Capítulo 2

El repertorio de instrucciones

2.1. Introducción

El funcionamiento de la CPU está determinado por las instrucciones que ejecuta. Estas instrucciones se denominan *instrucciones máquina* o *instrucciones del computador*. Al conjunto de instrucciones distintas que puede ejecutar la CPU se le denomina *repertorio de instrucciones* de la CPU.

En los computadores actuales las instrucciones se representan como números y los programas se pueden almacenar en memoria (concepto de *programa almacenado*). Cada instrucción máquina debe contener la información que necesita la CPU para su ejecución. Los elementos constitutivos de una instrucción máquina son:

- **Código de operación**: especifica la operación a realizar (suma, E/S, etc.). La operación se indica mediante un código binario.
- **Referencia a operandos fuente**: la operación puede implicar a uno o más operandos fuente, es decir, operandos que son entradas para la instrucción.
- **Referencia al operando resultado**: la operación puede producir un resultado.
- **Referencia a la siguiente instrucción**: dice a la CPU de dónde captar la siguiente instrucción tras completarse la ejecución de la instrucción actual.

En la mayoría de los casos la siguiente instrucción a captar sigue inmediatamente a la instrucción en ejecución. En tales casos no hay referencia explícita a la siguiente instrucción. Cuando sea necesaria una referencia explícita, debe suministrarse la dirección de memoria principal o virtual.

Los operandos fuente y resultado pueden estar en algunas de las siguentes áreas:

- **Memoria principal o virtual**: como en las referencias a instrucciones siguientes, debe indicarse la dirección de memoria principal o de memoria virtual.

- **Registro de la CPU**: salvo raras excepciones, una CPU contiene uno o más registros que pueden ser referenciados por instrucciones máquina. Si existe más de uno, cada registro tendrá asignado un número único, y la instrucción debe contener el número del registro deseado.

- **Dispositivo de entrada/salida (E/S)**: la instrucción debe especificar el módulo y dispositivo de E/S para la operación. En el caso de E/S asignadas en memoria, se dará otra dirección de memoria principal o virtual.

2.2. Representación de las instrucciones

Dentro del computador, cada instrucción se representa por una secuencia de bits. La instrucción está dividida en campos, correspondientes a los elementos constitutivos de la misma. La descripción de la instrucción en campos y bits se denomina *formato de instrucción*. La figura 2.1 muestra un ejemplo sencillo de formato de instrucción. En la mayoría de los repertorios de instrucciones se emplea más de un formato. Durante su ejecución, la instrucción se escribe en un registro de instrucción (IR) de la CPU. La CPU debe ser capaz de extraer los datos de los distintos campos de la instrucción para realizar la operación requerida.

Es difícil manejar las representaciones binarias de las instrucciones máquina, por ello, se utilizan *representaciones simbólicas* de estas instrucciones. Tanto los operandos como los codigos de operación se suelen representar simbólicamente. Por ejemplo, la instrucción

```
ADD R, Y
```

puede significar "sumar el valor contenido en la posición `Y` al contenido en el registro `R`".

Es raro encontrar ya programadores en lenguaje máquina. La mayoría de los programas actuales se escriben en un lenguaje de alto nivel o, en su ausencia, en lenguaje ensamblador.

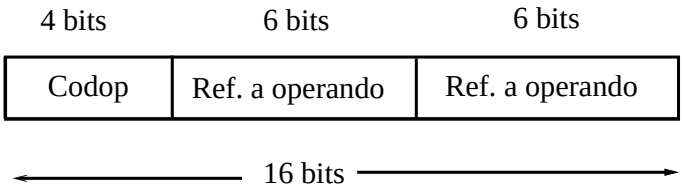

Figura 2.1: Un formato de instrucciones sencillo

2.3. Diseño del repertorio de instrucciones

Uno de los aspectos más interesantes y más analizados del diseño de un computador, es el diseño del repertorio de instrucciones del lenguaje máquina. El diseño de un repertorio de instrucciones es muy complejo, ya que afecta a muchos aspectos del computador. El repertorio de instrucciones define muchas de las funciones realizadas por la CPU y tiene, por tanto, un efecto significativo sobre la implementación de la misma. El repertorio de instrucciones es el medio que tiene el programador para controlar la CPU.

Los aspectos más importantes a tener en cuenta en el diseño del repertorio de instrucciones son:

- **Repertorio de operaciones**: cuántas y qué operaciones considerar, y cuán complejas deben ser.
- **Tipos de datos**: los distintos tipos de datos con los que se efectúan operaciones. Hay dos características importantes de los repertorios de instrucciones que dividen las arquitecturas de registros de propósito general. Ambas están relacionadas con la naturaleza de los operandos. La primera es el número de operandos (dos o tres) que pueden tener las instrucciones de la ALU, y la segunda, el número de operandos que se pueden direccionar en memoria. Las combinaciones posibles son:
  - *Registro-registro* (también llamada *carga/almacenamiento*)
  - *Registro-memoria*
  - *Memoria-memoria*
- **Formatos de instrucciones**: longitud de la instrucción (en bits), número de direcciones, tamaño de los distintos campos, etc.
- **Registros**: número de registros de la CPU que pueden ser referenciados por instrucciones, y su uso.

- **Direccionamiento**: el modo o modos de direccionamiento mediante los cuales puede especificarse la dirección de un operando.

Estos aspectos están fuertemente interrelacionados, y deben considerarse conjuntamente en el diseño de un repertorio de instrucciones.

2.4. Direccionamiento

Independientemente que una arquitectura sea registro-registro o permita que cualquier operando sea una referencia a memoria, debe definir cuántas direcciones de memoria son interpretadas y como se especifican. En esta sección trataremos ambas cuestiones.

2.4.1. Interpretación de las direcciones de memoria

¿Cómo se interpreta una dirección en memoria? Es decir, ¿qué objeto es accedido como una función de la dirección y la longitud? En años recientes casi todos los fabricantes de computadoras han adoptado como estándar una celda de 8 bits, que recibe el nombre de *byte*. Los bytes se agrupan a su vez en *palabras*. Una computadora con palabras de 32 bits tiene 4 bytes/palabra, mientras que una con palabras de 64 bits tiene 8 bytes/palabra. La importancia de las palabras es que casi todas las instrucciones operan con palabras enteras; por ejemplo, suman dos palabras. Así , una máquina de 32 bits tiene registros de 32 bits e instrucciones para manipular palabras de 32 bits, mientras que una máquina de 64 bits tiene registros de 64 bits e instrucciones para transferir, sumar, restar y manipular palabras de 64 bits.

En la mayoría de las arquitecturas nos encontramos con un requisito denominado *restricción de alineación* para el acceso a las direcciones de memoria. Por ejemplo, en una arquitectura MIPS, cuyo tamaño de palabra es siempre 32 bits, las palabras deben comenzar siempre en direcciones múltiplo de 4 (puesto que cada palabra se compone de 4 celdas o bytes). Esta restricción de alineación facilita la transferencia de datos más rápidamente.

Otro aspecto curioso y molesto, relacionado con la forma en que se referencian y se representan los bytes dentro de una palabra y los bits dentro de un byte, es el siguiente. Los bytes de una palabra pueden numerarse de izquierda a derecha o de derecha a izquierda. A primera vista podría parecer que esta decisión carece de importancia pero, sin embargo, tiene implicaciones importantes. Hay dos convenios diferentes para clasificar los bytes de una palabra:

- *Big Endian*: usa la dirección del byte de más a la izquierda o de "mayor peso" como dirección de palabra.
- *Little Endian*: usa el byte de más a la derecha o de "menor peso" como dirección de palabra.

2.4.2. Modos de direccionamiento

Se denominan modos de direccionamiento a aquellos algoritmos empleados por el procesador para calcular las direcciones de las instrucciones y datos.

Los operandos de la instrucción pueden venir especificados por un registro, una constante o una posición de memoria. Cuando se utiliza una posición de memoria la dirección real de memoria especificada por el modo de direccionamiento se denomina *dirección efectiva*.

El campo o campos de dirección en un formato de instrucción usual está bastante limitado. Nos gustaría poder referenciar un rango grande de posiciones de memoria principal o, en algunos sistemas, de memoria virtual. Para conseguir este objetivo se han empleado diversas técnicas de direccionamiento. Todas ellas implican algún compromiso entre el rango de direcciones y/o flexibilidad de direccionamiento de una parte, y por otra, el número de referencias a memoria y/o la complejidad de cálculo de las direcciones. En esta sección analizamos las técnicas de direccionamiento más comunes.

- **Direccionamiento inmediato**: el operando es una constante cuyo valor se almacena en el campo operando de la instrucción. La ventaja del direccionamiento inmediato es que una vez captada la instrucción no se requiere una referencia a memoria para obtener el operando. La desventaja es que el tamaño del número está restringido a la longitud del campo de direcciones.
- **Direccionamiento directo**: el operando se encuentra almacenado en memoria en la posición indicada por el campo operando. La limitación es que proporciona un espacio de direcciones reducido.
- **Direccionamiento indirecto**: la instrucción contiene en el campo operando la dirección de una posición de memoria en la que se almacena la dirección del operando deseado. La desventaja es que la ejecución de la instrucción requiere dos referencias a memoria para capturar el operando: una para captar su dirección y otra para obtener su valor.

- **Direccionamiento de registros**: el operando referenciado se encuentra en un registro. El número de registro se especifica dentro de la instrucción en el campo operando. Las ventajas son (1) que sólo es necesario un campo pequeño de direcciones en la instrucción y (2) que no se requieren referencias a memoria. El tiempo de acceso a un registro interno es mucho menor que a la memoria principal. La desventaja es que el espacio de direcciones está muy limitado.

- **Direccionamiento indirecto con registro**: la instrucción contiene en el campo operando el número del registro en el que se almacena la dirección de memoria del operando deseado. La ventaja es que este direccionamiento emplea una referencia menos a memoria que el direccionamiento indirecto.

- **Direccionamiento con desplazamiento**: el campo operando contiene una dirección relativa o desplazamiento D. La instrucción también especifica, implícita o explícitamente, otras posiciones de memoria de almacenamiento R (usualmente registros del procesador) conteniendo información adicional de direccionamiento. La dirección efectiva se calcula a través de una suma (D+R). Los tres usos más comunes del direccionamiento con desplazamiento son:

  - *Desplazamiento relativo*: el registro referenciado implícitamente es el contador de programa (PC). La dirección efectiva es un desplazamiento relativo a la dirección de la instrucción.
  - *Direccionamiento con registro-base*: el registro referenciado (implícita o explícitamente) contiene una dirección de memoria, y el campo de dirección contiene un desplazamiento desde dicha dirección.
  - *Indexado*: el campo de operando referencia una dirección de memoria y el registro referenciado contiene un desplazamiento positivo desde esa posición. Variaciones de este tipo son los métodos de direccionamiento con autoincremento y autodecremento.

En la figura 2.2 se muestra gráficamente un resumen de los modos de direccionamientos vistos aquí.

2.5. Operaciones del repertorio de instrucciones

El número de códigos de operación diferentes varía ampliamente de una máquina a otra. Sin embargo, en todas las máquinas podemos encontrar los mismos tipos generales de operaciones. Una clasificación típica y útil es la siguiente:

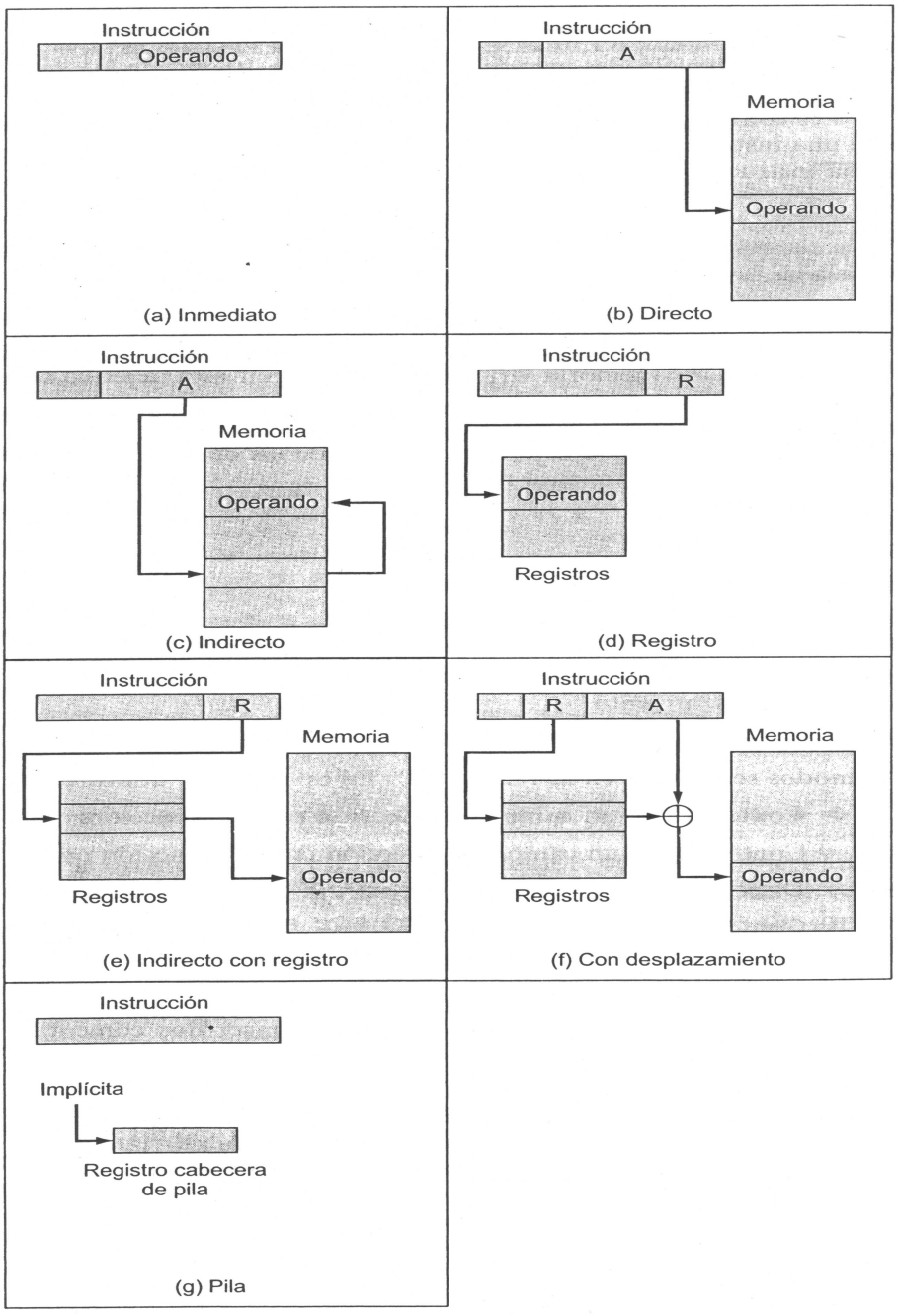

Figura 2.2: Modos de direccionamiento

- **Transferencias de datos**. Las instrucciones que transfieren datos entre memoria y registros se denominan instrucciones de *transferencia de datos*. Para acceder a una *palabra* en memoria, la instrucción debe proporcionar la *dirección de memoria*. La instrucción de transferencia que mueve datos de memoria a algún registro se denomina carga (*load*). La instrucción complementaria, llamada almacenar (*store*), transfiere datos de un registro a memoria.

- **Aritméticas**. La mayoría de las máquinas proporcionan las operaciones artiméticas básicas de suma, resta, multiplicación y división. Estas se tienen siempre para números enteros con signo y, a menudo, para números en coma flotante. Otras operaciones posibles son, por ejemplo, cálculo del valor absoluto, cambiar el signo al operando o incrementar o decrementar el operando.

- **Lógicas**. La mayoría de las máquinas también disponen de diversidad operaciones para manipular bits individuales dentro de una palabra o de otra unidad direccionable. Están basadas en operaciones booleanas.

- **De entrada/salida**. Las instrucciones de E/S se pueden hacer corresponder fácilmente con las órdenes de E/S que la CPU envía a un módulo de E/S, a menudo hay una simple relación de uno a uno.

- **De control del sistema**. Estas instrucciones son, por lo general, instrucciones privilegiadas que pueden ejecutarse sólo mientras el procesador está en un estado privilegiado concreto o está ejecutando un programa de una zona privilegiada específica de memoria. Normalmente estas instrucciones están reservadas para el sistema operativo.

- **De control de flujo**. En todos los tipos de operaciones discutidos hasta aquí, la siguiente instrucción a ejecutar es la inmediatamente posterior, en memoria, a la instrucción en curso. Sin embargo, una fracción significativa de las instrucciones de cualquier programa tienen como misión cambiar la secuencia de ejecución de instrucciones. La operación que realiza la CPU es actualizar el contador de programa para que contenga la dirección de alguna de las instrucciones que hay en memoria. Las operaciones de control de flujo que se pueden encontrar en los repertorios de instrucciones son:

  - *Instrucciones de bifurcación*, también llamadas de "salto". Tienen como uno de sus operandos la dirección de la siguiente instrucción a ejecutar.
  - *Instrucciones de salto condicional*. Se efectúa la bifurcación (se actualiza el contador de programa con la dirección especificada en el operando) sólo si se cumple una condición dada, en caso contrario se ejecuta la instrucción siguiente de la secuencia (se incrementa el contador de programa de la forma habitual).

- *Instrucciones de llamada a subrutina.* En cualquier punto del programa se puede invocar o *llamar* a la subrutina. Se ordena al computador que pase a ejecutar la subrutina y que retorne después al punto en que tuvo lugar la llamada. El uso de subrutinas requiere por tanto dos instrucciones básicas: una instrucción de llamada, que produce una bifurcación desde la posición actual al comienzo de la subrutina, y una instrucción de retorno de la subrutina al lugar desde el que se llamó.

RISC *versus* CISC

A finales de los años setenta se efectuaron muchos experimentos con instrucciones muy complejas, que eran posibles gracias al intérprete. A casi nadie se le ocurría diseñar máquinas más sencillas, pero hubo varios grupos que se opusieron a la tendencia y comenzaron a diseñar chips de CPU VLSI que no utilizaban interpretación. Ellos acuñaron el término RISC para este concepto.

La idea inicial era hacer instrucciones simples que pudieran ejecutarse con rapidez. Pronto se vio que la clave para un buen rendimiento era diseñar instrucciones que pudieran iniciarse rápidamente. El tiempo real que una instrucción tardaba era menos importante que el número de instrucciones que podían iniciarse por segundo. La característica que llamó la atención fue el número relativamente pequeño de instrucciones disponibles, por lo regular unas 50. Este número era mucho menor que las 200 o 300 que tenían computadoras como la DEC VAX.

El acrónimo RISC significa **computadora de conjunto de instrucciones reducido** (*Reduced Instruction Set Computer*), lo que contrasta con CISC que significa **computadora de conjunto de instrucciones complejo** (*Complex Intruction Set Computer*). Hoy en día poca gente piensa que el tamaño del conjunto de instrucciones sea crucial, pero el nombre ha persistido.

2.6. Repertorio de instrucciones del MIPS

En lo que resta del capítulo se estudiará el repertorio de instrucciones de un computador real. El repertorio de instrucciones escogido proviene del MIPS, usado por NEC, Nintendo, Silicon Graphics y Sony, entre otros, y es un típico ejemplo de los diseñados a principios de los años 80.

2.6.1. Tipos de instrucciones MIPS

Los tipos básicos de instrucciones que soporta el MIPS son los siguientes:

- Aritméticas y lógicas
- Transferencia de datos
- Salto condicional
- Bifurcación

Las características más importantes de estas instrucciones en el procesador MIPS son:

- La longitud de todas las instrucciones MIPS es de 32 bits.
- Los operandos de las operaciones aritméticas son siempre registros.
- El acceso a memoria se hace a través de las operaciones de carga y almacenamiento (transferencia de datos). MIPS es, por tanto, una arquitectura de carga/almacenamiento (registro-registro).
- Para acceder a una palabra en memoria hay que indicar su dirección. MIPS direcciona bytes individuales.

2.6.2. Arquitectura del MIPS

Las primeras implementaciones de la arquitectura MIPS (familia R20x0 y R30x0) consisten en una unidad de proceso de enteros (la CPU) y un conjunto de procesadores que realizan tareas auxiliares u operan con otros tipos de datos como números en coma flotante. En la figura 2.3 se muestra el esquema del MIPS R2000.

La arquitectura MIPS posée 32 registros genéricos de 32 bits ($0-$31), para utilización de la CPU, siendo el registro $0 sólo de lectura y con valor cero. De los restante registros, sólo el $31 es implícitamente usado por una instrucción (de invocación de una subrutina, para guardar la dirección de retorno, que veremos más adelante).

Adicionalmente, el MIPS contiene dos registros para poder operar con operandos de 64 bits, como sucede en el caso de la multiplicación y división, llamados hi(gh) y lo(w).

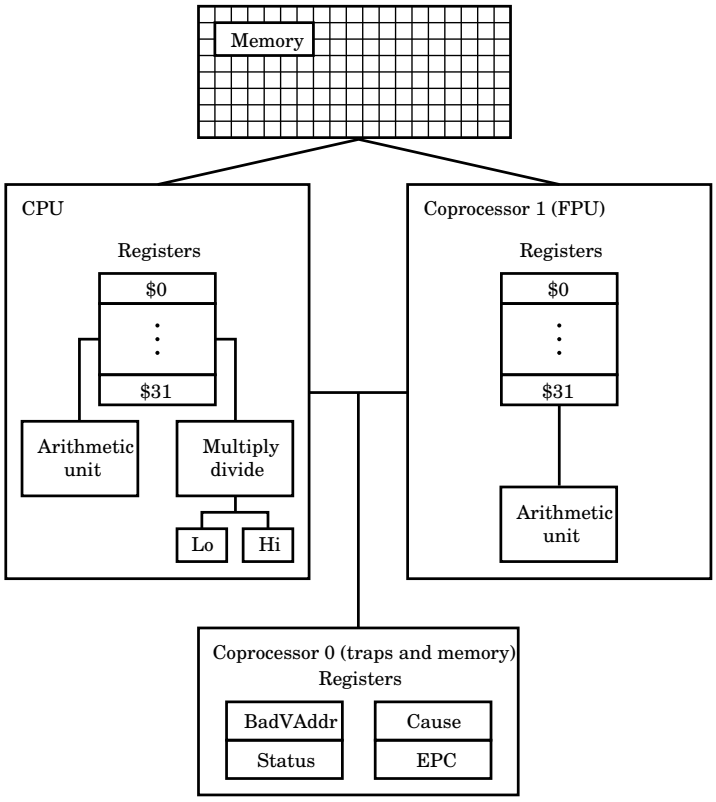

Figura 2.3: Esquema del MIPS R2000

Como se ha comentado antes, estos procesadores MIPS no disponen de unidad de coma flotante incluida en el microprocesador, implementando estas funciones en coprocesadores separados. La arquitectura MIPS tiene en cada coprocesador 32 registros de 32 bits para coma flotante ($f0-$f31), que pueden ser organizados en 16 registros de doble precisión, con 64 bits (las designaciones par de los registros).

En la tabla 2.1 se puede ver la lista de registros del MIPS y se describe sus usos.

Nombre	**Número**	**Uso**	**Preservado en llamada**
`$zero`	0	Valor constante 0	n.a.
`$v0-$v1`	2-3	Valores para resultados y evaluación de expresiones	no
`$a0-$a3`	4-7	Argumentos	sí
`$t0-$t7`	8-15	Temporales	no
`$s0-$s7`	16-23	Salvados	sí
`$t8-$t9`	24-25	Temporales	no
`$k0-$k1`	26-27	Reservado para núcleo de SO	n.a.
`$gp`	28	Puntero global	sí
`$sp`	29	Puntero de pila	sí
`$fp`	30	Puntero de bloque de activación	sí
`$ra`	31	Dirección de retorno	sí

Cuadro 2.1: Registros del MIPS y su uso convencional

2.6.3. Ensamblador

La codificación binaria de instrucciones es natural y eficiente para los computadores. Sin embargo, los humanos tienen gran dificultad para entender y manipular dicha codificación. El *lenguaje ensamblador* es la representación simbólica de la codificación binaria de un computador, el *lenguaje máquina*.

Una herramienta denominada *ensamblador* traduce lenguaje ensamblador a instrucciones binarias. Los lenguajes ensambladores ofrecen una representación más próxima al programador que los ceros y los unos del computador. La traducción del lenguaje ensamblador al lenguaje máquina se denomina *ensamblamiento*.

2.6.4. Formatos de las instrucciones MIPS

Veamos con un ejemplo el formato de las instrucciones MIPS. La instrucción representada simbólicamente como:

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

se representa en lenguaje MIPS como campos de números binarios de la siguiente forma:

000000	10001	10010	01000	00000	100000

correspondientes a la representación decimal:

0	17	18	8	0	32

Esta distribución de la instrucción se denomina *formato de instrucción*. Como comentabamos anteriormente, el número de bits de una instrucción MIPS es siempre de 32, el mismo tamaño que una palabra. A los campos MIPS se les da una serie de nombres para identificarlos fácilmente:

- *op*: operación básica de la instrucción, tradicionalmente llamada código de operación.
- *rs*: primer registro operando fuente.
- *rt*: segundo registro operando fuente.
- *rd*: registro operando destino, donde se almacena el resultado de la operación.
- *shamt*: tamaño del desplazamiento (*shift amount*).
- *funct*: función. Este campo selecciona la variante específica de la operación del campo *op*, y a veces se le denomina *código de función*.

Cuando una instrucción necesita campos más largos que los mostrados anteriormente aparece un problema. Por ejemplo, la instrucción de carga, debe especificar dos registros y una constante. Si la dirección usara uno de los campos de 5 bits del formato anterior, la constante dentro de la instrucción de carga estaría limitada a sólo 2^5 o 32. Esta constante se usa para seleccionar elementos de tablas grandes o de estructuras de datos, y frecuentemente necesita ser mucho mayor que 32. Este campo de 5 bits es demasiado pequeño para ser útil.

El compromiso elegido por los diseñadores del MIPS es guardar todas las instrucciones con la misma longitud, por eso se requieren diferentes clases de formatos de instrucción para diferentes clases de instrucciones. Los tres tipos de formatos en MIPS son:

- **Formato tipo R**: utilizado por las instrucciones aritméticas y lógicas.
- **Formato tipo I**: utilizado por las instrucciones de transferencia, las de salto condicional y las instrucciones con operandos inmediatos.
- **Formatos tipo J**: utilizado por las instrucciones de bifurcación.

En la figura 2.4 se muestran los campos para cada uno de los tres tipos de formato.

Aunque tener múltiples formatos complica la circuitería, se puede reducir la complejidad guardándolos de forma similar. Por ejemplo, los tres primeros campos de los formatos tipo-R y tipo-I son del mismo tamaño y tienen los mismos nombres. Los formatos se distinguen por el valor del primer campo: a cada formato se le asigna un conjunto de valores distintos en el primer campo y por lo tanto la ciruitería sabe si ha de tratar la última mitad de la instrucción como tres campos (tipo-R) o como un campo simple (tipo-I), o si la instrucción es tipo-J.

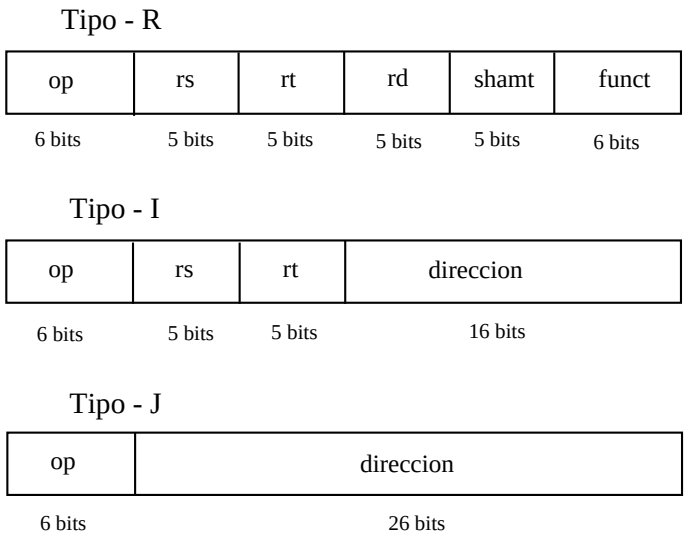

Figura 2.4: Codificación de instrucciones en el MIPS

2.6.5. Instrucciones aritmético-lógicas

El tipo de formato de las instrucciones aritmético-lógicas es tipo-R y el número de operandos en una operación de este tipo (aritmético-lógica) es siempre tres. Estos operandos son siempre registros, el modo de direccionamiento empleado, por tanto, es direccionamiento de registro.

En la figura 2.5 se muestran las estructuras del lenguaje máquina MIPS para dos ejemplos de instrucciones aritméticas, suma y resta.

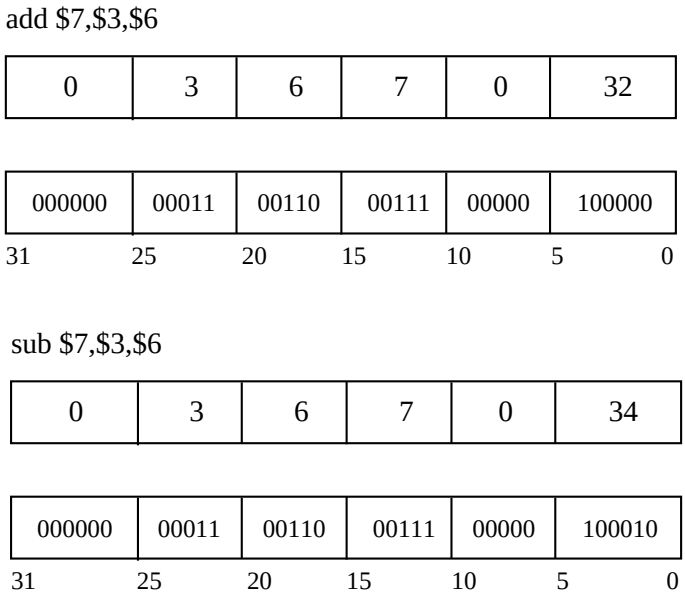

Figura 2.5: Estructuras del lenguaje MIPS para suma y resta

Muchas veces, los programas usan constantes en las operaciones para, por ejemplo, incrementar un índice y apuntar al siguiente elemento de una tabla. De hecho, en el compilador de C gcc, el 52 % de las operaciones aritméticas utilizan constantes. Usando las instrucciones de la figura 2.5, para usar una constante habría que cargarla de memoria al registro para posteriormente sumarla. Una alternativa que evita los accesos a memoria es ofrecer versiones de las instrucciones aritméticas en las cuales un operando es constante, con la nueva restricción de que esta constante se almacena dentro de la misma instrucción. Se usa en este caso el formato de instrucción tipo-I y el modo de direccionamiento es direccionamiento inmediato. En la figura 2.6 se muestra la estructura para el ejemplo de la operación suma inmediata.

Los operandos constantes aparecen con frecuencia, y situarlos dentro de las instrucciones aritméticas hace que se ejecuten mucho más rápido.

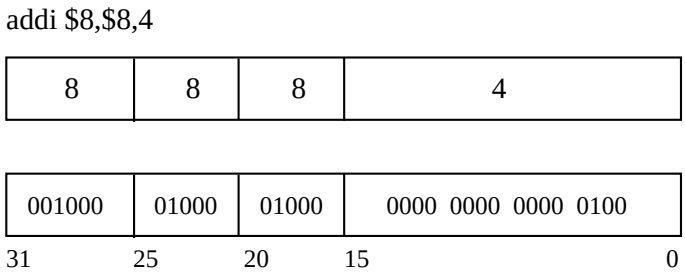

Figura 2.6: Estructura del lenguaje MIPS para la suma inmediata

2.6.6. Instrucciones de transferencia

Las instrucciones de transferencia son instrucciones tipo-I. En la figura 2.7 se muestran las estructuras del lenguaje MIPS para dos ejemplos de operaciones de transferencia muy comunes, la carga y el almacenamiento de una palabra. Los 16 bits de dirección significan que una instrucción, por ejemplo una instrucción de carga (*load*), puede cargar cualquier palabra dentro de la región 2^{15} o 32768 bytes de la dirección del registro base *rs*. El direccionamiento usado en este tipo de instrucciones es direccionamiento con desplazamiento (registro-base).

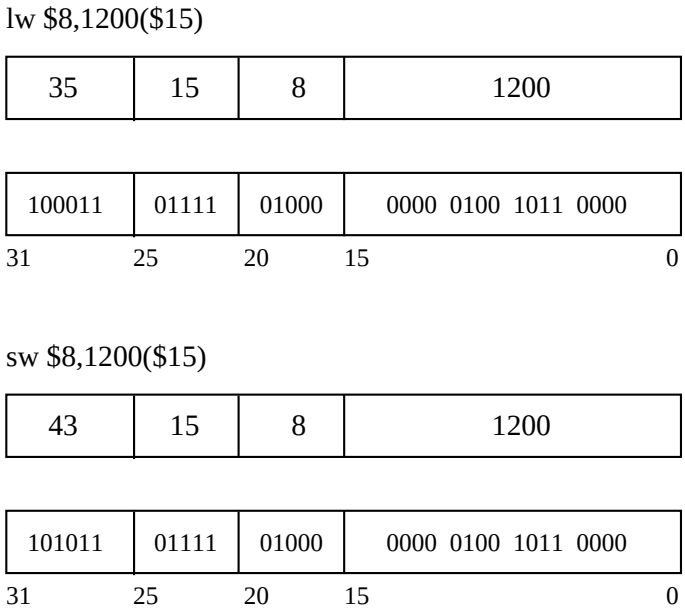

Figura 2.7: Estructura del lenguaje MIPS para las instrucciones de transferencia

Existe también otra instrucción de transferencia que implementa MIPS. Es la llamada instrucción *load upper inmediate* (`lui`) que sirve específicamente para almacenar los 16 bits de la parte alta de una constante en un registro. En la figura

2.8 se puede ver la operación lui. Aunque a estas alturas del capítulo no nos parezca muy útil, esta instrucción tiene, como veremos posteriormente, gran importancia.

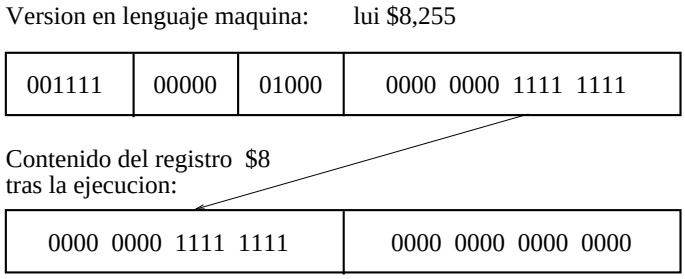

Figura 2.8: El efecto de la instrucción `lui`

2.6.7. Instrucciones de salto condicional

Lo que distingue a un computador de una simple calculadora es la habilidad de tomar decisiones. Basándose en los datos de entrada y los valores creados durante la computación, el computador ejecuta diferentes instrucciones. La toma de decisiones se representa comúnmente en los lenguajes de programación usando la sentencia *if* (si condicional), combinada a veces con sentencias *go to* (ir a) y etiquetas. El lenguaje ensamblador del MIPS incluye dos instrucciones de toma de decisiones, similares a una sentencia *if* con un *go to*. Estas instrucciones se muestran en la figura 2.9.

La instrucción `beq` (*branch if equal*) significa ir a la sentencia etiquetada con L1 si el valor del registro *rs* es igual al valor del registro *rt*. La instrucción `bne` (*branch if nt equal*) significa ir a la sentencia etiquetada con L1 si el valor de *rs* no es igual al valor en *rt*. Estas dos instrucciones se conocen tradicionalmente como *saltos condicionados*.

Las instrucciones de salto condicionado son de tipo-I. El modo de direccionamiento empleado por ambas es direccionamiento con desplazamiento relativo.

En ensamblador, el uso de etiquetas libera al programador del tedioso cálculo de las direcciones de salto.

La prueba de igualdad y desigualdad es probablemente la más habitual, pero a veces es útil establecer comparaciones del tipo "menor que". Para ello se dispone de la instrucción MIPS *set on less than* (activar si es menor que). También existe la versión de esta instrucción utilizando operandos inmediatos. La estructura de ambas instrucciones se muestra en la figura 2.10.

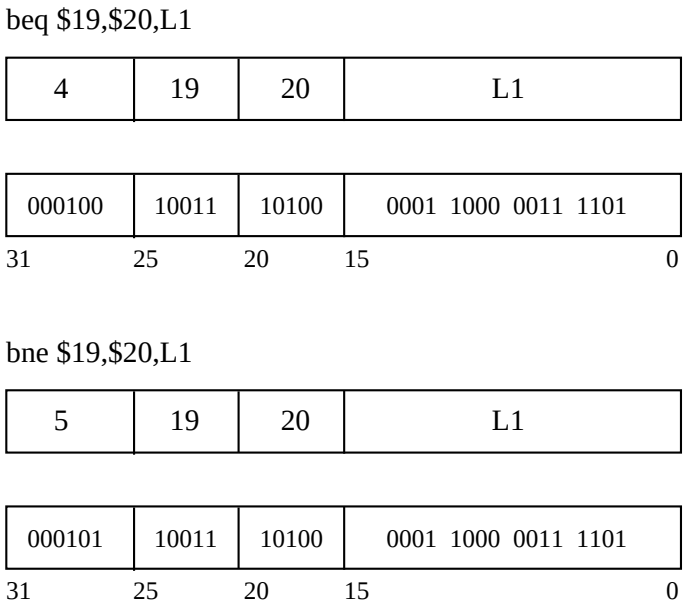

Figura 2.9: Estructura en lenguaje MIPS de las instrucciones de salto condicional

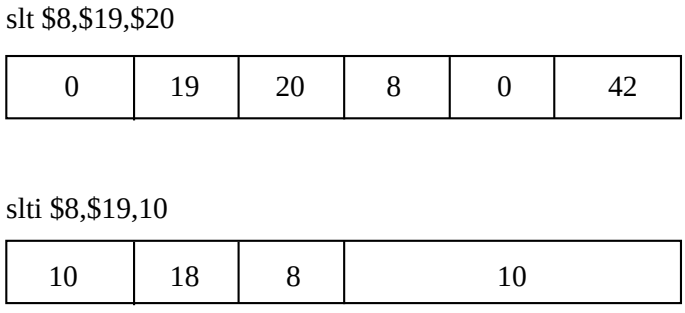

Figura 2.10: Estructura de las instrucciones *slt* y *slti* en el MIPS

2.6.8. Instrucciones de bifurcación

Una bifurcación se puede ver como un *salto incondicional*, es decir, la instrucción obliga a la máquina a seguir siempre el salto. Para distinguir entre saltos condicionales e incondicionales, el nombre MIPS para este tipo de instrucción es *jump*. En la figura 2.11 se muestra esta instrucción y su formato.

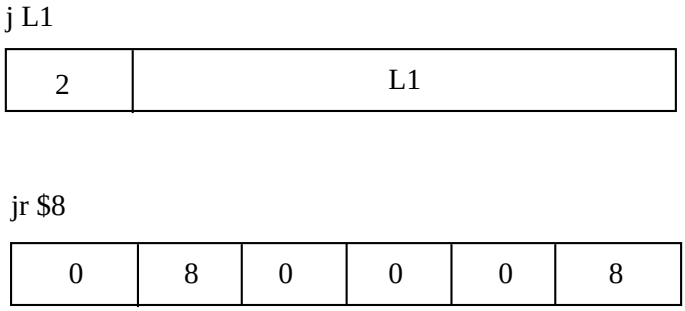

Figura 2.11: Estructura de las instrucciones *jump* y *jump register* en el MIPS

La instrucción de bifurcación *jump* es de tipo-J y su modo de direccionamiento es pseudodirecto.

Muchos lenguajes de programación tienen una sentencia alternativa *case* o *switch*, que permite al programador seleccionar una de las muchas alternativas dependiendo de un único valor. Una forma de realizar *switch* es a través de una secuencia de pruebas condicionales, conviertiendo la sentencia *switch* en una cadena de sentencias *if-then-else*. A veces las alternativas se pueden codificar eficientemente como una tabla de direcciones de secuencias de instrucciones alternativas, llamadas *tabla de direcciones de saltos*, y el programa necesita sólo acceder a la tabla y luego saltar a la secuencia apropiada. La tabla de saltos es entonces simplemente una tabla de palabras, que contiene direcciones que se corresponden con etiquetas en el código.

Para permitir este tipo de situaciones, los computadores como el MIPS, incluyen una instrucción denominada *jump register* (jr), que significa un salto incondicional a la dirección especificada en el registro. El programa carga previamente la entrada apropiada de la tabla de saltos en un registro, y luego salta a la dirección indicada usando un registro de salto.

En la figura 2.11 se muestra la estructura de esta instrucción. La instrucción *jump register* es de tipo-R y utiliza modo de direccionamiento indirecto con registro.

2.6.9. Limitaciones

Cargar una constante de 32 bits

En la sección 2.6.5, hablábamos del uso de las constantes en las operaciones aritméticas. En las operaciones aritméticas inmediatas se usa el formato tipo-I donde los últimos 16 bits (normalmente de dirección) almacenan ahora el valor de la constante. Aunque las constantes son habitualmente pequeñas y caben en este campo, a veces pueden ser más grandes. El repertorio de instrucciones MIPS incluye la instrucción *load upper inmediate* (`lui`), vista en la sección 2.6.6 específicamente para almacenar los 16 bits de la parte alta de una constante en un registro, permitiendo a la instrucción siguiente especificar los 16 bits más bajos de la constante. El modo de direccionamiento usado por esta instrucción es el direccionamiento inmediato. En la figura 2.8 se puede ver la operación lui.

Un ejemplo para ver la utilidad de esta instrucción es ver cómo se puede cargar una constante de 32 bits utilizando la instrucción `lui`. Supongamos que queremos cargar la siguiente constante:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Primero se cargarían los 16 bits de mayor peso, que valen 61 en decimal, usando lui:

```
lui $s0,61     # 61 = 0000 0000 0011 1101
```

El valor del registro $s0 después de esto será:

```
0000 0000 0011 1101 0000 0000 0000 0000
```

El siguiente paso es sumar los 16 bits de menor peso, cuyo valor decimal es 2304:

```
addi $s0, $s0, 2304     # 2304 = 0000 1001 0000 0000
```

El valor final del registro es el valor deseado:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Saltar más alla

Casi todos los saltos condicionales son a localizaciones cercanas, pero ocasionalmente se salta más lejos, más allá de lo que puede ser representado con los 16 bits de la instrucción de salto condicional. El ensamblador viene al rescate tal como ha hecho con las constantes o direcciones de gran tamaño: inserta un salto incondicional al objetivo del salto, e invierte la condición de forma que el salto condicional decide si esquiva el salto incondicional.

Veamos como funciona esto con un ejemplo. Supongamos que queremos reemplazar el salto condicional siguiente:

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

por un par de instrucciones que proporcionen una mayor distancia de salto. El salto

anterior puede ser reemplazado por las siguientes instrucciones:

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

2.6.10. Modos de direccionamiento del MIPS

Los modos de direccionamiento del MIPS son los siguientes:

- *Modo de direccionamiento registro*, donde el operando es un registro
- *Modo de direccionamiento base más desplazamiento*, donde el operando está en una localización de memoria cuya dirección es la suma de un registro y una constante presente en la propia instrucción.
- *Modo de direccionamiento inmediato*, donde el operando es una constante que aparece en la misma instrucción.
- *Modo de direccionamiento relativo al PC*, donde la dirección es la suma del contador de programa (PC) y la constante de la instrucción.
- *Modo de direccionamiento pseudodirecto*, donde la dirección de salto son los 26 bits de la instrucción concatenados con los bits de mayor peso del contador de programa. La circuitería debe ir con cuidado para evitar situar un programa más allá de los límites de 256MB puesto que, en caso contrario, los saltos incondicionales deben ser reemplazados por instrucciones *jump register* precedidas por otras instrucciones para cargar la dirección de 32 bits completa en el registro.

Una operación simple puede usar más de un modo de direccionamiento. La operación de sumar, por ejemplo, usa tanto el direccionamiento inmediado (si usamos la instrucción `addi`), como el direccionamiento registro (si usamos la instrucción `add`). La figura 2.12 muestra, para cada modo de direccionamiento, cómo se localiza el operando correspondiente.

2.6.11. Llamadas a subrutinas

Un procedimiento o subrutina es una herramienta que los programadores usan para estructurar programas, con el fin de hacerlos fácilmente comprensibles, y permite que el código sea reutilizado. Los procedimientos permiten al programador

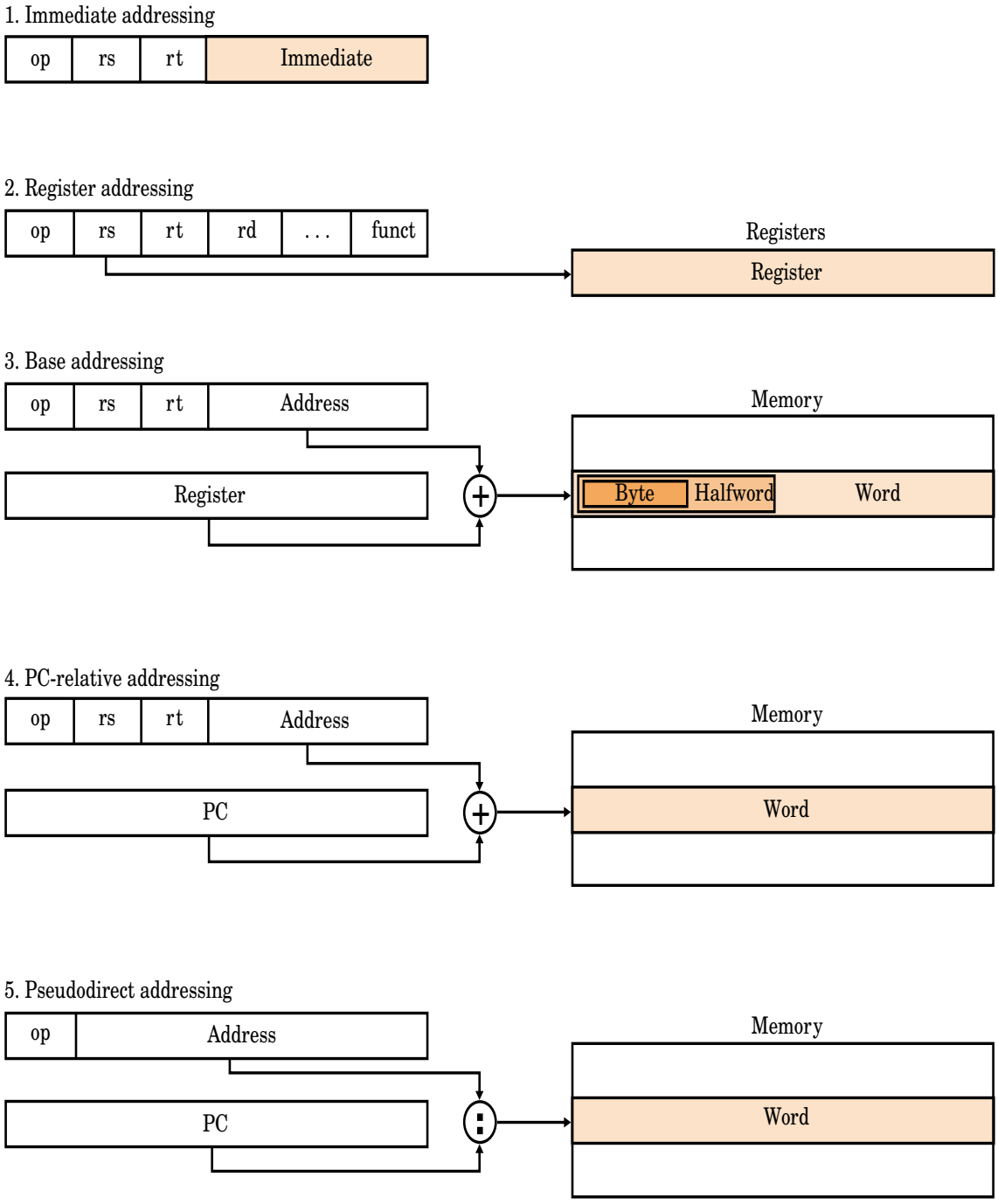

Figura 2.12: Modos de direccionamiento del MIPS.

concentrarse en una sola parte del trabajo cada vez. Los parámetros actúan a modo de barrera entre el procedimiento y el resto del programa y los datos, permitiendo pasar valores a la subrutina y que esta retorne resultados.

En la ejecución de una subrutina el programa debe seguir los siguientes pasos:

1. Situar los parámetros en un lugar donde la subrutina pueda acceder a ellos.
2. Transferir el control a la subrutina.
3. Adquirir los recursos de almacenamiento necesarios para el procedimiento.
4. Realizar la tarea deseada.
5. Situar el valor del resultado en un lugar donde el programa que lo ha llamado pueda acceder a él.
6. Retornar el control al punto de origen.

Los registros son el lugar más rápido para situar datos en el computador, por lo tanto se procura usarlos el mayor número de veces posible. De aquíque los programas MIPS, asignen los siguientes registros de los 32 disponibles en cada llamada de procedimiento:

- $a0-$a3: cuatro registros de argumentos en los cuales se pasan parámetros.
- $v0-$v1: dos registros de valores en los cuales se retornan valores.
- $ra: un registro de retorno de dirección para volver al punto de origen.

Además de esta asignación de registros, el lenguaje ensamblador del MIPS incluye una instrucción sólo para procedimientos: salta a una dirección y simultáneamente salva la dirección de la siguiente instrucción en el registro $ra. La instrucción *jump-and-link* (`jal`) se muestra en la figura 2.13.

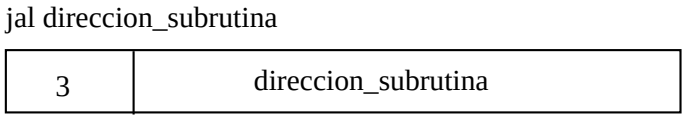

Figura 2.13: Estructura de la instrucción *jump-and-link* en MIPS.

La parte de enlace significa que una dirección o enlace está formado por aquellos puntos del lugar de la llamada que permiten al procedimiento volver a la dirección apropiada. Este *enlace* almacenado en el registro $ra se llama *dirección de retorno.*

La dirección de retorno es necesaria porque el mismo procedimiento se puede llamar desde diferentes puntos del programa.

En la idea de programa almacenado está implícita la necesidad de tener un registro para guardar la dirección de la instrucción actual que está siendo ejecutada. Por razones históricas, este registro casi siempre se llama *contador de programa* (PC). La instrucción `jal` guarda $(PC + 4)$ en el registro $ra para encadenar la siguiente instrucción con el retorno del procedimiento.

Para realizar el salto de retorno ya disponemos de la instrucción adecuada: jr $ra La instrucción *jump register*, que se ha comentado anteriormente, salta a la dirección almacenada en el registro $ra, que es justamente lo que se requiere.

Pila o Stack

Supongase que un compilador necesita más registros para un procedimiento que los cuatro registros de argumentos y los dos registros de retorno de valores. Como se supone que deben borrarse las huellas después de que la misión se complete, cualquier registro necesario por el invocador debe ser restaurado con los valores que contenían antes de que el procedimiento fuera invocado. Esta situación situación es un ejemplo en el cual es necesario volcar registros en memoria.

La estructura de datos ideal para volcar registros es una *pila* o *stack*. Una pila es un espacio de memoria con estructura tipo LIFO (*Last In First Out*), en la que el último que entra es el primero en salir. Esta estructura necesita un puntero a la dirección más recientemente utilizada de la pila, para guardar dónde debería de situar el siguiente procedimiento los registros que necesitará volcar, o para saber dónde se pueden encontrar los valores antiguos de los registros. El puntero de pila se ajusta a una palabra, por cada registro que se salva o restaura. Las pilas son tan habituales que disponen de sus propios nombres para transferir datos hacia la pila y desde ella: *push* (apilar) y *pop* (desapilar) respectivamente.

Los programas del MIPS reservan un registro sólo para la pila: *stack pointer* ($sp) o puntero de pila, usado para salvar los registros necesarios por el invocador. En la figura 2.14 se muestra gráficamente la estructura de una pila.

Por razones históricas, la pila crece de direcciones de memoria superiores a inferiores. Por tanto para poner valores en la pila tendremos que restar al puntero de pila y para quitar valores tendremos que sumar al puntero de pila. Veamos un par de ejemplos de como introducir datos en la pila y como transferirlos desde ella. Para realizar la operación de *push* salvando dos registros en la pila:

```
addi $sp,$sp,-8    # ajusto la pila para añadir dos elementos
sw $v0,0($sp)      # salvo el registro $2
```

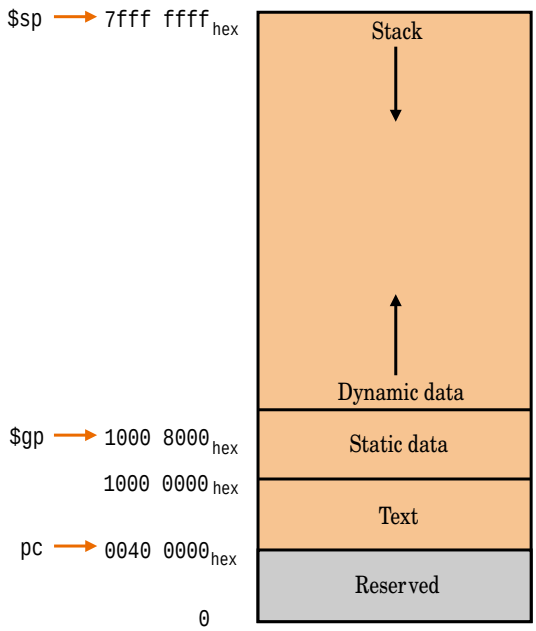

Figura 2.14: Distribución de la memoria para programas y datos en el MIPS

```
sw $v1,4($sp)      # salvo el registro $3
```

Para realizar la operación de *pop* transfiriendo los datos de la pila a dos registros:

```
lw $v0,0($sp)      # restaura el registro $2
lw $v1,4($sp)      # restaura el registro $3
addi $sp,$sp,8     # ajusta el puntero de la pila para eliminar
                   # dos elementos
```

Si un procedimiento modifica los registros utilizados por la rutina invocadora los valores de los registros deben ser guardados y restaurados. Para ello también se utiliza la pila.

Los dos convenios estándares para guardar y restaurar registros son:

- Guardar invocador (*caller save*): el procedimiento invocador es el responsable de guardar y restaurar los registros que deban conservarse.
- Guardar invocado (*callee save*): el invocado es el responsable de guardar y restaurar cualquier registro que pueda utilizar.

Veamos como funcionan estos dos convenios con el siguiente ejemplo:

```
Clear\_Array(int A[], int n)
```

```
{
  int j;
  for(j=0;j<n;j++)
    A[j]=0;
}
```

Si usamos el primer método (guardar invocador) el programa comienza guardando los registros en la pila antes de saltar al procedimiento y después de regresar de él se encarga de restaurarlos:

```
addi $sp,$sp,-12    # ajusto la pila para tres elementos
sw $19,0($sp)    # salvo 3 registros $19,$15 y $8
sw $15,4($sp)
sw $8,8($sp)

jal Clear_Array     # llamo al procedimiento

lw $19,0($sp)    # restauro los 3 registros
lw $15,4($sp)
lw $8,8($sp)
addi $sp,$sp,12    # restauro el espacio de pila
```

Por su parte el procedimiento quedaría de la siguiente forma:

```
Clear_Array:
   add $19,$zero,$zero    # j=0
for:
   slt $1,$19,$5     # evaluamos que j<n
   beq $1,$zero,exit     # saltamos a exit si j=n
   addi $8,$zero,4     # $8=4
   mul $15,$19,$8     # $15=4*j
   add $15,$15,$4     # $15=4*j+A
   sw $0,0($15)     # A[j]=0
   addi $19,$19,1     # j++
   j for
exit:

   jr $ra     # vuelvo al programa invocador.
```

Si utilizamos el segundo método (guardar invocado) es el procedimiento el que se encarga de salvar los registros que va a usar y después restaurarlos antes de regresar al programa que lo llamó. Comenzamos con la etiqueta del procedimiento y luego

salvamos el valor de tres registros en la pila:

```
Clear_Array:
    addi $sp,$sp,-12   # ajusto la pila para tres elementos
    sw $19,0($sp)    # salvo 3 registros $19,$15 y $8
    sw $15,4($sp)
    sw $8,8($sp)
```

A continuación el procedimiento queda igual que en el caso anterior:

```
    add $19,$zero,$zero     # j=0
for:
    slt $1,$19,$5     # evaluamos que j<n
    beq $1,$zero,exit      # saltamos a exit si j=n
    addi $8,$zero,4      # $8=4
    mul $15,$19,$8      # $15=4*j
    add $15,$15,$4      # $15=4*j+A
    sw $0,0($15)      # A[j]=0
    addi $19,$19,1      # j++
    j for
```

Por último tenemos que restaurar los registros usados antes de volver al punto en que el procedimiento fue invocado. exit:

```
    lw $19,0($sp)        # restauro los 3 registros
    lw $15,4($sp)
    lw $8,8($sp)
    addi $sp,$sp,12         # restauro el espacio de pila

    jr $ra               # vuelvo al programa invocador.
```

Para evitar salvar y restaurar un registro cuyo valor nunca se usa, cosa que podría suceder con un registro temporal, los programas MIPS ofrecen dos clases de registros:

- $t0-$t9: 10 registros temporales que no son preservados por el invocado (procedimiento llamado) en una llamada de procedimiento.

- $s0-$s7: 8 registros salvados que deben ser preservados en una llamada de procedimiento (si los usa, el invocado los salva y los restaura).

Esta simple convención reduce el volcado de registros.

Procedimientos anidados

Los procedimientos que no llaman a otros se denominan procedimientos *hoja*. La vida sería sencilla si todos los procedimientos fueran procedimientos hoja, pero no es así. En el caso de procedimientos que llaman a otros procedimientos la dirección de vuelta sería machacada. También puede haber conflictos en el paso de argumentos.

Para evitar problemas el invocador debe guardar en la pila cualquier registro de argumentos ($a0-$a3) o registros temporales ($t0-$t9) que le sea necesario después de la llamada.

El invocado, por su parte, debe guardar en la pila el registro con la dirección de retorno ($ra) y cualquier registro salvado ($s0-$s7) usado por el invocado. El puntero de pila $sp es ajustado para contar el número de registros situados en la pila. Antes del retorno los registros son restaurados de memoria y el puntero de pila reajustado.

Veamos un ejemplo: el procedimiento recursivo que calcula el factorial.

```
int fact (int n)
 {
   int tmp;
   if (n<2)
      tmp=1;
   else
      tmp=n*fact(n-1);
   return(tmp);
 }
```

El parámetro variable n se corresponde con el registro de argumentos $a0. El programa compilado comienza con la etiqueta del procedimiento y luego salva dos registros en la pila, la dirección de retorno y $a0:

```
fact:
    addi $sp,$sp,-8    # ajusto la pila para dos elementos
    sw $a0,0($sp)    # salvo el argumento n
    sw $ra,4($sp)    # salvo la dirección de retorno
```

La primera vez que se llama a `fact`, `sw` salva una dirección del programa que ha llamado a fact. Mediante las dos siguientes instrucciones se comprueba si n es menor que 2 y se va a `else` en caso contrario.

```
    slti $t0,$a0,2     # comprobar si n<2
    beq $t0,$zero,else    # si no se cumple ir a else
```

Si n es menor que 2, `fact` devuelve 1 colocando 1 en un registro de valor: se suma 1 con 0 y se coloca el resultado de la suma en $v0. Luego se eliminan los dos valores salvados de la pila y se salta a la dirección de retorno:

```
if:
    addi $v0,$zero,1    # tmp=1
    addi $sp,$sp,8     # eliminar dos elementos de la pila
    jr $ra               # retornar al punto después de jal
```

Antes de quitar los dos elementos de la pila se podía haber cargado $a0 y $ra. Como estos dos registros no varían cuando n es menor que dos se pueden ignorar estas dos instrucciones.

Si n no es menor que 2, se decrementa el argumento n y luego se llama de nuevo a `fact` con este nuevo valor:

```
else:
    addi $a0,$a0,-1     # el argumento se carga con (n-1)
    jal fact              # llamar a fact con (n-1)
```

`fact` retornará a la instrucción siguiente al `jal`. ahora la dirección de retorno antigua y el argumento antiguo son restaurados, usando el puntero de la pila:

```
    lw $a0,0($sp)    # retorno de jal: restaura el argumento n
    lw $ra,4($sp)    # restaura la dirección de retorno
    addi $sp,$sp,8    # ajusta el puntero de la pila para eliminar
                   # dos elementos
```

A continuación el registro de valor $v0 se carga con el producto del argumento antiguo $a0 y el valor actual del registro de valor.

```
    mul $v0,$v0,$a0    # tmp=n*fact(n-1)
```

Finalmente, fact salta de nuevo hacia la dirección de retorno:

```
    jr $ra              # retorna al invocador
```

2.6.12. Almacenando caracteres

Los computadores se inventaron para devorar números, pero tan pronto como llegó a ser viable su comercialización se usaron para procesar textos. Hoy en día

muchos computadores usan bytes de 8 bits para representar caracteres, y la representación que prácticamente todos utilizan es el American Standard Code for Information Interchange (ASCII). La tabla 2.2 resume la representación ASCII.

bits	**bits izquierda**							
dcha	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NULL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	EXC	+	;	K	[	k	{
1100	FF	FS	,	¡	L		l	—
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	¿	N		n	
1111	SI	US	/	?	O	_	o	DEL

Cuadro 2.2: Representación ASCII de los caracteres

Se puede usar una serie de instrucciones para extraer un byte de una palabra, por lo que las instrucciones `lw` y `sw` sirven para transferir bytes de la misma manera que palabras. De todos modos debido a la frecuencia con que aparece texto en algunos programas, el MIPS proporciona instrucciones especiales para mover bytes. La instrucción *load byte* (`lb`) carga un byte de memoria y lo sitúa en los 8 bits de más a la derecha de un registro. La instrucción *store byte* (`sb`) lee un byte de los 8 bits de más a la derecha de un registro y lo escribe en memoria. De este modo se copia un byte con la secuencia:

```
lb $t0, 0($sp)     # leer byte de la fuente
sb $t0, 0($sp)     # escribir byte en el destino
```

Los caracteres normalmente se combinan en cadenas, que se componen de un número variable de caracteres. Hay tres formas de representar una cadena: (1) la primera posición de la cadena se reserva para indicar el tamaño de la misma, (2) una variable complementaria dispone del tamaño de la cadena, o (3) la última posición de la cadena se indica con un carácter usado para marcar el final.

2.7. Casos reales: PowerPC y Pentium

2.7.1. PowerPC

El PowerPC es descendiente directo del primer sistema RISC, el IBM 801, y es uno de los sistemas basados en RISC más potentes y mejor diseñados del mercado. El PowerPC se usa en millones de máquinas Apple Macintosh y en sistemas con microprocesadores embebidos.

El PowerPC comparte muchas similitudes con el MIPS; ambos tienen 32 registros de enteros, las instrucciones tienen todas 32 bits de longitud, y la transferencia de datos es posible sólo con cargas y almacenamientos. La principal diferencia son dos modos más de direccionamiento y unas pocas operaciones.

Modos de direccionamiento en el PowerPC

En constraste con lo que en la próxima sección se verá para el Pentium, el PowerPC, como la mayoría de las máquinas RISC, emplea un conjunto de modos de direccionamiento sencillo y relativamente evidente. En la tabla 2.3 estos modos vienen clasificados con relación al tipo de instrucción.

El PowerPC proporciona un modo de direccionamiento, llamado *modo de direccionamiento indirecto indexado*, que permite a dos registros ser sumados conjuntamente. En la figura 2.15 se muestra los dos modos de direccionamiento para las operaciones de carga y memorización en el PowerPC.

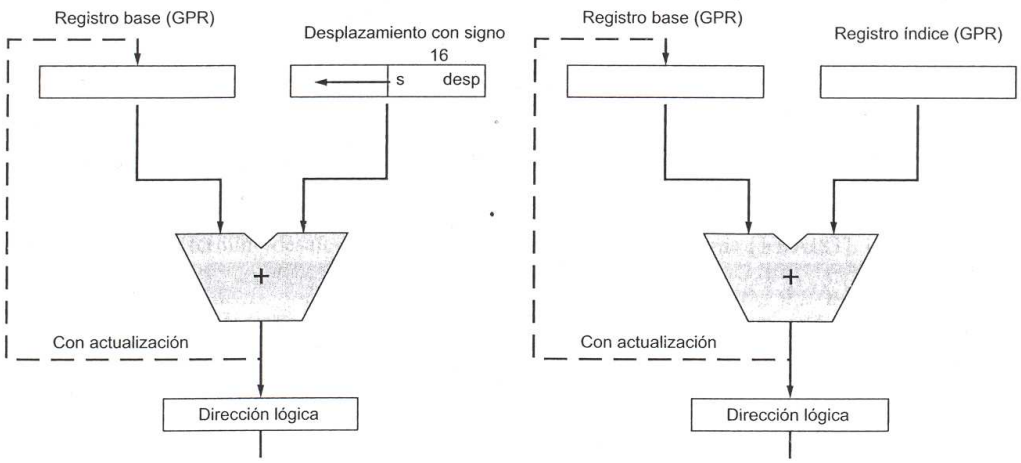

Figura 2.15: Modos de direccionamiento de operandos en memoria del PowerPC

Modo	**Algoritmo**
	Direccionamiento para Carga/Memorización
Indirecto	EA=(BR)+D
Indirecto Indexado	EA=(BR)+(IR)
	Direccionamiento de bifurcaciones
Absoluto	EA=I
Relativo	EA=(PC)+I
Indirecto	EA=(L/CR)
	Cálculos en punto fijo
Registro	EA=GPR
Inmediato	Operando=I
	Cálculos en punto flotante
Registro	EA=FPR

GPR = registro de uso general	EA = dirección efectiva
FPR = registro de punto flotante	(X) = contenido de X
D = desplazamiento	BR = registro base
I = valor inmediato	IR = registro índice
PC = contador de programa	L/CR = registro de enlace o de conteo

Cuadro 2.3: Modos de direccionamiento del PowerPC

El otro modo de indireccionamiento que posee el PowerPC a mayores que el MIPS es el llamado *modo de direccionamiento actualizado*. La idea de este modo es tener una nueva versión de las instrucciones de transferencia de datos, que incremente automáticamente el registro base, para apuntar a la siguiente palabra cada vez que se transfiere un dato. En la figura 2.16 se muestran gráficamente los modos de direccionamiento actualizado e indexado.

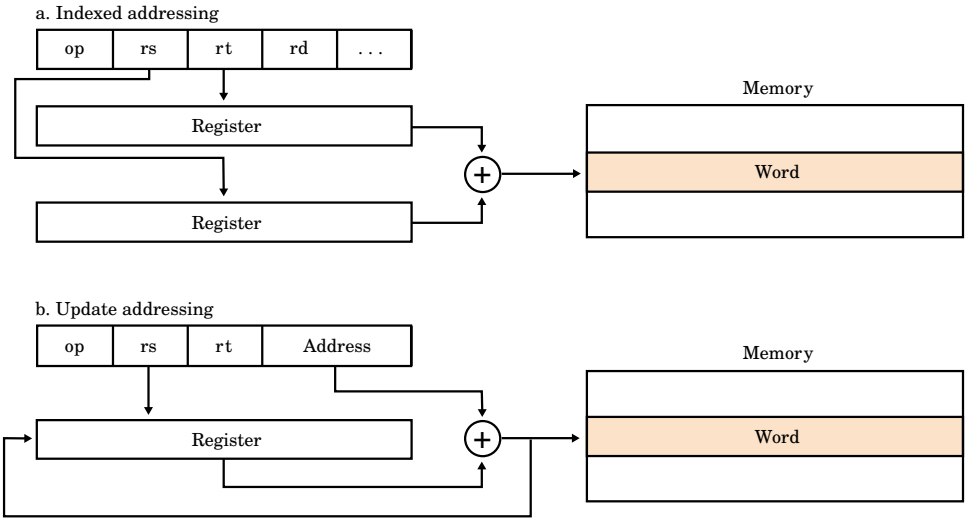

Figura 2.16: Modos de direccionamiento actualizado e indexado.

Instrucciones del PowerPC

Las instrucciones del PowerPC siguen el mismo estilo de arquitectura que el MIPS (figura 2.17), confiando ampliamente en una rápida ejecución de instrucciones simples para el rendimiento. Aquítenemos unas pocas excepciones.

La primera es la carga múltiple y almacenamiento múltiple. Estas pueden transferir hasta 32 palabras de datos en una sola instrucción, y se usan para hacer copias rápidas de posiciones de memoria, mediante el uso de cargas y almacenamientos múltiples conjuntamente. También ahorran el tamaño del código cuando salvan o restauran registros. En la figura 2.18 se muestran los tipos de operaciones del PowerPC.

Un segundo ejemplo son los bucles. El PowerPC tiene un registro contador especial, al margen de los otros 32 registros, para intentar mejorar el rendimiento de los bucle *for*.

(a) Instrucciones de bifurcacion

Bifurcacion	Inmediato largo			A	L
Bf cond.	Opciones	Bit CR	Desplazamiento de bifurcacion	A	L
Bf cond.	Opciones	Bit CR	Indirecta con el registro de enlace o el de conteo		L

(b) Instrucciones logicas con registros de condicion

CR	Bit destino	Bit fuente	Bit fuente	And, Or, Xor, etc	/

(c) Instrucciones de carga/almacenamiento

carga/mem indir.	Reg destino	Reg base	Desplazamiento		
carga/mem indir.	Reg destino	Reg base	Reg indice	Tam., signo, actualizar	/
carga/mem indir.	Reg destino	Reg base	Desplazamiento		XO

(d) Instrucciones artimeticas con enteros, logicas y de desplazamiento/rotacion

aritmeticas	Reg destino	Reg fuente	Reg fuente	O	suma, resta, etc			R
suma, etc	Reg destino	Reg fuente	valor inmediato con signo					
logicas	Reg fuente	Reg destino	Reg fuente	suma, resta, etc				R
and, or,etc	Reg fuente	Reg destino	valor inmediato con signo					
rotacion	Reg fuente	Reg destino	num despl.	inicio mascara	fin mascara			R
rot. o despl.	Reg fuente	Reg destino	Reg fuente	tipo de despl. o mascara				R
rotacion	Reg fuente	Reg destino	num despl.	Mascara	XO	S		R
rotacion	Reg fuente	Reg destino	Reg fuente	Mascara	XO			R
desplaz.	Reg fuente	Reg destino	num despl.	tipo de despl. o mascara		S		R

(e) Instrucciones en punto flotante

Flt sgl/dbl	Reg destino	Reg fuente	Reg fuente	Reg fuente	suma, resta, etc	R

Figura 2.17: Formato de instrucción del PowerPC

Instrucción	Descripción
	Dedicadas a bifurcaciones
b	Bifurcación o salto incondicional.
bl	Saltar a la dirección destino y colocar en el registro de enlace la dirección efectiva de la instrucción siguiente a la bifurcación.
bc	Salto condicional en función del registro de cuenta y/o bit del registro de condición.
sc	Llamada al sistema para invocar un servicio del sistema operativo.
trap	Comparar dos operandos y, si se cumple la condición especificada, invocar al gestor de interrupciones del sistema.
	Carga/memorización
lwzu	Cargar una palabra y poner a cero los restantes bits de la izquierda; actualizar el registro fuente.
ld	Cargar una palabra doble.
lmw	Cargar una palabra múltiple; cargar palabras consecutivas en registros contiguos, desde el especificado hasta el registro 31 de uso general.
lswx	Cargar una cadena de bytes en registros, comenzando por el registro indicado; cuatro bytes por registro; y con conexión cíclica del registro 31 al 0.
	Aritmética de enteros
add	Sumar los contenidos de dos registros y ubicar el resultado en un tercero.
subf	Restar los contenidos de dos registros y ubicar el resultado en un tercero.
mullw	Multiplicar los 32 bits menos significativos de los contenidos de dos registros y guardar el producto de 64 bits en un tercer registro.
divd	Dividir los contenidos de 64 bits de dos registros, y guardar el cociente en un tercer registro.
	Lógicas y desplazamientos
cmp	Comparar dos operandos y fijar cuatro bits de condición en el campo del registro de condición que se especifica.
crand	AND del registro de condición: se calcula el producto lógico de dos bits del registro de condición y el resultado se guarda en una de las dos posiciones de bit.
and	Producto lógico de los contenidos de dos registros y guardar el resultado en un tercero.
cntlzd	Cuenta el número de bits a 0 consecutivos del registro fuente, empezando por el bit cero, y guarda el resultado de la cuenta en el registro destino.
rldic	Rotar a la izquierda un registro de palabra doble, realizar la AND con una máscara, y almacenar el registro destino.
sld	Desplazar a la izquierda el registro fuente y almacenar en el registro de destino.
	Coma flotante
lfs	Cargar de memoria un número en coma flotante de 32 bits, convertir al formato de 64 bits, y memorizar en un registro de coma flotante.
fadd	Sumar los contenidos de dos registros y ubicar el resultado en un tercero.
fmadd	Multiplicar los contenidos de dos registros, sumar el contenido de un tercero, y colocar el resultado en un cuarto registro.
fcmpu	Comparar dos operandos en coma flotante y fijar los bits de condición.
	Gestión de cache
dcbf	Limpia un bloque de la cache de datos; busca en las direcciones concretas de la cache y realiza la operación.
icbi	Invalida un bloque de la cache de instrucciones.

Figura 2.18: Tipos de operaciones del PowerPC (con ejemplos de operaciones típicas)

2.7.2. Pentium

El Pentium representa el resultado de décadas de esfuerzo de diseño en computadores de repertorio complejo de instrucciones (CISC). Incorpora los sofisticados principios de diseño que antes se encontraban sólo en ordenadores grandes y supercomputadores, y es un excelente ejemplo de diseño CISC.

El Pentium ofrece un amplio abanico de tipos de operaciones, incluyendo diversas instrucciones especializadas. Con ello se ha intentado dotar de medios a los programadores de compiladores para producir traducciones óptimas a lenguaje máquina de los programas en lenguaje de alto nivel. La figura 2.20 resume los distintos tipos y da ejemplos de cada uno. La mayoría coinciden con instrucciones convencionales, que se pueden encontrar en la mayoría de repertorios de instrucciones máquina, pero algunos de estos tipos de instrucciones están adaptados a las arquitecturas 80x86/Pentium. Como ejemplos de estas últimas podemos citar las instrucciones para ejecutar llamadas/retornos a/de procedimientos: CALL, ENTER, LEAVE y RETURN. Otro conjunto de instrucciones especiales está dedicado a la segmentación de memoria. Son instrucciones privilegiadas, que pueden ejecutarse sólo desde el sistema operativo. Permiten cargar y leer tablas de segmentos locales y globales, y comprobar y alterar el nivel de privilegio de un segmento.

Modos de direccionamiento en el Pentium

El Pentium está equipado con diversos modos de direccionamiento, ideados para permitir la ejecución eficiente de lenguajes de alto nivel. La tabla 2.4 lista los 12 modos de direccionamiento del Pentium.

Instrucciones en el Pentium

El Pentium está equipado con varios formatos de instrucciones. De los elementos descritos a lo largo de este capítulo sólo el campo *código de operación* está presente siempre. La figura 2.19 ilustra el formato de instrucción general. Las instrucciones se componen de entre cero y cuatro prefijos de instrucción opcionales, un código de operación de uno o dos bytes, una especificación de dirección opcional, un desplazamiento opcional y un campo inmediato opcional.

Como puede suponerse, la codificación del conjunto de instrucciones del Pentium es muy compleja. Esto se debe, en parte a la necesidad de ser compatible con el 8086, y en parte al deseo de algunos diseñadores de suministrar todas las ayudas posibles al diseñador del compilador para que produzca código eficiente. Es un tema de controversia si un repertorio de instrucciones tan complejo como éste es preferible

Modo	Algoritmo
Inmediato	Operando = A
Registro	LA = R
Con desplazamiento	LA = (SR) + A
Base	LA = (SR) + (B)
Base con desplazamiento	LA = (SR) + (B) + A
Indice escalado con desplazamiento	LA = (SR) + (I)xS +A
Base con índice y desplazamiento	LA = (SR) + (B) + (I) + A
Base con índice escalado y desplazamiento	LA = (SR) + (I)xS + (B) + A
Relativo	LA = (PC) + A

LA = dirección lineal
PC = contador de programa
S = factor de escala
R = registro
A = contenido de un campo de dirección de la instrucción
SR = registro de segmento
(X) = contenido de X
B = registro base
I = registro índice

Cuadro 2.4: Modos de direccionamiento del Pentium

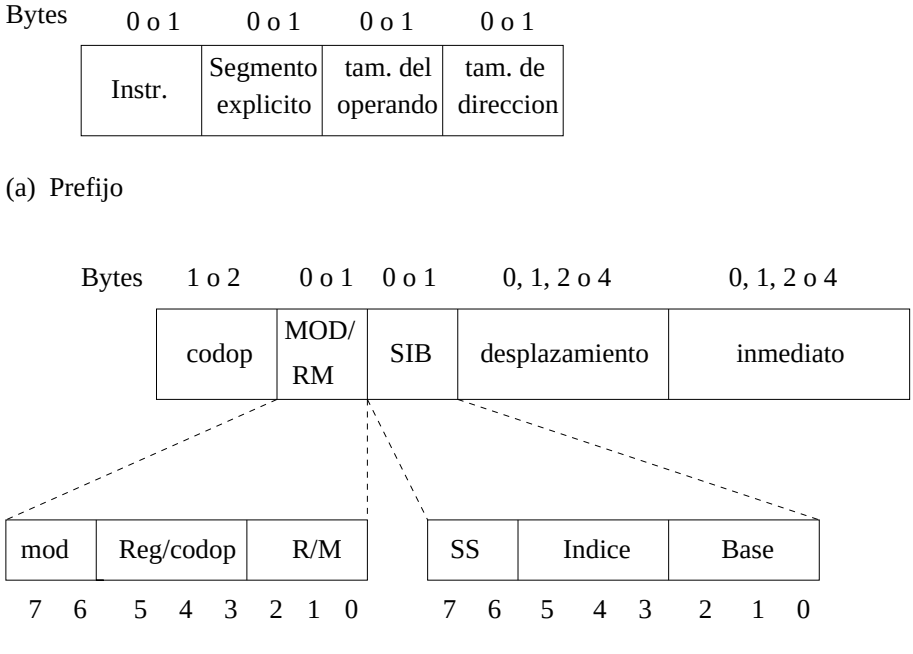

Figura 2.19: Formato de instrucción del Pentium

al repertorio de un RISC. En la figura 2.20 se muestran los tipos de operaciones del Pentium.

2.8. Jerarquía de traducciones

En la figura 2.22 se muestra la jerarquía de traducción, es decir, la transformación de un programa, en este ejemplo un programa en C, que se encuentra en un fichero en disco, en un programa ejecutable en un computador. Algunos sistemas combinan los cuatro pasos que se muestran en esta figura, para reducir el tiempo de traducción, pero estas son las cuatro fases lógicas que todos los programas necesitan.

Un programa en lenguaje de alto nivel primero se compila a un programa de lenguaje ensamblador y luego se ensambla en módulos objeto en lenguaje máquina. Los compiladores transforman el programa en lenguaje de alto nivel a un programa en *lenguaje ensamblador*, es decir, una forma simbólica de lo que la máquina entiende. Dado que los programas en lenguaje de alto nivel utilizan menos líneas de código que el lenguaje ensamblador, la productividad del programador es mucho mayor.

Hemos visto a lo largo de este capítulo que, a diferencia de los programas escritos en lenguajes de alto nivel, los operandos de las instrucciones aritméticas no pueden ser variables, se deben usar para ellos los registros. Para conseguir un alto rendimiento los compiladores deben usar los registros eficientemente. Por ejemplo, muchos programas tienen más variables que registros tiene la máquina. Consecuentemente, el compilador intenta mantener las variables usadas más frecuentemente en registros y colocar el resto en memoria, usando cargas y almacenamientos para mover variables entre registros y memoria. El proceso de poner las variables usadas menos frecuentemente (o aquellas necesarias más adelante) en memoria, se llama *spilling* (derramar o verter).

El ensamblador convierte el programa en lenguaje ensamblador en un *fichero objeto*, que es una combinación de instrucciones en *lenguaje máquina*, datos e información necesaria para situar las instrucciones correctamente en memoria.

El montador combina múltiples módulos con librerías de rutinas para resolver todas las dependencias. Luego el cargador sitúa el código máquina en las correspondientes localizaciones de memoria para la ejecución por parte del procesador.

Instrucción	Descripción
	Transferencias de datos
MOV	Transferir el operando entre registros o entre registro y memoria.
PUSH	Apilar el operando.
PUSHA	Apilar todos los registros.
MOVSX	«Move byte, word, dword, sign extended». Transfiere un byte a una palabra, o una palabra a una palabra doble, extendiendo hacia la izquierda el signo según la representación en complemento a dos.
LEA	«Load effective addres». Carga el desplazamiento del operando fuente, en lugar de su valor, en el operando destino.
XLAT	«Table lookup translation». Sustituye un byte de AL con otro de una tabla de traducción codificada por el usuario. Cuando se ejecuta XLAT, AL debe tener un índice sin signo de la tabla. XLAT cambia el índice que contiene AL por el correspondiente elemento de la tabla.
IN, OUT	Entrada o salida de operando desde el (o al) espacio de E/S.
	Aritméticas
ADD	Sumar los operandos.
SUB	Restar los operandos.
MUL	Multiplicación de enteros sin signo, con operandos de un byte, una palabra o una palabra doble, y como resultado una palabra, una palabra doble o una cuádruple.
IDIV	División con signo.
	Lógicas
AND	Producto lógico de los operandos.
BTS	«Bit test and set». Opera con un operando de campo de bits. La instrucción copia el valor actual de un bit en un indicador CF y pone a 1 el bit original.
BSF	«Bit scan forward». Busca en una palabra o en una palabra doble el primer bit a 1 y almacena su número de posición en un registro.
SHL/SHR	Desplazamiento lógico a izquierda o a derecha.
SAL/SAR	Desplazamiento aritmético a izquierda o a derecha.
ROL/ROR	Rotar a izquierda o a derecha.
SETcc	Pone un byte a cero o a uno, dependiendo de alguna de las 16 condiciones definidas por los indicadores de estado.
	Control de flujo
JMP	Salto incondicional.
CALL	Transfiere el control a otra posición. Antes de transferirlo, se introduce en la pila la dirección de la instrucción siguiente a la CALL.
JE/JZ	Salto si igual/cero.
LOOPE/LOOPZ	Bucle si igual/cero. Se trata de una bifurcación condicional que utiliza un valor almacenado en el registro ECX. La instrucción primero decrementa ECX antes de comprobar ECX para la condición del salto.
INT/INTO	Interrupción/Interrupción-si-desbordamiento. Transfiere el control a una rutina de servicio de interrupciones.
	Operaciones con cadenas
MOVS	«Move byte, word, dword string». Esta instrucción opera con un elemento de una cadena, indexada por los registros ESI y EDI. Después de cada operación con un elemento de la cadena, dichos registros se incrementan o decrementan automáticamente, para apuntar al siguiente elemento de la cadena.
LODS	Carga un byte, una palabra o una palabra doble de una cadena.

Figura 2.20: Tipos de operaciones del Pentium (con ejemplos de operaciones típicas)

Instrucción	Descripción
	Instrucciones de soporte a lenguajes de alto nivel
ENTER	Crea un marco de pila que puede utilizarse para implementar las reglas de un lenguaje de alto nivel estructurado por bloques.
LEAVE	Realiza la función inversa de la instrucción ENTER.
BOUND	Comprueba los extremos de una matriz. Verifica si el valor del operando 1 está entre ciertos límites inferior y superior. Los límites se encuentran en dos posiciones adyacentes, indicadas por el operando 2. Si el valor está fuera de dichos límites, se produce una interrupción. Esta instrucción se emplea para comprobar los índices de una matriz.
	Control de indicadores
STC	Activa el indicador de acarreo.
LAHF	Carga el registro A con los valores de los indicadores. Copia los bits SF, ZF, AF, PF, y CF en el registro A.
	Registro de segmentos
LDS	Carga un puntero en el registro de segmentos D.
	Control del sistema
HLT	Parar.
LOCK	Toma posesión de la memoria compartida para que el Pentium haga uso exclusivo de ella durante la instrucción que sigue inmediatamente a la LOCK.
ESC	Escape de ampliación del procesador. Es un código de escape para indicar que las instrucciones que siguen van a ser ejecutadas por un coprocesador aritmético que permite cálculos de alta precisión con enteros y en coma flotante.
WAIT	Espera hasta un BUSY negado. Suspende la ejecución del programa del Pentium hasta que el procesador detecte que el terminal BUSY está inactivo, indicando que el coprocesador aritmético ha finalizado su operación.
	Protección
SGDT	Memoriza una tabla global de descriptores.
LSL	Carga el limite de segmento. Carga un registro especificado por el usuario con un límite de segmento.
VERR/VERW	Verifica el segmento para lectura/escritura.
	Gestión de cache
INVD	Limpia la memoria cache interna.
WBINVD	Limpia la memoria cache interna después de escribir en memoria las líneas alteradas.
INVLPG	Invalida una entrada al buffer TLB.

Figura 2.21: Tipos de operaciones del Pentium (continuación)

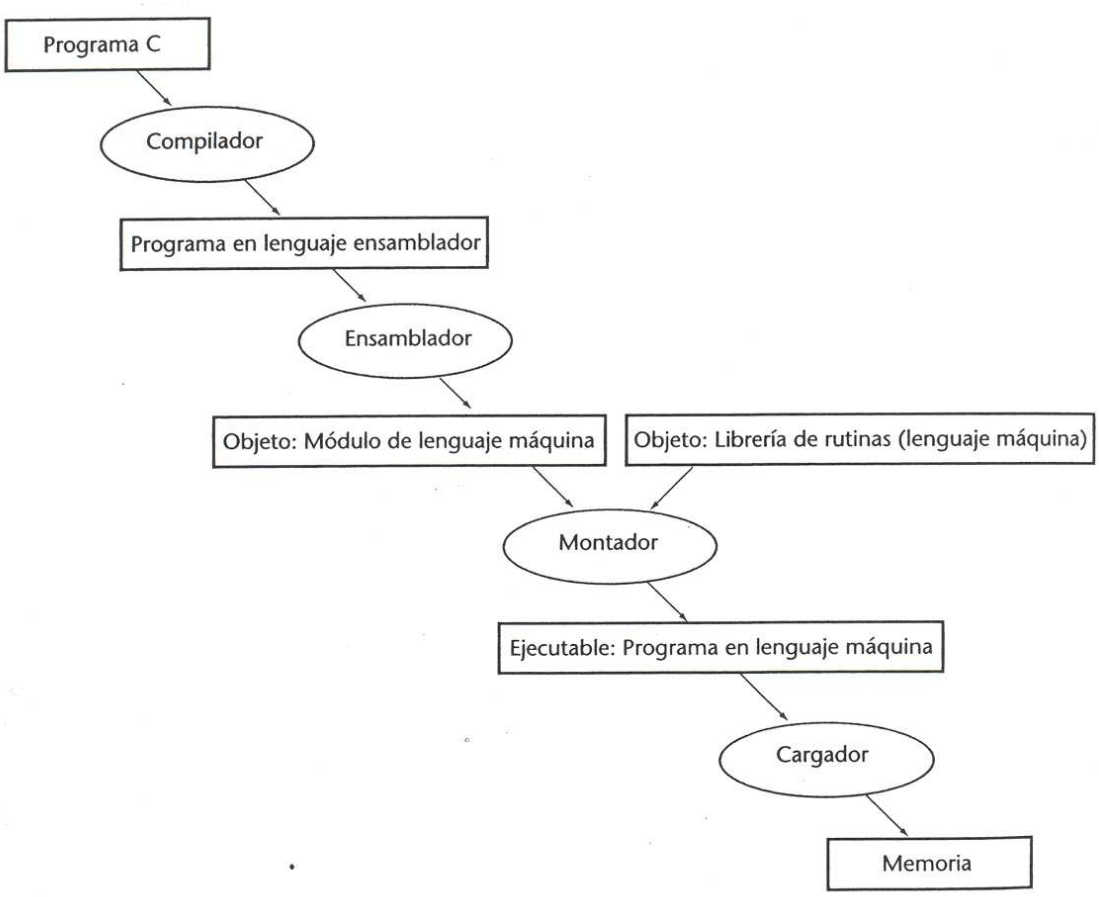

Figura 2.22: Una jerarquía de traducción