2. Arquitectura MIPS64

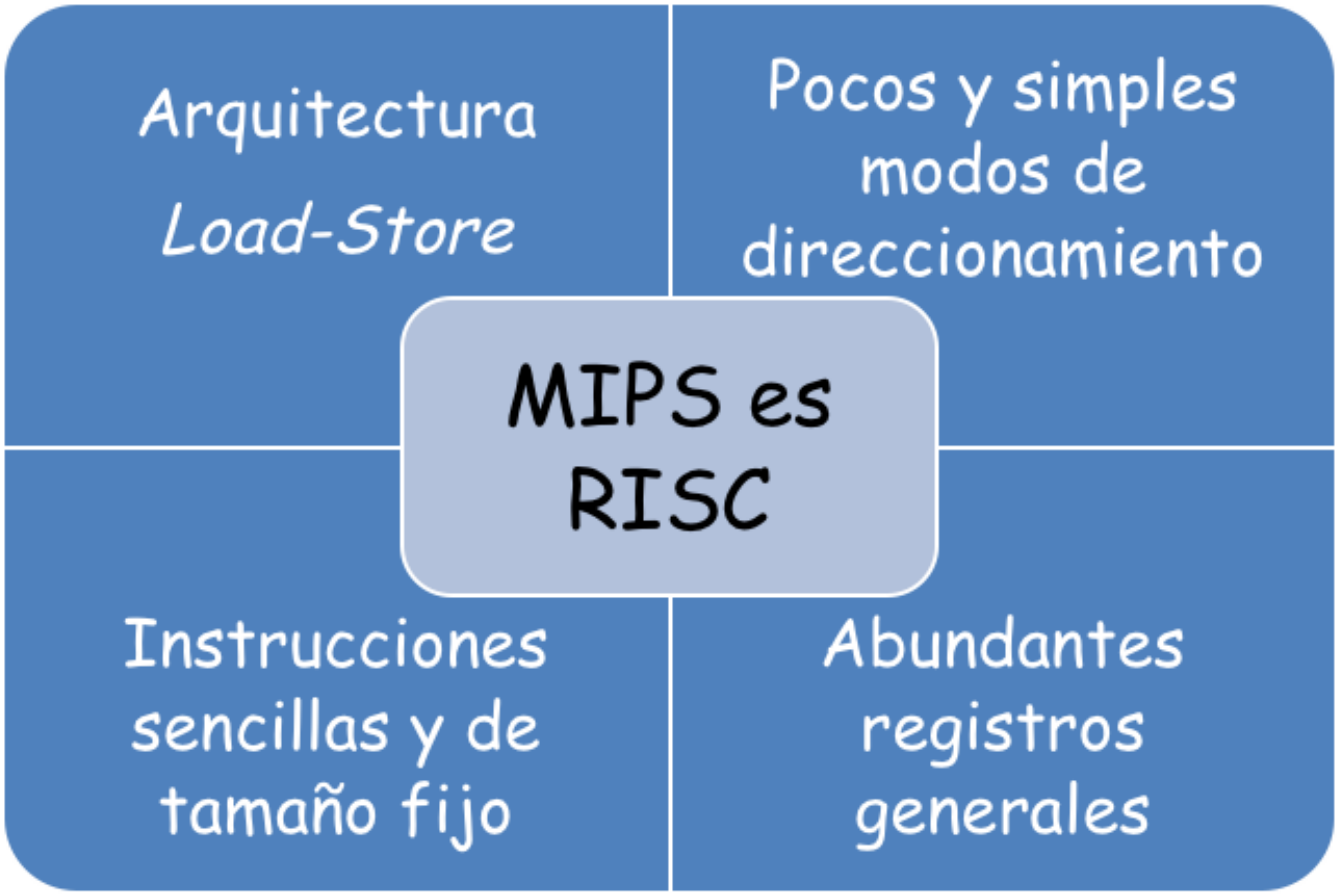
1. Características de la arquitectura MIPS64
2. Juego de instrucciones
3. Ensamblador de MIPS64

La descripción de esta arquitectura la vamos a organizar en tres apartados.

Comenzaremos presentando sus características generales, para luego ir viendo en detalle los tipos de datos que maneja, así como los modos de direccionamiento para acceder a ellos y los registros internos que le ofrece al programador.

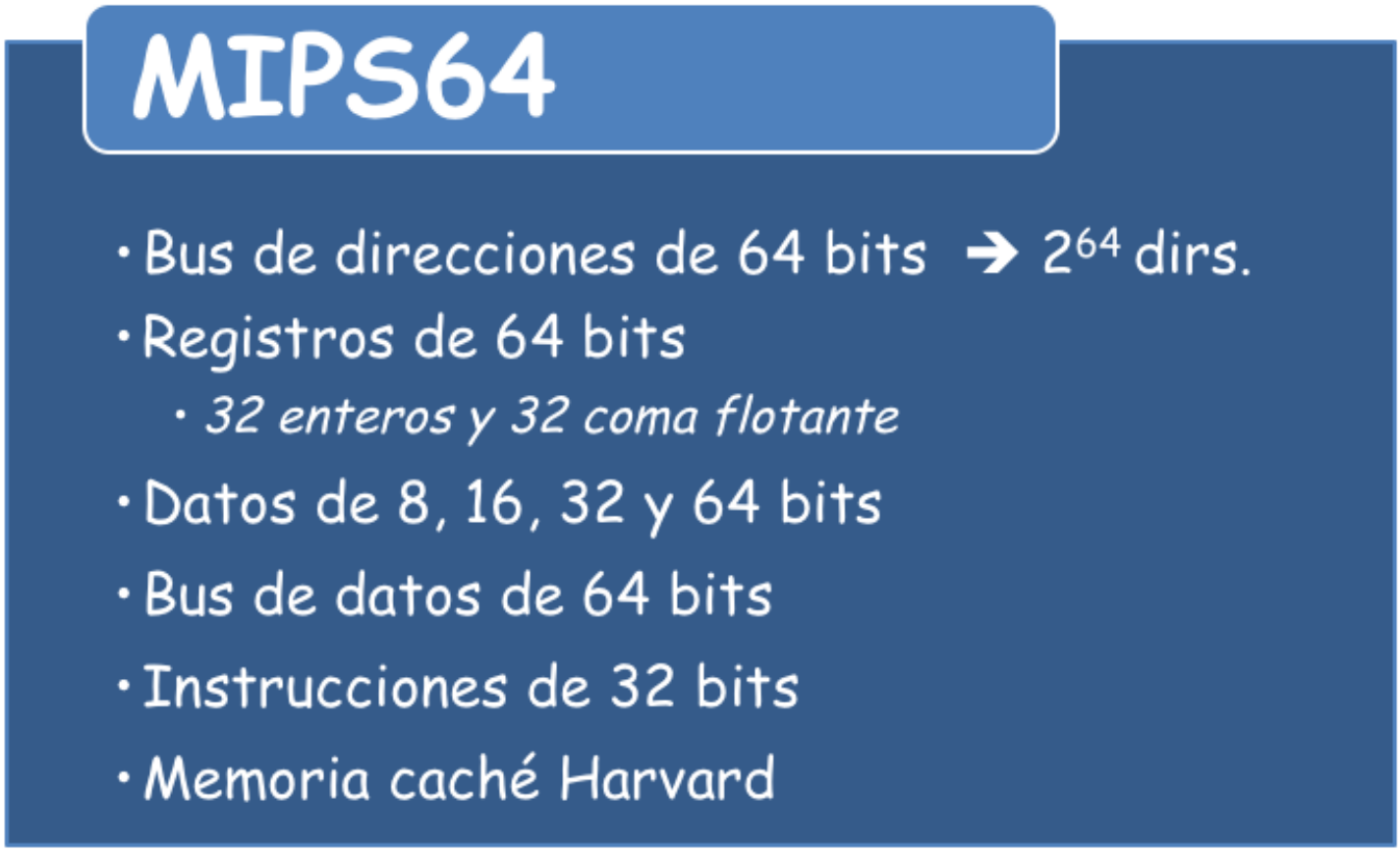
A continuación mostraremos su juego de instrucciones, centrándonos en el subconjunto que vamos a utilizar a lo largo de la asignatura, tanto para la explicación de los conceptos teóricos, como para la realización de las prácticas.

Por último se abordará el lenguaje ensamblador, viendo las directivas que ofrecen para organizar adecuadamente un programa MIPS64.



Las arquitecturas MIPS siguen la filosofía RISC, por lo que cumplen con las siguientes directrices:

- Es una **arquitectura *Load-Store***, es decir, para operar con datos, estos deben cargarse desde la memoria principal en registros internos de la CPU, donde, quedan disponibles para operar con ellos. Así, las únicas instrucciones que acceden a memoria principal son las de carga-almacenamiento, esto es, las que mueven los operandos entre memoria principal y los registros de la CPU. Estando los datos en registros de la CPU, el acceso a ellos es muchísimo más rápido que si estuvieran en memoria principal.
- Ya que las únicas instrucciones de acceso a memoria principal son las de carga-almacenamiento, se dispone de **pocos y sencillos modos de direccionamiento**, lo que facilita enormemente la decodificación de las instrucciones y la obtención de sus operandos.
- La organización del formato de las instrucciones también es muy sencillo, lo que facilita su decodificación, pues dispone de **pocos formatos**, compartiendo todos la misma **longitud fija** de instrucción.
- Ya que no se opera con los datos en memoria principal, se hace necesario disponer de un **generoso conjunto de registros generales** para albergar los distintos datos del programa.



La arquitectura MIPS64 fue desarrollada alrededor del año 2001 y, hasta el momento, es la más novedosa arquitectura de MIPS Technologies.

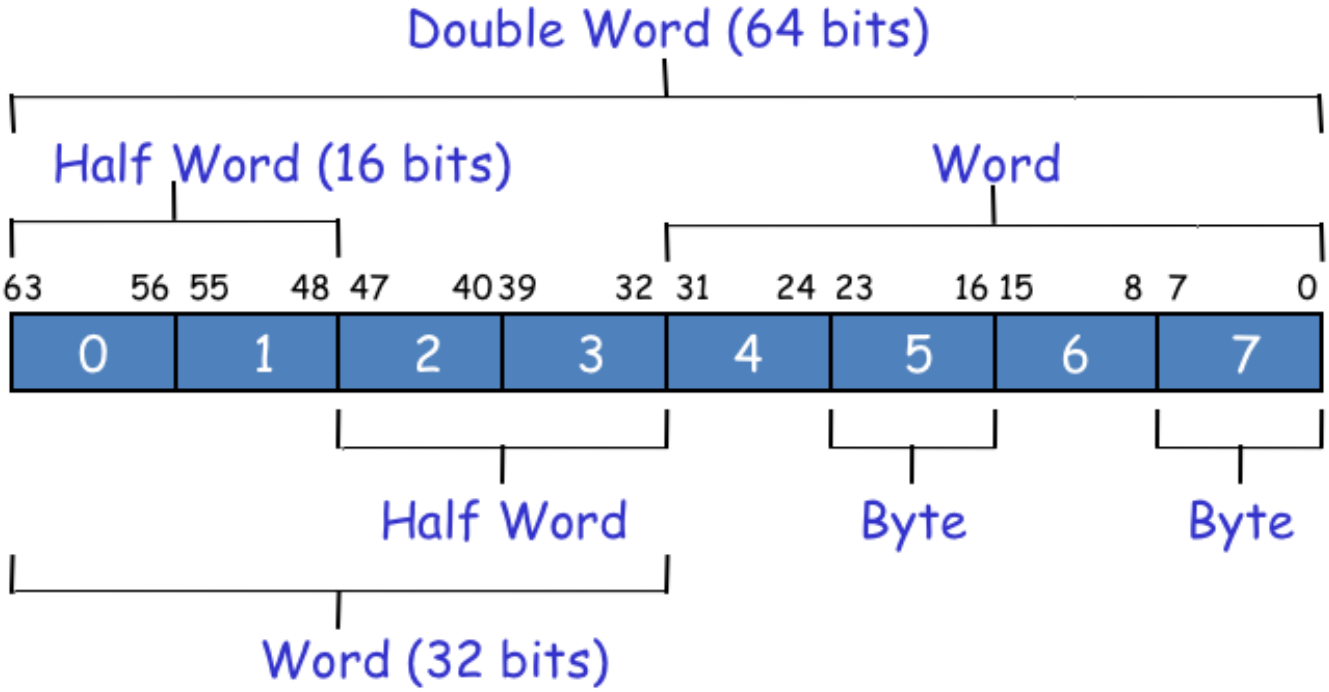
Como características más sobresalientes se puede destacar que tiene un espacio de direccionamiento de 2^{64} celdas de tamaño byte, así como un bus de datos de 64 bits, lo que permite el acceso simultáneo a 8 bytes.

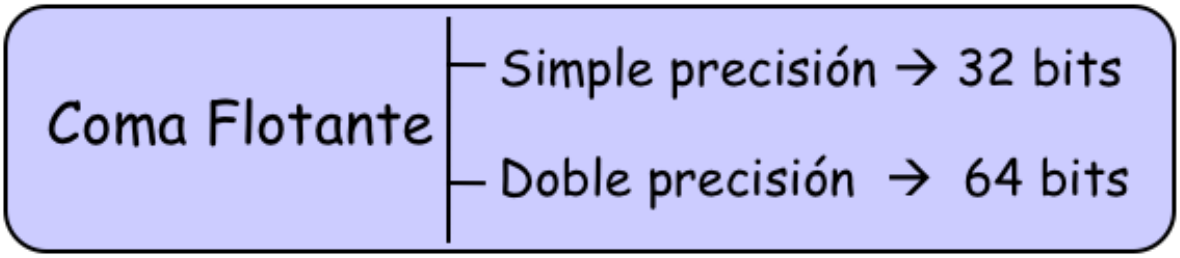
Los datos que maneja pueden ser de 8, 16, 32 y 64 bits.

Dispone de un buen banco de registros, pues tiene 32 registros para manejo de enteros y otros 32 para aritmética en coma flotante, todos ellos de 64 bits.

Por su filosofía RISC, todas las instrucciones son de 32 bits, para facilitar su acceso y decodificación.

Por último, diremos que dispone de cache Harvard, lo que supone tener dos cachés distintas para instrucciones y para datos, lo cual conlleva un mejor acceso simultáneo a instrucciones y datos desde distintas etapas del *pipeline*.





Esta arquitectura define tipos de datos para enteros y para aritmética de coma flotante.

Para los enteros se definen los siguientes tipos de datos:

(En letra **`courier`** se indica el nombre del tipo en ensamblador)

- Byte (8 bits) - **`byte`**
- Media palabra o *Halfword* (16 bits) - **`word16`**
- Palabra, o *Word* (32 bits) - **`word32`**
- Doble palabra, o *Doubleword* (64 bits) – **`word`**

Para coma flotante dispone de datos de simple (32 bits) y doble precisión (64 bits).

Se puede acceder a datos de 8, 16, 32 y 64 bits con las restricciones siguientes:

- ✓Los bytes pueden comenzar en cualquier dirección de memoria.
- ✓Las medias palabras deben comenzar en frontera par (0, 2, 4, ...).
- ✓Las palabras deben comenzar en direcciones divisibles por 4 (0, 4, 8, ...).
- ✓Las dobles palabras deben comenzar en direcciones divisibles por 8 (0, 8, 16, ...).

Los datos deben estar ALINEADOS

Los datos de tipo byte pueden ubicarse en cualquier dirección de memoria. Sin embargo, y según se mostrará en la siguiente página, para que el acceso a datos de múltiples bytes sea eficiente (en cuanto tiempo de acceso) estos deben ubicarse en memoria según las siguientes restricciones.

Los datos de tipo Media Palabra (2 bytes) deben alinearse en frontera par, esto es, deben estar ubicados en direcciones múltiplo de 2 (0, 2, 4, 6, 8, …).

Las Palabras (4 bytes) deben comenzar en una dirección que sea múltiplo de 4.

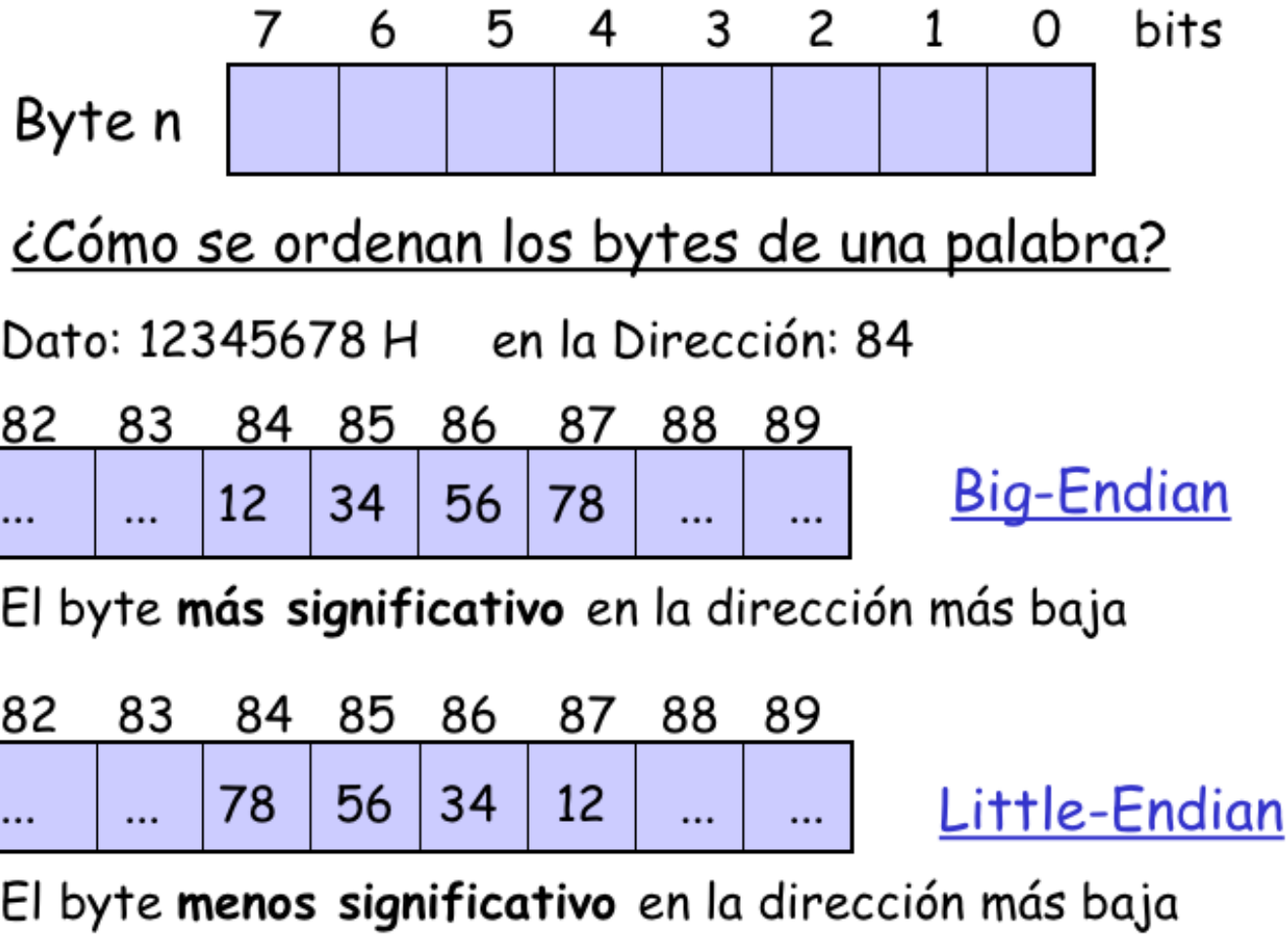
Las Dobles Palabras (8 bytes) deben ubicarse en una dirección que sea múltiplo de 8.

Veamos con un ejemplo del por qué interesa tener los datos alineados

El hecho de que el bus de datos sea de 64 bits implica que en el acceso a la memoria principal se realiza en bloques de 8 en 8 direcciones consecutivas. Así, para acceder a las primeras direcciones de memoria, en un solo acceso se puede leer o escribir en paralelo a los bytes de las 8 primeras direcciones (0, 1, 2, 3, 4, 5, 6 y 7). El siguiente bloque es el compuesto por las direcciones 8, 9, 10, 11, 12, 13, 14 y 15, y así sucesivamente con el 3er. bloque , el 4º bloque, etc.

Según esto, si se ubicase una palabra (4 bytes) en una dirección que no es múltiplo de 4, por ejemplo en la dirección 5, el dato de 4 bytes ocuparía las direcciones 5, 6, 7 y 8. Así, para acceder a esta palabra sería necesario realizar un primer acceso en el que se leerían (o escribirían) los bytes de las direcciones 0 a 7, de las cuales se tomaría el contenido de los bytes 5, 6 y 7; y sería necesario realizar un segundo acceso (a las direcciones 8 a 15) para tomar el cuarto byte de la palabra, el de la dirección 8.

Como vemos, se requerirían dos accesos (16 bytes) para obtener los 4 bytes de la palabra referenciada. Obviamente, esto supone el doble de tiempo que si se obtuviese la palabra en un solo acceso.



Ya hemos visto que la memoria es una colección de muchos bits agrupados, formando bytes, palabras, palabras largas o dobles palabras, etc. Veamos ahora cómo organizar los bits dentro de cada byte y cómo organizar los bytes dentro de cada palabra.

Los bits. Si vemos los bits de un byte como una ristra de ceros y unos de izquierda a derecha, para esa combinación de bits, que debe expresar un número, se debe tomar el acuerdo de cómo numerar los bits dentro del byte, y de establecer cuál es el peso de cada bit.

Aunque se pueden encontrar procesadores con cualquier orden, lo que se suele hacer es numerar los bits de derecha a izquierda, considerando que el bit de menor peso es el de orden cero, es decir, que el bit del extremo derecho, el de orden cero, es el de menor peso, y el bit del extremo izquierdo, el de orden siete, el de mayor peso.

Los bytes. Supongamos que tenemos el número hexadecimal 12345678H almacenado en una palabra de 32 bits de un ordenador cuya unidad de direccionamiento es el byte, y que lo ponemos en la dirección 84. El valor consta de 4 bytes, donde el menos significativo contiene el valor 78, y el más significativo el valor 12. Pues bien, como se puede ver en la figura, hay dos formas de almacenar el valor 12345678:

- La primera opción, conocida como ***big-endian***, almacena el byte más significativo del número en la dirección más baja de memoria. Esto es equivalente al orden normal de escritura de los lenguajes occidentales.
- En la representación inferior, denominada ***little-endian***, es el byte menos significativo del número el que se almacena en el byte de la dirección más baja de memoria.
- ¿Los *strings* cómo se ordenan?

La arquitectura MIPS64 es *bi-endian*, es decir, se puede configurar para comportarse de cualquiera de las dos maneras. El simulador WinMIPS se ejecuta en modo little-endian.

La arquitectura MIPS64 dispone de un grupo registros de propósito general (para enteros) y otro para coma flotante, así como 5 registros de control y un contador de programa.

Los 32 registros de propósito general (GPR) son de 64 bits y se nombran como *r0* a *r31*. Pueden utilizarse como operando fuente o destino, excepto *r0* que está cableado al valor 0.

r31 se suele utilizar, por convención, para almacenar la dirección de retorno en ciertas instrucciones de salto.

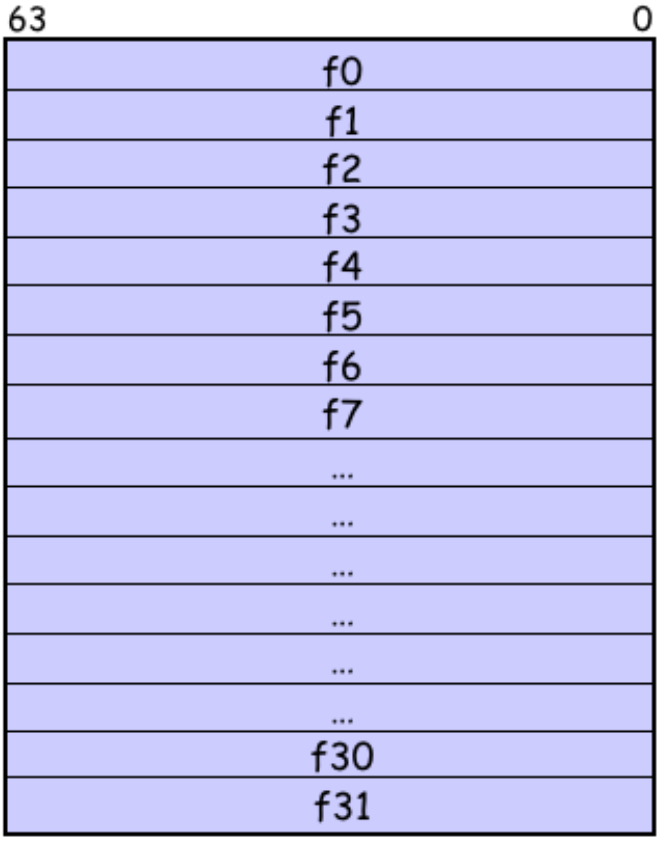
También cuenta con 2 registros especiales de 64 bits (HI y LO) para almacenar resultados de las instrucciones de multiplicación y división.

- En las multiplicaciones, HI y LO almacenan el resultado de la operación.
- En la división entera, el cociente se guarda en LO y el resto en HI.

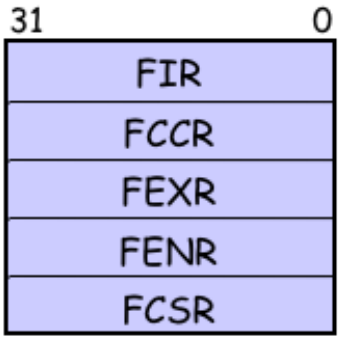
Por último, dispone de un Contador de Programa (PC), también de 64 bits, sólo modificable mediante instrucciones (como las de salto o bifurcación).

En el Simulador WinMIPS, la multiplicación utiliza un registro general para el resultado, es decir, no utiliza los registros especiales HI y LO.

También, en la división entera, el simulador no produce un cociente y un resto en los registros HI y LO, solamente el cociente truncado en el registro de destino.



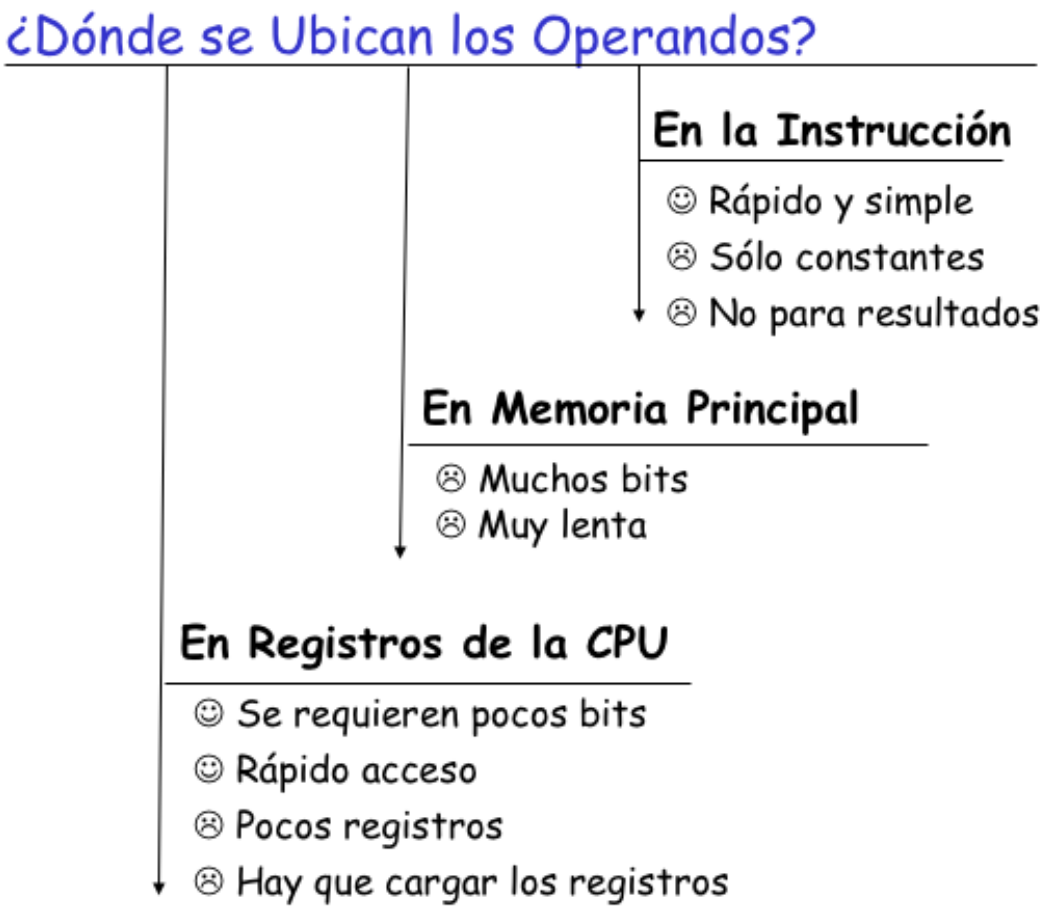
Registros de coma flotante (FPR)



Registros de propósito especial

Para la aritmética de coma flotante cuenta con 32 registros (*f0* a *f31*) que pueden utilizarse como de 32 o 64 bits, dependiendo de si se utiliza simple o doble precisión.

Dentro de esta aritmética, también hay 5 registros de control, de los que no añadiremos nada más, dado que en esta asignatura nos vamos a limitar a operaciones enteras.



El diseño cuidadoso de los códigos de operación es una parte importante del juego de instrucciones de una máquina. No obstante, la mayor parte de los bits de una instrucción se utilizan para especificar los operandos de la operación, por lo que también debe afrontarse con mucho cuidado el modo de direccionar los operandos de las instrucciones.

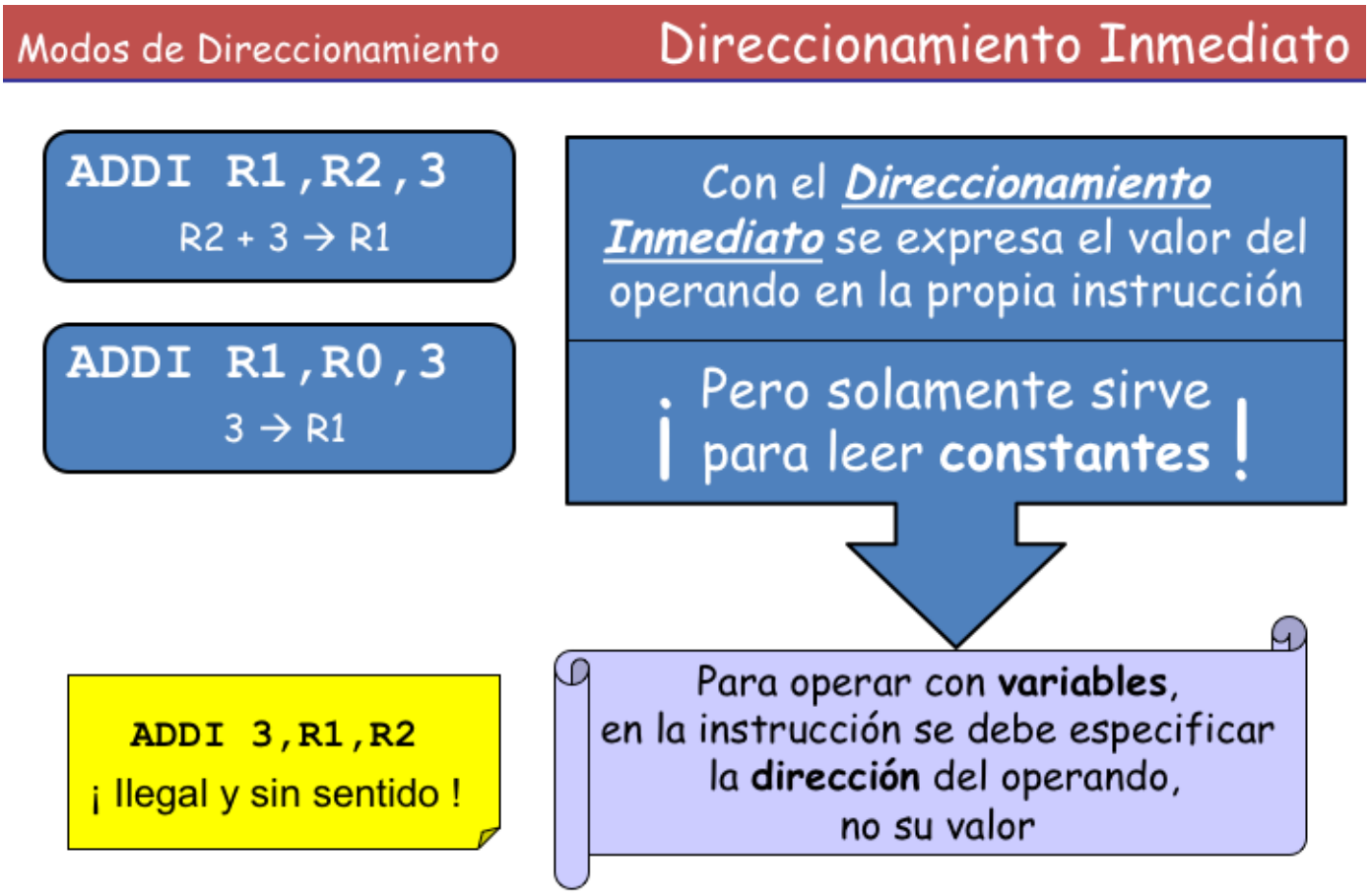
Vamos a considerar ahora dos factores sobre los operandos a tener en cuenta en el diseño del formato de las instrucciones:

- Dónde poner los operandos
- Cuántos operandos se indican en la instrucción

Los operandos pueden estar en tres sitios:

- En la propia instrucción
- En registros de la CPU
- En memoria principal

Con el operando en la misma instrucción, parece claro que el acceso a él es simple y rápido, no obstante estos operandos solamente pueden ser constantes, pues su valor se establece en tiempo de compilación; por esto mismo, tampoco se pueden utilizar como operandos de destino o de resultado. Así, si las variables están ubicadas en memoria principal, parece conveniente utilizar el campo de operando para indicar su **dirección en memoria principal**, sin embargo, para los procesadores actuales, es normal disponer de un bus de direcciones de 32 o 64 bits, lo que implica que el campo de operando en la instrucción requiere también 32 o 64 bits, y si consideramos una instrucción con tres operandos más el código de la operación, la longitud de la instrucción se dispara. Una alternativa puede ser **utilizar registros generales** para contener los operandos. Así, en una máquina con 8 registros se necesitan solamente 3 bits para indicar uno de ellos en el campo de operando. Esto tendría la mejora añadida de que el acceso a un registro es mucho más rápido que a memoria principal. Pero también tiene pegas. Una es que si se dispone de pocos registros generales y se ubican en ellos las variables, se pueden agotar enseguida. Por esto, las arquitecturas recientes tienden proporcionar un número generoso de registros. Otra pega es que para operar con operandos en registros, previamente hay que cargarlos desde memoria principal, lo que significa ejecutar instrucciones adicionales con direcciones de memoria (largas y costosas). Por esto, solamente merece la pena cargar los operandos en registros cuando se van a utilizar repetidamente (lo cual suele ser lo más habitual).



El formato de una instrucción máquina consta de código de operación y de operandos (es una costumbre aceptada en informática el considerar al resultado o destino de la operación también como un operando).

Si el código de operación indica lo que hay que realizar, cada operando debería contener su valor. Así tenemos en el ejemplo una instrucción que copia el valor 3 al registro `r1`. En esta situación en la que **el valor del operando está en la propia instrucción** (el 3, en nuestro ejemplo), decimos que se tiene un **direccionamiento inmediato** al operando.

Pero si nos fijamos, en el operando de destino no está su valor, sino la dirección donde hay que dejar el operando fuente (`r1` en nuestro ejemplo).

Se puede observar que con el direccionamiento inmediato el valor del operando es siempre el mismo (es una constante), puesto que se escribe en la propia instrucción en tiempo de compilación. Entonces ¿qué pasa si queremos operar con variables?

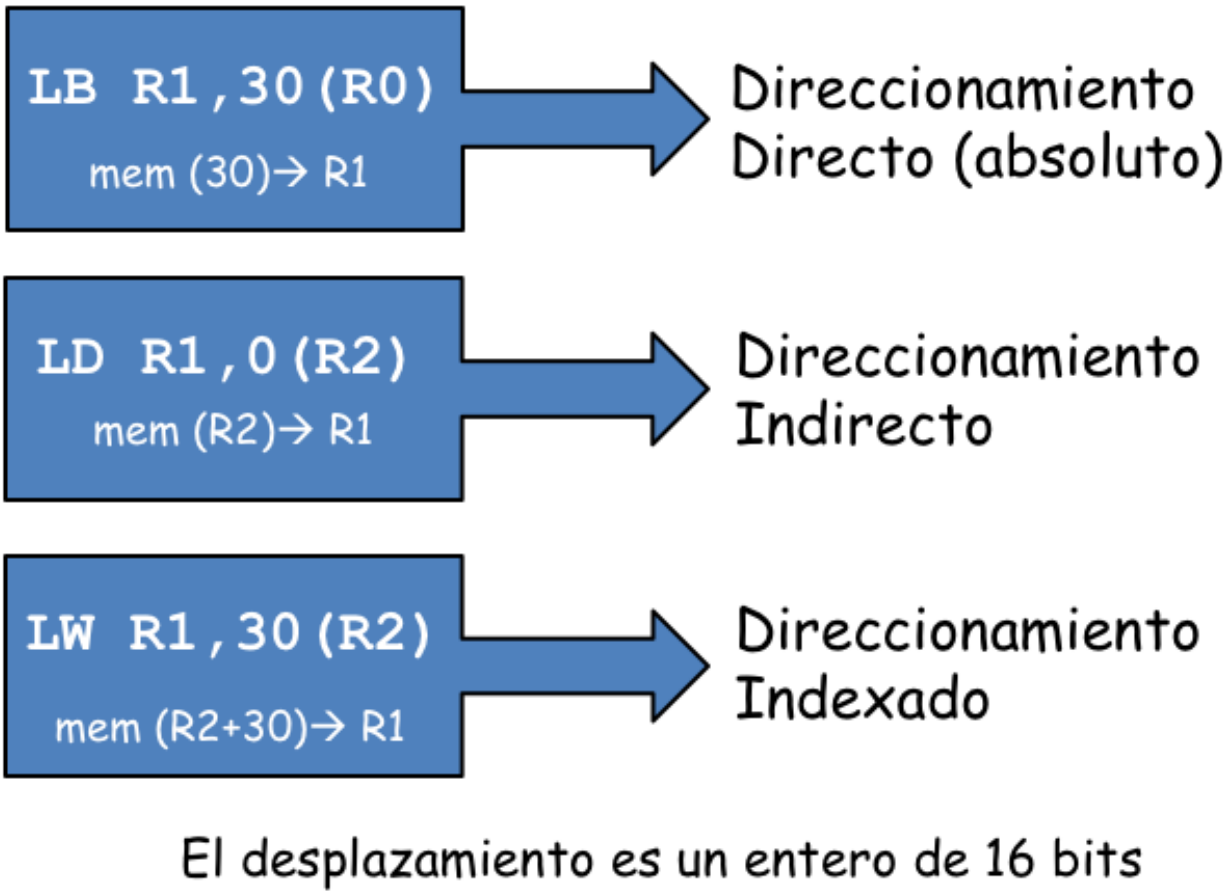
Está claro que **el direccionamiento inmediato no sirve para operar con variables**. En su lugar lo que se necesita es especificar el registro o la dirección de memoria donde está la variable, la cual tendrá valores distintos en distintas circunstancias.

Hay diversos mecanismos o “modos de direccionamiento” para indicar la dirección de un operando en MIPS64. Veámoslos a continuación.

Con **direccionamiento directo a registro**, **el valor del operando se encuentra en uno de los registros generales del MIPS64**. Por esto, en el campo de operando simplemente debe indicarse el nombre de tal registro.

Con este direccionamiento se puede acceder tanto a operandos fuente como de destino o de resultado, y como no implica accesos a memoria principal, resulta un modo rápido de direccionamiento.

Como registros de operandos puede utilizarse cualquiera de los registros generales, a excepción de `r0` (tiene valor fijo “0”) y `r31`, que, por convención, suele utilizarse para contener la dirección de retorno en las llamadas a subrutinas.



Mediante un único formato de direccionamiento (denominado direccionamiento por desplazamiento) y sus variaciones, MIPS64 consigue 3 modos distintos de direccionamiento a memoria:

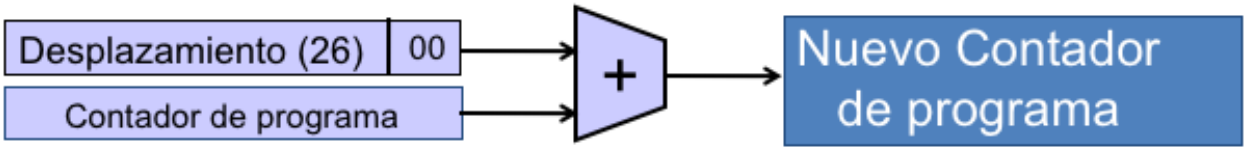
- **Direccionamiento directo o absoluto**, indicando ahora `r0` como el registro de indirección, con lo que la dirección se forma solamente con el valor del desplazamiento.
- **Direccionamiento indirecto**, al indicar un cero como desplazamiento, la dirección se obtiene simplemente a partir del contenido del registro (`r2`).
- **Direccionamiento por desplazamiento** (o indexado), en el que la dirección se forma mediante la suma del contenido de un registro (`r2`) más un desplazamiento explícito (30, en el ejemplo).

En todos estos casos, el desplazamiento es un entero de 16 bits, por lo que con un valor dado en un registro base se consigue un abanico de direccionamiento de $\pm 2^{15}$ respecto a la dirección del registro base.

La dirección de salto en las bifurcaciones se obtiene mediante un desplazamiento relativo al Contador de Programa

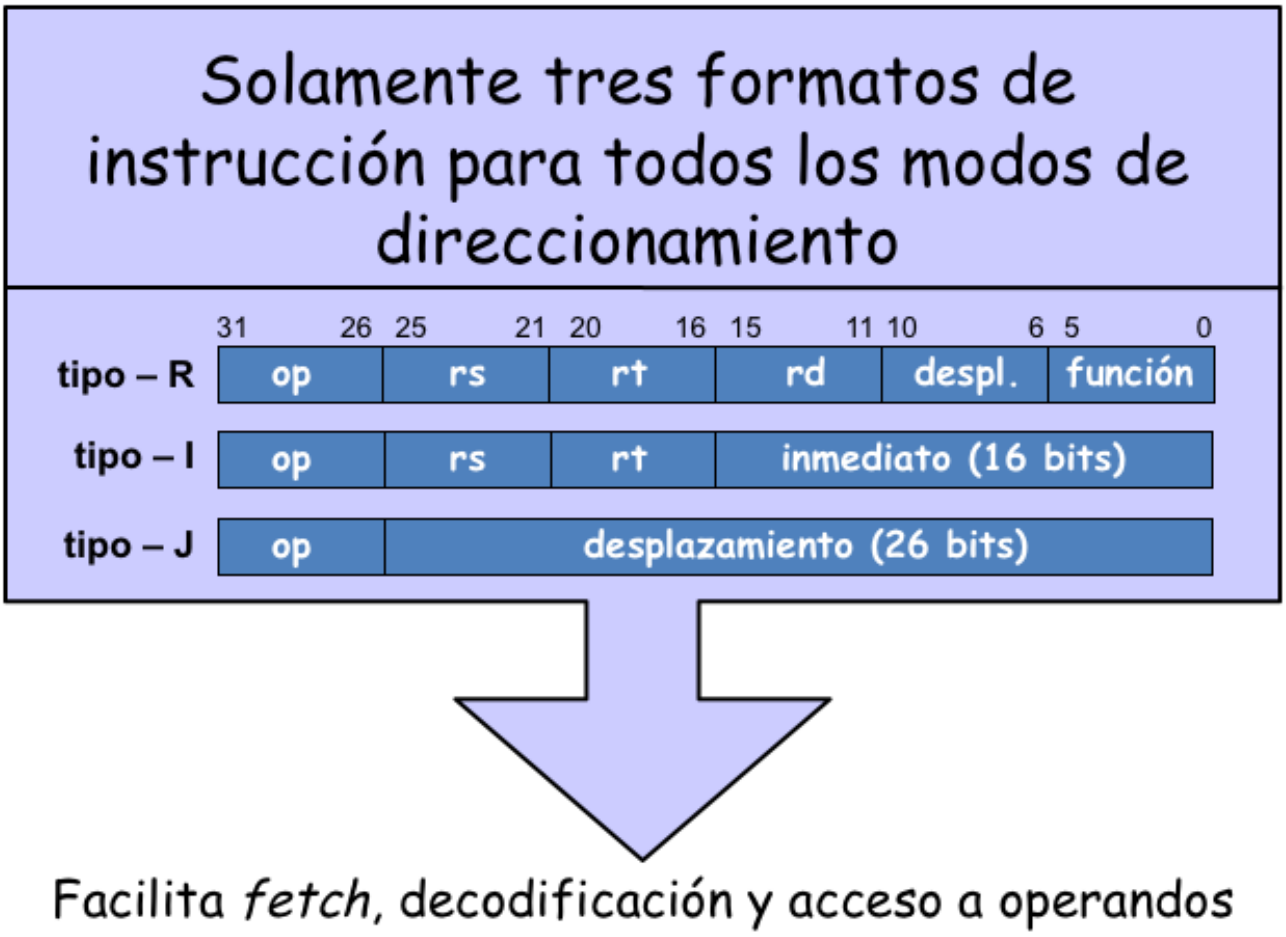
J bucle

PC+despl (26 bits) → PC



En las instrucciones de salto incondicional se utiliza un tipo especial de direccionamiento, donde la dirección no corresponde a un operando, sino que es simplemente la dirección de la siguiente instrucción a ejecutar.

En este caso, la dirección de la siguiente instrucción se forma mediante un desplazamiento (valor inmediato) más el valor del Contador de Programa. El destino de esta suma es el nuevo Contador de Programa.



Todos los modos de direccionamiento que se han indicado en las últimas páginas se consiguen con tan solo tres formatos de instrucción y, en todos los casos, con la misma longitud de instrucción: 32 bits.

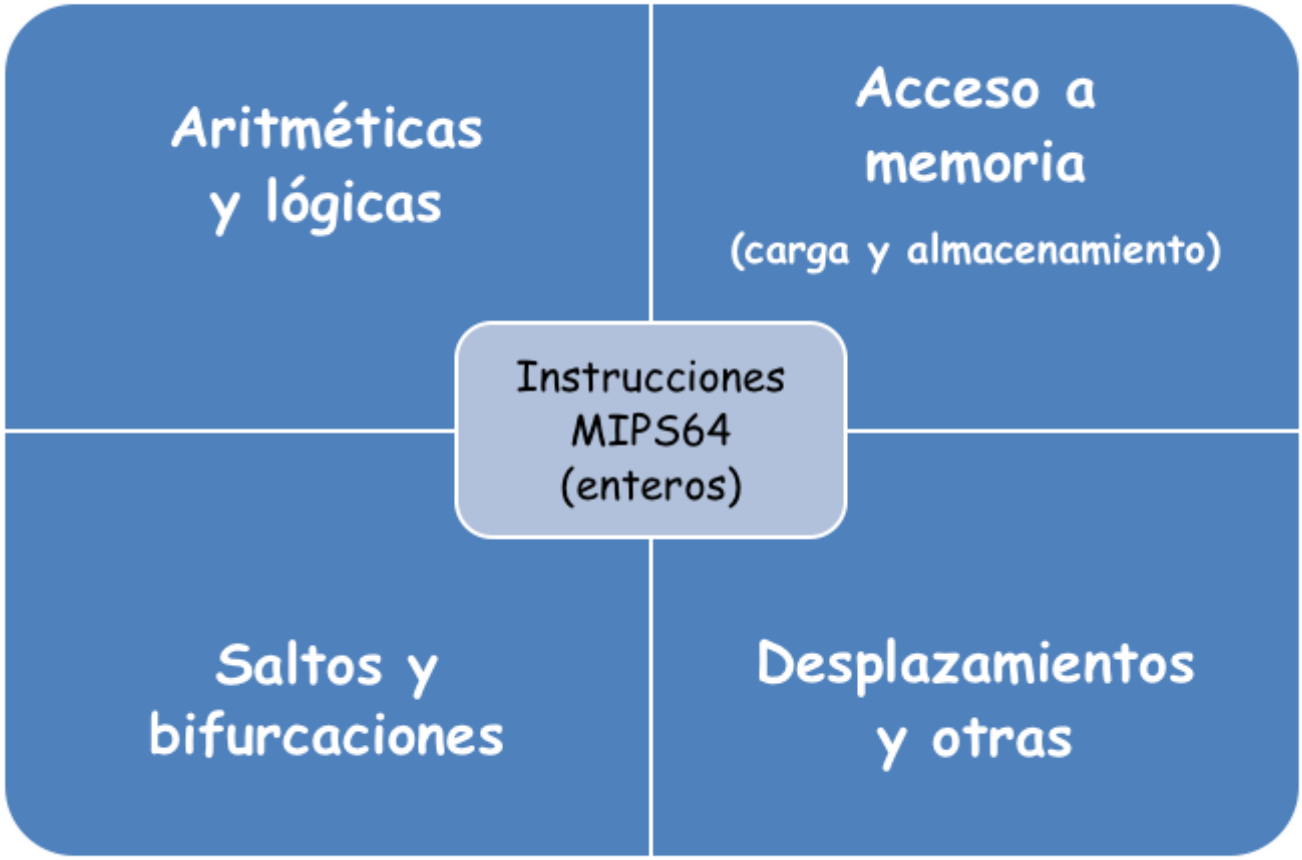
Con este reducido número de formatos y una longitud fija de instrucción, se consigue minimizar el tiempo de decodificación de la instrucción y del cálculo de la dirección de los operandos.

Las instrucciones con **formato de tipo R** corresponden a las operaciones de registro a registro en las que interviene la Unidad Aritmético-Lógica. en este caso, el campo *Función* establece la operación a realizar en la UAL (suma, resta, …).

El **formato de tipo I** es el utilizado por las instrucciones de carga y almacenamiento (*load-store*) y las instrucciones de salto condicional.

Las instrucciones de **tipo J** se utilizan exclusivamente para instrucciones de salto incondicional y retorno de excepciones.

2. Juego de Instrucciones



La arquitectura MIPS64 cuenta con instrucciones para operaciones con enteros y para operaciones con datos en coma flotante.

En estos apuntes vamos a comentar las instrucciones con enteros y en coma flotante, aunque nos vamos a centrar en el subconjunto de instrucciones que ofrece el simulador WinMIPS64, el cual es suficiente para la realización de las prácticas del asignatura. Según su funcionalidad, las instrucciones se pueden organizar en los siguientes grupos:

- **Aritméticas y lógicas**: Son las encargadas de las operaciones aritméticas básicas (suma, resta, multiplicación y división), y de las operaciones con al algebra booleana, concretamente, *AND*, OR y *EXCLUSIVE OR*).
- **Acceso a memoria**: Estas instrucciones se ocupan de cargar en registros el contenido de las celdas de memoria y viceversa, las que almacenan en memoria el contenido de los registros generales. Estas son las únicas instrucciones con acceso a operandos en memoria, el resto, salvo excepciones, operan únicamente con operandos en registros o inmediatos.
- **Saltos y bifurcaciones**: Estas instrucciones rompen la ejecución secuencial de instrucciones según su dirección en memoria. En las instrucciones de salto incondicional, simplemente se cambia el valor del Contador de Programa. Las bifurcaciones (saltos condicionales) también lo hacen, pero solamente si se cumple una cierta condición.
- **Desplazamientos y otras**: En este último grupo se encuentran las instrucciones de desplazamiento de datos en registros y otras instrucciones variadas que veremos más detalladamente un poco más adelante, cuando lleguemos a ellas.

En la arquitectura MIPS64 completa, en casi todos los tipos de instrucciones hay variaciones para operar con todo tipo de operandos (byte, media palabra, palabra y doble palabra), así como para datos alineados y sin alinear.

Los códigos de operación de algunas de las instrucciones de este simulador no corresponden exactamente con las de MIPS64.

Instrucción		Operación
`dadd Rd,Rs,Rt`	Suma doble palabra	`Rd = Rs + Rt`
`daddu Rd,Rs,Rt`	Suma doble palabra (sin signo)	`Rd = Rs + Rt`
`dsub Rd,Rs,Rt`	Resta doble palabra	`Rd = Rs - Rt`
`dsubu Rd,Rs,Rt`	Resta doble palabra (sin signo)	`Rd = Rs - Rt`
`dmul Rd,Rs,Rt`	Multiplica doble palabra	`Rd = Rs * Rt`
`dmulu Rd,Rs,Rt`	Multiplica dp (sin signo)	`Rd = Rs * Rt`
`ddiv Rd,Rs,Rt`	Divide doble palabra	`Rd = Rs / Rt`
`ddivu Rd,Rs,Rt`	Divide dp (sin signo)	`Rd = Rs / Rt`
`daddi Rd,Rs,inm`	Suma doble palabra inmediato	`Rd = Rs + inm`
`daddui Rd,Rs,inm`	Suma dp inm. (sin signo)	`Rd = Rs + inm`

Como ya dijimos, estas instrucciones se ocupan de las operaciones aritméticas básicas, suma, resta, multiplicación y división. Todas ellas se pueden realizar con aritmética entera o sin signo.

Todas las operaciones se realizan con operandos en registros o, como en el caso de la suma, también se permiten los valores inmediatos.

Las operaciones con signo producen un *trap* si hay desbordamiento (*overflow*). No así, las operaciones sin signo (*unsigned*).

Las instrucciones `dmul`, `dmulu`, `ddiv` y `ddivu` son propias del simulador WinMIPS64. En la arquitectura MIPS64 estas instrucciones corresponden a **`dmult`**, **`dmultu`**, **`ddiv`** y **`ddivu`**, que únicamente tienen 2 operandos fuente, siendo el destino los registros HI y LO del procesador. Para acceder a estos registros es necesario utilizar las instrucciones **`mfhi`** y **`mflo`**.

Instrucción		Operación
`and Rd,Rs,Rt`	And lógico	`Rd = Rs AND Rt`
`or Rd,Rs,Rt`	Or lógico	`Rd = Rs OR Rt`
`xor Rd,Rs,Rt`	Or exclusivo	`Rd = Rs XOR Rt`
`andi Rt,Rs,inm`	And lógico inmediato	`Rt = Rs AND inm`
`ori Rt,Rs,inm`	Or lógico inmediato	`Rt = Rs OR inm`
`xori Rt,Rs,inm`	Or exclusivo inmediato	`Rt = Rs XOR inm`

Como vemos, este subconjunto de instrucciones booleanas realizan las operaciones AND, OR y OR Exclusivo, con los operandos en registros, o con uno de ellos inmediato.

Carga en registros

Instrucción		Operación
`lb Rt,offset(Rs)`	Carga de 1 octeto con extensión de signo	$Rt_{[0..7]}$= MEM[Rs+offset]
`lh Rt,offset(Rs)`	Carga de media palabra (16 bits) con extensión de signo	$Rt_{[0..15]}$= MEM[Rs+offset]
`lw Rt,offset(Rs)`	Carga de una palabra (32 bits) con extensión signo	$Rt_{[0..31]}$= MEM[Rs+offset]
`ld Rt,offset(Rs)`	Carga de una doble palabra (64 bits)	$Rt_{[0..63]}$= MEM[Rs+offset]
`lbu Rt,offset(Rs)`	Carga de 1 octeto sin extensión de signo	$Rt_{[0..7]}$= MEM[Rs+offset]
`lhu Rt,offset(Rs)`	Carga de media palabra sin extensión de signo	$Rt_{[0..15]}$= MEM[Rs+offset]
`lwu Rt,offset(Rs)`	Carga de una palabra sin extensión signo	$Rt_{[0..31]}$= MEM[Rs+offset]
`lui Rt,imm`	Carga la palabra alta con un valor inmediato	Rt = imm || 0^{16}

En general, y siguiendo la filosofía RISC, las instrucciones de la arquitectura MIPS64 trabajan con operandos en registros, siendo únicamente las instrucciones de carga y almacenamiento las que acceden a memoria principal para realizar el intercambio de datos entre ésta y los registros.

Las instrucciones de carga se ocupan de mover o, mejor, copiar, el contenido de celdas de memoria a los registros generales, aunque también hay instrucciones para cargar los registros con valores inmediatos, como **`daddi`**.

Estas instrucciones operan con datos de todos los tamaños (hay instrucciones específicas para cada tamaño): byte, media palabra, palabra y doble palabra.

Las cargas pueden ser con signo o sin signo, es decir, realizando extensión de signo o sin ella. La extensión del signo de un byte que se carga de memoria a un registro (de 64 bits) consiste en que al copiar el valor del byte al registro, el bit de mayor peso del byte (el 7º) se extiende al resto de los bits de la izquierda del registro, es decir a los bits 8 a 63. Los bits se numeran de 0 a 63, de derecha a izquierda.

Obsérvese que no existe la instrucción **`ldu`**. No es necesaria, pues sería igual a la instrucción **`ld`**.

Almacenamiento en memoria

Instrucción		Operación
`sb Rt,offset(Rs)`	Almacena un octeto	MEM[Rs+offset] = $Rt_{[0..7]}$
`sh Rt,offset(Rs)`	Almacena media palabra (16 bits)	MEM[Rs+offset] = $Rt_{[0..15]}$
`sw Rt,offset(Rs)`	Almacena una palabra (32 bits)	MEM[Rs+offset] = $Rt_{[0..31]}$
`sd Rt,offset(Rs)`	Almacena una doble palabra (64 bits)	MEM[Rs+offset] = $Rt_{[0..63]}$

- Todas las direcciones deben estar **alineadas** en su correspondiente frontera
- En accesos multiocteto tendrá que tenerse en cuenta el modelo de "orden" elegido:
  - ***Big endian*** almacena el octeto más significativo en la dirección más baja
  - ***Little endian*** el octeto más significativo en la dirección más alta.

Las operaciones de almacenamiento se encargan de llevar a memoria el contenido de los registros generales.

Como vemos en el gráfico, hay instrucciones de almacenamiento para los cuatro tamaños de datos.

Aunque en el juego completo de instrucciones de MIPS64 hay instrucciones de almacenamiento para datos multiocteto no alineados, aquí solamente trabajaremos con datos alineados a la frontera correspondiente (1, 2, 4, 8).

No debe olvidarse que en MIPS64 se puede trabajar con los dos modos de ordenación de datos multybyte, por lo que en las operaciones con datos compuestos por múltiples octetos debe tenerse en cuenta el orden de almacenamiento de los datos. (Recordamos que el simulador WinMIPS utiliza el modo little-endian).

Instrucción		Operación
`j dest`	El salto está restringido a una zona de 256 MB	PC = PC $_{[63..28]}$ || instr_index || 0^2
`jr Rs`	Salto a cualquier dirección	PC = Rs
`jal dest`	Almacena la dirección de retorno (PC+8) en el registro R31.	PC = PC $_{[63..28]}$ || instr_index || 0^2
`jalr Rs`	Almacena la dirección de retorno (PC+8) en el registro R31	PC = Rs

¡La dirección de salto debe estar alineada!

El flujo de ejecución secuencial de un programa se puede romper mediante las instrucciones de salto incondicional o de bifurcaciones condicionales.

Aunque MIPS64 dispone de 6 instrucciones de salto incondicional, las cuatro instrucciones básicas son las siguientes:

- **`j (Jump)`**: Esta instrucción bifurca a una instrucción dentro de la actual región de 256 MB (una porción de memoria alineada a 256 MB). Para ello, los 28 bits de menor peso del PC se sustituyen por el operando inmediato de la instrucción (26 bits) desplazado 2 bits a la izquierda (ya que todas las instrucciones son de 4 bytes).
- **`jr (Jump Register)`**: Mediante esta instrucción se bifurca a la dirección contenida en el registro que se especifica como operando, por lo que se puede saltar a cualquier dirección del espacio de direccionamiento.
- **`jal (Jump and Link)`**: Esta instrucción cede el control a una rutina situada en la actual región de 256 MB. Antes del salto guarda la dirección de la siguiente instrucción en secuencia (PC+8) en el registro `r31`.
- **`jalr (Jump and Link Register)`**: Esta es otra variante del salto a subrutina, pero, en este caso, la dirección de la rutina está indicada en el registro que se especifica como operando. Igualmente la dirección de retorno (PC+8) se guarda en el registro `r31`.

Todas las direcciones de salto deben corresponder a instrucciones que se encuentran debidamente alineadas (frontera de 4).

Los saltos tienen un hueco de retardo: La instrucción que se encuentra en la dirección inmediatamente a continuación a la del salto, se dice que está en el hueco de retardo. En todos los saltos, **SIEMPRE** se ejecuta la instrucción del hueco de retardo, antes de pasar a ejecutar la instrucción de destino del salto.

Instrucción		Operación
`beq Rs,Rt,dest`	El salto está restringido a una zona de ±128 KB	Si Rs = Rt → PC = $PC_{[63..18]}$ + instr_index || 0^2
`bne Rs,Rt,dest`	El salto está restringido a una zona de ±128 KB.	Si Rs ≠ Rt → PC = $PC_{[63..18]}$ + instr_index || 0^2
`beqz Rs,dest`	El salto está restringido a una zona de ±128 KB.	Si Rs = 0 → PC = $PC_{[63..18]}$ + instr_index || 0^2
`bnez Rs,dest`	El salto está restringido a una zona de ±128 KB.	Si Rs ≠ 0 → PC = $PC_{[63..18]}$ || instr_index || 0^2

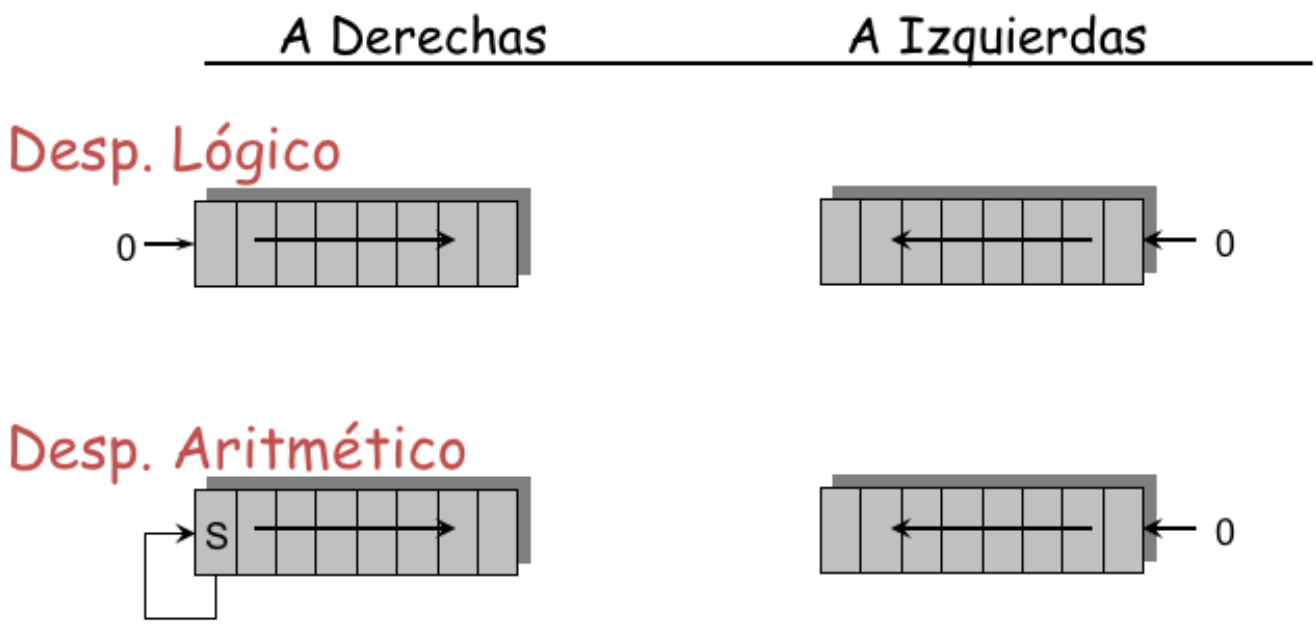
A los saltos condicionales se les suele denominar “bifurcaciones”. Aunque MIPS64 dispone de 26 instrucciones de bifurcación condicional, aquí veremos solamente el subconjunto necesario para la asignatura.

Todas las bifurcaciones contienen una condición, los operandos necesarios para la comparación según el código de condición, y una dirección de salto a la que se cede el control si se cumple la condición.

- **`beq (Branch on Equal)`**: Si los registros Rs y Rt son iguales se salta a la dirección de destino del salto.
- **`bne (Branch on Not Equal)`**: Si los registros Rs y Rt no son iguales se salta a la dirección de destino del salto.
- **`beqz (Branch on Equal to Zero)`**: Si el registro Rs es igual a cero, se salta a la dirección de destino del salto.
- **`bnez ( Branch on Not Equal to Zero)`**: Si el registro Rs no es igual a cero, se salta a la dirección de destino del salto.

La dirección de destino debe estar en un rango de ±128 KB de distancia respecto al PC en el momento de ejecución de la bifurcación.

Las bifurcaciones tienen un hueco de retardo: La instrucción que se encuentra en la dirección inmediatamente a continuación de la bifurcación, se dice que está en el hueco de retardo. Si la condición se cumple, después de ejecutar la instrucción que se encuentra en el hueco de retardo, se salta a la dirección de destino del salto, si no se cumple la condición, se continúa con la siguiente instrucción a la del hueco de retardo.



Desplazamientos

Los desplazamientos se aplican sobre un único operando, y pueden actuar a derechas o a izquierdas.

Los desplazamientos pueden ser lógicos o aritméticos. El desplazamiento aritmético a la derecha va replicando el bit de signo (el del extremo izquierdo) a medida que se va desplazando, por lo que se mantiene el signo en el resultado. En el desplazamiento aritmético a izquierdas, el bit de signo no se ve afectado (siempre que no haya desbordamiento), por lo que también se mantiene el signo en el resultado. Como ya es sabido, el efecto aritmético de un desplazamiento a izquierdas es el de ir multiplicando por las sucesivas potencias de 2, mientras que el desplazamiento aritmético a la derecha produce la división por las sucesivas potencias de 2.

Instrucción		Operación
`dsll Rd,Rt,imm`	Desplazamiento lógico a Izda.	`Rd = Rt << imm`
`dsrl Rd,Rt,imm`	Desplazamiento lógico a Dcha.	`Rd = Rt >> imm`
`dsra Rd,Rt,imm`	Desplazamiento aritm. a Dcha.	`Rd = Rt >> imm`
`dsllv Rd,Rt,Rs`	Desplazamiento lógico a izda. Sólo se utilizan los bits 0..5 de Rs	`Rd = Rt << Rs`
`dsrlv Rd,Rt,Rs`	Desplazamiento lógico a dcha. Sólo se utilizan los bits 0..5 de Rs	`Rd = Rt >> Rs`
`dsrav Rd,Rt,Rs`	Desplazamiento aritmético a dcha. Sólo se utilizan los bits 0..5 de Rs	`Rd = Rt >> Rs`

En todas las operaciones, el contenido del 2º operando (Rt), se desplaza el número de bits indicado en el 3er. operando (valor inmediato de 5 bits) y el resultado se deja en el 1er. operando (Rd).

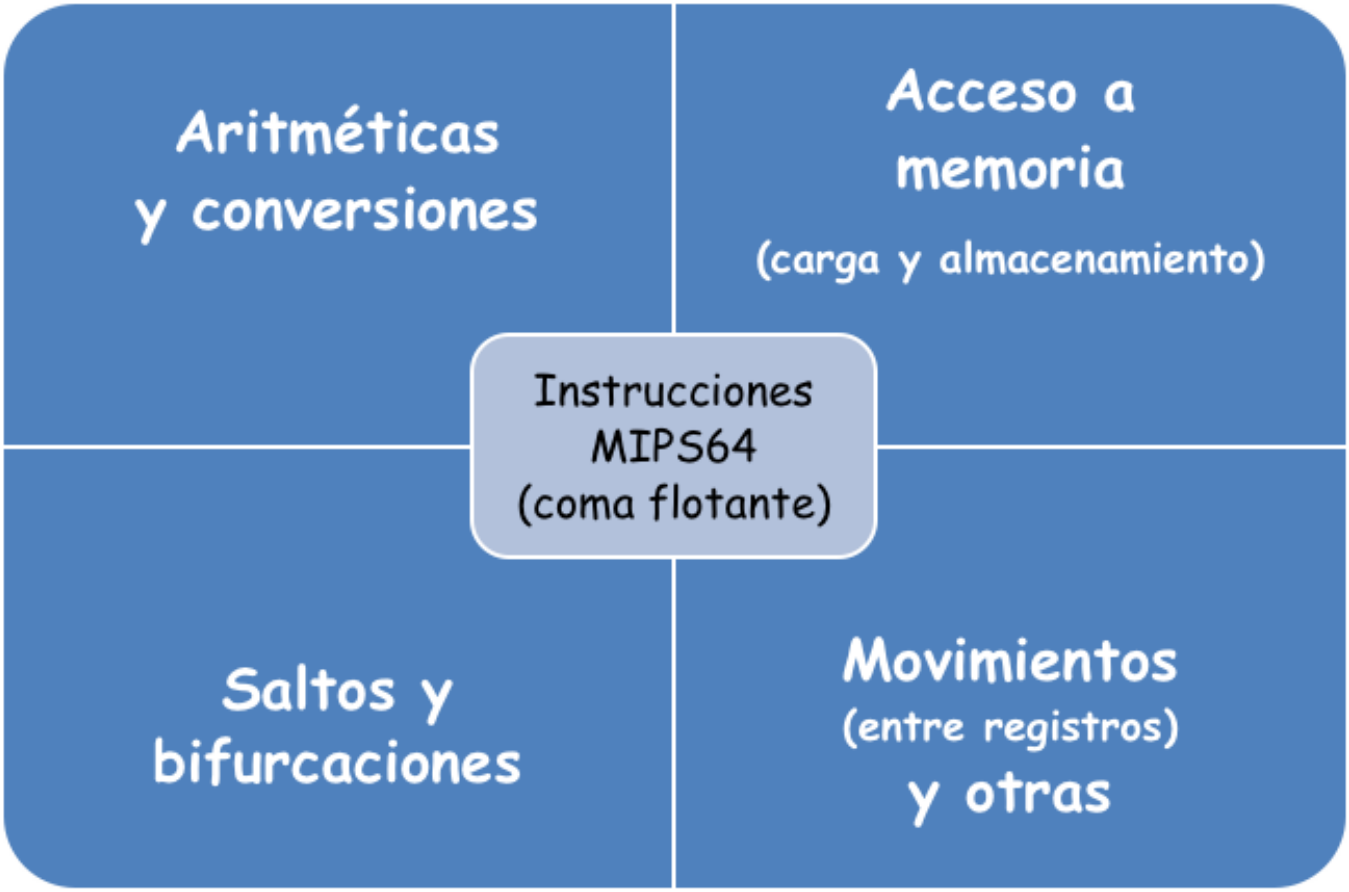
Cuando el número de desplazamientos es variable (se indica mediante el registro Rs), solamente se consideran los 5 bits de menor peso de dicho registro.

- **`dsll (Doubleword Shift Left Logical) Rt`** se desplaza a la izquierda el número de bits indicado en el operando inmediato.
- **`dsrl (Doubleword Shift Right Logical) Rt`** se desplaza a la derecha el número de bits indicado en el operando inmediato.
- **`dsra (Doubleword Shift Right Arithmetic) Rt`** se desplaza a la derecha el número de bits indicado en el operando inmediato y manteniendo el signo del contenido original.
- **`dsllv (Doubleword Shift Left Logical Variable) Rt`** se desplaza a la izquierda el número de bits indicado en registro **`Rs`**.
- **`dsrlv (Doubleword Shift Right Logical Variable) Rt`** se desplaza a la derecha el número de bits indicado en registro **`Rs`**.
- **`dsrav (Doubleword Shift Right Arithmetic Variable) Rt`** se desplaza a la derecha el número de bits indicado en registro **`Rs y manteniendo el signo del contenido original`**.

Obsérvese que no es necesaria la operación de desplazamiento aritmético a la izquierda, ya que el objetivo se consigue con el desplazamiento lógico a la izquierda .

Instrucción		Operación
`movz Rd,Rs,Rt`	Movimiento condicional	`Si Rt=0 → Rd = Rs`
`movn Rd,Rs,Rt`	Movimiento condicional	`Si Rt≠0 → Rd = Rs`
`slt Rd,Rs,Rt`	Comparación entera con signo	`Si Rs<Rt → Rd=1 else Rd = 0`
`sltu Rd,Rs,Rt`	Comparación entera sin signo	`Si Rs<Rt → Rd=1 else Rd = 0`
`slti Rd,Rs,imm`	Comp. inmediata con signo	`Si Rs<ext_sign(imm) → Rd = 1 else Rd = 0`
`sltiu Rd,Rs,imm`	Comp. Inmediata sin signo	`Si Rs<ext_sign(imm) → Rd=1 else Rd = 0`
`nop`	Corresponde a la instrucción sll r0,r0,0	`No Operación`
`halt`	¡No existe en MIPS64!	`Vuelta al S.O.`

- **`movz (Move Conditional on Zero)`** Si el valor de Rt es cero, el contenido de Rs se copia a Rd.
- **`movn (Move Conditional on Not Zero)`** Si el valor de Rt no es cero, el contenido de Rs se copia a Rd.
- **`slt (Set on Less Than)`** Si Rs es menor que Rt, pone un 1 (cierto) en Rd; si no, pone un cero (falso). (La comparación es de enteros con signo).
- **`sltu (Set on Less Than Unsigned)`** Si Rs es menor que Rt, pone un 1 (cierto) en Rd; si no, pone un cero (falso). (La comparación es de enteros sin signo).
- **`slti (Set on Less Than Inmediate)`** Si Rs es menor que el valor inmediato de 16 bits, pone un 1 (cierto) en Rd; si no, pone un cero (falso). La comparación es de enteros con signo. Antes de la comparación, el valor inmediato se extiende a 64 bits con signo.
- **`sltiu (Set on Less Than Inmediate Unsigned)`** Si Rs es menor que el valor inmediato de 16 bits, pone un 1 (cierto) en Rd; si no, pone un cero (falso). La comparación es de enteros sin signo. Antes de la comparación, el valor inmediato se extiende a 64 bits con signo.
- **`nop (No Operation)`** Esta operación es realmente **`sll r0,r0,0 (Shift Word Left Logical)`**. Es la operación que se utiliza normalmente en los huecos de retardo de los saltos.
- **`Halt`** La instrucción máquina `halt` no existe realmente en MIPS64, solamente la ofrece el simulador WinMIPS64 para detener la ejecución del programa, por lo que debe utilizarse siempre como la última instrucción de un programa que tiene fin, es decir, que no consta de un bucle infinito.



Como ya dijimos anteriormente, la arquitectura MIPS64 dispone de un juego de instrucciones específicas para operaciones en coma flotante.

Veremos aquí un resumen muy escueto de las instrucciones de esta aritmética que están disponibles en el simulador de prácticas de la asignatura. En algunos casos, los códigos nemotécnicos de las instrucciones ni siquiera se corresponden con los códigos de un procesador MIPS64 real.

Instrucción		Operación
`add.d Fd,Fs,Ft`	Suma dos registros de coma flotante	`Fd = Fs + Ft`
`sub.d Fd,Fs,Ft`	Resta de dos registros de coma flotante	`Fd = Fs - Ft`
`mul.d Fd,Fs,Ft`	Multiplicación de dos registros	`Fd = Fs * Ft`
`div.d Fd,Fs,Ft`	División en coma flotante	`Fd = Fs / Ft`

Poco hay que añadir a la descripción de estas instrucciones de aritmética básica en coma flotante (*floating point*), pues se comportan de igual manera que sus hermanas de aritmética entera.

En este caso se utilizan registros del banco de coma flotante.

Copias y conversiones (coma flotante)

Instrucción		Operación
`mov.d Fd,Fs`	Copia de registros de coma flotante	`Fd = Fs`
`mtc1 Rt,Fs`	Copia de un registro de enteros a uno de coma flotante	$Fs_{[31..0]} = Rt_{[31..0]}$
`mfc1 Rt,Fs`	Copia de un registro de coma flotante a uno de enteros	$Rt = ext\_signo(Fs_{[31..0]})$
`cvt.d.l Fd,Fs`	Convierte un entero de 64 bits a uno en coma flotante	`Fd = toDoubleFP(Fs)`
`cvt.l.d Fd,Fs`	Convierte un número en coma flotante a un entero de 64 bits	`Fd = to64integer(Fs)`

La instrucción **`mov.d`** se utiliza para la copia de valores entre registros de coma flotante.

Las instrucciones **`mtc1`** y **`mfc1`** copian valores entre registros de bancos distintos (enteros y coma flotante). En este caso la copia afecta solamente a los 32 bits de menor peso de los registros. En el procesador MIPS existe una versión *double* que realiza la copia de los 64 bits.

Por último, en este grupo, las instrucciones **`cvt`** convierten datos entre coma fija (aritmética entera) y coma flotante.

Instrucción		Operación
`l.d Ft,offset(Rs)`	Carga un número de 64 bits en coma flotante	$Ft_{[0..63]}$ = MEM[Rs+offset]
`s.d Ft,offset(Rs)`	Almacena un número de 64 bits en coma flotante	MEM[Rs+offset] = $Ft_{[0..63]}$

Estas son las instrucciones de carga y almacenamiento para registros de coma flotante, con cometido equivalente al de **`ld`** y **`sd`** con los registros de coma fija (enteros).

Saltos Condicionales (coma flotante)

Instrucción		Operación
`c.lt.d Fs,Ft`	Establece el código de condición si se cumple la condición "menor que"	`CC = (Fs < Ft)`
`c.le.d Fs,Ft`	Establece el código de condición si se cumple la condición "menor o igual que"	`CC = (Fs <= Ft)`
`c.eq.d Fs,Ft`	Establece el código de condición si se cumple la condición "igual"	`CC = (Fs = Ft)`
`bc1f offset`	Salta a *offset* si el flag de código de condición es "falso" (*false*)	`Si CC=0 → Saltar`
`bc1t offset`	Salta a *offset* si el flag de código de condición es "cierto" (*true*)	`Si CC=1 → Saltar`

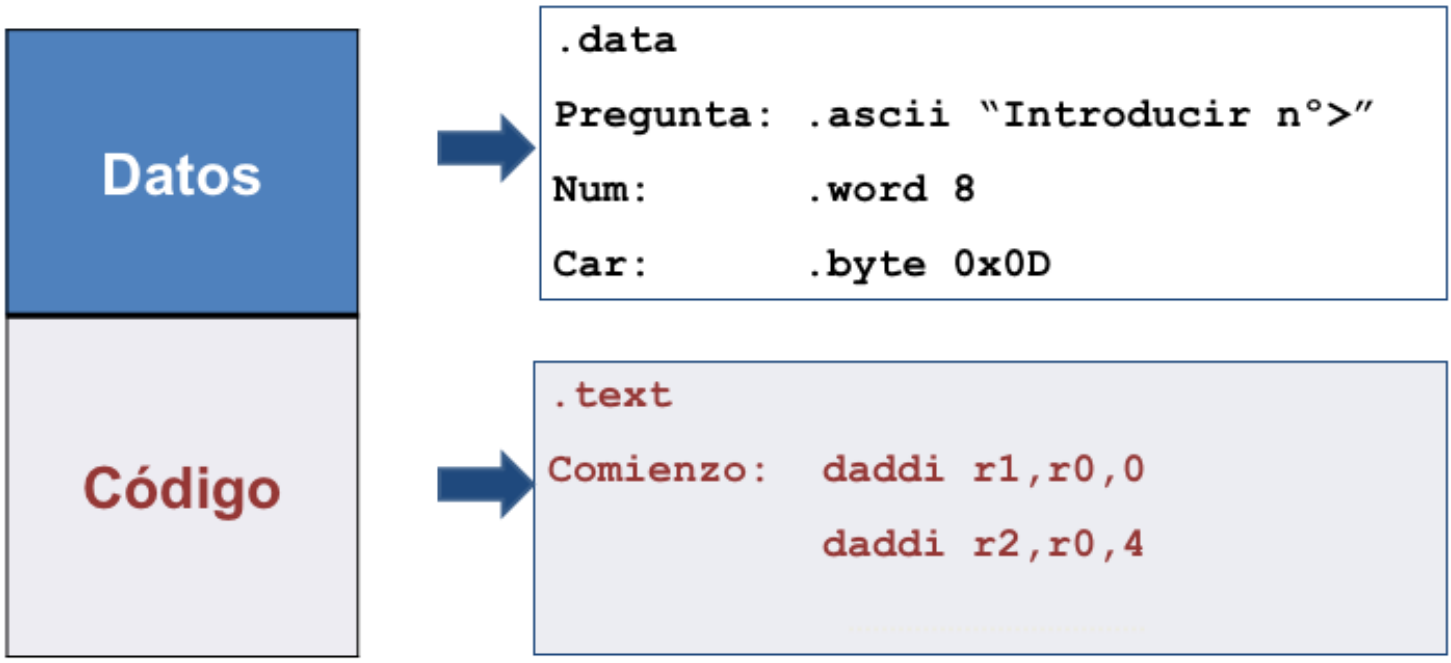
Cuando se desean efectuar saltos condicionales dependiendo de los valores de los registros de coma flotante (f0-f31), se deben realizar en dos pasos:

1. Comprobar la condición y establecer un valor (TRUE o FALSE) en el registro de código de condición de coma flotante (un registro del coprocesador matemático C1).

   Esta comprobación se realiza mediante las instrucciones **`c.COND.d`** donde **COND** puede ser **`lt`** (menor que, *lower than*), **`le`** (menor o igual, *less or equal*) o **`eq`** (igual, *equal*).

2. Bifurcar o saltar dependiendo de si el código de condición indica cierto o falso (TRUE, FALSE). las instrucciones encargadas de esto son **`bc1f`** (saltar si falso, *false*) y **`bc1t`** (saltar si cierto, *true*).

3. Ensamblador de MIPS64



Un programa del simulador **`winmips64`** consta de dos secciones bien diferenciadas: la sección de datos y la sección de código.

En la **sección de datos** se declaran las variables del programa. La declaración de variables consiste en reservar un espacio de memoria para alojarlas en una dirección de memoria principal y asignarlas un nombre simbólico para referenciarlas de una manera sencilla en la sección de código.

Para definir las variables, el ensamblador proporciona una serie de directivas que establecen las distintas características de cada variable.

Conviene que la declaración de cada variable comience en la columna 0.

La **sección de código** comienza tras la directiva "**`.text`** " o "**`.code`**". Las primeras columnas de la línea se reservan para situar una etiqueta como posible destino de una instrucción de salto.

La ejecución de un programa comienza por la primera instrucción de la sección de código.

Directiva	Descripción
`.text o .code`	Comienzo de código
`.data`	Comienzo de datos
`.space N`	Reserva `N` octetos de memoria
`.byte b1,b2,…`	Inicializa una zona de memoria con octetos
`.word16 w1,w2,…`	Inicializa una zona de memoria con medias palabras (16 bits)
`.word32 w1,w2,…`	Inicializa una zona de memoria con palabras (32 bits)
`.word n1,n2,…`	Inicializa una zona de memoria con dobles palabras (64 bits)
`.ascii STRING`	Inicializa una zona de memoria con `STRING`
`.asciiz STRING`	Idéntica a `.ascii` añadiendo un carácter NULL al final
`.double d1,d2,…`	Inicializa una zona de memoria con números en coma flotante
`.org DIR`	Dirección de comienzo de ensamblaje

Las **directivas** de ensamblaje no son instrucciones de nuestro programa, sino instrucciones al programa ensamblador (el que traduce nuestro programa a lenguaje máquina), para indicarle cómo debe realizar el proceso de traducción.

Las directivas más significativas que vamos a ver son las dedicadas a la reserva de espacio de memoria para ubicar variables de distintos tamaños (lo que en lenguajes de alto nivel se denomina “declaración de variables”).

La directiva `.space N` reserva N octetos de memoria sin especificar su valor inicial.

Las directivas `.byte`, `.word16`, `.word32` y `.word` reservan memoria, respectivamente, para un byte, 2 octetos, 4 octetos y una palabra de 8 bytes; y además les asigna el valor inicial que se especifica. Si en una misma línea se indican varios valores iniciales separados por comas, se reservan otras tantas unidades de memoria, con sus correspondientes valores iniciales.

La directiva `.ascii` reserva una zona de memoria de tantos bytes como caracteres se indiquen en el valor inicial encerrado entre comillas dobles.

`.asciiz` es similar a la directiva `.ascii`, pero además añade el valor binario cero (*null*) al final del string con el que inicializa la zona de memoria. Esto es útil para los mensajes de texto pues el cero indica el final del mensaje.

También hay directivas para declaración de variables de coma flotante: `.double`.

La directiva `.org` se utiliza para indicar la dirección de memoria donde se ubicarán las instrucciones o datos que le siguen a continuación.

Como veremos, en cada reserva de espacio para una variable, se puede añadir una **etiqueta** con el nombre con el que queremos referirnos a tal variable en el programa.

PROGRAMA

DATOS

```
; Este programa calcula la expresión X = Y + Z
        .data
X:      .word 0x0                   ; en Hexadecimal
Y:      .word 0x0123456789012345    ; en Hexadecimal
Z:      .word 10                    ; en Decimal
```

CÓDIGO

```
        .text
```

Etiquetas	*Instrucciones*	*Comentarios*
	`ld r1,Y(r0)`	; Carga en r1 el valor de la var. Y (64 bits)
	`ld r2,Z(r0)`	; Carga en r2 el valor de la var. Z (64 bits)
	`dadd r3,r1,r2`	; Realiza la suma de r1 y r2
	`sd r3,X(r0)`	; Almacena en la var. X el valor de r3 (64 bits)
	`halt`	; Para el procesador (fin del programa)

Como vemos en el ejemplo, **las etiquetas** de las variables se indican al comienzo de la línea (columna 1) de la declaración de la variable. Las etiquetas son nombres que se forman con las mismas reglas que los identificadores del lenguaje C, y terminan con el símbolo de los dos puntos":". Para referirse a una variable en la sección de código, simplemente hay que indicar el nombre de la variable, sin los dos puntos.

Los valores iniciales de las variables numéricas se pueden indicar en **distintas bases numéricas**. Así, si el valor contiene solamente dígitos numéricos, la base es decimal, y si el número comienza por "0x" el valor está expresado en hexadecimal.

En una línea, todo lo que haya a la derecha del símbolo ";" (punto y coma) es un **comentario**, es decir, que el programa ensamblador lo ignora cuando va traduciendo el programa, y solamente se indica a fin de ayudar al lector en la comprensión del programa.

La instrucción máquina **`halt`** se emplea para detener la ejecución del programa, por lo que debe utilizarse siempre como la última instrucción de un programa que tiene fin, es decir, que no consta de un bucle infinito.

Como se puede apreciar, **las directivas de ensamblaje comienzan por un punto**, mientras que las instrucciones máquina están formadas directamente por el nombre de la instrucción (sin el punto inicial). Tanto las directivas como las instrucciones máquina tienen que estar a partir de la columna 2, pues el comienzo en la columna 1 está reservado para las etiquetas.

Ejemplo 1

C

```
int main (){
  int i;
  double d;
  char c;
  int vector[5];
  short s;
}
```

MIPS64

```
.data

i:      .word32  0
d:      .double  0.0
c:      .byte    0
vector: .word32  0,0,0,0,0
s:      .word16  0
```

Ejemplo 2

C

```
int main (){
  int i;

  i = 10;
  i = i + 30;
}
```

MIPS64

```
        .data
i:      .word32  0
        .text
MAIN:   daddi R3,R0,10 ;R3=10
        daddi R3,R3,30 ;R3=R3+30
        sw    R3,i(R0)
        halt
```


En el **Ejemplo 1** tenemos un fragmento de un programa C en el que se declara una serie de variables, y a la derecha está su correspondencia en MIPS:

- Un entero (`int`) que, en C, normalmente ocupa 4 bytes.
- Una variable de coma flotante de doble precisión (8 bytes).
- Un carácter que ocupa un byte.
- Un vector de 5 enteros.
- Un entero “corto” de 16 bits.

En el **Ejemplo 2** tenemos un pequeño programa C en el que se declara una variable `i` de tipo entero, se asigna el valor 10 y, por último, se le suma el valor 30.

En la derecha se muestra su contrapartida en ensamblador MIPS, cuyo código comienza cargando el valor de la variable `i` en el registro `R3`, seguidamente le asigna el valor 10 mediante una suma en la que un sumando es cero y el otro es el valor inmediato10. A continuación le suma el valor 30 y, por último almacena el valor del registro `R3` en la variable `i`. El programa termina con la instrucción `HALT`.

C

```
int main (){
  int i;
  int j;
  int k;

  i = 10;
  j = 20;

  if (i<j)
     k = 1;
  else
     k = 2;
}
```

MIPS64

```
        .data
i:      .word32  0
j:      .word32  0
k:      .word32  0

        .text
MAIN:   daddi R3,R0,10   ;i R3=10
        sw    R3,i(R0)   ;i=R3
        daddi R4,R0,20   ;j R4=20
        sw    R4,j(R0)   ;j=R4
IF:     slt   R8,R3,R4
        beqz  R8,IFELSE
        daddi R10,R0,1   ;R10=1
        sw    R10,k(R0)  ;k=R10
        j     IFEND
IFELSE: daddi R12,R0,2   ;R12=2
        sw    R12,k(R0)  ;k=R12
IFEND:  halt
```


En este ejemplo veremos cómo se realiza en ensamblador una bifurcación condicional.

Ya que el programa C no requiere explicación de su cometido, comentaremos su implementación en ensamblador MIPS.

Obsérvese, en la sección de código, que no se dispone de instrucciones de movimiento o copia de datos (de tipo *move*) y que en su lugar se utilizan instrucciones de suma (`daddi`) en las que un operando es inmediato y el otro es cero (`R0`). A continuación deben almacenarse estos valores en las zonas de memoria correspondientes a las variables `i` y `j` mediante sentencias de almacenamiento `sw` (*store word*).

La bifurcación condicional se realiza mediante un par de instrucciones:

`slt R8,R3,R4` (*set on less than*) pone un 1 (cierto) en el registro R8 si el contenido de R3 es menor que R4. En caso contrario, pone un 0 (falso).

`beqz R8,IFELSE` (*branch on equal to zero*) bifurca a la instrucción con la etiqueta `IFELSE` si el contenido de R8 es igual a cero. En caso contrario, continúa la ejecución con la siguiente instrucción en secuencia (`daddi R10,R0,1`). Esta rama de la bifurcación finaliza con un salto incondicional a la instrucción etiquetada `IFEND`. Como se puede observar, después de la instrucción de salto (`j IFEND`) hay una instrucción `nop` (*no operation*), la cual es necesaria debido a que los saltos tienen un hueco de retardo, cuyo concepto abordaremos en un capítulo posterior.

Observación: Debido al modo de funcionamiento de MIPS, al ejecutar estos programas no se comportarán correctamente debido ciertos problemas que se producen en las instrucciones de salto o bifurcación, por lo que deberán ser retocados para conseguir el comportamiento correcto.

Ejemplos de código (3)

C

```
int main (){

 int i = 0;
 int j = 0;

 while ( i < 10){
    j = j + 5;
    i = i + 1;
 }
}
```

MIPS64

```
            .data
i:          .word32  0
j:          .word32  0

            .text
MAIN:       daddi R2,R0,0   ;i R2=0
            daddi R3,R0,0   ;j R3=0
            daddi R5,R0,10  ;R5=10
WHILE:      slt   R6,R2,R5
            beqz  R6,ENDWHILE
            daddi R3,R3,5   ;R3=R3+5
            daddi R2,R2,1   ;R2=R2+1
            j     WHILE
ENDWHILE:   sw    R3,j(R0)  ;j=R3
            sw    R2,i(R0)  ;i=R2
            halt
```


Ahora abordaremos la implementación de un bucle de tipo *while* con la ayuda de un programa en el que mientras la variable de control `i` sea menor que 10, se ejecuta un bucle en el que la variable `j` se incrementa de 5 en 5, y la variable de control, de 1 en 1.

En el ejemplo en ensamblador vemos que las instrucciones del bloque `MAIN` inicializan las variables `i` y `j` a cero, y dejan una copia de las mismas en `R2` y `R3`. Por último, en R5 se pone el valor final de la variable de control (10). (Téngase en cuenta que en este caso la etiqueta WHILE no es necesaria; aquí la indicamos para ayudar en la descripción del programa).

El bloque `WHILE` comienza con la instrucción `slt R6,R2,R5`, la cual establece el valor cierto (1) o falso (0) en `R6` en función de si `R2` es menor que `R5` o no. Así, la bifurcación condicional `beqz` salta a `ENDHILE` si la comparación fue falsa.

Las siguientes instrucciones se encargan de incrementar las variables `j` e `i` (en los registros `R3` y `R2`).

Este bloque finaliza con una instrucción `nop`, ya que el salto `j WHILE` tiene un hueco de retardo. No obstante, en este caso, tampoco importaría que se ejecutara la instrucción `sw R3,j(R0)`.

La instrucción `beqz` del bloque `WHILE` también tiene un hueco de retardo, por lo que siempre se ejecuta la instrucción `daddi R3,R3,R5` incluso en el caso en que la condición de bifurcación a `ENDWHILE` sea cierta; no obstante, en esta circunstancia, aunque se ejecute esta instrucción de suma, ya no afecta al resultado del programa.

En el bloque `ENDWHILE` se actualizan en memoria los valores de las variables `j` e `i` mediante instrucciones `sw` y, por último, con `halt` se detiene la ejecución del programa.

Ejemplos de código (4)

C

```
int main ()
{
    int vector[10];

  for (i=0;i<10;i++)
{
    vector[i]= i;
  }
}
```

MIPS64

```
        .data
vector: .word32  0,0,0,0,0,0,0,0,0,0

        .text
MAIN:   daddi R2,R0,10   ;tamaño
        daddi R4,R0,0    ;i en R4
        daddi R11,R0,4   ;incremento
        daddi R12,R0,0   ;ptr al vector
FOR:    slt   R7,R4,R2
        beqz  R7,END
        dmul  R12,R11,R4
        sw    R4,vector(R12)
        daddi R4,R4,1
        j     FOR
END:    halt
```


Este programa cuenta inicialmente con un vector de 10 enteros, a los que, mediante un bucle `for` se les asignan, respectivamente, los valores 1 a 10.

Ya que los enteros de C ocupan 4 bytes, en MIPS se declaran de tipo `WORD32`, y con un valor inicial cero.

En el bloque `MAIN` se asigna el valor 10 al tamaño (en `R2`); y 0, a la variable de control del bucle (en `R4`). A `R11` se le asigna el valor 4, para utilizarlo como una constante con el tamaño de un `integer`. `R12` se utilizará como el puntero que recorre el vector (desplazamiento de cada elemento, respecto a la dirección de comienzo del vector), y se inicializa a cero.

En las dos primeras instrucciones del bloque FOR se comprueba la condición de entrada en el bucle, y si no se cumple, se salta al final del programa en END.

En el cuerpo del bucle se multiplica `R11` por `R4`, dejando el resultado en `R12`; es decir, se calcula el desplazamiento del siguiente elemento del vector (`R12`) multiplicando el índice de control del bucle (`R4`) por el tamaño de cada elemento del vector (`R11`). Ya que no se dispone de instrucciones de multiplicación con valores inmediatos, hay que utilizar el registro `R11` como valor constante 4.

El bucle FOR se termina almacenando el valor del nuevo elemento del vector (`sw R4,vector(R12)`). `R4` es el valor a almacenar (que es el índice de control del bucle) y `vector (R12)` es la dirección donde se debe almacenar, siendo `R12` el desplazamiento respecto del comienzo del vector.

Ya que los saltos tienen un hueco de retardo, después del salto final del bucle, debe rellenarse un hueco con una instrucción `NOP`.

Ejemplos de código (5)

C

```
int factorial(int n){
  int i, fact = 1
  if (n>1)
    for (i=2;i<=n;i++)
      fact= fact*i;
  return fact;
}
int main(){
   int x=factorial(4);
}
```

MIPS64

```
FACTORIAL: daddi R2,R0,1    ;fact R2=1
IF:        daddi R10,R0,1   ;R10=1
           slt   R5,R10,R4  ;¿1<n?
           beqz  R5,END
           daddi R7,R0,2    ;i R7=2

FOR:       slt   R11,R7,R4
           slt   R12,R4,R7
           or    R13,R11,R12   } i <= n
           slti  R14,R13,1
           or    R14,R14,R11
           beqz  R14,END
           dmul  R2,R2,R7   ;R2=R2*R7
           daddi R7,R7,1    ;R7=R7+1
           j     FOR
ENDFOR:
END:       daddi R24,R2,0 ;valor retorno
           jr    R31    ; R31 dir retorno
```

MIPS64

```
           .data
x:         .word32  0

           .text
MAIN:    daddi R4,R0,4 ;n R4=4
         jal factorial ;R31
         nop ;hueco de retardo
         sw R24,x(R0)  ;x=R24
         halt
```


Este programa C calcula el factorial de un número N a base de multiplicar 2*3*4* …*N. Si N es igual a 1, se devuelve 1. En este caso, se le encarga calcular el factorial de 4.

En su equivalente en ensamblador MIPS, el bloque `MAIN` se encarga de pasar el parámetro n=4 a la subrutina `FACTORIAL` encargada de realizar el cálculo. Al llamar a la subrutina `FACTORIAL`, la dirección de retorno queda guardada en `R31`. Por último, almacena en `x` el resultado recogido en `R24`.

Ya en la rutina `FACTORIAL`, en el bloque `IF` se comprueba si n>1. Si no lo es, salta a `END`, donde se copia `R2` (que tiene el valor 1) a `R24`, para devolver éste último como resultado.

Si n>1, se entra en el bloque `FOR`, de tal manera que mientras n>1, se multiplica `R2` por `R7`, donde `R2` es el factorial parcial que se va calculando y `R7` va tomando los valores 2, 3, 4, …

Cuando no se cumple que n>1, se salta a `END`, donde se acaba poniendo el resultado en `R24` y se devuelve el control al programa principal cuya dirección de retorno había quedado guardada en `R31`.

Obsérvese la instrucción `nop` del bloque MAIN. Esta instrucción, cuyo único cometido es incrementar el Contador de Programa, es necesaria aquí para dar tiempo a que quede actualizado el valor en `R24`, al final del procedimiento `FACTORIAL`, antes de almacenarlo mediante la instrucción `sw`. En el siguiente capítulo veremos con detalle esta situación.

Terminamos el capítulo recordando que debido al modo de funcionamiento de MIPS, al ejecutar estos programas sobre un procesador MIPS, no se comportarían correctamente debido ciertos problemas que se producen en las instrucciones de salto o bifurcación, por lo que deberán ser retocados para conseguir el comportamiento correcto. Veremos esto en el siguiente capítulo dedicado a los procesadores segmentados o en *pipeline*.