Capítulo 4

**Introducción**

Ya sabemos realizar saltos (condicionales o no) para lograr que el microprocesador ejecute instrucciones situadas en **otras posiciones de memoria**. En esta sesión aprenderemos a hacer **subrutinas**, que son trozos de código que realizan una **función específica**, y que al terminar **devuelven el control** a punto siguiente en el que **fueron invocadas**. Esto es importantísimo, porque nos permite **reutilizar** código fácilmente, y **organizar** nuestros programas. Para resolver un problema complejo, **dividimos** el código en trozos más pequeños: **las subrutinas**

Las subrutinas es lo que permite que en los **lenguajes de alto nivel** (python, C, Ada, C++...) se implementen **funciones**, **procedimientos** o **métodos**. Por eso usaremos indistintamente los términos **subrutina** o **función** para referirnos al mismo concepto. Se podría decir que una subrutina es una función en alto nivel.

## Ideas claves: Organización y reutilización

Las **funciones** tienen dos misiones fundamentales:

* **Organizar y estructurar** nuestro código: Aplicamos la idea de "divide y vencerás". Para resolver un problema complejo lo dividimos en pequeños trozos (funciones). Cada función realiza una operación muy concreta
* **Reutilizar el código**: El mismo código lo podemos invocar desde diferentes partes de nuestro programa, ahorrando memoria, y haciendo más fácil el mantenimiento y depuración de los programas

Esto es válido en **cualquier lenguaje de programación**. Cuando trabajamos en ensamblador, las funciones se llaman **subrutinas**

## Programa principal y una subrutina

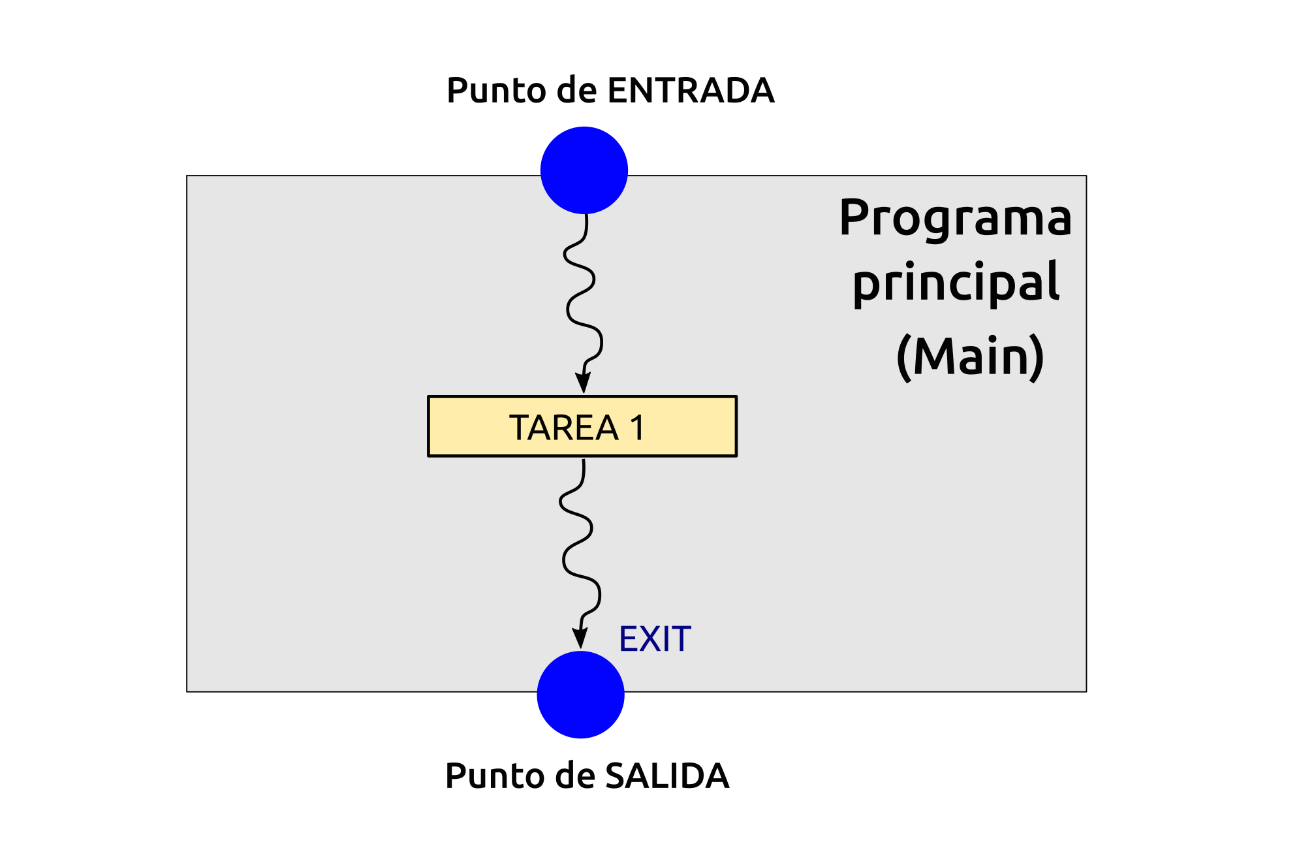
Para comprender el **mecanismo de funcionamiento de las subrutinas** partiremos de lo que ya conocemos e iremos introduciendo el resto de conceptos poco a poco. Partiremos de un **ejemplo** en el que hay que realizar una **tarea genérica** y terminar. Esta tarea puede ser cualquiera, pero la representaremos mediante la **impresión en la consola** del mensaje **"TAREA 1"**

Vamos ir viendo las diferentes **formas de organizarlo**

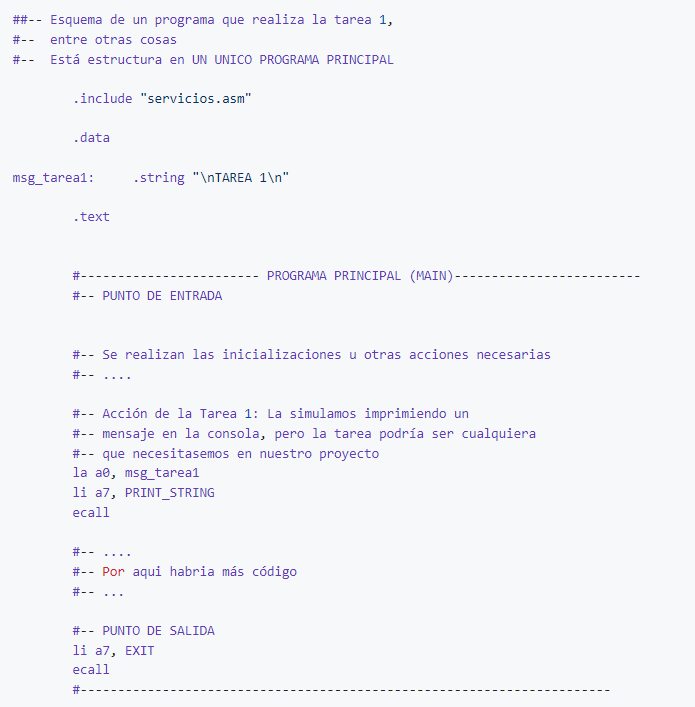
### Programa principal

Definimos **programa principal** como el **código** que está comprendido entre el **PUNTO DE ENTRADA** y **EL PUNTO DE SALIDA**. El sistema operativo pasa el control a nuestro programa, y se empieza a ejecutar su **primera instrucción** (que en el caso del simulador RARs sabemos que la primera del segmento de código: 0x00400000). Luego realiza la **Tarea 1** y finalmente termina

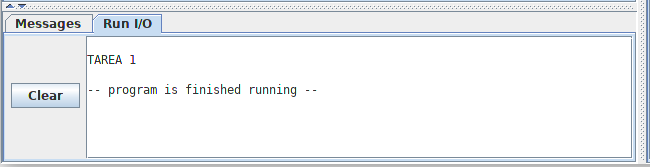
Esto lo **esquematizamos** con el siguiente **diagrama**



El **programa** de ejemplo, que realiza la **tarea 1**, es el siguiente:



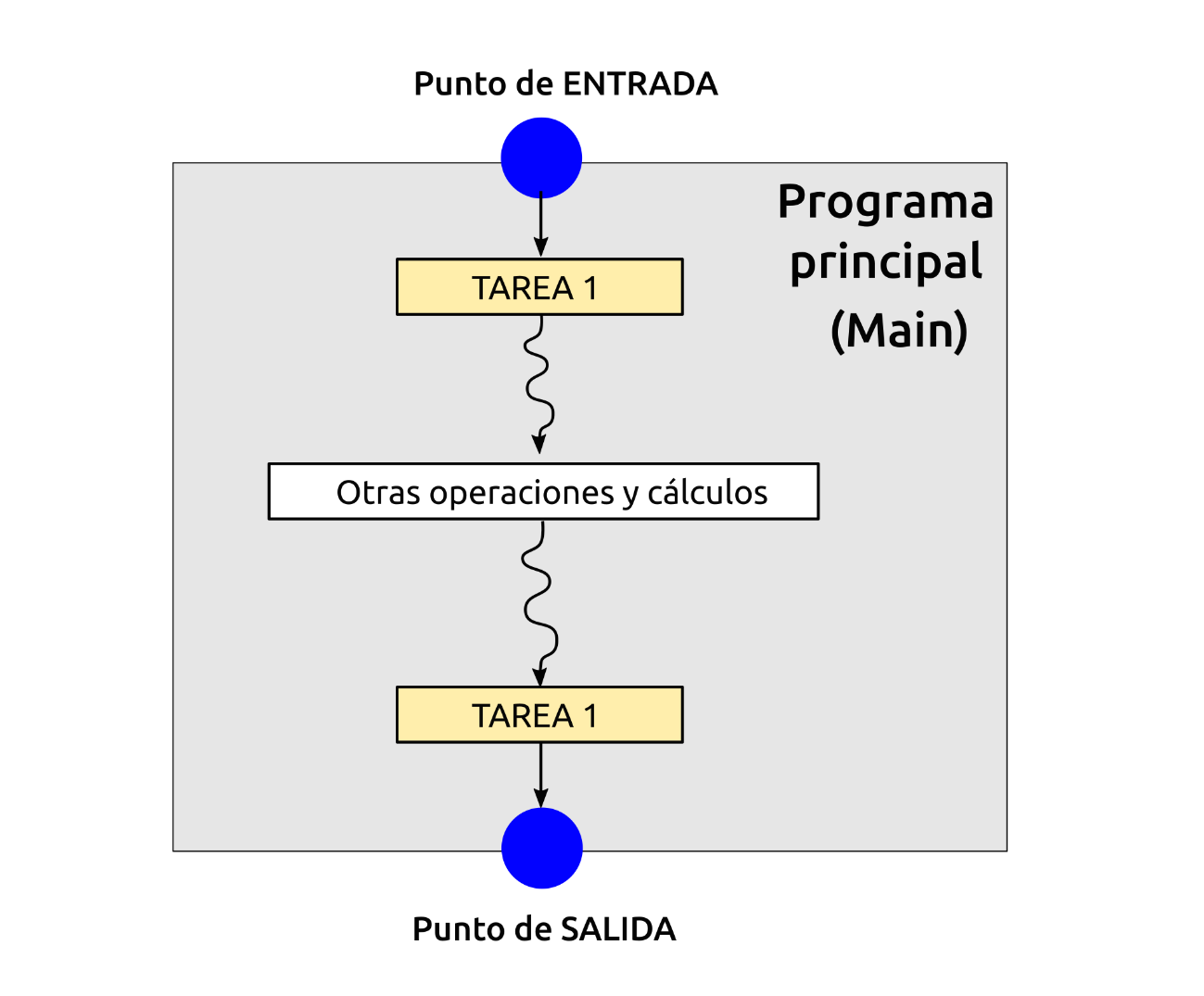
Lo **ensamblamos** y lo **ejecutamos**. En la consola vemos el mensaje de que la TAREA 1 se ha ejecutado



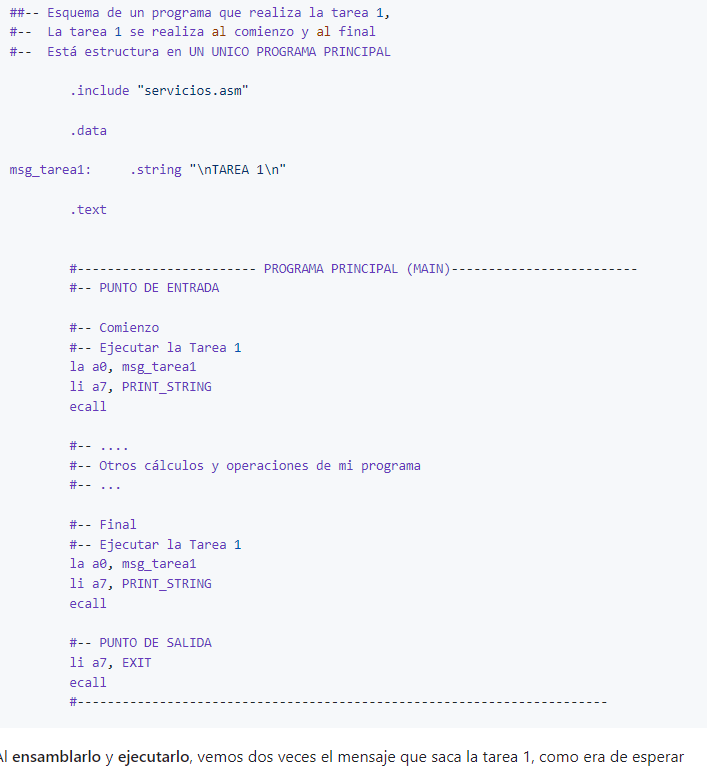
### Repetición de la tarea 1

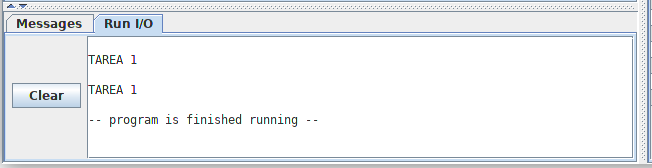
Supongamos que ahora nuestro proyecto ha sufrido **modificaciones** y es necesario **ejecutar la Tarea 1 dos veces**: una al **comenzar**, y otra al **terminar**

Con lo que sabemos hasta ahora, nuestra única opción es hacer "copy" y "paste" del **código de la tarea 1**. El **esquema** quedaría ahora así:



Y este es el **código**:





### Análisis del programa

Como se trata de un ejemplo muy sencillo, y la tarea 1 tiene muy pocas líneas de código, nos puede parecer una solución correcta. Sin embargo **NO LO ES**. Tiene los siguientes **problemas**:

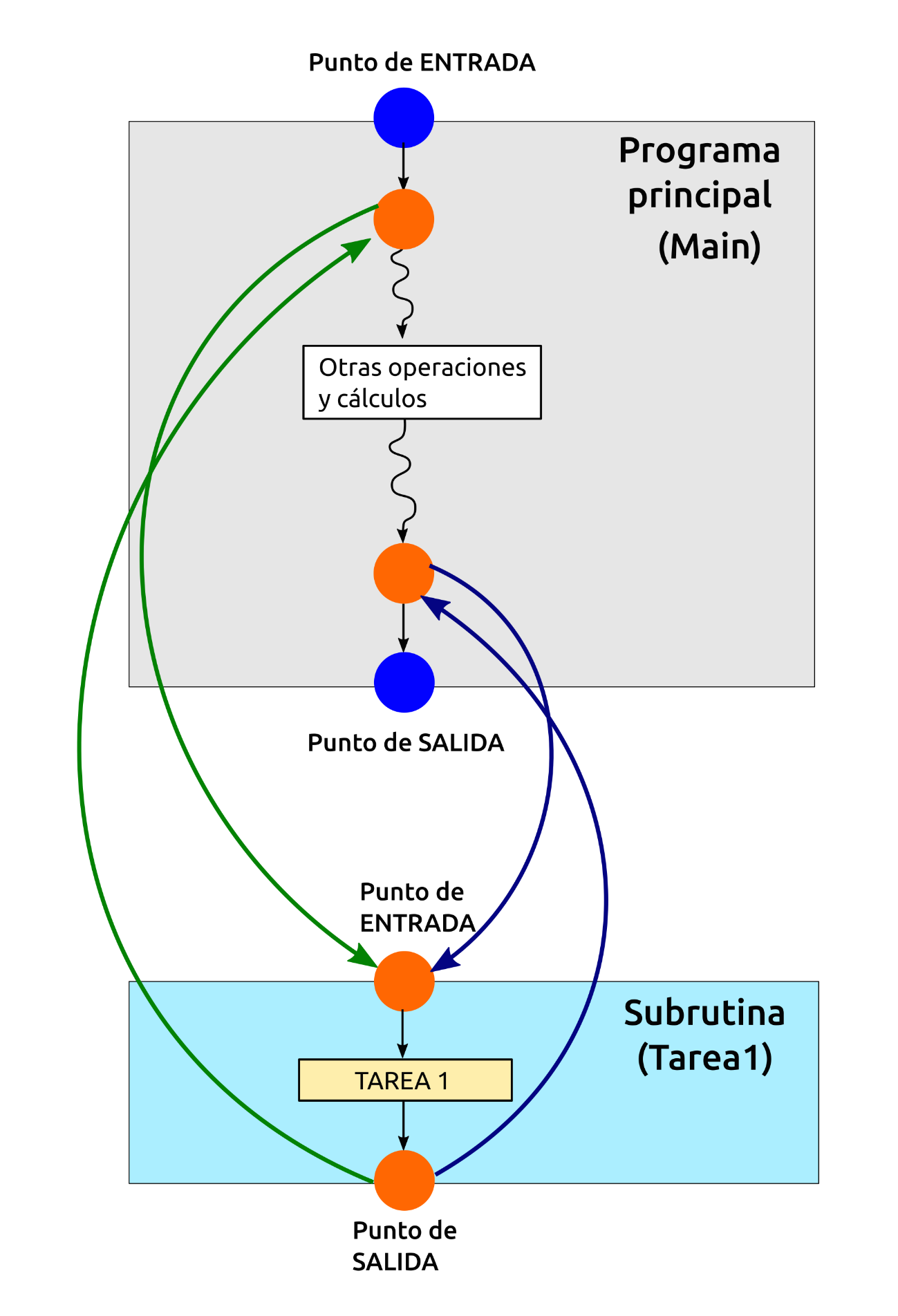
* **Duplicación de código**: En dos partes del programa estamos duplicando el código, por lo que nuestro programa va a aumentar de tamaño. Si la tarea 1 fuese más compleja y tuviese muchísimas instrucciones, crecería mucho en tamaño
* **No escalable**: ¿Y si ahora fuese necesario ejecutar la tarea 3 veces? Habría que triplicar su tamaño... Esta solución, no escala. No es general
* **Difícil de mantener**. Si hay que modificar la tarea 1 para que añadir más funcionalidad... hay que hacer en dos sitios. Esto es propenso a errores y es una **mala práctica de programación**

### **Solución: La tarea 1 es una SUBRUTINA**

El **diseño correcto** es convertir la **tarea 1** en una **función** o **subrutina**: Es un fragmento de código que realiza una **tarea específica**, y que el programa principal debe invocar cuando considere necesario. El nuevo diseño de mi programa está dividido en **dos partes**:

* **El programa principal**: Es el que se comienza a ejecutar, llama a la función tarea1 al comienzo realiza los cálculos pertinentes, vuelve a invocar la tarea1 y termina (punto de salida)
* **Una subrutina para realizar la tarea 1**

El **esquema** es el siguiente:



Ahora el **programa principal** realiza una **llamada a la subrutina**. Se ejecuta la **tarea 1**. Y esta **devuelve el control** al punto del programa principal desde donde se **invocó**

El **programa principal** realiza el resto de operaciones. Y por último vuelve a **invocar** a la subrutina tarea 1. Cuando termina, le **devuelve el control** al mismo punto desde donde fue invocada

Estas son las **ideas importantes**:

* La **subrutina** tiene un **ÚNICO PUNTO DE ENTRADA**, y un **ÚNICO PUNTO DE SALIDA**. Por supuesto, igual que ocurre con el programa principal, podría haber varios puntos de salida. Sin embargo, es una **buena práctica de programación** el tener sólo un único punto de salida
* La **subrutina** devuelve el control al punto desde donde fue invocada. Por tanto, existe un mecanismo que debe **recordar** a qué dirección **retornar**. ¿Cuál es este mecanismo?

Con este nuevo esquema se **solucionan** los problemas anteriores. Ahora sólo hay un **único código** que ejecuta la tarea 1 (NO está duplicado). El programa principal simplemente lo invoca dos veces, pero no está duplicado en memoria. Si es necesario, el programa principal lo puede **invocar tantas veces como quiera**, sin incrementar apenas el tamaño del programa

Y por supuesto, esta subrutina es más **fácil de mantener**. Normalmente las subrutinas las implementan personas diferentes. El **jefe de proyecto** hace la división de las tareas, y cada ingeniero se encarga de asegurar que su subrutina funciona correctamente

### **Las instrucciones jal/ret**

El **punto de entrada** de una subrutina lo definimos mediante una **etiqueta**, como hacemos siempre. Es la dirección donde está la **primera instrucción de la subrutina**

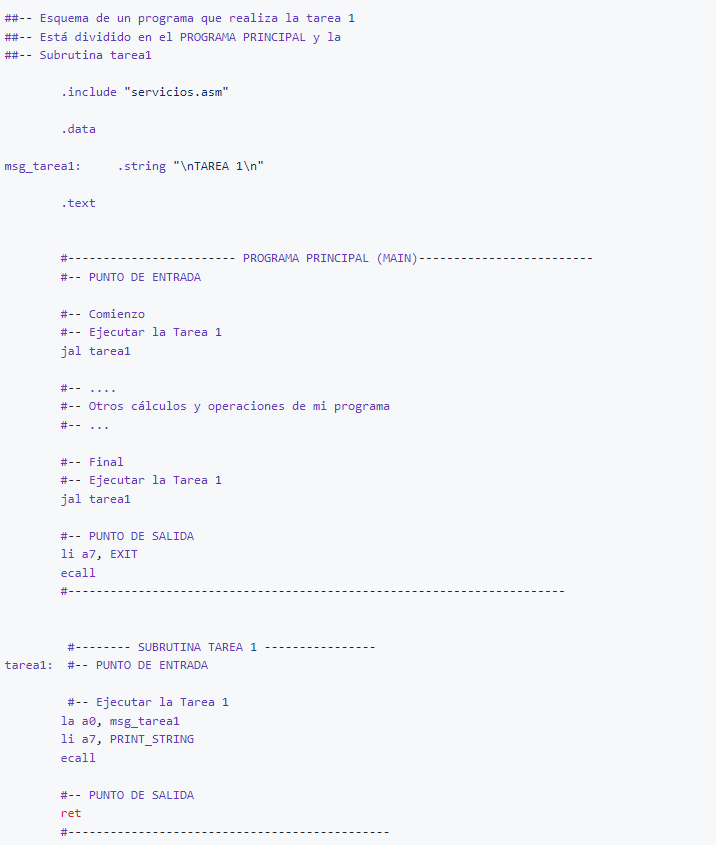
La **llamada** se realiza con la **instrucción jal**. Esta instrucción realiza un **salto incondicional** a la subrutina, pero **almacena la dirección de retorno**: es la dirección de la siguiente instrucción a jal

El **punto de salida** de la subrutina se define con la **instrucción ret**. Cuando se ejecuta esta instrucción, se realiza un **salto incondicional** a la dirección de retorno, que había sido almacenada previamente por la instrucción jal

Estas ideas se esquematizan en esta **figura**



El ejemplo anterior lo vamos a separar en el **programa principal** y la **subrutina tarea1**. Este es el **código**:



### **La magia de la instrucción ret**

**Ejecutamos** el programa anterior **paso a paso**. La **magia de las subrutina**s está en que al ejecutar la **instrucción ret**, siempre **se retorna a la instrucción siguiente al jal** usado para su invocación. Así, el primer jal está en la línea 19 (Dirección 0x00400000). Se invoca tarea1 y cuando termina devuelve el control a la siguiente instrucción, que está en la línea 27 (Dirección 0x00400004)

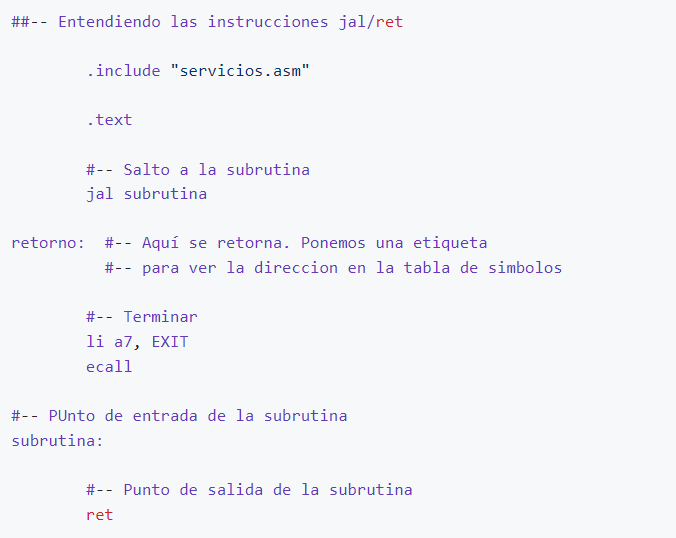
La **siguiente instrucción** es otro **jal**, que vuelve a llamar a la **subrutina tarea1**. Al ejecutar el ret, ahora se **devuelve el control** a la instrucción de la línea 30 (dirección 0x00400008)

## El registro ra

¿Cuál es el **mecanismo** que permite al procesador **recordar** la **dirección de retorno** al ejecutar ret?. Cada vez que se ejecuta la **instrucción jal**, se guarda la dirección de la siguiente instrucción en el **registro ra** (return address). Al ejecutarse la **instrucción ret**, el procesador salta a la dirección almacenada en ra

El **registro ra** es en realidad el **registro x1**. Es un registro de propósito general, que se puede usar para cualquier otra cosa. Sin embargo, en la **ABI del RISCV** se ha establecido que **SOLO LO PODEMOS USAR** con este propósito de **almacenar la dirección de retorno**. Por ello, **NO DEBEMOS USARLO NUNCA CON OTROS FINES**

Para entender el funcionamiento, usaremos este código, que simplemente llama a una subrutina vacía, que sólo contiene la **instrucción ret**



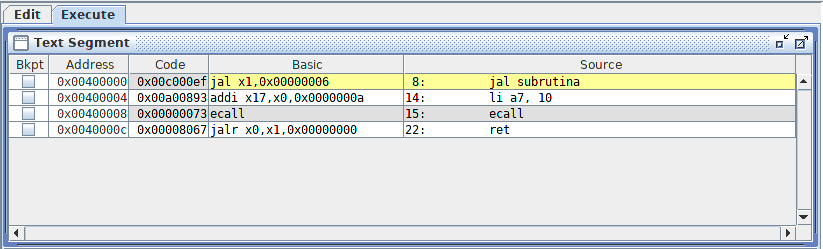
Lo **ensamblamos** y nos fijamos en la **tabla de símbolos**:

| **Etiqueta** | **Dirección** | **Descripción** |
| --- | --- | --- |
| retorno | 0x00400004 | Dirección de retorno (siguiente a jal) |
| subrutina | 0x0040000c | Punto de entrada de la subrutina |

Al **ejecutarlo paso a paso** vemos lo que ocurre: Con la instrucción **jal subrutina**, se almacena en **ra** la **dirección de retorno**, que es la **0x00400004**, la siguiente al jal

El procesador **salta** a la dirección **0x0040000c**, que es el **punto de entrada** de la subrutina. Ahí se encuentra la **instrucción ret**. Al ejecutarla, se realiza un salto a la dirección indicada por ra, con lo cual se retorna

En realidad, **jal / ret** son pseudoinstrucciones. En el ejemplo anterior, el ensamblador las ha convertido a las **instrucciones reales**:



El **salto a la subrutina** se hace mediante la instrucción:

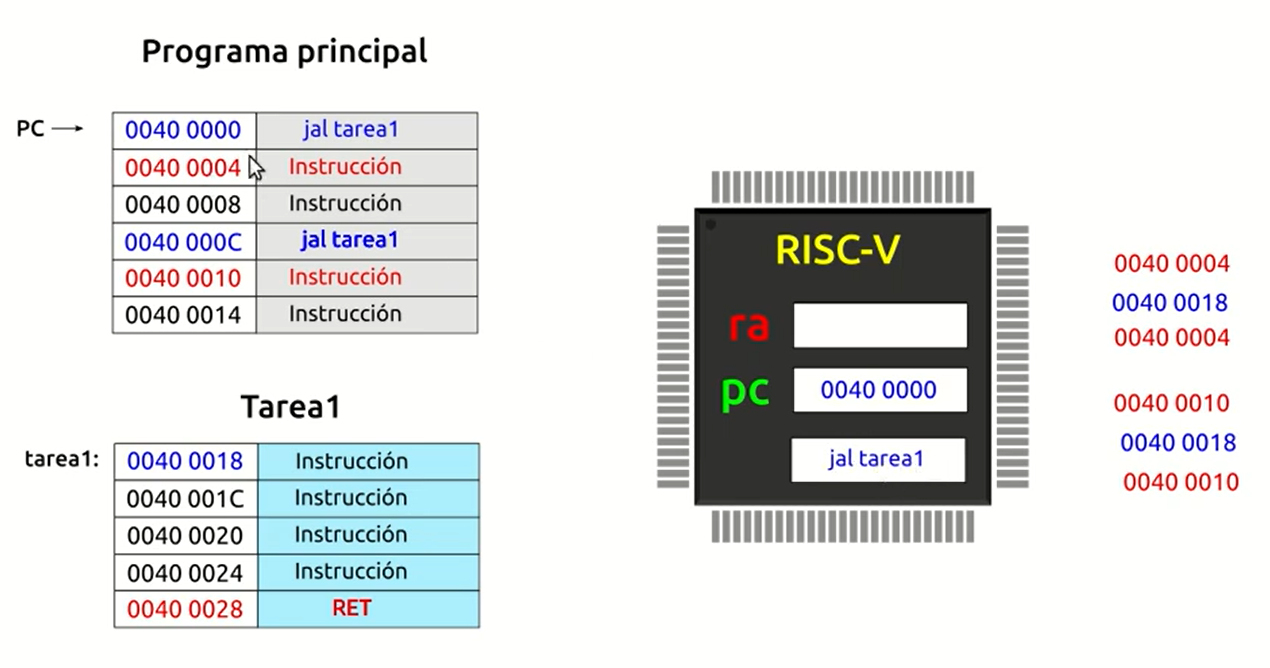
Esta instrucción significa: Saltar hacia adelante **6 medias palabras** (3 instrucciones) y almacenar en el **registro x1** (que es ra) la dirección siguiente a la propia instrucción jal

La **instrucción ret** se ha sustituido por:

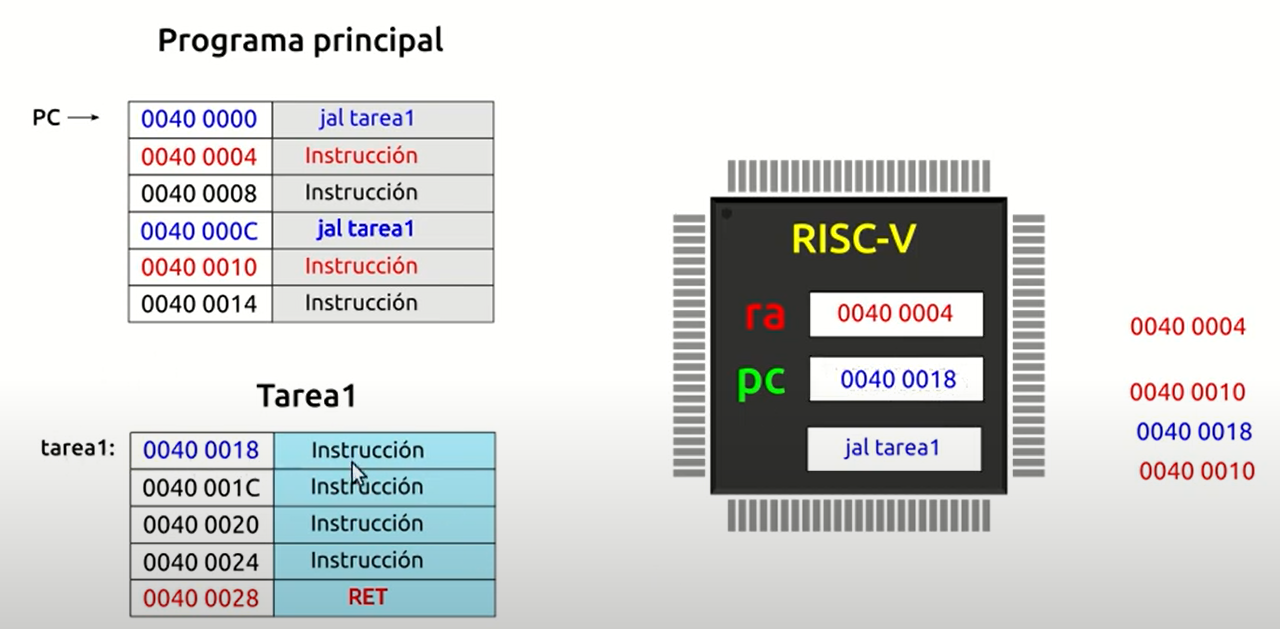
Su significado es: Salta a la dirección dada por el **registro x1 + 0**, y la dirección siguiente a jalr guardarla en x0 (es decir, ignorarla)

En realidad, en "humano" la escribimos así:

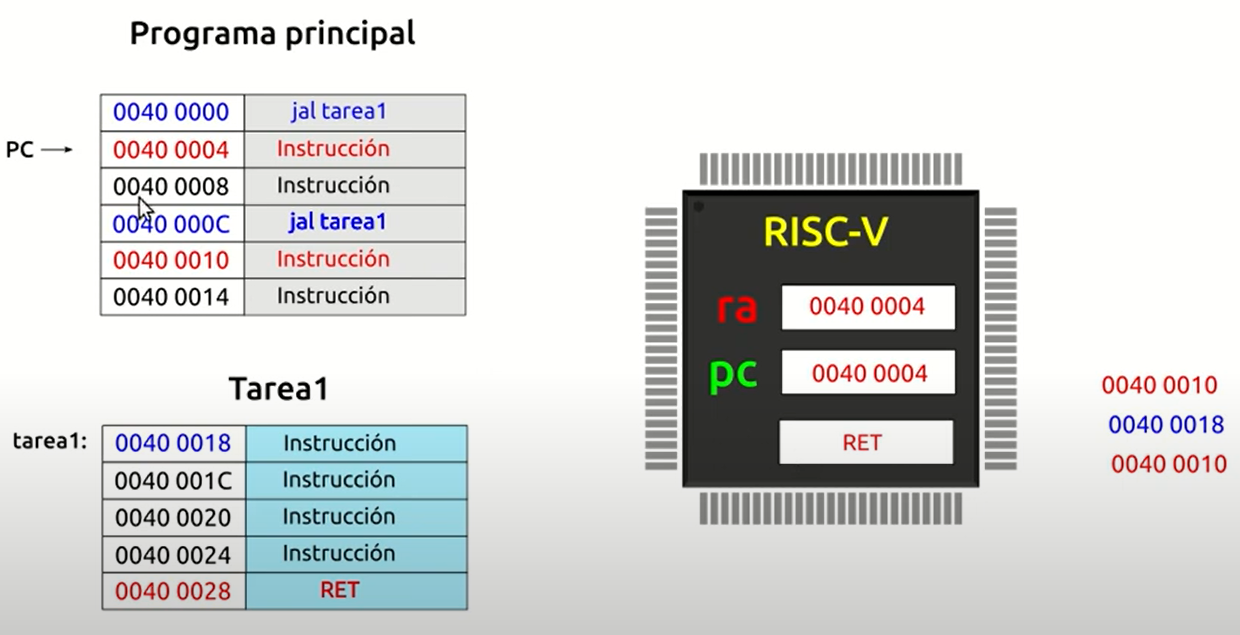
Explicación para dummies si no lo has entendido:



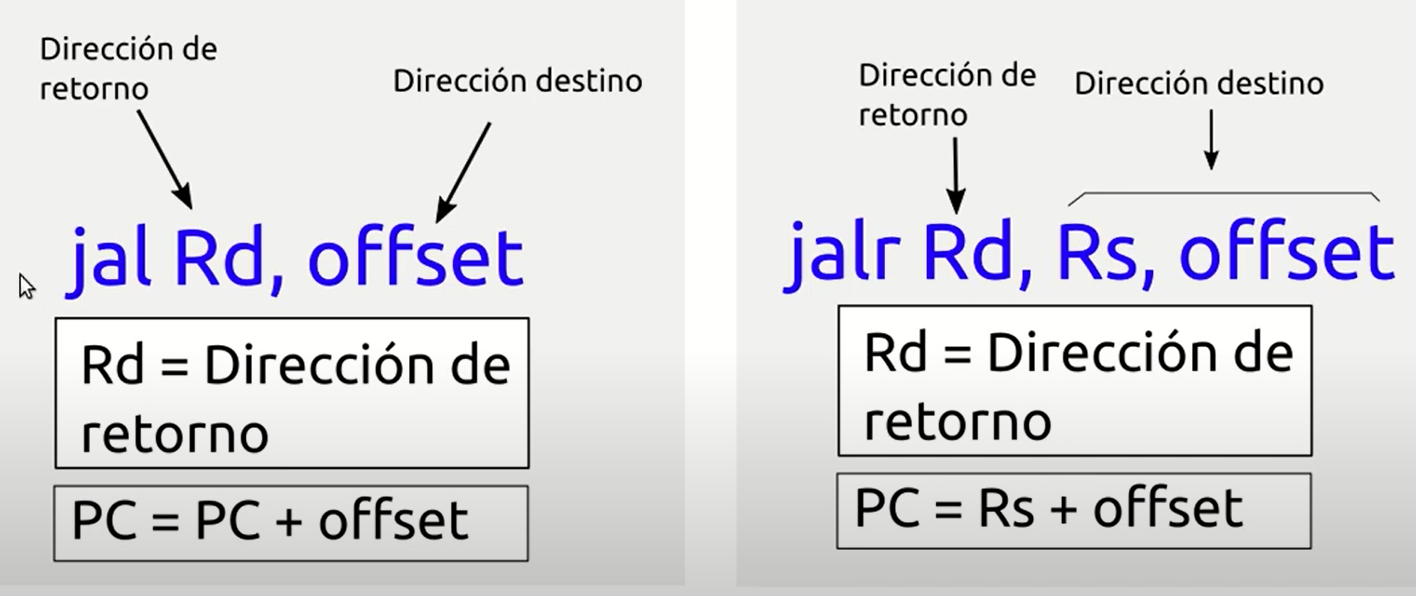
Nuestro programa se inicia, como sabemos, en el registro PC se guarda la siguiente instrucción a ejecutar. Una vez se ejecuta la acción de la dirección 0040 0000, en el registro PC vamos a tener la primera dirección de la tarea (0040 0018) y automáticamente se guarda en RA la dirección de retorno, para que cuando se termine de ejecutar tarea 1, el procesador sepa donde tiene que regresar.



Una vez entramos en Tarea1, se van ejecutando las instrucciones que hay en esta subrutina. Y una vez llegamos a RET, lo que se hace es poner la dirección que tenemos en RA al registro PC, para que la siguiente instrucción que se ejecute sea la dirección de retorno.



También recordamos:



Donde jal (JUMP AND LINK) es relativo a nuestra posición, saltaría a una posición que es la suma de la dirección en PC y una posición indicada. Y jalr es saltar a una posición que es la suma de un registro que le pasemos mas una posición que también le pasemos.

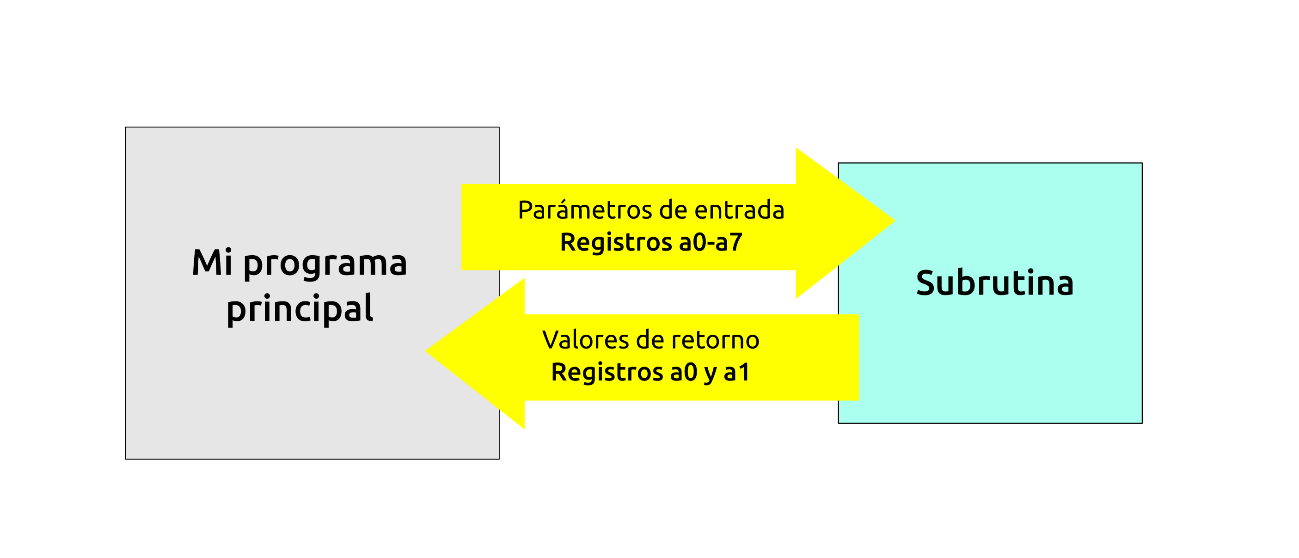


**Paso de parámetros**

Tanto el **programa principal** como la **subrutina** pueden intercambiar **información**. El programa principal envía información mediante los **parámetros de entrada** y la subrutina devuelve información mediante los **valores de retorno**

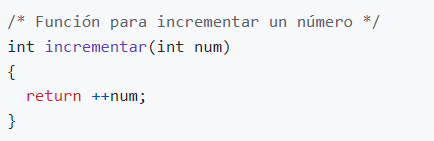
El **convenio** establecido para el **paso de parámetros** y de **valores de retorno** es el siguiente:

* El **programa principal** envía datos a la subrutina mediante los **registros a0-a7** (¡¡¡Y SÓLO ESOS!!!). Además, se hace en orden: si sólo tiene un argumento, se pasa por a0. Si tiene dos, se pasa el primero por a0 y el segundo por a1. Y así sucesivamente
* La **subrutina** devuelve los **valores de retorno** en los **registros a0 y a1**, también en orden. Si sólo devuelve un valor, tiene que ser por a0. Pero como máximo puede retornar dos valores. En caso de que quisieras pasar muchos parámetros como retorno, lo que se hace es que tu programa principal te de como entrada un puntero con una dirección de memoria para depositar ahí los resultados.
* Una subrutina sólo tiene acceso a los registros que se le pasan como entrada con algunas excepciones que veremos mas adelante.



### **Ejemplo: función de incremento**

Vamos a programar una **subrutina** (función) a la que se le pasa como **entrada un número** y **devuelve** su valor incrementado en una unidad. El nombre de la función es **incrementar**. Si la implementásemos en **lenguaje C**, sería así:



Si la implementásemos en **python**, sería así:

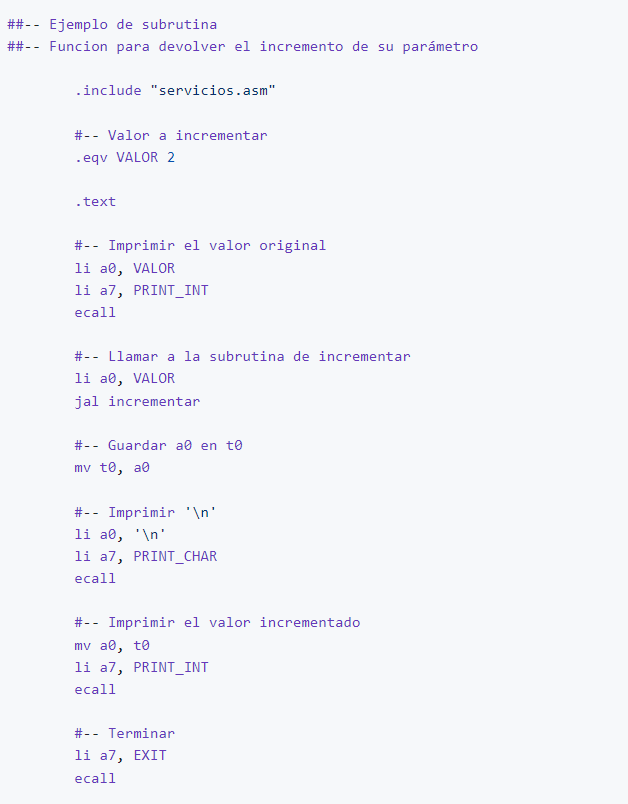


La implementación de la **subrutina en ensamblador** sería así:



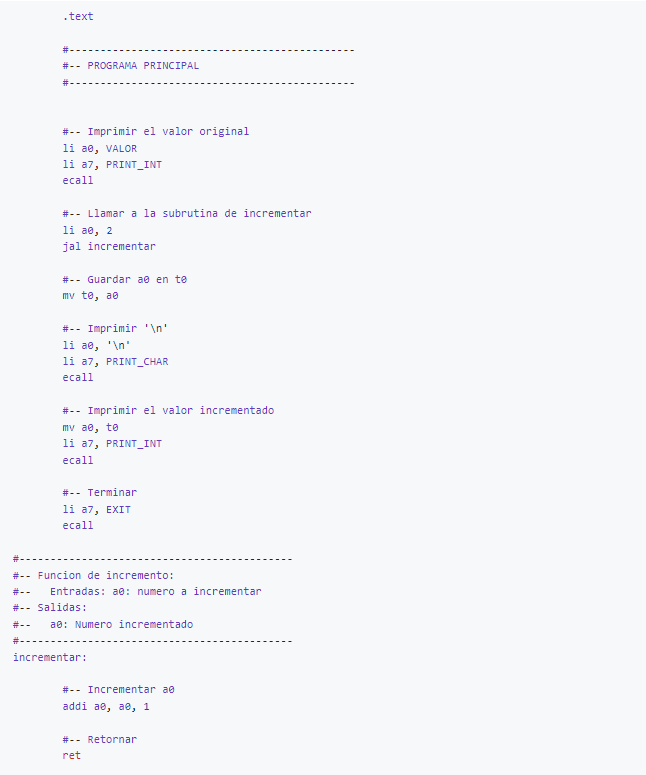
Como sólo tiene **un parámetro de entrada**, es **obligatorio** pasarlo por **a0**. Como sólo tiene **un parámetro de salida**, es obligatorio devolverlo por **a0**

El **programa principal** imprime el valor original, llama a la función incrementar, imprime un salto de línea, luego imprime el resultado y termina. Lo implementamos de esta manera:

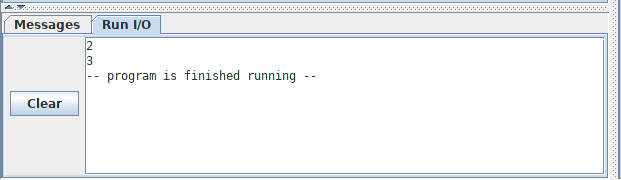


Es importante hacer notar que se trata de **dos partes independientes** del programa. Una parte es el **programa principal**, y la otra la **función incrementar**. Sólo se comunican entre ellas a través del **registro a0**, y nada más

A la hora de implementar el **programa completo** tenemos varias opciones. Una es **meterlo todo en un único fichero**. Típicamente ponemos el **programa principal al comienzo**, y tras su **punto de salida** la(s) subrutina(s):



**Ensamblamos** el programa y lo **ejecutamos**. En la consola vemos primero el número 2 y debajo el 3: es el 2 incrementado en una unidad por nuestra función

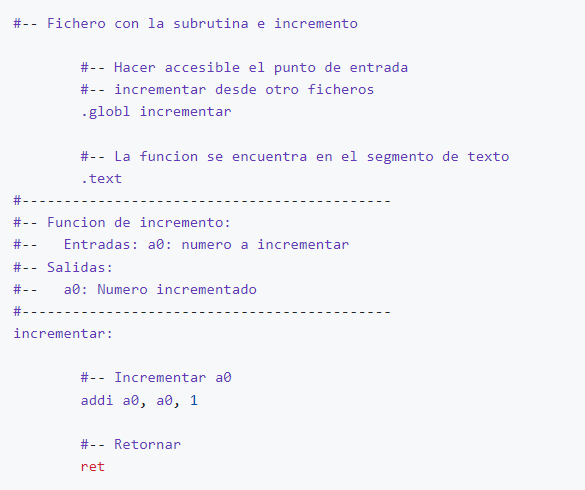


### **Separación por ficheros**

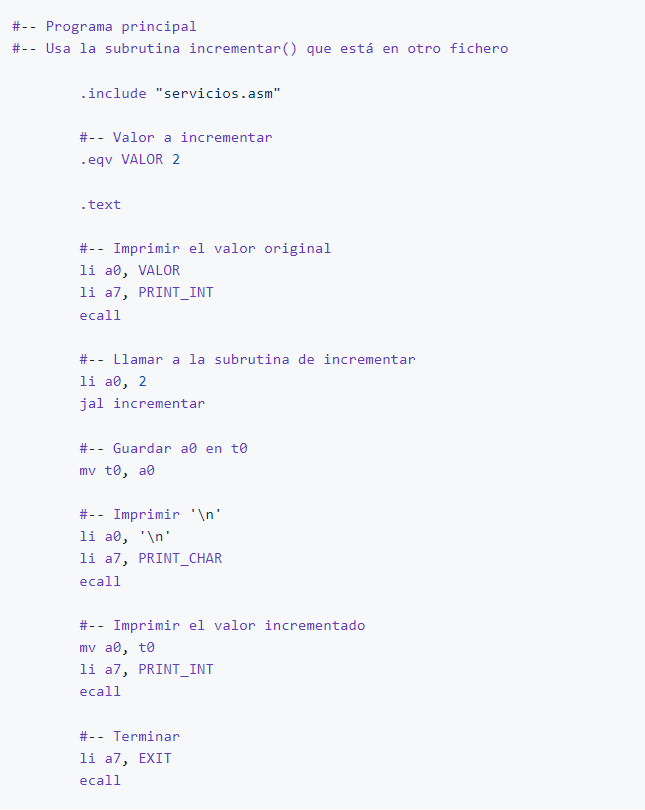
Sin embargo, tanto el **programa principal** como las subrutinas las podemos colocar en **ficheros separados**. El **ensamblado** se hace de **forma independiente**, cada fichero por separado y un programa llamado **enlazador** (linker) se encarga de unirlos y colocarlos correctamente en memoria

Para realizar la separación en ficheros, hay que usar la **directiva .globl** en el fichero de la **subrutina** para que la etiqueta con su **punto de entrada** sea accesible desde el otro fichero. Así, el ejemplo anterior lo dividimos en dos ficheros:

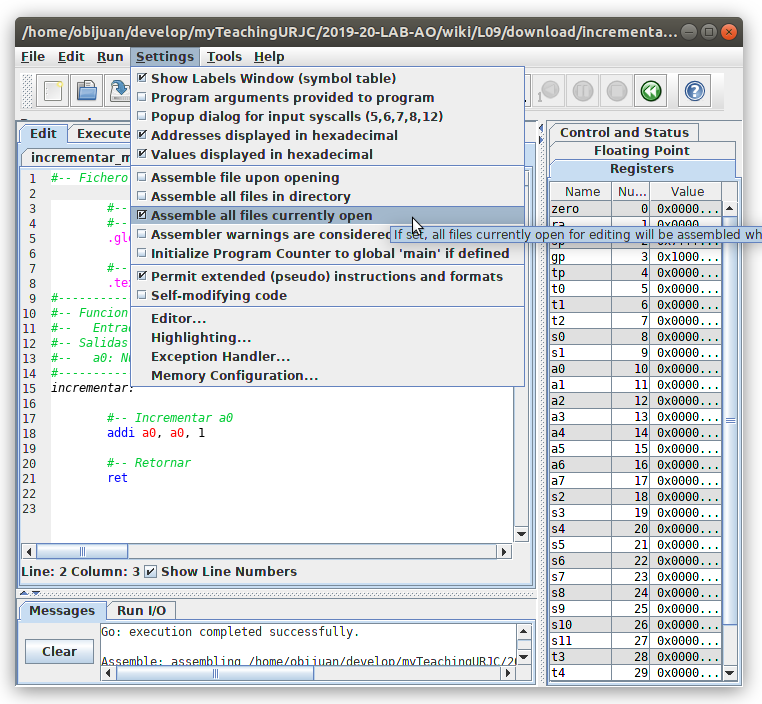
* Fichero **incrementar.s**: contiene la subrutina



* Fichero **incrementar\_main.s**: Contiene el programa principal

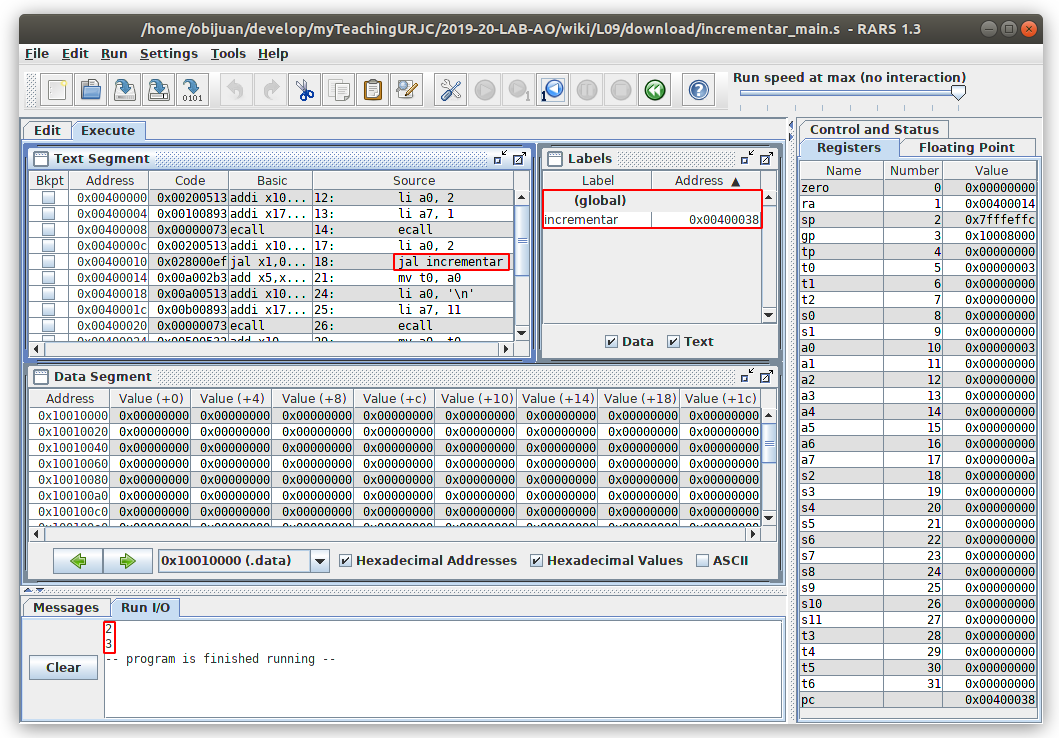


Para que el **simulador** pueda **generar** correctamente el código máquina cuando hay **varios ficheros**, tennemos que **abrir todos los ficheros implicados** (y sólo esos ficheros) en las pestañas del editor y **activar** la opción: **settings/assemble all files currently open**



Ahora **NOS SITUAMOS SOBRE EL PROGRAMA PRINCIPAL** y lo ensamblamos. Si tenemos activada la visualización de la **tabla de símbolos**, veremos que la **etiqueta incrementar** está marcada como **global**

**Para decirle al programa cual es el programa principal, que el no sabe cual es, cuando ensamblamos tenemos que estar en la pestaña del programa principal. El programa reconoce el programa que hay en esta pestaña como el principal.**

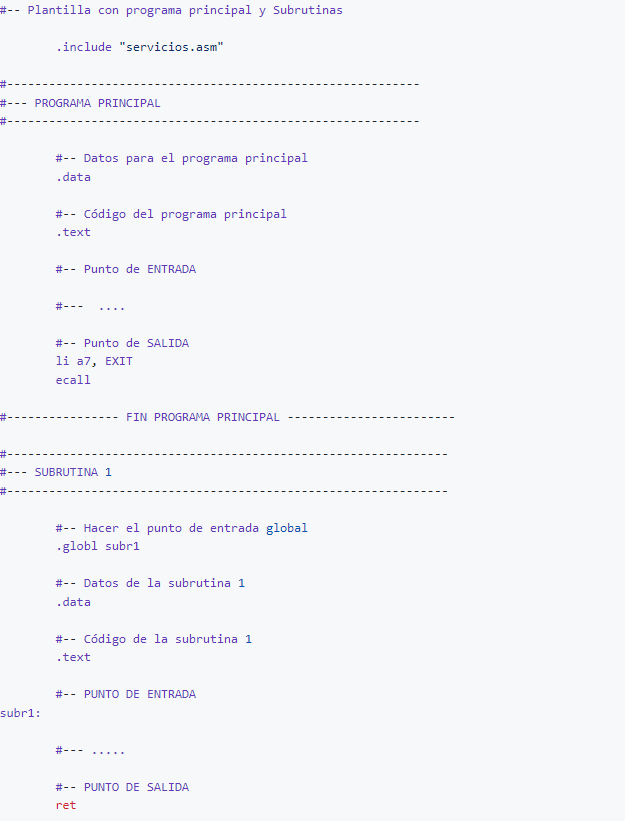


Lo ejecutamos normalmente

### **Datos y código**

Cuando se hace la **división** en **programa principal** y **subrutinas**, cada parte tendrá **código** y opcionalmente **datos**. De esta manera, en cada subrutina habrá que colocar también la **directiva .text** para indicar la parte del código y **.data** si hubiese datos que son sólo de la subrutina. Esto es especialmente importante si cada parte está en un fichero diferente

La **plantilla** sería así:



Bien lo podemos poner todo en el **mismo fichero**, o bien lo podemos **separar** en varios ficheros

**Convenio de uso de los registros**

Los registros **NO PODEMOS** usarlos como queramos, porque hay unos convenios para que podamos reutilizar los programas. La violación de estos convenios se considera un **ERROR GRAVE**, y aunque el programa realice su función correctamente, sería incompatible con el código hecho por otros miembros del equipo

* Sólo se pueden **pasar parámetros** a nuestras subrutinas mediante los **registros a0-a7**. NUNCA lo haremos usando otros registros
* El paso de parámetros a las subrutinas se hace en orden: El primer argumento se pasa por a0, el segundo por a1, el tercero por a2, etc.
* Sólo se pueden **recibir valores de retorno** usando los **registros a0 y a1**
* Los valores de retorno se hacen en orden. Si la función tiene un único valor de retorno se devuelve por a0. Si tiene dos, el segundo por a1
* Los **registros s0-s11** son registros **estáticos**. Esto significa que NO se pueden modificar al ejecutar una subrutina: Su valor, antes y después de llamar a una subrutina tiene que ser el mismo (no puede cambiar)
* Los **registros del t0-t6** son **temporales**. Esto significa que tras la llamada a una subrutina debemos suponer que **HAN CAMBIADO**. O, dicho de otra manera, si después de llamar a una subrutina usamos los registros temporales, es OBLIGATORIO inicializarlos con algún valor (no podemos dar por supuesto que tienen el mismo valor que antes de llamar a la subrutina)
* La misma regla anterior se aplica a los **registros a0-a7**: Se consideran temporales, aunque su uso está establecido para el paso de parámetros

**Algunos consejos para no violar el convenio de uso de los registros**

El **convenio de uso de registros** es muy sencillo, pero es **traicionero**. A veces pensamos que NO lo estamos violando, pero en realidad sí. Aquí exponemos algunas reglas que seguiremos en los ejercicios para garantizar que se cumple el convenio:

* En el **programa principal** (main) usaremos los **registro estáticos s0-s11** para almacenar información que necesitamos durante el programa principal: punteros a cadenas, punteros a variables, contadores, etc. Puedes usar cualquier de ellos sin temor
* En el **programa principal**, usaremos los **registro temporales t0 - t6 entre llamadas a subrutina**. Una vez que llamemos a una subrutina, **DEBEMOS SUPONER** que **los registros temporales han perdido sus valores anteriores**. Si los queremos usar, habrá que inicializarlos otra vez
* Dentro de las **subrutinas** sólo usaremos **registros temporales**
* Los **registros a0-a7** los reservamos para el **paso de parámetros**

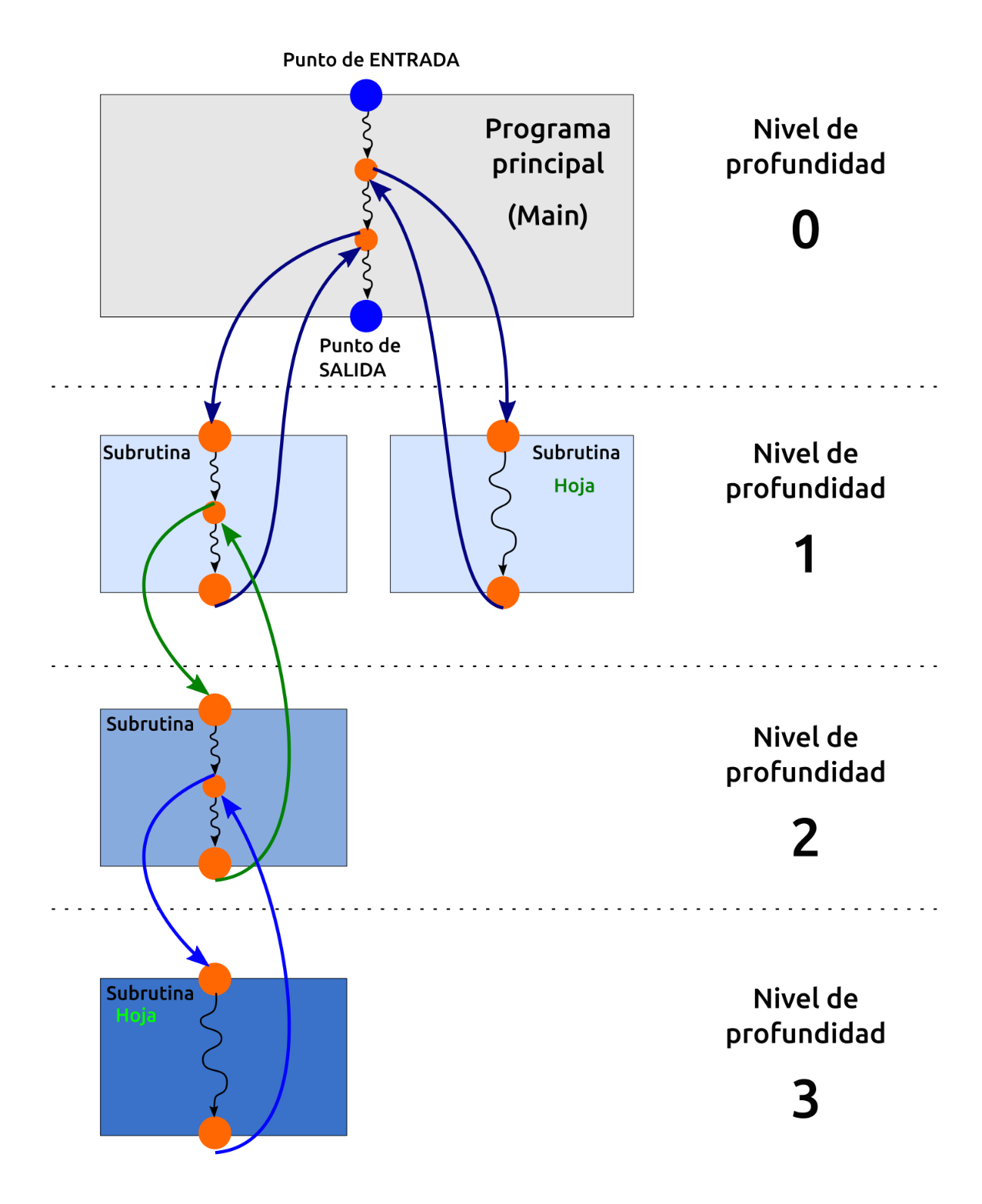
## Pila

Para completar el estudio de las **subrutinas**, necesitamos conocer el **mecanismo** que nos permite realizar llamadas entre subrutinas a **varios niveles**, y no sólo con un nivel, como hemos visto hasta ahora. Además, aparecen problemas al mezclar el **uso de los registros**, definidos en la ABI del RISCV, con las **llamadas a subrutinas**. Para solucionar todos esos problemas necesitamos utilizar **la pila**

## Niveles de profundidad

Ya sabemos **dividir** nuestros programas en **dos partes**: el **programa principal** y una **subrutina**, que podemos llamar tantas veces como queramos, y que nos **devolverá el control** al punto siguiente desde donde fue invocada

Para abordar **problemas más complejos**, nos gustaría poder **dividir la subrutina**, a su vez, en **más subrutinas**. Así, tenemos una serie de **subrutinas encadenadas**, unas llamando a otras. Cada vez que llamamos a una subrutina, decimos que **descendemos un nivel de profundidad**. El **programa principal** está en el **Nivel de profundidad 0**. Al llamar a una subrutina, estamos en el **nivel 1**. Si desde el nivel 1, llamamos a otra subrutina, estaremos en el **nivel de profundidad 2**. Y así sucesivamente



El **número de niveles** depende del número total de funciones en las que se haya particionado el problema y de cómo están organizadas. Es algo que define el **jefe de proyecto**. Nosotros en esta asignatura nos limitaremos a implementar las subrutinas tal cual nos las pidan

Según el **nivel de profundidad** dónde se encuentra cada función, establecemos la siguiente **clasificación**:

* **Programa principal**: Es el que está en el **Nivel 0**. Se encarga de invocar al resto de funciones
* **Subrutina Hoja** (o función Hoja): Es una subrutina que NO invoca a más subrutinas. Por tanto, está en su último nivel de profundidad
* **Subrutina intermedia**: Es una subrutina que **NO** es el programa principal **NI una subrutina hoja**. Se encuentran a partir del nivel 1 y realizan al menos una llamada a otra función (que estará en el nivel 2)

**Registros preservados y NO preservados**

La **ABI del RISCV** divide los registros en **dos categorías**, según el **comportamiento** que tienen al **invocar** a las **subrutinas**:

* **Registros preservados**: Su **valor** debe ser el mismo **ANTES** y **DEPUÉS** de realizar la **llamada a la subrutina**. Es decir, son registros que está **prohibida su MODIFCACIÓN** en la subrutina. Se pueden usar dentro de la subrutina, pero sin alterar su valor. Antes de usarlos hay que **guardar su valor en memoria**, y **recuperarlo** antes de retornar de la subrutina. Así, el nivel superior los debe ver como si no se hubiesen modificado nunca

Los registros que se **preservan** son los **estáticos**: *s0-s11* y el **puntero de pila**: *sp*

* **Registros NO preservados**: Las subrutinas pueden **alterar** su valor. No sabemos qué subrutinas alteran qué registros. Por eso **SIEMPRE** supondremos que, al llamar a una subrutina, estos registros **PIERDEN SU VALOR**. Y, por tanto, tras cada llamada habrá que **inicializarlos** otra vez

Los registros que **NO preservan** su valor son los **temporales**: *t0-t6* y los de **argumentos**: *a0-a7*

## Respetando la ABI RISCV

Debido a la existencia de estos dos grupos de registros: los **estáticos** (cuyo valor se preserva entre llamadas) y los **temporales** (cuyo valor se pierda entre llamadas) hay que tener **cuidado** al usarlos. Dependiendo de la parte del programa en la que estemos trabajando, podremos usar unos registros u otros

### Trabajando en el programa principal

Los **registros estáticos** los podemos usar a **discreción**. Los temporales también, pero teniendo en cuanta que los tenemos que **INICIALIZAR** siempre después de cada llamada a subrutina

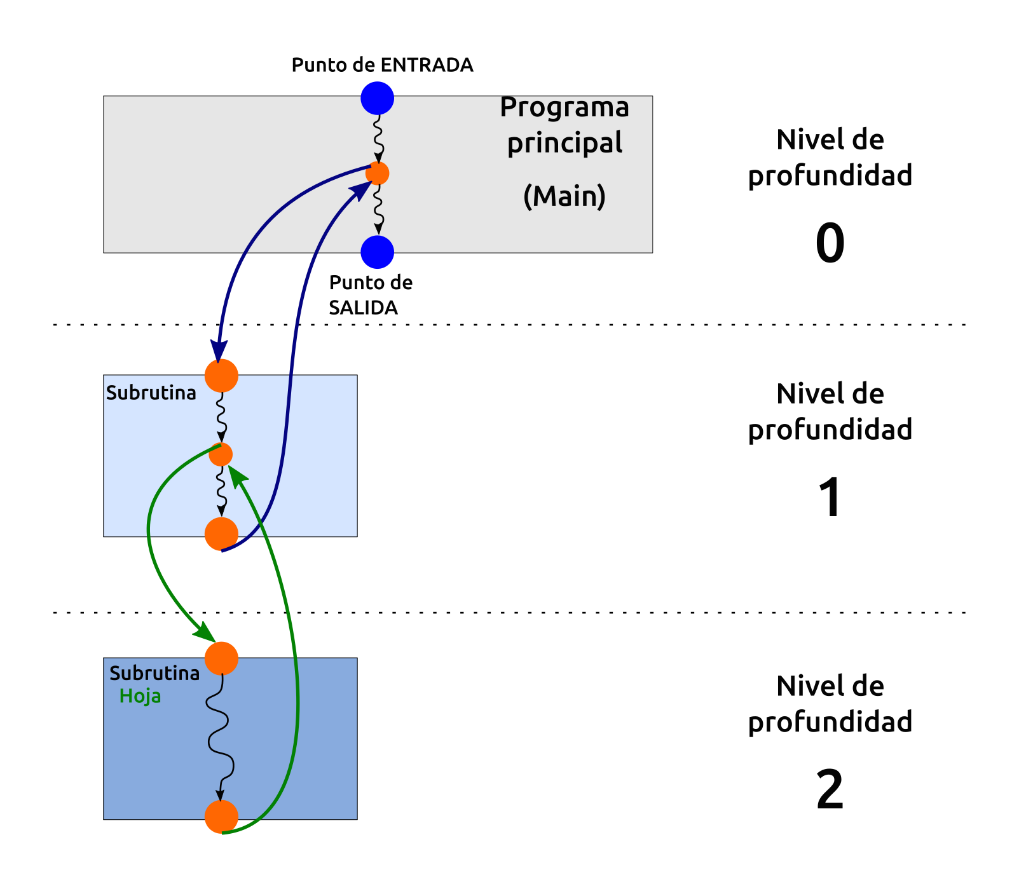
#### Usando registros estáticos en el programa principal

En el **programa principal**, al estar en el **nivel 0**, podemos usar los **registros estáticos** s0-s11 como queramos y cuando queramos. Tenemos garantizado que su valor **NO LO CAMBIARÁ** ninguna subrutina

## Subrutinas intermedias y dirección de retorno

Cuando desde el **programa principal** llamamos a una subrutina que está en el **nivel 1**, la dirección de retorno se almacena en el **registro ra**. Esto garantiza que al ejecutar la instrucción ret se vuelve a la siguiente posición desde donde se invocó la subrutina

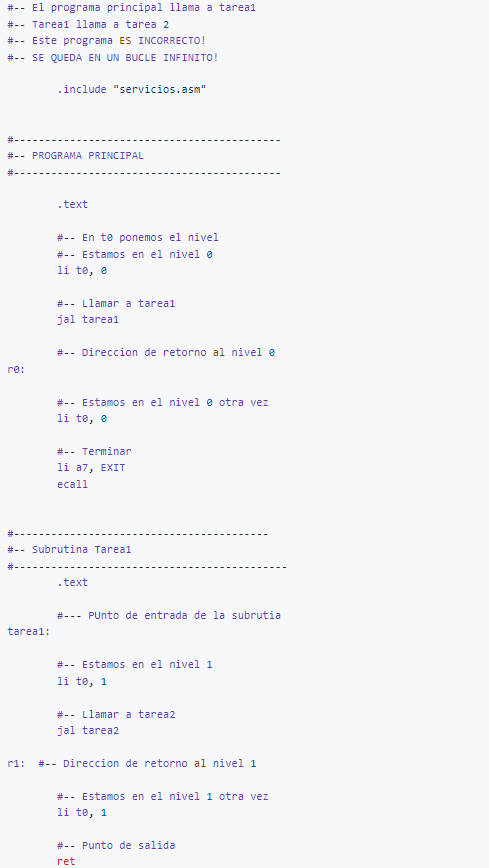
Pero ¿qué ocurre si desde esta subrutina de **nivel 1** llamamos a su vez a otra de **nivel 2**?



Ahora es necesario **recordar 2 direcciones de retorno**. Una es la de retorno al **nivel 0** (programa principal), y la otra es la del retorno al **nivel 1**. La subrutina de nivel 2 es una función hoja, que no llama a ninguna otra, por lo que no tiene que recordar nada

Pero como sólo tenemos un **registro ra** con ese propósito, al guardar la dirección de retorno de la segunda llamada, **machacamos** el valor de la primera, perdiéndose su valor.

Esto lo podemos probar con el siguiente **ejemplo**, en el que se ha definido un **programa principal** que invoca a **tarea1** y esta a su vez a **tarea 2**. Ninguna función realiza nada, sólo las llamadas y el retorno



Al **ensamblar** y **ejecutar** este programa, vemos que ¡se queda en un **bucle infinito**! Al realizar la llamada a la segunda subrutina, su dirección de retorno se guarda en ra, **machacando** la anterior, por lo que **NUNCA** ¡retorna al programa principal!

### ¿Cómo lo solucionamos?

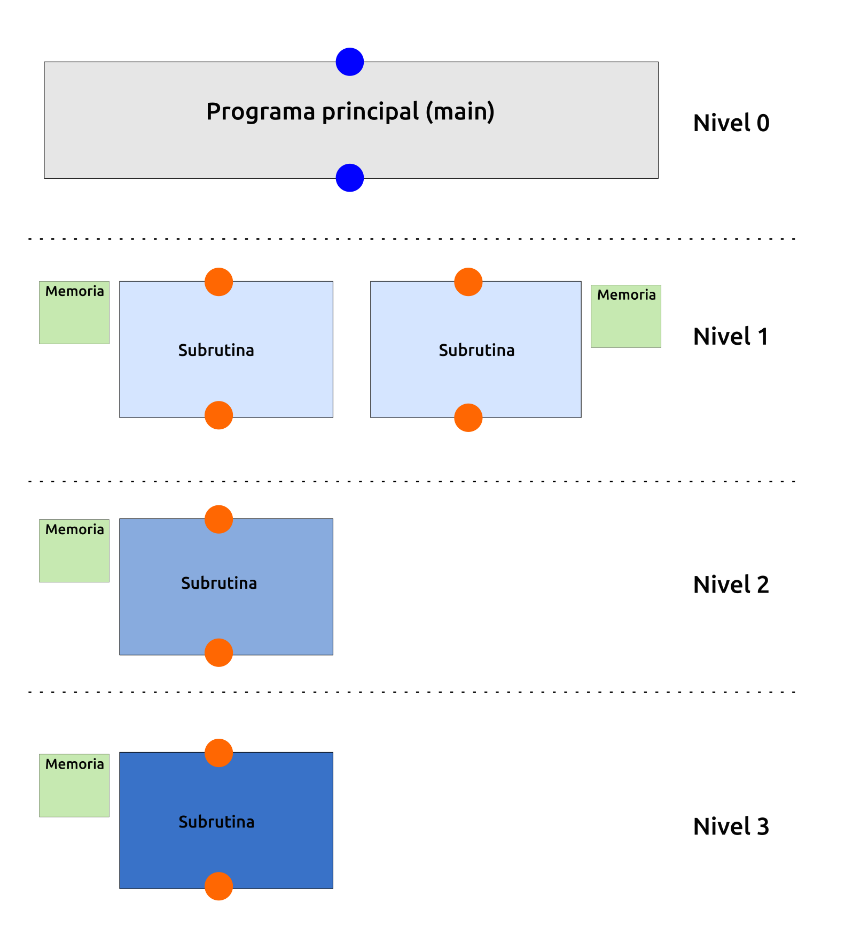
En todas las **subrutinas INTERMEDIAS** (en las hojas no hace falta) hay que **guardar** el contenido del **registro ra** en la **pila (una zona de memoria segura)** y recuperarlo antes de ejecutar la instrucción RET.

Como todavía no sabemos crear la Pila, esto lo veremos más adelante.

## Subrutinas y memoria

Hemos visto que las **subrutinas** necesitan una **zona de memoria propia** para almacenar la **dirección de retorno**, **preservar los registros** y también para guardar **variables locales**

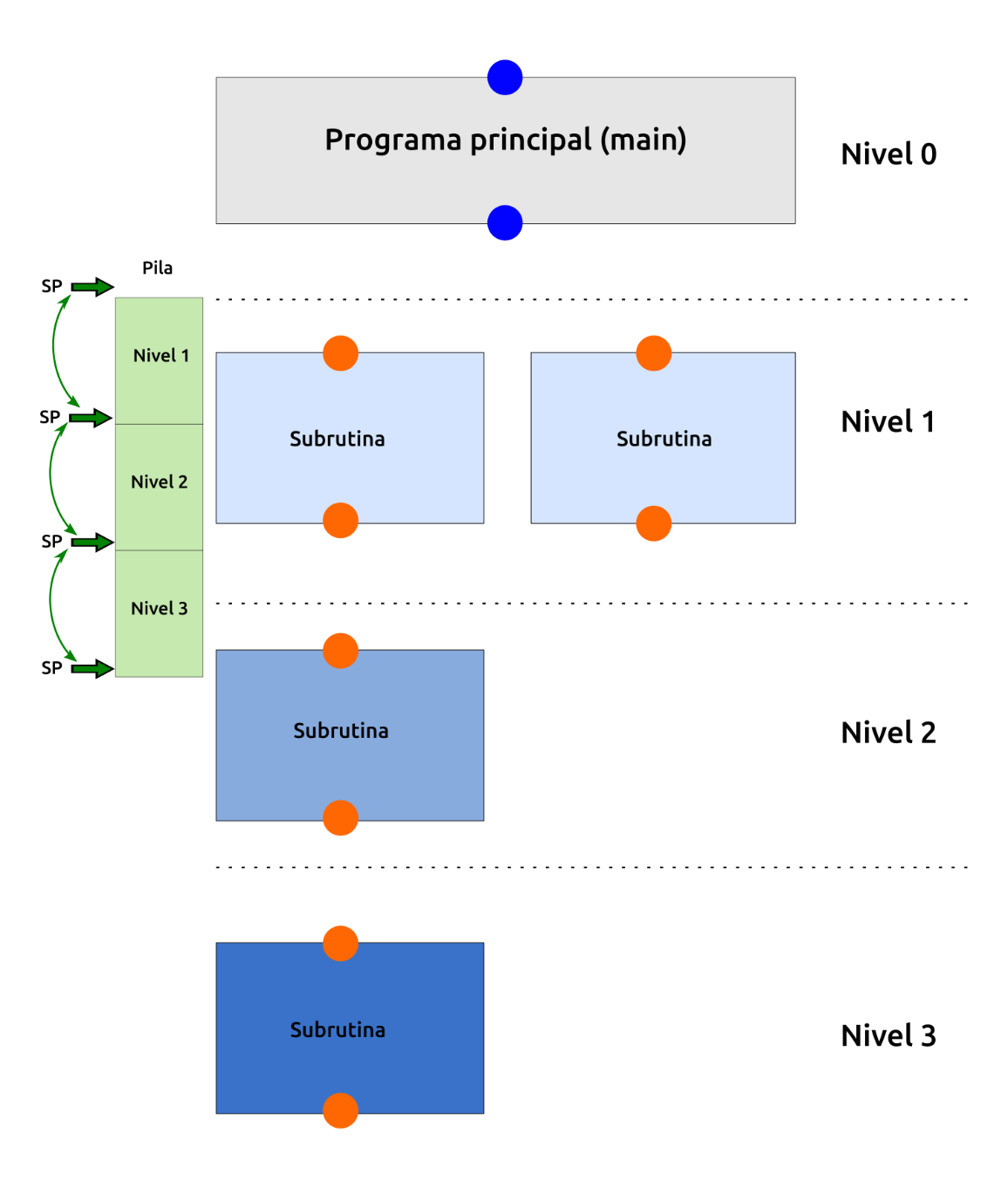
Este **trozo de memoria** asignado a cada subrutina lo podemos visualizar de esta forma:



Sería una posible implementación. Sin embargo, es un gasto excesivo de memoria el asignar una zona a cada subrutina. Si tenemos muchas subrutinas, necesitaríamos mucha memoria

¿Cómo podríamos asignar memoria a cada subrutina, pero gastando menos? La solución es usar una **pila**. Las subrutinas que están en el mismo nivel pueden **reutilizan la memoria**, ya que mientras se ejecuta una subrutina, las otras de su nivel no tienen que almacenar nada

En esta figura está representada esta **idea** de un **trozo de memoria** por cada **nivel** de profundidad



El **puntero de pila** (SP), apunta en un principio a la **cima de la pila**. Se usa como dirección base para el almacenamiento de la subrutina de cada nivel. Así, cuando el programa principal llama a la primera subrutina, esta **decrementa** el puntero de pila y deposita la información que necesita guardar en esa zona. Al llamar al siguiente nivel, este puntero se vuelve a **decrementar** y esa zona es la usada por la subrutina de segundo nivel.

Cuando una subrutina retorna, **incrementa** el puntero de pila para que vuelva a la posición que tenía al invocar a la función. De esta forma, **la memoria se reutiliza** con diferentes subrutinas. Es mucho más eficiente

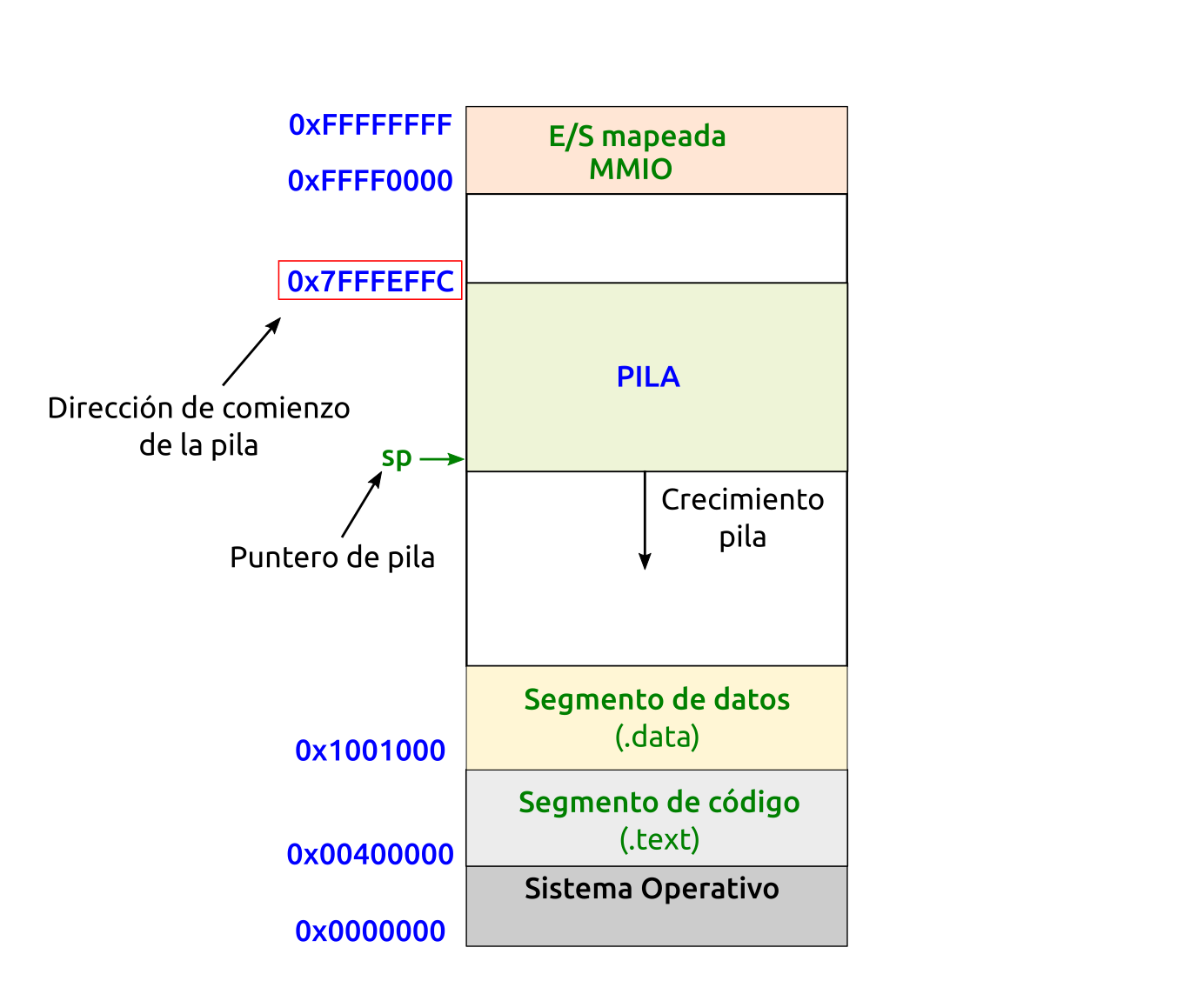
Cada Nivel en la pila es memoria donde se podrán depositar datos. Si por ejemplo estamos en la subrutina 1, los datos que necesitemos guardar, como la dirección de retorno, los podremos depositar en el nivel 1. En caso de estar en la subrutina 2, se libera el nivel 2 y ahora es en esa parte de la memoria donde depositaríamos datos a guardar.

Si no he entendido mal, esta memoria no está reservada, sino que se va reservando dinámicamente a medida que la vayamos solicitando al bajar a una subrutina nueva.

## La pila

La **pila** es una **zona de memoria** disponible para asignar a las **subrutinas**. Cada subrutina tiene **su propia zona de pila** en exclusividad: una zona de memoria sólo para ella, para **almacenar** la **dirección de retorno**, el contenido de los **registros** que se quieren preservar y las **variables locales**

La pila se sitúa en las **zonas altas** de la memoria y crece hacia las direcciones bajas. Su **tamaño es variable**. Inicialmente es de 0 palabras. Cada vez que una subrutina necesita la pila, crea su zona en ella, incrementando el tamaño. Al terminar de ejecutarse la subrutina se recupera el espacio. Este es el **mapa de memoria**:

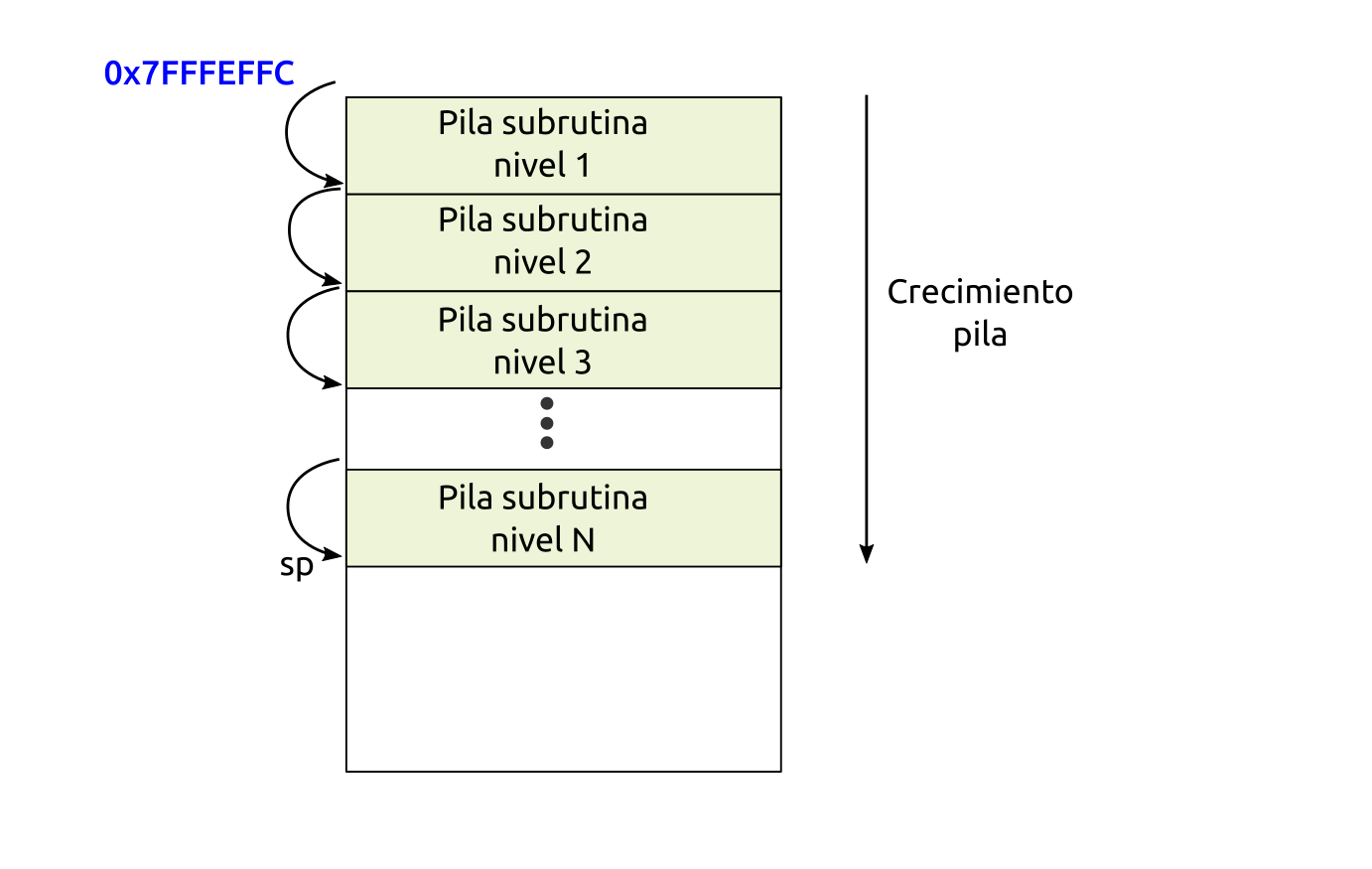


Para **gestionar** la pila se usa el **registro sp** (stack pointer, X2). NO SE PUEDE USAR PARA OTRA COSA. Su uso es exclusivo para la pila. El registro SP apunta al último registro que podemos usar.

### **Creación y liberación de la pila en las subrutinas**

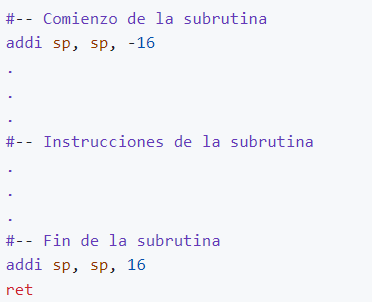
Cada vez que una **subrutina** necesita usar la pila, tiene que reservar espacio. Esto lo hace **decrementando el registro sp** en el **tamaño en bytes** que se quiere para su pila. Por convenio siempre es en **múltiplos de 16 bytes** (4 palabras). Típicamente usaremos pilas de **16 bytes** (4 palabras) ó **32 bytes** (8 palabras)

Así, si hay varias **subrutinas encadenadas**, la pila **crecerá**. Cada zona de la pila es exclusiva de cada subrutina. En este dibujo se muestra el crecimiento de la pila cuando se han **encadenado N subrutinas**



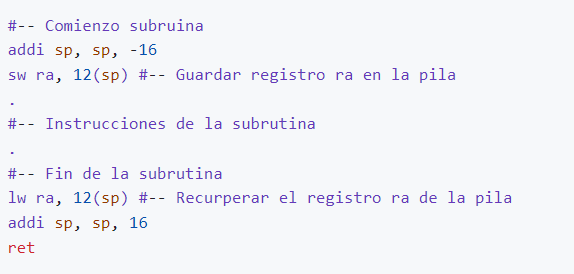
El **esquema de creación** de la pila será siempre el mismo, aunque el **tamaño** se puede variar: Se decrementa el registro SP al comenzar, y se vuelve a su valor original antes de retornar

Este es un ejemplo en el que se **reserva** espacio para **4 palabras** (16 bytes). Pero se podría utilizar cualquier múltiplo de 16: 32, 48, 64.... según el espacio que necesitemos

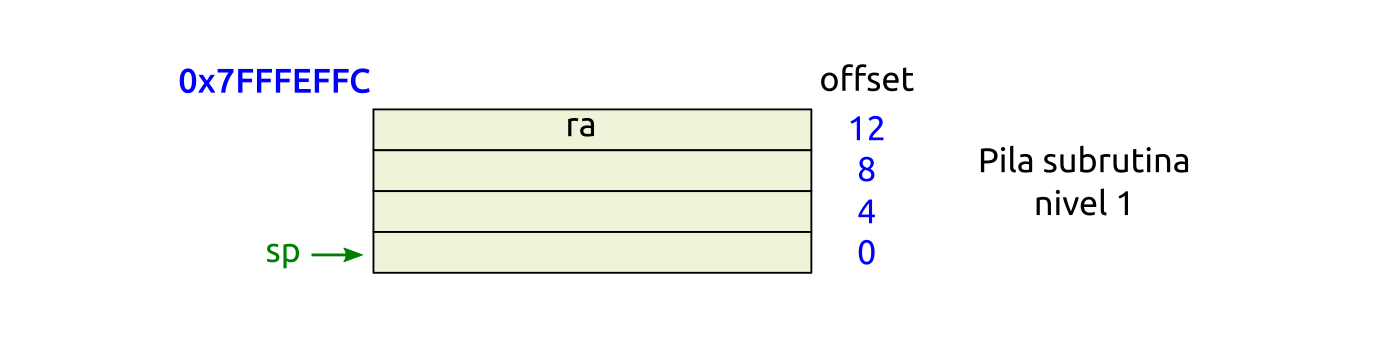


### **Almacenamiento de la dirección de retorno**

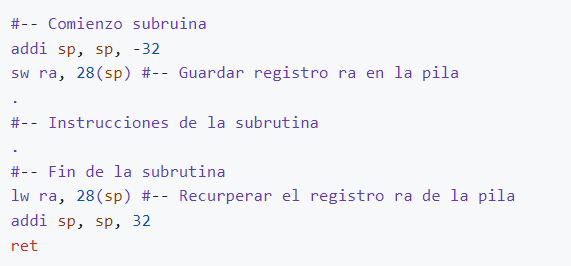
Una vez creada la pila, lo primero que se hace es **almacenar el registro ra**, que contiene la **dirección de retorno** a la subrutina anterior. Antes de terminar la subrutina, se **recupera** este valor y se retorna. Así, el esquema de uso de la pila queda así. Reservamos espacio para 4 palabras:

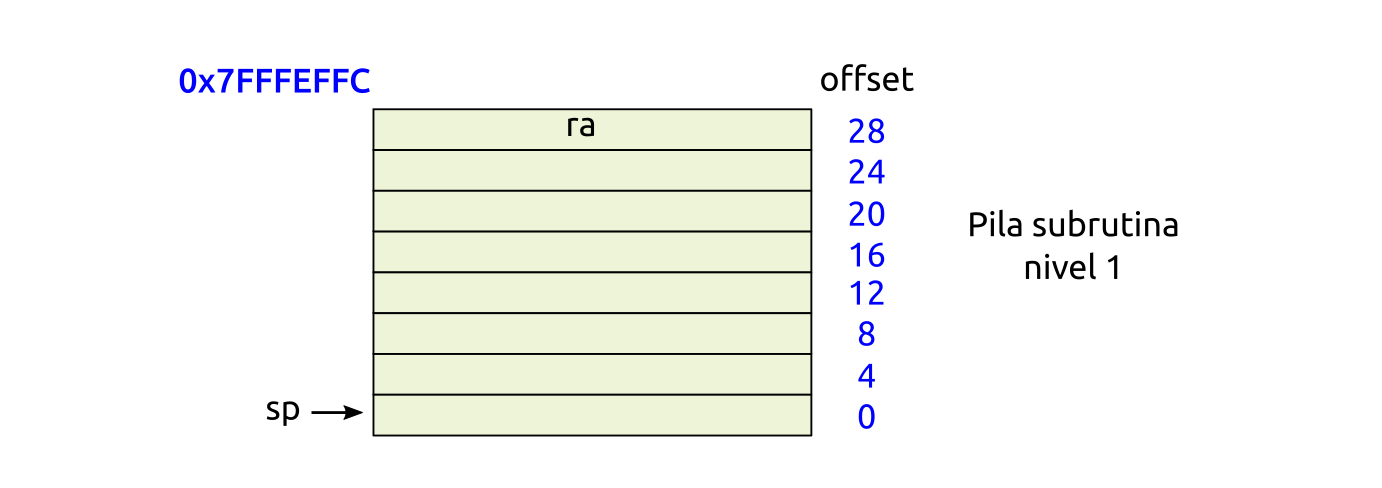


Este es el **aspecto de la pila** de una subrutina de **nivel 1**: tiene espacio para guardar **4 palabras**. En la superior está almacenada la dirección de retorno (registro ra) y el **puntero de pila** apunta a la posición más baja



Si necesitamos una pila de mayor tamaño, usamos unas de **32 bytes**:

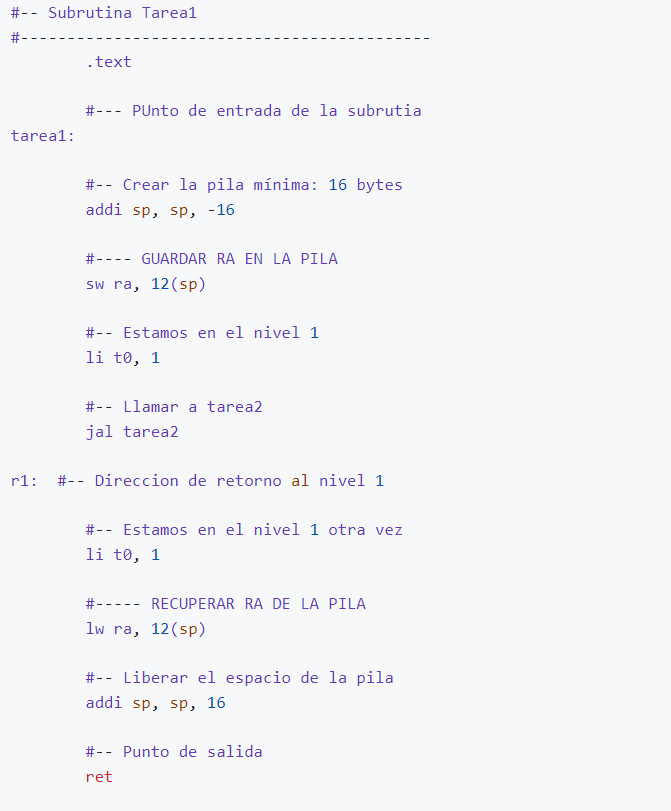


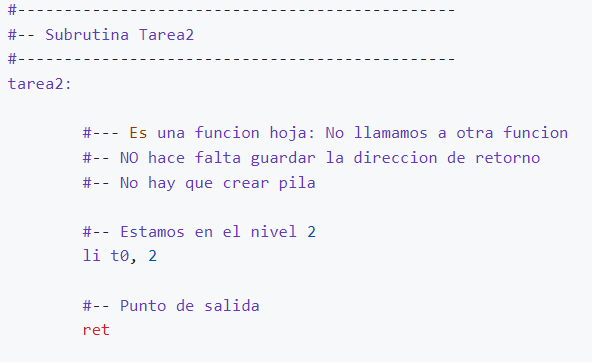
El mapa de los offset es el siguiente. Ahora tenemos espacio para guardar **8 palabras**  


### Ejemplo: Llamada a subrutina del nivel 2

Ahora ya podemos **solucionar el problema** que nos encontramos anteriormente: Cómo invocar a la subrutina de nivel 2. Esta solución es válida para encadenar N subrutinas. Lo que tenemos que hacer es **crear una pila** para la subrutina del **nivel 1**, para almacenar el **registro ra**. NO es necesario crearla para la subrutina de nivel 2, ya que es una **función hoja** y no llama a ninguna otra



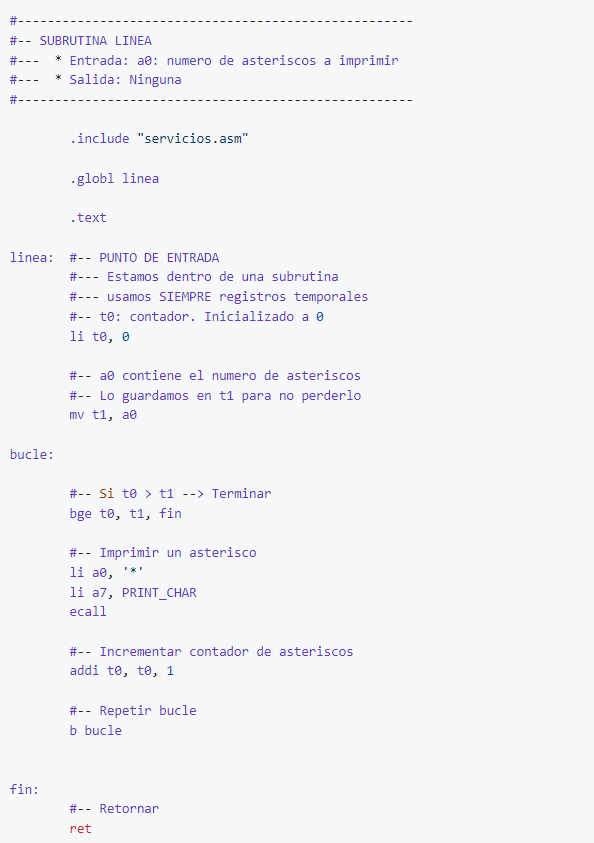
. 



**Ensamblamos** y **Ejecutamos** el programa paso a paso. Ahora ya finaliza correctamente. Retorna sin problemas al programa principal. Ya no hay bucle infinito.

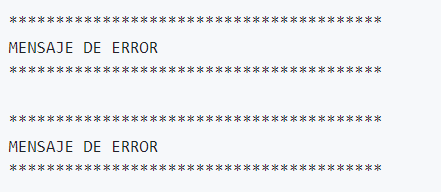
## Ejemplo: Función de impresión de mensaje de error

Partimos de la **función linea(num)** que imprime tantos caracteres **asterisco** (\*) como indique su parámetro **num**. Esta función se propuso en el **ejercicio 2** de la sesión anterior. La subrutina, guardada en su propio fichero, es esta:



Definimos la subrutina void print\_error(pcad) de forma que se **imprima el mensaje** apuntado por su parámetro **pcad**, enmarcado en una línea superior de 40 asteriscos y una línea inferior de también 40 asteriscos. Debe estar en su propio fichero

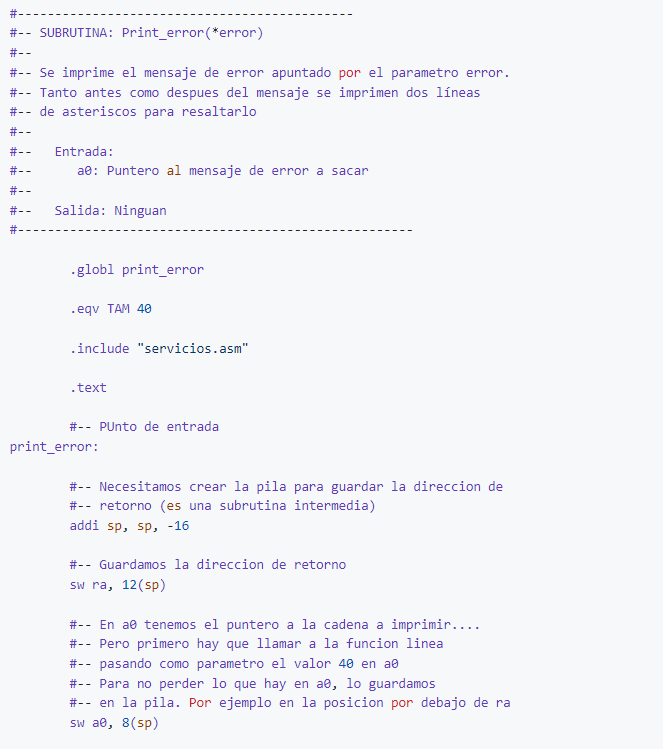
Por último, el programa principal llama dos veces a la subrutina print\_error(), con la cadena "MENSAJE DE ERROR". De esta forma, al ejecutar el programa principal, debe aparecer en la consola lo siguiente:

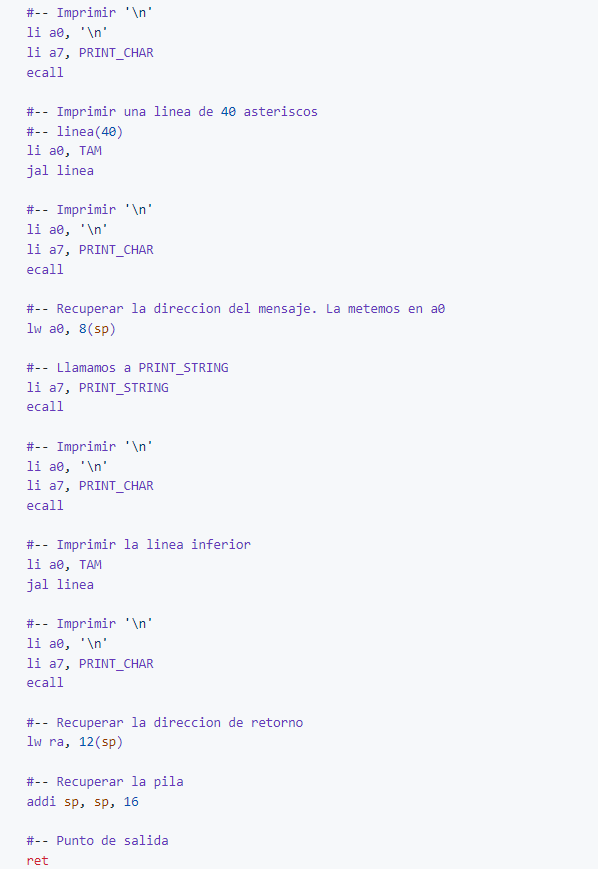


**¿Cómo sería el programa completo?** Cada parte debe estar en un fichero separado

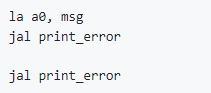
La subrutina **print\_error()** es una **subrutina intermedia**: llama a su vez a otra subrutina. Por tanto, será necesario **crear la pila** para guardar la dirección de retorno. Por el parámetro a0 le pasamos la dirección del mensaje a imprimir. Pero como tenemos que imprimir primero la **línea superior**, hay que usar el registro a0 para indicar a la **función línea** la cantidad de asteriscos a imprimir (que son 40, según me indican en las especificaciones)

Por ello, antes de llamar a linea() es necesario guardar a0 en la pila, o de lo contrario perderíamos su valor. Así, en la pila habrá que guardar la dirección de retorno (ra) y el primer parámetro (a0)



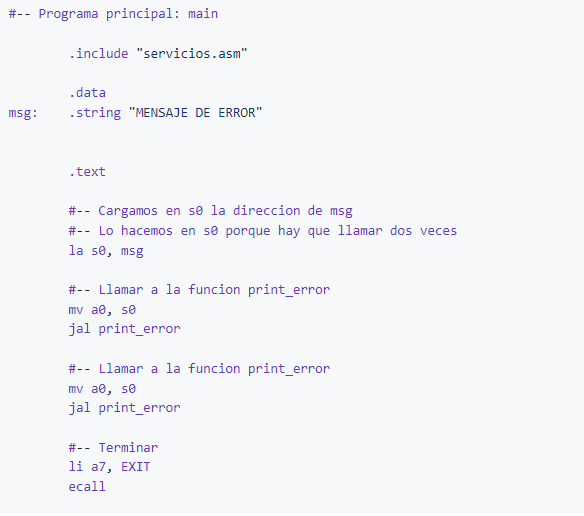


Por último, hacemos el **programa principal**. Hay que llamar **dos veces** a la función print\_error(). PERO.... hay que tener cuidado. Uno estaría tentado de hacer esto:



Si embargo es ¡UN ERROR! ¡Estamos violando el convenio! El registro a0 es temporal. Al llamar a print\_error() la primera vez, tenemos que dar por PERDIDO el valor de a0, porque es un registro que no preserva su valor al llamar a las subrutinas

Para **preservar su valor**, como estamos en el **programa principal**, podemos usar los **registros estáticos**. Por ejemplo, **s0**. Así el programa principal completo será:



Hay que tener un poco de **cuidado** al usar los registros. Por ello hay que **practicar mucho**

## Funciones recursivas

Las [funciones recursivas](https://es.wikipedia.org/wiki/Funci%C3%B3n_recursiva) son las que se llaman a sí mismas para realizar un cálculo. El ejemplo clásico es el cálculo del **factorial** de un número. Si **fact(n)** es la función que calcula el factorial del número n, sabemos que esta función se define de esta forma:

fact(n) = n \* fact(n-1), con n>=2 y fact(1) = 1

Estas funciones las implementamos en el RISC-V usando **subrutinas**. Son un caso particular de las **subrutinas encadenadas** que vimos en la sesión anterior, por lo que tendremos que usar la **pila** para guardar las **direcciones de retorno** y los **registros** que se pasan como argumentos

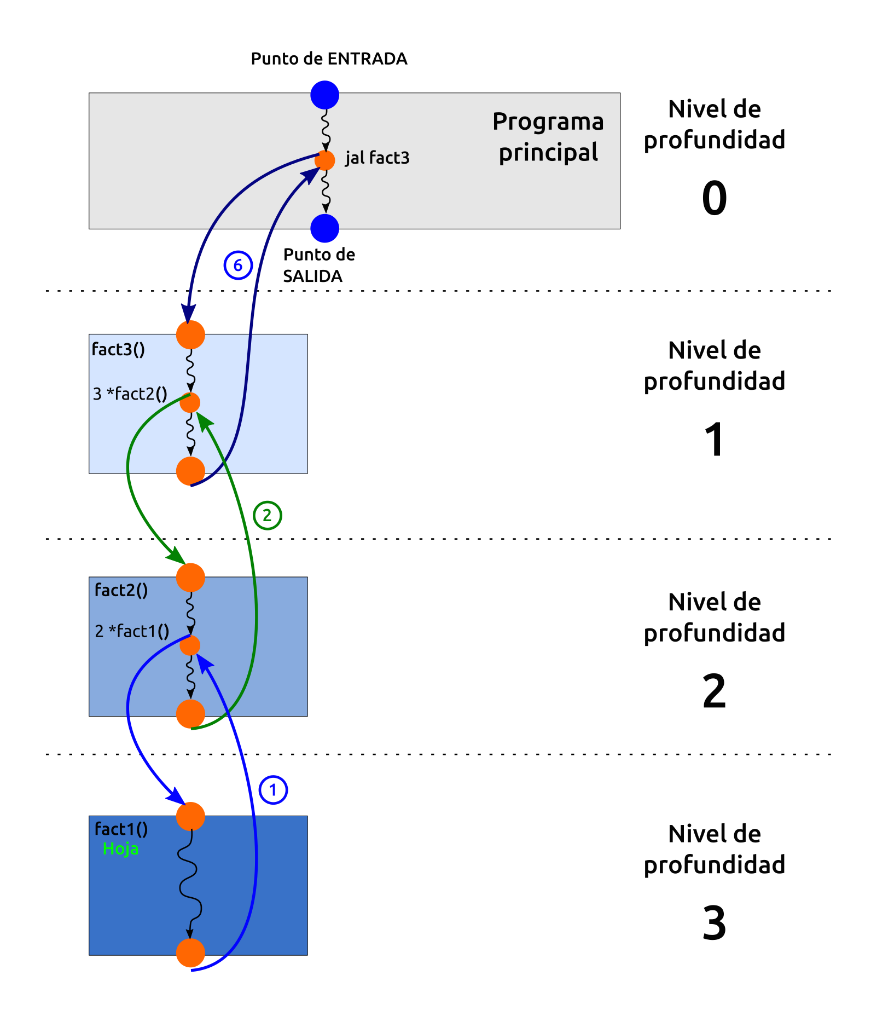
## Factorial de 3 con subrutinas encadenadas

Antes de aprender a hacer **subrutinas recursivas** en ensamblador vamor a hacer un ejemplo del **cálculo del factorial de 3** usando **subrutinas encadenadas**. Así, el programa principal llamará a la **función fact3()** que calcula el factorial de 3. No tiene ningún parámetro de entrada, y devolverá 6

La función fact3(), a su vez, llamará a la **función fact2()** que tiene los mismos parámetros que fact3() pero devuelve el factorial de 2:

Por último, la función fact2() llamará a la **función fact1()**:

La función fact1() devolverá 1, ya que el factorial de 1 es 1. Si representamos los **niveles de profundidad** tenemos lo siguiente:

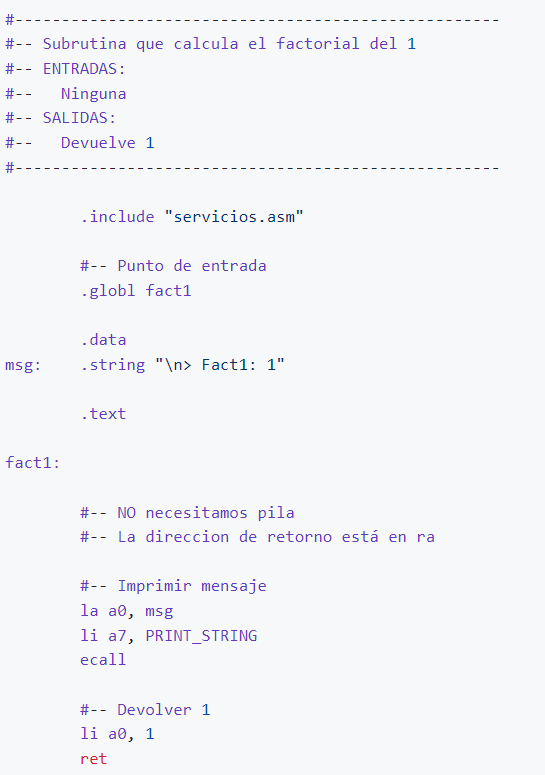


En todos los niveles excepto el último se **multiplica un número constante** por lo que devuelve la **función de nivel inferior**. En el último caso sólo se devuelve la constante 1

Haremos que cada función **imprima un mensaje** para comprobar que se están ejecutando.

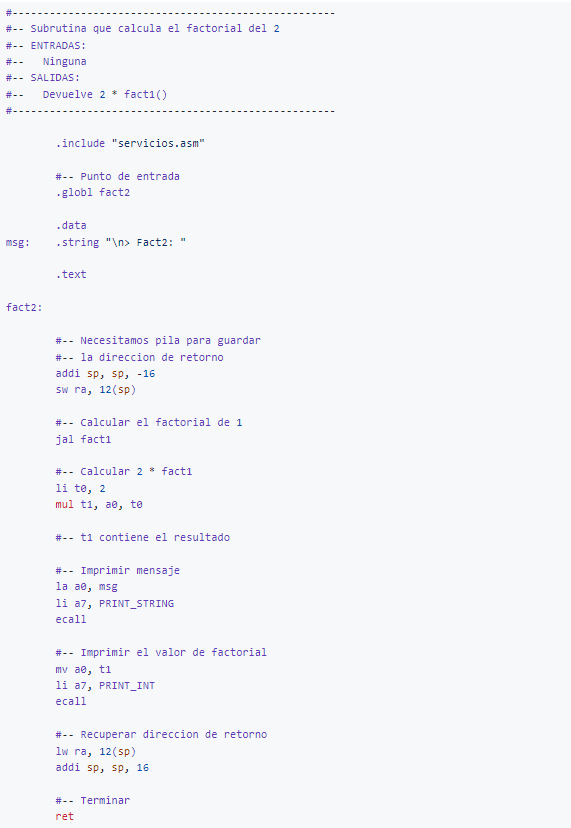
* **Función fact1()**

La **función fact1()** es la última (hoja), y no llama a ninguna otra, por lo que **no será necesario crear una pila** para guardar su **dirección de retorno**. La dirección de retorno está en ra



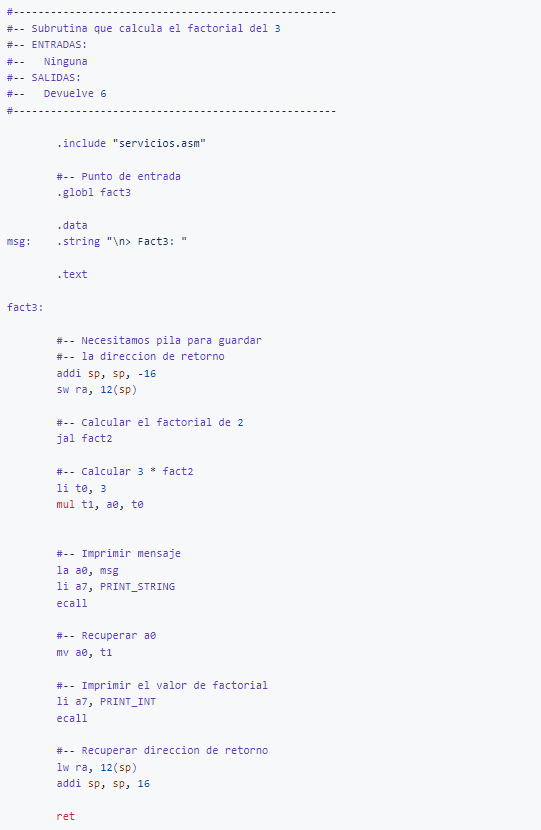
* **Función fact2()**

Se calcula el **factorial de 2** multiplicando 2 por el resultado de llamar a fact1(): 2 \* fact(1). En esta función necesitamos crear la pila para guardar la dirección de retorno. También se imprime un mensaje para saber por dónde va la ejecución



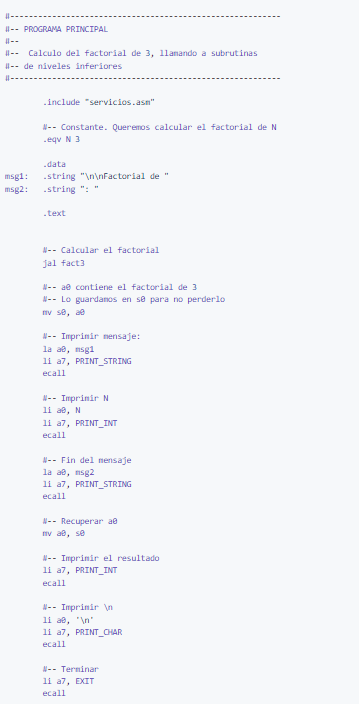
* **Función fact3()**

Igual que fact2(), pero calculando el producto de 3 \* fact2(). También es necesaria la pila para guardar la dirección de retorno



* **Programa principal**

Se llama a fact3() para calcular el factorial. Luego se imprime el resultado



**Factorial de 3 con subrutina recursiva**

Básicamente la subrutina recursiva sería reutilizar las funciones que son prácticamente iguales, para no tener tanto código. En el ejemplo anterior las funciones fact2 y fact3 son prácticamente iguales.

El cálculo del factorial lo podemos simplificar mucho usando una **subrutina recursiva**. Ahora sólo hay una función: fact(), que tiene un argumento de entrada y retorna un valor de salida. El prototipo de la función es:

* **int fact(int n)**

Esta función **se llama a sí misma para realizar el cálculo**, pero en como si la hubiésemos implementado con varias subrutinas encadenadas, como en el ejemplo anterior. Dentro de ella hay que distinguir **dos casos**:

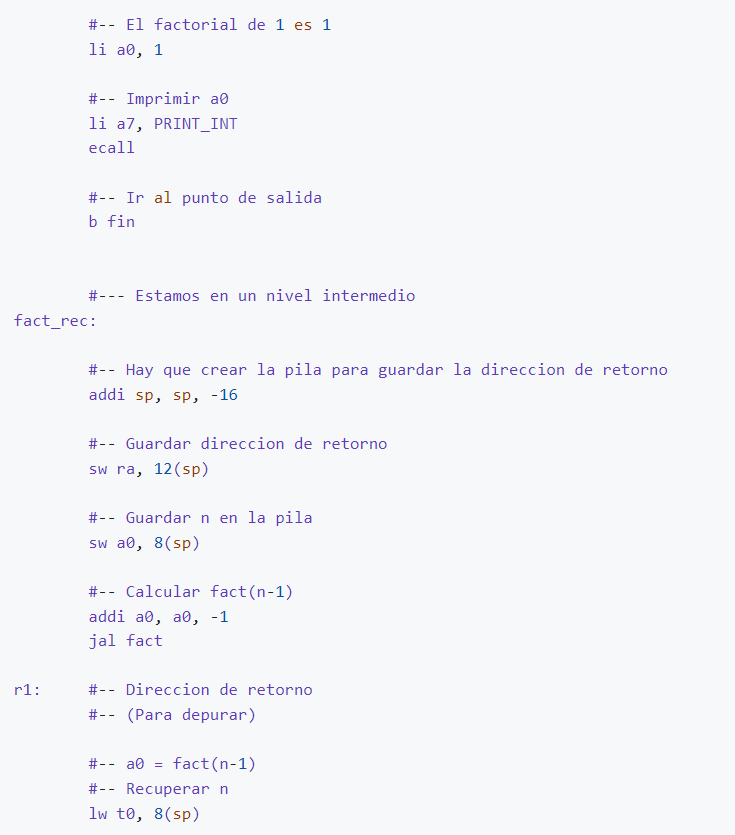
* Si el **n es 1**, entonces la función simplemente deberá devolver el factorial de 1, que es 1
* Si el **n es mayor a 1**, entonces deberá devolver el producto de n por el resultado de calcular fact(n-1): n \* fact(n-1)

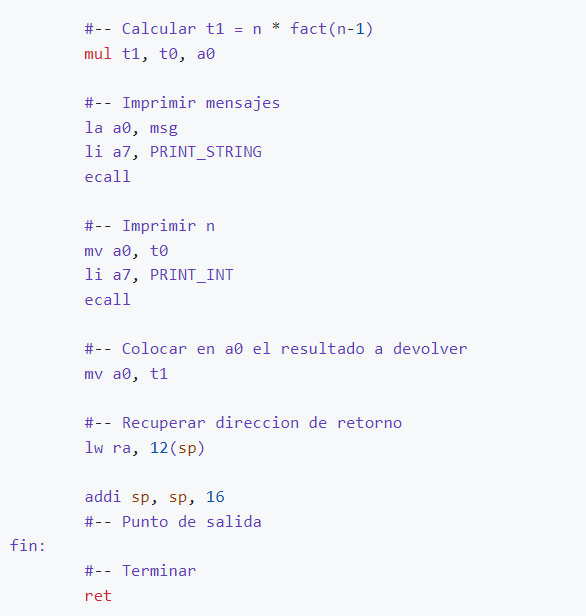
Puesto que la función recibe por **a0** el número n, y tiene que llamar a una subrutina con n-1, será necesario **guardar el valor n** en la **pila**, para no perderlo, y respetar el convenio

Exactamente igual que en el ejemplo anterior, para calcular fact(3) se realizan **dos llamadas** a subrutinas. Una para calcular fact(2) y la última a fact(1)

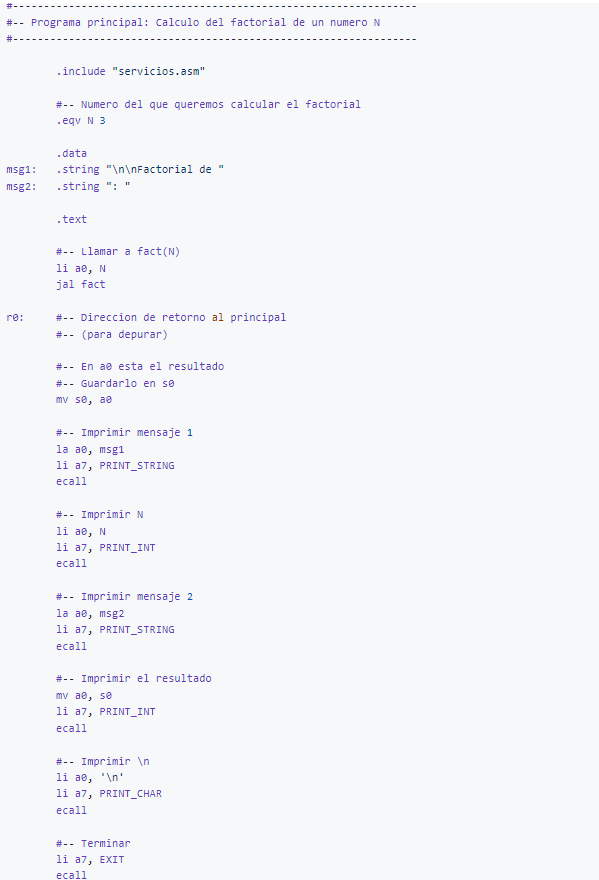
* **Función fact(n)**



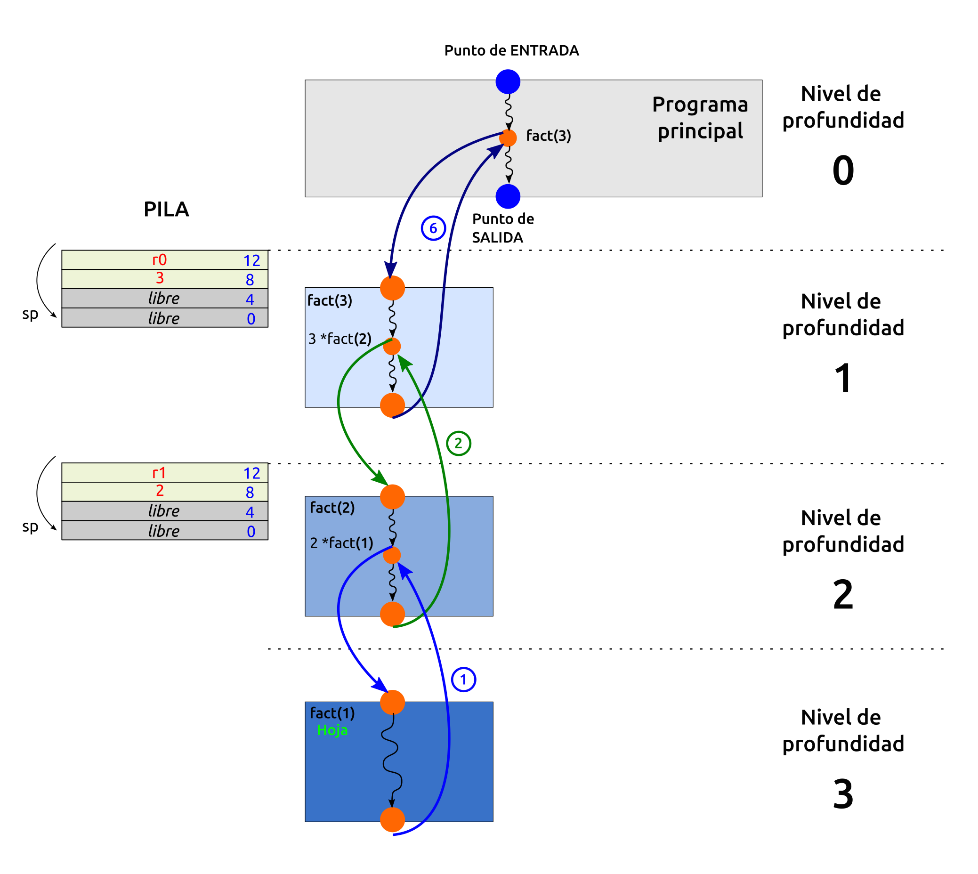




* **Programa principal**:



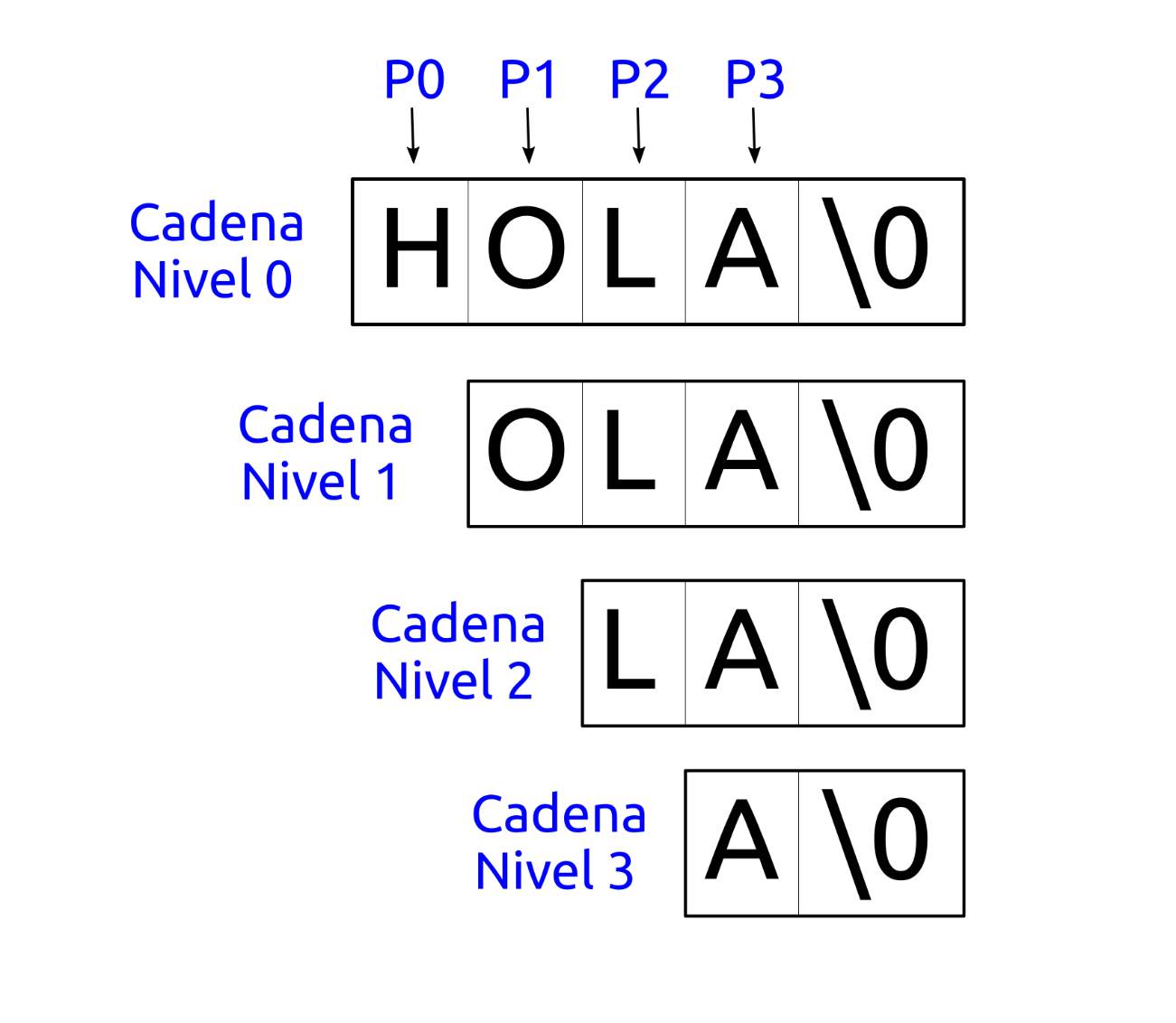
Cuando se **ejecuta** para **N=3**, el comportamiento es exactamente igual que en el ejemplo anterior. En los niveles 1 y 2 se crea **pila** para almacenar las **direcciones de retorno** a los niveles 0 y 1 respectivamente, así como los **argumentos** recibidos: 3 y 2



## Recursividad en cadenas

Las **cadenas de caracteres** que estamos utilizando las definimos como **secuencias de caracteres** que terminan con el carácter **'\0'**. Bien, dentro de cualquier cadena de caracteres encontramos a su vez **más cadenas de caracteres**. Es por tanto un ejemplo de estructura en la que podemos aplicar **funciones recursivas**.

Por ejemplo, si tenemos la **cadena** "Hola", en su interior encontramos la **cadena** "ola", que a su vez tiene la **cadena** "la", que a su vez tiene la cadena "a"



Puesto que para acceder a una cadena usamos el **puntero a su primer carácter**, basta con incrementar este puntero en una unidad para seleccionar la siguiente subcadena

Como las cadenas son **estructuras recursivas**, podemos definir funciones recursivas que trabajen con ellas. Así, por ejemplo, para calcular **la longitud de la cadena**, basta con **sumar 1** a la **longitud de su primera subcadena** (aquella que se obtiene al incrementar su puntero en una unidad)

En el **lenguaje Python**, si cad es una cadena, cad[1:] representa la subcadena formada por todos los caracteres menos el primero. La función recursiva para calcular la longitud de una cadena se puede definir así:



Usamos la propia función len() para implementar la función len()

La **parte recursiva** representa el **caso general**: en el **patrón que se repite** en los diferentes niveles. Siempre hay un **caso particular**, **final**, que es diferente al general, donde la **recursividad deja de existir**. En el ejemplo de la longitud de la cadena es la **cadena nula: '\0'**. En ella ya no hay otra sub-cadena dentro. Por eso **NO es recursiva**. Hay que definir el valor de su longitud, que es 0