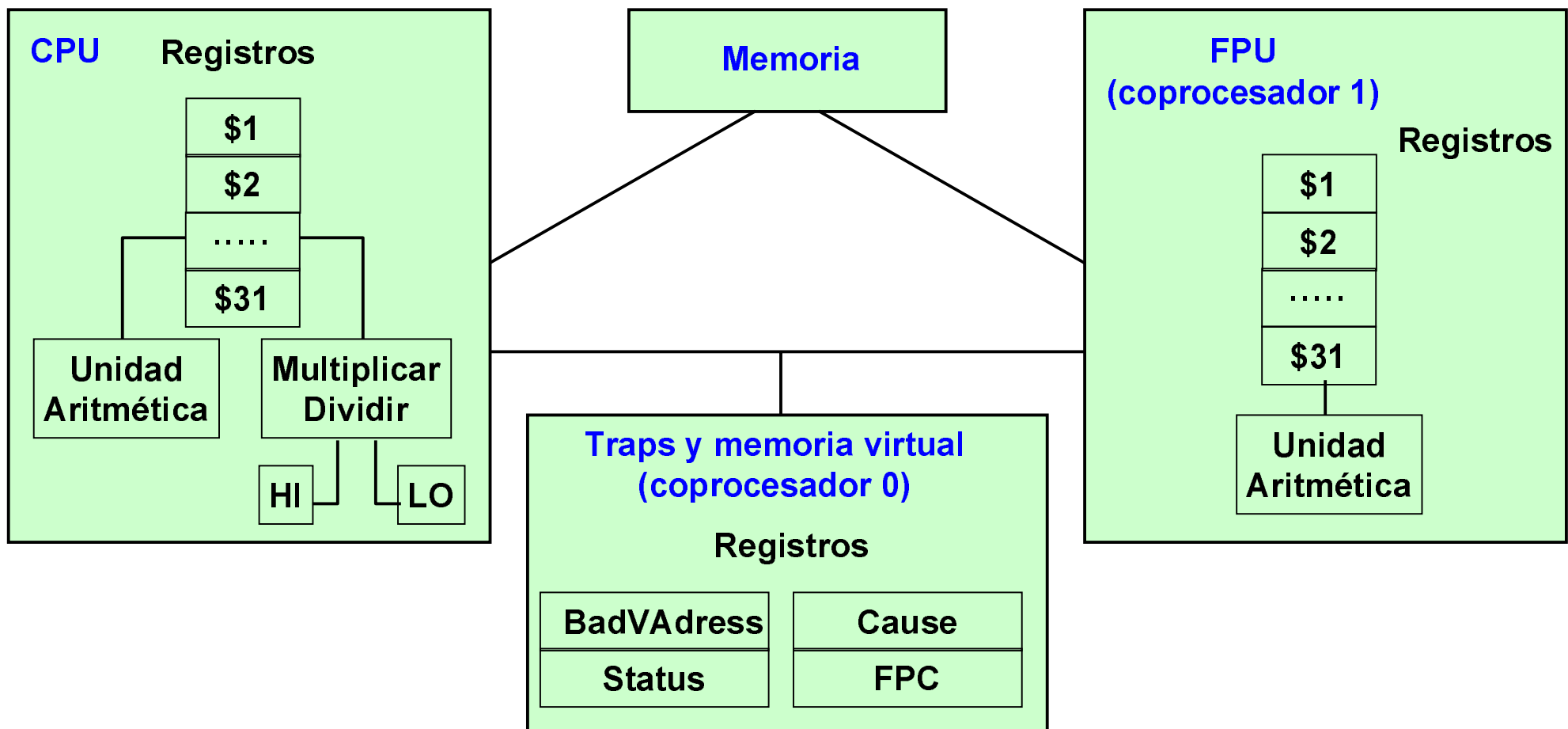




ENSAMBLADOR MIPS

- La arquitectura MIPS es un ejemplo simple y claro de una máquina RISC de 32 bits.
 - » Procesador MIPS = CPU + coprocesadores auxiliares
 - » Coprocesador 0 = excepciones, interrupciones y sistema de memoria virtual
 - » Coprocesador 1 = FPU (Unidad de Punto Flotante)



- ◆ **Lenguaje ensamblador** es la representación simbólica de la codificación binaria del computador (**lenguaje máquina**).
- ◆ Los lenguaje máquina son bastante similares.
- ◆ Una herramienta llamada **ensamblador** (assembler) traduce el lenguaje ensamblador a instrucciones binarias.
- ◆ Un ensamblador lee un único **archivo fuente** en lenguaje ensamblador, y produce un **archivo objeto** que contiene instrucciones máquina e información de mantenimiento que ayuda a combinar varios archivos objeto en un programa.
- ◆ Un módulo puede tener referencias a subrutinas y datos definidos en otros módulos y bibliotecas. El código de un módulo no puede ejecutarse cuando contiene referencias sin resolver a rótulos de otros archivos objeto o bibliotecas. Otra herramienta llamada enlazador (linker) combina archivos objeto y de biblioteca en un archivo ejecutable.
- ◆ El lenguaje ensamblador juega dos papeles:
 - » Es el lenguaje de salida de los compiladores. Un compilador traduce un programa escrito en lenguaje de alto nivel (lenguaje fuente, C) en un programa equivalente en lenguaje máquina o ensamblador (lenguaje objeto).
 - » Es un lenguaje más con el que escribir programas en los que la velocidad o tamaño son críticos, o para explotar las características hardware que no tienen análogos en lenguajes de alto nivel.

- Cada línea puede contener, como máximo, una **sentencia**.
- Los **comentarios** van al final de la sentencia, precedidos por **#**. El ensamblador los ignora.
- Los nombres seguidos por **:** son **rótulos** (etiquetas). Nombran la siguiente posición de memoria, y van al principio de la línea

sentencia

Rótulo	Código instrucción, pseudo instr. ó directiva	Operandos	Comentarios
--------	--	-----------	-------------

- Los operandos pueden ser:
 - » Un registro: **\$4** o **\$a0**, que puede ir entre paréntesis.
 - » Un identificador: Caracteres alfanuméricos, **_** y **.**, sin comenzar con un número.
 - » Una cadena alfanumérica encerrada entre dobles comillas ("**"**), donde puede haber
 - » caracteres especiales según convenio C (nueva línea=**\n**, tabulador=**\t**, comillas=**\"**...).
 - » Un valor numérico, en base 10. Si van precedidos de **0x** indican valores hexadecimal

- Ejemplo:

```

        .data
item:    .word 1
        .text
        .globl main      # Debe ser global
main:    lw $t0, item
    
```

Un programa puede estar escrito en códigos distintos:

- **M.C** Código en lenguaje de alto nivel (fuelle).
- **M.S** Código ensamblador. Puede jugar dos papeles, dependiendo del tipo de compilador:
 - » Es el lenguaje de salida de los compiladores (código ensamblador sin rótulos).
 - » Lenguaje con el que escribir programas (código ensamblador con rótulos). Es aún importante para escribir programas en los que la velocidad o tamaño son críticos, o para explotar características hardware que no tienen análogos en lenguajes de alto nivel.
- **M.OBJ** Código máquina (objeto). Contiene instrucciones máquina (binarias) e información de mantenimiento que ayuda a combinar varios archivos objeto en un programa.
- **M.EXE** Código ejecutable.

compilador

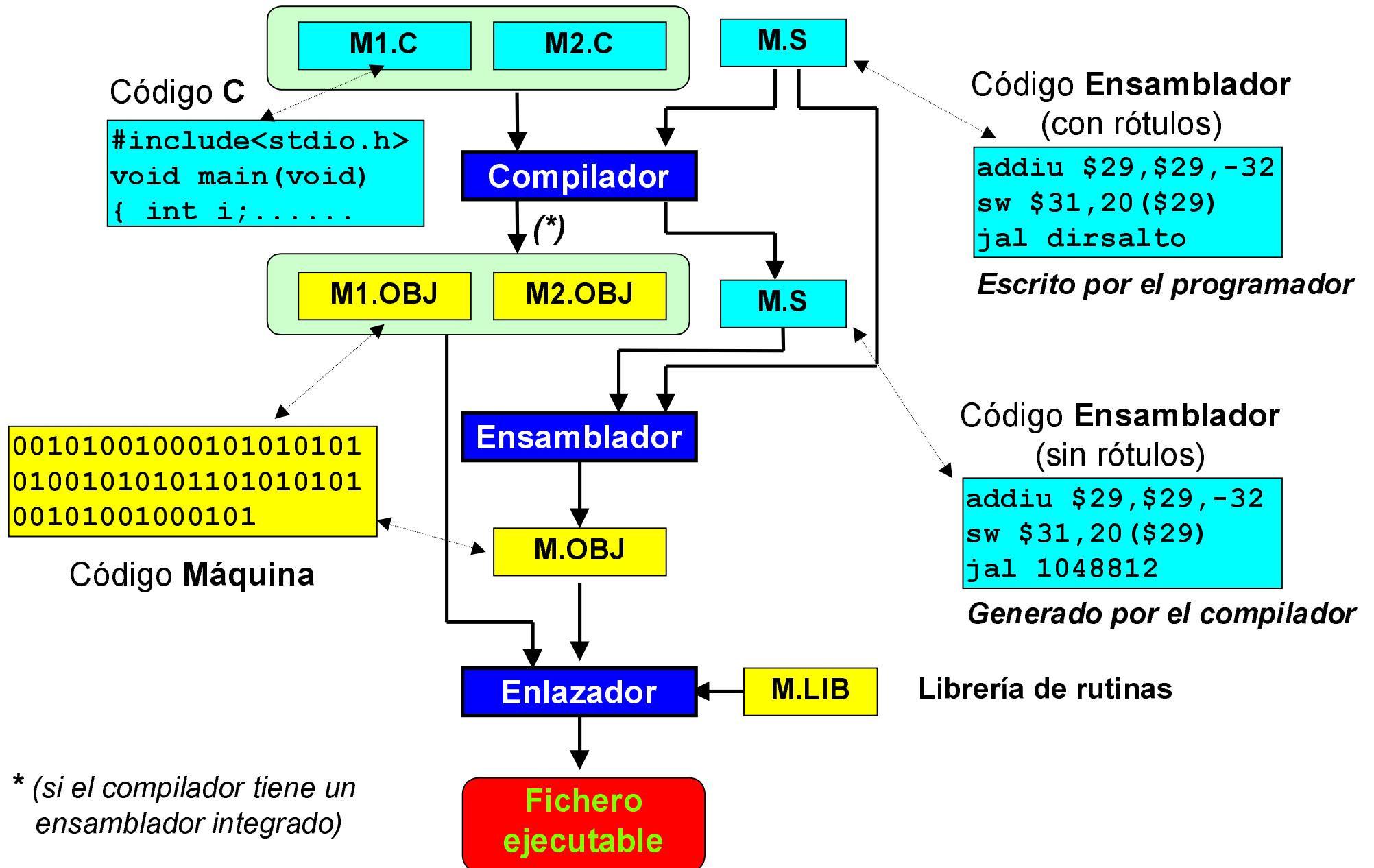
- Traduce un código en **lenguaje de alto nivel** en uno equivalente en **lenguaje ensamblador** (puede también producir lenguaje máquina directamente con la ayuda de un ensamblador integrado).
- Crean saltos y rótulos cuando no aparecen en el lenguaje de programación

ensamblador

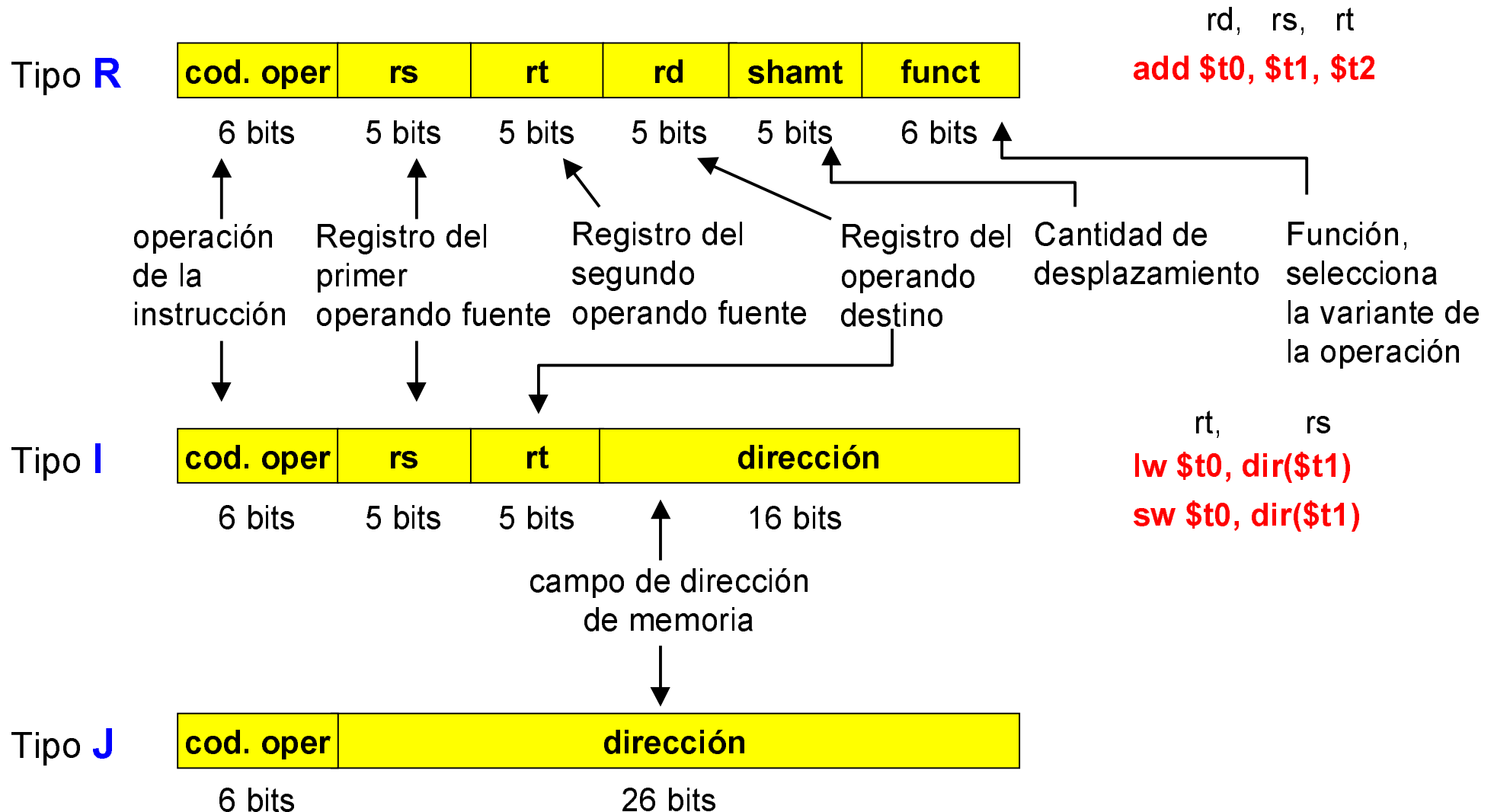
- Programa que traduce de **lenguaje ensamblador** a **lenguaje máquina**.
- Lee un único archivo fuente en ensamblador y produce un archivo objeto
- Un módulo puede contener referencias a subrutinas y datos definidos en otros módulos y bibliotecas. El código de un módulo no puede ejecutarse si contiene referencias sin resolver a rótulos de otros archivos objeto o bibliotecas.

enlazador

- Combina una colección de archivos objeto y de biblioteca en un archivo ejecutable, que el computador ejecuta

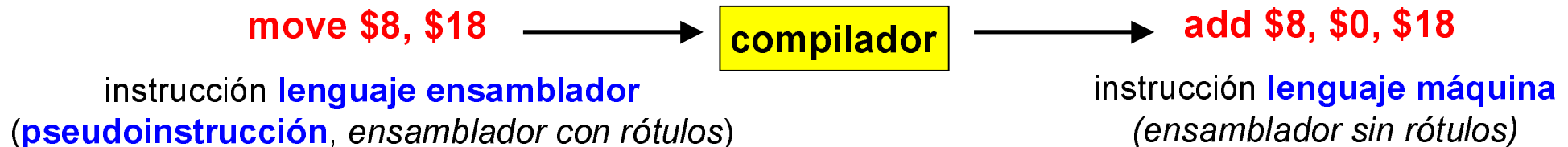


- Toda **instrucción** MIPS necesita **32 bits** (igual que una **palabra de datos** y un **registro**)



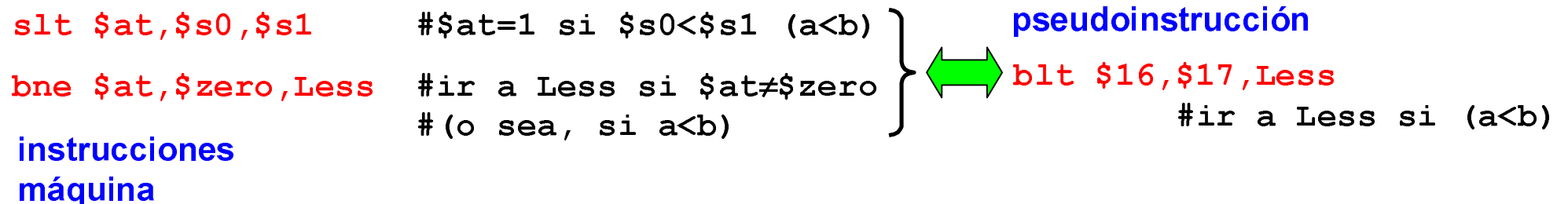
- El ensamblador **incrementa el nº de instrucciones disponibles** a los programadores en lenguaje ensamblador y a los compiladores, mediante las **pseudoinstrucciones**.
- Son instrucciones del lenguaje ensamblador que **no necesitan ser implementadas en hardware**.
- El ensamblador MIPS maneja pseudoinstrucciones **reservando \$1** para su uso como registro temporal

Ejemplo 1



Ejemplo 2

- **blt** = pseudoinstrucción "saltar sobre < que"
- Implementar blt en hardware lo complicaría, pues aumentaría la duración del ciclo de reloj, o bien esta instrucción tendría ciclos de reloj extra por instrucción (por simplicidad, queremos que las instrucciones se ejecuten en un nº de ciclos de reloj iguales). Dos instrucciones más rápidas serían más útiles



Los nombres que comienzan con un punto (**.data**, **.globl**, ..) son directivas del ensamblador: indican al ensamblador cómo traducir un programa, pero no producen instrucciones máquina

Directiva	Descripción
.align n	Alinea el siguiente dato sobre un límite de 2^n byte (.align 2, .align 0,...)
.ascii str	Almacena la cadena str en memoria, pero no la termina con el caracter nulo.
.asciiz str	Almacena la cadena str en memoria, y la termina con el caracter nulo.
.byte/half/word b1, ..., bn	Almacena los n valores de 8/16/32 bits en bytes/medias palabras/palabras consecutivas de memoria.
.double/float f1,...,fn	Almacena los n números de punto flotante de doble/simple precisión en posiciones consecutivas de memoria.
.data	Los elementos siguientes son almacenados en el segmento de datos.
.kdata	Los elementos siguientes son almacenados en el segmento de datos del núcleo.
.text	Los elementos siguientes son almacenados en el segmento de texto. Estos elementos sólo pueden ser instrucciones o palabras.
.ktext	Los elementos siguientes son almacenados en el segmento de texto del núcleo. Estos elementos sólo pueden ser instrucciones o palabras.
.space n	Asigna n bytes de espacio en el segmento actual (debe ser seg. datos en SPIM)
.extern sym size	Declara que el dato almacenado en <i>sym</i> ocupa <i>size</i> bytes y es un global. El ensamblador lo pone en parte del segmento de datos fácilmente accesible via \$gp
.globl sym	Declara <i>sym</i> como global: se puede referenciar desde otros archivos.

- Hay **32** registros (\$0, \$1, ..., \$31) de **32 bits** (palabra).
- Cuando un programa tiene más variables que registros, el compilador almacena las variables más utilizadas en los registros, y las restantes en memoria: **derramar registros** (**spilling**).

Nombre	Número	Uso
zero	0	Constante 0 (valor cableado)
at	1	Reservado para el ensamblador
v0, v1	2, 3	Evaluación de expresión y resultado de una función
a0, ..., a3	4, ..., 7	Argumentos a rutina (resto de argumentos, a pila)
t0, ..., t7	8, ..., 15	Temporales (no preservados a través de llamada, guardar invocador)
s0, ..., s7	16, ..., 23	Guardado temporalmente (preservado a través de llamada, guardar invocado)
t8, t9	24, 25	Temporales (no preservados a través de llamada, guardar invocador)
k0, k1	26, 27	Reservados para el núcleo del S.O.
gp	28	Puntero global, apunta a la mitad de un bloque de 64K en seg. datos estáticos
sp	29	Puntero de pila, apunta la primera posición libre en la pila
fp	30	Puntero de encuadre
ra	31	Dirección de retorno (usada por llamada de procedimiento)

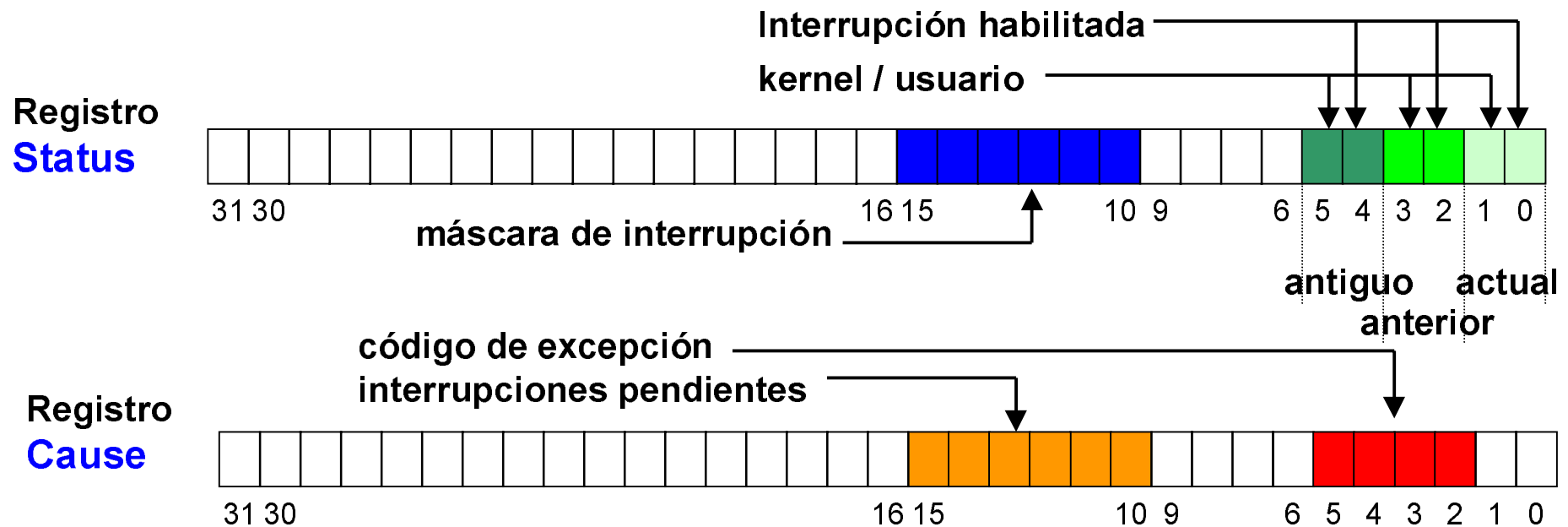
- MIPS es una arquitectura de carga/almacenamiento
- Sólo las instrucciones de carga y almacenamiento acceden a memoria, según los cálculos efectuados sobre valores en los registros.
- Alineación: una cantidad está alineada si su dirección de memoria es un múltiplo de su tamaño en bytes
Ejemplo: Una palabra de 32 bits debe almacenarse en direcciones múltiplo de 4.
- Casi todas las instrucciones de carga y almacenamiento operan sólo sobre datos alineados.
- Instrucciones para alinear datos no alineados: *lwl*, *lwr*, *swl* y *swr*.

Modos de direccionamiento

Formato	Cálculo de la dirección	Ejemplo
(registro)	Contenido del registro (cr)	<code>lw \$t0, (\$t2)</code>
valor	Valor inmediato (vin)	<code>lw \$t0, 0x10010008</code>
valor (registro)	$\text{vin} + \text{cr}$	<code>lw \$t0, 0x10010000 (\$t1)</code>
identificador	dirección del identificador (did)	<code>lw \$t0, array</code>
identificador +/- valor	$\text{did} \pm \text{vin}$	<code>lw \$t0, array+8</code>
identificador (registro)	$\text{did} + \text{cr}$	<code>lw \$t0, array (\$t1)</code>
identificador +/- valor (registro)	$\text{did} \pm \text{vi} + \text{cr}$	<code>lw \$t0, array+4 (\$t1)</code>

- Registros para tratar las **excepciones** e **interrupciones**.
- Son accedidos por las instrucciones **lwc0**, **mfc0**, **mtc0** y **swc0**
- SPIM sólo implementa los siguientes :

Nombre	Número	Uso
BadVAdress	8	Dirección de memoria en donde ocurre la excepción
Status	12	Máscara de interrupción y bits de habilitación
Cause	13	Tipo de excepción y bits de interrupción pendiente
EPC	14	Dirección de la instrucción que causó la excepción



Instrucciones para manejar excepciones

rfe	Vuelta desde excepción	Restaura el registro Status
syscall	Llamada al sistema	El registro \$v0 contiene el número de la llamada al sistema (ver la tabla de llamadas al sistema)
break	Produce excepción n	Provoca la excepción n. La exception 1 se reserva para el depurador (debugger)
nop	no operación	No hace nada

Códigos de excepciones

Número	Nombre	Uso
0	INT	Interrupción externa
4	ADDRL	Excepción error dirección (carga desde memoria o captura de instrucción)
5	ADDRS	Excepción error dirección (almacenamiento en memoria)
6	IBUS	Error de bus durante una captura de instrucción
7	DBUS	Error de bus durante una carga o almacenamiento
8	SYSCALL	Excepción llamada al sistema
9	BKPT	Excepción provocada por un punto de ruptura (breakpoint)
10	RI	Excepción de instrucción reservada
12	OVF	Excepción provocada por overflow aritmético

- Conjunto de servicios parecidos al SO a través de la instrucción **syscall**.
- \$v0: código de llamada al sistema
- \$a0, \$a1, o \$f12: argumentos
- Resultados tras syscall: \$v0 (o \$f0)

Valor que toma
\$v0 antes de
llamar a syscall

Argumentos que hay que
establecer antes de llamar a
syscall, para que ésta los trabaje

Si syscall devuelve alguna
cosa, lo pone en uno de
estos registros

Servicio	Código	Argumentos	Resultados
print_int	1	\$a0 = entero	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = cadena	
read_int	5		entero (en \$v0)
read_float	6		float (en \$f0)
read_double	7		float (en \$f0)
read_string	8	\$a0 = buffer, \$a1=longitud	
sbrk	9	\$a0 = cantidad	dirección (en \$v0)
exit	10		

Propósito

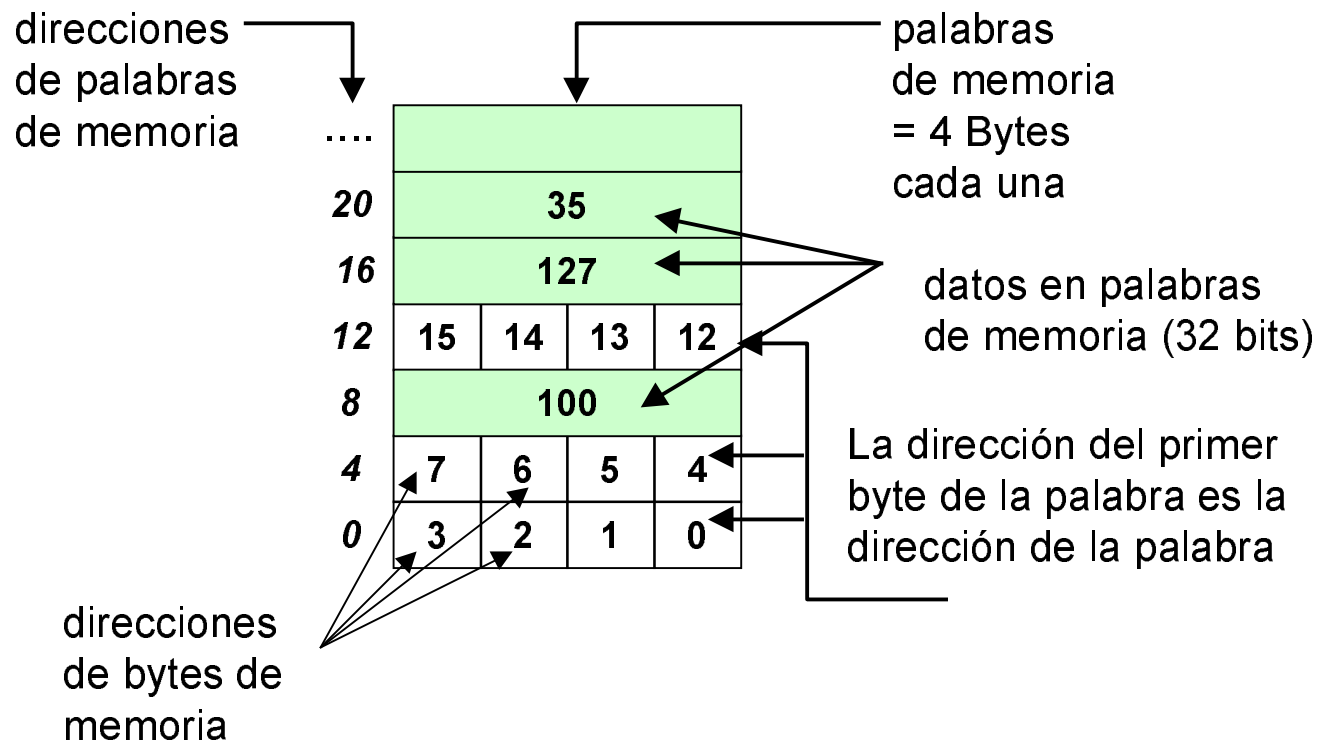
Imprimen en consola.
Hay que pasar el argumento, e imprimir.
No devuelven resultado

Lee de la consola (un número o una cadena)
No necesita argumentos (excepto cadena)
Espera a que introduzcamos núm. o cadena
Devuelve el valor leído a un registro

Añade páginas de mem. vir. al seg. datos dinám.

Termina la ejecución de un programa

- El ensamblador MIPS **direcciona bytes** individuales.
- Las **palabras** de memoria son de **32 bits** (4 Bytes).
- La memoria direccionable tiene un tamaño de 2^{30} palabras.
- Las instrucciones de carga y almacenamiento comunican datos entre memoria y registros.
La instrucción debe suministrar una dirección de memoria.
- Las direcciones comienzan en 0.
- El compilador asigna estructuras de datos (como los **arrays**) a posiciones de memoria, colocando la dirección adecuada de comienzo en las instrucciones de carga o almacenam.



Convenio “**little endian**”
(final menor peso)

3	2	1	0
---	---	---	---

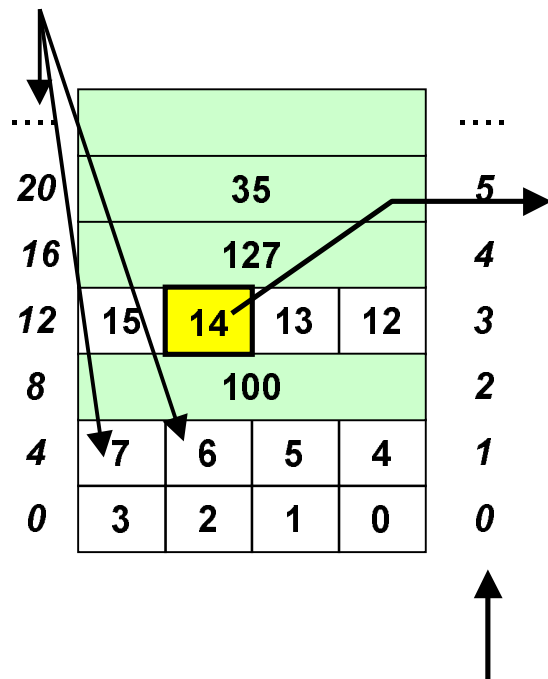
Convenio “**big endian**”
(final mayor peso)

0	1	2	3
---	---	---	---

- Una dirección de memoria es la dirección de 1 byte de memoria
- Si esa dir. es la del primer byte de una palabra (00), es como si fuera la dir. de la palabra.
- MIPS considera este direccionamiento para la **memoria virtual**

Estructura de bytes:

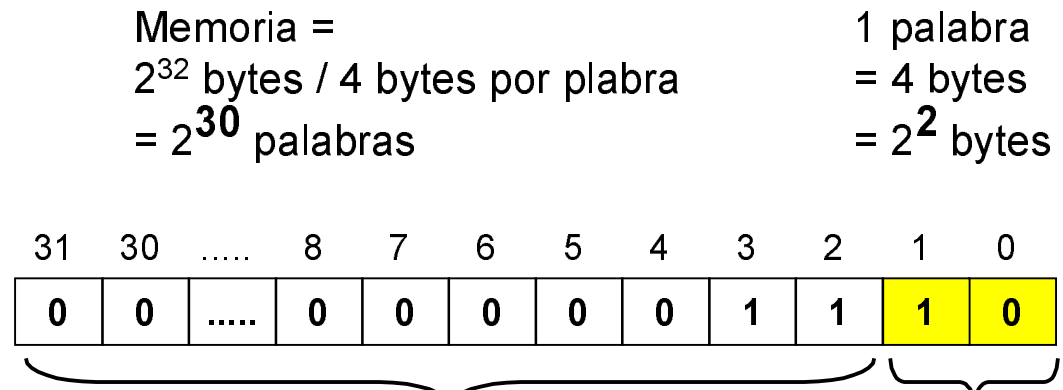
La memoria se direcciona como sucesión de bytes



Estructura de palabras:

La memoria se estructura como sucesión de palabras

Dirección de memoria



Campo

“Estructura de palabras”:

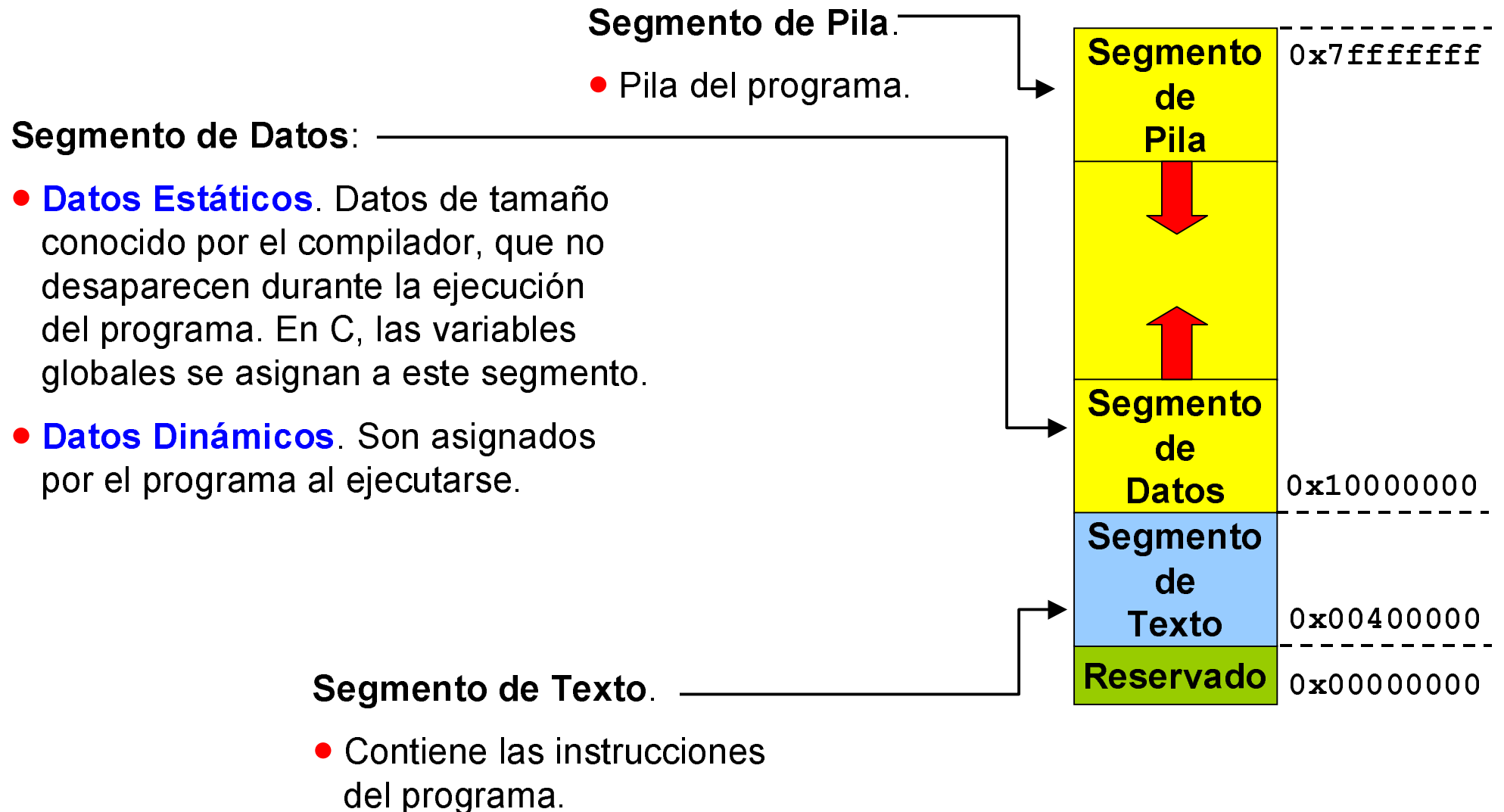
Direcciona la posición de la palabra dentro de la estructura de palabras

Campo

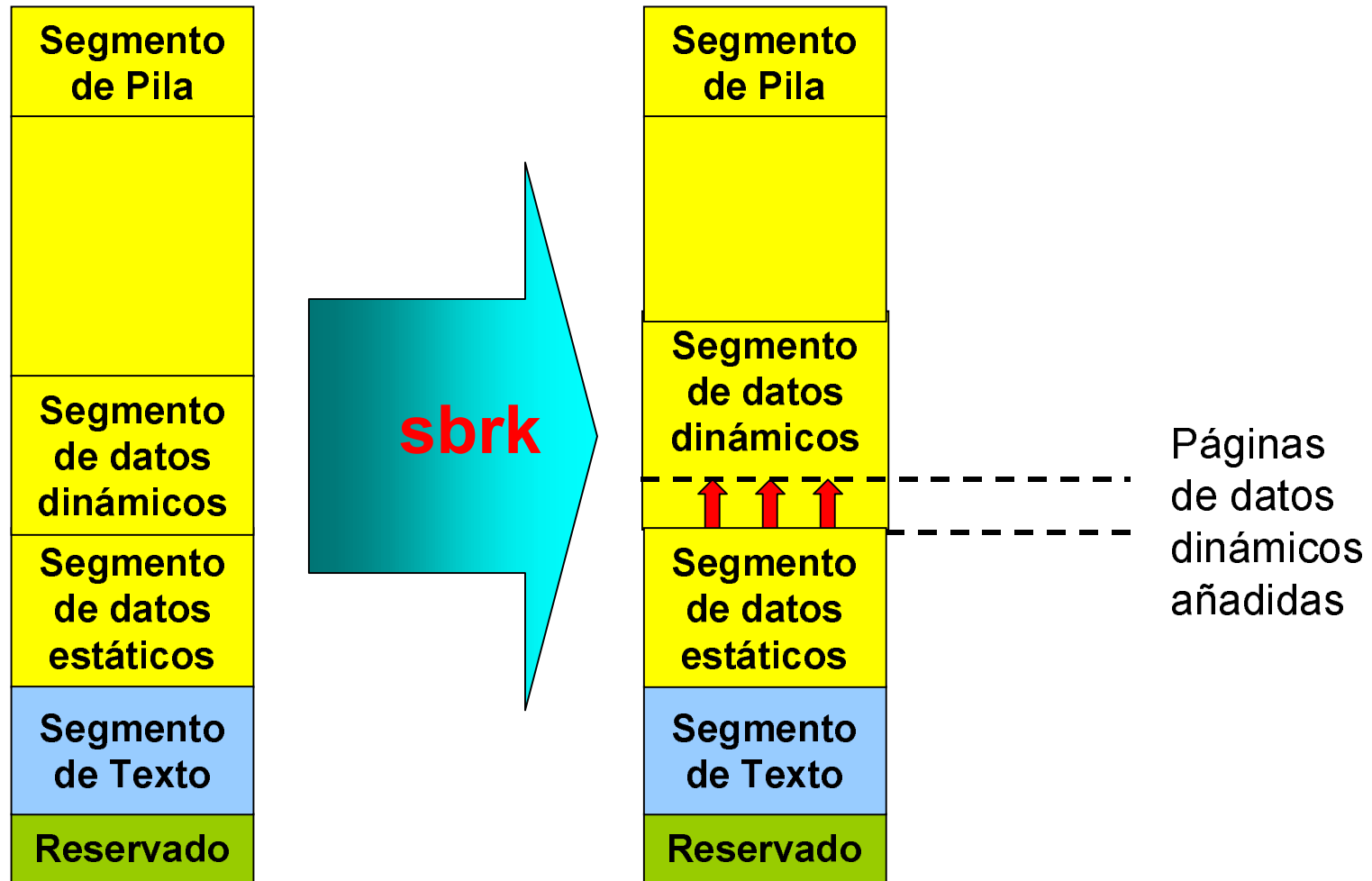
“Desplazamiento”:

Direcciona la posición del byte dentro de la palabra

- Los segmentos de pila y datos son expandibles dinámicamente.
- Están tan distantes como sea posible, y pueden crecer para utilizar el espacio completo de direcciones del programa.



En C, **malloc** encuentra y devuelve un nuevo bloque de memoria, expandiendo el área dinámica con la llamada del sistema **sbrk**, que hace que el S.O. añada más páginas al espacio de direcciones virtuales del programa, inmediatamente antes del segmento de datos dinámicos.



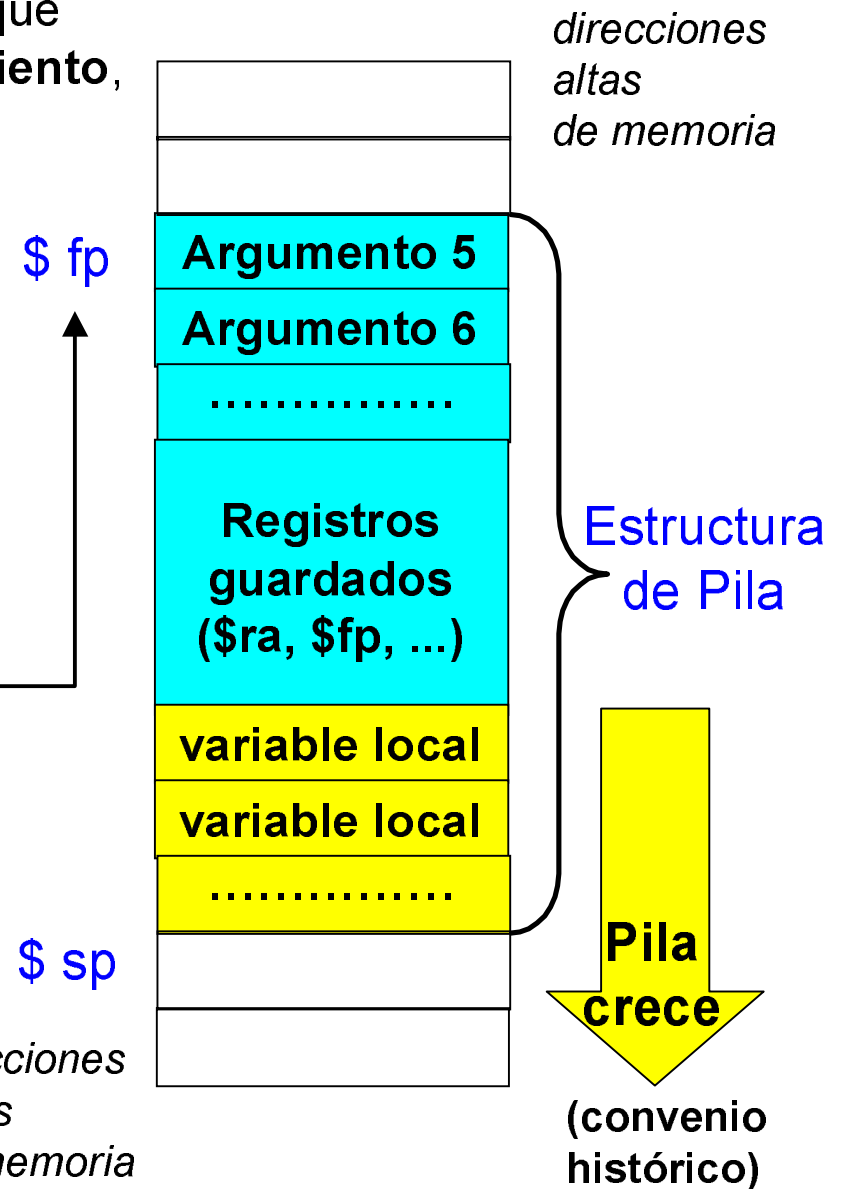
- El mantenimiento a una llamada se realiza con un bloque de memoria llamado **marco de llamada de procedimiento**, o **marco de pila**, que sigue un orden **FIFO**.
- Contiene valores pasados a un procedimiento como argumentos.
- Ahorra registros que el procedimiento invocado puede modificar, pero que el invocador no quiere cambiar.
- Da al procedimiento espacio para variables locales.
- Consta de la memoria entre dos punteros **\$fp** y **\$sp**:

Puntero de estructura, o de marco de pila, \$fp

- Apunta a la 1ª palabra de la estructura de pila. Se utiliza para acceder a valores en la estructura de pila

Puntero de pila, \$sp

- Apunta a la 1ª palabra de la pila después de la estructura, o sea, a la 1ª palabra libre de la pila



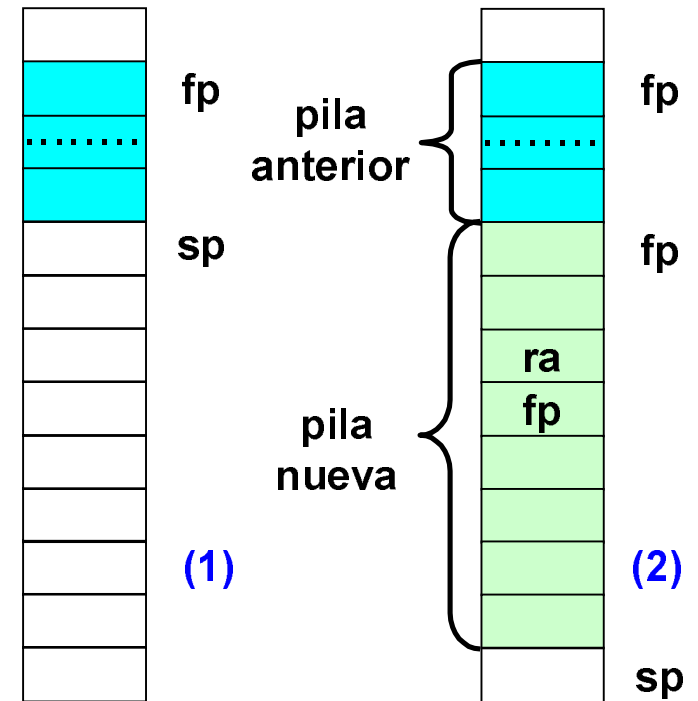
Creación de una nueva pila

- Creamos una pila de 32 bytes.
- Guardamos **\$ra** (si estamos en una subrutina y vamos a hacer una llamada a subrutina) y **\$fp** (para identificar la anterior pila) en la nueva pila
- Si estamos en una subrutina, y los argumentos de llamada no caben en \$a0...\$a3, los ponemos en esta pila

(1) antes

```
subu    $sp, $sp, 32
sw      $ra, 20($sp)
sw      $fp, 16($sp)
addu    $fp, $sp, 32
```

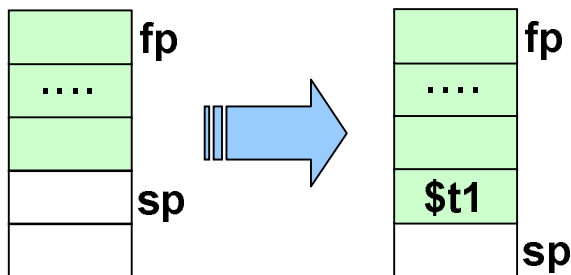
(2) después



Introducir dato en pila

- Almacenar dato en **0(\$sp)**
- Restar a **\$sp** el tamaño (bytes) del dato

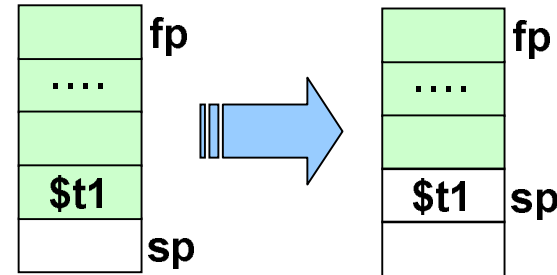
```
sw      $t1, 0($sp)
subu    $sp, $sp, 4
```



Sacar dato de pila

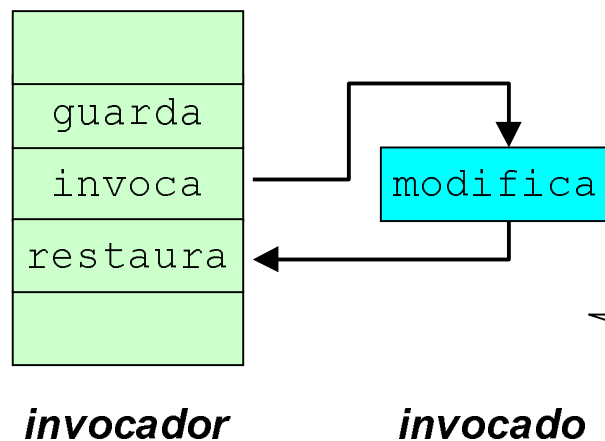
- Sumar a **\$sp** el tamaño (bytes) del dato
- Leer dato de **0(\$sp)**

```
addu    $sp, $sp, 4
lw      $t1, 0($sp)
```

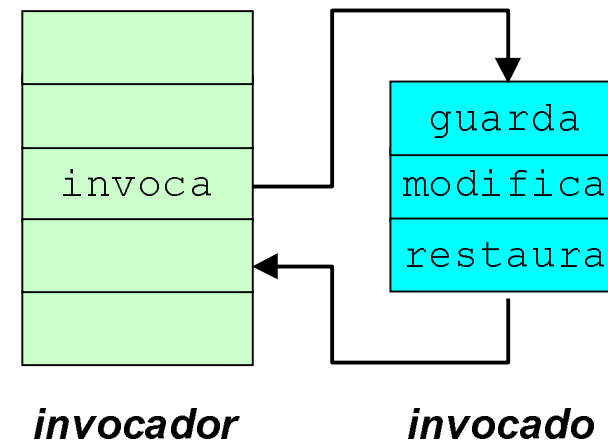


- Gobierna el uso de registros cuando los procedimientos de un programa se compilan separadamente.
- Son convenios software, no hardware
- Un compilador debe conocer los registros que puede usar y los reservados a otras rutinas.
- Hay dos convenios estándares (se aplica uno u otro en una llamada a procedimiento):

Guardar Invocador (*caller save*)



Guardar Invocado (*callee save*)



- » El invocador guarda y restaura los registros que se deban conservar a través de la llamada (específicamente **t0...t9**)
- » El invocado puede modificar cualquier registro sin preocuparse.

- » El invocado guarda y restaura los registros que pueda usar (específicamente **s0...s9**)
- » El invocador usa los registros sin preocuparse de restaurarlos después de una llamada.

• Antes de que el invocador llama al invocado

- » **Pasar argumentos (1)**. Invocador pone los 4 primeros argumentos en $\$a0...\$a3$, y el resto en la pila del invocador, cuyo marco $\$fp$ está en (2)
- » **Guardar en pila los registros de “guardar-invocador” (3)**. Si usamos “guardar invocador”, se guardan en pila, aquellos de $\$a0-\$a3$ y $\$t0-\$t9$ que el invocador necesite inalterados después de la llamada, pues $\$t0...\$t9$ los usa el invocado.
- » **Ejecutar jal**: bifurca al invocado y guarda la dirección de vuelta en $\$ra$.

• Al inicio de la ejecución del invocado

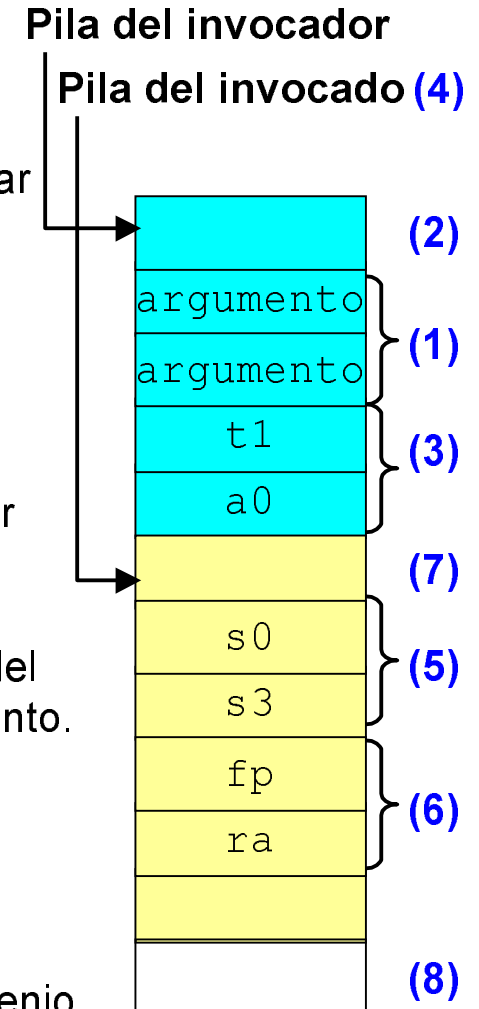
- » **Crear pila del invocado (4)**: $\$sp = \$sp - (\text{tamaño en bytes})$. (mín, 32 B)
- » **Guardar en pila los registros de “guardar-invocado” (5)**. Si usamos “guardar invocado”, se guardan en pila aquellos de $\$s0-\$s7$ que el invocador necesite inalterados a la vuelta, pues $\$s0-\$s7$ los usa el invocado.
- » **Guardar $\$fp$ y $\$ra$ (6)**. $\$fp$ se guarda para poder, a la vuelta, identificar la pila del invocador, y $\$ra$ se guarda si el invocado hace una nueva llamada a procedimiento.
- » **Actualizar el puntero de marco**. Hacer que $\$fp$ apunte al nuevo marco del invocado: $\$fp$ (7) = tamaño de pila del invocado + $\$sp$ (8).

• Al final del invocado

- » Si el invocado devuelve un valor, se coloca éste en $\$v0$.
- » Restaurar los registros de “guardar-invocado” (5), si estamos usando este convenio.
- » Restaurar $\$fp$ y $\$ra$ del invocador (6). Ahora $\$fp$ está en (2)
- » Destruir pila del invocado: $\$sp = \$sp + \text{tamaño de pila}$. Ahora $\$sp$ está en (7)
- » Retornar saltando (instrucción j) a la dirección contenida en el registro $\$ra$.

• Al volver al invocador

- » Restaurar los registros de “guardar invocador” (3) que se guardaron antes de llamar al invocado.



- SPIM es un simulador que ejecuta programas ensamblador del MIPS R2000
- Contiene un depurador y proporciona algunos servicios como el sistema operativo
- SPIM es un **simulador virtual**

Para hacer frente a la difícil programación de la arquitectura MIPS, SPIM hace:

- » Oculta instrucciones (saltos y cargas) retardadas (segmentación), reorganizando las instrucciones para ajustar los huecos del retardo.
- » Amplia el conjunto de instrucciones del hardware real mediante pseudoinstrucciones, simulando éstas mediante cortas secuencias de instrucciones reales.

The image shows the PCSpim MIPS simulator interface. The main window is titled "PCSpim" and contains several panes:

- Messages:** Displays assembly instructions and their addresses. An arrow points from the label "Mensajes del simulador" to this pane.
- Text Segment:** Displays the text segment of the program. An arrow points from the label "Segmento de texto" to this pane.
- Data Segment:** Displays the data segment of the program. An arrow points from the label "Segmento de datos" to this pane.
- Registers:** Displays the current values of the MIPS registers. An arrow points from the label "Registros" to this pane.
- Console:** A separate window titled "Console" showing the output of the program. An arrow points from the label "Consola" to this window.

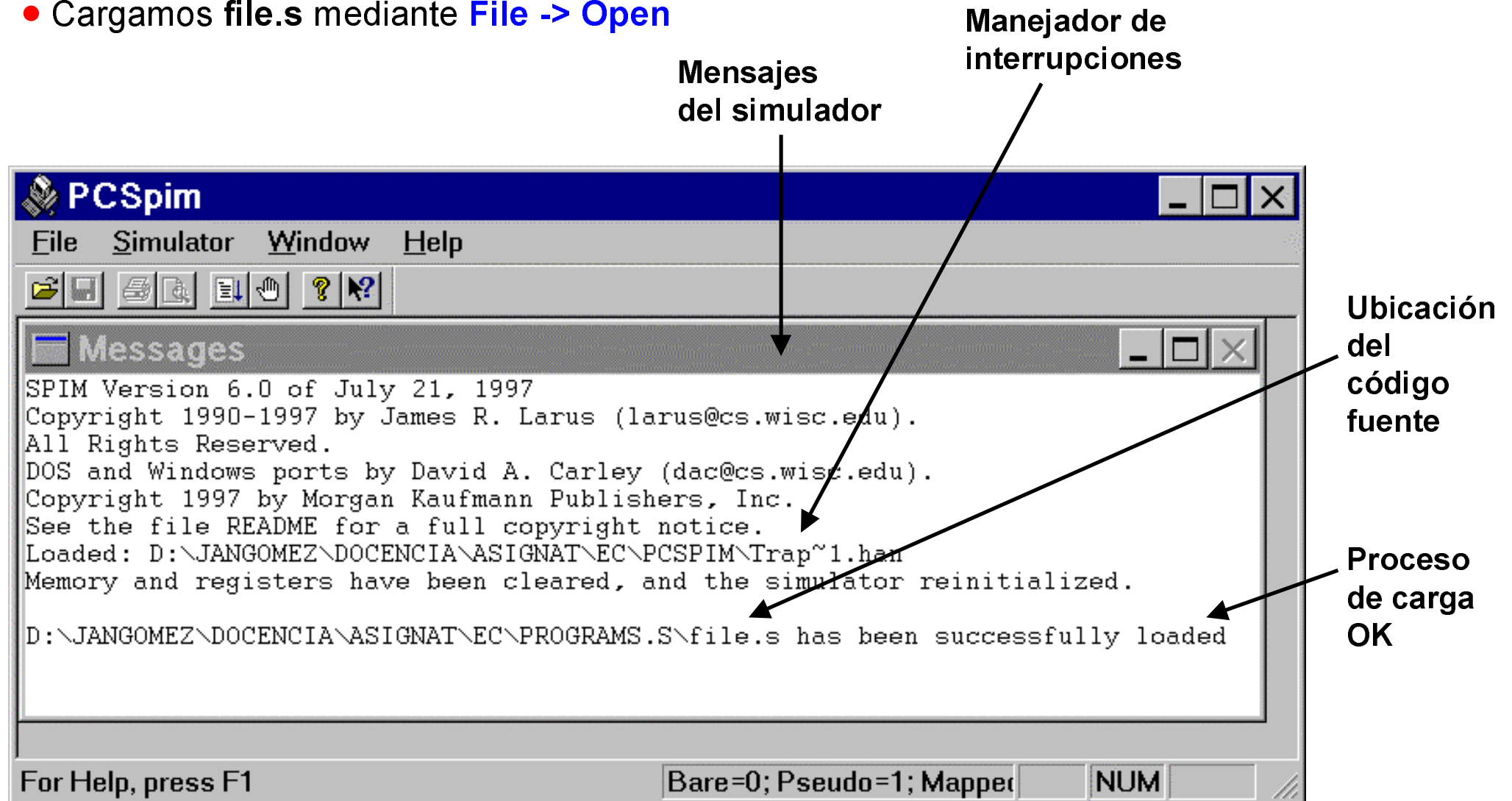
At the bottom of the PCSpim window, there is a status bar with the text "For Help, press F1" and a set of buttons labeled "Bare=0; Pseudo=1; Mapper" and "NUM".

Ensamblador MIPS

Opciones de SPIM

Bare machine <i>(máquina reducida)</i>	On	Simula un MIPS reducido en instrucciones, sin pseudo-instrucciones ni otros modos de direccionamiento proporcionados por el ensamblador.
	Off	SPIM simula la máquina virtual
Mapped I/O <i>(E/S con correspondencia directa a memoria)</i>	On	Habilita la E/S con correspondencia directa a memoria. Los programas que usen <i>syscall</i> para leer desde el terminal, no pueden usar E/S de correspondencia directa con memoria.
	Off	Deshabilita la E/S con correspondencia directa a memoria
Allow pseudo-instructions <i>(permite pseudo-inst.)</i>	On	Permite pseudoinstrucciones, y se ve cómo las traduce instrucc. máquina.
	Off	No permite pseudo-instrucciones, sólo instrucciones máquina.
Quiet	On	No imprime ningún mensaje en las excepciones
	Off	Imprime un mensaje cuando se presenta una excepción
Load trap file	On	Carga manipulador de excepciones y código de arranque (archivo trap.han). Durante una excepción, SPIM salta a 0x80000080, donde hay código para tratarla. Trap.han contiene el código de arranque que invoca la rutina main. Sin la rutina de arranque, SPIM comienza la ejecución en el rótulo __start

- Editamos el código ensamblador **file.s** en cualquier procesador de texto **ASCII**.
- Abrimos **PCSPIM**
- Cargamos **file.s** mediante **File -> Open**



Dirección de la instrucción máquina

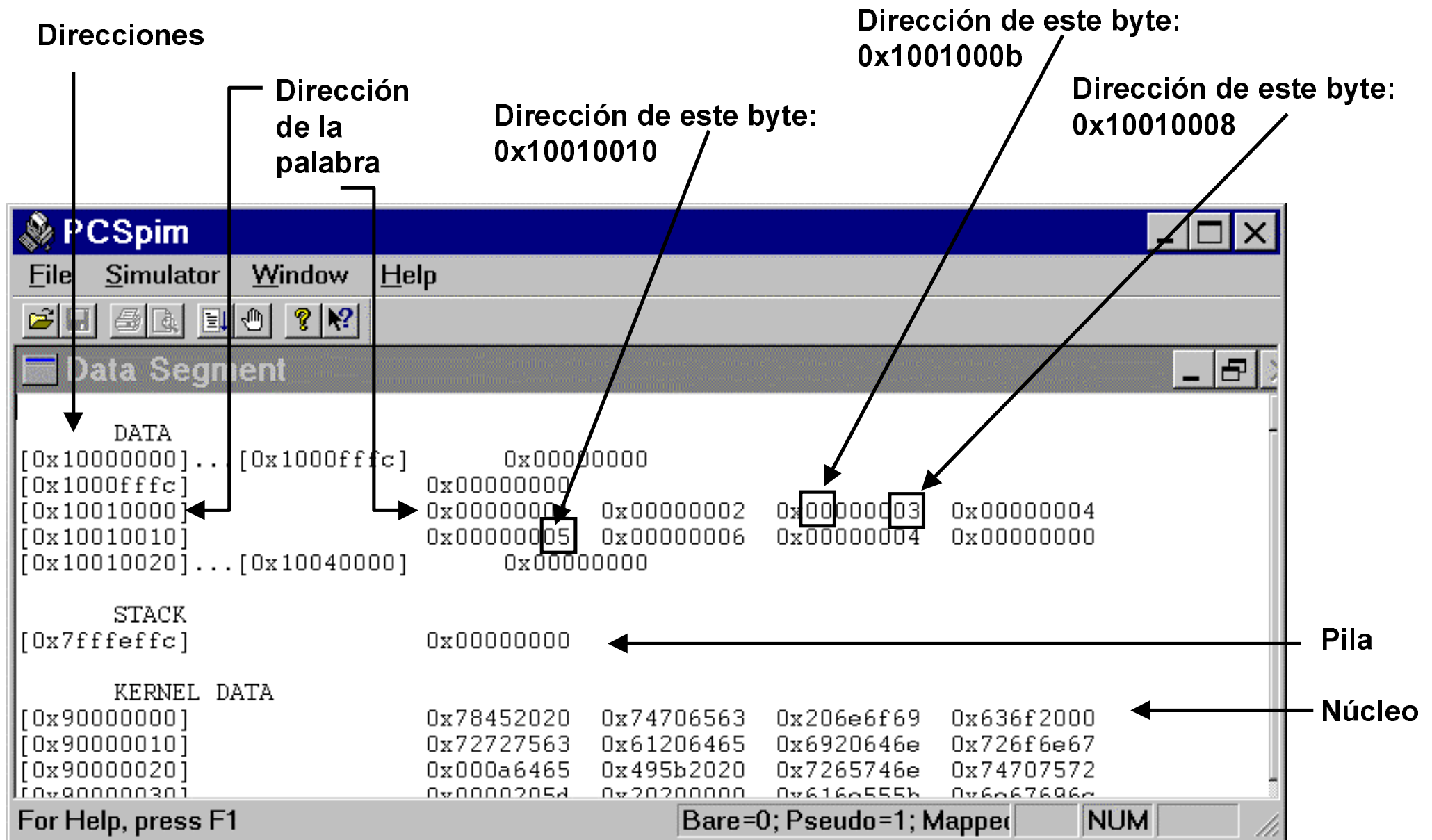
Código máquina (ensamblador sin rótulos,
sin pseudoinstrucciones, ...)

```

[0x00400000] 0x8fa40000 lw $4, 0($29) ; 102: lw $a0, 0($sp)
[0x00400004] 0x27a50004 addiu $5, $29, 4 ; 103: addiu $a1, $sp, 4 # ar
[0x00400008] 0x24a50004 addiu $6, $5, 4 ; 104: addiu $a2, $a1, 4 # er
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 105: sll $v0, $a0, 2
[0x00400010] 0x00c43021 addu $6, $6, $2 ; 106: addu $a2, $a2, $v0
[0x00400014] 0x0c100008 jal 0x00400020 [main] ; 107: jal main
[0x00400018] 0x3402000a ori $2, $0, 10 ; 108: li $v0, 10
[0x0040001c] 0x0000000c syscall ; 109: syscall
[0x00400020] 0x34090008 ori $9, $0, 8 ; 9: li $t1, 8
[0x00400024] 0x3c011001 lui $1, 4097 ; 10: li $t2, 0x10010008
[0x00400028] 0x342a0008 ori $10, $1, 8 ;
[0x0040002c] 0x8d480000 lw $8, 0($10) ; 12: lw $t0, ($t2)
[0x00400030] 0x0c100024 jal 0x00400090 [print] ; 13: jal print
[0x00400034] 0x3c011001 lui $1, 4097 ; 15: lw $t0, 0x10010008
[0x00400038] 0x8c280008 lw $8, 8($1) ;
[0x0040003c] 0x0c100024 jal 0x00400090 [print] ; 16: jal print
[0x00400040] 0x3c011001 lui $1, 4097 ; 18: lw $t0, 0x10010000($t1)
[0x00400044] 0x00290821 addu $1, $1, $9 ;
[0x00400048] 0x8c280000 lw $8, 0($1) ;
[0x0040004c] 0x0c100024 jal 0x00400090 [print] ; 19: jal print
[0x00400050] 0x3c011001 lui $1, 4097 ; 21: lw $t0, array
[0x00400054] 0x8c280000 lw $8, 0($1) ;
[0x00400058] 0x0c100024 jal 0x00400090 [print] ; 22: jal print
[0x0040005c] 0x3c011001 lui $1, 4097 ; 24: lw $t0, array+8
[0x00400060] 0x8c280008 lw $8, 8($1) ;
  
```

For Help, press F1 Bare=0; Pseudo=1; Mapper NUM

Código fuente
(ensamblador
con rótulos,
escrito por el
usuario)



PCSpim

File Simulator Window Help

Registers

PC = 00000000 EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
 Status = 00000000 HI = 00000000 LO = 00000000

General Registers

R0 (r0) = 0x00000000	R8 (t0) = 0x00000004	R16 (s0) = 0x00000000	R24 (t8) = 0x00000000
R1 (at) = 0x10010000	R9 (t1) = 0x00000008	R17 (s1) = 0x00000000	R25 (t9) = 0x00000000
R2 (v0) = 0x0000000a	R10 (t2) = 0x10010008	R18 (s2) = 0x00000000	R26 (k0) = 0x00000000
R3 (v1) = 0x00000000	R11 (t3) = 0x00000000	R19 (s3) = 0x00000000	R27 (k1) = 0x00000000
R4 (a0) = 0x00000004	R12 (t4) = 0x00000000	R20 (s4) = 0x00000000	R28 (gp) = 0x10008000
R5 (a1) = 0x7ffff000	R13 (t5) = 0x00000000	R21 (s5) = 0x00000000	R29 (sp) = 0x7ffff000
R6 (a2) = 0x7ffff004	R14 (t6) = 0x00000000	R22 (s6) = 0x00000000	R30 (s8) = 0x00000000
R7 (a3) = 0x00000000	R15 (t7) = 0x00000000	R23 (s7) = 0x00000000	R31 (ra) = 0x00400000

Double Floating Point Registers

FP0 = 0x00000000, 0x00000000	FP8 = 0x00000000, 0x00000000	FP16 = 0x00000000, 0x00000000	FP24 = 0x00000000, 0x00000000
FP2 = 0x00000000, 0x00000000	FP10 = 0x00000000, 0x00000000	FP18 = 0x00000000, 0x00000000	FP26 = 0x00000000, 0x00000000
FP4 = 0x00000000, 0x00000000	FP12 = 0x00000000, 0x00000000	FP20 = 0x00000000, 0x00000000	FP28 = 0x00000000, 0x00000000
FP6 = 0x00000000, 0x00000000	FP14 = 0x00000000, 0x00000000	FP22 = 0x00000000, 0x00000000	FP30 = 0x00000000, 0x00000000

Single Floating Point Registers

FP0 = 0x00000000	FP8 = 0x00000000	FP16 = 0x00000000	FP24 = 0x00000000
FP2 = 0x00000000	FP10 = 0x00000000	FP18 = 0x00000000	FP26 = 0x00000000
FP4 = 0x00000000	FP12 = 0x00000000	FP20 = 0x00000000	FP28 = 0x00000000
FP6 = 0x00000000	FP14 = 0x00000000	FP22 = 0x00000000	FP30 = 0x00000000

For Help, press F1

Bare=0; Pseudo=1; Mapper NUM

Menú del simulador:

- Clear Registers Pone todos los registros con el valor cero (0x00000000).
- Reinitialize Borra el contenido de los registros y de la memoria, e inicia el simulador.
- Reload Reinicializa el simulador, y carga el código ensamblador actual.
- Go Ejecuta el código actual.
- Break/Continue Durante la ejecución, la para. Si está parada, la continúa.
- Single Step Ejecuta instrucción a instrucción.
- Multiple Step... Ejecuta un número de instrucciones especificado por el usuario.
- Breakpoints... Abre una ventana para especificar las posiciones de los puntos de ruptura.
- Set Value... Abre una ventana para establecer valores de memoria o de registros.
- Symbol table Muestra la tabla de símbolos en la ventana de mensajes.
- Settings Ventana para seleccionar las opciones de SPIM.

trap.handler

- Contiene el código de arranque que el SO usa para arrancar el programa ensamblador del usuario.
- Proceso de arranque: Invoca la rutina **main**, sin argumentos.
- A la vuelta, se finaliza la ejecución

```
.text
.globl __start
__start:
    lw $a0, 0($sp)      # argc
    addiu $a1, $sp, 4   # argv
    addiu $a2, $a1, 4   # envp
    sll $v0, $a0, 2
    addu $a2, $a2, $v0
    jal main
    {
        li $v0 10
        syscall
    }
```

```
.text
.globl main
main:
    .....
    jr $ra #Retorna al cód.arranque
```

Código
usuario

```
.text
..... # primera intr.
.....
li $v0,10 # instr. para
syscall # terminar
```

- **Con trap.handler activado**, la ejecución comienza en la etiqueta `__start` (dir. 0x00400000). El código del usuario no tiene que indicar dicha etiqueta (sí main).
- El programa de usuario finaliza con jr \$ra, ejecutandose las dos últimas instrucciones del código de arranque que finalizan la ejecución y devuelven el control al S.O.
- `main` (invocado) debe guardar \$ra y \$fp del código de arranque, y debe recuperarlos antes de terminar.
- **Con trap.handler desactivado**, para arrancar hay que especificar como comienzo de la ejecución la dirección de la 1ª instrucción del programa (0x00400000).
- Hay que indicar el final del programa con instrucciones de terminación de la ejecución.

