

Universidad de Murcia  
Facultad de Informática

---

---

TÍTULO DE GRADO EN  
INGENIERÍA INFORMÁTICA

# Estructura y Tecnología de Computadores

Tema 3: Lenguaje ensamblador

Apuntes

CURSO 2010 / 11

---

---

VERSIÓN 2.0

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores



## Índice general

3.1. Introducción . . . . .	3
3.2. Repertorios de instrucciones . . . . .	3
3.3. Operandos del computador . . . . .	4
3.3.1. Los registros en MIPS . . . . .	5
3.4. El ISA de MIPS . . . . .	7
3.4.1. Instrucciones aritmético-lógicas . . . . .	8
3.4.2. Instrucciones de acceso a memoria . . . . .	12
3.4.3. Instrucciones de salto . . . . .	15
3.4.4. Instrucciones para soportar procedimientos . . . . .	18
3.4.5. Instrucciones de coma flotante . . . . .	23
3.5. Manejo de interrupciones y excepciones . . . . .	28
3.6. Llamadas al sistema operativo . . . . .	29
3.7. Codificación de las instrucciones en MIPS . . . . .	30
3.7.1. Formato de instrucción R . . . . .	31
3.7.2. Formato de instrucción I . . . . .	32
3.7.3. Formato de instrucción J . . . . .	34
3.7.4. Pseudoinstrucciones . . . . .	34
3.8. Alternativas al ISA de MIPS . . . . .	36
<b>A3. Apéndices</b>	<b>39</b>
A3.1. Manipulación de datos a nivel de bits . . . . .	39
A3.1.1. Lectura de campos de bits . . . . .	39
A3.1.2. Escritura de campos de bits . . . . .	40
A3.2. Uso de la memoria en MIPS . . . . .	41
A3.2.1. Modos de direccionamiento a memoria del ensamblador de MIPS . . . . .	41
A3.2.2. Organización de la memoria en MIPS . . . . .	41
A3.2.3. Datos y punteros a datos . . . . .	44
A3.2.4. Ejercicios . . . . .	48
<b>B3. Boletines de prácticas</b>	<b>53</b>
B3.1. Uso del simulador MARS . . . . .	53
B3.1.1. Objetivos . . . . .	53
B3.1.2. Prerequisitos . . . . .	53
B3.1.3. Plan de trabajo . . . . .	53
B3.1.4. El simulador MARS . . . . .	54
B3.1.5. Anatomía de un programa en ensamblador . . . . .	56
B3.1.6. Ejercicios . . . . .	58
B3.2. Convenciones de programación en MIPS . . . . .	58
B3.2.1. Objetivos . . . . .	58
B3.2.2. Prerequisitos . . . . .	58
B3.2.3. Plan de trabajo . . . . .	58
B3.2.4. Ejemplos de paso de parámetros . . . . .	58
B3.2.5. Ejemplo de procedimiento recursivo . . . . .	62
B3.2.6. Ejercicios . . . . .	62
B3.3. Proyecto de programación en MIPS . . . . .	64

B3.3.1. Objetivos . . . . .	64
B3.3.2. Pong . . . . .	65
B3.3.3. Descripción del programa inicial . . . . .	65
B3.3.4. Modificaciones propuestas al programa . . . . .	72
B3.3.5. Criterios de evaluación . . . . .	77
B3.3.6. Requisitos de entrega . . . . .	77

### 3.1. Introducción

Una de las propiedades más interesantes de los computadores es su capacidad para realizar funciones muy variadas con el mismo hardware, gracias a que son programables.

Un *programa* es el conjunto ordenado de instrucciones que dirigen el funcionamiento del computador. Una *instrucción* es un símbolo o conjunto de símbolos que representa una orden concreta para el computador. Dependiendo del nivel de abstracción del lenguaje, una instrucción puede representar una operación simple como “sumar dos números” o más compleja como “mostrar un número por pantalla”.

Los programas se pueden expresar mediante una gran variedad de *lenguajes*. En esta asignatura nos ocuparemos de los lenguajes llamados de “bajo nivel”, en oposición a los lenguajes de “alto nivel”. Esta clasificación se refiere al nivel de *abstracción* del lenguaje respecto al funcionamiento del computador.

Los lenguajes de bajo nivel se abstraen muy poco de los detalles del computador concreto que ejecutará el programa, ofreciendo instrucciones simples de ejecutar, específicas y dependientes de la máquina. Por el contrario, los lenguajes de alto nivel se abstraen más del funcionamiento de la máquina que ejecutará los programas y ofrecen instrucciones y construcciones más complicadas de ejecutar, más genéricas y más independientes de la máquina concreta. Gracias a esta abstracción, los lenguajes de alto nivel permiten al programador expresar algoritmos de forma más natural y más directa, por lo que son los utilizados normalmente.

Aunque existen miles de lenguajes de programación diferentes, la mayoría permite expresar cualquier programa<sup>1</sup>. En particular, no se debe confundir el que un lenguaje sea de más alto o más bajo nivel con la capacidad del lenguaje de expresar un programa concreto.

Para que un programa escrito en un lenguaje determinado pueda ser ejecutado, debe primero traducirse a un lenguaje de muy bajo nivel que depende de la máquina que deseamos utilizar para ejecutar el programa. A este lenguaje se le llama normalmente “código máquina”. Esta traducción se realiza casi siempre mecánicamente usando programas llamados “compiladores” y “ensambladores”, con los que el alumno ya estará familiarizado.

Gracias a asignaturas cursadas antes que *Estructura y Tecnología de Computadores*, el alumno debe estar familiarizado con, al menos, algún lenguaje de programación de alto nivel y al menos un lenguaje ensamblador. El objetivo de este tema es estudiar el lenguaje ensamblador del MIPS y su traducción a código máquina con suficiente detalle para permitir la comprensión de cómo se implementa un circuito (procesador) capaz de ejecutar el código máquina resultante. También se pretende que el alumno sea capaz de realizar programas de complejidad moderada directamente en ensamblador y que sepa cómo se traducen a ensamblador algunas de las construcciones más usuales en lenguajes de alto nivel.

### 3.2. Repertorios de instrucciones

El repertorio de instrucciones (en inglés, *ISA* o *Instruction Set Architecture*) de un computador es el conjunto de instrucciones ejecutables por dicho computador.

El ISA de un computador también determina características de la máquina como el número y tipo de registros, modos de direccionamiento, manejo de interrupciones y excepciones, y manejo de la entrada/salida.

En general, los repertorios de instrucciones de los distintos modelos de computador son bastante similares. Esto se debe a que su *hardware* se diseña usando unos principios fundamentales similares y, por lo tanto, existe un repertorio básico de funciones que todos los computadores deben, de una u otra forma, proporcionar.

Asimismo, el mismo ISA puede ser utilizado por varios computadores diferentes, incluso computadores totalmente diferentes y diseñados por fabricantes diferentes. Por ejemplo, el ISA IA-32 (también conocido como x86) es utilizado por procesadores fabricados por Intel y AMD, entre otros. El propio ISA MIPS que estudiaremos en este tema se utiliza en los procesadores de productos tan diferentes entre sí como estaciones de trabajo (algunos modelos de *Silicon Graphics*, p.e.), supercomputadores (*NEC*, *ORIGIN 2000*), ordenadores

<sup>1</sup>Hay lenguajes de propósito específico que solo permiten expresar algunos programas que cumplen ciertas condiciones.

de bolsillo (*palmtops*), móviles, impresoras (modelos láser de Hewlett-Packard), cámaras digitales, videoconsolas (*Nintendo 64* o *Playstation 2* de *Sony*, entre otras), *routers* o incluso robots (como el *AIBO* de *Sony*).

En principio, es fácil elegir un repertorio de instrucciones que sea capaz de representar cualquier programa ejecutable por un computador. Sin embargo, la elección de un conjunto de instrucciones adecuado es importante para optimizar el rendimiento y el coste del sistema final. Los objetivos principales a seguir cuando se diseña un repertorio de instrucciones (ISA) son los siguientes:

1. Facilitar la construcción del hardware: a no ser que la complejidad esté debidamente justificada, debe diseñarse un repertorio de instrucciones que permita que el diseño del procesador sea lo más sencillo posible.
2. De igual modo, debe permitir que el diseño del compilador encargado de pasar de lenguaje de alto nivel a lenguaje ensamblador sea sencillo, en la medida de lo posible.
3. Ha de maximizar el rendimiento obtenido, entendido éste, en un sentido amplio, como la cantidad de procesamiento efectivo útil por unidad de tiempo.
4. Debe minimizarse el coste de implementación del procesador capaz de ejecutar dicho repertorio.

Para que se pueda programar de forma efectiva un computador, además de conocer el ISA del mismo es necesario conocer y seguir una serie de convenciones sobre el uso de registros, la llamada a procedimientos o el interfaz con el sistema operativo y con la entrada/salida. Estas convenciones de programación junto con el ISA del computador determinan el interfaz de programación binaria (en inglés *ABI*, o *Application Binary Interface*) particular de cada plataforma.

En el resto de este tema se explicará el repertorio de instrucciones y las convenciones de programación de un computador MIPS, como ejemplo de un ABI típico. MIPS es un ISA usado en varios sistemas reales y representativo de las arquitecturas tipo RISC (*Reduced Instruction Set Computer*). Como tal se caracteriza por ofrecer instrucciones sencillas, formatos de instrucción regulares, muchos registros de propósito general y modos de direccionamiento sencillos.

### 3.3. Operandos del computador

Los lenguajes de alto nivel utilizan *variables* y *constantes* de distintos tipos (enteros, reales, caracteres, *arrays* o tablas, etc) como *operandos* de las instrucciones. Cuanto mayor sea el nivel de abstracción del lenguaje, menos tendrá que preocuparse el programador de detalles tales como dónde se almacenan esos operandos, cuánto ocupan o qué pasos en concreto es necesario dar para realizar una operación concreta con esos operandos. Por ejemplo, en un lenguaje como C++ es posible definir un tipo *matriz* que represente matrices de  $m \times n$  y a continuación definir e inicializar dos variables de ese tipo, para luego definir una tercera como la suma de los anteriores. En este ejemplo, el compilador se encargará de decidir cómo almacenar las tres variables y de generar las instrucciones necesarias para realizar la suma de matrices, lo cual involucrará seguramente varios accesos a memoria y varias sumas de los elementos de las matrices.

En los computadores, los operandos siguen una filosofía similar. Sin embargo, debido al bajísimo nivel de abstracción del código máquina y del lenguaje ensamblador, sólo hay disponible un número pequeño y fijo de tipos de operandos. A la hora de realizar operaciones sobre ellos, los operandos deben estar almacenados en lugares determinados y limitados, siguiendo restricciones según el tipo de los operandos, la operación concreta, o si el operando es un dato de entrada para la operación o un resultado. El programador (o, más frecuentemente, el compilador) debe encargarse de gestionar el movimiento de los datos de un lugar a otro para poder realizar las operaciones deseadas.

Las instrucciones necesitan referirse a los operandos. Para ello, es necesario codificar de alguna manera dónde se encuentra el operando (su dirección). Los *modos de direccionamiento* se refieren a la manera en que

los operandos están especificados dentro de la representación binaria de la instrucción, y por tanto determinan dónde se pueden encontrar esos operandos.

Dado que la representación binaria de una instrucción es de longitud limitada (en MIPS, por ejemplo, todas las instrucciones se codifican usando 32 bits), la codificación elegida impondrá unos límites en el número de direcciones de los operandos (por ejemplo, en MIPS se utilizan 5 bits para codificar el número de registro, por lo que solo se pueden direccionar 32 registros).

En el caso de MIPS, los modos de direccionamiento disponibles son:

**Direccionamiento inmediato:** el operando se encuentra codificado en la propia instrucción. En ensamblador, el operando aparece como un número entero (en decimal o hexadecimal). Es el caso de las constantes que se pueden codificar en menos de 16 bits (que es el número de bits reservado en el formato de instrucción I, ver sección 3.7.2).

**Direccionamiento a registro:** el operando se encuentra en un registro. Es el caso más frecuente, y en ensamblador el registro aparece identificado por su número o por un nombre (ambas formas son equivalentes). El número de registro se codifica en la instrucción en uno de tres posibles campos, de 5 bits cada uno. En algunos casos, el registro concreto donde se pueda colocar el operando estará sujeto a algunas restricciones según el tipo del operando (p.e.: números enteros o en punto flotante) o la operación (p.e.: el cociente de una división siempre se almacenará en el registro especial HI, o los operandos de doble precisión siempre se encontrarán en registros pares). Véase la sección 3.3.1 para más detalles.

**Direccionamiento base más desplazamiento:** el operando se encuentra en memoria, y la dirección de memoria donde se encuentra se calcula sumando el valor almacenado en un registro (base) y una constante (desplazamiento). En ensamblador aparece el valor del desplazamiento seguido del nombre del registro entre paréntesis. En la instrucción se codifica el número del registro en un campo de 5 bits y la constante en complemento a 2 en un campo de 16 bits. Véase la sección 3.4.2 para más detalles.

**Direccionamiento relativo al contador de programa (PC):** el operando se encuentra en memoria y es una instrucción destino de un salto. En ensamblador, la instrucción de destino aparecerá normalmente identificada por una etiqueta<sup>2</sup>, mientras que en la instrucción se codifica en complemento a 2 y en un campo de 16 bits el número de instrucciones que hay que avanzar (o retroceder, si el número es negativo) el contador de programa<sup>3</sup> desde la instrucción siguiente a la actual para llegar a la instrucción de destino. Véanse las secciones 3.4.3 y 3.7.2 para más detalles.

**Direccionamiento pseudodirecto:** de nuevo, el operando se encuentra en memoria y es una instrucción destino de un salto. En ensamblador, la instrucción de destino aparecerá normalmente identificada por una etiqueta o por su dirección en memoria. En la instrucción se codifican 26 de los 32 bits de la instrucción de destino. Véanse las secciones 3.4.3 y 3.7.3 para más detalles.

No todos los modos de direccionamiento se pueden utilizar en cualquier momento. Al contrario, cada instrucción permitirá sólo unos pocos modos de direccionamiento fijos. En particular, en MIPS y otras arquitecturas RISC casi todas las instrucciones utilizan preferentemente el direccionamiento a registro y sólo las instrucciones específicas de acceso a memoria utilizan el direccionamiento base más desplazamiento.

### 3.3.1. Los registros en MIPS

Los operandos sobre los que trabajan las instrucciones de un computador pueden estar almacenados en *registros* o en *memoria*:

<sup>2</sup>También puede aparecer el número de **bytes** que es necesario avanzar (o retroceder si es negativo) el PC para apuntar a la instrucción de destino.

<sup>3</sup>El contador de programa (llamado también PC) es un registro especial que contiene la dirección de la siguiente instrucción a ejecutar.

**Registros:** hay pocos disponibles, pero tienen un tiempo de acceso muy rápido.

**Memoria:** es de gran tamaño pero con un tiempo de acceso mucho más lento que los registros.

MIPS dispone de 32 registros enteros de propósito general y 32 registros de coma flotante de simple precisión (que también se pueden utilizar como 16 de doble precisión, ver sección 3.4.5). También dispone de algunos registros especiales como los registros HI y LO descritos en la sección 3.4.1, o el contador de programa (PC) a los que sólo se puede acceder con determinadas instrucciones.

En MIPS, por ser una arquitectura de tipo RISC, la mayoría de las operaciones necesitan que sus operandos se encuentran en registros. Debido a que los datos de un programa no caben todos en el número limitado de registros disponibles, estos datos se almacenan en general en la memoria y el programador (o el compilador) necesita encargarse de colocar los datos adecuados en los registros cuando necesita operar con ellos y almacenar los resultados en memoria para poder utilizar los registros en otra operación (ver sección 3.4.2).

Supongamos, como analogía, que tenemos una carpintería, donde fabricamos a mano todo tipo de muebles. En cada momento, en nuestra mesa de trabajo tendremos sólo las piezas que necesitamos para construir el mueble actual, mientras que en el almacén (o *memoria*, en nuestra analogía con el computador) tendremos guardados todos los tornillos, maderas y demás utensilios necesarios para construir muebles, ya que todos no caben a la vez en nuestra mesa de trabajo. Esta última haría, por tanto, el papel del *archivo* o *banco* de registros.

El hecho de disponer sólo de 32 registros se debe a que un número demasiado alto de registros incrementaría el tiempo de acceso al banco de registros y, por tanto, también aumentaría el tiempo que tardan los programas en ejecutarse. Disponer de pocos registros también aumentaría el tiempo de ejecución de los programas porque sería necesario mover más frecuentemente los operandos de la memoria a los registros para trabajar con ellos, incrementando el número de accesos a memoria. El número de registros también afecta al formato de instrucción, debido a que un mayor número de registros implicaría instrucciones más largas para poder codificar el número de los registros usados por la instrucción (ver sección 3.7). Por tanto, ofrecer 32 registros de propósito general es un compromiso para no penalizar demasiado ni el tiempo de acceso a los registros, ni el número de accesos de memoria, ni el tamaño de las instrucciones.

Los 32 registros enteros se identifican en ensamblador por su número precedido de un signo de dólar ( $\$0$ ,  $\$1$ ,  $\$2$ , ...,  $\$31$ ) o, equivalentemente, por su nombre precedido de un signo de dólar. La tabla 1 muestra los nombres de los registros enteros de propósito general de MIPS y algunas características de cada uno.

Para cada registro, en la tabla 1 se muestra cuál es su uso habitual y si es un registro cuyo valor debe ser preservado por los procedimientos que lo modifiquen o no (ver sección 3.4.4 para una explicación de las convenciones para llamada a procedimientos en MIPS).

El registro  $\$zero$  es especial, ya que cualquier escritura en ese registro se ignora y su valor es siempre la constante  $0^4$ .

El registro  $\$at$  lo usa el ensamblador para implementar las pseudoinstrucciones que necesitan un registro temporal, como se explica en la sección 3.7.4.

Los registros  $\$v0$  y  $\$v1$  se utilizan para que los procedimientos comuniquen sus resultados, mientras que los registros  $\$a0$ ,  $\$a1$ ,  $\$a2$  y  $\$a3$  se utilizan para comunicar a un procedimiento sus argumentos, como se explica en la sección 3.4.4.

Los registros  $\$t0$ ,  $\$t1$ ,  $\$t2$ ,  $\$t3$ ,  $\$t4$ ,  $\$t5$ ,  $\$t6$ ,  $\$t7$ ,  $\$t8$  y  $\$t9$  son los registros de uso temporal general. Se utilizan para valores intermedios que no es necesario conservar entre llamadas. Por su parte, los registros  $\$s0$ ,  $\$s1$ ,  $\$s2$ ,  $\$s3$ ,  $\$s4$ ,  $\$s5$ ,  $\$s6$  y  $\$s7$  se utilizan para valores temporales de más larga duración que se desea preservar cuando se hace una llamada a otro procedimiento. El valor de estos últimos se debe almacenar en la pila antes de modificarlo, como se explica en la sección 3.4.4.

Los registros  $\$k0$  y  $\$k1$  están reservados para su uso por el sistema operativo (más concretamente, para su uso por las rutinas de atención de interrupciones, ver sección 3.5) y no deben ser usados por el resto de programas, ya que su valor puede cambiar en cualquier momento (desde el punto de vista del programa).

<sup>4</sup>Esta propiedad es muy útil, entre otras cosas, para implementar pseudoinstrucciones como se muestra en la sección 3.7.4.

Números	Nombres	Uso	Preservados entre llamadas
0	\$zero	Valor constante 0	No aplicable
1	\$at	Reservado para uso temporal del ensamblador	No
2, 3	\$v0, \$v1	Resultados de las llamadas a procedimiento y evaluación de expresiones	No
4, ..., 7	\$a0, ..., \$a3	Argumentos de las llamadas a procedimiento	No <sup>5</sup>
8, ..., 15	\$t0, ..., \$t7	Temporales	No
16, ..., 23	\$s0, ..., \$s7	Temporales guardados	Sí
24, 25	\$t8, \$t9	Temporales	No
26, 27	\$k0, \$k1	Reservado para su uso por el núcleo del sistema operativo	No
28	\$gp	Puntero global	Sí
29	\$sp	Puntero de pila	Sí
30	\$fp	Puntero de marco	Sí
31	\$ra	Dirección de retorno	Sí

Tabla 1: Registros enteros de MIPS

El registro \$gp se utiliza para apuntar al comienzo del área de memoria reservada para las variables globales del programa o librería.

El registro \$sp apunta siempre a la cima de la pila, tal y como se explica en la sección 3.4.4.

El registro \$fp se usa para apuntar al marco de pila del procedimiento actual, es decir, a la zona de la pila que se utiliza para almacenar variables locales. Su uso no es estrictamente necesario (depende del compilador y de las opciones de compilación), por lo que se puede utilizar alternativamente como un registro temporal adicional preservado entre llamadas.

El registro \$ra se utiliza para almacenar la dirección de retorno del procedimiento actual, como se explica en la sección 3.4.4.

Obsérvese que, salvo en el caso del registro \$zero, la información sobre el uso de los registros y su preservación entre llamadas es sólo una convención: el hardware no impide que se usen de forma diferente si se quiere. Estas convenciones forman parte del ABI de la plataforma, y deben ser seguidas por programadores (y compiladores) para que sus procedimientos puedan llamar a procedimientos escritos por otros programadores.

Además de los registros enteros mencionados aquí, el ISA de MIPS también ofrece un conjunto de registros para trabajar en coma flotante, descrito en la sección 3.4.5.

### 3.4. El ISA de MIPS

Clasificaremos las instrucciones de MIPS según el tipo de acción que realizan y mostraremos ejemplos de algunas instrucciones de cada tipo.

Algunas de las instrucciones que se mencionarán en esta sección son pseudoinstrucciones. Es decir, son instrucciones de ensamblador que no tienen una traducción directa a código máquina (ver sección 3.7.4).

<sup>5</sup>Aviso: algunas ediciones de «Estructura y Diseño de Computadores» de David A. Patterson y John L. Hennessy clasifican incorrectamente estos registros como preservados entre llamadas. Este error ha sido subsanado en ediciones modernas de dicho libro, aunque se ha propagado a otros materiales disponibles en Internet. En particular, la figura 3.13 de la página 131 de la 1ª edición en español del libro citado anteriormente es incorrecta. Hacer estos registros preservados entre llamadas sería una mala decisión en el diseño del ISA porque incrementaría el tráfico entre los registros y la pila.

### 3.4.1. Instrucciones aritmético-lógicas

El primer tipo de operación, y quizás el más natural, puesto que es el que realiza el verdadero procesamiento de los datos, es implementado por las **instrucciones aritméticas**. Los computadores han de ser capaces de realizar sumas, restas, multiplicaciones y divisiones con los datos. Un ejemplo de ello sería la instrucción de suma de registros en MIPS:

```
add $a, $b, $c # $a=$b+$c
```

En la línea anterior,  $\$a$ ,  $\$b$  y  $\$c$  indican cualesquiera de los 32 registros enteros de MIPS. Esta operación indica al computador que sume ( $\text{add}^6 = \text{sumar}$ ) el contenido de los registros  $\$b$  y  $\$c$  y que guarde el resultado en  $\$a$ .

Se puede observar que el formato es rígido: las operaciones tienen, forzosamente, dos operandos fuentes y un operando destino. Si se quisieran sumar 4 valores, por ejemplo, los almacenados en los registros  $\$b$ ,  $\$c$ ,  $\$d$  y  $\$e$ , para colocar el resultado en el registro  $\$a$ , se podría hacer así:

```
add $a, $b, $c # $a=$b+$c
add $a, $a, $d # $a=$a+$d=$b+$c+$d
add $a, $a, $e # $a=$a+$e=$b+$c+$d+$e
```

Muy similares a las instrucciones aritméticas (suma, resta, multiplicación, división, etc) son las **instrucciones lógicas**, que trabajarán siguiendo un esquema idéntico (dos operandos fuentes y uno destino), pero realizando operaciones lógicas sobre los bits de dichos operandos, en lugar de aritméticas (por ejemplo, `and`, `or`, `not`, `xor`). Dentro de este tipo de operaciones también encontramos operaciones de manipulación de bits (desplazamientos de bits a derecha o a izquierda, etc).

En este apartado explicaremos algunas de las instrucciones aritmético-lógicas más representativas del ensamblador de MIPS, que trabajan con datos de tipo entero (véase la sección 3.4.5 para las instrucciones que trabajan con números en coma flotante). Comenzamos con las instrucciones aritméticas (suma, resta, multiplicación, etc).

#### Instrucciones aritméticas

Las dos instrucciones aritméticas más comunes son la suma y la resta. Por ejemplo:

```
add rd, rs, rt # rd=rs+rt
sub rd, rs, rt # rd=rs-rt
```

Siempre tienen tres operandos. El operando de destino es cualquiera de los 32 registros enteros<sup>7</sup>. En el caso de las instrucciones anteriores, los operandos fuente son también dos registros enteros cualquiera (que pueden coincidir entre sí y con el registro destino).

Existe un tipo de instrucción adicional, denominada *de tipo inmediato*, uno de cuyos operandos no es un registro sino un valor constante (al cual se denomina *valor inmediato*). Por ejemplo, la siguiente instrucción suma un valor inmediato (una constante entera de cómo máximo 16 bits en complemento a 2) al registro  $rs$  para almacenar el resultado en  $rd$ :

```
addi rd, rs, inm # rd=rs+inm
```

Por ejemplo:

<sup>6</sup>A la palabra que determina el tipo de operación en ensamblador se le llama *mnemónico* de la instrucción; ejemplos de *mnemónicos* que veremos en este capítulo serán `add`, `sub`, `bne`, `lw`, etc.

<sup>7</sup>En realidad, aunque  $\$0$  puede ser registro destino, es de sólo lectura, y siempre contiene la constante cero. Esto se puede utilizar para incluir en el programa instrucciones que no tienen ningún efecto (*nops*).

```

addi $t0,$s0,100    # Suma 100 al registro $s0 y guarda el
                    # resultado en $t0.
addi $t0,$s0,-100   # Resta 100 al registro $s0 y guarda el
                    # resultado en $t0.

```

El repertorio de instrucciones de MIPS también contiene otras instrucciones aritméticas como la multiplicación y la división. Hay, sin embargo, un par de particularidades que tenemos que tener en cuenta a la hora de trabajar con estas instrucciones. La primera es que el programador debe determinar si la operación que quiere hacer (multiplicación o división) es con o sin signo. Este problema no aparecía al sumar y al restar, puesto que, como sabemos, se suman de igual manera los números sin o con signo en complemento a 2. Sin embargo, al multiplicar, obviamente no es lo mismo, por ejemplo, multiplicar por el valor 0xFFFFFFFF si se considera con signo (que correspondería al valor -1) que si se considera sin signo (que correspondería al valor  $2^{32} - 1$ ). Para diferenciar estos dos tipos de operación, se emplean *mnemónicos* diferentes. Así, las instrucciones `mult` y `div`, respectivamente, multiplicarán o dividirán sus registros fuentes considerándolos con signo, mientras que `multu` y `divu` lo harán sin tomar en cuenta éste.

La segunda consideración es que, a diferencia de la suma y la resta, un registro de 32 bits puede no ser suficientemente grande para almacenar el resultado. En efecto, si multiplicamos dos valores de 32 bits, el resultado puede llegar a ocupar hasta 64 bits.

Por ello, la instrucción de multiplicación en MIPS tiene sólo dos operandos fuente explícitos y, en lugar de un operando destino indicado directamente en la instrucción, dos registros de 32 bits implícitos llamados LO y HI donde siempre se coloca el resultado de la operación. Así, la instrucción

```
mult $10,$11 # Multiplica $10 por $11, con signo
```

multiplica con signo los valores de los registros \$10 y \$11 (de 32 bits cada uno) para colocar el resultado en los registros HI y LO. En estos registros se almacenan, respectivamente, las mitades superior (*high*) e inferior (*low*) del resultado (que puede tener, por tanto, hasta 64 bits). Por su parte, la instrucción

```
multu $10,$11 # Multiplica $10 por $11, sin signo
```

realiza la misma operación pero considerando los valores de los registros \$10 y \$11 como enteros de 32 bits sin signo. El resultado será un número de 64 bits entre 0 y  $2^{64} - 1$  almacenado entre los registros HI y LO.

Los registros HI y LO no pertenecen al grupo de los 32 registros enteros de uso general, por lo que no pueden utilizarse como operandos de otro tipo de instrucciones. Así, si queremos acceder a sus valores, tenemos que pasarlos previamente a alguno de los registros generales del procesador. Para ello, MIPS ofrece las siguientes instrucciones:

```
mflo rd # Mueve el contenido de LO al registro rd
mfhi rd # Mueve el contenido de HI al registro rd
```

Por ejemplo para almacenar en el registro \$t0 cinco más el triple del valor de \$t1 podríamos hacer lo siguiente:

```

addi $t2, $0, 3 # Ponemos la contante 3 en un registro auxiliar
mult $t1, $t2   # HI y LO juntos contienen 3 * $t1
mflo $t0       # Copiamos a $t0 los 32 bits menos
                # significativos del resultado de la
                # operación anterior.
addi $t0, 5     # Realizamos la suma

```

Obsérvese que en el ejemplo anterior se descartan los 32 bits más significativos del resultado de la multiplicación, por lo que el resultado solo será correcto cuando  $t1$  valga entre  $\frac{-2^{31}}{3}$  y  $\frac{2^{31}-1}{3}$ .

Con las divisiones ocurre una situación similar: cuando hacemos una división entera de dos valores de 32 bits, generalmente nos interesará conocer tanto el cociente como el resto de la operación y cada uno de ellos puede llegar a ocupar también 32 bits. Así, la instrucción

```
div $10,$11 # Divide $10 por $11, con signo.
```

divide (con signo) el valor contenido en \$10 por el valor contenido en \$11, colocando el cociente entero resultante en el registro LO y el resto de la división en el registro HI. Y la instrucción

```
divu $10,$11 # Divide $10 por $11, sin signo.
```

opera análogamente pero sin considerar el signo de los valores de los registros \$10 y \$11. De nuevo, el programador puede acceder a estos resultados utilizando las instrucciones `mfhi` y `mflo`.

### Desbordamientos

Cuando realizamos operaciones aritméticas con operandos de rango limitado puede ocurrir que el resultado de la operación se encuentre fuera de dicho rango. En el caso de MIPS, el rango de los operandos para la mayoría de las operaciones está limitado, en el caso de los enteros con signo, a los valores representables en complemento a 2 con 32 bits (desde  $-2^{31}$  hasta  $2^{31} - 1$ ) y, en el caso de los enteros sin signo, a los valores representables en binario natural con 32 bits (desde 0 hasta  $2^{32} - 1$ ).

Las situaciones de desbordamiento son fáciles de detectar cuando se trabaja con números con signo. En el caso de la suma, por ejemplo, ocurre cuando se suman dos números positivos y el resultado es negativo, o cuando se suman dos números negativos y el resultado es positivo. Por tanto, basta con mirar los bits de signo de los 3 operandos para detectarlo. El ISA de MIPS (y otros ISAs) proporciona mecanismos para detectar estas situaciones cuando se usan operaciones aritméticas sobre enteros con signo (mediante instrucciones como `add` o `addi`).

La gestión de los desbordamientos, es decir, el procedimiento a seguir cuando se produce un desbordamiento en una operación aritmética durante la ejecución de un programa, depende de la implementación concreta de cada procesador. En el caso del MIPS, un desbordamiento produce una *excepción* (o *interrupción provocada por el software*). Esta excepción produce una llamada no planificada a un procedimiento especial (véase la sección 3.5 para más detalles) que se encarga de tratar la situación (el tratamiento de la excepción puede consistir en mostrar un mensaje de error, terminar con el programa, o cualquier otra acción adecuada).

Como muchas veces el programador quiere hacer operaciones aritméticas sin preocuparse del desbordamiento, MIPS proporciona instrucciones para ignorarlo si éste se produce. Por ejemplo, en el caso de la suma en MIPS, existe la instrucción `addu` (*add unsigned*, suma sin considerar el signo), que realiza la suma bit a bit ignorando el posible desbordamiento (es decir, sin provocar la excepción en caso de que se produzca, y continuando con la ejecución de forma normal).

### Instrucciones lógicas y de desplazamiento de bits

Las instrucciones lógicas son análogas a las instrucciones aritméticas, sólo que las instrucciones lógicas trabajan sobre los registros fuente bit a bit en lugar de interpretar el contenido de los registros como números (realizando el `and` o el `or` lógico de cada bit, por ejemplo). Algunos ejemplos de instrucciones lógicas son:

```
and rd,rs,rt    # rd es el AND lógico de los registros rs y rt
or  rd,rs,rt    # rd es el OR lógico de los registros rs y rt
```

También existen las correspondientes versiones inmediatas:

```
andi rd,rs,imm    # rd es el AND lógico de rs y el valor imm
ori  rd,rs,imm    # rd es el OR lógico de rs y el valor imm
```

Por ejemplo, si el registro \$10 contiene el valor 0xf0f0f0f0 y el \$9 contiene el valor 0x0ff00000, las siguiente instrucciones:

```
and  $11,$10,$9   # $11 = $10 & $9 = 0x00f00000
or   $12,$10,$9   # $11 = $10 | $9 = 0xfff0f0f0
andi $13,$10,0xff # $11 = $10 & 0xff = 0x000000f0
ori  $14,$10,0xff # $11 = $10 & 0xff = 0xf0f0f0ff
```

almacenarán en los registros \$11, \$12, \$13 y \$14 los valores 0x00f00000, 0xfff0f0f0, 0x000000f0 y 0xf0f0f0ff, respectivamente.

En el grupo de las instrucciones lógicas suelen también añadirse las *instrucciones de desplazamiento de bits*. Las instrucciones de desplazamiento que veremos son las siguientes:

```
sll rd,rs,n       # Desplaza a la izquierda (lógico)
srl rd,rs,n       # Desplaza a la derecha (lógico)
sra rd,rs,n       # Desplaza a la derecha (aritmético)
```

La instrucción `sll` desplaza los bits contenidos en el registro `rs`  $n$  posiciones hacia la izquierda, introduciendo  $n$  ceros por la derecha y descartando los  $n$  bits más significativos del valor original. De esta forma, el valor almacenado en `rd` será siempre igual al valor en el registro `rs` multiplicado por  $2^n$ . Por ejemplo, si el registro \$10 contiene el valor 0xfedcba98, la siguiente instrucción:

```
sll $11,$10,4     # $11 = 0xfedcba98 << 4 = 0xedcba980
```

almacenará el valor 0xedcba980 en el registro \$11.

Por su parte, `srl` funciona de forma similar, pero los bits se desplazan hacia la derecha, introduciendo  $n$  ceros por la izquierda y descartando los  $n$  bits menos significativos del valor original. El valor almacenado en el registro `rd` será el resultado de dividir el valor en `rs` por  $2^n$  **si y sólo si el valor de `rs` no era negativo**. El valor resultante siempre será positivo. Por ejemplo, si volvemos a suponer que el registro \$10 contiene el valor 0xfedcba98, la siguiente instrucción:

```
srl $11,$10,4     # $11 = ((unsigned int) 0xfedcba98) >> 4 = 0x0fedcba9
```

almacenará el valor 0x0fedcba9 en el registro \$11.

Por último, `sra` realiza un desplazamiento *aritmético* (en lugar de *lógico*) a la derecha. El funcionamiento es similar a la instrucción anterior, pero en lugar de introducir  $n$  ceros por la izquierda, se replica  $n$  veces el bit más significativo de `rs`. De esta forma se consigue que el valor almacenado en el registro `rd` sea **siempre** el resultado de dividir el valor en `rs` por  $2^n$ , y el valor resultante siempre tendrá el mismo signo que tenía `rs`. Por ejemplo, si \$10 contiene el valor 0xfedcba98, la siguiente instrucción:

```
sra $11,$10,4     # $11 = ((int) 0xfedcba98) >> 4 = 0xffedcba9
```

almacenará el valor 0xffedcba9 en el registro \$11.

Otras instrucciones similares a las de desplazamiento son las de rotación de bits:

```
rol rd,rs,n       # Rota a la izquierda.
ror rd,rs,n       # Rota a la derecha.
```

La instrucción `rol` desplaza los bits contenidos en el registro `rs`  $n$  posiciones hacia la izquierda, introduciendo  $n$  por la derecha y los  $n$  bits más significativos del valor original. `ror` funciona análogamente pero hacia la derecha. Así, si el registro `$10` contiene el valor `0xfedcba98`, las siguientes instrucciones:

```
rol $11, $10, 4
ror $12, $10, 4
```

almacenaran en `$11` y `$12` los valores `0xedcba98f` y `0x8fedcba9`, respectivamente.

Las instrucciones lógicas de desplazamiento y rotación de bits son muy útiles a la hora de examinar o manipular el valor de los bits individuales almacenados en un registro.

### Instrucciones de comparación

Finalmente, nos encontramos con las instrucciones de comparación. En MIPS disponemos de la siguiente instrucción:

```
slt rd, rs, rt # Pone rd a 1 o a 0 si rs es menor o no que rt
```

que compara los valores contenidos en `rs` y `rt` (restándolos) y almacena en `rd` un 1 si el contenido de `rs` es menor que el de `rt` y un 0 si es mayor o igual.

Las instrucciones de comparación suelen combinarse con las instrucciones de salto condicional (`beq` y `bne`, ver sección 3.4.3) para realizar saltos condicionales dependiendo de si un valor es mayor, menor, mayor o igual o menor o igual que otro.

### 3.4.2. Instrucciones de acceso a memoria

Como se ha visto ya, la mayoría de las instrucciones de MIPS operan sólo con registros y constantes. Sin embargo, en el banco de registros caben muy pocos datos, mientras que los programas actuales necesitan manejar grandes cantidades de información que no queda más remedio que almacenar en la memoria, de mucha más capacidad. Por ello, para trabajar con esa información es necesario moverla primero de la memoria a los registros y volverla a mover después de los registros a la memoria. Estas transferencias de información entre registros y memoria se realizan mediante las *instrucciones de acceso a memoria*.

Siguiendo con la analogía de la carpintería utilizada anteriormente (ver sección 3.3.1), estas instrucciones se encargarían de traer el material a la mesa de trabajo del carpintero y de llevar la silla, una vez fabricada, de vuelta al almacén.

Hay dos tipos de instrucciones de transferencia de memoria: las de *carga*, que copian en un registro un dato que se encuentra en una determinada dirección de memoria; y las de *almacenamiento*, que copian en una determinada dirección de memoria un dato que se encuentra en un registro.

Desde el punto de vista del ISA, la memoria está estructurada como una gran tabla unidimensional, y la dirección de memoria actúa como índice de esa tabla. En concreto, en MIPS las direcciones tienen 32 bits, y cada una de las  $2^{32}$  posibles direcciones apunta a una posición de memoria que contiene un byte. Por tanto, hay espacio para  $2^{32}$  bytes (4 GB) o, alternativamente,  $2^{30}$  palabras de 4 bytes, almacenadas cada 4 posiciones de memoria.

Como ya se ha dicho, para leer datos de memoria se utilizan las instrucciones de carga (*load*). Por ejemplo, en el repertorio de instrucciones de MIPS existe la instrucción

```
lw rt, dirección # Load word (carga palabra)
```

que lee los cuatro bytes almacenados en la posición indicada por *dirección* y en las tres siguientes y los escribe en el registro *rt*. La dirección se expresa con el modo de direccionamiento “base más desplazamiento”<sup>8</sup>, es decir mediante un registro y un desplazamiento constante de 16 bits (positivo o negativo). Por ejemplo, si *\$t1* contiene el valor `0x1004000`, la instrucción:

```
lw $t2, 8($t1)
```

leerá las posiciones de memoria desde `0x1004008` a `0x100400b` inclusive y copiará los datos en el registro *\$t2*.

El resultado no será el mismo si los bits del primer byte leído de memoria se copian en los 8 bits más significativos del registro, los bits del segundo byte de memoria en los 8 bits siguientes y así sucesivamente, que si los bytes de memoria se van colocando en el registro empezando por los bits menos significativos del registro. Al primer modo de proceder se le denomina *big-endian byte order*, mientras que al segundo se le denomina *little-endian byte order*. La mayoría de las implementaciones del ISA del MIPS son *big-endian*, aunque también existen implementaciones *little-endian* e incluso algunas que pueden operar de las dos maneras. Nosotros asumiremos ordenación *little-endian* (el byte apuntado por la dirección de memoria corresponde al byte de menos peso de una palabra de 32 bits) siempre que no se indique lo contrario<sup>9</sup>.

Si en la memoria estuvieran almacenados los bytes `0x12`, `0x34`, `0x56` y `0x78` en las direcciones `0x1004008`, `0x1004009`, `0x100400a` y `0x100400b` respectivamente, la instrucción del ejemplo anterior escribiría en *\$t2* el valor `0x12345678` en una arquitectura *big-endian* o el valor `0x78563412` en una arquitectura *little-endian*.

`lw` es una instrucción de carga *alineada*. Es decir, la dirección de memoria del primer byte a leer debe ser múltiplo del tamaño del dato (una palabra de 4 bytes en este caso). Todas las instrucciones de acceso a memoria que veremos son alineadas<sup>10</sup>. La mayoría de las veces los datos estarán colocados adecuadamente en memoria para acceder directamente a ellos con instrucciones alineadas, pero en ocasiones será necesario utilizar varias instrucciones alineadas (que accedan a datos más pequeños) para cargar un dato que se encuentre en memoria en una posición no alineada con su tamaño.

Las instrucciones de almacenamiento son análogas a las de carga. Por ejemplo, tenemos la instrucción

```
sw rt,dirección # Store word (almacena palabra)
```

que copia el contenido de *rt* en los cuatro bytes de memoria indicados por *dirección*. Por ejemplo, si *\$t1* contiene el valor `0x1004000` y *\$t2* contiene el valor `0x12345678`, la instrucción:

```
sw $t2, 8($t1)
```

escribirá `0x78`, `0x56`, `0x34`, `0x12` en las posiciones de memoria `0x1004008`, `0x1004009`, `0x100400a` y `0x100400b`, respectivamente, cuando se usa una ordenación de bytes *little-endian*.

Al igual que las instrucciones `lw` y `sw` que sirven para cargar y almacenar palabras (4 bytes), existen otras instrucciones para trabajar con bytes individuales, o con valores de 16 bits (llamados medias palabras). Así, también están disponibles las siguientes instrucciones:

```
lb rt,dirección      # Carga un byte con signo
lbu rt,despl(rs)    # Carga un byte sin signo
sb rt,dirección      # Almacena un byte
```

<sup>8</sup>Aunque éste es el único modo de direccionamiento admitido por las instrucciones de acceso a memoria a nivel de código máquina, el ensamblador ofrece algunos modos de direccionamiento adicionales que se implementan mediante pseudoinstrucciones (véase la sección A3.2.1).

<sup>9</sup>MARS, el simulador de MIPS que usamos en la asignatura, usa ordenamiento *little-endian*.

<sup>10</sup>De hecho, en MIPS todas las instrucciones de acceso a memoria son alineadas aunque existen pseudoinstrucciones para acceder a memoria con direcciones no alineadas (p.e.: `ulw`). Algunas otras ISAs incluyen instrucciones de acceso a memoria no alineado.

```
lh rt,dirección      # Carga media palabra (2 bytes) con signo
lhu rt,despl(rs)    # Carga media palabra (2 bytes) sin signo
sh rt,dirección     # Almacena media palabra (2 bytes)
```

Estas instrucciones obtienen la dirección de memoria exactamente igual que `lw` y `sw`, aunque en este caso la dirección no tendrá que ser múltiplo de 4, sino de 2 en el caso de `lh`, `lhu` y `sh`, y de 1 en el caso de `lb`, `lbu` y `sb`.

Todas estas instrucciones trabajan con los 8 o 16 bits de menos peso del registro `rt`. En el caso de los almacenamientos, se copiará en el byte/media palabra de memoria direccionado los 8 o 16 bits de menor peso del registro `rt`.

En el caso de las cargas, el valor leído de memoria se copiará en los 8 o 16 bits de menor peso del registro `rt` y los 24 o 16 bits superiores se rellenarán replicando el bit de signo del byte/palabra cargado en el caso de `lb` y `lh` o con ceros en el caso de `lbu` y `lhu`.

Por ejemplo, si la dirección de memoria `0x1001000b` contiene el valor `0xf1`, y `$t0` contiene el valor `0x10010000`, la instrucción

```
lb $t1,0xb($t0)
```

cargará el valor `0xfffffffff1` en el registro `$t1`. Por el contrario, la instrucción

```
lbu $t1,0xb($t0)
```

cargará el valor `0x000000f1` en `$t1`.

### Traducción de acceso a arrays

Los datos usados por un programa normalmente se almacenan en memoria usando estructuras de datos. Unas de las estructuras más usadas y más simples son los *arrays*. Un array almacena consecutivamente en memoria varios elementos del mismo tipo (y tamaño), de forma similar a una tabla. Los elementos almacenados se identifican mediante números enteros consecutivos llamados índices. En C y muchos otros lenguajes, el primer elemento tiene el índice 0 y al elemento *i*-ésimo del array *a* se le denota `a[i]`.

Para utilizar un array en ensamblador es necesario conocer:

1. La dirección de memoria a partir de la que están almacenados los elementos del array (*base*). Esta dirección puede ser un argumento de un procedimiento (en cuyo caso estará almacenada en un registro) o una etiqueta del programa (en cuyo caso habrá que moverla a un registro con la instrucción `la`).
2. El tamaño de cada elemento del array (*t*).
3. El índice del elemento al que queremos acceder, ya sea para leer o escribir (*i*). Supondremos que el índice del primer elemento es 0.

Para acceder a un elemento de un array, tanto para lectura como para escritura, primero es necesario calcular su dirección, que vendrá dada por la expresión  $base + t \times i$ . Para realizar la multiplicación se deberá utilizar la instrucción `sll` cuando *t* sea potencia de 2, lo cual es muy frecuente.

Por ejemplo, el siguiente fragmento de código:

```
int v[20];
...
v[i] = v[i + 1];
```

se podría traducir de la siguiente forma si suponemos que  $i$  es una variable entera y está almacenada en el registro  $\$s0$ :

```
.data
v: .space 80      # 20 elementos de 4 bytes cada uno
...
.code
la $t0,v          # Carga la base de v en $t0
sll $t1,$s0,2     # Calcula el desplazamiento desde el inicio del
                  # array hasta el elemento i-ésimo: 4 * i
add $t2,$t0,$t1  # Calcula la dirección de v[i]
lw $t4, 4($t2)   # Lee v[i+1], cuya dirección es 4 bytes mayor
                  # que la de v[i]
sw $t4, 0($t2)   # Escribe v[i]
```

### 3.4.3. Instrucciones de salto

La ejecución normal de los programas se realiza en orden secuencial, es decir, las instrucciones se ejecutan una tras otra tal y como están situadas en memoria, incrementando cada vez en 4 el valor del registro contador de programa (PC), ya que cada instrucción ocupa 4 bytes. Sin embargo se caracteriza por la capacidad de tomar decisiones. Esto se traslada en cambios en el flujo de ejecución del programa en función del valor de algunos operandos (registros en MIPS). Es decir, se ejecutará un código determinado si se cumple una condición, u otro trozo de código si no se cumple.

El repertorio de instrucciones MIPS proporciona varias instrucciones para alterar el flujo del programa en función de una condición. Estas instrucciones se conocen como *saltos condicionales* debido a que el salto, es decir, la alteración del orden de ejecución, se produce en función de una condición:

```
beq rs,rt,etiq   # Salta a la dirección del programa situado en
                  # etiq si el contenido de los dos registros
                  # es igual.

bne rs,rt,etiq   # Salta a la dirección del programa situada en
                  # etiq si el contenido de los dos registros
                  # es diferente.
```

También existen instrucciones de *salto incondicional*, en las que el cambio en el flujo de ejecución del programa se realiza siempre que se ejecuta la instrucción, independientemente de ninguna otra condición:

```
j etiqueta      # Salta a la instrucción en la
                  # dirección apuntada por etiqueta.

jr rd           # Salta a la instrucción en la
                  # dirección contenida en rd.
```

### Traducción de expresiones condicionales

Los programas de alto nivel incluyen instrucciones que, dependiendo de los datos de entrada y los creados durante la computación, hacen que el computador ejecute un camino del programa u otro. Un ejemplo de instrucción de toma de decisiones es la construcción condicional

```

if <condición> then
    <código a ejecutar si la condición es cierta>
else
    <código a ejecutar si la condición es falsa>

```

que se encuentra, con una u otra sintaxis, en prácticamente todos los lenguajes de alto nivel. Este código analiza la condición (la cual puede ser, por ejemplo, una comparación de magnitud entre dos valores), y ejecuta las instrucciones que hay después del `then`, si se cumple la condición, o las que hay después del `else` en caso contrario.

La manera de traducir estas instrucciones a ensamblador es a través de combinaciones de las instrucciones de comparación y las de salto (ver secciones 3.4.1 y 3.4.3).

Así, por ejemplo, la secuencia

```

slt $10,$11,$12
bne $10,$0,etiqueta

```

serviría para saltar a la dirección apuntada por `etiqueta` si `$11` contiene un valor menor que `$12`. Jugando con el orden de los registros comparados por la primera instrucción (`slt`) y usando como segunda instrucción `bne` y `beq` con el registro `$0` como uno de los operandos, pueden realizarse todos los posibles saltos condicionales en función de la comparación de dos enteros.

El ensamblador de MIPS proporciona instrucciones y pseudoinstrucciones para realizar saltos condicionales en función de cualquier comparación de valores enteros, como se puede ver en la tabla 2.

Tabla 2: Instrucciones y pseudoinstrucciones para realizar saltos condicionales.

(Pseudo)instrucción	Comparación	Expansión en instrucciones
<code>beq rs,rt,etiq</code>	$=$	No aplicable
<code>bne rs,rt,etiq</code>	$\neq$	No aplicable
<code>bgt rs,rt,etiq</code>	$>$	<code>slt \$at,rt,rs</code> <code>bne \$at,\$0,etiq</code>
<code>blt rs,rt,etiq</code>	$<$	<code>slt \$at,rs,rt</code> <code>bne \$at,\$0,etiq</code>
<code>bge rs,rt,etiq</code>	$\geq$	<code>slt \$at,rs,rt</code> <code>beq \$at,\$0,etiq</code>
<code>ble rs,rt,etiq</code>	$\leq$	<code>slt \$at,rt,rs</code> <code>beq \$at,\$0,etiq</code>

Utilizando las instrucciones anteriores, el siguiente fragmento de programa en C:

```

if (i < j)
    k = k + 1;
k = 4 * k

```

se podría traducir en ensamblador de la siguiente manera, suponiendo que las variables `i`, `j` y `k` están almacenadas en los registros `$t0`, `$t1` y `$t2`, respectivamente:

```

slt $t3,$t0,$t1 # $t3 = 1 si i < j o 0 si i >= j
beq $t3,$0,l1 # Salta hasta "l1" si la condición es falsa
addi $t2,$t2,1 # Se ejecuta si la condición es cierta
l1: sll $t2,$t2,2 # Se ejecuta si la condición es cierta o falsa

```

Obsérvese que las dos primeras instrucciones se podrían substituir por “bge \$t0,\$t1,l1”<sup>11</sup> (que salta a l1 cuando es cierta la condición opuesta a la del fragmento de código en C).

Si el fragmento a traducir tuviera una parte a ejecutar si no se cumple la condición, se procedería de manera similar. Por ejemplo:

```
if (i >= j)
    k = k * i;
else
    k = 4 * k
k = k + j
```

Se podría traducir (suponiendo que las variables se encuentran en los mismos registros que en el ejemplo anterior):

```
blt $t0,$l1,l1    # Salta hasta "l1" si la condición es falsa
mult $t2,$t0      # Se ejecuta si la condición es cierta
mflo $t2          # Se ejecuta si la condición es cierta
j l2              # evita ejecutar la parte siguiente
l1: sll $t2,$t2,2  # Se ejecuta si la condición es falsa
l2: add $t2,$t2,$t1 # Se ejecuta si la condición es cierta o falsa
```

### Traducción de bucles

Los lenguajes de alto nivel también ofrecen construcciones para repetir varias veces la ejecución de algunas instrucciones. Hay diversas de estas construcciones (bucles *for*, *while*, *repeat*...). Cada *iteración* se repite el mismo conjunto de instrucciones, aunque el valor de las variables será (normalmente) distinto. Por ejemplo, el siguiente fragmento de código en C calcula la suma de los elementos de un array de *n* elementos enteros llamado *a* y deja el resultado en la variable *s*:

```
int s = 0;
for (int i = 0; i < n; i++) {
    s = s + a[i]
}
```

Los bucles también se traducen mediante combinaciones de instrucciones de comparación y de salto (ver secciones 3.4.1 y 3.4.3). Si suponemos que las variables *p*, *i* y *n* están almacenadas en los registros \$v0, \$t0 y \$a1 respectivamente, y que la dirección de inicio del array *a* está almacenada en \$a0, el ejemplo anterior se podría traducir así:

```
addi $v0,$0,0    # Inicialización de p
addi $t0,$0,0    # Inicialización de i
l1: bge $t0,$a1,l2 # Comprobación de la condición
# Comienzo del cuerpo del bucle
sll $t1,$t0,2    # Cálculo del desplazamiento de a[i] en a
add $t2,$a0,$t1 # Cálculo de la dirección de a[i]
lw $t3,0($t2)   # Carga de a[i]
add $v0,$v0,$t3 # Suma a p
# Fin del cuerpo del bucle
addi $t0,$t0,1  # Incrementa i
j l1            # Salta a la siguiente iteración
l2:
```

<sup>11</sup>Al hacer esta substitución, ya no se usaría el registro \$t3 para almacenar el resultado de la comparación, sino el registro \$at

### 3.4.4. Instrucciones para soportar procedimientos

Los *procedimientos* son una herramienta que los programadores usan para estructurar mejor los programas. Permiten que el programa sea más legible y también permiten que el código sea reutilizado (tanto varias veces dentro de un mismo programa, como incluso para otros programas). Los procedimientos podrían definirse como secuencias de instrucciones que ejecutan una tarea determinada sobre unos operandos de entrada para producir unos determinados resultados. En la jerga de la programación en ensamblador, a los procedimientos a menudo se les llama también *rutinas* o *subrutinas*. Algunos lenguajes, como C, llaman *funciones* a los procedimientos.

Un procedimiento se caracteriza por tener unos parámetros de entrada, que son sobre los que opera, y unas variables de salida, en donde guarda los resultados para su uso posterior en el programa. Los procedimientos pueden usar variables locales, que son privadas a cada invocación del procedimiento. Los procedimientos también pueden leer y modificar variables globales, que son compartidas entre varios procedimientos.

En general, un programa se estructura como una serie de procedimientos que se llaman entre sí. Cuando un procedimiento quiere llamar a otro, al procedimiento que realiza la llamada se le denomina *invocador* (*caller*) y al que es llamado se le denomina *invocado* (*callee*). Estos dos procedimientos deben seguir los siguientes pasos:

1. El invocador sitúa los parámetros en un lugar donde el invocado pueda acceder a ellos.
2. Se transfiere el control al invocado.
3. El invocado adquiere los recursos de almacenamiento (memoria y registros) necesarios.
4. El invocado realiza la tarea deseada.
5. El invocado sitúa los resultados en un lugar donde el invocador pueda acceder a ellos.
6. El invocado libera los recursos adquiridos en el paso 3.
7. Se retorna el control al punto de invocación del procedimiento, que será distinto para invocaciones distintas del mismo procedimiento.

Las distintas ISAs tienen que ofrecer instrucciones que den soporte eficiente a los procedimientos. Además, son necesarias una serie de convenciones a nivel del ABI para organizar el paso de parámetros entre procedimientos y para que los procedimientos puedan compartir los registros y la memoria sin estorbarse entre sí.

En MIPS, disponemos de la siguiente instrucción para transferir el control a un procedimiento de forma que luego se pueda regresar a la posición anterior del programa<sup>12</sup>:

```
jal etiqueta          # Salta a la instrucción en la dirección
                      # apuntada por etiqueta, guardando la
                      # dirección de la instrucción siguiente
                      # en $ra ($31)
```

que sirve para almacenar en el registro `$ra` la dirección de la instrucción siguiente y transferir la ejecución del programa al principio de la subrutina cuyo código comienza en la dirección apuntada por `etiqueta`. Gracias a que la dirección de la instrucción siguiente se almacena en `$ra`, la subrutina invocada puede devolver el control de la ejecución al código invocador mediante la instrucción `jr`:

```
jr $ra                # Salta a la dirección contenida en $ra ($31)
```

<sup>12</sup>La instrucción `jal` es análoga a la instrucción `j` que se vio en la sección 3.4.3. También se dispone de la instrucción `jalr`, análoga a la instrucción `jr`.

El paso de parámetros se realiza mediante un conjunto de registros reservados para ello. Como se mencionó en la sección 3.3.1, estos registros son  $\$a0$ ,  $\$a1$ ,  $\$a2$  y  $\$a3$  para los parámetros enteros (y  $\$f12$ ,  $\$f13$ ,  $\$f14$ ,  $\$f15$  para valores en coma flotante); y  $\$v0$  y  $\$v1$  para los resultados enteros (y  $\$f0$  y  $\$f1$  para valores en coma flotante). Si el espacio de almacenamiento ofrecido por dichos registros no es suficiente para los parámetros o los resultados, se utiliza también la pila (como se explicará en la sección 3.4.4).

Con todo lo anterior, veamos cómo se traduce a MIPS la siguiente función escrita en C:

```
int ejemplo1(int g, int h, int i, int j) {
    f = (g + h) - (i + j);
    return f;
}
```

Como  $g$ ,  $h$ ,  $i$  y  $j$  son los parámetros (o argumentos) de la función, estarán en los registros  $\$a0$ ,  $\$a1$ ,  $\$a2$  y  $\$a3$ , respectivamente. Por otro lado,  $f$  es el valor que se retorna y estará en el registro  $\$v0$ . De esta forma, se podría traducir así:

```
ejemplo1:                # etiqueta para llamar al procedimiento
    add $t0,$a0,$a1       # Suma de g y h
    add $t1,$a2,$a3       # Suma de i y j
    sub $v0,$t0,$t1       # Resta de las dos suma, el resultado se
                        #   almacena en $v0 por ser el resultado
                        #   de la función
    jr $ra                # Volvemos al programa principal
```

Obsérvese cómo se utilizan los registros  $\$t0$  y  $\$t1$  para almacenar los valores intermedios. Como se explicará a continuación, estos registros son *no preservados entre llamadas*, por lo que el procedimiento puede usarlos libremente.

Normalmente los procedimientos llaman a su vez a otros procedimientos, y esto hace que surjan algunos problemas. Por ejemplo, si un procedimiento es invocado y se le pasa un parámetro por el registro  $\$a0$ , y éste a su vez tiene que llamar a un segundo procedimiento que también espera un parámetro en el registro  $\$a0$ , se perdería el valor que había antes de realizar la llamada al segundo procedimiento. Un problema similar surge con la dirección de retorno almacenada en  $\$ra$ . Imaginemos que el procedimiento A llama a B, guardando en  $\$ra$  la dirección de retorno de B a A. Si B a su vez llama al procedimiento C, la dirección de retorno de C a B se guardará en  $\$ra$ , perdiendo el valor anterior, que era el de retorno de B a A. De hecho, el mismo problema surge con cualquier registro.

Estos problemas se pueden resolver utilizando la pila. Hay dos opciones sobre cómo proceder:

1. En el procedimiento invocador y antes de llamar al procedimiento invocado, guardar los registros cuyo valor se quiera utilizar después de que el procedimiento invocado acabe. Después de realizar la llamada, se restaura el valor de los registros usando los valores guardados.
2. En el procedimiento invocado, guardar el valor de los registros que se vayan a modificar antes de operar sobre ellos. Los valores antiguos se restauran justo antes de devolver el control al procedimiento invocador.

En MIPS se utilizan los dos métodos anteriores, dependiendo del registro involucrado<sup>13</sup>:

<sup>13</sup>La razón de MIPS para ofrecer los dos tipos de registros es de eficiencia, ya que se ahorran operaciones de apilado y desapilado. Así, los registros no preservados entre llamadas pueden utilizarse libremente en un procedimiento sin tener que guardar antes su contenido; y a la vez, los registros preservados entre llamadas no tienen que guardarse cada vez que se llama a otro procedimiento.

**Registros no preservados entre llamadas:** son registros cuyo valor no se conserva cuando se llama a un procedimiento. Por tanto, es responsabilidad del procedimiento que realiza la llamada guardar su valor usando el método 1 si quiere usarlo después de la llamada. Los registros de este tipo son  $\$at$ ,  $\$v0$ ,  $\$v1$ ,  $\$v2$ ,  $\$v3$ ,  $\$a0$ ,  $\$a1$ ,  $\$a2$ ,  $\$a3$ ,  $\$t0$ ,  $\$t1$ ,  $\$t2$ ,  $\$t3$ ,  $\$t4$ ,  $\$t5$ ,  $\$t6$ ,  $\$t7$ ,  $\$t8$ ,  $\$t9$ ,  $\$k0$  y  $\$k1$ . También son de este tipo los registros de coma flotante  $\$f0$  a  $\$f19$ .

**Registros preservados entre llamadas:** son registros cuyo valor sí se conserva cuando se llama a un procedimiento. El procedimiento invocado tiene la responsabilidad de que, cuando el control se devuelva al procedimiento invocador, el contenido de los registros sea el mismo que antes de realizar la llamada. Para ello, el procedimiento invocado debe guardar todos los registros de este tipo antes de modificarlos, usando el método 2. Los registros enteros de este tipo son  $\$s0$ ,  $\$s1$ ,  $\$s3$ ,  $\$s4$ ,  $\$s5$ ,  $\$s6$ ,  $\$s7$ ,  $\$gp$ ,  $\$sp$ ,  $\$fp$  y  $\$ra$ . También son de este tipo los registros de coma flotante  $\$f20$  a  $\$f31$ .

Por último, los procedimientos necesitan espacio de almacenamiento en memoria para realizar su trabajo (para almacenar variables locales). Este espacio también se reserva en la pila.

### La pila en MIPS

La *pila* es una zona de memoria cuyo uso sigue una serie de normas. Siguiendo estas normas, se consigue que varios procedimientos puedan usarla para:

1. Almacenar valores de registros para conservar su valor cuando se realizan llamadas a procedimientos.
2. Comunicar parámetros y resultados.
3. Conseguir memoria para variables temporales.

El acceso a la pila se hace de tal forma que el último elemento que entra es el primero que sale (*LIFO*, *Last In First Out*). Este modo de funcionar es adecuado para que los procedimientos que se llamen entre sí (e incluso para que un procedimiento que se llame a sí mismo) puedan compartir la pila, siempre y cuando desapilen todo lo que hayan apilado antes de devolver el control al procedimiento invocador.

Las dos posibles operaciones que se pueden realizar sobre una pila son:

**Apilar (*push*):** añade un valor a la pila.

**Desapilar (*pop*):** recupera el último elemento apilado, eliminándolo de la pila.

Llamaremos *cima* de la pila a la posición de memoria de la última palabra (4 bytes) apilada. En MIPS, se utiliza el registro  $\$sp$  para apuntar a la cima actual de la pila (es decir  $\$sp$  siempre apunta al último elemento apilado). Al apilar y desapilar elementos, la cima de la pila cambia. En MIPS, cuando se apilan elementos la cima se mueve a direcciones más bajas ( $\$sp$  se decrementa), mientras que cuando se desapilan la cima sube ( $\$sp$  se incrementa).

Por ejemplo, en la figura 1 se muestra el estado de una pila tras el apilamiento de tres valores A, B y C, en ese orden.

Para apilar un valor, se ajusta  $\$sp$  decrementándolo (normalmente 4 unidades, para hacer sitio para una palabra de 4 bytes) y se escribe el valor en la nueva cima. Por ejemplo, la figura 2a muestra el estado de la pila anterior después de apilar un nuevo valor “D”. Para ello, si el valor “D” estuviera almacenado en el registro  $\$s1$ , se podría haber utilizado el siguiente código:

```
addi $sp, $sp, -4    # Hace sitio en la pila
sw $s1, 0($sp)     # Copia el valor a almacenar en la cima
```

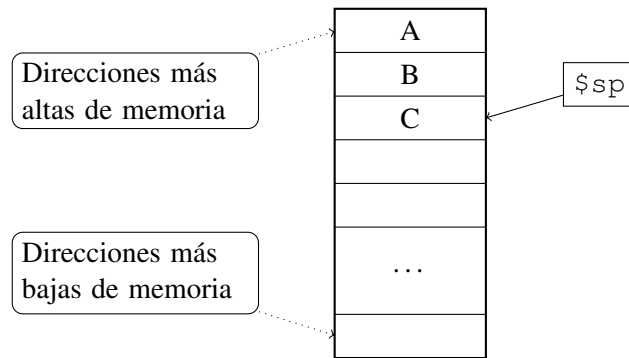
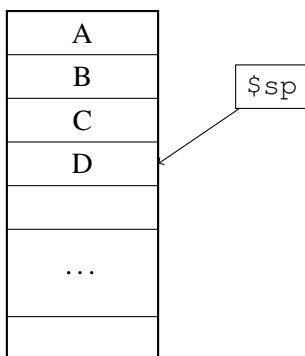
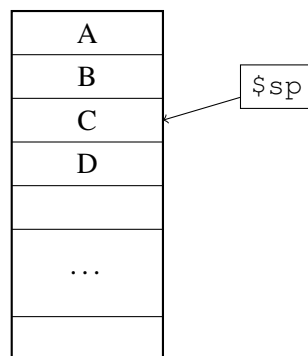


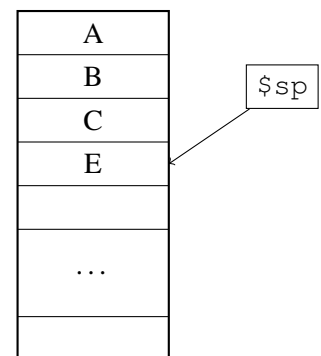
Figura 1: Pila con tres valores apilados (A, B y C).



(a) Pila con cuatro valores apilados (A, B, C y D).



(b) Pila con tres valores apilados (A, B y C). El valor D será sobrescrito cuando se apile E.



(c) Pila con cuatro valores apilados (A, B, C y E).

Figura 2: Evolución de la pila al apilar y desapilar.

Para desapilar un valor, basta con leerlo y ajustar `$sp`, incrementándolo para que apunte a la nueva cima. En la figura 2b se puede ver el estado de la pila después de desapilar “D”. Para ello, se habría utilizado el siguiente código:

```
lw $s1, 0($sp)    # Lee el valor de la cima y lo escribe en $s1
addi $sp, $sp, 4  # Devuelve el espacio a la pila
```

Obsérvese que el valor de D sigue almacenado en memoria después de desapilarlo, pero será sobrescrito cuando se apile un nuevo valor, como se puede ver en la figura 2c.

Además de almacenar el valor de registros, la pila también se puede usar para conseguir memoria para las variables locales de un procedimiento. Para ello, basta con, cuando se entra en el procedimiento, decrementar `$sp` tantos bytes como sea necesario para almacenar las variables, incrementándolo en la misma cantidad antes de abandonar el procedimiento.

Teniendo en cuenta todo lo visto, el siguiente procedimiento:

```
int ejemplo2(int a) {
    int x = a + 7;
    int y = a;
    int p = ejemplo1(x, y, 2, 3);
    return p + y;
}
```

se podría traducir así:

```
ejemplo2:          # etiqueta para llamar al procedimiento
    addi $sp, $sp, -8 # hace sitio en la pila para guardar 2 registros
    sw $s0, 4($sp)   # apila $s0
    sw $ra, 0($sp)   # apila $ra
    addi $t0, $a0, 7  # calcula x en $t0
    add $s0, $0, $a0  # calcula y en $s0
    add $a0, $0, $t0  # primer argumento para llamar a ejemplo1
    add $a1, $0, $s0  # segundo argumento para llamar a ejemplo1
    addi $a2, $0, 2   # tercer argumento para llamar a ejemplo1
    addi $a3, $0, 3   # cuarto argumento para llamar a ejemplo1
    jal ejemplo1     # llamada a ejemplo1, p queda en $v0
    add $v0, $v0, $s0 # suma p e y, calculando el resultado final en $v0
    lw $ra, 0($sp)   # recupera el valor de $ra
    lw $s0, 4($sp)   # recupera el valor de $s0
    addi $sp, $sp, 8 # devuelve el espacio usado a la pila
    jr $ra           # vuelve al invocador
```

Obsérvese cómo se utiliza la pila para guardar los registros *preservados entre llamadas* que son modificados por `ejemplo2` y cómo se recupera su valor anterior antes de la instrucción `jr`. Se guarda el registro `$ra`, ya que se modifica en la línea 11 (instrucción `jal`), y el registro `$s0` que se modifica en la línea 4.

Se ha asignado la variable `x` al registro `$t0` y la variable `y` al `$s0`. En general, lo más común es asignar las variables a los registros temporales no preservados entre llamadas para evitar tener que almacenar su valor en la pila, tal como se ha hecho con la variable `x`. Sin embargo, la variable `y` se ha asignado a un registro preservado entre llamadas (`$s0`) para poder utilizar su valor después de la llamada a `ejemplo1`. Si se hubiera asignado la variable `y` a un registro no preservado entre llamadas como `$t1`, sería necesario almacenarlo en la pila antes de llamar a `ejemplo1`, así:

```

ejemplo2:          # etiqueta para llamar al procedimiento
    addi $sp,$sp,-8 # hace sitio en la pila para guardar 2 registros
    sw $ra,4($sp)   # apila $ra
    addi $t0,$a0,7  # calcula x en $t0
    add $t1,$0,$a0  # calcula y en $t1
    add $a0,$0,$t0  # primer argumento para llamar a ejemplo1
    add $a1,$0,$s0  # segundo argumento para llamar a ejemplo1
    addi $a2,$0,2   # tercer argumento para llamar a ejemplo1
    addi $a3,$0,3   # cuarto argumento para llamar a ejemplo1
    sw $t1,0($sp)   # apila $t1
    jal ejemplo1    # llamada a ejemplo1, p queda en $v0
    lw $t1,0($sp)   # desapila $t1
    add $v0,$v0,$t1 # suma p e y, calculando el resultado final en $v0
    lw $ra,4($sp)   # recupera el valor de $ra
    addi $sp,$sp,8  # devuelve el espacio usado a la pila
    jr $ra          # vuelve al invocador

```

De esta forma no es necesario apilar \$s0 (porque no se usa) pero es necesario apilar \$t1 para conservar su valor después del `jal`, con lo que el número de instrucciones y de accesos a memoria permanece igual. Una tercera opción sería asignar las dos variables a dos registros preservados entre llamadas (x a \$s0 e y a \$s1, por ejemplo), pero en ese caso habría que realizar más accesos a memoria para apilar más registros:

```

ejemplo2:          # etiqueta para llamar al procedimiento
    addi $sp,$sp,-12 # hace sitio en la pila para guardar 3 registros
    sw $ra,8($sp)   # apila $ra
    sw $s0,4($sp)   # apila $s0
    sw $s1,0($sp)   # apila $s1
    addi $s0,$a0,7  # calcula x en $s0
    add $s1,$0,$a0  # calcula y en $s1
    add $a0,$0,$s0  # primer argumento para llamar a ejemplo1
    add $a1,$0,$s1  # segundo argumento para llamar a ejemplo1
    addi $a2,$0,2   # tercer argumento para llamar a ejemplo1
    addi $a3,$0,3   # cuarto argumento para llamar a ejemplo1
    jal ejemplo1    # llamada a ejemplo1, p queda en $v0
    add $v0,$v0,$s1 # suma p e y, calculando el resultado final en $v0
    lw $s1,0($sp)   # recupera el valor de $s1
    lw $s0,4($sp)   # recupera el valor de $s0
    lw $ra,8($sp)   # recupera el valor de $ra
    addi $sp,$sp,12 # devuelve el espacio usado a la pila
    jr $ra          # vuelve al invocador

```

### 3.4.5. Instrucciones de coma flotante

Hasta ahora todas las instrucciones que hemos visto trabajan con valores enteros, y por ello utilizan los 32 registros de uso general, \$0 a \$31. Pero MIPS también permite trabajar con datos en punto flotante. Para ello, ofrece un conjunto adicional de instrucciones que trabajan con 32 registros adicionales que se llaman \$f0 a \$f31, y se encuentran en una zona del procesador llamada *coprocesador 1*. Cada uno de esos registros permite almacenar 32 bits.

MIPS trabaja con el estándar IEEE-754 para representación de números en coma flotante, por lo que cada uno de estos registros puede contener un dato en simple precisión. Alternativamente, estos registros también

se utilizan por pares ( $\$f0$  y  $\$f1$ ,  $\$f2$  y  $\$f3$ ,  $\$f4$  y  $\$f5$ ...) en algunas instrucciones para contener valores en doble precisión (64 bits). En este último caso, el nombre del registro es el correspondiente al número par. Es decir, se puede disponer de 16 registros de 64 bits llamados  $\$f0$ ,  $\$f2$ ,  $\$f4$ ...

A la hora de pasar valores en coma flotante entre procedimientos, se utilizan los registros  $\$f12$ ,  $\$f13$ ,  $\$f14$  y  $\$f15$  para pasar argumentos a un procedimiento y los registros  $\$f0$  y  $\$f1$  para devolver los resultados. Los registros  $\$f20$  al  $\$f31$  se consideran *preservados entre llamadas* (ver sección 3.4.4).

Al igual que ocurría con las instrucciones enteras, existen instrucciones para cargar y almacenar datos en punto flotante, para operar aritméticamente y para realizar comparaciones. A continuación se muestran algunos ejemplos de cada tipo.

### Conversiones de tipo de datos

El banco de registros del coprocesador 1, también llamado banco de registros en coma flotante (o punto flotante), contiene 32 registros de 32 bits. A pesar de su nombre, estos registros pueden contener un dato de cualquier tipo, al igual que los 32 registros de propósito general del banco principal de registros.

En ensamblador, cada instrucción puede interpretar la misma secuencia de bits de diferente forma. En realidad, la diferencia principal<sup>14</sup> entre una instrucción de suma de enteros y una instrucción de suma en punto flotante de simple precisión es la forma en la que interpretan las dos secuencias de 32 bits con las que operan. En el primer caso, cada secuencia se interpreta como un número entero codificado en complemento a 2. En el segundo caso, cada secuencia se interpreta como un número en coma flotante representado mediante el estándar IEEE-754.

Muy frecuentemente es necesario convertir un dato representado en complemento a 2 a su equivalente representado en coma flotante mediante el estándar IEEE-754. Para ello, MIPS proporciona las instrucciones `cvt.s.w` y `cvt.d.w`:

```
cvt.s.w $f0, $f1      # Pone en $f0 la representación en IEEE-754 de
                       # simple precisión del entero en complemento
                       # a 2 almacenado en $f1.
cvt.d.w $f0, $f2      # Pone en $f0 y $f1 la representación en
                       # IEEE-754 de doble precisión del entero en
                       # complemento a 2 almacenado en $f1.
```

Para realizar la conversión en sentido contrario se proporciona más variedad de instrucciones que se diferencian entre sí en la forma en la que realizan el redondeo. Estas instrucciones son:

```
cvt.w.s $f0, $f1      # Pone en $f0 la representación en complemento
                       # a 2 del entero resultado de truncar el
                       # valor en coma flotante de simple precisión
                       # IEEE-754 almacenado en $f1
cvt.w.d $f0, $f2      # Pone en $f0 la representación en complemento
                       # a 2 del entero resultado de truncar el
                       # valor en coma flotante de doble precisión
                       # IEEE-754 almacenado en $f2 y $f3
trunc.w.s $f0, $f1     # Pone en $f0 la representación en complemento
                       # a 2 del entero resultado de truncar el
                       # valor en coma flotante de simple precisión
                       # IEEE-754 almacenado en $f1
trunc.w.d $f0, $f2     # Pone en $f0 la representación en complemento
```

<sup>14</sup>Por supuesto, hay más diferencias. Por ejemplo, cada una de las instrucciones mencionadas obtiene el par de secuencias de 32 bits con el que opera de bancos de registros diferentes.

```

# a 2 del entero resultado de truncar el
# valor en coma flotante de doble precisión
# IEEE-754 almacenado en $f2 y $f3
round.w.s $f0, $f1 # Pone en $f0 la representación en complemento
# a 2 del entero más cercano al valor en
# coma flotante de simple precisión IEEE-754
# almacenado en $f1
round.w.d $f0, $f2 # Pone en $f0 la representación en complemento
# a 2 del entero más cercano al valor en
# coma flotante de doble precisión IEEE-754
# almacenado en $f2 y $f3
ceil.w.s $f0, $f1 # Pone en $f0 la representación en complemento
# a 2 del menor entero mayor o igual al valor
# en coma flotante de simple precisión
# IEEE-754 almacenado en $f1
ceil.w.d $f0, $f2 # Pone en $f0 la representación en complemento
# a 2 del menor entero mayor o igual al valor
# en coma flotante de doble precisión
# IEEE-754 almacenado en $f2 y $f3
floor.w.s $f0, $f1 # Pone en $f0 la representación en complemento
# a 2 del mayor entero menor o igual al valor
# en coma flotante de simple precisión
# IEEE-754 almacenado en $f1
floor.w.d $f0, $f2 # Pone en $f0 la representación en complemento
# a 2 del mayor entero menor o igual al valor
# en coma flotante de doble precisión
# IEEE-754 almacenado en $f2 y $f3

```

La diferencia entre el comportamiento de las instrucciones anteriores se muestra en la tabla 3 (`cvt.w.s` y `cvt.w.d` se comportan exactamente igual que `trunc.w.s` y `trunc.w.d`, respectivamente).

Tabla 3: Comparación de las instrucciones de conversión de coma flotante a entero ofrecidas por MIPS.

Coma flotante	<code>cvt.w.s</code>	<code>trunc.w.s</code>	<code>round.w.s</code>	<code>ceil.w.s</code>	<code>floor.w.s</code>
-3.7	-3	-3	-4	-3	-4
-3.5	-3	-3	-4	-3	-4
-3.3	-3	-3	-3	-3	-4
-3.0	-3	-3	-3	-3	-3
3.0	3	3	3	3	3
3.3	3	3	3	4	3
3.5	3	3	4	4	3
3.7	3	3	4	4	3

### Movimiento de datos entre bancos de registros

Frecuentemente es necesario llevar un dato del banco de registros de enteros al banco de registros de coma flotante para operar con él, después de realizar las conversiones de tipo correspondientes. Para ello, se dispone de las instrucciones `mfcl` y `mtcl`:

```

mfc1    $t1, $f1      # Copia el contenido del registro $f1 a $t1
mtc1    $t1, $f1      # Copia el contenido del registro $t1 a $f1
mfc1.d  $t1, $f1      # Copia el contenido del registro $f1 a $t1
          # y el contenido de $f2 a $t2
mtc1.d  $t1, $f1      # Copia el contenido del registro $t1 a $f1
          # y el contenido de $t2 a $f2

```

Obsérvese el orden inusual de los operandos de las instrucciones `mfc1` y `mfc1.d`. Las instrucciones `mfc1.d` y `mtc1.d` son pseudoinstrucciones que se traducen en pares de las otras dos instrucciones. Estas instrucciones copian los 32 (o 64) bits de un registro a otro sin realizar ninguna transformación, por lo que habitualmente irán acompañadas de instrucciones de conversión como las descritas en la sección 3.4.5. Por ejemplo, el siguiente fragmento de código almacena en `$a0` el valor entero más aproximado del área de un círculo cuyo radio está almacenado como entero en la dirección de memoria apuntada por `$t0`, suponiendo que la constante  $\pi$  está almacenada como valor en coma flotante de simple precisión en la dirección de memoria apuntada por la etiqueta `pi`.

```

l.s     $f12, ($t0)    # Carga en $f12 el valor entero del radio
cvt.s.w $f12, $f12    # Lo convierte a coma flotante
l.s     $f13, pi      # Carga en $f13 el valor de  $\pi$ 
mul.s   $f12, $f12, $f12 # Calcula el cuadrado del radio
mul.s   $f12, $f13, $f12 # Lo multiplica por  $\pi$ 
round.w.s $f12, $f12  # Lo redondea al entero más cercano
mfc1    $a0, $f12     # Lo mueve al banco de registros de enteros

```

### Carga y almacenamiento

Las instrucciones de carga y almacenamiento de valores en coma flotante son similares a las instrucciones descritas en la sección 3.4.2. La principal diferencia es que el registro que se utiliza para el valor cargado o almacenado será un registro de coma flotante.

Un ejemplo de instrucción de carga de un valor en simple precisión sería el siguiente:

```

l.s $f4, 0x100c($29)    # Carga $f4 con la palabra de la
                        # dirección $29+0x100c.

```

Como vemos, la instrucción utiliza el mismo modo de direccionamiento que `lw`, una base más un desplazamiento inmediato de 16 bits para obtener la dirección destino. El valor cargado de memoria se copia al registro sin ningún tipo de modificación. Esto quiere decir que si el valor almacenado en memoria no estaba ya en formato IEEE-754 deberá ser convertido antes de operar con él (ver sección 3.4.5).

De modo análogo funciona la instrucción de almacenamiento `s.s`:

```

s.s $f4, 0x100c($29)    # Almacena $f4 en la dirección
                        # $29+0x100c.

```

Para valores en doble precisión, los mnemónicos correspondientes son `l.d` y `s.d`. Estas instrucciones acceden a 8 bytes (64 bits) de memoria y a dos registros de coma flotante (el indicado directamente en la instrucción y el siguiente). Por ejemplo:

```

l.d $f4, 0x100c($29)    # Carga ($f4, $f5) con los 8 bytes a
                        # partir de la dirección $29+0x100c.
s.d $f4, 0x100c($29)    # Almacena ($f4, $f5) en 8 bytes a
                        # partir de la dirección $29+0x100c.

```

En realidad, `l.s`, `l.d`, `s.s` y `s.d` son *pseudoinstrucciones* (ver sección 3.7.4), que se traducen a secuencias de las instrucciones reales `lwc1` y `swc1` (carga y almacenamiento del *coprocesador 1*). Así, por ejemplo, una instrucción `l.s $f4, 0x100c($29)` se traducirá como:

```
lwc1 $f4, 0x100c($29)
```

Y una instrucción `l.d $f4, 0x100c($29)` como el par de instrucciones:

```
lwc1 $f4, 0x100c($29)
lwc1 $f5, 0x1010($29)
```

### Aritméticas

Las instrucciones básicas (suma, resta, multiplicación y división) se encuentran también tanto para simple como para doble precisión. Por ejemplo, para la suma:

```
add.s $f4, $f5, $f6      # $f4 = $f5 + $f6 (simple precisión)
add.d $f4, $f6, $f8      # ($f4, $f5) = ($f6, $f7) + ($f8, $f9)
                          # (doble precisión)
```

De modo análogo a las instrucciones de suma funcionan las instrucciones de resta (`sub.s` y `sub.d`), multiplicación (`mul.s` y `mul.d`), y división (`div.s` y `div.d`).

### Comparación y salto

Las instrucciones para comparar datos en punto flotante en MIPS tienen un funcionamiento ligeramente distinto a las comparaciones con enteros. En aquellas, las instrucciones como `slt` utilizaban un registro general explícito de la arquitectura para colocar un 0 o un 1 con el resultado de la comparación. Sin embargo, en las instrucciones de punto flotante, se utiliza un registro especial de 1 bit en el coprocesador 1. Este registro (*flag* o *bit de bandera*) almacena el resultado de la última comparación realizada sobre valores en coma flotante.

Todas las instrucciones de comparación disponibles son del tipo `c.X.s` y `c.X.d` (donde *X* puede ser `eq`, `neq`, `lt`, `le`, `gt` o `ge` para comparaciones del tipo `=`, `≠`, `<`, `≤`, `>`, `≥`, respectivamente). De esta manera tenemos:

```
c.X.s $fi, $fj          # Compara los datos en simple precisión
                        # $fi y $fj, activando o desactivando el
                        # flag del coprocesador 1 dependiendo de
                        # si se cumple o no la condición X.
c.X.d $fi, $fj          # Idem, para datos en doble precisión.
```

Así, después de realizarse una comparación, se utilizará una instrucción de salto condicional que comprobará si dicho *flag* fue o desactivado por dicha comparación. Las correspondientes instrucciones de salto condicional, que comprueban el valor del *flag* para realizar el salto, son:

```
bclt      # Salta si la última comparación fue cierta
bc1f      # Salta si la última comparación fue falsa
```

### 3.5. Manejo de interrupciones y excepciones

Durante la mayoría del tiempo, un computador ejecuta instrucciones en el mismo orden en el que están almacenadas en memoria, o **siguiendo el orden indicado por el programador** mediante instrucciones de salto (ver sección 3.4.3). Esta forma de funcionar es sencilla de comprender y es suficiente para la mayoría de las tareas, pero no permite que el computador pueda reaccionar a eventos externos (como la pulsación de una tecla) o situaciones inesperadas (como la aparición de una instrucción incorrecta).

Las *excepciones e interrupciones* son eventos que modifican el flujo de ejecución de instrucciones de forma *asíncrona*. Ambos conceptos son muy similares<sup>15</sup>, pero haremos la siguiente distinción:

**Interrupción:** suceso cuyo origen es *externo al procesador* que requiere la atención de éste. Por ejemplo: el movimiento del ratón podría activar una señal que indicaría al procesador que necesita interrumpir momentáneamente la ejecución del programa actual para leer la nueva posición del ratón y actualizar el cursor en la pantalla.

**Excepción:** suceso cuyo origen es *interno al procesador* y que requiere la modificación del flujo de ejecución del programa. Además de las excepciones por desbordamiento que se mencionaron en la sección 3.4.1, existen otras causas que también pueden requerir que el procesador detenga momentáneamente su funcionamiento para atender una situación excepcional, producida durante la ejecución del programa. Algunos ejemplos serían el intento de acceso a una dirección de memoria no válida (no perteneciente al *espacio de direcciones* del programa), el intento de ejecución de una instrucción inexistente (con un código inválido, por ejemplo), etc.

Cuando se produce una interrupción o una excepción, se transfiere el control a un procedimiento especial denominado *manejador de interrupción* (o *manejador de excepción* en su caso).

En MIPS, el proceso que se sigue cuando se produce una interrupción o una excepción es el siguiente:

1. Se almacena en el registro especial EPC (*Exception Program Counter*, contador de programa en la excepción) la dirección de la instrucción que ha producido la excepción (en el caso de las excepciones) o la de la siguiente instrucción a ejecutar (en el caso de las interrupciones). Este registro podrá ser utilizado por el manejador correspondiente para analizar la causa de la excepción y para devolver el control al programa una vez tratada la excepción o interrupción.
2. El programa salta a una subrutina almacenada en una dirección predefinida donde se tratará la interrupción o excepción. Como se ha mencionado, esta subrutina se denomina *manejador de interrupción* o *manejador de excepción* y es parte del sistema operativo. Puede haber un manejador disponible para cada tipo de interrupción o excepción, o un manejador común a todos los tipos. En este último caso, se almacena la causa de la excepción o interrupción en el registro especial CAUSE. En el caso de que la excepción sea debida a un acceso a memoria, se almacena la dirección implicada en el registro especial VADDR (por ejemplo, para un acceso a memoria no alineado o un fallo de página).
3. El manejador realiza las acciones apropiadas según la excepción o interrupción particular.
4. En los casos que sea posible, una vez tratada la excepción se continúa con la ejecución normal del programa, posiblemente notificándole de la excepción o interrupción mediante un mecanismo que depende del sistema operativo. En otras ocasiones, el sistema operativo aborta la ejecución del programa.

Un manejador de interrupciones o excepciones es, habitualmente, parte del sistema operativo. Estos procedimientos son especiales porque pueden ser llamados desde cualquier punto del programa (sin que el programador o compilador pueda preverlo). Por tanto, entre otras cosas, usarán un convenio de uso de registros

<sup>15</sup>De hecho, en la documentación de muchas arquitecturas se utiliza diferente terminología a la que aquí se presenta, o incluso no hace distinción entre ambos conceptos.

diferente. Para que sea posible devolverle el control al programa en el paso 4, es necesario que durante el paso 3 no se modifique el contenido de los registros que usa el programa<sup>16</sup>. Para realizar su trabajo, los manejadores de interrupción en MIPS tienen reservados los registros `$k0` y `$k1`.

En el paso 4, el manejador puede decidir terminar el proceso causante de la excepción o informarle del problema (por ejemplo, en POSIX, mediante una señal) para que el programador pueda decidir cómo tratar la excepción (por ejemplo, mostrando un mensaje de error). Para hacer esto posible, MIPS ofrece la instrucción `mfc0` (*move from coprocessor 0*<sup>17</sup>) que permite copiar el registro EPC en un registro normal, para poder después continuar con la ejecución del programa usando la instrucción `jr`. También se dispone de la instrucción `eret`, que copia el contenido del registro EPC en el registro PC y reactiva el procesamiento de posteriores interrupciones.

Las interrupciones juegan un papel fundamental en el manejo de la entrada y salida, como se explicará en temas posteriores de esta asignatura.

### 3.6. Llamadas al sistema operativo

La mayoría de los computadores utilizan un sistema operativo para, entre otras cosas, ofrecer a los programas que se ejecutan en la máquina determinados servicios que requieren el acceso al hardware a bajo nivel. Por ejemplo, como se verá en posteriores temas de esta asignatura, la implementación de las operaciones de entrada y salida requiere código que manipula direcciones concretas de memoria o que se encarga de atender determinadas interrupciones. Ese código es parte del sistema operativo y éste pone esta funcionalidad a disposición del resto de programas mediante determinadas *llamadas al sistema*. De esta forma, el programa de usuario no tiene que conocer los detalles concretos del hardware<sup>18</sup>.

En general, el sistema operativo y el resto de programas que se ejecutan en la máquina se mantienen estrictamente separados entre sí. Esta separación hace posible que varios programas se puedan ejecutar en el mismo computador y se consigue, en la mayoría de los casos, mediante modos diferentes de funcionamiento del procesador<sup>19</sup> y el uso espacios de direcciones virtuales diferentes como se verá en temas posteriores. Debido a esta separación entre el sistema operativo y los programas, un programa no puede llamar a un procedimiento del sistema operativo usando los mecanismos habituales que se han descrito en la sección 3.4.4.

Por tanto, es necesario un mecanismo diferente. MIPS ofrece para este fin la instrucción `syscall`. Cuando se ejecuta, esta instrucción genera una excepción (ver sección 3.5) que se utiliza para transferir el control al sistema operativo. El sistema operativo realiza las acciones asociadas a la llamada al sistema solicitada y devuelve el control al sistema operativo.

Una llamada al sistema se identifica mediante un número que se escribe en el registro `$v0` y se invoca mediante la instrucción `syscall`. La tabla 4 muestra una selección de algunas de las llamadas al sistema ofrecidas en el entorno del simulador MARS. De igual forma que los procedimientos normales, las llamadas al sistema reciben sus argumentos en los registros `$a0`, `$a1`, `$a2` y `$a3` (excepto las que trabajan con números en coma flotante) y devuelven sus resultados en `$v0` y `$v1`. Sin embargo, a diferencia del caso de los procedimientos normales, no es necesario guardar los registros temporales en la pila antes de realizar una llamada al sistema.

Por ejemplo, el siguiente fragmento de código lee un entero, le suma 1 y a continuación lo escribe:

```
li $v0,5          # Código de la llamada al sistema read int
```

<sup>16</sup>Por supuesto, el manejador puede usar cualquier registro si almacena primero su valor y lo restaura antes del paso 4

<sup>17</sup>Se llama así porque el registro EPC se encuentra en una parte del procesador a la que se le llama *coprocesador 0*

<sup>18</sup>De hecho, la mayoría de los sistemas operativos no permiten que un programa realice estas acciones directamente.

<sup>19</sup>Normalmente, existe un modo de funcionamiento de *usuario*, usado por la mayoría de los programas, y un modo de funcionamiento *privilegiado*, usado por el sistema operativo. El modo de usuario no tiene acceso a algunas instrucciones (o direcciones de memoria) que sí están disponibles en el modo privilegiado y que son necesarias sólo para implementar la funcionalidad del sistema operativo.

Tabla 4: Ejemplos de llamadas al sistema ofrecidas por el simulador MARS.

Código	Nombre	Argumentos	Resultado
1	print integer	Número a imprimir en $\$a0$ .	Imprime el entero.
2	print float	Número a imprimir en $\$f12$ .	Imprime el número en coma flotante.
3	print double	Número a imprimir en $\$f12$ .	Imprime el número en coma flotante.
4	print string	Dirección de la cadena a imprimir en $\$a0$ . La cadena debe acabar en 0.	Imprime la cadena.
5	read integer		Espera a que el usuario introduzca un entero mediante un cuadro de diálogo. Lo almacena en $\$v0$ .
6	read float		Espera a que el usuario introduzca un número en coma flotante mediante un cuadro de diálogo. Lo almacena con precisión simple en $\$f0$ .
7	read double		Espera a que el usuario introduzca un número en coma flotante mediante un cuadro de diálogo. Lo almacena con precisión doble en $\$f0$ y $\$f1$ .
8	read string	Dirección del buffer en el que almacenar la cadena leída en $\$a0$ y tamaño máximo de la cadena a leer en $\$a1$ .	
9	sbrk	Número de bytes de memoria a reservar en $\$a0$	Reserva memoria en el <i>heap</i> (montículo) y devuelve la dirección en $\$v0$ .
10	exit		Termina la ejecución del programa.

```

syscall      # Lee el entero y coloca el resultado en $v0
addi $a0,$v0,1 # Suma 1 al valor leído y pone el resultado en $a0
li $v0,1      # Código de la llamada al sistema print int
syscall      # Imprime el entero almacenado en $a0

```

### 3.7. Codificación de las instrucciones en MIPS

En esta sección se verá cómo se representan las instrucciones del ensamblador dentro del computador, para que éste las pueda leer, decodificar y ejecutar. En realidad, al igual que los datos, las instrucciones se representarán en el computador como simples series de ceros y unos. Para interpretarlas, estas series se dividirán en *campos*, cada uno de los cuales servirá para almacenar una información necesaria para la ejecución de la instrucción.

Por ejemplo, una instrucción `slt $t2, $t0, $t1`, una vez codificada, tendrá que tener un campo con

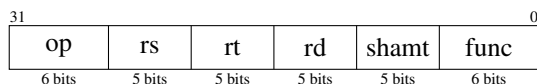
un valor determinado para indicar que se trata de un `slt` (y no un `add`, o un `lw`, etc). Asimismo, tendrá que tener dos campos para indicar que los registros a comparar son `$t0` y `$t1`, y otro campo para indicar que el registro destino (donde escribir la constante 1 si se cumple la condición, o 0 en caso contrario) es `$t2`. A la distribución concreta de estos campos en la secuencia de bits completa para codificar la instrucción se le denominará *formato* de la instrucción.

Siguiendo la filosofía RISC, lo ideal a la hora de decidir cómo codificar las instrucciones sería elegir un formato único para todas las instrucciones, ya que de esta forma se simplifica el hardware necesario para realizar la decodificación. Sin embargo, esta estrategia podría acarrear problemas, tales como obtener instrucciones demasiado largas, con muchos campos sin utilizar (por ejemplo, en MIPS algunas instrucciones necesitan codificar una constante de hasta 16 bits, mientras que otras no lo necesitan y tendrían el correspondiente campo vacío). En MIPS, como solución de compromiso, existen unos pocos formatos de instrucciones<sup>20</sup> de 32 bits todos ellos. A la hora de decidir qué formato utilizar para cada instrucción, el factor determinante es qué modos de direccionamiento (ver sección 3.3) usa dicha instrucción.

A continuación se pasan a describir los tres formatos de instrucciones (formatos R, I y J) disponibles en la arquitectura MIPS, con ejemplos de cómo son utilizados por los diferentes tipos de instrucciones, acomodándose en cada caso al modo de direccionamiento empleado.

### 3.7.1. Formato de instrucción R

En este formato de instrucción, los operandos fuente y destino se especifican usando el *modo de direccionamiento registro*. El modo concreto de distribuir los 32 bits de la instrucción es el siguiente (el bit 31, de más peso, se encuentra en el extremo izquierdo):



El significado de los distintos campos es el siguiente:

**op:** indica el código de operación. Se utiliza para diferenciar los distintos tipos de instrucciones. Este campo es el único que aparece en todos los formatos de instrucción (no sólo en las de tipo R, sino también las de tipo I y J), puesto que sólo una vez que se lee, y se sabe el tipo de instrucción de la que se trata, la unidad de decodificación de la instrucción podrá conocer el formato real que tiene la instrucción.

**rs:** primer registro fuente de la operación.

**rt:** segundo registro fuente de la operación.

**rd:** registro del operando destino, donde se guardará el resultado de la instrucción.

**shamt:** tamaño del desplazamiento. Sólo se utiliza para almacenar la cantidad de desplazamiento en las instrucciones de desplazamiento o rotación (`sll`, `srl...`), y vale 0 en el resto.

**func:** sirve para distinguir instrucciones que, por ser muy similares, tienen el mismo código de operación (por ejemplo, todas las aritmético-lógicas que trabajan con tres registros). Le indica a la ALU qué función de las posibles (suma, resta, AND lógico...) debe realizar.

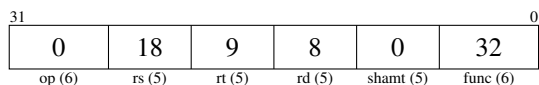
El formato R se utiliza en las instrucciones aritmético-lógicas que operan sobre dos registros fuente para colocar el resultado en un determinado operando destino. Por ejemplo, la instrucción

<sup>20</sup>Otra solución posible es la adoptada por la arquitectura x86 (IA-32) o la amd64 (IA-64). En estas arquitecturas, las instrucciones tienen longitud variable y la decodificación es un proceso más complejo. De hecho, implementaciones modernas de estas arquitecturas funcionan traduciendo las instrucciones x86 a un formato interno más regular antes de ejecutarlas.

Esta opción, sin embargo, es contraria a la filosofía RISC seguida por MIPS, una de cuyas características principales es la uniformidad en el tamaño de sus instrucciones.

```
add $8, $18, $9
```

se representaría de la siguiente manera:

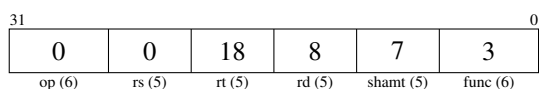


El código de operación **op** indica que se trata de una instrucción aritmético-lógica (**op**=0) con dos registros fuente (**rs**=18, **rt**=9) y un operando destino (**rd**=8). El campo **shamt** no se utiliza, puesto que no es una instrucción de desplazamiento, y se deja a 0. Finalmente, el campo **func** determina el tipo concreto de operación, en este caso una suma (**func**=32). Por tanto, esta instrucción estará almacenada en memoria como la ristra de bits 0000 0010 0100 1001 0100 0000 0010 0000 (0x02494020, en hexadecimal).

De forma similar, la instrucción

```
sra $8, $18, 7
```

se representaría de la siguiente manera:



En este caso, el campo **func** indica que se trata de un desplazamiento hacia la derecha, y el campo **shamt** contiene el número de bits a desplazar. El registro a desplazar se encuentra codificado en el campo **rt**, mientras que el campo **rs** no se utiliza y se deja a cero.

### 3.7.2. Formato de instrucción I

Este formato se caracteriza por tener un campo dedicado a almacenar un valor inmediato de 16 bits. Se utiliza para codificar las instrucciones en las que uno de los operandos utiliza el *direccionamiento inmediato*, el *direccionamiento base más desplazamiento* o el *direccionamiento relativo al contador del programa*. Es decir, será utilizado por instrucciones aritméticas cuando uno de los operandos es un valor constante, instrucciones de transferencia de datos y saltos condicionales. La distribución de campos es la siguiente:



**op:** código de operación.

**rs:** registro fuente.

**rt:** registro fuente/destino dependiendo de la operación.

**imm:** valor inmediato. Su interpretación depende de la operación.

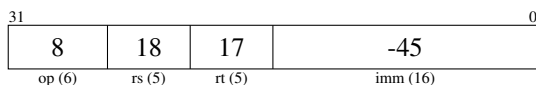
A continuación mostramos como codificar los diferentes tipos de instrucción que utilizan este formato:

#### Instrucciones aritméticas con operandos constantes

Estas instrucciones utilizan el campo **imm** para codificar en complemento a dos el valor del operando. El campo **rs** codifica el otro operando fuente y el campo **rt** codifica el registro de destino. Por ejemplo, la instrucción:

```
addi $17, $18, -45
```

se codifica:

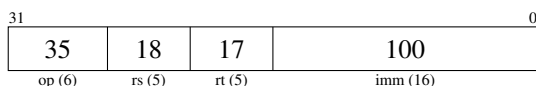


### Instrucciones de acceso a memoria

Las instrucciones de carga y almacenamiento utilizan el campo **imm** para codificar en complemento a dos el desplazamiento que hay que sumarle al contenido del registro base, codificado en **rs**. El campo **rt** codifica el registro fuente o destino, según sea una instrucción de carga o almacenamiento, respectivamente. Por ejemplo, la instrucción:

```
lw $17, 100($18)
```

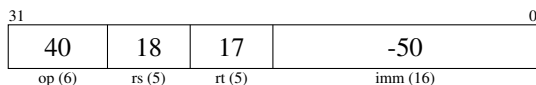
se codifica:



Las instrucciones de almacenamiento se codificará de modo análogo, por ejemplo:

```
sb $17, -50($18)
```

se codifica:



### Instrucciones de salto condicional

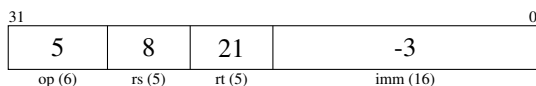
En este caso, el campo **imm** se utiliza para codificar en complemento a dos el número de instrucciones<sup>21</sup> a avanzar, empezando en la instrucción siguiente al salto para llegar a la instrucción de destino (si hubiera que retroceder, se codificaría un número negativo).

Cuando se ejecuta la instrucción, se utiliza el contenido del campo **imm** y el valor del contador de programa (PC) para calcular la dirección de memoria de la instrucción de destino. En MIPS la memoria se direcciona por bytes, pero, como sabemos, cada instrucción ocupa 4 de ellos. Por tanto, una dirección de una instrucción MIPS es siempre múltiplo de 4 (los dos últimos bits siempre valdrán 00). Por esta razón no es necesario codificar en **imm** el número de bytes que hay que sumarle al PC, y se consigue abarcar más instrucciones de destino con el campo inmediato de sólo 16 bits

Por ejemplo, la instrucción `bne` en la siguiente secuencia:

```
l1: sub $t0, $t2, $t3
    addi $s5, $s5, 1
    bne $t0, $s5, l1
    ...
```

se codificaría:



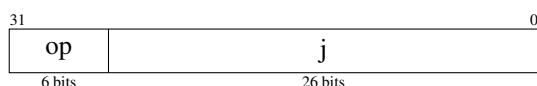
<sup>21</sup>Por supuesto, es el número de instrucciones reales de MIPS, no pseudoinstrucciones.

Como se puede observar, la dirección apuntada por la etiqueta se encuentra dos instrucciones antes de `bne`. Además, hay que tener en cuenta que el desplazamiento es relativo al valor del contador de programa en el momento de ejecutar la instrucción `bne` (que es  $PC + 4$ , es decir, la dirección de memoria de la instrucción siguiente a `bne`). Por tanto, el valor almacenado en el campo **imm** de la instrucción es  $-3$  (como siempre, en complemento a dos).

### 3.7.3. Formato de instrucción J

Este último tipo de formato se usa principalmente para las instrucciones de salto incondicional, que usan el *direccionamiento pseudodirecto*. Estas instrucciones especifican casi directamente la dirección destino del salto, al contrario que los saltos condicionales, donde la dirección destino se codifica con respecto a la dirección de la instrucción actual.

Al tener que especificarse directamente la dirección destino, los 16 bits del campo inmediato en el formato I son insuficientes, por lo que se crea un nuevo formato. Es imposible codificar la dirección completa de la instrucción (que tiene 32 bits) de destino junto con el código de operación porque sólo hay 32 bits en cada instrucción. Por tanto, en este formato se reservan todos los bits que no son del código de instrucción (26) para la dirección, y se aprovecha de nuevo el que todas las instrucciones estén almacenadas en direcciones múltiplo de 4. Los campos son los siguientes:



**op:** especifica el código de operación.

**j:** almacena los bits del 27 al 2 (inclusive) de la dirección destino.

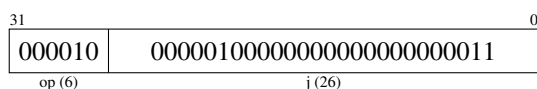
Además de los 26 bits almacenados en el campo **j**, son necesarios 6 bits más para conocer la dirección de la instrucción de destino. Los dos bits menos significativos no es necesario codificarlos, porque serán siempre 0 al ser la dirección múltiplo de 4. Los cuatro bits restantes para formar una dirección de 32 bits se toman de los 4 bits más significativos del PC de la instrucción en curso (la de salto).

En resumen, la dirección de memoria de la instrucción destino se formará (de mayor a menor peso) cogiendo los 4 bits más significativos del PC de la instrucción de salto, más los 26 indicados en el campo **j**, más dos ceros.

A continuación se muestra un ejemplo para ilustrar esto. Supongamos que la etiqueta `bucle` se corresponde con la dirección `0x0040000c`. En ese caso, suponiendo que la propia instrucción `j` se encontrase en una dirección que tuviese sus cuatro bits de más peso como ceros (es decir, fuese de la forma `0x0NNNNNNN`), la codificación de la siguiente instrucción:

```
j bucle
```

sería (en binario):



### 3.7.4. Pseudoinstrucciones

El lenguaje ensamblador contiene instrucciones que no se implementan directamente sobre el hardware del procesador, sino que tienen que ser traducidas por el ensamblador (el programa que traduce de lenguaje ensamblador a lenguaje máquina).

Este tipo de instrucciones (que no tienen correspondencia directa con una instrucción máquina, pero que son traducidas a una secuencia de instrucciones máquina equivalente) se conocen como *pseudoinstrucciones*. Las pseudoinstrucciones simplifican la programación en lenguaje ensamblador, puesto que aumentan la potencia expresiva del mismo sin suponer ningún incremento de complejidad en la construcción del hardware. En otras palabras, las pseudoinstrucciones dan al MIPS un repertorio de instrucciones más rico que el que implementa la circuitería.

Veamos un ejemplo. Como sabemos, la arquitectura MIPS asegura que el registro `$zero` siempre tenga el valor cero. Este registro se utiliza para traducir la instrucción de lenguaje ensamblador *move*, una pseudo-instrucción que copia el contenido de un registro en otro. El ensamblador de MIPS acepta esta instrucción, aunque no se encuentra en la arquitectura MIPS:

```
move $t0,$t1      # Copia $t1 en $t0.
```

Lo que realmente hace el ensamblador es convertir esta instrucción en una equivalente que sí tiene correspondencia directa en lenguaje máquina, valiéndose del registro `$zero`:

```
add $t0,$t1,$zero # Copia $t1 en $t0
```

Algunas pseudoinstrucciones se traducen en más de una instrucción máquina. Y, en algunos casos, estas instrucciones necesitan comunicarse algún valor temporal entre sí. Para ello, MIPS tiene reservado el registro `$at` (`$1`). Un ejemplo sería la pseudoinstrucción

```
blt $s0,$s1,destino # Salta a destino si $s0 < $s1
```

El ensamblador traduce esta pseudoinstrucción en la siguiente secuencia de instrucciones máquina reales, que implementan el comportamiento deseado:

```
slt $at,$s0,$s1    # Activa $at si $s0 < $s1
bne $at,$zero,destino # Realiza el salto condicional
```

Hay todo un repertorio de pseudoinstrucciones para los distintos tipos de saltos, como se mencionó en la sección 3.4.3. Todas ellas se traducirán a una secuencia de como mucho dos instrucciones, una *slt* y otra *beq* o *bne*.

Otro ejemplo típico de pseudoinstrucción es la relacionada con la carga de constantes grandes en un registro. Por ejemplo, supongamos que queremos cargar la constante de 32 bits `0x1234abcd` en el registro `$10`. Para ello, se dispone de la pseudoinstrucción

```
li $10,0x1234abcd # Carga 0x1234abcd en $10.
```

El problema de esta operación es el tamaño de la constante manejada. Como se ha visto en el apartado 3.7.2, dicha constante no puede ser codificada en una sola instrucción, puesto que ocupa 32 bits y las instrucciones MIPS poseen esa misma longitud. Puesto que, además de la constante, será obligatorio codificar también en otros campos otra información de interés (como el código de operación de la instrucción, el propio registro sobre el que se quiere realizar la carga, etc), es obvio que el trabajo requerido tiene que realizarse con más de una instrucción. La solución, pues, pasa por “partir” la constante en dos mitades de 16 bits, las cuales se cargan en dos pasos en el registro destino, primero la mitad superior y luego la inferior. Así, la pseudoinstrucción anterior se traduciría a las dos instrucciones máquina reales:

```
lui $at,0x1234    # Carga la mitad superior (y pone los 16 bits
                  # inferiores a 0) ...
ori $10,$at,0xabcd # ... y carga la mitad inferior.
```

La instrucción *lui* carga la constante en la mitad superior de el registro auxiliar (`$at`), dejando la mitad inferior con ceros, y posteriormente el *ori* completa el valor a colocar en `$10` con los 16 bits inferiores.

### 3.8. Alternativas al ISA de MIPS

Hemos presentado hasta ahora el ISA del MIPS como ejemplo de un ISA moderno típico. Muchos ISAs son similares entre sí, pero existen algunas alternativas al diseño de MIPS que conviene conocer.

El diseño del ISA de un computador tiene implicaciones en la complejidad, el coste y el rendimiento final del mismo. Por ejemplo, un modelo sencillo del tiempo de ejecución de un programa es el siguiente:

$$Tiempo = \text{Numero de instrucciones} \times \frac{\text{ciclos de reloj}}{\text{instruccion}} \times \frac{\text{segundos}}{\text{ciclos de reloj}}$$

Como se ve, el tiempo de ejecución de un programa es proporcional al número de instrucciones ejecutadas<sup>22</sup> del mismo. Por tanto, una forma de reducir el tiempo de ejecución sería diseñar el ISA de tal manera que hicieran falta menos instrucciones para realizar algunas operaciones. Es decir, incluir instrucciones más complejas.

El problema de complicar el repertorio de instrucciones es que, normalmente, conlleva una complicación del hardware del procesador. Este aumento de complejidad no sólo incrementa el coste (ahora hay que invertir más tiempo en diseñar y probar el procesador), sino que también puede hacer aumentar el segundo o el tercer factor de la ecuación (ciclos por instrucción o tiempo de ciclo), ya que las instrucciones más complejas necesitarán un mayor número de ciclos o ciclos más largos para realizar su tarea.

Por tanto, cualquier mejora que se proponga para un ISA debe ser evaluada cuidadosamente para evitar que empeore el rendimiento en lugar de mejorarlo.

Por ejemplo, el *PowerPC* es otro ISA muy extendido que se puede encontrar en consolas y gran variedad de dispositivos multimedia y de telecomunicaciones. Es una arquitectura muy similar a la de MIPS en cuanto a número de registros y formato de las instrucciones. Sin embargo, incorpora dos modos nuevos de direccionamiento que añaden más funcionalidad que la proporcionada por MIPS:

**Modo de direccionamiento indizado:** en este modo, se pueden usar dos registros para especificar una dirección de memoria. Para obtener la dirección, se ha de sumar el contenido de ambos registros. Por ejemplo el siguiente código MIPS

```
add $t0,$a0,$s3      # $a0 es la base de la tabla, $s3 el índice
lw  $t1,0($t0)
```

se podría expresar de la siguiente manera en PowerPC:

```
lw  $t1,$a0+$s3     # $a0 es la base de la tabla, $s3 el índice
```

**Modo de direccionamiento actualizado o autoincremento:** es utilizado en recorridos de tablas, donde se van leyendo todas las posiciones consecutivamente. La siguiente secuencia de instrucciones MIPS:

```
lw  $t0,8($s3)      # $s3 contiene la dirección de la siguiente
add $s3,$s3,4        # posición a leer
```

en PowerPC se haría con una sola instrucción, que, aparte de cargar en \$t0 el contenido de la memoria cuya dirección es \$s3+8, también incrementa \$s3 en 4:

```
lwu $t0,8($s3)      # Carga $t0 de Mem[$s3+8] e incrementa $s3 en
                    # 4 para apuntar a la siguiente posición
```

<sup>22</sup>Esto se refiere al número de veces que se ha ejecutado alguna instrucción, no al número de instrucciones que tiene el código del programa. Es decir, habrá instrucciones que se ejecuten varias veces (por ejemplo, en los bucles) e instrucciones que no se ejecuten ninguna.

Además, el PowerPC dispone de operaciones más potentes relacionadas con la transferencia de datos y el salto condicional al final de los bucles. En concreto, dispone de una instrucción que permite transferir hasta 32 palabras de datos a la vez, y se usa para hacer copias rápidas de posiciones de memoria y para salvar y restaurar registros. También dispone de una instrucción especial, usada al final de bucles, que decrementa el índice del bucle, lo compara con cero y salta, todo ello en una sola instrucción.

Por otro lado, un ISA radicalmente diferente a la de MIPS es la de la familia de procesadores x86 (usada por el Intel 386, el Pentium IV y los procesadores más modernos de 32 bits de Intel y AMD). Las diferencias principales entre MIPS y x86 son las siguientes:

1. Los registros de x86 tienen usos específicos, al contrario que en MIPS, donde son de propósito general.

Aunque es cierto que en MIPS los registros se usan según un convenio acordado por los programadores (es decir, hay algunos destinados al paso de parámetros, otros para variables temporales, otros para la devolución de resultados, etc), esto es sólo una convención, no una limitación real de la arquitectura. Es decir, en realidad, la arquitectura MIPS permite que casi todos los registros puedan ser usados para todas las funciones.

En cambio, x86 limita el uso de los registros, y unos son usados para índices de tablas, otros para cálculos aritméticos, etc. De hecho, x86 sólo dispone de 8 registros de uso general, por 31 MIPS (no contamos ya el registro 0, que sabemos que está cableado al valor cero).

2. Las instrucciones aritmético-lógicas y las de transferencia de datos son de dos operandos. Uno de ellos es a la vez fuente y destino de la operación.
3. Existen modos de direccionamiento adicionales. Por ejemplo:

**Registro indirecto:** la dirección del operando está en un registro.

**Modo base con desplazamiento de 8 ó 32 bits:** la dirección es un registro base más un desplazamiento de 8 ó 32 bits.

**Base más índice escalado:** la dirección del operando es  $base + 2^{escala} \times índice$ , donde *escala* puede ser 0, 1, 2 ó 3.

**Base más índice escalado con desplazamiento de 8 ó 32 bits:** la dirección del operando es  $base + (2^{escala} \times índice) + desplazamiento$ .

Existen restricciones sobre qué registros pueden ser usados en cada modo de direccionamiento (esto entra dentro de la limitación expuesta en el punto 1).

4. Casi todas las instrucciones ofrecen muchas combinaciones de modos de direccionamiento:
  - Registro y registro.
  - Registro e inmediato.
  - Registro y memoria.
  - Memoria y registro.
  - Memoria e inmediato.

Por tanto, en x86 alguno de los operandos de una operación aritmético lógica puede estar en memoria (pero no los dos)

5. El tamaño de las instrucciones es variable (al contrario de MIPS, que siempre usa 32 bits para codificarlas).

Las arquitecturas que, como MIPS, requieren que los datos estén casi siempre en registros a la hora de operar con ellos se las conoce como de *registro-registro*, y es una característica típica de las máquinas modernas tipo RISC. Por el contrario, las arquitecturas que, como x86, permiten que alguno de los operandos de las instrucciones de procesamiento estén directamente en la memoria se las conoce como de *registro-memoria*, y es una característica típica de repertorios de instrucciones CISC.

La realidad es que, hoy en día, los repertorios de instrucciones CISC como x86 son traducidos a la hora de ejecutarlos por el propio hardware a secuencias de instrucciones internas más simples (similares a las RISC), en las cuales, entre otras cosas, se añaden instrucciones de carga y almacenamiento explícitas para realizar las operaciones aritméticas que incluyen operandos en memoria. La razón es que es mucho más fácil diseñar hardware eficiente para repertorios de instrucciones RISC que para repertorios CISC, dada la mayor simplicidad y uniformidad de los RISC.

## Apéndices

### A3.1. Manipulación de datos a nivel de bits

MIPS, al igual que la mayoría de las ISAs, permite direccionar los datos en memoria con una granularidad mínima de un byte. Es decir: es posible referirse de forma independiente a cualquier byte de memoria mediante las instrucciones `lb`, `lbu` y `sb`, las cuales nos permiten leer o escribir a la vez los 8 bits correspondientes al byte indicado por la dirección efectiva que se desee, pero no es posible acceder de forma directa e independiente a un subconjunto de los bits de un byte.

En ocasiones resulta útil utilizar menos de un byte para representar un dato. Por ejemplo, supongamos que queremos almacenar una lista de valores numéricos entre 1 y 3. Si utilizamos 2 bits para almacenar cada elemento, podremos almacenar en la misma cantidad de memoria una lista 4 veces más larga que si usáramos 1 byte. El caso más extremo (pero relativamente común) es el de un array de valores booleanos, en el que cada elemento se puede almacenar en un solo bit.

Los datos representados mediante un número de bits menor que un byte (o que una palabra) se suelen llamar «campos de bits» («bitfields»). El caso extremo de un dato que se almacena usando un solo bit (y por tanto sólo puede tomar dos valores) se le suele llamar «bandera» («flag»).

El acceso a campos de bits almacenados en memoria se realiza mediante combinaciones de las instrucciones de acceso a memoria habituales (`lw`, `sw`, `lhu`, `lh`, `sh`, `lbu`, `lb` y `sb`), operaciones de desplazamiento lógico (`sll`, `srl`, `sllv` y `srlv`) y operaciones lógicas (`and`, `andi`, `or` y `ori`).

#### A3.1.1. Lectura de campos de bits

Para leer el contenido de un campo de bits de la memoria se pueden realizar las siguientes acciones:

1. Leer el byte, media palabra o palabra que contiene los bits de interés (además de otros bits adicionales, por supuesto). Para esta tarea se utilizarán las instrucciones habituales de lectura de memoria.
2. Alinear los bits leídos en el registro destino, de forma que el bit de interés de menor peso esté colocado en el bit 0 del registro. Para esto se utilizará la instrucción `srl` o la `srlv`.
3. Descartar (poner a cero) los bits «sobrantes» del registro, utilizando las instrucciones `and` o `andi` con una máscara de bits adecuada.

Por ejemplo, si suponemos que un array de elementos enteros en el que cada elemento ocupa 3 bits comienza en la dirección denotada por la etiqueta `ll`, el acceso al elemento cuyo índice se encuentra en el registro `$t0` se podría realizar así:

```
# Cálculo de la dirección del byte que contiene al elemento:
#   - Cada byte contiene 8 / 3 elementos, por lo que el elemento
#     i-ésimo se encontrará en el byte cuya dirección es
#     "ll + (i * 3 / 8)".
#   - El resto de la división anterior nos dará el desplazamiento
#     en bits en el que comienza el campo de bits dentro del byte.
li    $t1, 3
mul   $t1, $t0, $t1
sra   $t2, $t1, 3    # división entre 8, cociente
andi  $t1, $t1, 0x7  # división entre 8, resto
lbu   $t2, ll($t2)   # carga el byte
```

```
srlv $t2, $t2, $t1 # alinea el campo de bits en el registro
andi $t2, $t2, 0x7 # pone a 0 todos los bits salvo los 3 primeros
```

Muy frecuentemente el tamaño del campo de bits será potencia de 2, por lo que se podrá utilizar una instrucción de desplazamiento aritmético en lugar de la instrucción de multiplicación utilizada en el fragmento de código anterior.

### A3.1.2. Escritura de campos de bits

La escritura en un campo de bits se realiza de forma similar a la lectura, pero teniendo en cuenta que antes de escribir en la posición de memoria destino es necesario leerla, ya que su contenido sólo se modificará parcialmente (los bits de esa posición de memoria que no pertenezcan al campo de bits actual deben quedar igual al final del proceso). Los pasos a dar para escribir un campo de bits son los siguientes:

1. Leer el byte, media palabra o palabra que contiene los bits de interés (además de otros bits adicionales que se deberán mantener intactos). Para esta tarea se utilizarán las instrucciones habituales de lectura de memoria.
2. Poner a cero los bits a modificar en el byte (o palabra) leído utilizando una instrucción `and` o `andi` con una máscara de bits adecuada.
3. Alinear los bits a escribir en el registro origen, de forma que los bits a escribir queden en la posición correspondiente del byte leído de memoria. Para esto se utilizará la instrucción `sll` o la `sllv`.
4. Combinar los bits a escribir con los bits del byte (o palabra) leído de memoria utilizando la instrucción `or`. También se puede utilizar la instrucción `ori` si los bits a escribir se dan como valor inmediato.
5. Escribir el resultado en la misma dirección de memoria de la que se leyó inicialmente.

Por ejemplo, si suponemos que un array de elementos enteros en el que cada elemento ocupa 3 bits comienza en la dirección denotada por la etiqueta `l1`, la escritura de los tres bits contenidos en el registro `$t1` en la posición cuyo índice se encuentra en el registro `$t0` se podría realizar así:

```
# Cálculo de la dirección del byte que contiene al elemento:
#   - Cada byte contiene 8 / 3 elementos, por lo que el elemento
#     i-ésimo se encontrará en el byte cuya dirección es
#     "l1 + (i * 3 / 8)".
#   - El resto de la división anterior nos dará el desplazamiento
#     en bits en el que comienza el campo de bits dentro del byte.
li    $t2, 3
mul   $t2, $t0, $t2
sra   $t3, $t2, 3 # división entre 8, cociente
andi  $t2, $t2, 0x7 # división entre 8, resto
lbu   $t4, l1($t3) # carga el byte
# Cálculo de la máscara para poner a cero los bits a sobrescribir
li    $t5, 0x7
sllv  $t5, $t5, $t2
not   $t5, $t5 # $t5 contendrá ceros solo en los 3 bits a sobrescribir
# Modificación del byte leído de memoria
and   $t4, $t4, $t5 # pone a cero los bits
sllv  $t1, $t1, $t2 # alinea los bits a escribir
or    $t4, $t4, $t1 # combina los bits con el byte leído anteriormente
sb    $t4, l1($t3) # escribe el byte
```

## A3.2. Uso de la memoria en MIPS

El objetivo de este apéndice es ofrecer información al alumno para que comprenda la organización de la memoria en MIPS, conozca el uso habitual que los lenguajes de programación hacen de cada zona, sepa cómo almacenar y acceder a datos almacenados en memoria utilizando las estructuras de datos más simples y comunes (arrays y registros) y entienda el concepto de puntero utilizado en muchos lenguajes de programación (como C).

### A3.2.1. Modos de direccionamiento a memoria del ensamblador de MIPS

MIPS es una arquitectura de carga/almacenamiento, lo que significa que sólo las instrucciones de carga y almacenamiento (ver sección 3.4.2) acceden a la memoria. El resto de instrucciones operan sólo con valores almacenados en los registros.

La máquina real proporciona un solo modo de direccionamiento para acceder a memoria llamado “*base más desplazamiento*”, que se ha explicado en la sección 3.4.2. Sin embargo, el ensamblador proporciona modos de direccionamiento adicionales para todas las instrucciones de carga y almacenamiento que implementa mediante pseudoinstrucciones. Las maneras de acceder a una posición desde una instrucción en nuestro código fuente se resumen en la tabla A3.1.

Tabla A3.1: Modos de direccionamiento a memoria del MIPS

Cálculo de dirección	Ejemplo
Contenido de registros	<code>lw \$t0, (\$a0)</code>
Inmediato	<code>lw \$t0, 0x1001001c</code>
Inmediato + contenido de registro	<code>lw \$t0, 0x1000(\$a0)</code>
Dirección de etiqueta	<code>lw \$t0, variable</code>
Dirección de la etiqueta + inmediato	<code>lw \$t0, variable+0x1000</code>
Dirección de la etiqueta + inmediato + contenido de registro	<code>lw \$t0, variable+0x1000(\$a0)</code>

El ensamblador traducirá las instrucciones que usen estos modos adicionales a las secuencias adecuadas de instrucciones reales. Por ejemplo, la instrucción `lw $t0, variable+0x1000($a0)`, que utiliza el último de los modos de direccionamiento adicionales que aparecen en la tabla A3.1, el ensamblador la traduce internamente por la siguiente secuencia de instrucciones reales si suponemos que `variable` está en la dirección `0x10012340` del segmento de datos:

```
lui $at, 0x1001 # Carga 0x1001 en la mitad superior de $at
addu $at, $at, $a0 # Le suma el registro $a0
lw $t0, 0x3340($1) # Añade la constante 0x1000
# y la parte baja del 0x10012340 correspondiente a
# la dirección de la variable (0x2340 + 0x1000).
```

### A3.2.2. Organización de la memoria en MIPS

Como sabemos, la memoria del computador es una tabla indexada por direcciones. Al conjunto de direcciones disponibles se le llama normalmente “*espacio de direcciones*”. Como se verá en el tema 6 de la asignatura, los programas de usuario no acceden directamente al espacio de direcciones físico, sino que cada programa de usuario tiene disponible un espacio de direcciones virtual privado. En el caso de MARS, dado que sólo se ejecuta una aplicación cada vez, se simula un espacio de direcciones virtual único compuesto de todas las direcciones representables con 32 bits.

El espacio virtual de direcciones se divide en varias zonas destinadas a distintos usos (almacenamiento del código del programa, almacenamiento de datos, pila...). Cada plataforma (combinación de procesador, sistema operativo y librerías del sistema) impone unas reglas sobre cómo se debe organizar el espacio de direcciones privado.

La organización básica de la memoria en MIPS (o, más correctamente, de la plataforma virtual ofrecida por MARS) es similar a la de casi todos los computadores. De hecho, podemos ver y cambiar dicha organización mediante la entrada de menú `Settings/Memory Configuration...` En la figura A3.1 podemos ver la configuración por defecto, que será la que usaremos.

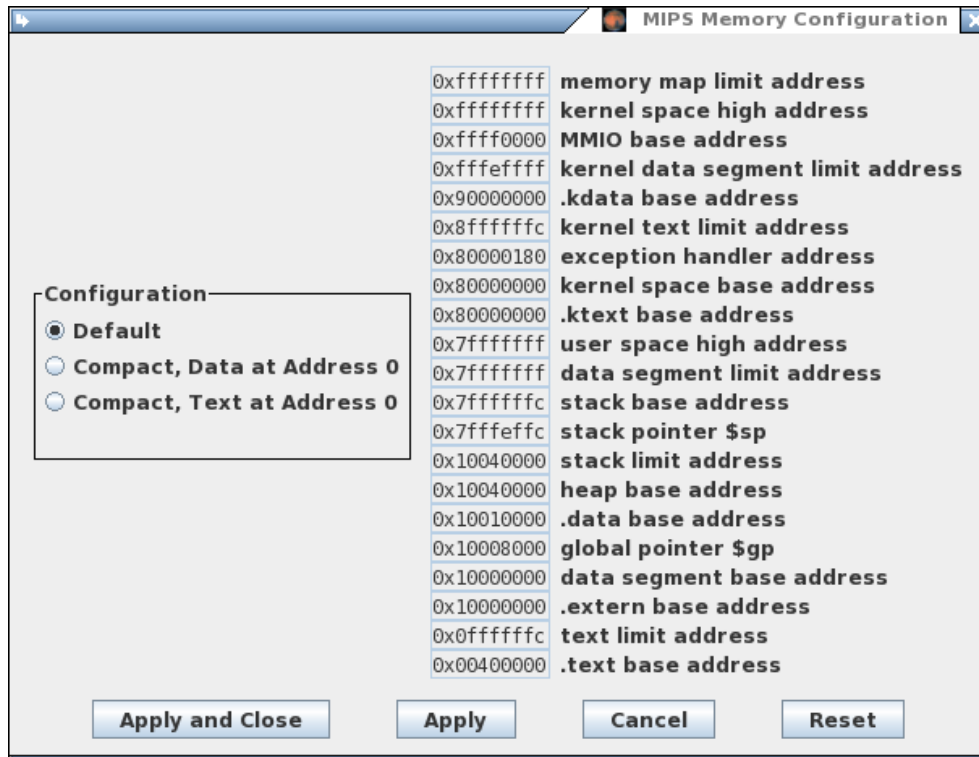


Figura A3.1: Configuración por defecto de la memoria en MARS

En primer lugar, el espacio de direcciones está dividido en dos mitades:

- Direcciones entre 0x00000000 a 0x7ffffff, reservadas para el programa de usuario.
- Direcciones entre 0x80000000 a 0xffffffff, reservadas para el sistema operativo (kernel).

En un sistema real, un programa de usuario no podrá acceder a las direcciones reservadas para el sistema operativo (saltaría una excepción si lo intentase, ver sección 3.5). Como se ve en la figura A3.1, el espacio reservado para el sistema operativo se divide a su vez en varias áreas más pequeñas. Veremos la función de algunas de ellas en temas posteriores de la asignatura.

Por su parte, la zona de memoria reservada para el usuario se divide también en varias zonas. De ellas, destacamos las siguientes:

- Las direcciones entre 0x00400000 y 0x0ffffff son el segmento de texto, que contiene las instrucciones del programa.
- El segundo bloque, comprendido entre la dirección 0x10000000 y la 0x7ffffff es el segmento de datos. Este bloque se divide a su vez en:

- El área de datos estáticos, que incluye las direcciones entre 0x10010000 y 0x1003ffff. En este área se colocan los datos declarados con la directiva `.data`. El tamaño de estos datos se conoce en el momento de realizar el ensamblado y su tiempo de vida es toda la ejecución del programa.
- El área de datos dinámicos, que comienza en la dirección 0x10040000 y crece hacia el límite superior del segmento de datos. En este área se colocan los datos cuyo tamaño no se conoce hasta el momento de la ejecución del programa o cuyo tiempo de vida no es toda la ejecución del programa. Para colocar datos en esta zona, se utiliza la llamada al sistema `sbrk` (ver sección A3.2.2). A esta zona también se la conoce como montículo, montón o `heap`.
- El tercer bloque es el espacio de memoria reservado para la pila (ver sección 3.4.4). Comienza en la dirección 0x7fffffff y crece hacia direcciones inferiores. MARS inicializa el puntero de pila a 0x7ffefffc, por lo que los 4 KB superiores de este área no se usan.  
Obsérvese que la pila crece hacia el área de datos dinámicos, y el área de datos dinámicos crece hacia la pila. Por tanto, el tamaño máximo de cada área depende del espacio ocupada en la otra.

En la mayoría de los casos (cuando programemos directamente en ensamblador), nuestros datos se colocarán en la zona de datos estáticos usando la directiva `.data` (como se ha hecho hasta ahora en todos los ejemplos). Es donde se suelen almacenar las variables **globales** de los lenguajes de alto nivel (en C: aquellas variables que se declaran fuera del cuerpo de cualquier función).

Sin embargo, los datos también se suelen almacenar en la pila y en el área de datos dinámicos. En la pila se almacenan las variables **locales** de los lenguajes de alto nivel (en C: aquellas que se declaran dentro del cuerpo de alguna función). Las variables locales también se pueden almacenar exclusivamente en registros cuando haya registros libres suficientes<sup>1</sup>, como hemos hecho hasta ahora en todos los ejemplos.

Para almacenar una variable en la pila usando ensamblador, basta con decrementar el puntero de pila tantos bytes como sea el tamaño del dato a almacenar. Es importante recordar que cuando el procedimiento termina, tiene que dejar el puntero de pila en la misma posición que lo encontró. Por tanto, una variable almacenada en la pila será local y *solo estará disponible hasta que el procedimiento acabe*<sup>2</sup>. Hasta ahora hemos usado la pila principalmente para almacenar el valor de los registros preservados entre llamadas. Obsérvese que la forma en la que la hemos usado es equivalente a crear variables locales para almacenar dichos valores.

Por último, en el área de datos dinámicos se almacenan (sorprendentemente) las variables **dinámicas**, las cuales se denominan así porque son creadas y destruidas de forma dinámica durante la ejecución del programa (en C: son los datos cuya memoria se reserva con `malloc` y se libera con `free`).

### La llamada al sistema `sbrk`

Para poder almacenar datos en el área de memoria dinámica, es necesario primero reservar el espacio que ocuparán. Para ello, se utiliza la llamada al sistema `sbrk`. Dicha llamada, recibe el número de bytes que queremos reservar en el registro `$a0` y nos devuelve en `$v0` la dirección de memoria donde comienza un bloque del tamaño deseado y disponible para su uso.

Esta llamada al sistema se puede utilizar para implementar las funciones `malloc` y `free` de la librería de C<sup>3</sup>. En este caso, `sbrk` se utilizaría para pedirle memoria al sistema operativo, y la librería que implementara `malloc` y `free` debería mantener información sobre el uso de esa memoria, para saber qué partes están

<sup>1</sup>Hay más condiciones (que dependen del lenguaje de alto nivel) para poder evitar reservar espacio en la pila para una variable local. Por ejemplo, en C será necesario almacenar una variable local en la pila si se le aplica en algún momento el operador “&” para obtener su dirección de memoria, porque sería imposible obtener la dirección de memoria de una variable que se almacena exclusivamente en un registro.

<sup>2</sup>Un error común de programación consiste en olvidar esto y almacenar la dirección de una variable local en una variable global (de tipo puntero) y utilizar dicha variable posteriormente a que el procedimiento que reservó espacio para la variable haya acabado. El resultado de esto es que cuando se accede a dicha variable, la memoria que usaba puede haber sido sobrescrita por cualquier otro procedimiento que haya usado la pila.

<sup>3</sup>Una implementación moderna de la librería de C podría utilizar otras técnicas además de, o en lugar de, `sbrk`.

reservadas y cuales han sido ya liberadas con `free` (y por tanto, pueden volver a reservarse a consecuencia de otra llamada a `malloc`).

### A3.2.3. Datos y punteros a datos

La mayoría de los datos usados por un programa se almacenan en la memoria, excepto en el momento justo en el que necesitan ser manipulados directamente (momento en el que se mueven a los registros del procesador, como hemos visto).

Cuando se declara una variable en un lenguaje de alto nivel, el compilador le asigna una porción de la memoria adecuada para el tamaño del dato a almacenar<sup>4</sup>. Dependiendo de varios factores (algunos de los cuales ya hemos mencionado), la variable se almacenará en la pila, el área de datos estáticos o el área de datos dinámicos.

La posición de la variable en memoria puede estar sujeta a restricciones según el tipo del dato. Por ejemplo: en MIPS es conveniente almacenar los enteros en posiciones de memoria cuya dirección sea múltiplo de 4 para poder acceder a ellos de forma rápida.

En un lenguaje de alto nivel, el compilador se encargará de acceder a las variables de la forma adecuada cuando se necesite, moviendo los datos entre la memoria y los registros de forma transparente al programador. Sin embargo, en ensamblador es el programador quien debe realizar este trabajo. La forma de referirse en ensamblador a una variable almacenada en memoria es mediante su **dirección**, la cual normalmente podremos nombrar mediante una etiqueta. Esta dirección se utilizará para cargar el dato mediante instrucciones de carga y almacenamiento (sección 3.4.2).

Algunos lenguajes de programación ofrecen la posibilidad de trabajar directamente con las direcciones de memoria de las variables. Por ejemplo, si en C tenemos las siguientes variables:

```
char c = 'x';
int i = 5;
double d = 7.0;
```

podremos referirnos a sus direcciones mediante las expresiones `&c`, `&i` y `&d`.

A su vez, la dirección de una variable se puede almacenar en otra variable. Las variables que almacenan direcciones se llaman *punteros*. En C, el tipo de un puntero a una variable de tipo “X” se denota con “X\*”<sup>5</sup>. Por ejemplo, a las declaraciones anteriores podemos añadir las siguientes:

```
char *pc = &c;
int *pi = &i;
double *pd = &d;
```

En C, el valor (o la memoria) al que apunta un puntero “x” en cada momento se denota por “\*x”. Así, después de las declaraciones anteriores, las siguientes expresiones son ciertas:

- `i == *pi`
- `&i == pi`

Suponiendo que las variables anteriores sean globales, un compilador de C podría emitir el siguiente código MIPS para traducir las declaraciones anteriores:

<sup>4</sup>En ocasiones, no es necesario asignarle memoria a algunas variables locales si su tiempo de vida es muy corto, ya que es suficiente con asignarles uno o varios registros. Esto se puede considerar una optimización, aunque ha sido el caso común hasta ahora en los ejemplos que hemos visto.

<sup>5</sup>Dado que los punteros son variables, también se almacenan en memoria y es posible tener punteros a punteros, que se denotarían con un asterisco adicional.

```
.data

c: .byte 'c'
i: .word 5
d: .double 7.0
pc: .word c
pi: .word i
pd: .word d
```

Obsérvese que todos los punteros ocupan el mismo espacio en memoria (una palabra de 4 bytes) independientemente del espacio ocupado por el dato al que apuntan.

En un lenguaje de alto nivel, el uso de punteros es útil sólo en contadas ocasiones. Sin embargo, en algunos lenguajes de no tan alto nivel, usar punteros es la única manera de realizar algunas tareas importantes. Por ejemplo, en C (y en ensamblador) es necesario usar punteros para:

- Pasar variables entre funciones *por referencia* en lugar de *por valor*. Esto permite que una función modifique el valor almacenado en una variable que se le pasa.
- Utilizar memoria reservada dinámicamente. En C podemos escribir:

```
int *e = malloc(sizeof(int));
```

para reservar 4 bytes (el tamaño de un entero en MIPS) en el área de memoria dinámica y guardar la dirección de comienzo de esos 4 bytes en la variable `e`. Esto sería aproximadamente equivalente a usar la llamada al sistema `sbrk` en ensamblador.

- Representar eficientemente algunas estructuras de datos, tales como listas enlazadas o árboles.

## Arrays

En la sección 3.4.2 vimos cómo traducir el acceso a elementos de un array. Si utilizamos el esquema descrito en dicha sección para traducir el siguiente código en C:

```
int v[20];

...

for (int i = 0; i < 20; ++i) {
    v[i] = v[i] + 7;
}
```

obtenemos el siguiente código en ensamblador:

```
.data
v: .space 80 # 20 elementos de 4 bytes

...

.text

li      $t0, 0      # Inicialización de i
ll:    bge     $t0, 20, 12 # Comprobación de la condición de salida
```

```

# Cálculo de la dirección de v[i]
la      $t1, v          # Carga la dirección de comienzo de v
sll     $t2, $t0, 2     # carga el desplazamiento del elemento i-ésimo
add     $t3, $t1, $t2   # Calcula &v[i] (dirección de v[i])
lw      $t4, 0($t3)     # Carga v[i]
addi    $t4, $t4, 7     # Calcula v[i] + 7
sw      $t4, 0($t3)     # Almacena v[i]
addi    $t0, $t0, 1     # ++i
j       l1

```

l2:

Podríamos haber utilizado alguno de los modos de direccionamiento que acabamos de ver en la sección [A3.2.1](#) para hacer que el cuerpo del bucle fuera más corto, pero el resultado a la hora de la ejecución en la máquina hubiera sido el mismo debido a la traducción que realizaría el ensamblador.

Si nos fijamos en lo que hace el código que acabamos de mostrar, nos damos cuenta de que en cada iteración se calcula la dirección de  $v[i]$ . Para ello, se carga la dirección de inicio de  $v$ , se multiplica el índice al que queremos acceder por el tamaño del elemento (4 en este caso, por lo que podemos utilizar la instrucción `sll`) y se suman ambos valores. En total, tres instrucciones por iteración para calcular la dirección de memoria a la que tenemos que acceder.

Sin embargo, si analizamos a qué direcciones se accede en cada iteración nos daremos cuenta de que se accede a posiciones consecutivas del array, cada una en una dirección que es 4 bytes mayor que la anterior. Una forma más inteligente de proceder sería: calcular la dirección del primer elemento al principio y, en cada iteración, sumarle 4 a la dirección accedida para obtener la dirección a la que se accederá en la siguiente iteración, hasta que lleguemos a la dirección final del array.

Para expresar en C la forma de recorrer el array descrita en el párrafo anterior, podríamos usar punteros de la siguiente forma<sup>6</sup>:

```

int v[20];

...

for (int *p = &v[0]; p < &v[20]; ++p) {
    *p = *p + 7;
}

```

que se podría traducir con el siguiente código ensamblador:

```

.data
v:    .space 80 # 20 elementos de 4 bytes

...

.text

# Cálculo de la dirección de v[0]
la    $t0, v          # Carga la dirección de comienzo de v == &v[0]
# Cálculo de la dirección de v[20]

```

<sup>6</sup>En C, cuando se incrementa un puntero en  $n$  unidades el valor almacenado en la variable se incrementa en  $n \times s$ , donde  $s$  es el tamaño del tipo al que apunta el puntero. Es decir, al sumarle 1 a un puntero a entero, el puntero pasa a apuntar al siguiente entero almacenado en memoria, no al siguiente byte.

```

# (v[20] no es realmente parte de v, cuyos elementos van de v[0] a v[19])
add    $t1, $t0, 80
l1:    bge    $t0, $t1, l2 # Comprobación de la condición de salida
        lw     $t2, 0($t0) # Carga *p (== v[i])
        addi   $t2, $t2, 7 # Calcula v[i] + 7
        sw     $t2, 0($t0) # Almacena *p (== v[i])
        addi   $t0, $t0, 4 # ++p
        j      l1
l2:

```

Como vemos, el cuerpo del bucle es ahora más corto, por lo que el número total de instrucciones ejecutadas es menor (y, por tanto, el programa tarda menos en ejecutarse). La transformación anterior la realizan automáticamente todos los compiladores modernos.

### Estructuras

Muchos lenguajes de programación permiten la declaración de estructuras (también llamadas “registros”). Cuando se declara una estructura en C, se crea un nuevo tipo que representa el producto cartesiano de sus componentes. La forma de almacenar una estructura en memoria es almacenando consecutivamente sus componentes, por lo que el tamaño de una variable de un tipo estructura será, al menos, la suma de los tamaños de sus componentes. Por ejemplo:

```

struct Punto {
    int x;
    int y;
};

```

```

struct Punto p;

```

declara una variable `p` que tiene dos campos enteros (`x` e `y`). El tamaño de `p` será de 8 bytes, de los que los primeros 4 bytes se utilizarán para almacenar `x` y los siguientes para `y`.

Para acceder en ensamblador a los campos de `p` necesitaremos conocer la dirección de `p` y tendremos que tener en cuenta el desplazamiento de cada campo. Por ejemplo:

```

        .data
p:      .space 8

...

        .text
la      $t0, p # Carga la dirección de p
li      $t1, 3
li      $t2, 4
sw      $t1, 0($t0) # p.x = 3
sw      $t2, 4($t0) # p.y = 4

```

El tamaño de una estructura puede ser mayor que el de sus componentes debido a restricciones de alineamiento. Por ejemplo, las variables del siguiente tipo:

```

struct A {
    char c;
    int i;
};

```

en MIPS ocuparían 8 bytes y no 5. Aunque el campo `c` sólo ocupa un 1 byte, se dejan 3 bytes de espacio entre `c` e `i`, porque los enteros deben almacenarse en direcciones múltiplos de 4 para facilitar el acceso.

#### A3.2.4. Ejercicios

1. Traduzca el código del programa en C de la figura [A3.2](#), tratando de ser lo más literal posible y teniendo cuidado de colocar cada variable en la zona de memoria adecuada según su declaración.
2. Modifique la traducción anterior para que los recorridos de los arrays se hagan mediante punteros (o mediante subíndices si el programa escrito ya usa punteros). Compare el número de instrucciones ejecutadas en cada caso (para ello, se puede utilizar la herramienta “*Instruction Counter*” de MARS).

La solución del ejercicio 1 se puede ver en las figuras [A3.3](#) y [A3.4](#). La solución del ejercicio 2 sería igual excepto para el procedimiento `main` que se puede ver en la figura [A3.5](#).

---

```
14 int i1;
15 char a1[200];
16
17 struct S {
18     int e1;
19     char e2;
20 };
21
22 struct S a2[94];
23
24 // Este procedimiento recibe un puntero a una variable de tipo
25 // "struct S" y modifica la variable apuntada
26 void proc2(struct S* p) {
27     p->e1 = 1 - p->e1;
28     print_int(p->e1);
29     print_char(p->e2);
30     print_char('\n');
31 }
32
33 void proc1(int *i, int j, int k) {
34     struct S s;
35     s.e1 = *i;
36     s.e2 = a1[j];
37     proc2(&s);
38     a2[k] = s;
39     *i = *i + j; // modifica la variable entera apuntada por i
40 }
41
42 int main(int argc, char *argv[]) {
43     i1 = 8;
44     for (int i = 0; i < 200; ++i) {
45         // almacena los 8 bits menos significativos de i en a1[i]
46         a1[i] = (char) i;
47     }
48     for (int j = 32; j < 126; ++j) {
49         proc1(&i1, j, j - 32);
50     }
51     for (int k = 0; k < 94; ++k) {
52         proc2(&a2[k]);
53     }
54     print_int(i1);
55     return 0;
56 }
```

---

Figura A3.2: ej1.c

---

```

1      .data
2
3  i1:   .space 4      # espacio para un entero
4  a1:   .space 200   # espacio para 200 chars
5
6  # struct S { int e1; char e2; };
7  # El tamaño de S será de 5 bytes, pero el tamaño en un array sería de
8  #   8 para preservar el alineamiento de e1
9  a2:   .space 752   # 94 * 8, espacio para 94 S
10
11     .text
12 # recibe int* i en $a0, int j en $a1 e int k en $a2
13 proc1:
14     addi    $sp, $sp, 24      # reserva espacio en la pila para $ra, $s0, $s1, $s2 y s (8 bytes)
15     sw     $ra, 20($sp)
16     sw     $s0, 16($sp)
17     sw     $s1, 12($sp)
18     sw     $s2, 8($sp)
19     # s.e1 está en 0($sp) y s.e2 en 4($sp)
20     move   $s0, $a0
21     move   $s1, $a1
22     move   $s2, $a2
23     lw     $t0, 0($s0)      # carga el valor apuntado por i
24     sw     $t0, 0($sp)      # s.e1 = *i;
25     lb     $t1, a1($s1)     # carga a1[j];
26     sb     $t1, 4($sp)      # s.e2 = a1[j];
27     la     $a0, 0($sp)      # pone la dirección de s en $a0
28     jal    proc2           # proc2(&s);
29     la     $t0, a2
30     sll   $t1, $s2, 3      # k * 8
31     add   $t0, $t0, $t1    # &a2[k]
32     lw   $t1, 0($sp)      # s.e1
33     sw   $t1, 0($t0)      # a2[k].e1 = s.e1;
34     lb   $t1, 4($sp)      # s.e2
35     sb   $t1, 4($t0)      # a2[k].e2 = s.e2;
36     lw   $t1, 0($s0)      # *i
37     add  $t1, $t1, $s1     # *i + j
38     sw   $t1, 0($s0)      # *i = *i + j;
39     lw   $s2, 8($sp)
40     lw   $s1, 12($sp)
41     lw   $s0, 16($sp)
42     lw   $ra, 20($sp)
43     addi $sp, $sp, -24
44     jr   $ra

```

---

Figura A3.3: Procedimiento proc1 y sección .data de ej1.s

```

46 # recibe p, un puntero a un S, en $a0
47 proc2:
48     move    $t0, $a0        # guarda el valor de p (para poder usar $a0 libremente)
49     lw     $t1, 0($t0)     # carga p->e1
50     li     $t2, 1
51     sub    $t2, $t2, $t1   # calcula 1 - p->e1
52     sw    $t2, 0($t0)     # almacena p->e1
53     move   $a0, $t2
54     li    $v0, 1
55     syscall                    # print_int(p->e1);
56     lb    $a0, 4($t0)     # carga p->e2
57     li    $v0, 11
58     syscall                    # print_char(p->e2);
59     li    $a0, '\n'
60     li    $v0, 11
61     syscall                    # print_char('\n');
62     jr    $ra
63
64     .globl main
65 main:
66     addi   $sp, $sp, 12    # reserva espacio en la pila para $ra, $s0, $s1
67     sw    $ra, 8($sp)
68     sw    $s0, 4($sp)
69     sw    $s1, 0($sp)
70     li    $t0, 8
71     sw    $t0, i1         # i1 = 8;
72     li    $t1, 0         # i = 0
73 11:     bge  $t1, 200, 12   # i < 200
74     la    $t2, a1
75     add   $t2, $t2, $t1
76     sb    $t1, 0($t2)    # a1[i] = (char) i
77     addi  $t1, $t1, 1    # ++i
78     j     11
79 12:     li    $s0, 32     # j = 32
80 13:     bge  $s0, 126, 14  # j < 126
81     la    $a0, i1       # &i1
82     move  $a1, $s0
83     subi  $a2, $s0, 32
84     jal   proc1         # proc1(&i1, j, j - 32);
85     addi  $s0, $s0, 1   # ++j
86     j     13
87 14:     li    $s1, 0     # k = 0
88 15:     bge  $s1, 94, 16  # k < 94
89     la    $t3, a2
90     sll  $a0, $s1, 3    # k * 8
91     add  $a0, $t3, $a0  # &a2[k]
92     jal   proc2         # proc2(&a2[k]);
93     addi  $s1, $s1, 1   # ++k
94     j     15
95 16:     lw    $a0, i1
96     li    $v0, 1
97     syscall
98     li    $v0, 10
99     syscall
100     lw    $s1, 0($sp)
101     lw    $s0, 4($sp)
102     lw    $ra, 8($sp)
103     addi  $sp, $sp, -12
104     jr    $ra

```

Figura A3.4: Procedimientos proc2 y main de ej1.s

---

```

64      .globl  main
65  main:
66      addi   $sp, $sp, 12      # reserva espacio en la pila para $ra, $s0, $s1
67      sw     $ra, 8($sp)
68      sw     $s0, 4($sp)
69      sw     $s1, 0($sp)
70      li     $t0, 8
71      sw     $t0, i1          # i1 = 8;
72      li     $t1, 0          # i = 0
73      la     $t2, a1         # $t2 apunta al primer elemento de a1
74  11:     bge  $t1, 200, 12   # i < 200
75         sb   $t1, 0($t2)    # a1[i] = (char) i
76         addi $t2, $t2, 1    # $t2 apunta al siguiente elemento
77         addi $t1, $t1, 1    # ++i
78         j    11
79  12:     li   $s0, 32        # j = 32
80  13:     bge  $s0, 126, 14   # j < 126
81         la   $a0, i1        # &i1
82         move $a1, $s0
83         subi $a2, $s0, 32
84         jal  proc1          # proc1(&i1, j, j - 32);
85         addi $s0, $s0, 1    # ++j
86         j    13
87  14:     la   $s1, a2        # $s1 = &a2[0]
88         add  $s2, $s1, 752  # $s2 = &a2[94] (94 * 8 = 752)
89  15:     bge  $s1, $s2, 16   # k < 94 ==> $s1 < $s2
90         move $a0, $s1      # &a2[k]
91         jal  proc2          # proc2(&a2[k]);
92         addi $s1, $s1, 8    # ++k ==> $s1 + 8
93         j    15
94  16:     lw   $a0, i1
95         li   $v0, 1
96         syscall
97         li   $v0, 10
98         syscall
99         lw   $s1, 0($sp)
100        lw   $s0, 4($sp)
101        lw   $ra, 8($sp)
102        addi $sp, $sp, -12
103        jr   $ra

```

---

Figura A3.5: Procedimiento main de ej2.s

## Boletines de prácticas

### Normas sobre la entrega de prácticas

Será necesario seguir las siguientes reglas para entregar las prácticas de todos los boletines, además de cualquier otra regla específica que se indique en un boletín particular:

- Las prácticas se entregarán únicamente mediante la opción de contenidos del alumno en SUMA.
- Se entregará un único archivo comprimido en formato *.tar.gz* o *.zip* que contendrá la memoria en formato PDF, los circuitos y programas que se hayan generado (código fuente) y cualquier otro fichero que se considere oportuno. El nombre del archivo será *prácticas-DNI-BOLETIN.FORMATO* (por ejemplo: *prácticas-12345678-B2.3.tar.gz*).
- La memoria incluirá, al menos, la siguiente información en un solo documento PDF:
  - Nombre y DNI del autor o autores de la práctica.
  - Descripción de los ficheros y directorios contenidos en el archivo entregado.
  - Contestación a las preguntas planteadas en los boletines. La respuesta a cada pregunta debe ser independiente, y debe estar claramente identificada.
  - Explicación de las pruebas realizadas para comprobar la corrección de la práctica entregada e instrucciones para su reproducción. Cuando sea posible, se incluirán los ficheros utilizados en dichas pruebas.
  - Explicación del trabajo realizado y cualquier aclaración que el alumno considere pertinente.
  - Lista de bibliografía y otras fuentes de información consultadas.

No se corregirá ninguna práctica que no se ciña estrictamente a los formatos especificados anteriormente.

### B3.1. Uso del simulador MARS

#### B3.1.1. Objetivos

Los objetivos de esta sesión son que el alumno se familiarice con el simulador MARS y que sea capaz de seguir la ejecución de programas sencillos en ensamblador de MIPS.

#### B3.1.2. Prerequisitos

- Lectura de los apuntes de teoría, en especial las secciones 3.1, 3.2, 3.3, 3.4 y 3.6.


#### B3.1.3. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura por parte del alumno de las secciones B3.1.4 y B3.1.5.
2. Realización, en grupos de dos personas, de los ejercicios propuestos en el boletín (con supervisión del profesor).

### B3.1.4. El simulador MARS

MARS es un programa para simular el ISA de MIPS diseñado para uso educacional. Puede ser descargado libremente de la dirección <http://cs.missouristate.edu/MARS/>, aunque para la realización de las prácticas de la asignatura se deberá utilizar siempre la versión disponible en la página web de la asignatura (<http://ditec.um.es/etc/>). Esta última versión incluye algunas modificaciones necesarias para los ejercicios y el proyecto que se propondrán a lo largo del curso.

Cuando se arranca el programa y se abre el fichero `ejemplo1.s` (cuyo contenido se muestra en la figura B3.3), la ventana de MARS tiene el aspecto que se puede ver en la figura B3.1. Podemos editar directamente el programa usando MARS, o usar cualquier otro editor de texto. También se puede ensamblar el programa usando la tecla `F3` o el botón  de la barra de herramientas, tras lo que el programa tendrá el aspecto de la figura B3.2.

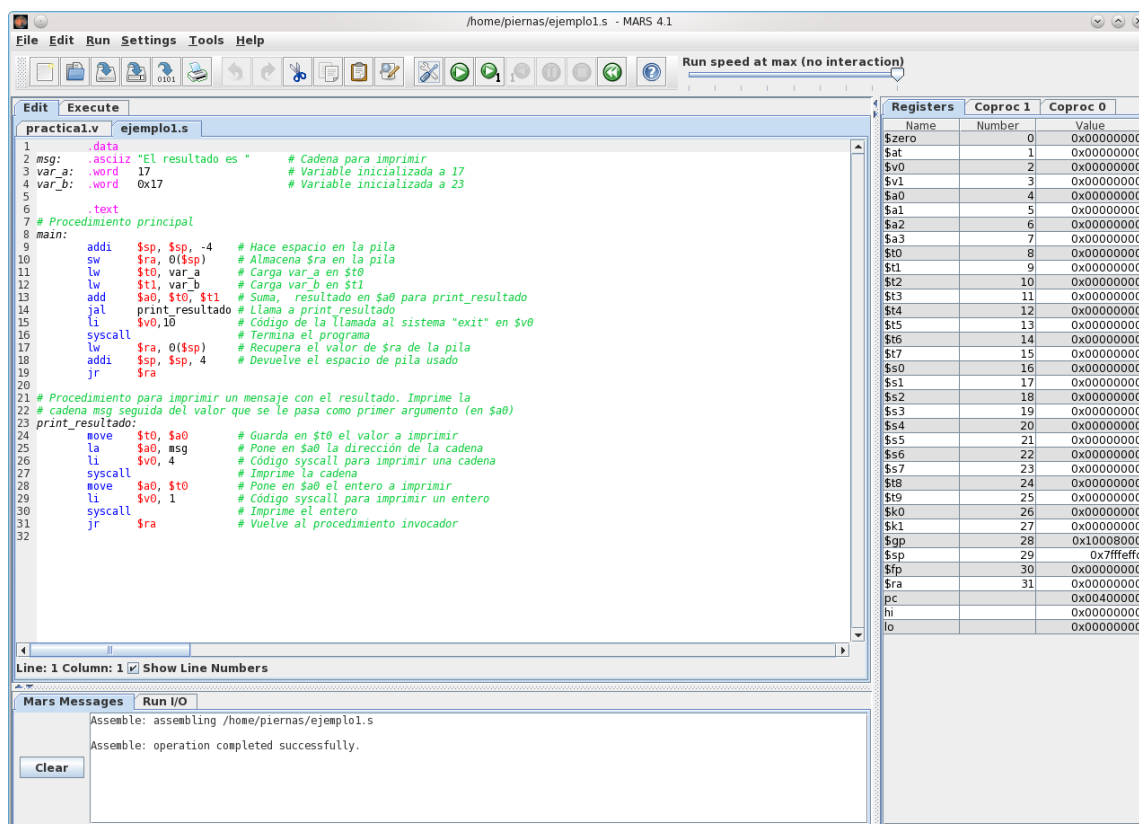


Figura B3.1: Aspecto de MARS después de cargar un programa

MARS incluye en su distribución una documentación escueta pero útil. Esta documentación está accesible desde el menú “Help” o pulsando la tecla `F1`. Se recomienda al alumno que se familiarice con la documentación disponible. Ésta incluye un listado de las instrucciones, pseudoinstrucciones, directivas y llamadas al sistema disponibles en MARS.

En la figura B3.2 podemos ver que MARS nos muestra varias ventanas y paneles de información:

- La ventana titulada “Text Segment” nos permite ver el contenido del *segmento de texto*, es decir, de la zona de memoria que contiene el programa recién ensamblado. Cada línea de la tabla se corresponde con una palabra de 4 bytes, es decir: una instrucción de código máquina. De cada palabra, podemos ver su dirección, su contenido en hexadecimal, la instrucción correspondiente tal y como la vería el procesador, y la línea de ensamblador que generó la instrucción. En el caso de pseudoinstrucciones que generan varias instrucciones en código máquina, la línea de ensamblador aparece asociada solo

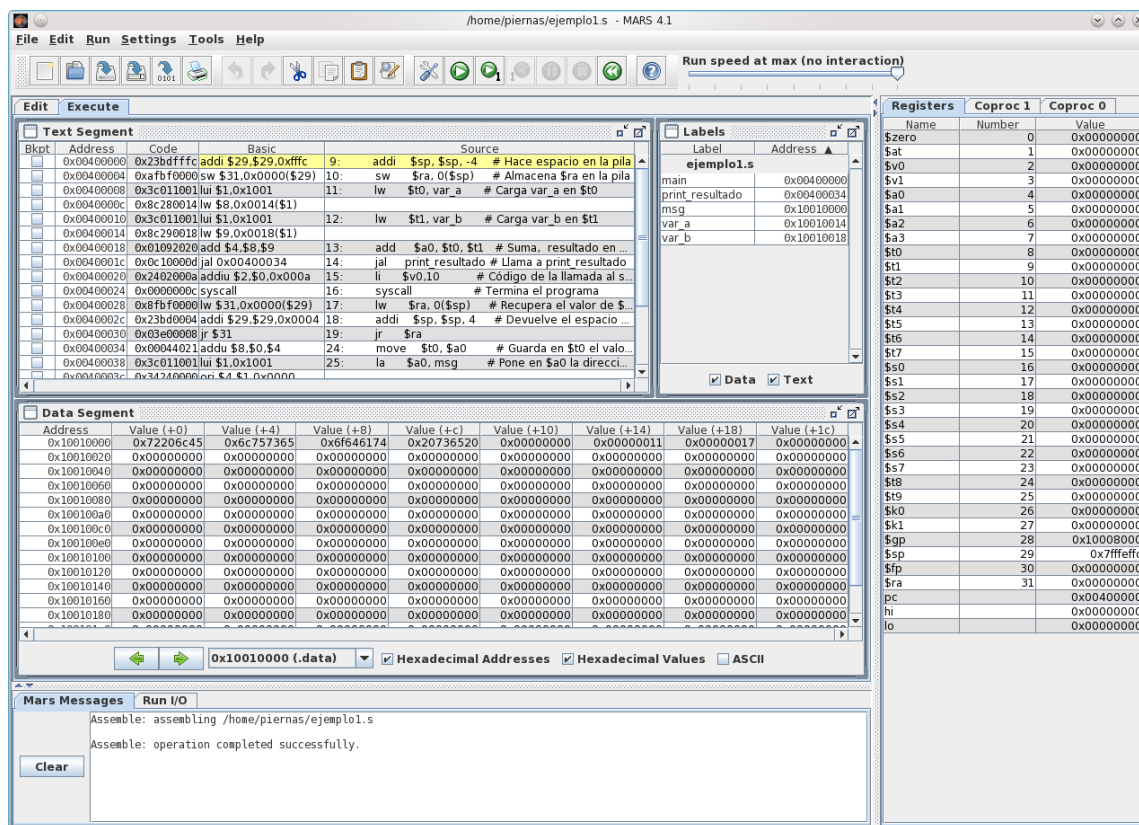


Figura B3.2: Aspecto de MARS después de ensamblar un programa

a la primera instrucción real (como es el caso de las instrucciones almacenadas en las direcciones 0x00400008 y 0x0040000c, que se generan a partir de la línea 11 de ejemplo1.s).

- En la ventana “Data Segment” podemos ver el contenido de la zona de memoria que almacena los datos. De nuevo, la información aparece separada en palabras de 4 bytes. Esta ventana también nos permite modificar el contenido de la memoria escribiendo directamente en la celda correspondiente.
- La ventana “Labels” nos muestra la dirección de memoria que el ensamblador ha asociado con cada una de las etiquetas declaradas en el programa.
- A la derecha de la ventana principal disponemos de un panel que muestra el contenido de los registros del procesador y permite modificarlo. La información está dividida en 3 pestañas:

**Registers:** muestra el contenido de los 32 registros de propósito general y de tres registros especiales: el contador de programa (pc) y los registros hi y lo usados por las instrucciones de multiplicación y división.




**Coproc0:** muestra algunos de los registros del coprocesador 0 relacionados con el manejo de excepciones (véase la sección 3.5 para más información).


**Coproc1:** muestra el banco de registros en coma flotante (sección 3.4.5). Los registros aparecen tanto individualmente (para los valores de simple precisión) como en parejas (para los valores de doble precisión).

- En la parte inferior de la ventana disponemos de un panel con 2 pestañas. La primera de ellas muestra los mensajes de MARS, incluyendo los errores encontrados al ensamblar un programa. La segunda muestra la salida del programa simulado.

Si observamos la información mostrada por MARS justo después de ensamblar el programa, podemos ver, por ejemplo, que la etiqueta `var_b` está asociada a la dirección `0x1001018`. Si miramos en la ventana “*Data Segment*” el valor almacenado en dicha dirección, vemos que es `0x17`, tal y como se especifica en la línea 4 de `ejemplo1.s`. Análogamente, la etiqueta `print_resultado` está asociada con la dirección `0x00400034`, la cual podemos comprobar en la ventana “*Text Segment*” que contiene la primera instrucción del procedimiento, definida en la línea 24 de `ejemplo1.s`.

También podemos ver que el registro `pc` contiene el valor `0x0040000`, que se corresponde con el comienzo del segmento de texto. Cuando se ensambla un programa, MARS asigna a dicho registro la dirección del procedimiento principal<sup>1</sup> o la dirección de inicio del segmento de texto si no se encuentra un procedimiento principal. Por su parte, el registro `sp` se inicializa al valor `0x7fffffc` donde se encuentra la cima de la pila vacía.

Podemos iniciar la ejecución del programa actual con la tecla `F5` o el botón . MARS permite ralentizar la ejecución del programa para facilitar su seguimiento paso a paso usando el control etiquetado “*Run speed*” situado a la derecha de la barra de herramientas. La ejecución también se puede realizar instrucción a instrucción usando la tecla `F7` o el botón , e incluso hacia atrás con la tecla `F8` o el botón .

Si ejecutamos el programa después de ensamblarlo, aparecerá en la pestaña “*Run I/O*” el mensaje “El resultado es 40” seguido de un mensaje indicando que el programa ha acabado correctamente. Para volver a ejecutar el programa, podemos volver a ensamblar el programa o devolver el simulador a su estado original con la tecla `F12` o el botón .

También podemos usar la casilla “*Bkpt*” que se encuentra a la izquierda de cada instrucción en la ventana “*Text segment*” para conseguir que la ejecución se pare cada vez que se llegue a esa instrucción.

Además de simular el ISA de MIPS de forma suficientemente completa para ejecutar programas simples, MARS también pone a disposición del programador un conjunto de llamadas al sistema. En una máquina real, estas llamadas al sistema serían provistas por el sistema operativo.

Las llamadas al sistema disponibles incluyen las necesarias para realizar la entrada y salida. El listado completo de llamadas disponibles puede consultarse en la documentación de MARS (tecla `F1`). En la sección [B3.1.5](#) se comentará el uso de las llamadas al sistema usadas por `ejemplo1.s` (por ejemplo, para mostrar el mensaje con el resultado).

### B3.1.5. Anatomía de un programa en ensamblador

En la figura [B3.3](#) se muestra el programa `ejemplo1.s`. Este programa es muy simple: realiza la suma de dos valores almacenados en sendas variables e imprime un mensaje con el resultado.

El programa está dividido en dos secciones: una de datos y otra de código. Las directivas `.data` y `.text` marcan el inicio de cada una de ellas, respectivamente.

En la sección de datos se definen tres etiquetas. La primera de ellas (`msg`) apuntará a una cadena que se usará posteriormente para mostrar un mensaje. El mensaje se codifica con la directiva `.asciiz`, que define una cadena codificada en ASCII y terminada en 0. Las dos restantes (`var_a` y `var_b`) apuntan al espacio reservado para las dos variables. Este espacio se reserva y se inicializa mediante las directivas `.word`.

En el segmento de código se definen dos procedimientos: el procedimiento principal `main` y el procedimiento `print_resultado`, que es llamado desde `main`.

El procedimiento `main` es el que se empieza a ejecutar al principio del programa. Este procedimiento, al igual que cualquier otro, podemos dividirlo en tres partes:

**Prólogo:** de la línea 9 a la 10. En él, se guarda en la pila el valor de los registros de tipo *preservados entre llamadas* (ver sección [3.4.4](#)) que se van a modificar en el cuerpo del procedimiento.

<sup>1</sup>Para que el procedimiento principal sea reconocido como tal debe tener una etiqueta llamada “`main`” que haya sido declarada como global (con la directiva “`.global`”). En particular, el procedimiento “`main`” de `ejemplo1.s` no sería reconocido como procedimiento principal porque no se ha incluido la directiva “`.global`” correspondiente.

```

1      .data
2  msg:  .asciiz "El resultado es "      # Cadena para imprimir
3  var_a: .word 17                      # Variable inicializada a 17
4  var_b: .word 0x17                   # Variable inicializada a 23
5
6      .text
7  # Procedimiento principal
8  main:
9      addi    $sp, $sp, -4             # Hace espacio en la pila
10     sw      $ra, 0($sp)              # Almacena $ra en la pila
11     lw      $t0, var_a               # Carga var_a en $t0
12     lw      $t1, var_b               # Carga var_b en $t1
13     add     $a0, $t0, $t1            # Suma, resultado en $a0 para print_resultado
14     jal    print_resultado           # Llama a print_resultado
15     li     $v0, 10                   # Código de la llamada al sistema "exit" en $v0
16     syscall                               # Termina el programa
17     lw      $ra, 0($sp)              # Recupera el valor de $ra de la pila
18     addi    $sp, $sp, 4              # Devuelve el espacio de pila usado
19     jr      $ra
20
21 # Procedimiento para imprimir un mensaje con el resultado. Imprime la
22 # cadena msg seguida del valor que se le pasa como primer argumento (en $a0)
23 print_resultado:
24     move    $t0, $a0                 # Guarda en $t0 el valor a imprimir
25     la     $a0, msg                  # Pone en $a0 la dirección de la cadena
26     li     $v0, 4                    # Código syscall para imprimir una cadena
27     syscall                               # Imprime la cadena
28     move    $a0, $t0                 # Pone en $a0 el entero a imprimir
29     li     $v0, 1                    # Código syscall para imprimir un entero
30     syscall                               # Imprime el entero
31     jr      $ra                      # Vuelve al procedimiento invocador

```

Figura B3.3: ejemplo1.s

**Cuerpo:** de la línea 11 a la 16. Se realiza el trabajo del procedimiento.

**Epílogo:** de la línea 17 a la 19. Se recuperan los valores guardados anteriormente por el prólogo y se retorna el control al procedimiento invocador.

En el prólogo de este procedimiento se guarda en la pila<sup>2</sup> sólo el registro `$ra`, ya que es el único del conjunto de registros de tipo *preservados entre llamadas* que se modifica en el cuerpo del procedimiento (en concreto, se modifica en la línea 14).

El cuerpo del procedimiento carga el valor almacenado en las variables `var_a` y `var_b`, lo suma y llama al procedimiento `print_resultado`. Obsérvese que el resultado de la suma (línea 13) se almacena en el registro `$a0` para que sea el primer argumento de `print_resultado`. Por último, el cuerpo del procedimiento invoca la llamada al sistema número 10, la cual finaliza la ejecución del programa.

Por otro lado, el procedimiento `print_resultado` se limita a recibir un argumento, imprimir un mensaje e imprimir el valor del argumento recibido. Obsérvese que el prólogo de este procedimiento está vacío dado que no modifica el valor de ningún registro de tipo *preservado entre llamadas* y el epílogo se limita a devolver el control al procedimiento invocador mediante la instrucción `jr $ra`.

<sup>2</sup>En el caso concreto de este procedimiento, no sería estrictamente necesario guardar nada, ya que el programa acaba siempre en la línea 16, antes de que se llegue a ejecutar el epílogo.

### B3.1.6. Ejercicios

Para cada ejercicio, el portafolios deberá incluir una explicación de cómo se ha resuelto y, en caso de que se realice o modifique algún programa, el código realizado y al menos un ejemplo de su ejecución.

1. Cargue, ensamble y ejecute el programa `ejemplo1.s`. Compruebe que el mensaje mostrado en la pestaña “Run I/O” es “El resultado es 40”. Utilice MARS para, sin modificar el código en ensamblador, conseguir que la salida del programa en la siguiente ejecución sea “El resultado es 5”.
2. Modifique el programa `ejemplo1.s` para que, en vez de sumar el contenido de las variables `var_a` y `var_b`, le pregunte cada vez al usuario qué números sumar.
3. Escriba un programa para sumar fracciones. El programa le debe preguntar al usuario el valor de cuatro variables ( $a$ ,  $b$ ,  $c$  y  $d$ ) y debe calcular y mostrar el resultado de  $\frac{a}{b} + \frac{c}{d}$ . No es necesario que la fracción resultante aparezca simplificada y se puede asumir que todos los resultados y valores intermedios caben en 32 bits.

## B3.2. Convenciones de programación en MIPS

### B3.2.1. Objetivos

El objetivo de la sesión es que el alumno sea capaz de escribir procedimientos que realicen tareas sencillas utilizando correctamente las convenciones de usos de registros y de llamadas a procedimientos de MIPS.

### B3.2.2. Prerequisitos

- Lectura de los apuntes de teoría, en especial las secciones 3.3.1 y 3.4.4.

### B3.2.3. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura por parte del alumno de las secciones B3.2.4 y B3.2.5.
2. Realización, en grupos de dos personas, de los ejercicios propuestos en el boletín (con supervisión del profesor).

### B3.2.4. Ejemplos de paso de parámetros

Vamos a comenzar analizando un pequeño programa de ejemplo, cuyo código fuente se encuentra disponible en el fichero `procedimientos.s`. En este programa se incluyen, entre otras cosas, los siguientes procedimientos:

**main:** el procedimiento principal que, en este caso, se limita a llamar a `proc1` y a finalizar el programa después.

**proc1:** es un procedimiento que no recibe ningún argumento ni devuelve ningún resultado. Sin embargo, realiza llamadas a otros procedimientos (figura B3.4).

**proc2:** un procedimiento que recibe tres parámetros y devuelve dos. Siguiendo el convenio de paso de parámetros del ABI de MIPS, los parámetros se comunicarán a `proc2` usando los registros `$a0`, `$a1` y `$a2`. Por su parte, los resultados se recibirán en los registros `$v0` y `$v1` (figura B3.5).

**proc3:** un procedimiento que recibe seis parámetros y devuelve uno. Siguiendo los mismos convenios, los cuatro primeros parámetros se comunicarán usando los registros \$a0, \$a1, \$a2 y \$a3, mientras que el quinto y el sexto parámetro se pasarán usando la pila (como se describe más adelante). Además, se utilizará el registro \$v0 para recibir el resultado del procedimiento (figura B3.6).

**printlnInt:** procedimiento que imprime un entero (que recibe en \$a0) y salta a la línea siguiente de la consola (figura B3.7).

---

```

22 # proc1: no recibe ni devuelve nada
23 proc1:
24     addi    $sp, $sp, -12
25     sw     $s1, 8($sp)
26     sw     $s0, 4($sp)
27     sw     $ra, 0($sp)
28
29     # llama a proc2(3, 4, 5)
30     li     $a0, 3
31     li     $a1, 4
32     li     $a2, 5
33     jal    proc2
34     move   $s0, $v0    # Primer valor devuelto por proc2
35     move   $s1, $v1    # Segundo valor devuelto por proc2
36
37     # imprime los resultados en orden inverso
38     move   $a0, $s1
39     jal    printlnInt
40     move   $a0, $s0
41     jal    printlnInt
42
43     # llama a proc3(8, 7, 6, 5, 4, 3)
44     li     $a0, 8
45     li     $a1, 7
46     li     $a2, 6
47     li     $a3, 5
48     # Los dos argumentos restantes se pasan usando la pila
49     addi   $sp, $sp, -8 # Hace sitio en la pila
50     li     $t0, 4
51     sw     $t0, 0($sp) # Quinto argumento: 4
52     li     $t0, 3
53     sw     $t0, 4($sp) # Sexto argumento: 3
54     jal    proc3
55     addi   $sp, $sp, 8  # Devuelve el espacio a la pila
56
57     # imprime el resultado
58     move   $a0, $v0
59     jal    printlnInt
60
61     lw     $ra, 0($sp)
62     lw     $s0, 4($sp)
63     lw     $s1, 8($sp)
64     addi   $sp, $sp, 12
65     jr     $ra

```

---

Figura B3.4: procedimientos.s – proc1

El procedimiento `proc1` comienza guardando en la pila los registros *preservados entre llamadas* que modifica (\$ra, \$s0 y \$s1), los cuales se desapilan al final del procedimiento. A continuación, este procedi-

```

68 # proc2: recibe tres enteros (x, y, z) y devuelve dos (x + y + z, x * y * z)
69 proc2:
70     add    $v0, $a0, $a1
71     add    $v0, $v0, $a2
72     mul    $v1, $a0, $a1
73     mul    $v1, $v1, $a2
74     jr     $ra

```

Figura B3.5: procedimientos.s – proc2

```

77 # proc3: recibe seis enteros y devuelve uno (la suma de todos)
78 proc3:
79     add    $v0, $a0, $a1
80     add    $v0, $v0, $a2
81     add    $v0, $v0, $a3
82     lw     $t0, 0($sp) # El quinto parámetro está en la cima de la pila
83     add    $v0, $v0, $t0
84     lw     $t1, 4($sp) # Y el sexto parámetro en la siguiente posición
85     add    $v0, $v0, $t1
86     jr     $ra

```

Figura B3.6: procedimientos.s – proc3

```

89 # imprime un entero (recibido en $a0) y un retorno de carro
90 printlnInt:
91     li    $v0, 1 # el entero a imprimir ya se encuentra en $a0
92     syscall
93     la    $a0, msg_cr # dirección de la cadena "\n"
94     li    $v0, 4
95     syscall
96     jr    $ra

```

Figura B3.7: procedimientos.s – printlnInt

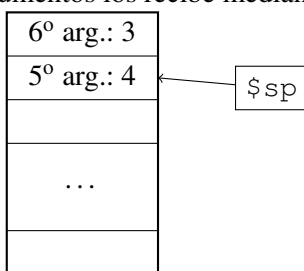
miento realiza varias llamadas consecutivas a procedimientos:

1. A `proc2` pasándole los parámetros 3, 4 y 5. Para ello, simplemente se colocan dichos valores en los registros `$a0`, `$a1` y `$a2`, y se utiliza la instrucción `jal` para guardar el contador de programa en `$ra` y saltar a la primera instrucción de `proc2`. Obsérvese que los valores devueltos por `proc2` mediante los registros `$v0` y `$v1` se copian a los registros `$$s0` y `$$s1`. Esto se hace así para evitar que las llamadas posteriores a otros procedimientos sobrescriban esos valores.
2. A `printlnInt`, pasándole en el registro `$a0` el valor almacenado en `$$s1`, es decir, el segundo valor que había sido devuelto anteriormente por `proc2`.
3. A `printlnInt` de nuevo, pasándole ahora el valor almacenado en `$$s0`, es decir, el primer valor que había sido devuelto anteriormente por `proc2`. Obsérvese que, si no se hubiera guardado anteriormente en `$$s0` el valor que se quiere imprimir ahora, éste hubiera sido sobrescrito por la ejecución anterior de `printlnInt`.
4. A `proc3` pasándole los parámetros 8, 7, 6, 5, 4 y 3. Debido a que el número de parámetros que recibe `proc3` es mayor que el número de registros dedicados para el paso de parámetros (4), es necesario

utilizar la pila para comunicar algunos de ellos. Los cuatro primeros parámetros se comunican usando los registros `$a0`, `$a1`, `$a2` y `$a3`, mientras que el quinto y el sexto parámetros se apilan. Para ello, se decrementa `$sp` en 8 unidades (porque cada parámetro va a ocupar 4 bytes de la pila) y se mueve el quinto parámetro a la posición de memoria 0 (`$sp`) y el sexto a 4 (`$sp`). Es decir, la pila quedará como si se hubiera apilado primero el sexto parámetro y después el quinto<sup>3</sup>. El procedimiento que realiza la llamada (`proc3` en este caso) es el responsable de devolver el espacio a la pila una vez que recupera el control (línea 55).

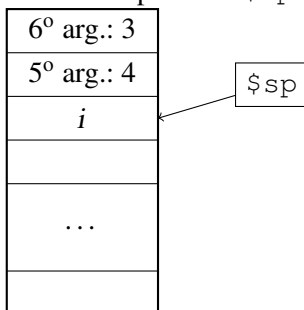
El procedimiento `proc2` se limita a realizar las operaciones aritméticas requeridas para calcular los resultados que devuelve en `$v0` y `$v1`. No necesita manipular la pila, ya que no escribe en ningún registro preservado entre llamadas. De hecho, sólo escribe en `$v0`, `$v1` y los registros `HI` y `LO`<sup>4</sup>.

El procedimiento `proc3` tiene la peculiaridad de que recibe más de 4 argumentos. El quinto y el sexto argumentos los recibe mediante la pila, cuyo estado al comienzo del procedimiento será el siguiente:



Es decir, `proc3` debe acceder a 0 (`$sp`) para leer su quinto argumento y a 4 (`$sp`) para el sexto. Los cuatro primeros argumentos estarán, como es habitual, en los registros `$a0`, `$a1`, `$a2` y `$a3`.

Sin embargo, en estos casos se ha de tener en cuenta que si `proc3` necesita apilar algo, la posición de los argumentos respecto de `$sp` cambiará. Por ejemplo, el estado de la pila si apilamos un entero `i` será:



y a partir de ese momento `proc3` deberá acceder a 4 (`$sp`) para leer su quinto argumento y a 8 (`$sp`) para el sexto.

Alternativamente, se puede utilizar el registro `$fp` para almacenar el valor de `$sp` a la entrada del procedimiento y usar dicho registro como base para acceder a los parámetros extra (o variables locales) con desplazamientos estables independientemente de lo que se apile en el cuerpo del procedimiento. En este último caso, habrá que guardar previamente el valor de `$fp` en la pila porque es un registro preservado entre llamadas.

En algunos casos, podríamos pensar que no es necesario seguir estrictamente los convenios de llamada a procedimiento y las reglas de uso de la pila. Por ejemplo, `proc1` guarda en la pila el valor de los registros `s0` y `s1` antes de modificarlos pero el procedimiento `main` que llama a `proc1` no usa dichos registros, por lo que no se vería afectado si `proc1` no los guardara. Sin embargo, si hiciéramos eso, `proc1` no podría ser llamado desde otro procedimiento que sí usara esos registros (reduciendo la reusabilidad) y, si en algún momento

<sup>3</sup>Aunque apilar los parámetros en orden inverso puede parecer extraño, es el habitual en la mayoría de ABIs porque facilita la implementación de funciones que reciben un número variable de argumentos (como por ejemplo, la función `printf` de C).

<sup>4</sup>La instrucción `mul` no es una pseudoinstrucción, pero usa el mismo hardware que la instrucción `mult` descrita en la sección 3.4.1.

modificáramos el procedimiento `main`, tendríamos que recordar no usar esos registros (arriesgándonos a introducir un error en caso contrario).

Es importante recordar que es necesario cumplir las convenciones de llamadas siempre, incluso aunque en algunos casos sepamos que el programa funcionaría correctamente sin ellas. Estas convenciones aseguran que procedimientos escritos por distintos programadores (o compilados por distintos compiladores) pueden llamarse unos a otros correctamente.

### B3.2.5. Ejemplo de procedimiento recursivo

Frecuentemente es útil que un procedimiento se llame a sí mismo. Por ejemplo, el elemento  $i$ -ésimo de la conocida sucesión de Fibonacci se define de la siguiente manera:

$$f_i = \begin{cases} 0 & \text{si } i = 0 \\ 1 & \text{si } i = 1 \\ f_{i-2} + f_{i-1} & \text{en otro caso} \end{cases}$$

Una traducción directa (aunque muy ineficiente) de la definición anterior sería un procedimiento que recibiera un argumento entero ( $i$ ) y siguiera los siguientes pasos:

1. Comprobar si el valor de  $i$  es 0, acabando y devolviendo 0 en ese caso.
2. Comprobar si el valor de  $i$  es 1, acabando y devolviendo 1 en ese caso.
3. Calcular  $f_{i-2}$ , llamándose a sí mismo.
4. Calcular  $f_{i-1}$ , llamándose a sí mismo.
5. Sumar los valores calculados en los 2 pasos anteriores, acabando y devolviendo el resultado de la suma.

En el ensamblador de MIPS, un procedimiento puede llamarse a sí mismo sin ningún problema, siempre y cuando se sigan correctamente las convenciones de programación vistas hasta ahora. Por tanto, una implementación del algoritmo anterior sería el procedimiento de la figura B3.8.

### B3.2.6. Ejercicios

Para cada ejercicio, el portafolios deberá incluir una explicación de cómo se ha resuelto y, en caso de que se realice o modifique algún programa, el código realizado y al menos un ejemplo de su ejecución.

1. El triángulo de Pascal (también conocido como triángulo de Tartaglia) es un conjunto de números dispuestos en forma de triángulo que expresan coeficientes binomiales. Dicho triángulo permite calcular de forma sencilla números combinatorios.

El triángulo se construye siguiendo las siguientes reglas:

- La primera fila tiene un elemento (el 1) y cada fila tiene un elemento más que la anterior.
- El primer y último elemento de todas las filas es el 1.
- El valor de todos los elementos (excepto el primero y último de cada fila) es la suma del elemento que se encuentra en la misma posición en la fila inmediatamente superior y del elemento que se encuentra en la posición previa de la fila inmediatamente superior.

---

```

25 fibo:  addi    $sp, $sp, -12
26         sw     $s1, 8($sp)
27         sw     $s0, 4($sp)
28         sw     $ra, 0($sp)
29
30         beq    $a0, 0, es0
31         beq    $a0, 1, es1
32
33         # guarda $a0 en $s0
34         move   $s0, $a0
35
36         # calcula fibo(i - 1)
37         subi   $a0, $s0, 1
38         jal    fibo
39         move   $s1, $v0          # guarda el resultado en $s1
40
41         # calcula fibo(i - 2)
42         subi   $a0, $s0, 2
43         jal    fibo
44
45         # suma fibo(i - 2) y fibo(i - 1)
46         add    $v0, $v0, $s1
47         j      fin
48
49 es1:    li     $v0, 1
50         j      fin
51
52 es0:    li     $v0, 0
53
54 fin:    lw     $ra, 0($sp)
55         lw     $s0, 4($sp)
56         lw     $s1, 8($sp)
57         addi   $sp, $sp, 12
58         jr     $ra

```

---

Figura B3.8: fibo.s

En la figura B3.9 se pueden ver las 6 primeras líneas del triángulo ya construido.

Si numeramos los elementos del triángulo como se muestra en la parte izquierda de la figura B3.9, cada elemento se corresponde con el coeficiente binomial  $\binom{n}{k}$ , que a su vez se corresponde con el número de combinaciones de  $n$  objetos tomados en grupos de  $k$  elementos sin considerar el orden ( $C(n, k)$ ).

El valor de cada elemento del triángulo viene dado por la siguiente fórmula:

$$C(n, k) = \binom{n}{k} = \begin{cases} 1 & \text{si } n = k \\ 1 & \text{si } k = 0 \\ C(n-1, k-1) + C(n-1, k) & \text{en otro caso} \end{cases}$$

El objetivo del ejercicio es hacer un programa que permita calcular  $C(n, k)$  y permita también imprimir algunas filas del triángulo de Pascal.

El programa deberá tener, como mínimo, los siguientes procedimientos:

- a) Un procedimiento principal «main» que le muestre al usuario un menú con tres opciones:
  - 1) Calcular un coeficiente binomial o número combinatorio.

	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	...
$n = 0$	1						
$n = 1$	1	1					
$n = 2$	1	2	1				
$n = 3$	1	3	3	1			
$n = 4$	1	4	6	4	1		
$n = 5$	1	5	10	10	5	1	
...							

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
						...

Figura B3.9: Triángulo de Pascal. A la izquierda se muestra en forma de tabla, mientras que a la derecha se muestra de forma que se aprecia mejor la forma triangular y la simetría. En la segunda representación, cada elemento es la suma de los dos elementos situados sobre él (el que está arriba a la izquierda y el que está arriba a la derecha).

- 2) Mostrar el triángulo de Pascal.
- 3) Salir del programa.

Este procedimiento deberá llamar a «cmd\_combinaciones» o a «cmd\_triangulo» según la opción elegida por el usuario. Después de cada operación, se volverá a presentar el menú.

- b) Un procedimiento «pregunta\_int» que recibe la dirección de una cadena, la cual muestra por pantalla y espera a que el usuario introduzca un número entero, el cual devuelve.
- c) Un procedimiento «cmd\_combinaciones» que pregunte al usuario el valor de  $n$  y de  $k$ , llame al procedimiento «coeficiente» para calcular  $\binom{n}{k}$ , e imprima el resultado.
- d) Un procedimiento «cmd\_triangulo» que pregunte al usuario un número entero  $N$  y llame al procedimiento «triangulo» para que muestre las primeras  $N$  filas del triángulo de Pascal.
- e) Un procedimiento recursivo «coeficiente» para calcular el coeficiente binomial propiamente dicho **usando la fórmula explicada arriba**. Deberá recibir dos enteros y devolver otro.
- f) Un procedimiento «triangulo» que reciba un número entero  $N$  e imprima las primeras  $N$  filas del triángulo de Pascal. Este procedimiento debe llamar al anterior para calcular cada uno de los elementos a mostrar. Es suficiente con mostrar los elementos de cada fila separados por un espacio (sin que queden necesariamente alineados).

El fichero `pascal.s` contiene la implementación de algunos de los procedimientos mencionados arriba. Complete el programa prestando especial atención a respetar los convenios de programación explicados en las secciones 3.3.1 y 3.4.4.

Tenga en cuenta que esta forma de generar el triángulo de Pascal es muy ineficiente porque se calcula varias veces el mismo valor. Existen algoritmos mucho más eficientes pero no son el objetivo de esta práctica.

## B3.3. Proyecto de programación en MIPS

### B3.3.1. Objetivos

El objetivo del proyecto de programación es que el alumno demuestre que es capaz de entender y modificar correctamente un pequeño programa en ensamblador MIPS.

### B3.3.2. Pong

El programa que se estudiará y mejorará en esta práctica es una versión de PONG, uno de los primeros videojuegos disponibles comercialmente. PONG fue publicado por primera vez en 1972 por Atari y es posible encontrar gran cantidad de versiones de este juego hoy en día.

El juego está inspirado en el tenis de mesa. Participan dos jugadores, cada uno de los cuales maneja una raqueta que puede moverse hacia arriba o hacia abajo. El campo de juego es un rectángulo en el que se mueve una pelota. La pelota se mueve en línea recta hasta que rebota con la pared superior, con la pared inferior o con alguna de las raquetas, que los jugadores deberán de utilizar para evitar que la pelota llegue a la banda que defienden. Inicialmente, cada jugador está situado en el centro de cada una de las bandas izquierda y derecha del campo, y la pelota comienza a moverse lentamente en horizontal. La pelota incrementa su velocidad y cambia su dirección cada vez que rebota. Cuando la pelota llega a la banda derecha o a la izquierda, el jugador del lado contrario gana un punto y la pelota se vuelve a situar en el centro moviéndose lentamente.

Para la realización de la práctica se partirá de una versión del programa simplificada pero funcional. El alumno deberá mejorar el programa realizando una serie de modificaciones que se detallan en la sección **B3.3.4**. Para ello, deberá primero invertir suficiente tiempo en estudiar el código para entender cómo funciona. Es muy aconsejable leer detenidamente el código del programa `pong.s` a la vez que se leen las explicaciones presentes en este documento.

### B3.3.3. Descripción del programa inicial

El programa se puede encontrar en el fichero `pong.s`. Para su ejecución, deberemos utilizar la versión de MARS disponible en la página web de la asignatura (<http://ditec.um.es/etc/>, ver sección **B3.1.4**).

Si se carga el programa en MARS, se ensambla y se ejecuta, comenzará una partida de PONG en el panel de entrada y salida de ejecución<sup>5</sup>. El jugador (o jugadores) interactúa con el programa a través del teclado, para lo cual hay que tener en cuenta que el panel de entrada y salida debe tener el foco para recibir las pulsaciones de teclas (normalmente es suficiente con hacer click con el ratón en el cuadro de texto). Solo unas pocas teclas tienen alguna función asociada, como se puede ver en la tabla **B3.1**.

Tecla	Función
a	Mover la pala del jugador izquierdo hacia arriba.
z	Mover la pala del jugador izquierdo hacia abajo.
k	Mover la pala del jugador derecho hacia arriba.
m	Mover la pala del jugador derecho hacia abajo.
x	Terminar la ejecución del programa.

Tabla B3.1: Pulsaciones de teclas reconocidas por el programa `pong.s`.

El estado actual del juego se representa mediante caracteres en el panel de entrada y salida de MARS. La figura **B3.10** muestra ejemplos de la ejecución del programa inicial. La salida se limpia y se vuelve a generar completamente varias veces por segundo. La primera línea de la salida muestra la puntuación de los jugadores. Los límites del campo se representan con los caracteres «#» y «+», la pelota con el carácter «O» y las palas de los jugadores derecho e izquierdo con los caracteres «[» y «]», respectivamente. La última línea muestra varios números que proporcionan información sobre el funcionamiento del programa. El significado de estos valores, en el orden que aparecen, es:

1. Milisegundos transcurridos entre el inicio de las dos últimas actualizaciones de pantalla.

<sup>5</sup>Conviene ajustar el tamaño del panel de forma que quepa entero el campo de juego. Para ello, puede ser necesario utilizar las pequeñas flechas situadas en la esquina superior izquierda del panel.

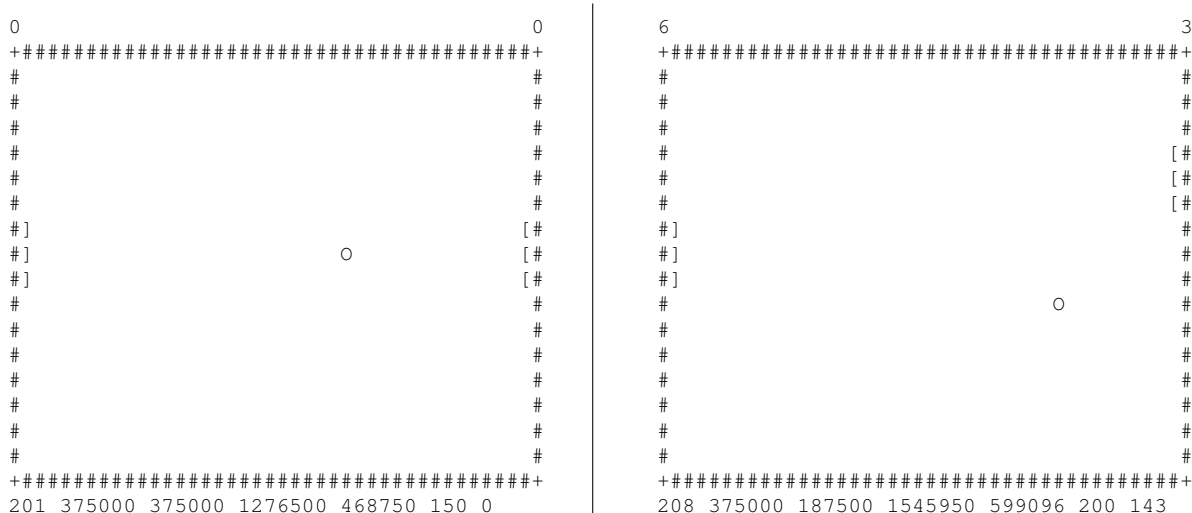


Figura B3.10: Dos ejemplos del aspecto de la salida del programa en diferentes momentos del juego.

2. Coordenada vertical del extremo superior de la pala del jugador izquierdo (`posicion_jugador_izquierdo_y`).
3. Coordenada vertical del extremo superior de la pala del jugador derecho (`posicion_jugador_derecho_y`).
4. Coordenada horizontal del centro de la pelota (`posicion_pelota_x`).
5. Coordenada vertical del centro de la pelota (`posicion_pelota_y`).
6. Componente horizontal de la velocidad de la pelota (`velocidad_pelota_x`).
7. Componente vertical de la velocidad de la pelota (`velocidad_pelota_y`).

El espacio de coordenadas tiene el origen en la esquina superior izquierda del campo, y las coordenadas crecen hacia la derecha y hacia abajo. El campo mide 2000000 unidades de ancho y 1000000 unidades de alto.

### Funcionamiento general

El programa mantiene en todo momento el estado actual del juego en unas pocas variables globales que se explican en una sección posterior de este documento. El funcionamiento habitual del programa consiste en repetir un bucle que realiza las siguientes acciones:

1. Comprueba si se ha pulsado alguna tecla, y actúa en consecuencia si es así.
2. Obtiene la hora actual del sistema (en milisegundos) y calcula el tiempo transcurrido desde la última vez que se actualizó la pantalla.
3. Si el tiempo transcurrido es mayor que el *tiempo entre fotogramas*:
  - Se actualizan las variables de estado del juego según el tiempo que haya transcurrido desde la última vez.
  - Se limpia la pantalla y se imprime el nuevo estado del programa.

El *tiempo entre fotogramas* se calcula al comienzo del programa y depende del número de fotogramas máximos por segundo deseados (`fps_max`), que inicialmente vale 5.

El programa utiliza un espacio de coordenadas para codificar el estado del juego que es distinto del espacio de coordenadas utilizado para la representación mediante caracteres. Esto es así porque el espacio de coordenadas de la representación viene impuesto por las limitaciones del dispositivo de salida (es decir, el panel de salida de MARS que simula la pantalla). La limitación principal es que las coordenadas tienen que ser enteros en rangos muy limitados (de 0 a 39 en horizontal y de 0 a 15 en vertical). Esto implica, por ejemplo, que no podemos utilizar este sistema de coordenadas para codificar la posición de la pelota con suficiente precisión para conseguir un movimiento suave (dentro de las limitaciones de la representación mediante caracteres y del simulador) y una aceleración gradual.

Habría dos formas sencillas de solucionar el problema:

- Utilizar números en coma flotante para representar la posición de la pelota y otros objetos, utilizando un espacio de coordenadas con el mismo rango que el de la pantalla.
- Utilizar números enteros para representar todas las posiciones, pero aumentar el rango del espacio de coordenadas para permitir mayor precisión efectiva. Es decir, tendremos un espacio de coordenadas para el modelo (con un tamaño del campo de  $2000000 \times 1000000$ ) y otro para la vista (con un tamaño del campo de  $40 \times 16$ ). En este caso habrá que hacer una traducción de las coordenadas del modelo a las de la vista cada vez que se necesite dibujar la pantalla.

Ambas soluciones son igual de buenas. En el programa de la práctica se ha elegido la segunda arbitrariamente.

### Datos del programa

En la sección de datos del programa (`.data`) se declaran las siguientes variables globales:

- Variables relativas a la configuración del juego, las cuales se inicializan al principio del programa y no se modifican durante el estado de la partida (se utilizan como constantes). Se muestra el valor por defecto entre paréntesis:

**alto\_campo:** Altura del campo de juego (1000000).

**ancho\_campo:** Anchura del campo de juego (2000000).

**aceleracion\_rebote\_x:** Incremento del valor absoluto de la componente horizontal de la velocidad cada vez que la pelota rebota contra una pala (25).

**aceleracion\_rebote\_y:** Incremento del valor absoluto de la componente vertical de la velocidad cada vez que la pelota rebota contra el extremo superior o inferior de una pala. El incremento efectivo en cada rebote es proporcional a la distancia del punto del rebote hasta el centro de la pala (150).

**alto\_jugador:** Tamaño de las palas de los jugadores ( $(\text{alto\_campo}/\text{alto\_dibujo}) \times 3$ ).

**velocidad\_jugador:** Incremento (o decremento) de la posición vertical de los jugadores cada vez que se pulsa la tecla correspondiente ( $\text{alto\_campo}/\text{alto\_dibujo}$ ).

**ancho\_dibujo:** Ancho del campo en la pantalla, en caracteres (40).

**alto\_dibujo:** Alto del campo en la pantalla, en caracteres (16).

**fps\_max:** Número máximo de fotogramas por segundo a dibujar (5).

- Variables relativas al estado de la partida actual:

**posicion\_pelota\_x:** Coordenada horizontal del centro de la pelota.

**posicion\_pelota\_y:** Coordenada vertical del centro de la pelota

**velocidad\_pelota\_x:** Componente horizontal de la velocidad de la pelota, en unidades por milisegundo.

**velocidad\_pelota\_y:** Componente vertical de la velocidad de la pelota, en unidades por milisegundo.

**posicion\_jugador\_izquierdo\_x:** Coordenada horizontal de la pala izquierda (se inicializa al valor  $(\text{ancho\_campo}/\text{ancho\_dibujo})/2$ ).

**posicion\_jugador\_izquierdo\_y:** Coordenada vertical del extremo superior de la pala izquierda.

**posicion\_jugador\_derecho\_x:** Coordenada horizontal de la pala derecha (se inicializa al valor  $(\text{ancho\_campo} - \text{ancho\_campo}/\text{ancho\_dibujo})/2$ ).

**posicion\_jugador\_derecho\_y:** Coordenada vertical del extremo superior de la pala derecha.

**puntuacion\_jugador\_izquierdo:** Puntos conseguidos por el jugador izquierdo.

**puntuacion\_jugador\_derecho:** Puntos conseguidos por el jugador derecho.

### Procedimientos del programa

**prod\_vector\_int:** Calcula el producto de un vector de dos enteros por un escalar entero.

Argumentos:

**v.x:** (Entero) Primera componente del vector.

**v.y:** (Entero) Segundo componente del vector.

**c:** (Entero) Escalar.

Resultados:

**ret.x:** (Entero)  $v.x \times c$ .

**ret.y:** (Entero)  $v.y \times c$ .

**suma\_vector:** Suma dos vectores de enteros.

Argumentos:

**a.x:** (Entero) Primera componente del primer vector.

**a.y:** (Entero) Segundo componente del primer vector.

**b.x:** (Entero) Primera componente del segundo vector.

**b.y:** (Entero) Segundo componente del segundo vector.

Resultados:

**ret.x:** (Entero)  $a.x \times b.x$ .

**ret.y:** (Entero)  $a.y \times b.y$ .

**dibuja\_campo:** Dibuja el campo de juego, la pelota y las palas según el estado actual de las variables globales.

Este procedimiento no recibe ningún argumento ni devuelve ningún resultado, solo produce salida por pantalla.

En primer lugar, el procedimiento realiza la transformación del espacio de coordenadas del modelo al espacio de coordenadas de la pantalla. Para ello, se utilizan operaciones aritméticas en coma flotante y se calculan las siguientes variables locales a partir de las variables globales:

**escala\_x:** Factor de escala horizontal. Asignada al registro  $\$f0$ .

Valor:  $\text{ancho\_dibujo}/\text{ancho\_campo}$

**escala\_y:** Factor de escala vertical. Asignada al registro  $\$f1$ .

Valor:  $\text{alto\_dibujo}/\text{alto\_campo}$

**e\_pelota\_x:** Posición horizontal escalada del centro de la pelota. Asignada al registro  $\$s0$ .

Valor:  $\text{posicion\_pelota\_x} \times \text{escala\_x}$

**e\_pelota\_y:** Posición vertical escalada del centro de la pelota. Asignada al registro  $\$s1$ .

Valor:  $\text{posicion\_pelota\_y} \times \text{escala\_y}$

**e\_jug\_izq\_y\_sup:** Posición vertical escalada del extremo superior de la pala del jugador izquierdo. Asignada al registro  $\$s2$ .

Valor:  $\text{posicion\_jugador\_izquierdo\_y} \times \text{escala\_y}$

**e\_jug\_izq\_y\_inf:** Posición vertical escalada del extremo inferior de la pala izquierda. Asignada al registro  $\$s3$ .

Valor:  $(\text{posicion\_jugador\_izquierdo\_y} + \text{alto\_jugador}) \times \text{escala\_y}$

**e\_jug\_izq\_x:** Posición horizontal escalada de la pala izquierda. Asignada al registro  $\$s4$ .

Valor:  $\text{posicion\_jugador\_izquierdo\_x} \times \text{escala\_x}$

**e\_jug\_der\_y\_sup:** Posición vertical escalada del extremo superior de la pala del jugador derecho. Asignada al registro  $\$s5$ .

Valor:  $\text{posicion\_jugador\_derecho\_y} \times \text{escala\_y}$

**e\_jug\_der\_y\_inf:** Posición vertical escalada del extremo inferior de la pala del jugador derecho. Asignada al registro  $\$s6$ .

Valor:  $(\text{posicion\_jugador\_derecho\_y} + \text{alto\_jugador}) \times \text{escala\_y}$

**e\_jug\_der\_x:** Posición horizontal escalada de la pala del jugador derecho. Asignada al registro  $\$s7$ .

Valor:  $\text{posicion\_jugador\_derecho\_x} \times \text{escala\_x}$

Para el cálculo del valor de estas variables, se utiliza el registro  $\$f2$  de forma temporal. Aunque las operaciones que necesitamos realizar requieren números en coma flotante ( $\text{escala\_x}$  y  $\text{escala\_y}$  son valores entre 0 y 1), tanto las variables globales de las que partimos como los resultados que necesitamos son valores enteros. Por tanto, es necesario realizar transferencias entre los bancos de registros de enteros y de coma flotante (utilizando las instrucciones  $\text{mtc1}$  y  $\text{mfc1}$ ), transferencias de memoria a los registros de coma flotante (utilizando la instrucción  $\text{lwc1}$ ) y conversiones entre ambos tipos de datos (utilizando las instrucciones  $\text{cvt.s.w}$  y  $\text{cvt.w.s}$ ).

Por ejemplo, para calcular  $\text{e\_jug\_izq\_y\_inf}$  se siguen los siguientes pasos:

1. Se carga en  $\$t0$  el contenido de la variable  $\text{posicion\_jugador\_izquierdo\_y}$ .
2. Se carga en  $\$t1$  el contenido de la variable  $\text{alto\_jugador}$ .
3. Se suman los valores cargados anteriormente, dejando el resultado en  $\$t0$ .
4. Se copia el contenido de  $\$t0$  a  $\$f2$  utilizando la instrucción « $\text{mtc1 } \$t0, \$f2$ ». Esta instrucción copia el contenido del registro literalmente, por lo que el resultado es que tendremos en  $\$f2$  el resultado de la suma anterior codificado como entero en complemento a dos.
5. Se utiliza la instrucción « $\text{cvt.s.w } \$f2, \$f2$ » para convertir el valor almacenado en  $\$f2$  en un número en coma flotante de simple precisión.
6. Se multiplica el valor contenido en  $\$f2$  por  $\text{escala\_y}$ , que está almacenado en  $\$f1$ . Se utiliza la instrucción « $\text{mul.s}$ » y el resultado se deja en el registro  $\$f2$ .

7. Se utiliza la instrucción `«cvt.w.s $f2, $f2»` para convertir el resultado en un entero representado en complemento a 2. Esta conversión realiza redondeo por truncamiento.
8. Se copia el contenido de `$f2` a `$s3` utilizando la instrucción `«mfc1 $s3, $f2»`. Esta copia es literal, por lo que en `$s3` queda almacenado el resultado final codificado en complemento a 2.

El resto de variables se calculan de forma análoga.

Obsérvese que se utilizan los registros `$f0`, `$f1` y `$f2` para almacenar valores en coma flotante y realizar cálculos con ellos. Estos registros no son preservados entre llamadas según el ABI de MIPS que utilizamos en las prácticas, por lo que no hay que almacenarlos en la pila.

Una vez calculadas estas variables, se procede a mostrar el tablero por pantalla:

1. Se imprime la pared superior, para lo que se imprime un carácter «+», se utiliza un bucle que imprime un carácter «#» en cada iteración, y finalmente se imprime un carácter «+» seguido de un salto de línea.
2. Se utilizan dos bucles anidados para imprimir las líneas intermedias del campo. En cada iteración se imprime una fila:
  - a) Se imprime el carácter «#».
  - b) Se utiliza el bucle interior para imprimir un carácter para cada columna de esta fila. En el cuerpo de este bucle, el registro `$t2` contiene la fila actual y el `$t3` contiene la columna actual. Para cada posición, se calcula qué hay que imprimir:
    - 1) Si es la posición de la pelota, se imprime el carácter «O».
    - 2) Si la columna actual coincide con la posición horizontal de la pala izquierda (`e_jug_izq_x`) y la fila actual es mayor o igual que `e_jug_izq_y_sup` pero menor que `e_jug_izq_y_inf`, se imprime un carácter «]».
    - 3) Si la columna actual coincide con la posición horizontal de la pala derecha (`e_jug_der_x`) y la fila actual es mayor o igual que `e_jug_der_y_sup` pero menor que `e_jug_der_y_inf`, se imprime un carácter «[».
    - 4) En otro caso, se imprime un espacio.
  - c) Se imprime el carácter «#» seguido de un salto de línea.
3. Se imprime la pared inferior de la misma forma que la superior.

**dibuja\_marcadores:** Dibuja los marcadores, alineando uno a la izquierda y otro a la derecha.

Este procedimiento no recibe ningún argumento ni devuelve ningún resultado, solo produce salida por pantalla.

El procedimiento imprime el marcador del jugador izquierdo, imprime cierto número de espacios (`nespacios`), imprime el marcador del jugador derecho, e imprime un salto de línea.

El número de espacios a imprimir debe ser el adecuado para que el marcador del jugador derecho quede correctamente alineado. Para calcularlo, se le resta a `ancho_dibujo` el número de cifras necesario para imprimir ambos marcadores. Para calcular el número de cifras, se utilizan dos bucles que dividen repetidamente entre 10 los marcadores.

**inicia\_bola:** Prepara el estado del juego para comenzar a disputar una bola. Es decir, coloca la bola en el centro del campo moviéndose lentamente y los jugadores en sus posiciones iniciales.

Argumentos:

**dir\_bola:** (Entero) Dirección inicial de la bola. Se utiliza el 0 para indicar que la bola se mueva hacia la izquierda y el 1 para que se mueva hacia la derecha.

El procedimiento se limita a calcular los valores iniciales necesarios para las variables globales relacionadas con la posición de la pelota, su velocidad y la posición de los jugadores.

La posición inicial de la pelota se ajusta de forma que el centro de la misma esté alineado con el centro de las palas, de forma que los rebotes iniciales sean horizontales (para facilitar el juego).

La componente horizontal de la velocidad de la pelota se establece a 150 o -150 unidades por milisegundo según la dirección inicial sea hacia la derecha o hacia la izquierda, respectivamente.

**inicia\_partida:** Pone a cero los marcadores y empieza una partida nueva.

Este procedimiento no recibe ningún argumento.

El procedimiento inicializa las variables correspondientes, para lo cual delega gran parte del trabajo en el procedimiento `inicia_bola`.

**avanza\_pong:** Actualiza el estado del juego.

Argumentos:

**transcurrido:** (Entero) Indica cuántos milisegundos han pasado desde la última actualización.

Este procedimiento es el que simula el juego propiamente dicho. Se encarga de mover la pelota teniendo en cuenta su velocidad actual y el tiempo que ha pasado desde la última actualización, detectar las colisiones de la pelota contra las paredes y las palas, y actualizar la velocidad y dirección de la pelota cuando rebota.

La posición de la pelota se actualiza mediante sencillas operaciones vectoriales para las cuales se utilizan los procedimientos `prod_vector_int` y `suma_vector`.

Una vez calculada la nueva posición de la pelota, se comprueba si se ha producido algún rebote. Para ello, se compara la posición de la pelota con las coordenadas de las dos palas y las cuatro paredes.

En caso de detectarse una colisión con una de las palas, la dirección y velocidad de la pelota cambian. La componente horizontal de la velocidad cambia de signo y su valor absoluto se ve incrementado en `aceleracion_rebote_x` unidades. Por su parte, la componente vertical cambiará de forma diferente dependiendo del punto de contacto de la pelota y la pala siguiendo la siguiente expresión:

$$\Delta \text{velocidad\_pelota\_y} = \text{aceleracion\_rebote\_y} \times \frac{\text{posicion\_pelota\_y} - (p + \frac{\text{alto\_jugador}}{2})}{\frac{\text{alto\_jugador}}{2}}$$

En la expresión anterior,  $p = \text{posicion\_jugador\_derecho\_y}$  en el caso de la pala derecha y  $p = \text{posicion\_jugador\_izquierdo\_y}$  en el caso de la pala izquierda. Obsérvese además que la velocidad de la pelota puede aumentar o disminuir dependiendo de si el punto de contacto con la pala se encuentra por encima o por debajo del centro de la pala y de la dirección de la pelota en el momento del contacto. Para calcular este resultado se utiliza aritmética en coma flotante y se realizan las conversiones de datos necesarias mediante las instrucciones `mfcl`, `mtcl`, `cvt.w.s` y `cvt.s.w`.

En caso de colisión con una de las paredes izquierda o derecha, se incrementa la puntuación del jugador correspondiente y se llama al procedimiento `inicia_bola`.

Finalmente, en caso de colisión con las paredes superior o inferior, se invierte el signo de la componente vertical de la velocidad de la pelota.

**keyio\_poll\_key:** Realiza sondeo (polling) del teclado.

Este procedimiento no recibe argumentos.

Resultado:

**tecla:** (Entero) Código del carácter correspondiente a la última tecla pulsada, o 0 si no se ha pulsado ninguna tecla desde la última vez que se comprobó.

El sondeo se realiza como se explica en el tema 7 de la asignatura. En concreto, se puede consultar el boletín de prácticas B7.1 para más detalles.

**poll\_entrada:** Comprueba si se ha pulsado alguna tecla y la procesa si es así.

Este procedimiento no recibe argumentos.

Resultado:

**seguir:** (Entero) Vale 0 si se ha pulsado una tecla que indique que la partida debe finalizar, o 1 en cualquier otro caso.

El procedimiento llama a `keyio_poll_key` para ver si se ha pulsado alguna tecla. Si se ha pulsado alguna, comprueba si es una de las teclas mostradas en la tabla B3.1 y en ese caso actualiza las variables globales de forma adecuada.

**main:** Esta es la función principal del programa que ejecuta el bucle que se explica en la sección B3.3.3.

#### B3.3.4. Modificaciones propuestas al programa

Para completar la práctica, cada grupo deberá elegir varias de las modificaciones propuestas en esta sección e implementarlas en su programa. Cada modificación o mejora tiene asociada una puntuación máxima acorde con su dificultad, de forma que cada grupo conseguirá una puntuación diferente según qué mejoras haya elegido realizar. Por supuesto, es posible obtener una puntuación parcial si alguna mejora no se implementa correctamente o no se explica suficientemente bien (tanto en la memoria como a la hora de la entrevista de prácticas).

Las mejoras realizadas deben explicarse claramente en la memoria de la práctica. La explicación de cada mejora no ha de ser demasiado extensa, pero debe incluir, al menos:

- Explicación general de la implementación de la mejora. Se valorará la discusión de otras alternativas consideradas por el grupo para resolver el problema y la justificación de la estrategia de implementación elegida.
- Lista de cambios realizados al código.
- Lista de pruebas realizadas para comprobar el funcionamiento de la mejora, con instrucciones para reproducirlas y capturas de la salida del programa cuando proceda.
- Cualquier aclaración que se considere relevante para la evaluación del trabajo realizado.

Cualquier alumno puede proponer mejoras alternativas a las propuestas en este enunciado. Para ello, deben consultar con el profesor de la asignatura, quien decidirá si la mejora se puede tener en cuenta y qué puntuación máxima tendrá asociada. Las mejoras propuestas por un grupo serán públicas de forma que otros grupos podrán realizar la misma mejora, pero el grupo que proponga la mejora originalmente dispondrá de un 20 % extra para la puntuación máxima asociada a la mejora.

A continuación, se muestra la lista de mejoras propuestas:

1. Hacer que la pelota tenga una aceleración uniforme.

En el programa original, la pelota sigue un movimiento uniforme (con aceleración 0) excepto cuando rebota contra las palas o las paredes. Para esta mejora, se debe modificar el programa de forma que la pelota siga un movimiento uniformemente acelerado. La aceleración deberá estar especificada mediante una variable global.

Puntuación máxima: 8 puntos.

2. Hacer que se puedan controlar las palas con teclas mayúsculas.

En programa base solo responde a pulsaciones de teclas minúsculas (por ejemplo: no funciona si el bloqueo de mayúsculas está activado). Para esta mejora se debe solucionar este problema.

En caso de que se implemente la mejora 13 y ésta a la vez, la puntuación de esta mejora será mayor siempre y cuando el usuario pueda configurar sus teclas usando letras mayúsculas o minúsculas y el programa las reconozca siempre en ambos casos.

Puntuación máxima: 10 puntos (15 puntos si se implementa también la mejora 13).

3. Hacer que se acabe la partida al alcanzar una puntuación determinada.

Se deberá decidir la puntuación máxima que se debe alcanzar para dar una partida por terminada. El alumno puede decidir si se exige que haya diferencia de más de un punto entre ambos jugadores.

Una vez alcanzada la puntuación objetivo, el programa debe mostrar un mensaje indicando qué jugador ha ganado la partida y ofrecer la posibilidad de empezar una nueva partida o salir del programa.

Puntuación máxima: 10 puntos.

4. Hacer que los marcadores sigan las reglas del tenis.

El programa deberá mantener un conteo de puntos, juegos y sets siguiendo las reglas habituales del tenis (que se pueden consultar, por ejemplo, en <http://es.wikipedia.org/wiki/Tenis#Puntuaci.C3.B3n>).

Esta mejora requiere que se implemente también la mejora 3.

Puntuación máxima: 15 puntos.

5. Hacer que la pelota se acelere también al rebotar contra las paredes superior e inferior (no solo contra las palas como en el programa original).

La velocidad sólo debe cambiar en valor absoluto, la dirección del vector debe permanecer inalterada respecto a la que resulta actualmente cuando se produce un rebote. El incremento de velocidad de la pelota en cada rebote deberá ser aleatorio y comprendido entre 1 y 100 unidades. Para conseguir esto, se utilizará la llamada al sistema de MARS que permite generar números aleatorios.

Puntuación máxima: 15 puntos.

6. Añadir un menú inicial.

Modificar el programa para que muestre un menú con varias opciones cuando empieza, en lugar de comenzar una partida directamente. El menú deberá incluir al menos las siguientes opciones:

- Comenzar partida.
- Mostrar instrucciones del juego.
- Salir del programa.

A este menú se le deberán añadir más opciones si se implementan otras mejoras.

Puntuación máxima: 10 puntos.

7. Añadir sonido.

Utilizar las llamadas al sistema que permiten generar sonidos ofrecidas por MARS para añadir efectos sonoros al programa.

Puntuación máxima: 30 puntos.

8. Hacer que los jugadores se puedan mover horizontalmente.

En el programa base, los jugadores se encuentran siempre a en la misma posición horizontal (concretamente: a 25000 unidades de distancia de la banda que defienden). Esta mejora consiste en permitir que se puedan mover horizontalmente mediante pulsaciones de teclas.

Puntuación máxima: 15 puntos.

9. Permitir al usuario cambiar diversas opciones del programa.

Esta mejora consiste en añadir opciones para cambiar el tamaño del tablero, la frecuencia de actualización de la pantalla (FPS), la velocidad de la pelota, aceleración, tamaño de las palas, etc.

Esta mejora requiere que se implemente primero la mejora 6.

Puntuación máxima: 15 puntos.

10. Hacer que uno de los jugadores (o los dos) se mueva solo.

El programa base solo permite que dos jugadores humanos se enfrenten en un partida (o un solo jugador manejando una pala con cada mano). Esta mejora consiste en hacer que una de las palas pueda moverse automáticamente.

Para ello, es suficiente con usar un algoritmo muy sencillo de forma que el programa decida cada cierto tiempo si mover su pala hacia arriba o abajo en función de la posición relativa de la pala y la pelota.

Habrà que tener cuidado de evitar que la pala de la máquina se pueda mover mucho más rápido que la pala del jugador humano. También se valorará si se añade un cierto no determinismo para permitir que el jugador humano gane alguna partida.

Esta mejora requiere que se implemente primero la mejora 6 para que se pueda elegir si la partida es entre dos jugadores humanos o entre un humano y la máquina.

Puntuación máxima: 60 puntos.

11. Permitir al usuario elegir la dirección del juego.

En el programa base, así como el PONG original, los jugadores están situados a la derecha y a la izquierda de la pantalla y el movimiento de la pelota es principalmente horizontal. Esta mejora consiste en permitir al usuario jugar colocando las palas en las bandas superior e inferior y que la pelota se mueva principalmente de arriba a abajo.

El programa resultante deberá funcionar correctamente tanto en modo horizontal como vertical.

Esta mejora requiere que se implemente primero la mejora 6.

Puntuación máxima: 40 puntos.

12. Permitir al usuario elegir la dirección del juego especificando cualquier ángulo.

Esta mejora propuesta es similar a la 11, pero el usuario podría especificar cualquier ángulo para orientar el campo.

Puntuación máxima: 60 puntos.

13. Permitir al usuario cambiar las teclas que se usan para controlar el juego.

El usuario podrá elegir las teclas para controlar el movimiento de las palas y cualquier otra función del juego.

Esta mejora requiere que se implemente primero la mejora 6. También afecta a la mejora 2.

Puntuación máxima: 15 puntos.

## 14. Añadir un histórico de mejores puntuaciones.

Consiste en añadir un ranking de puntuaciones obtenidas por los jugadores, ordenándolas por la diferencia entre la puntuación de los dos jugadores. Para ello, el programa le preguntará el nombre a los jugadores cuando acabe cada partida.

Esta mejora requiere que se implementen primero la mejoras 6 y la mejora 3. Habrá que añadir una opción al menú para permitir consultar el ranking.

Puntuación máxima: 15 puntos si los marcadores se almacenan solo en memoria, 30 puntos si se almacenan en disco.

## 15. Hacer que aparezcan varias pelotas.

Esta mejora consiste en que se pueda jugar con varias pelotas a la vez. El alumno debe decidir cómo aparece más de una pelota. Por ejemplo: podrían ir apareciendo al alcanzar determinada puntuación, o cuando algún jugador pulse determinada tecla.

Puntuación máxima: 30 puntos.

## 16. PONG a 4 palas.

Permitir que participen 4 jugadores en una partida, uno en cada una de las bandas del campo. Las palas adicionales se manejarían con otras teclas distintas a las de la tabla B3.1. En esta modalidad de juego, la pelota no rebotaría al alcanzar las paredes superior o inferior, sino que se incrementaría la puntuación de todos los jugadores menos el que defendía la banda correspondiente.

Puntuación máxima: 30 puntos.

## 17. Añadir efectos especiales.

Esta mejora consiste en añadir animaciones (mediante caracteres) para conseguir efectos visuales. Por ejemplo: la pelota podría explotar cuando alcance una de las bandas.

Puntuación máxima: 30 puntos.

## 18. Mostrar la puntuación con números grandes.

La puntuación de los jugadores se mostraría mediante caracteres de mayor tamaño dibujados mediante varios caracteres normales, como se muestra en la figura B3.11.

Puntuación máxima: 30 puntos.

## 19. Permitir controlar el saque a los jugadores

Esta mejora consiste en hacer que sean los jugadores los que saquen pulsando el espacio. En lugar de aparecer la pelota en el centro moviéndose, ésta aparecería en la pala de uno de los jugadores que podría moverse con ella hasta para decidir desde dónde sacar. Cuando el jugador pulsara el espacio, la pelota se pondría en movimiento. Cada vez sacaría un jugador (o bien sacaría el jugador que anotó el último punto).

Puntuación máxima: 20 puntos.

## 20. Poner obstáculos en medio del campo.

Para esta mejora, se debe añadir diversos obstáculos en el campo contra los que la pelota debe rebotar. Los obstáculos pueden ser fijos o diferentes para cada punto.

Puntuación máxima: 15 puntos si los obstáculos son fijos, 25 si cambian para cada punto.

```

****                               ****
 *                               *
 ***                             ****
 *                               *
 ****                             ****
+#####+
#                               #
#                               #
#                               #
#                               #
#                               #
#]                               [#
#]                               [#
#]                               [#
#                               #
#                               #
#                               #
#                               #
#                               #
+#####+
201 375000 375000 1276500 468750 150 0
    
```

Figura B3.11: Dos ejemplos del aspecto de la salida del programa en diferentes momentos del juego.

21. Reducir el número de instrucciones necesarios para dibujar la pantalla y actualizar el estado del juego

El programa base no ha sido optimizado y utiliza más instrucciones de las estrictamente necesarias para actualizar el estado de la partida y dibujar la pantalla. Esta mejora consiste en optimizar el programa, respetando siempre todas las convenciones de programación vistas en clase. El número de instrucciones antes y después de realizar las optimizaciones se debe medir mediante la herramienta «Instruction counter» de MARS.

Puntuación máxima: 35 puntos (para una reducción de al menos un 30 % de instrucciones ejecutadas por iteración).

22. Eliminar (o reducir) el parpadeo usando un buffer.

Para mostrar el estado del juego, el procedimiento `dibuja_campo` realiza muchas llamadas al sistema (una por carácter), de forma que se puede apreciar a simple vista cómo se dibuja el campo incluso si el programa se ejecuta en una máquina muy rápida. El parpadeo que se produce es molesto a la hora de jugar.

Es posible eliminar el parpadeo casi completamente si se utiliza un buffer para generar la imagen completa del campo de juego, en lugar de imprimir carácter a carácter. De esta forma, una vez que se ha generado una imagen actualizada en memoria sin utilizar ninguna llamada al sistema, se limpia la pantalla y se imprime el contenido del buffer inmediatamente.

Puntuación máxima: 45 puntos.

23. Utilizar variables en coma flotante para representar el estado del juego.

Como se explica en la sección B3.3.3, el programa base utiliza variables enteras para representar todo el estado del juego, por lo que es necesario realizar una transformación de un espacio de coordenadas interno al espacio de coordenadas utilizado para dibujar la pantalla. Esta mejora consistiría en cambiar la forma de trabajar del programa de forma que se utilicen variables en coma flotante para representar el estado del juego.

Puntuación máxima: 25 puntos.

### B3.3.5. Criterios de evaluación

El 80 % de la nota final de la práctica depende de la puntuación total de todas las mejoras realizadas. Se alcanzaría la puntuación máxima con 100 puntos, aunque la puntuación máxima total de las mejoras intentadas por el alumno puede ser mayor (de esta forma se pueden incrementar las posibilidades de obtener buena nota si no todas las mejoras intentadas funcionan perfectamente). El 20 % restante de la nota dependerá de la calidad general del programa, la documentación entregada y el resultado de la prueba de evaluación individual que se menciona a continuación.

Una vez evaluado el trabajo, se convocará a los alumnos a una prueba de evaluación individual en la que tendrán que demostrar su conocimiento del programa entregado mediante la realización de pequeños ejercicios que implicarán modificar dicho programa. El resultado de esta prueba determinará si la práctica está aprobada.

A la hora de evaluar el trabajo, se consideraran (entre otros) los siguientes aspectos:

- Funcionamiento correcto del programa con las mejoras realizadas.
- Claridad del código entregado.
- Uso correcto de todos los convenios de programación vistos en la asignatura. No será posible aprobar la práctica si no se siguen correctamente los convenios de programación explicados en clase.
- Completitud, claridad y concisión de la memoria entregada.

Para obtener una nota de sobresaliente será necesario realizar satisfactoriamente al menos una mejora valorada en 30 puntos o más.

### B3.3.6. Requisitos de entrega

La práctica se entregará por medio de SUMA empaquetada en un archivo “.tar.gz” o “.zip”. El archivo debe contener al menos:

- Información suficiente para identificar de forma sencilla e inequívoca a los miembros del grupo.
- Memoria explicativa de la práctica en formato PDF.
- Código fuente de la práctica.
- Casos de prueba o ejemplos de ejecución mostrando todas las características implementadas.

Se debe entregar un solo archivo por cada grupo de prácticas. El nombre del archivo debe seguir el siguiente formato:

`pong-dniA-dniB.extensión`

Donde *dniA* y *dniB* son los DNI de los integrantes del grupo ordenados ascendentemente y *extensión* es «tar.gz» o «zip».