ESTRUCTURAS DE CONTROL CONDICIONAL:

IF: esta estructura evalúa una condición y si se cumple, ejecuta un determinado código; en caso contrario, no hace nada.

```
1  int main(int argc, char** argv) {
2    int dato1=40;
3    int dato2=30;
4    int res;
5
6    res=0;
7    if (dato2!=0) {
8      res=dato1/dato2;
9    }
10 }
```

```
1           .data
2  dato1:   .word  40
3  dato2:   .word  30
4  res:     .space  4
5
6           .text
7  main:    lw    $t1, dato1($0)   # Carga dato1 en $t1
8           lw    $t2, dato2($0)   # Carga dato2 en $t2
9           and   $t0, $0, $0      # $t0 <- 0
10 si:      beq   $t2, $0, finsi   # Si $t2 = 0 salta a finsi
11 entonces: div  $t1, $t2         # $t1/$t2
12          mflo  $t0              # Almacena LO en $t0
13 finsi:   sw    $t0, res($0)     # Almacena $t0 en res
```

ESTRUCTURAS DE CONTROL REPETITIVAS:

WHILE: esta estructura permite la ejecución reiterada de una serie de instrucciones mientras se cumpla una determinada condición.

Estructura:
while (condición) {
cuerpo-bucle
}

Traducción 1: dejar condición intacta y saltar condicionalmente sobre el salto incondicional al final del bucle.

etiqueta-bucle:
evaluación condición utilizando después branch a etiqueta-cuerpo
j etiqueta-fin
etiqueta-cuerpo:
cuerpo-bucle
j etiqueta-bucle
etiqueta-fin:

Traducción 2: complementar condición y saltar condicionalmente al final del bucle.

etiqueta-bucle:
    evaluación condición complementada utilizando después branch a etiqueta-fin
    cuerpo-bucle
    j etiqueta-bucle
etiqueta-fin:

DO…WHILE: casi igual que la estructura anterior, con la diferencia de que el cuerpo del bucle se ejecuta al menos una vez.

Estructura:

do {
    cuerpo-bucle
} while (condición);

Traducción:

etiqueta-bucle:
    cuerpo-bucle
    evaluación condición saltando a etiqueta bucle

FOR: la estructura de control **for** se diferencia de la **while** en que el número de iteraciones es conocido de antemano. Así pues, es necesaria una variable que almacene el número de iteraciones que se van realizando para compararlo con el número de iteraciones deseado.

Estructura:

for (inicialización;condición;actualización) {
    cuerpo-bucle
}

Traducción: traducirlo previamente a un bucle while en pseudocódigo y después traducir dicho bucle while a ensamblador.

Paso 1.

inicialización
while (condición) {
    cuerpo-bucle
    actualización
}

GESTIÓN DE SUBRUTINAS:

Una subrutina no es más que un conjunto de instrucciones separadas del programa principal que realizan una determinada tarea y que puede ser invocada desde cualquier punto del programa.

Las subrutinas nos permiten estructurar un problema largo y complejo en subproblemas más sencillos, proporcionando una mayor facilidad a la hora de escribir, depurar y probar cada uno de los problemas por separado. De igual forma, si una tarea debe realizarse en varios puntos del programa, no es necesario replicar el código, sino que podemos hacer uso de las llamadas a estas subrutinas. Por último, subproblemas que suelan aparecer con frecuencia en el desarrollo de programas pueden ser implementados como subrutinas y agruparse en lo que llamamos bibliotecas (libraries) de forma que, cuando un programador quiera resolver un determinado problema y implementado, le basta con recurrir a una determinada biblioteca en invocar a la subrutina adecuada (reutilización de código).

Para que el uso de las subrutinas sea eficiente, un procesador debe proporcionar herramientas eficientes para facilitar las siguientes acciones: **llamada a una subrutina**; **paso de parámetros**; **devolución de resultados**; **continuación de la ejecución** del programa a partir de la siguiente instrucción a la que invocó a la subrutina.

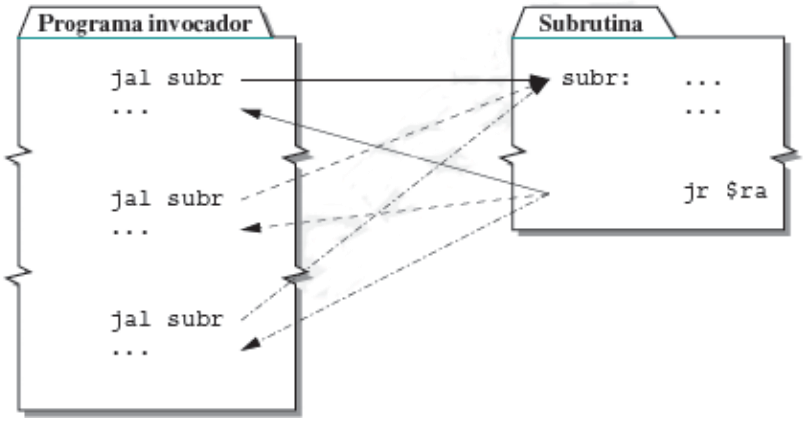
Llamada y retorno de una subrutina:

En el repertorio de instrucciones de MIPS32 disponemos de 2 instrucciones para gestionar la llamada y el retorno de una subrutina:

- **<< jal etiqueta >>** se utiliza el el programa invocador para llamar a la subrutina que comienza en la dirección de memoria indicada por la etiqueta. Como todos sabemos ya, la ejecución de la instrucción jal conlleva las siguientes acciones:
    - Almacenar la dirección de memoria de la siguiente instrucción a la que contiene a la instrucción jal en el registro $ra (return address): $ra ← PC+4.
    - Se lleva el control del flujo del programa a la dirección indicada en el campo etiqueta, o lo que es lo mismo, se realiza un salto incondicional a la dirección especificada por etiqueta: PC ← etiqueta.
- **<< jr $ra >>** se emplea en la subrutina para retornar al programa invocador. Esta instrucción realiza un salto incondicional a la dirección contenida en el registro $ra: PC ← $ra.

Así pues, si utilizamos correctamente estas dos instrucciones, podemos realizar de forma sencilla la llamada y retorno de desde una subrutina. El programa invocador debe llamar a la subrutina utilizando la instrucción jal. Esta instrucción almacena en $ra la

dirección de vuelta y salta a la dirección indicada por etiqueta. Por último, cuando finalice la subrutina, ésta debe ejecutar la instrucción jr para retornar al programa que lo invocó. **Este mecanismo funcionará siempre que no se altere el contenido del registro $ra durante la ejecución de la subrutina.**



Tras haber presentado como realizar la gestión básica de las llamadas a subrutinas, debemos seguir describiendo como se gestionan otros aspectos más complejos relacionados con los registros y la memoria de la máquina en el caso de las llamadas a subrutina. Podemos hablar de la gestión de la información requerida por la subrutina, diferenciando las siguientes tareas:

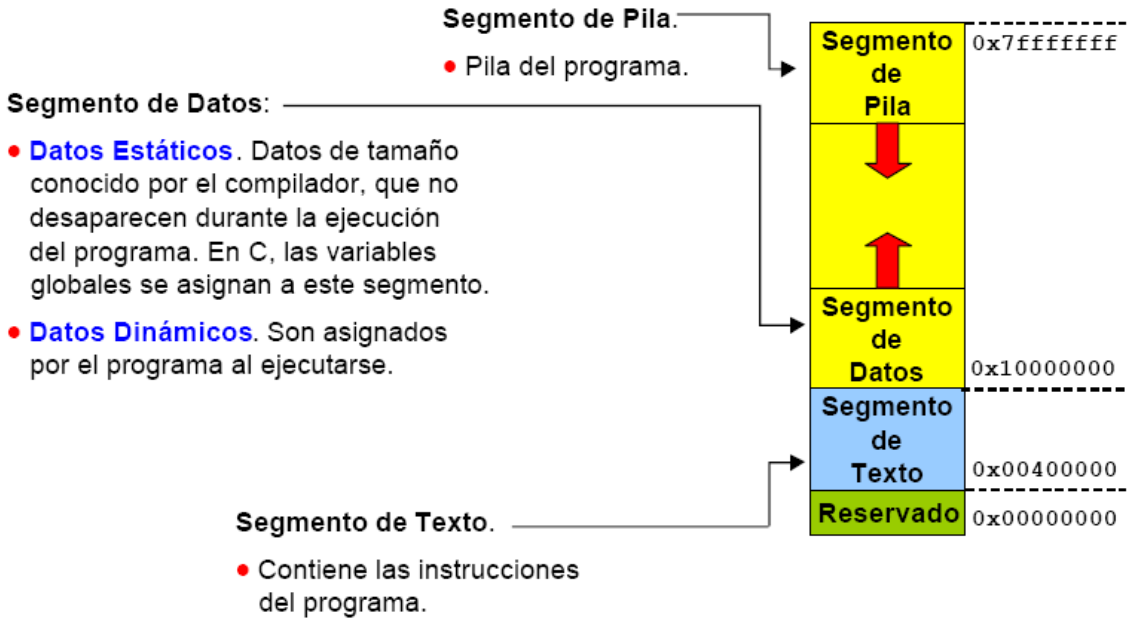
- Almacenamiento y posterior recuperación de la información almacenada en determinados registros.
- El almacenamiento y recuperación de la dirección de retorno para permitir a una subrutina llamar a otras subrutinas o a sí misma (recursividad).
- La creación y utilización de variables temporales y locales de la subrutina.

Para ello es necesario crear y gestionar un espacio de memoria donde la subrutina pueda almacenar la información necesaria durante su ejecución: **bloque de activación de la subrutina.**

En esta sesión, describiremos como se estructura la memoria en los sistemas MIPS y después introduciremos unas primeras nociones sobre como gestionar el segmento de pila. Dejaremos para sesiones posteriores la gestión completa del bloque de activación de la subrutina.

GESTIÓN DE MEMORIA:

Por regla general, los sistemas MIPS dividen la memoria en 3 partes.



La primera parte (la más próxima al extremo inferior del espacio de direcciones) que comienza en la dirección $400000_{)hex}$ es el **segmento de texto** y es el encargado de almacenar las instrucciones del programa.

La segunda parte, sobre el segmento de texto, es el **segmento de datos**, que se divide a su vez en dos partes:

Datos estáticos: comienza en la dirección $1000000_{)hex}$, y contiene los objetos cuyos tamaños son conocidos a priori por el compilador y cuyo tiempo de vida (intervalo durante el cual un programa puede accederlos) es el tiempo de ejecución completo del programa en cuestión. *Ej.: en C, las variables globales se almacenan estáticamente, puesto que pueden ser referenciadas en cualquier momento a lo largo de la ejecución del programa.*

Datos dinámicos: se encuentra inmediatamente por encima del segmento de datos estáticos. Tal y como su nombre indica, este segmento es creado por el programa durante su ejecución. Se utiliza para satisfacer las demandas de memoria realizadas por el compilador que no pueden ser predichas de antemano. Tal y como indica la figura anterior, la memoria dinámica se expande hacia arriba, añadiendo más páginas al espacio de direcciones virtual del programa. *Ej.: el programas escritos en C, la función malloc busca y encuentra un nuevo bloque de memoria libre (reservas de memoria en tiempo de ejecución).*

La tercera parte de la memoria se corresponde con el **segmento de pila**, que reside en la parte superior del espacio de direccionamiento del programa (comenzando en la dirección $7fffffff_{)hex}$). Al igual que en caso del segmento de datos dinámico, el tamaño máximo del segmento de pila de un programa no se puede conocer a priori por lo que,

a medida que el programa va apilando valores en este espacio, el sistema operativo va expandiendo el segmento de pila hacia abajo, contra el segmento de datos.

Gestión del segmento de pila: una pila o cola LIFO (Last in First Out) es una estructura de datos en la que el último dato almacenado es el primero en ser recuperado. Se caracteriza por dos operaciones principales: apilar (push) y desapilar (pop), de forma que no puede accederse de forma aleatoria a los elementos intermedios de la pila, sino que solo puede accederse al último elemento de la misma (tope o cima de la pila).

Para implementar una pila en la memoria de una máquina, lo primero es decidir a partir de que dirección va a crecer la pila y, además, disponer de un puntero que indique la dirección de memoria en la que se encuentra el último dato introducido (tope de la pila). Como ya hemos comentado, la pila sigue el convenio de que crece de direcciones de memoria altas a direcciones de memoria bajas, por lo que el tope de la pila decrecerá al añadir elementos a la pila y aumentará al eliminarlos. Ese tope de la pila se implementa por MIPS en forma del puntero de pila (stack pointer), almacenado en el registro $29, que también es conocido como $sp (stack pointer).

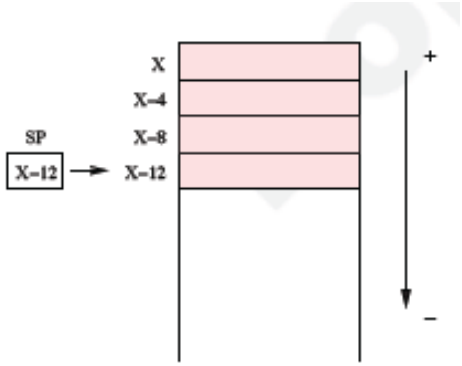
Así pues, para **apilar** un elemento, será necesario realizar los siguientes pasos:

- Decrementar el puntero de pila en tantas posiciones como el tamaño de los datos que se desean apilar.

- Almacenar el dato que se quiere apilar en la dirección indicada por el puntero de pila.

Ejemplo: Imaginemos que queremos apilar el contenido de $t0. La situación original la memoria puede verse en la siguiente figura:

Las instrucciones a ejecutar para apilar $t0 serían las siguientes:

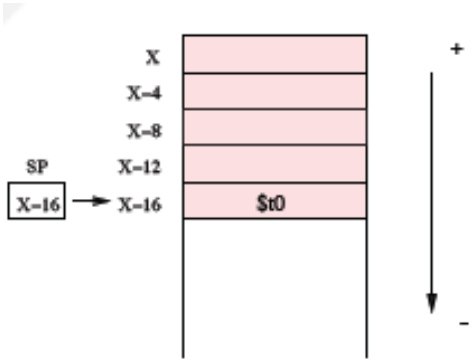


Antes de apilar el registro $t0

addi $sp, $sp, -4 (o también subi $sp, $sp, 4) → decrementamos el puntero de pila en 4 posiciones, ya que $t0 tiene un tamaño de 4 bytes.

sw $t0, 0($sp) almacena el contenido de $t0 en la dirección indicada por $sp.

Tras la ejecución de estas dos instrucciones, el estado de la memoria es el siguiente:



Después de apilar el registro $t0

La operación de desapilar un elemento también consta de dos pasos:

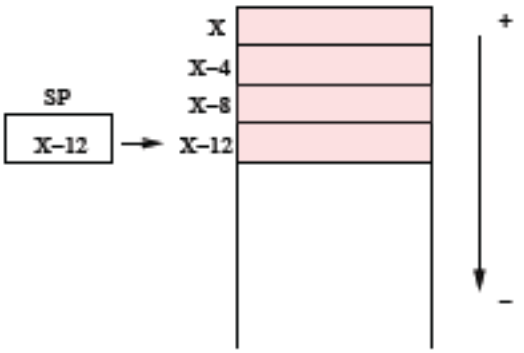
- Primero debemos recuperar el dato que se encuentra en la cima de la pila.
- Después, es necesario incrementar el puntero de pila tantas posiciones como el tamaño del dato que se desea desapilar.

Ejemplo: partiendo de la situación final del ejemplo anterior, si quisiéramos desapilar una palabra y guardarla en el registro $t0, los pasos a realizar serían los siguientes:

lw $t0, 0($sp) cargamos el dato que se encuentra en la posición indicada por $sp en el registro $t0

addi $sp, $sp, 4 incrementamos el puntero de pila en 4 posiciones, ya que $t0 tiene un tamaño de 4 bytes.

Tras la ejecución de las dos instrucciones anteriores, el estado de la pila es el siguiente:



Después de desapilar el registro $t0

Este sencillo mecanismo permite, entre otras cosas, realizar llamadas anidadas a subrutinas almacenando en la pila del procedimiento el valor del registro $ra.

En esta sesión, continuamos con la gestión de la memoria en las llamadas a subrutinas. En la sesión anterior, aprendimos a apilar y desapilar parámetros en la pila de memoria.

Ahora, nos centraremos en que información hace falta almacenar en dicha pila, es decir, en la creación del bloque de activación de la subrutina.

Como ya hemos mencionado, el bloque de activación de la subrutina está formado por el segmento de pila que contiene la información manejada por una subrutina durante su ejecución. Dicho bloque tiene varios propósitos:

- En el caso de llamadas anidadas, almacenar la dirección de retorno (algo que ya hemos trabajado).
- Proporcionar espacio para las variables locales de la subrutina.
- Almacenar los registros que la subrutina necesita modificar pero que el programa que ha hecho la llamada no espera que sean modificados.
- Mantener los valores que se han pasado como argumentos de la subrutina.

Variables locales de la subrutina:

Las subrutinas pueden requerir para su ejecución el uso de variables locales que sólo existirán mientras se están ejecutando dichas subrutinas. Dependiendo del tipo de variables, bastará con utilizar los registros no ocupados (caso de tipos escalares), o será necesario el uso de la memoria principal (caso de datos estructurados). Si es necesario utilizar la memoria principal, la variable deberá almacenarse en el bloque de activación de la subrutina. Para ello, al inicio de la subrutina tendremos que reservar el espacio necesario para almacenar dichas variables (debe estimarse un número de palabras necesario a priori), y antes del retorno se deberá liberar dicho espacio.

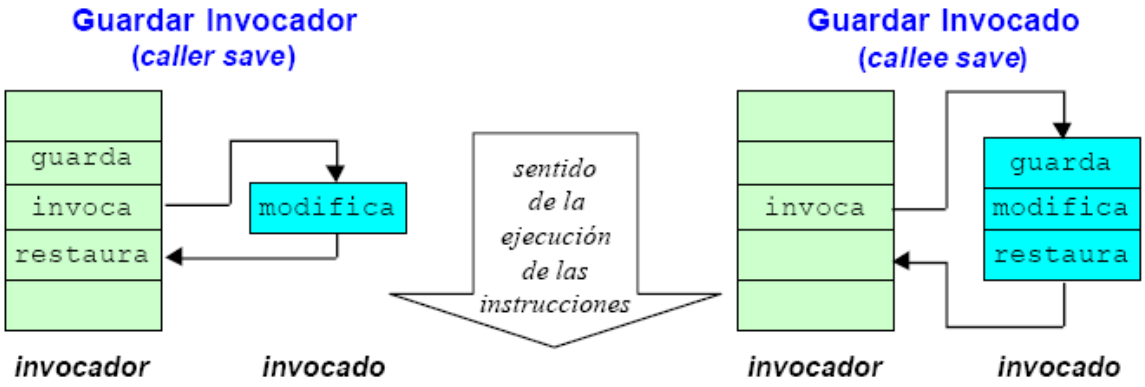
Almacenamiento de los registros utilizados por la subrutina:

Tal y como hemos comentado, las subrutinas pueden tener que utilizar los registros como variables temporales, y por tanto, el contenido original de los registros puede perderse en un momento dado. Si la información que contenían dichos registros era relevante para que el programa invocador pueda continuar su ejecución tras el retorno, será necesario almacenar temporalmente dicha información en el bloque de activación de la subrutina.

Para ello, la subrutina, antes de modificar el contenido de los registros, los apila en el bloque de activación. Una vez finalizada la ejecución de la subrutina, y justo antes del retorno, los recupera. Deberán ser apilados/desapilados todos aquellos registros que la subrutina vaya a modificar. Para disminuir el número de registros que deben ser guardados y restaurados, MIPS establece el siguiente convenio:

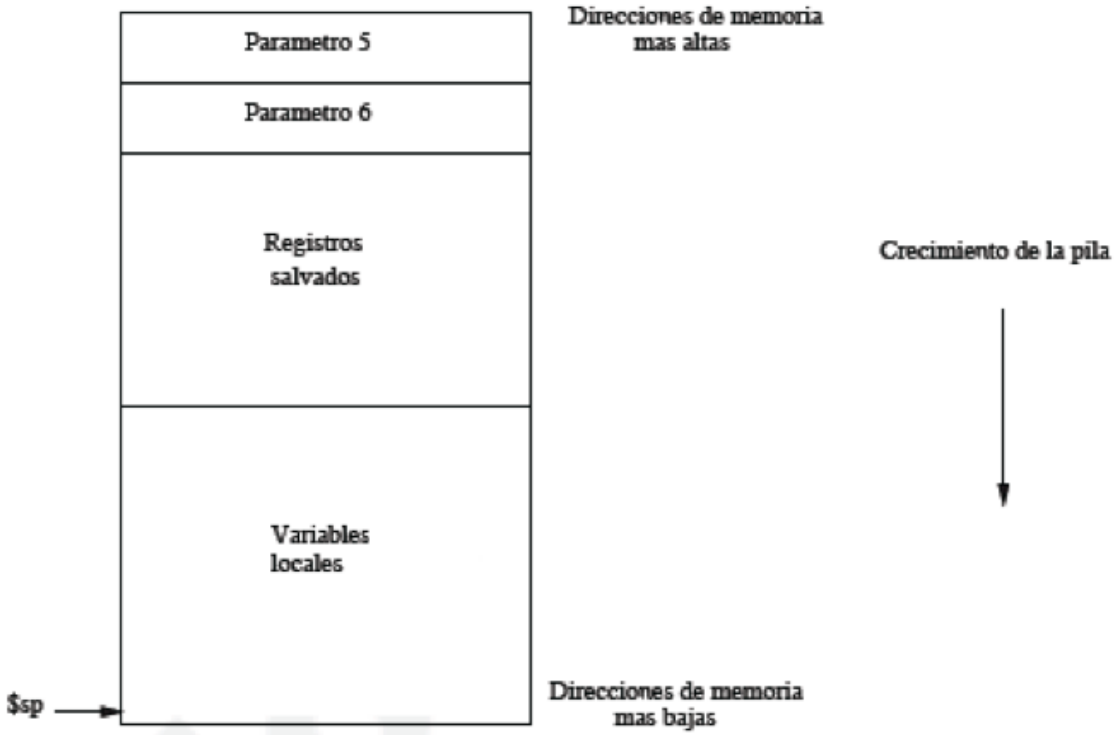
- Los registros **$t0 a $t9** pueden ser utilizados por el programa invocado (subrutina) para almacenar datos temporales sin que sea necesario guardar sus valores originales. Se supone que la información que contienen no es relevante para el programa que realiza la llamada. En el caso de que lo fuera, es el programa invocador el que debe apilarlos antes de hacer la llamada y recuperarlos tras el retorno **(GUARDAR INVOCADOR).**

- Los registros **$s0 a $s7** también pueden ser utilizados por el programa invocado para almacenar datos temporales, sin embargo, estos registros si deben ser almacenados por el programa invocado para posteriormente recuperar su valor original **(GUARDAR INVOCADO).**



En cuanto a que registros puede modificar o no la subrutina, debemos considerar también los registros **$a0 a $a3**, utilizados para el paso de parámetros y que, generalmente, deben ser preservados, por lo que si van a ser modificados, debeos almacenarlos también en el bloque de activación.

Como ya hemos visto, el bloque de activación de una subrutina está ubicado en memoria y se implementa mediante una estructura tipo pila. El bloque de activación visto hasta ahora se muestra en la siguiente figura:



Un aspecto que influye definitivamente en la eficiencia de los programas que manejan subrutinas y sus respectivos bloques de activación es el modo en el que se accede a la información contenida en los respectivos bloques de activación (el acceso a la memoria en un programa siempre resulta un aspecto clave de cara a la eficiencia del mismo).

El modo más sencillo de acceder un dato en un bloque de activación es utilizando el **modo de direccionamiento indexado**. Mediante este modo, la dirección de un dato se calcula sumando una **dirección base y un desplazamiento**. Como dirección base se puede utilizar el contenido del puntero de pila, que apunta a la posición más baja del bloque de activación. El desplazamiento sería entonces la posición relativa del dato con respecto al puntero de pila. Así pues, si sumamos el contenido del registro **$sp** y un determinado desplazamiento, obtendríamos la dirección de memoria de cualquier dato que se encontrara en el bloque de activación. Como ya hemos visto en sesiones anteriores, si hemos apilado una palabra en la posición 8 por encima del $sp, podríamos recuperar su valor utilizando la instrucción "**lw** $t0, 8($sp)". Algunos de vosotros ya habréis detectado que utilizar el registro $sp como registro base para calcular la dirección de los datos tiene un inconveniente: *el contenido del registro $sp puede variar durante la ejecución de la subrutina, por lo que habría que variar de manera acorde el desplazamiento necesario para acceder a cada uno de los datos.*

Para conseguir que el desplazamiento de cada uno de los datos permanezca constante durante la ejecución de la subrutina, es necesario disponer de una **referencia fija al bloque de activación**. Dicha referencia puede almacenarse en un registro específico que deberá ser utilizado como dirección base para el cálculo de las direcciones en lugar de $sp.

En ensamblador del MIPS32 existe un registro que cumple esta función**: $fp (frame pointer o puntero de marco), que se corresponde con el registro $30 del banco de registros.**

Por convenio, se utiliza como referencia fija al bloque de activación la **dirección de memoria que indica el registro $sp cuando se entra en la subrutina menos 4.** Así pues, cuando se requiera acceder a algún **dato apilado por la subrutina**, se utilizará un **desplazamiento negativo** respecto a dicha referencia. Si lo que queremos es acceder a algún **dato apilado por el programa invocador** justo antes de llamar a la subrutina, se utilizará un **desplazamiento positivo** respecto a dicha referencia.

Dado que $fp mantiene una referencia fija al bloque de activación de la subrutina en ejecución, **el valor de dicho registro debe ser preservado entre llamadas a subrutinas**. Esto significa que el valor previo de $fp debe ser apilado por la subrutina para poder restaurarlo antes de devolver el control al programa invocador.

Convenio para llamada a subrutinas:

En las sesiones dedicadas a la gestión de subrutinas (I a IV) hemos desglosado las diferentes acciones necesarias para respetar el convenio de llamada a subrutinas establecido en el ensamblador MIPS32. A continuación haremos un resumen con el

objetivo final de recapitular todas las acciones necesarias para invocar correctamente a una subrutina:

Tanto el programa invocador como el invocado intervienen en la creación y gestión del bloque de activación de una subrutina. La gestión del bloque se produce, fundamentalmente en los siguientes momentos:

- Justo antes de que el programa invocador pase el control a la subrutina.
- En el momento en el que la subrutina toma el control.
- Justo antes de que la subrutina devuelva el control al programa invocador.
- En el momento en el que el programa invocador recupera el control.

Para gestionar correctamente el bloque de activación, habrá que realizar las siguientes acciones en los 4 momentos descritos anteriormente:

Justo antes de que el programa invocador pase el control a la subrutina:

1. Almacenar en la pila los registros que deben ser salvados por el invocador. Deberá apilar cualquiera de los $a0 a $a3 y $v0 a $v1 que necesite después. Además, según el convenio, si al invocador le interesa preservar el contenido de algunos de los registros $t0 a $t9, éstos deben ser apilados antes de hacer la llamada.

2. Paso de parámetros. Cargar los parámetros en los lugares establecidos. Los cuatro primeros en registros $a0 a $a3, y los restantes se apilan en el bloque de activación. $a0 a $a3 no se apilan si no es necesario, pero se reserva espacio para ello.

En el momento en el que la subrutina toma el control:

1. Reservar memoria en la pila para el resto del bloque de activación. El tamaño se calculará sumando el espacio (en bytes) que ocupa el contenido de los registros que vaya a apilar la subrutina ($fp, $s0 a $s7, $ra) más el espacio que ocupan las variables locales que se vayan a almacenar en el bloque de activación.

2. Almacenar el contenido del registro $fp en la dirección más alta del espacio reservado en el bloque de activación.

3. Actualizar el puntero de bloque de activación (registro $fp) para que apunte a la dirección más alta del espacio reservado (la misma en la que se ha guardado su valor previo).

4. Almacenar en el bloque de activación aquellos registros que vaya a modificar la subrutina.

Justo antes de que la subrutina devuelva el control al programa invocador:

1. Cargar el valor (o valores) que deba devolver la subrutina en los registros $v0 y $v1.

2. Desapilar los registros apilados por la subrutina, excepto el registro $fp.

3. Copiar el contenido del registro $fp en el registro $sp (con lo que liberamos el espacio de pila ocupado por la subrutina para las variables locales).

4. Desapilar el registro $fp.

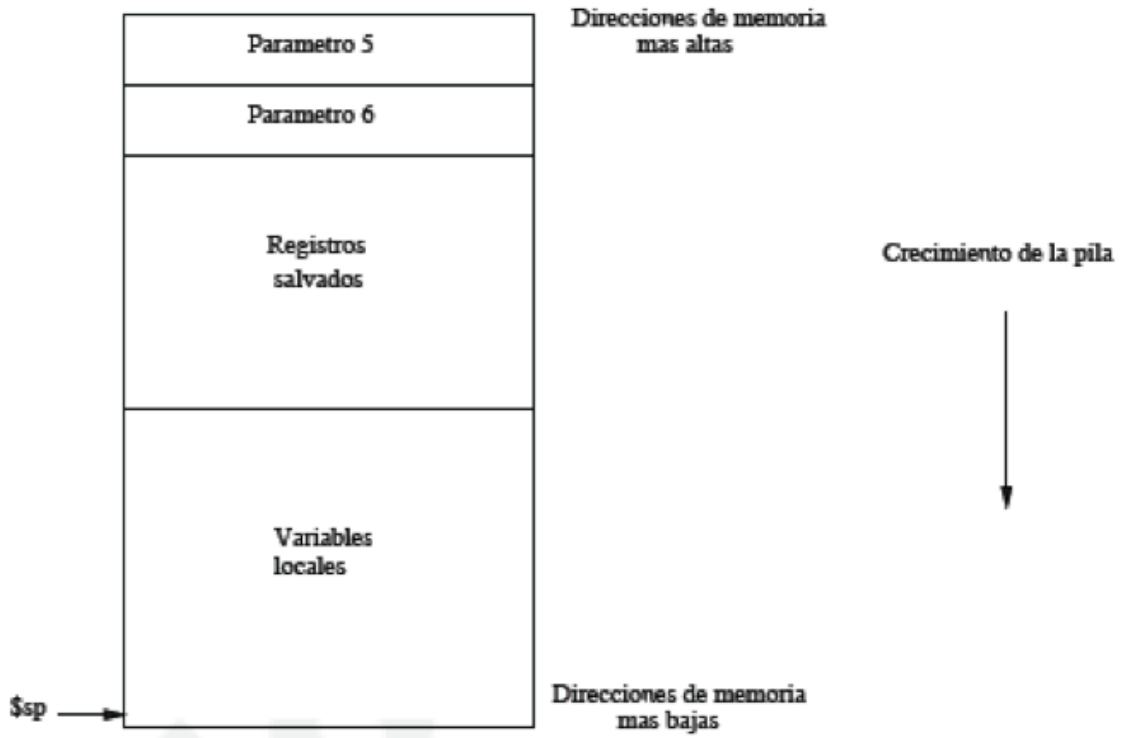
En el momento en el que el programa invocador recupera el control.

1. Eliminar del bloque de activación los parámetros que hubiera apilado.

2. Desapilar los registros que había apilado.

3. Recoger los parámetros devueltos.

Como ya hemos visto, el bloque de activación de una subrutina está ubicado en memoria y se implementa mediante una estructura tipo pila. El bloque de activación visto hasta ahora se muestra en la siguiente figura:



Un aspecto que influye definitivamente en la eficiencia de los programas que manejan subrutinas y sus respectivos bloques de activación es el modo en el que se accede a la información contenida en los respectivos bloques de activación (el acceso a la memoria en un programa siempre resulta un aspecto clave de cara a la eficiencia del mismo).

El modo más sencillo de acceder un dato en un bloque de activación es utilizando el **modo de direccionamiento indexado**. Mediante este modo, la dirección de un dato se calcula sumando una **dirección base y un desplazamiento**. Como dirección base se puede utilizar el contenido del puntero de pila, que apunta a la posición más baja del bloque de activación. El desplazamiento sería entonces la posición relativa del dato con respecto al puntero de pila. Así pues, si sumamos el contenido del registro **$sp** y un determinado desplazamiento, obtendríamos la dirección de memoria de cualquier dato que se encontrara en el bloque de activación. Como ya hemos visto en sesiones anteriores, si hemos apilado una palabra en la posición 8 por encima del $sp, podríamos recuperar su valor utilizando la instrucción "**lw** $t0, 8($sp)". Algunos de vosotros ya habréis detectado que utilizar el registro $sp como registro base para calcular la dirección de los datos tiene un inconveniente: *el contenido del registro $sp puede variar durante la ejecución de la subrutina, por lo que habría que variar de manera acorde el desplazamiento necesario para acceder a cada uno de los datos.*

Para conseguir que el desplazamiento de cada uno de los datos permanezca constante durante la ejecución de la subrutina, es necesario disponer de una **referencia fija al**

bloque de activación. Dicha referencia puede almacenarse en un registro específico que deberá ser utilizado como dirección base para el cálculo de las direcciones en lugar de $sp.

En ensamblador del MIPS32 existe un registro que cumple esta función**: $fp (frame pointer o puntero de marco), que se corresponde con el registro $30 del banco de registros.**

Por convenio, se utiliza como referencia fija al bloque de activación la **dirección de memoria que indica el registro $sp cuando se entra en la subrutina menos 4.** Así pues, cuando se requiera acceder a algún **dato apilado por la subrutina**, se utilizará un **desplazamiento negativo** respecto a dicha referencia. Si lo que queremos es acceder a algún **dato apilado por el programa invocador** justo antes de llamar a la subrutina, se utilizará un **desplazamiento positivo** respecto a dicha referencia.

Dado que $fp mantiene una referencia fija al bloque de activación de la subrutina en ejecución, **el valor de dicho registro debe ser preservado entre llamadas a subrutinas**. Esto significa que el valor previo de $fp debe ser apilado por la subrutina para poder restaurarlo antes de devolver el control al programa invocador.

Convenio para llamada a subrutinas:

En las sesiones dedicadas a la gestión de subrutinas (I a IV) hemos desglosado las diferentes acciones necesarias para respetar el convenio de llamada a subrutinas establecido en el ensamblador MIPS32. A continuación haremos un resumen con el objetivo final de **recapitular todas las acciones necesarias para invocar correctamente a una subrutina**:

Tanto el programa invocador como el invocado intervienen en la creación y gestión del bloque de activación de una subrutina. La gestión del bloque se produce, fundamentalmente en los siguientes momentos:

- Justo antes de que el programa invocador pase el control a la subrutina.
- En el momento en el que la subrutina toma el control.
- Justo antes de que la subrutina devuelva el control al programa invocador.
- En el momento en el que el programa invocador recupera el control.

Para gestionar correctamente el bloque de activación, habrá que realizar las siguientes acciones en los 4 momentos descritos anteriormente:

Justo antes de que el programa invocador pase el control a la subrutina:

1. Almacenar en la pila los registros que deben ser salvados por el invocador. Deberá apilar cualquiera de los $a0 a $a3 y $v0 a $v1 que

necesite después. Además, según el convenio, si al invocador le interesa preservar el contenido de algunos de los registros $t0 a $t9, éstos deben ser apilados antes de hacer la llamada.

2. Paso de parámetros. Cargar los parámetros en los lugares establecidos. Los cuatro primeros en registros $a0 a $a3, y los restantes se apilan en el bloque de activación. $a0 a $a3 no se apilan si no es necesario, pero se reserva espacio para ello.

En el momento en el que la subrutina toma el control:

1. Reservar memoria en la pila para el resto del bloque de activación. El tamaño se calculará sumando el espacio (en bytes) que ocupa el contenido de los registros que vaya a apilar la subrutina ($fp, $s0 a $s7, $ra) más el espacio que ocupan las variables locales que se vayan a almacenar en el bloque de activación.

2. Almacenar el contenido del registro $fp en la dirección más alta del espacio reservado en el bloque de activación.

3. Actualizar el puntero de bloque de activación (registro $fp) para que apunte a la dirección más alta del espacio reservado (la misma en la que se ha guardado su valor previo).

4. Almacenar en el bloque de activación aquellos registros que vaya a modificar la subrutina.

Justo antes de que la subrutina devuelva el control al programa invocador:

1. Cargar el valor (o valores) que deba devolver la subrutina en los registros $v0 y $v1.

2. Desapilar los registros apilados por la subrutina, excepto el registro $fp.

3. Copiar el contenido del registro $fp en el registro $sp (con lo que liberamos el espacio de pila ocupado por la subrutina para las variables locales).

4. Desapilar el registro $fp.

En el momento en el que el programa invocador recupera el control.

1. Eliminar del bloque de activación los parámetros que hubiera apilado.

2. Desapilar los registros que había apilado.

3. Recoger los parámetros devueltos.

Registros

Nombre	Número	Uso	Preservado en llamada
$zero	$0	constante entera 0	**sí**
$at	$1	temporal del ensamblador	no
$v0–$v1	$2–$3	Valores de retorno de funciones y evaluación de expresiones	no
$a0–$a3	$4–$7	Argumentos de funciones	no
$t0–$t7	$8–$15	Temporales	no
$s0–$s7	$16–$23	Temporales salvados	**sí**
$t8–$t9	$24–$25	Temporales	no
$k0–$k1	$26–$27	Reservados para el núcleo del SO	no
$gp	$28	puntero global	**sí**
$sp	$29	puntero de pila	**sí**
$fp	$30	puntero de "frame"	**sí**
$ra	$31	dirección de retorno	no

no → el invocador es el responsable si → el invocado es el responsable

RECURSIVIDAD:

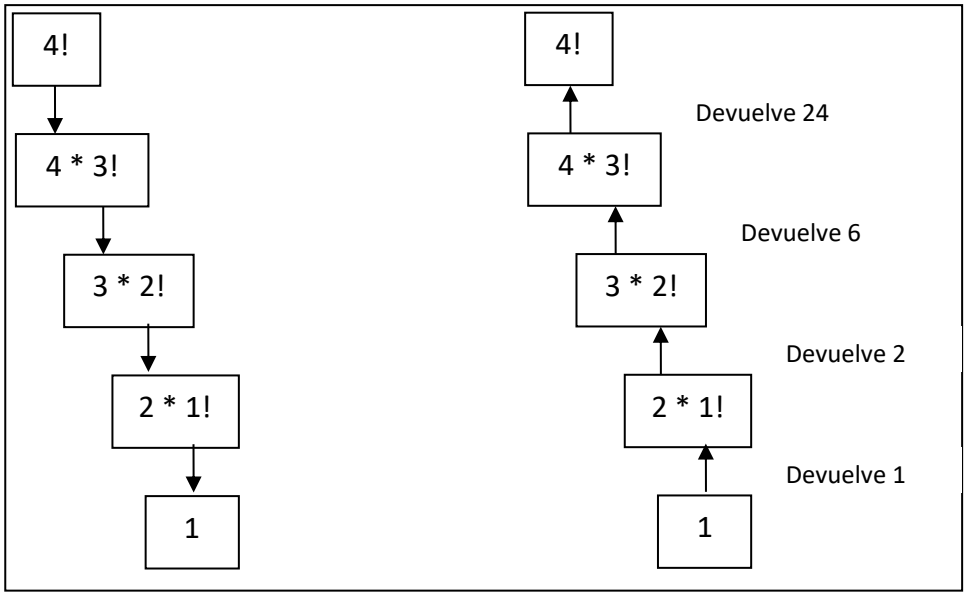
Una función recursiva es una función que se llama a sí misma, ya sea dentro de su propia definición (recursividad directa) o llamando a otra función que a su vez contenga una llamada directa o indirecta a la primera (recursividad indirecta). La recursividad resulta muy útil para programar problemas o estructuras, como los árboles, definidos de modo recursivo.

Para resolver un determinado problema, una función recursiva sólo proporciona directamente la solución para el caso más simple (caso base). Si la función es llamada con un caso más complejo, lo resolverá en términos de otro caso más simple del mismo problema. La función se llamará a sí misma con casos cada vez más sencillos, hasta llegar al caso base. Como la solución a este último está programada directamente, una vez que se conoce permite llegar hasta la solución del problema global subiendo de nuevo en la jerarquía.

Como ejemplo, una definición recursiva del factorial sería:

$$n! = \begin{cases} n \cdot (n-1)!, & \forall n > 0 \\ 1, & n = 0 \end{cases}$$

Para evaluar el factorial de 4, la secuencia de llamadas recursivas (izquierda) y de valores devueltos (derecha) sería:



Cuando se llama a un procedimiento o función recursivo los parámetros y las variables locales toman nuevos valores y el procedimiento o función trabaja con estos nuevos valores y no con los de las anteriores llamadas. Cada vez que se hace una llamada a un procedimiento o función los parámetros de entrada y variables locales son almacenados en memoria y cuando termina la ejecución del procedimiento o función son accedidos en orden inverso a como se introdujeron.

El espacio requerido para almacenar los valores crece pues conforme a los niveles de anidamiento de las llamadas.

El cuerpo del procedimiento o función debe disponer de una o varias instrucciones selectivas donde establecer la condición de salida.

Todo algoritmo recursivo puede ser implementado en forma iterativa utilizando una pila.

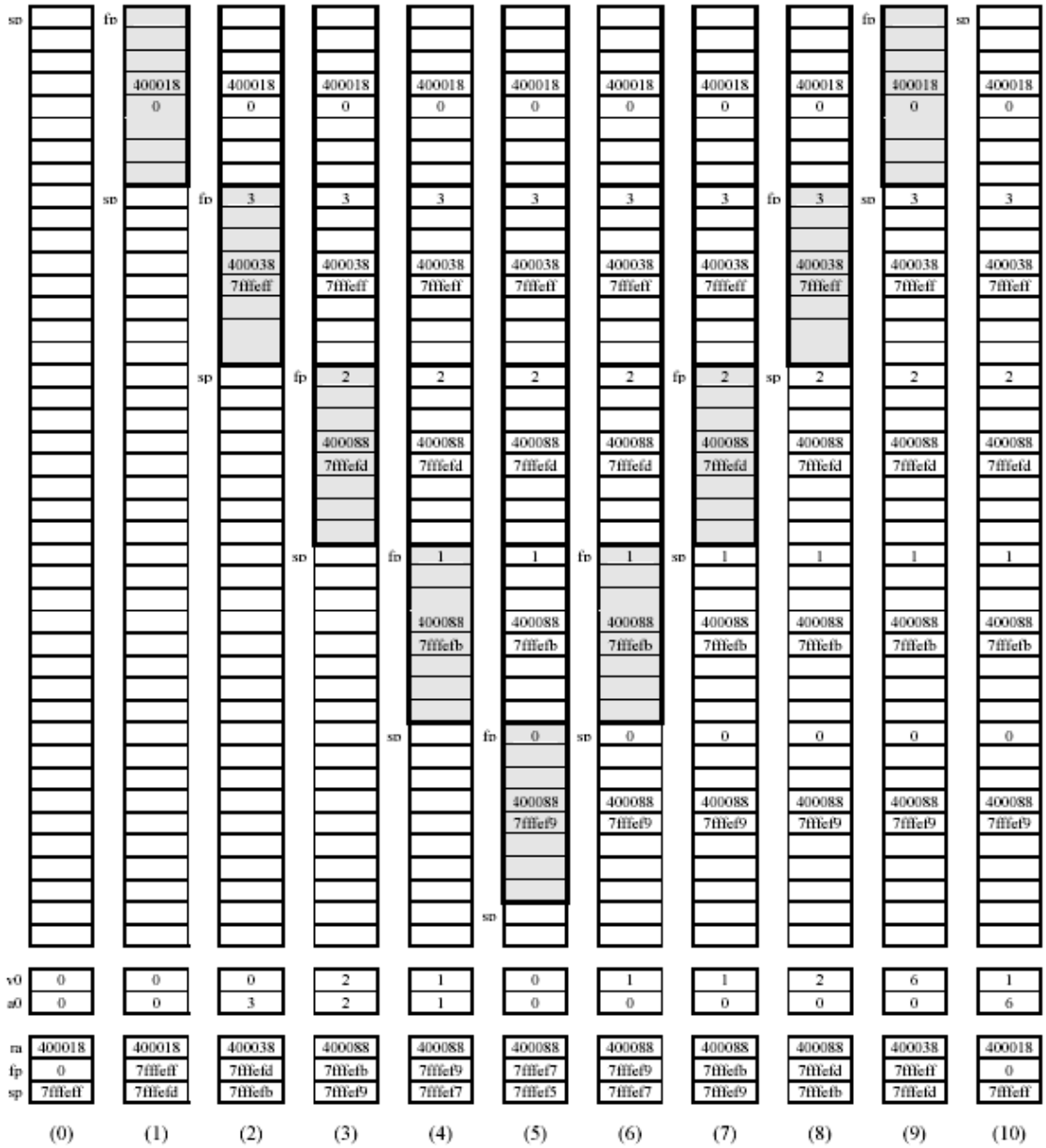
El siguiente programa en lenguaje ensamblador del MIPS calcula el factorial de un número n, utilizando un procedimiento de tipo recursivo llamado fact:

```
# Prácticas ensamblador MIPS
# UEX 2022/2023

            .text
            .globl main

main:       #----------------------------# La rutina main crea su marco de pila
                                         # (de 32 bytes) en memoria.
            # (0)

            subu    $sp, $sp, 32         # Nuevo stack pointer (puntero de pila): $sp <-- $sp+32
            sw      $ra, 20($sp)         # M[$sp+20] <-- $ra. Guarda direcc. de vuelta
            sw      $fp, 16($sp)         # M[$sp+16] <-- $fp. Guarda $fp antiguo
            addu    $fp, $sp, 32         # Nuevo frame pointer (puntero de estructura):
                                         # $fp <-- $sp+32 (donde estaba sp antes)
            # (1)

            li      $a0, 3               # Pone argumento (n=3) en $a0
            jal     fact                 # Llama fact, almacena en $ra dir. sig. instrucc.

            move    $a0, $v0             # Resultado está en $v0.
            li      $v0, 1               # Se escribe en consola
            syscall

            lw      $ra, 20($sp)         # Restaura registros
            lw      $fp, 16($sp)
            addu    $sp, $sp, 32

            # (10)

            j       $ra                  # Finaliza

#-----------------------------------------# Rutina fact: cálculo de factorial
fact:       subu    $sp, $sp, 32         # Crea marco de pila
            sw      $ra, 20($sp)         # M[$sp+20] <-- $ra. Guarda direcc. de vuelta
            sw      $fp, 16($sp)         # M[$sp+16] <-- $fp. Guarda $fp antiguo
            addu    $fp, $sp, 32         # Nuevo frame pointer (puntero de estructura):
                                         # $fp <-- $sp+32 (donde estaba sp antes)
            sw      $a0, 0($fp)          # Guarda argumento $a0 en marco de pila (n)

            # (2), (3), (4), (5)

            lw      $v0, 0($fp)
            bgtz    $v0, L2
            li      $v0, 1
            j       L1
L2:         lw      $v1, 0($fp)
            subu    $v0, $v1, 1
            move    $a0, $v0
            jal     fact
            lw      $v1, 0($fp)
            mul     $v0, $v0, $v1
L1:         lw      $ra, 20($sp)
            lw      $fp, 16($sp)
            addu    $sp, $sp, 32

            # (6), (7), (8), (9)
```

```
        jr      $ra
```

La rutina fact está implementada de forma recursiva, o sea, es una rutina que se llama a sí misma. Por lo tanto, ha de guardar en memoria (en una pila) sus valores actuales puesto que, al llamarse a sí misma, esos valores serán alterados y los necesitaremos a la vuelta.

Para explicar el funcionamiento del programa, en el siguiente gráfico se presenta el contenido de la memoria y de algunos registros en ciertos momentos de la ejecución del programa, rotulados como (1), (2), (3), ... en el código fuente.



La ejecución transcurre de la siguiente forma:

- Justo antes de comenzar la ejecución de la función main, la situación de la memoria y los registros es la descrita en (0). Para que el programa "arranque" hay una serie de instrucciones suministradas por el sistema operativo (invocador), que se ejecutan y que producen que en (0) la situación de los registros sea la descrita.

- La rutina main (invocado) comienza creando un marco de pila para guardar la dirección de vuelta y el puntero de estructura, que al ser recuperados al final del programa, nos llevarán de nuevo al invocador, es decir, a las instrucciones de fin de programa:

- El tamaño mínimo de una estructura de pila debe ser de 32 bytes (que es bastante mayor que lo que necesitamos para nuestros propósitos).

- Después, main llama a la rutina fact (habiendo puesto previamente en $a0 el argumento a pasarle). Cuando fact termine su ejecución, devolverá (en $v0) el resultado, que main va a escribir en consola.

- Finalmente, main debe restaurar los valores que guardó inicialmente en la pila (dirección de vuelta y puntero del marco de pila) y debe hacer que el puntero de pila apunte a la posición primitiva (lo que equivale a "borrar" la pila de main). Así, mediante la instrucción j $ra podemos volver a la posición donde se llamó a main.

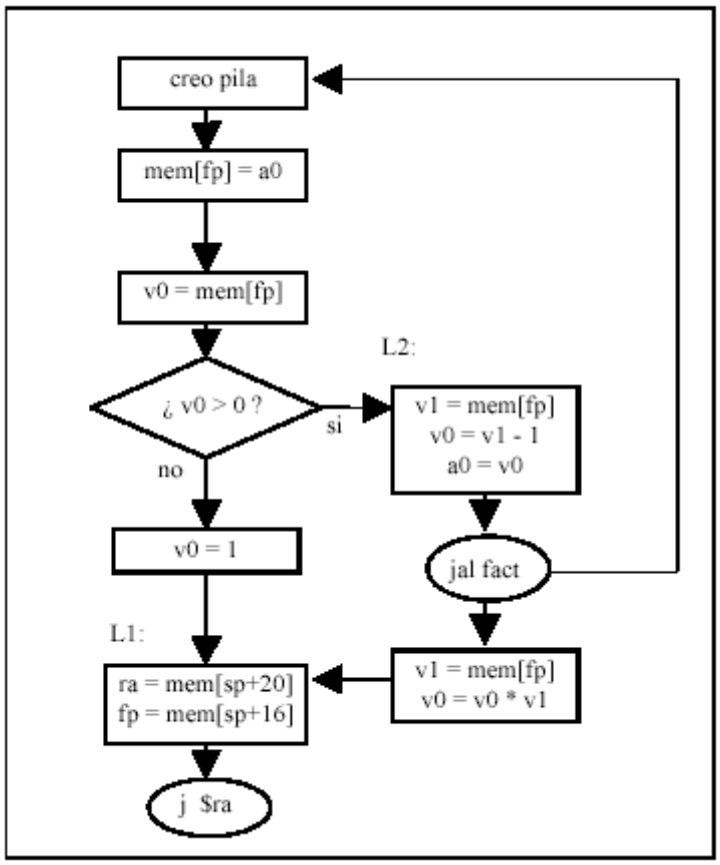
- En esa posición están las instrucciones necesarias para terminar la ejecución del programa, devolviendo el control al S.O.

En cuanto al funcionamiento de la rutina fact, como es una rutina que se llama a sí misma, actúa como invocador e invocado al mismo tiempo. Por lo tanto, puede que nos interese guardar en pila el valor de algún registro (en nuestro caso $a0, que contiene el valor del argumento) durante las sucesivas llamadas. Usaremos pues el convenio de "guardar invocador":

- Al principio de fact creamos un marco de pila, en el que guardamos la dirección de retorno y el puntero del marco de pila del invocador.
- Guardamos en esa pila el valor de $a0 (argumento n).
- Tras unas operaciones condicionales, fact se invoca a sí misma.
- El proceso se repetirá hasta alcanzar el caso base.
- Finalmente habrá que ir retornando a los sucesivos invocadores: se realizan las operaciones con los valores que se han ido guardando en las pilas para calcular el factorial, se restauran los registros con las direcciones de vuelta y los punteros de estructura y se "borran" las pilas mediante la suma del puntero de pila.

El diagrama de flujo de la rutina fact es:

Rutina **fact**


creo pila
mem[fp] = a0
v0 = mem[fp]
¿ v0 > 0 ?
si
no
L2:
v1 = mem[fp]
v0 = v1 - 1
a0 = v0
jal fact
v1 = mem[fp]
v0 = v0 * v1
v0 = 1
L1:
ra = mem[sp+20]
fp = mem[sp+16]
j $ra