CAPÍTULO 2

## **Definiendo constantes: Directiva .eqv**

En nuestros programas tendremos que usar **constantes** numéricas. Por ejemplo, para establecer el valor inicial de un contador, o el de determinadas variables. Estas constantes siempre tienen el **mismo valor** durante toda la ejecución del programa

Es una **buena práctica** de programación el **definir** las constantes al **principio** del programa, de forma que se puedan cambiar fácilmente, sin tener que navegar por el código

Para **definir constantes** usaremos la **directiva .eqv**. Este es un **ejemplo** de uso

Se trata de una **directiva**, por lo que **NO** genera código máquina. Le dice al ensamblador que cada vez que vea el identificador INICIAL lo sustituya por el valor 5, en este caso

### Ejemplo: Contador con valor inicial

Modificaremos el programa del contador para que empiece su cuenta con un **valor inicial**, establecido mediate una **constate**. El programa es:

#-- Contador con valor inicial definido mediante

#-- un identificador

#-- Definir el valor inicial para el contador

.eqv INICIAL 20

.text

#-- Inicializar contador

#-- El ensamblador sustituye el identificador INICIAL

#-- por el numero 20

li x5, INICIAL

bucle:

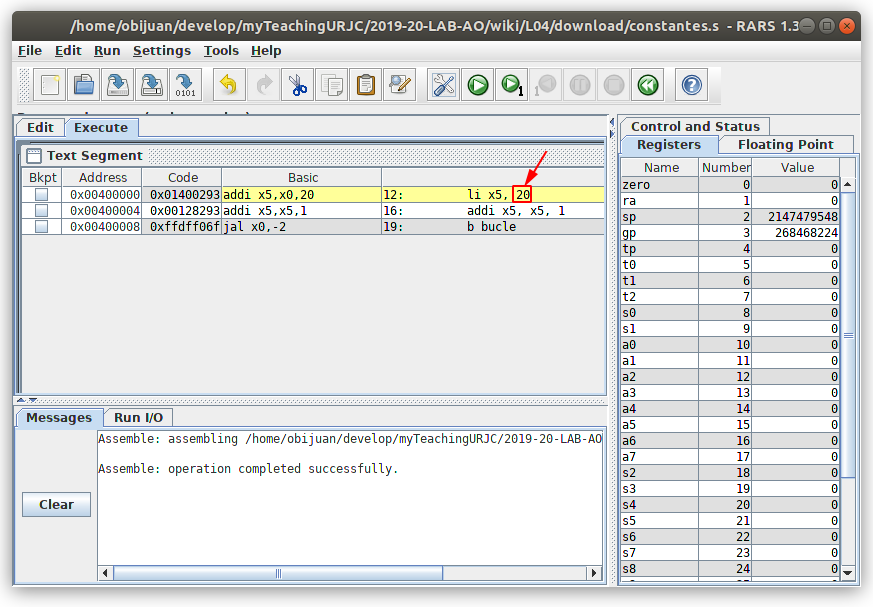
#-- Incrementar contador: x5 = x5 + 1

addi x5, x5, 1

#-- Repetir

b bucle

**Ensamblamos** el programa, y nos fijamos en la ventana del segmeno de código. Vemos que la instrucción de inicilización que aparece es **li x5, 20**



En el editor habíamos especificado **li x5, INICIAL**, pero el ensamblador ha sustituido el identificador INICIAL por su **valor 20**. Es el mismo comportamiento que ocurre con el **#define** del **lenguaje C**

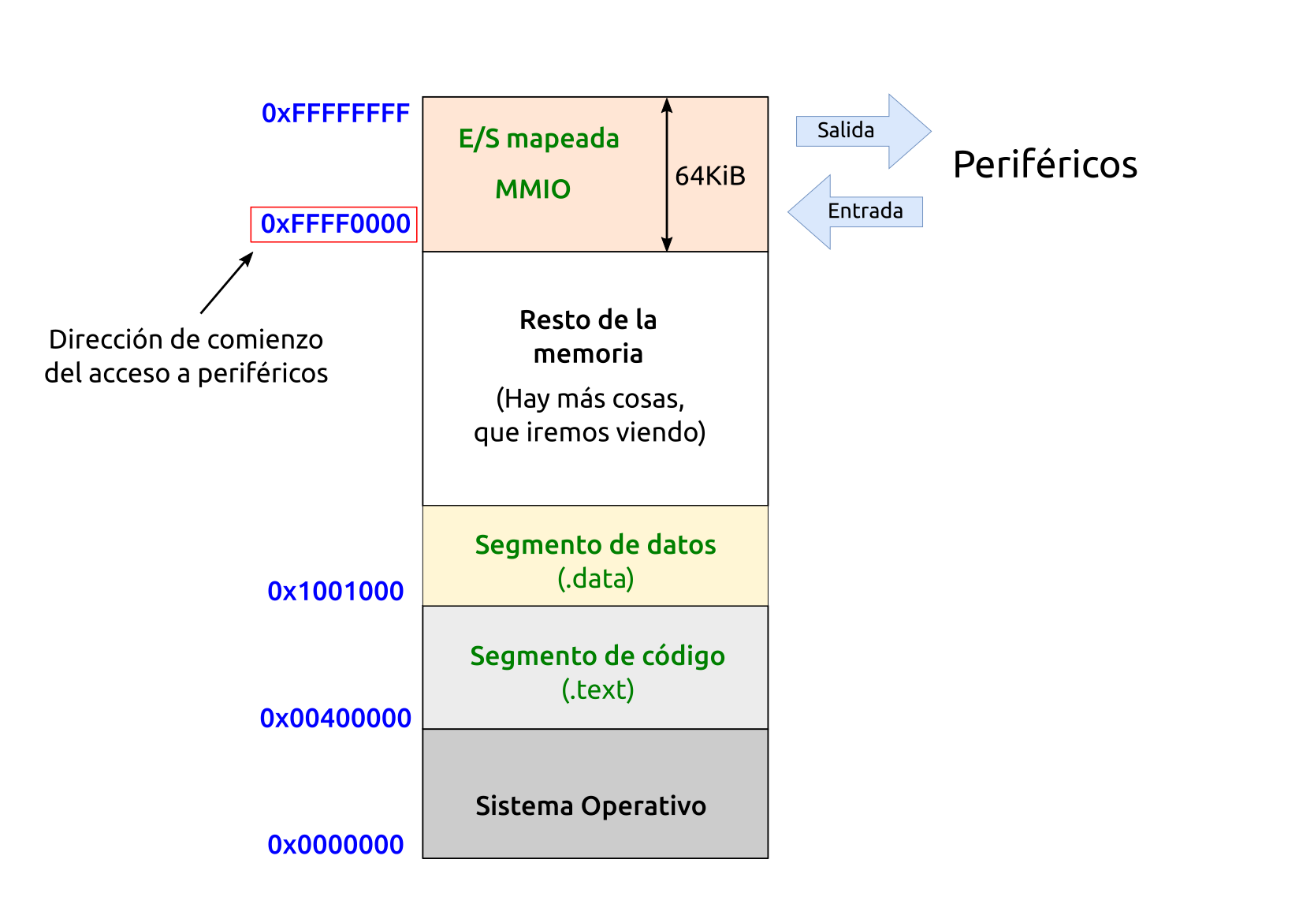
Entradas y salidas de periféricos

**Nuestros programas** se dividen en **código** y **datos**, ambos almacenados en la **memoria**. Cada uno en su segmento: de código o de datos. Las **instrucciones** del programa se almacenan en el **segmento de código**, y las **variables** en el **segmento de datos**. Nos falta una parte muy importante. ¿Cómo se comunica el procesador con el exterior? Necesitamos una forma de realizar la **Entrada/Salida** de datos (E/S)

**Los periféricos se mapean en la memoria, es decir, tienen su entrada y salida conectadas a la memoria y desde el punto de vista del procesador se ve igual que entrar en memoria.**

## **Periféricos mapeados en memoria (MMIO)**

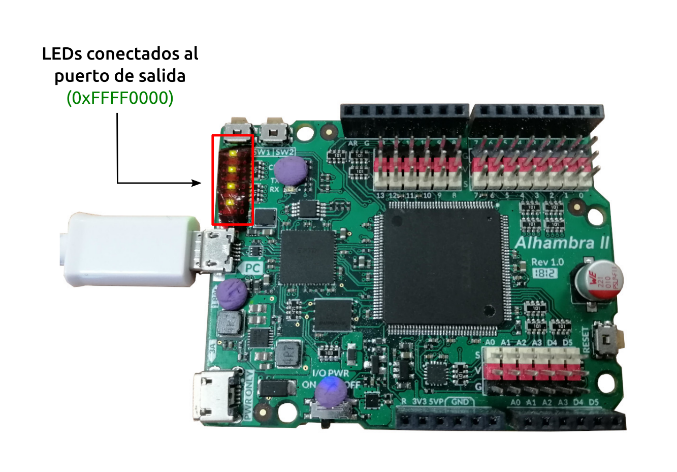
La **zona** del mapa de memoria situada en la parte superior se denomina **MMIO** (Memory mapped I/O): **Entrada/salida mapeada en memoria**. En esas direcciones se encuentran todos los **periféricos** de nuestro sistema. La **comunicación** con ellos se realiza igual que un **acceso a memoria**: Con la instrucción **store** se envía información a los periféricos y con la instrucción **load** se lee de ellos. Pero para el RISC-V es **exactamente igual que cualquier otro acceso a memoria**



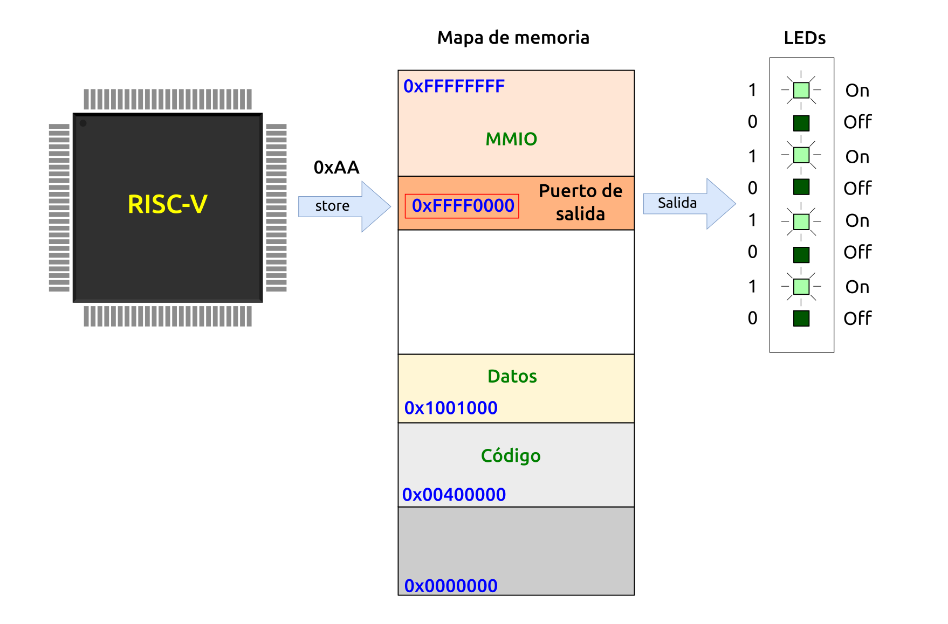
El **acceso** a estas posiciones de memoria es igual que al resto de la memoria: necesitamos colocar en un registro la **dirección** en donde se encuentra el periférico. Usaremos las instrucciones de **load** y **store** para leer o escribir en el periférico respectivamente

### Puerto de salida de 8 bits

Una manera muy utilizada para **enviar información** hacia el **exterior** es mediante los **puertos de salida** de 8 bits (aunque pueden ser de mayor cantidad de bits). Lo que se encuentra **mapeado** es un **registro**. Al hacer un **store** se escribe el valor en este registro, y sus **bits** salen al **exterior** a través de los pines



El **puerto de salida** del [RISC-V que tenemos en la FPGA](https://github.com/Obijuan/RISC-V-FPGA) está en la dirección **0xFFFF0000**, y los **8 LEDs** están conectados a este puerto. De esta forma, al escribir un valor de 8 bits en esa dirección, aparece en binario en los LEDs. Cada **bit a 1** se correponde con un **LED encendido**, y cada **bit a 0** con un **LED apagado**

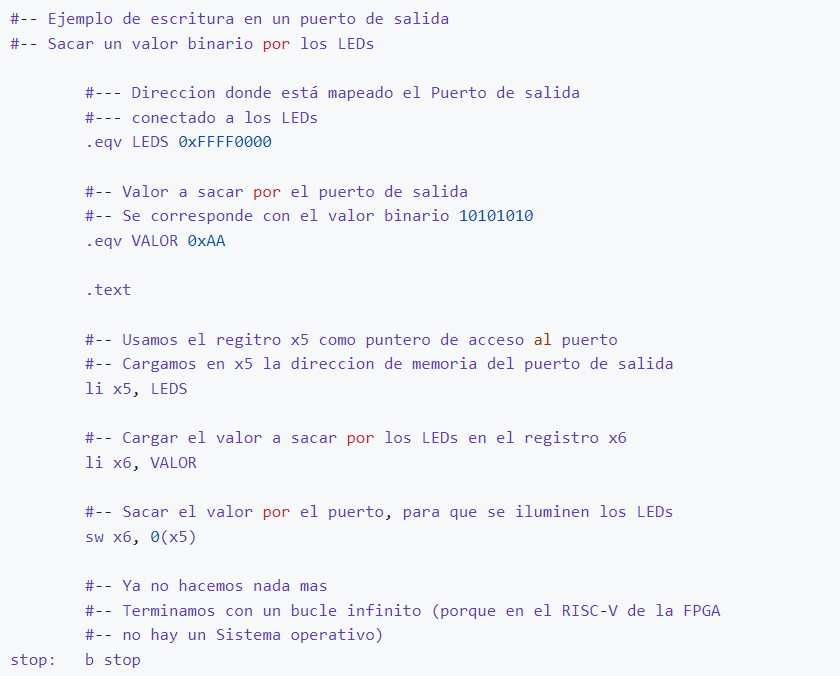


Así, si **escribimos** el valor **0xAA** (que en binario es 10101010) en la dirección **0xFFFF0000** se encenderán **4 leds**, correspondientes a los cuatro unos del valor 0xAA

### Escribiendo un valor en el puerto de salida

Vamos a hacer un programa para **escribir** nuestro primer dato en el puerto de salida. El valor que **típicamente** se utiliza es **0xAA** por tener los bits a 1 y 0 alternados. En el **simulador** simplemente veremos este valor **almacenado** en "la memoria", pero en el **sistema real** comprobaremos que se **encienden los LEDs**

Para **escribir** en este **puerto de salida** necesitamos almacenar en un registro **0xFFFF0000**, que es la **dirección** del puerto de salida. Luego cargamos 0xAA en otro registro y hacemos el **store**



## **Display de 7 segmentos**

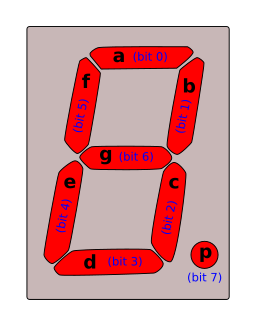
A través del **puerto de salida** hemos controlado **8 LEDs** que estaban colocados en **línea recta**, uno detrás de otro. Un **display de 7 segmentos** está formado por **8 LEDs colocados en forma de dígito**. 7 de los leds se corresponden con los **segmentos** verticales y horizontales y el octavo es el **punto**. Esta es la pinta que tiene el componente

### 

### Nomenclatura

Para dibujar los dígitos necesitamos **nombrar** los diferentes **segmentos**, igual que hemos hecho en los ejemplos anteriores. La nomenclatura estándar es usar las letras **a**, **b**, **c**, **d**, **e**, **f** y **g** para los segmentos. El **punto** lo llamaremos **p**

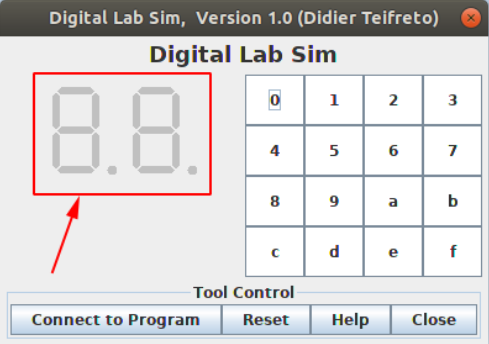
Cada dígito lo controlaremos con un **bit independiente**. El **bit 0** se corresponde con el **segmento a**, el **bit 1** con el **b**,..., el **bit 6** con el **f** y finalmente el **bit 7** (de mayor peso) con el **punto**



### Displays de 7 segmentos en el Simulador

En el **simulador RARs** tenemos acceso a **dos displays de 7 segmentos**, además de un **teclado hexadecimal**. Están accesibles desde la opción **Tools/Digital Lab Sim**

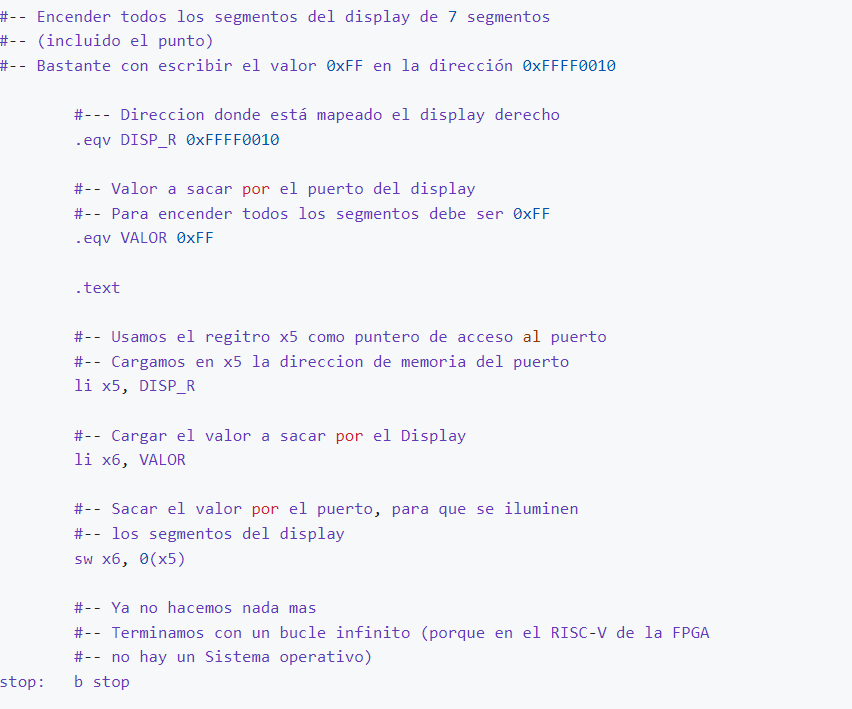
Al abrirlo nos aparecen en la izquierda los **dos displays** de 7 segmentos y en la derecha un **teclado** de 16 teclas



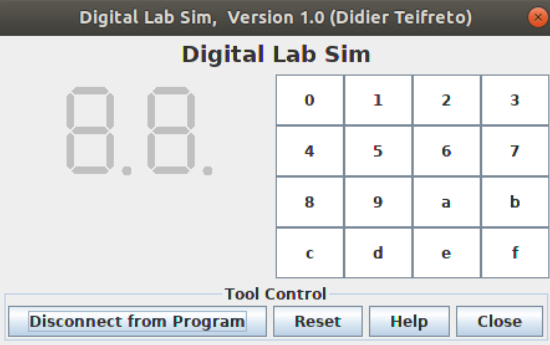
### Encendiendo todos los segmentos de un display

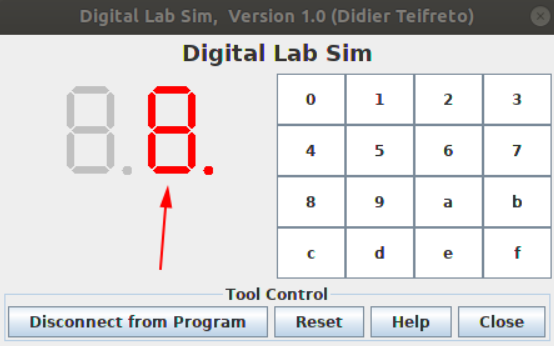
Los **dos diplays**, el derecho y el izquierdo, están **mapeados** en las direcciones **0xFFFF0010** y **0xFFFF0011** respectivamente. Para practicar, utilizaremos el display derecho. Vamos a **encender todos sus segmentos**. Para ello tenemos que **poner a 1** todos sus bits. Basta con **escribir** el valor **0xFF** en la dirección **0xFFFF0010**

El programa es exactamente igual que el que hemos usado para sacar un valor por los LEDs. Pero ahora cambiamos la **dirección** del puerto de salida, y el **valor** a enviar



Para probarlo **ensamblamos** el programa y arrancamos la herramienta de los displays. Apretamos el **botón** que pone **connect to program**



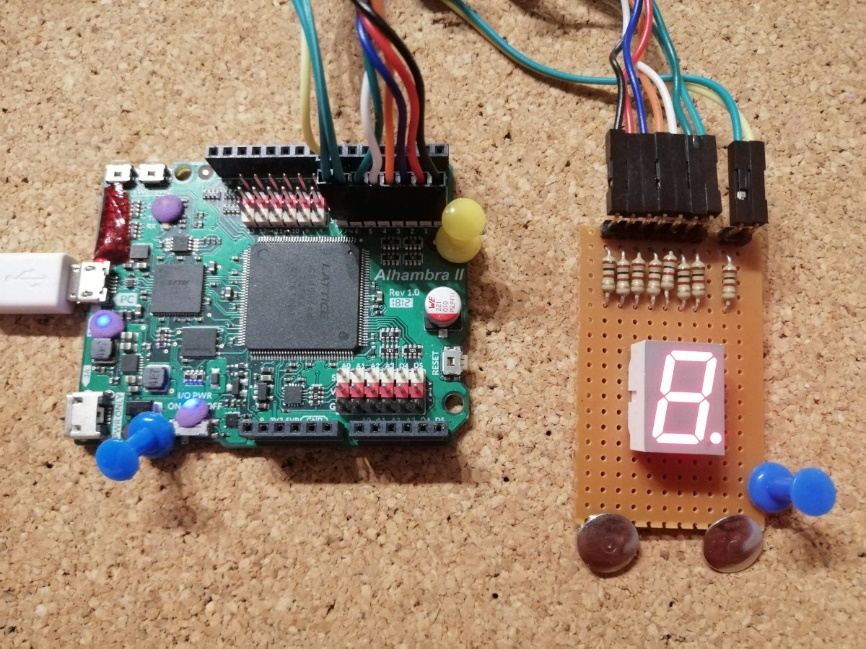


### Probando el display de 7 segmentos real

El programa anterior lo vamos a probar en la **placa real**, donde hay tenemos conectado un display de 7 segmentos. **Ensamblamos** el programa y lo exportamos en **formato binario**. Lo grabamos en la flash con este comando:

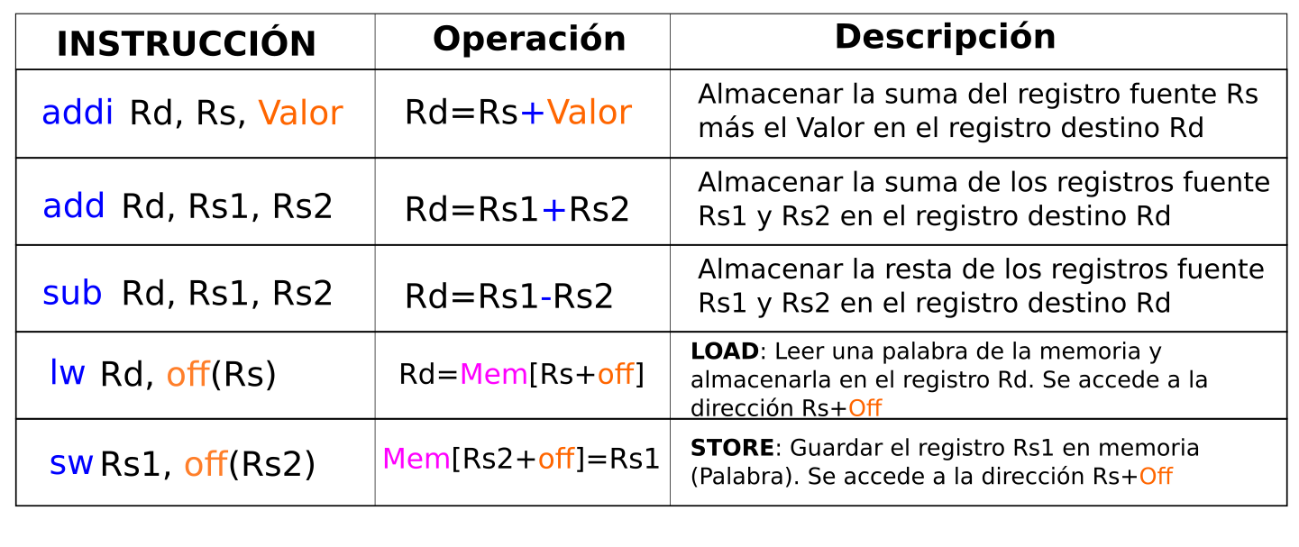
iceprog -o 1M Disp-7seg-all.bin

Veremos cómo se han **encendido todos los segmentos** de nuestro **display real**, incluido el punto. Igual que en el simulador 😄

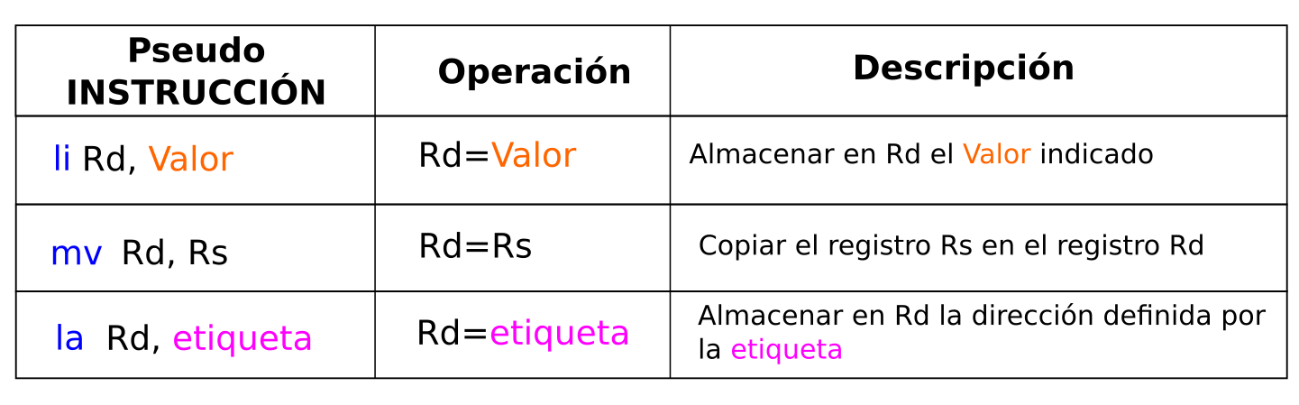


## Recopilación de instrucciones hasta el momento

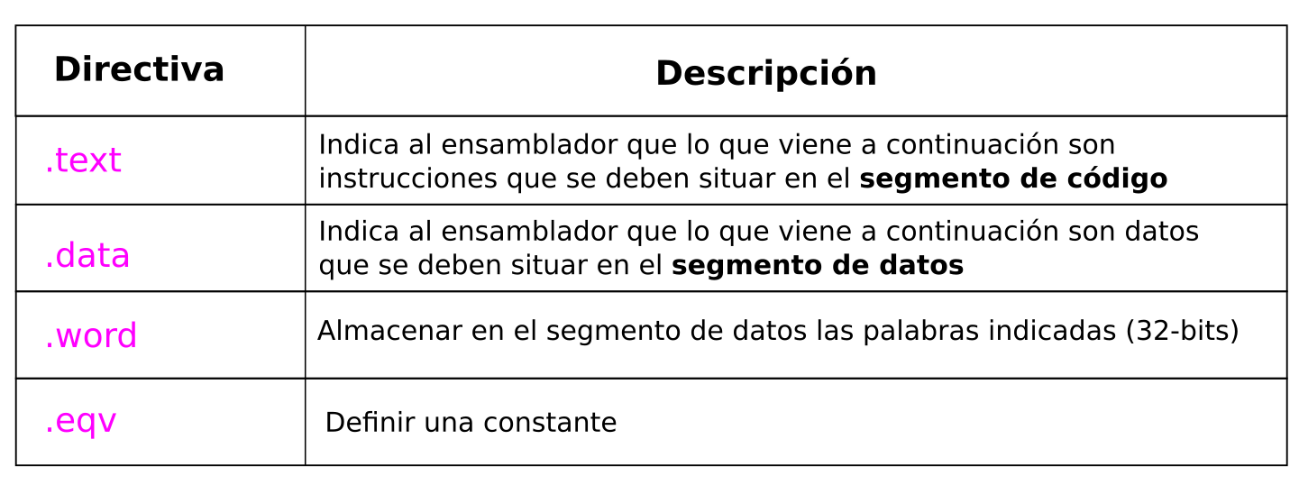
* **Instrucciones básicas**: Son las que se transforman a código máquina y que ejecuta el procesador



* **Pseudo-instrucciones**: No existen realmente como instrucciones. El **ensamblador** las **transforma** en instrucciones básicas. Una pseudo-instrucción puede dar lugar a 1 ó varias instrucciones básicas



* **Directivas**: Dar información al programa ensamblador. No generan código máquina



**Normas ABI**

La **Entrada/salida** se realiza mediante el mecanismo de **dispositivos mapeados en memoria**. Sin embargo, los **sistemas operativos** incorporan servicios para gestionar la entrada/salidas y facilitar su acceso desde nuestros programas. Aprenderemos cómo **invocar** estos servicios. Es lo que denominamos **llamadas al sistema**

## La Interfaz binaria de aplicaciones (ABI) de RISC-V

Nuestros programas no se ejecutan de forma aislada, sino que **interactúan** con el **sistema operativo** y las **bibliotecas del sistema**, que son programas diseñados por otras personas

Para que dos programas (módulos) puedan interactuar entre sí, se definen una [ABI](https://es.wikipedia.org/wiki/Interfaz_binaria_de_aplicaciones): **Interfaz binaria de aplicaciones**. En esta interfaz se determina cómo deben **intercambiar información** estas aplicaciones para que sean compatibles

### Nombre ABI de los registros

Una de las cosas que se especifica en la ABI es el **nombre de los registros**, y el **uso** que se les da. En el RISC-V tenemos **32 registros**, denotados como **x0-x31**. A **nivel de procesador**, todos los registros (salvo el x0), **son iguales**: los podemos utilizar en cualquier instrucción. No hay diferencia a nivel hardware entre ellos

Sin embargo, a **nivel de programación**, a cada **registro** se le asigna un **uso diferente**, y por tanto, un nombre distinto. Esta nueva denominación la llamamos **nombres ABI**. En esta **tabla** se resumen los nombres ABI asignados a los registros

A modo de analogía, se podría decir que la normativa ABI es como la de circulación. Tú si no quieres, puedes no usarla y el programa seguirá funcionando, así como si estás conduciendo y no te quieres parar en un STOP. Pero si todos siguen las normas habrá una mejor confluencia entre programas de distintos creadores.

Básicamente le da a cada registro un nombre y un uso específico.



### Registros en el simulador RARs

En la **columna de la izquierda**, dentro de la ventana de **registros** del simulador RARs, están puestos los **nombres ABI** de los registros. En la columna central se encuentra el número de registro real: x0,x2,x3... omitiendo la x.



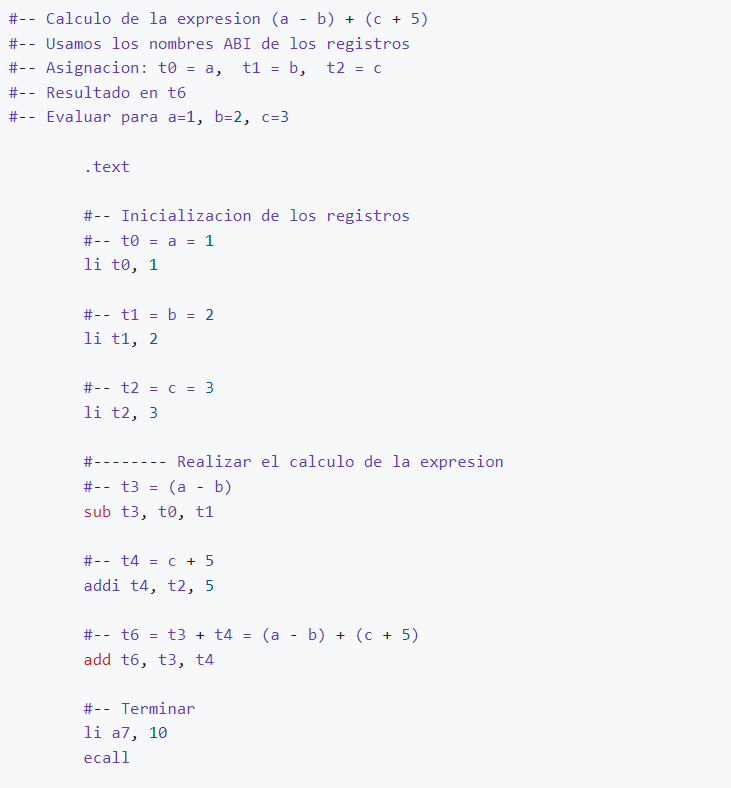
A partir de ahora, **SIEMPRE USAREMOS LOS NOMBRES ABI**

### Ejemplo: Cálculo de una expresión

Como ejemplo vamos a **calcular** la expresión (a - b) + (c + 5) para los valores iniciales de a=1, b=2 y c=3, pero usando los registros con sus nombres ABI

Usamos los **registros temporales**: t0 - t6. Podemos usar cualquier de ellos para almacenar los valores de a, b y c. Por ejemplo: t0=a, t1=b y t2=c. El resultado final lo almacenamos en **t6**

El **programa** para evaluar la expresión es:



## **Llamadas al sistema**

Hemos visto cómo **nuestros programas** se comunican con los **periféricos** mediante **entrada/salida mapeada**. Esto es típico de los sistemas empotrados. Sin embargo, la mayoría de las veces existe un **sistema operativo** debajo que se encarga de la **gestión** de toda la **entrada/salida**, no teniendo que implementarla nosotros.

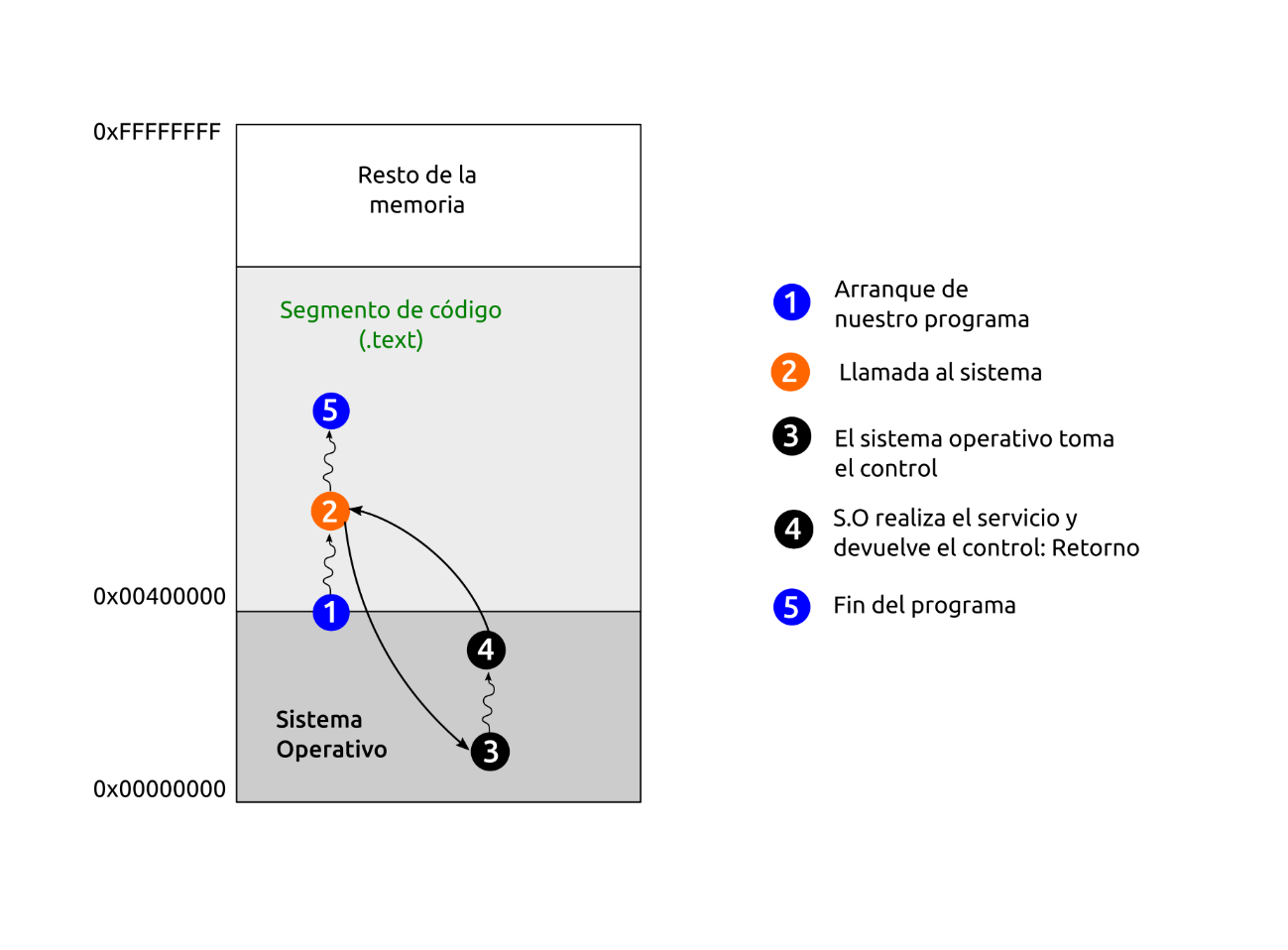
El **mecanismo** que tienen nuestros programas para utilizar **servicios** proporcionados por el **sistema operativo** se denomina **llamadas al sistema**

Esta idea está esquematizada en la siguiente **figura**. Una vez que nuestro programa se **empieza a ejecutar**, en cualquier momento se puede realizar una **llamada al sistema** para **acceder** a uno de los servicios que ofrece, como por ejemplo imprimir un número en la consola, o leerlo de ella



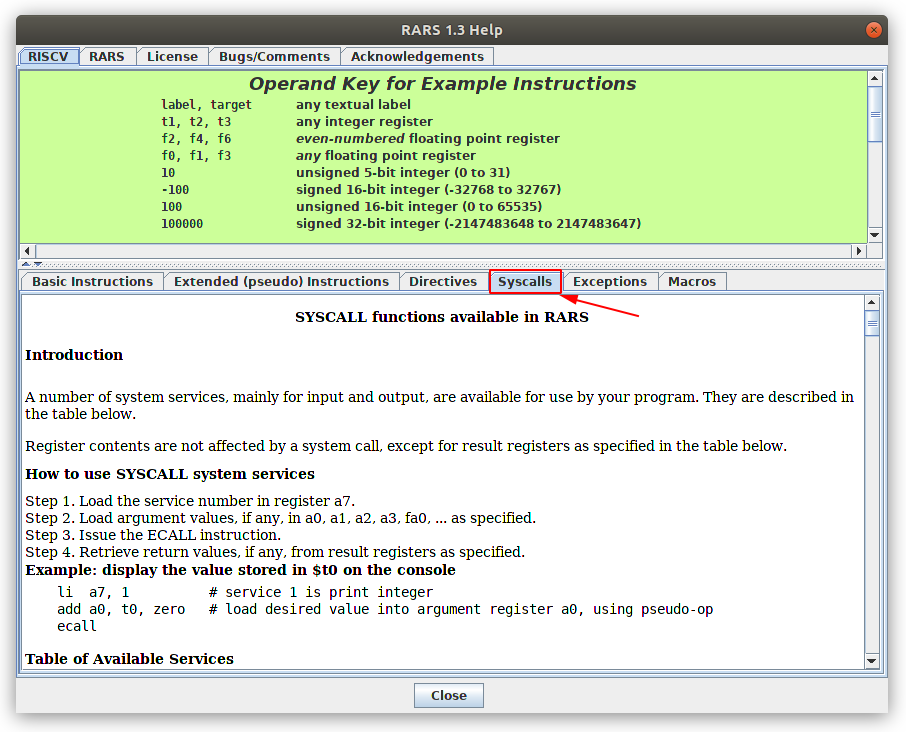
En ese momento, el **control** lo toma el **sistema operativo**, que realiza la acción pedida. Una vez finalizada, **devuelve el contro**l a nuestro **programa**, que se sigue ejecutando hasta que se **termina**

El **sistema operativo**, en nuestro simulador, se encuentra en las **direcciones bajas**, mientras que nuestro **programa** está en el **segmento de código**. El esquema anterior lo podemos representar también de esta forma:



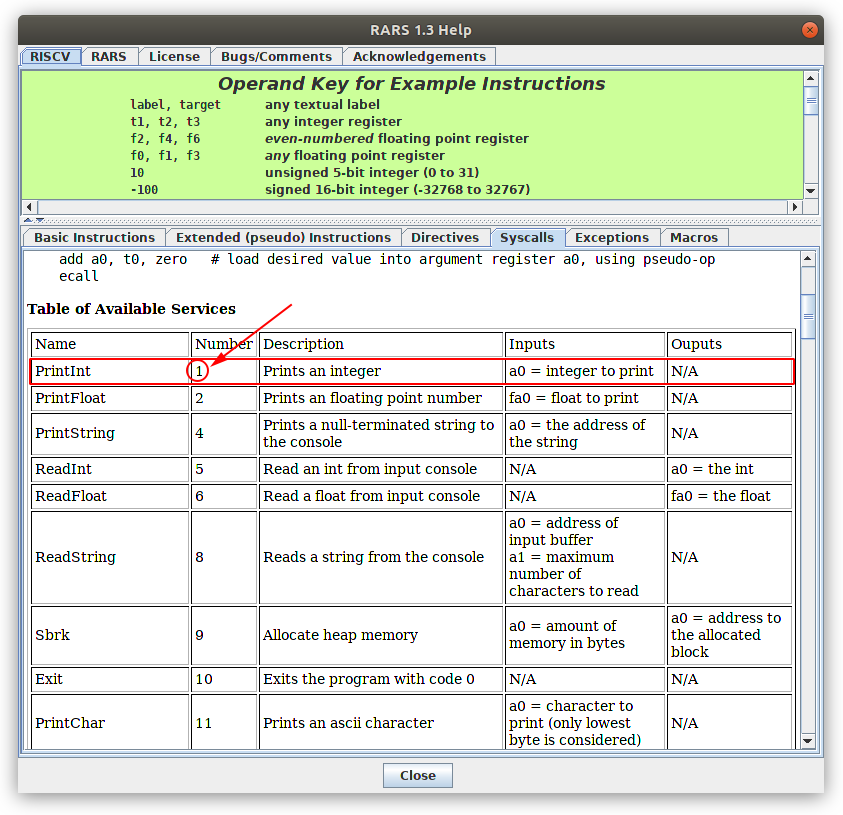
### Llamando al sistema: ecall

Para realizar una **llamada al sistema** utilizamos la instrucción **ecall**. Desde el **menú de ayuda** (Tecla F1) tenemos acceso a todas las llamadas al sistema disponibles en el simulador RARs. Están en la pestaña **syscalls**



Cada **servicio** proporcionado por el sistema operativo tiene asignado **un número**, que debemos introducir en el **registro a7**. Además, si el servicio necesita de unos parámetros de entrada, se los pasaremos a través de los registros a0, a1 y a2. Los valores devueltos por el servicio estarán en a0 y a1. Toda esta información está disponible en la ayuda. La primera columna indica el **Nombre del servicio** y la segunda su **código**

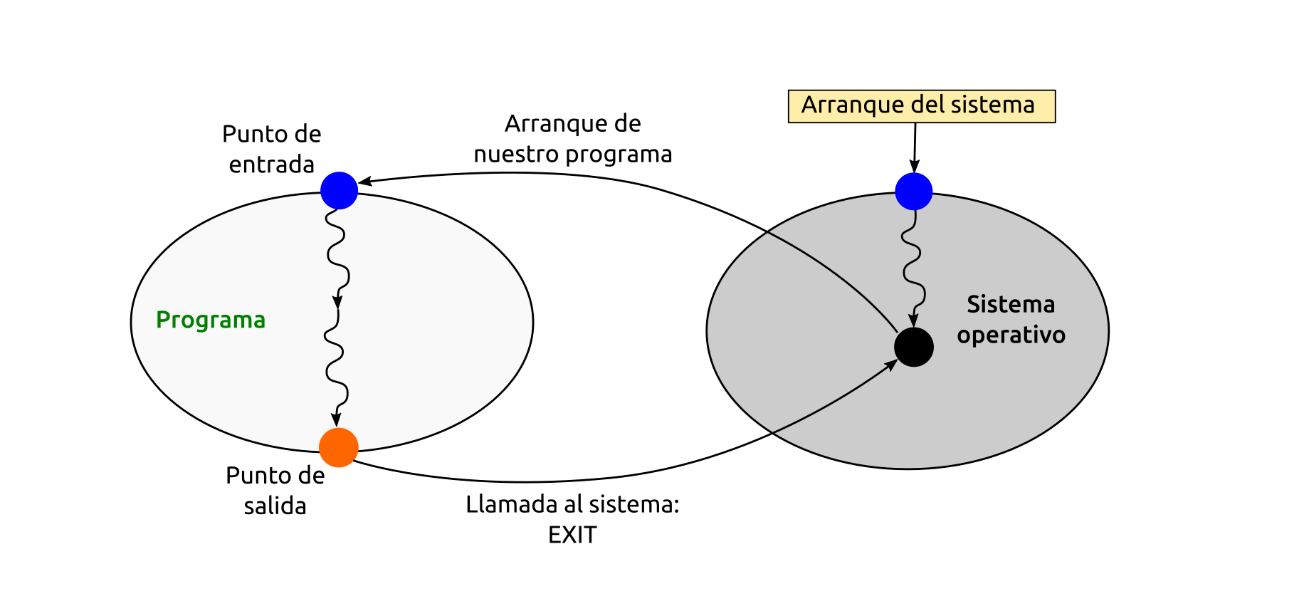
Así, por ejemplo, el primer servicio mostrado en la ayuda es la llamada al sistema **prinInt**, cuyo código es **1**. Nos permite **imprimir un número entero en la consola**. Como **parámetro de entrada** hay que introducir el **número a imprimir** en el **registro a0**. El servicio no devuelve ningún valor. En las siguientes secciones veremos **ejemplos** de uso



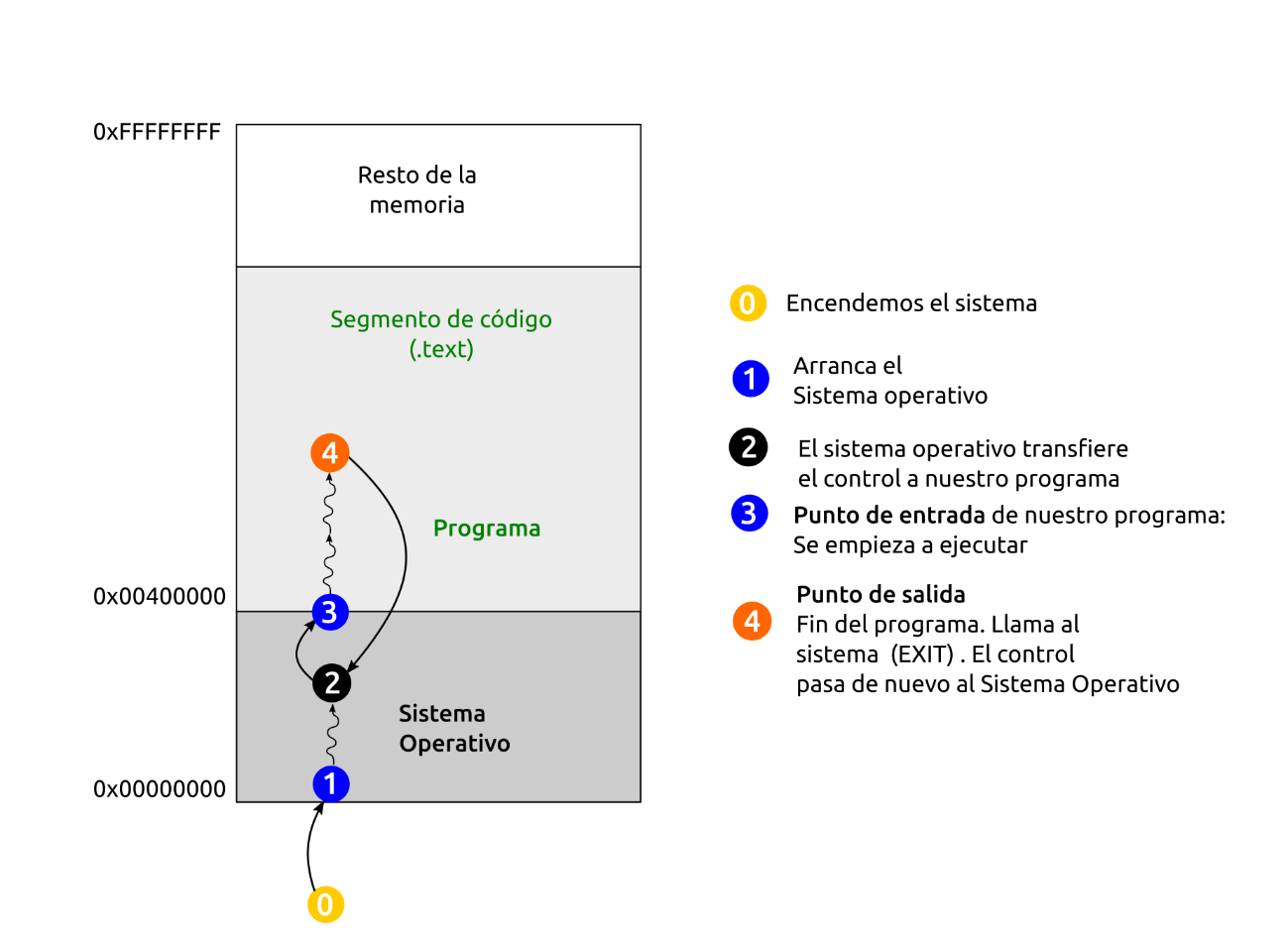
**Puntos de entrada y salida del programa principal**

Vamos a entender qué es lo que ocurre en un **sistema RISC-V** desde que lo **encendemos**. Al darle alimentación, el **contador de programa** se inicializa con la dirección de la **primera instrucción** del **sistema operativo**. Es decir, que es el sistema operativo el que **toma el control**

Típicamente realiza la **inicialización** y se queda **esperando**. En un momento determinado, le decimos que **arranque** nuestro **programa**, por lo que le pasa el **control**, y comienza su ejecución. Nuestro programa tiene una dirección, llamada **punto de entrada**, que es donde se encuentra **la primera instrucción**. De forma **genérica**, lo esquematizamos así:



Sabemos que en el RARs el **punto de entrada** está en la dirección **0x00400000**. Una vez que el sistema operativo nos pasa el control, nuestro programa realiza los cálculos que sean y **termina**. Esta operación de "terminación" significa que hay que **devolver el control** otra vez al **sistema operativo**. Para ello hay que **invocar** la llamada al sistema **EXIT**. En esta figura se muestra el mismo esquema anterior, pero particularizado para el **mapa de memoria del RARS**



El lugar donde se invoca el **servicio exit** se denomina **punto de salida**

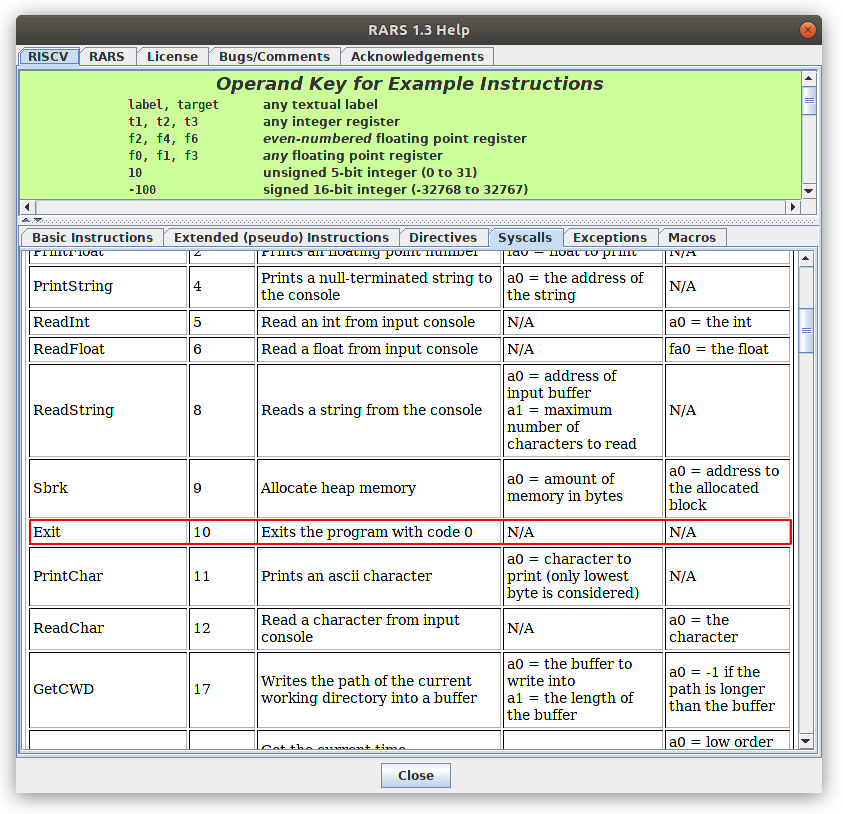
Lo siguiente es una **regla de buenas prácticas de programación**:

* El **programa principal** deberá tener un **único punto de entrada** y un **único punto de salida**

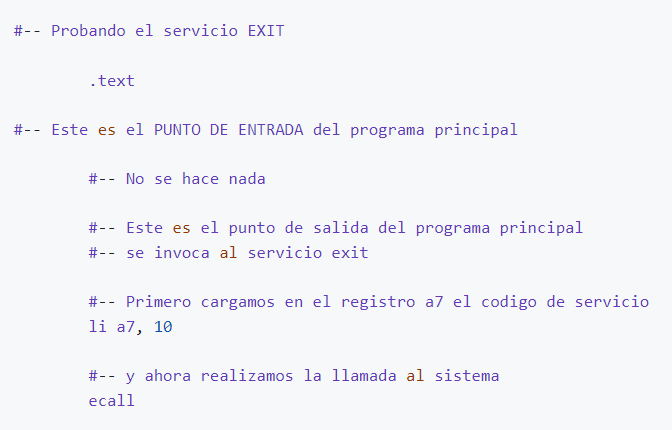
### Terminando un programa: llamada al sistema EXIT

Vamos a probar **nuestra primera llamada al sistema**, que será el **servicio de EXIT**. Todos nuestros programas, a partir de ahora, acabarán invocando este servicio. La única es excepción será en el **sistema RISC-V real en la FPGA**, ya que en él no tenemos sistema operativo. Por lo que terminaremos con un **bucle infinito** (aunque en el laboratorio sólo trabajaremos con el simulador)

Si miramos el **servicio de EXIT** en la **ayuda**, vemos que tiene el **código 10**, y que no hay que pasarle ningún argumento (ni de entrada ni de salida):

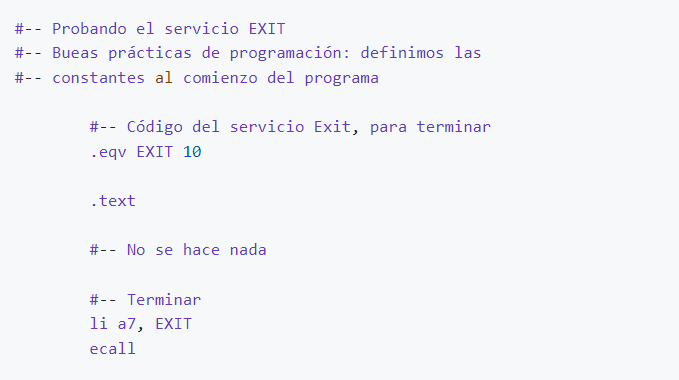


En este **programa** estamos invocando el **servicio Exit**. Es un programa que no hace nada: en cuanto recibe el control del sistema operativo, lo devuelve otra vez (termina)



¿Podríamos mejorar este programa? ¿Ves algo que te llame la atención?

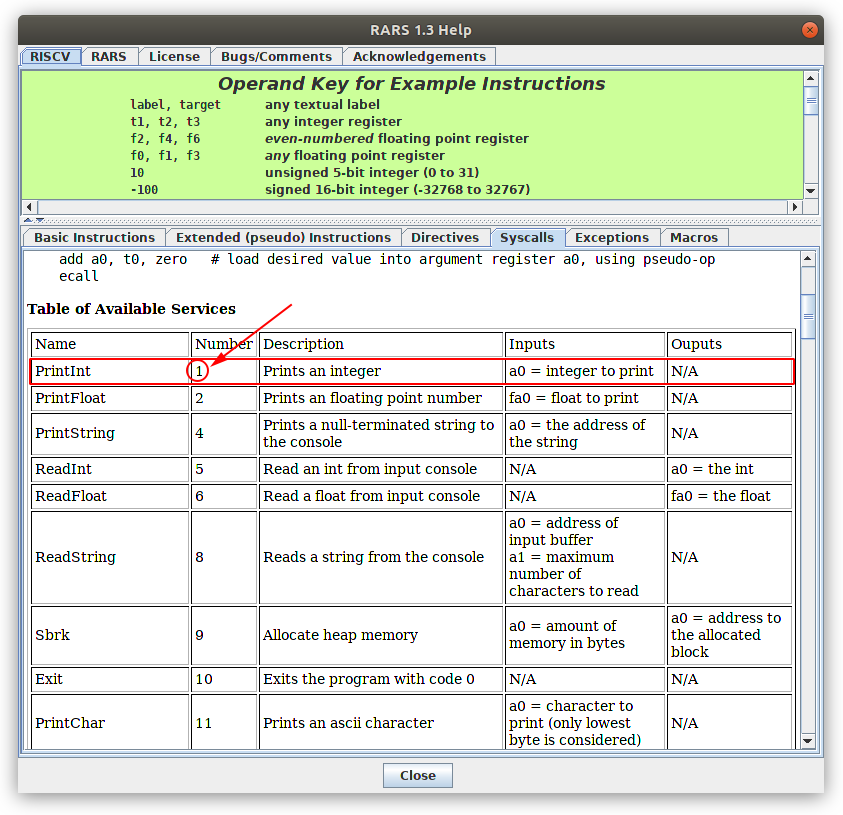
¡¡Si!!! ¡Hay un **número mágico**! El **10**, que se corresponde con el código del **servicio de exit**. Siguiendo las **buenas prácticas** de programación, conviene usar **constantes** para estos números mágicos. Esta es la nueva versión del programa:



### PrintInt: Imprimiendo un número entero en la consola

El servicio de impresión de un número decimal en la consola: **PrintInt**.

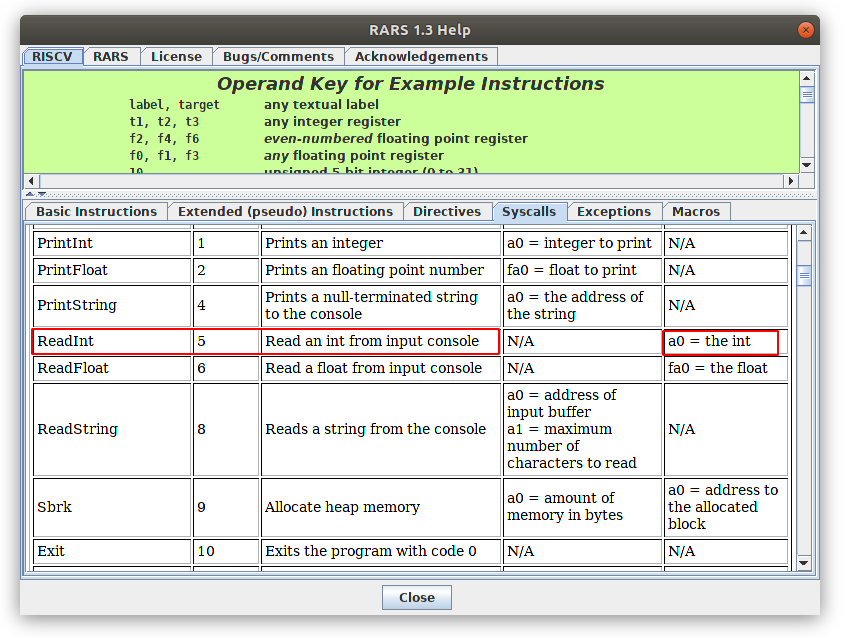
Primero miramos los **parámetros** que tiene el **servicio PrintInt**, así como su **código de servicio**:



Esto nos dice que la llamada al sistema se hace con el 01 y el número a imprimir que hay que pasarle lo tenemos que meter en a0.

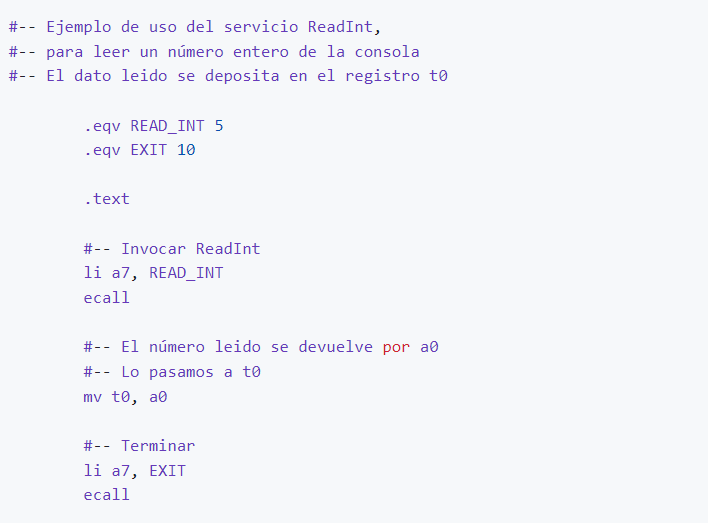
### ReadInt: Leyendo un número entero de la consola

Servicio ReadInt, nos permite leer un número entero desde la consola. Primero miramos la documentación, para ver el código y los argumentos

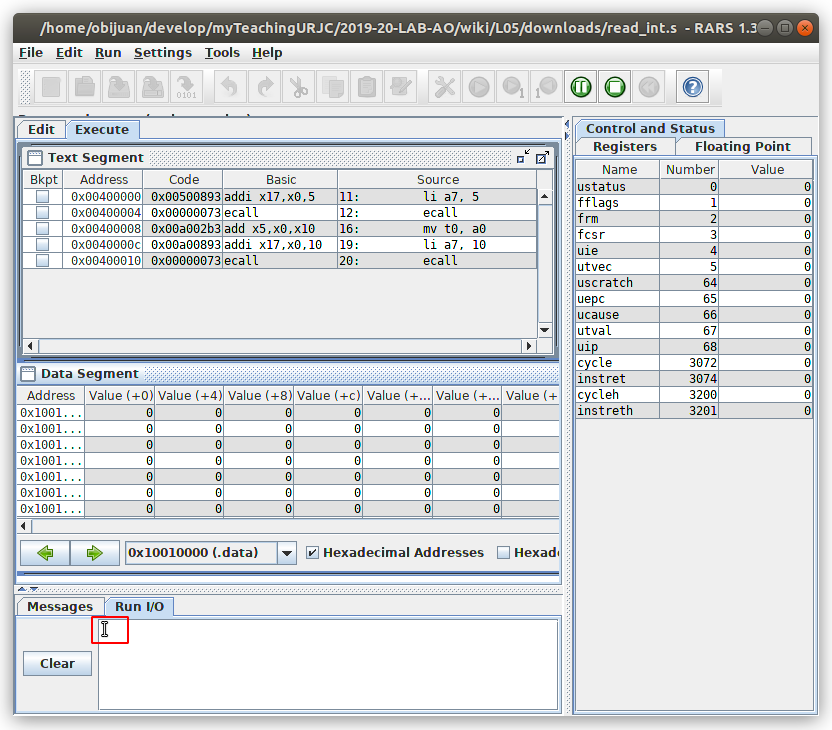


Su **código** es **5**. No hay que pasarle **ningún argumento de entrada**. El número introducido por el usuario se devuelve en el **registro a0**

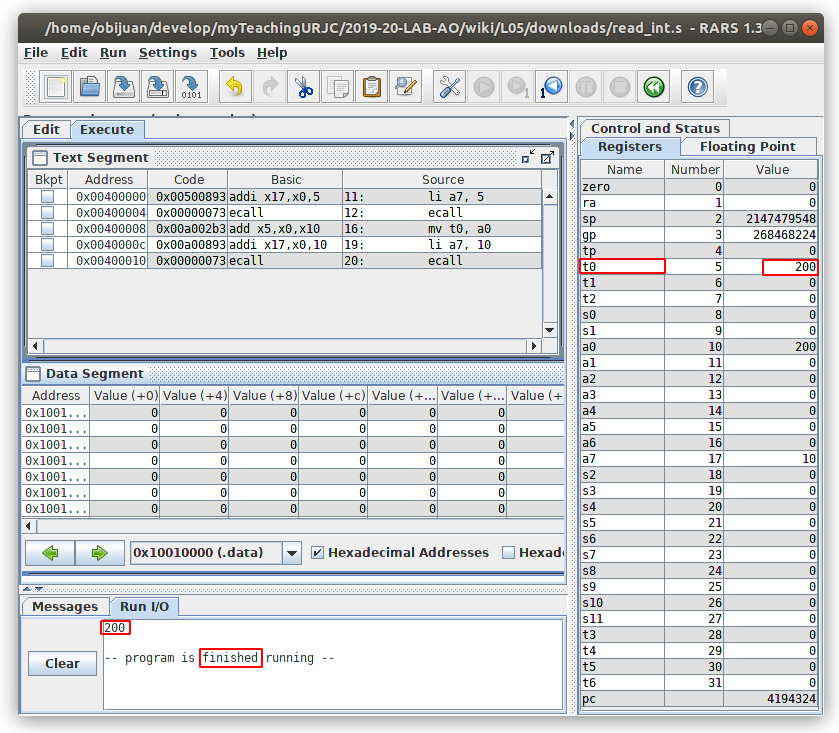
Para **probarlo** vamos a hacer un programa que almacene en el **registro t0** el valor introducido por el usuario. Igual que antes, definimos la constante **READ\_INT** con el valor del código:



Lo **ensamblamos** y lo **ejecutamos**. Al arrancar el programa se queda **esperando** a que introduzcamos un número en la **consola**. Vemos que en la consola aparece el **cursor**:



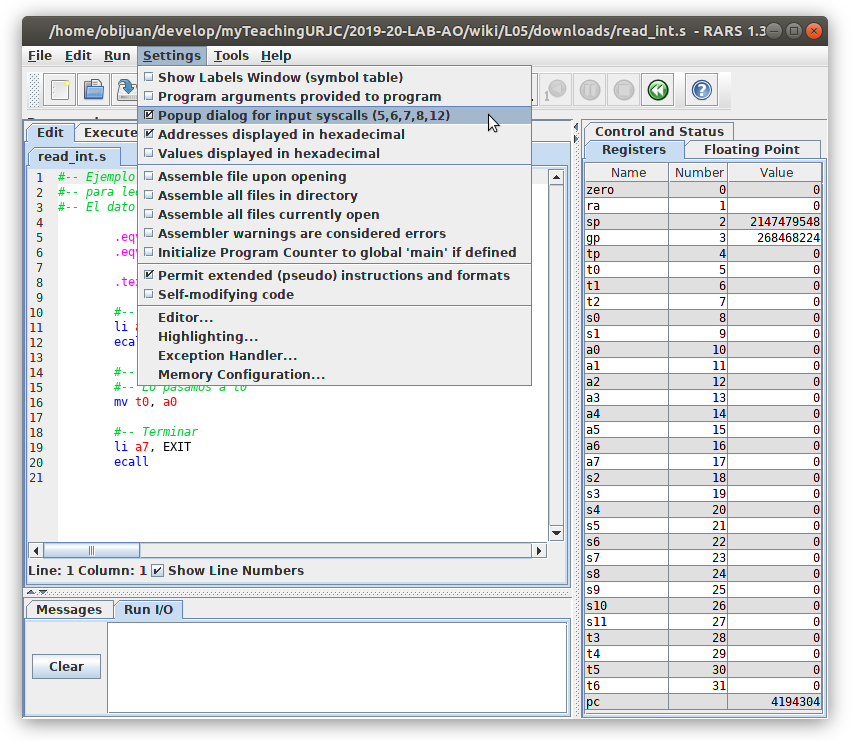
Introducimos el **número 200** (por ejemplo) y apretamos **ENTER**. El programa termina. En la derecha vemos que el **registro t0** tiene el valor que hemos introducido



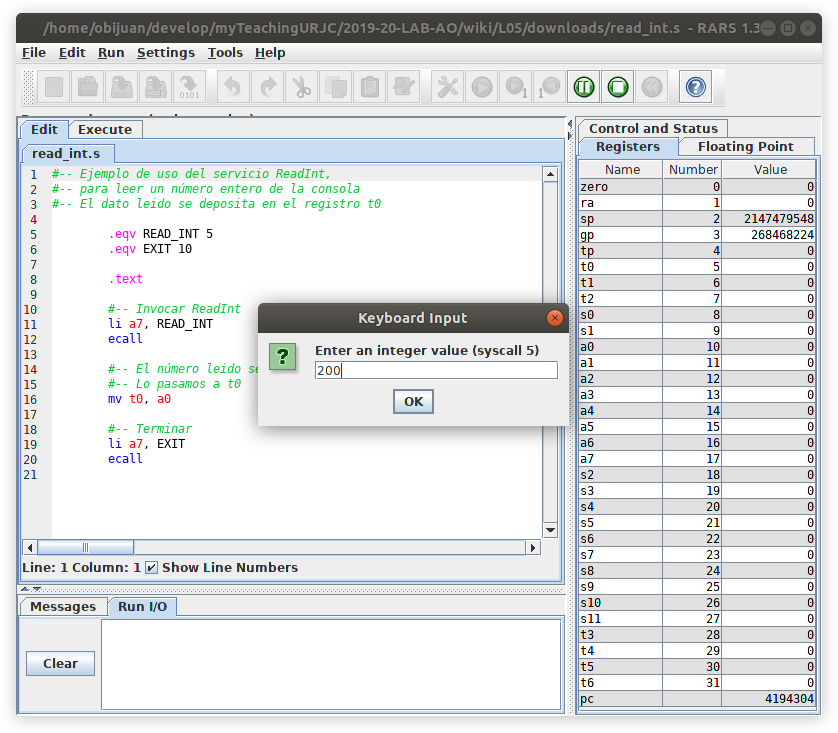
#### **Activando el diálogo de entrada**

Para hacer **más sencilla** la **entrada de datos**, se pueden activar las **ventanas de diálogo** para nos aparezca una **ventana nueva** cada vez que necesitemos que el **usuario** introduzca datos de entrada, en vez de hacerlo por la consola

Esta opción se activa en el menú **Settings/Pop up dialog for input syscalls (5,6,7,8,12)**



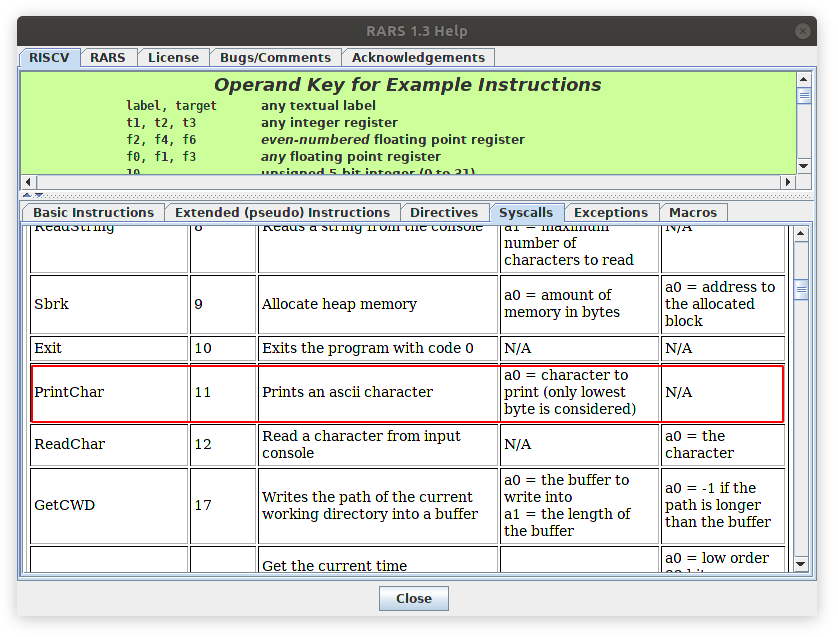
Ahora, al **ejecutar** el programa nos aparecerá la **ventana de diálogo** pidiéndolos el **número entero**:



El programa funciona exactamente igual, simplemente que la entrada de datos es más visual. En esta **animación** se muestra en **funcionamiento**

## PrintChar: Imprimiendo un carácter

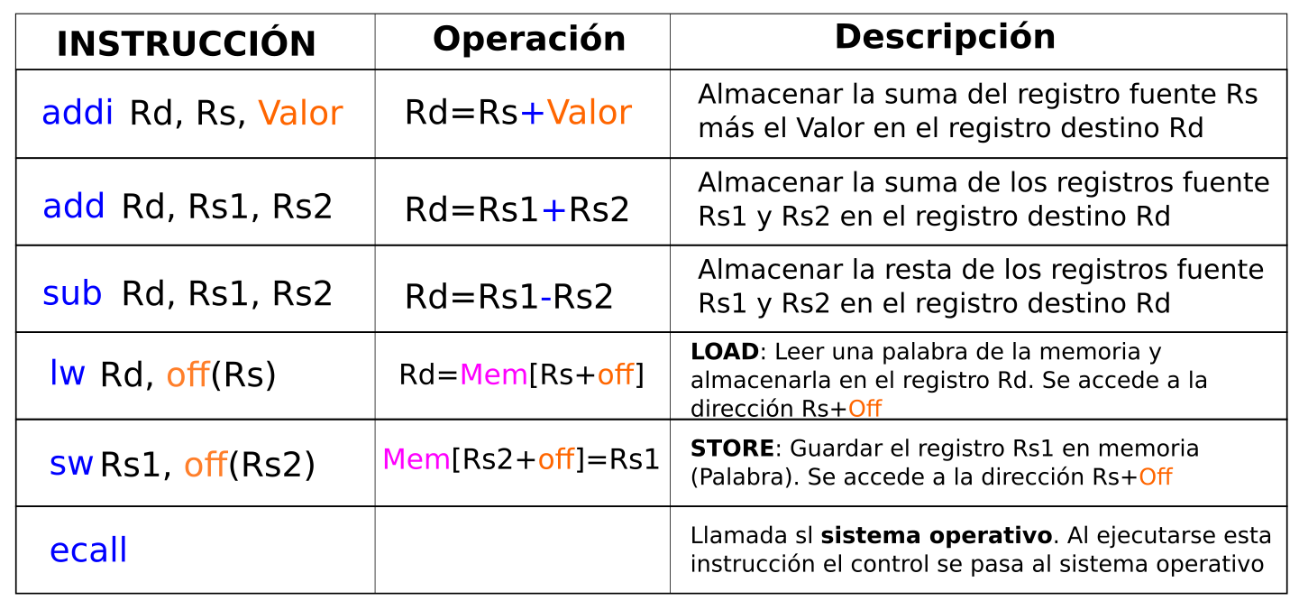
Podemos **imprimir** cualquier **carácter ASCII** mediante el **servicio PrintChar**



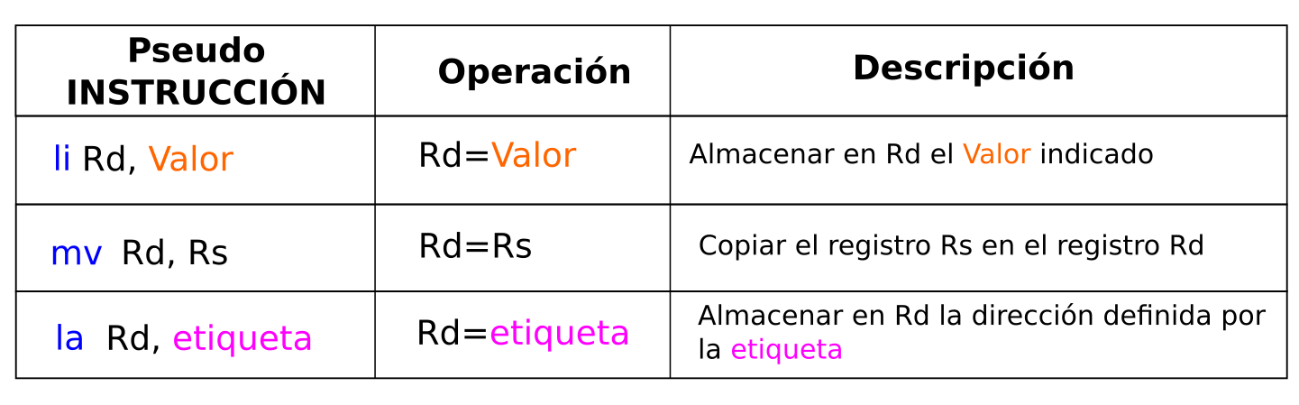
El **código** de servicio es el **11**. En el **registro a0** se introduce el **valor ascii** del carácter a imprimir. Las **constantes ASCII** se pueden introducir bien con su número, decimal o hexadecimal, o bien directamente como el carácter entre **comillas simples**: 'a', '\n', 'Z', etc...

## **Recopilación de instrucciones hasta el momento**

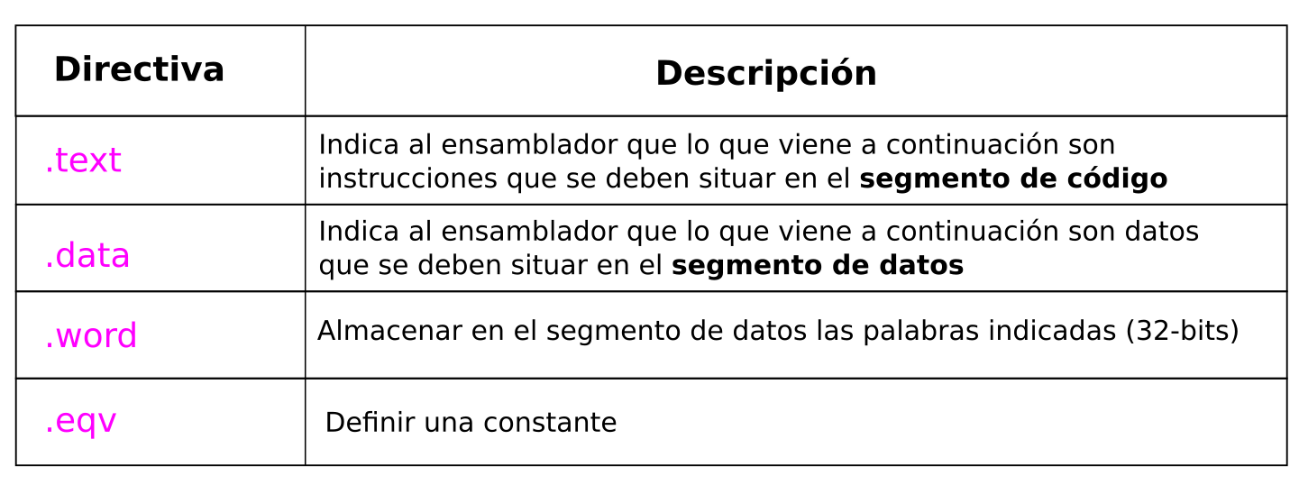
* **Instrucciones básicas**: Son las que se transforman a código máquina y que ejecuta el procesador



* **Pseudo-instrucciones**: No existen realmente como instrucciones. El **ensamblador** las **transforma** en instrucciones básicas. Una pseudo-instrucción puede dar lugar a 1 ó varias instrucciones básicas



* **Directivas**: Dar información al programa ensamblador. No generan código máquina



## **Cadenas**

Las **cadenas de caracteres** son importantísimas. Toda nuestra comunicación con los ordenadores se basa en cadenas de texto. Aprenderemos a **definirlas** y **trabajar** con ellas a bajo nivel, desde nuestro procesador RISC-V

## Definiendo cadenas

Las **cadenas** son **secuencias de caracteres** almacenados en posiciones **consecutivas** de memoria. En las cadenas que utilizaremos cada carácter ocupa **un byte**, y será almacenado por su **código ASCII**. Por tanto, **no usaremos** la letra **ñ** ni vocales son acentos: á, é, í, ó, ú: sólo los caracteres de la **tabla ASCII**

Las cadenas las definimos mediante la directiva **.string** (que es un alias de la directiva .asciiz) y su **último elemento** debe ser siempre el **byte 00**. Esto nos permite detectar **dónde** acaba una cadena. Como el resto de datos, las cadenas se almacenan en el **segmento de datos**

Este es un ejemplo de la **definición de dos cadenas**: "En un laboratorio de la URJC" y "de cuyo número no quiero acordarme"

#-- Ejemplo de definición de cadenas

#-- Se definen en el segmento de datos

.data

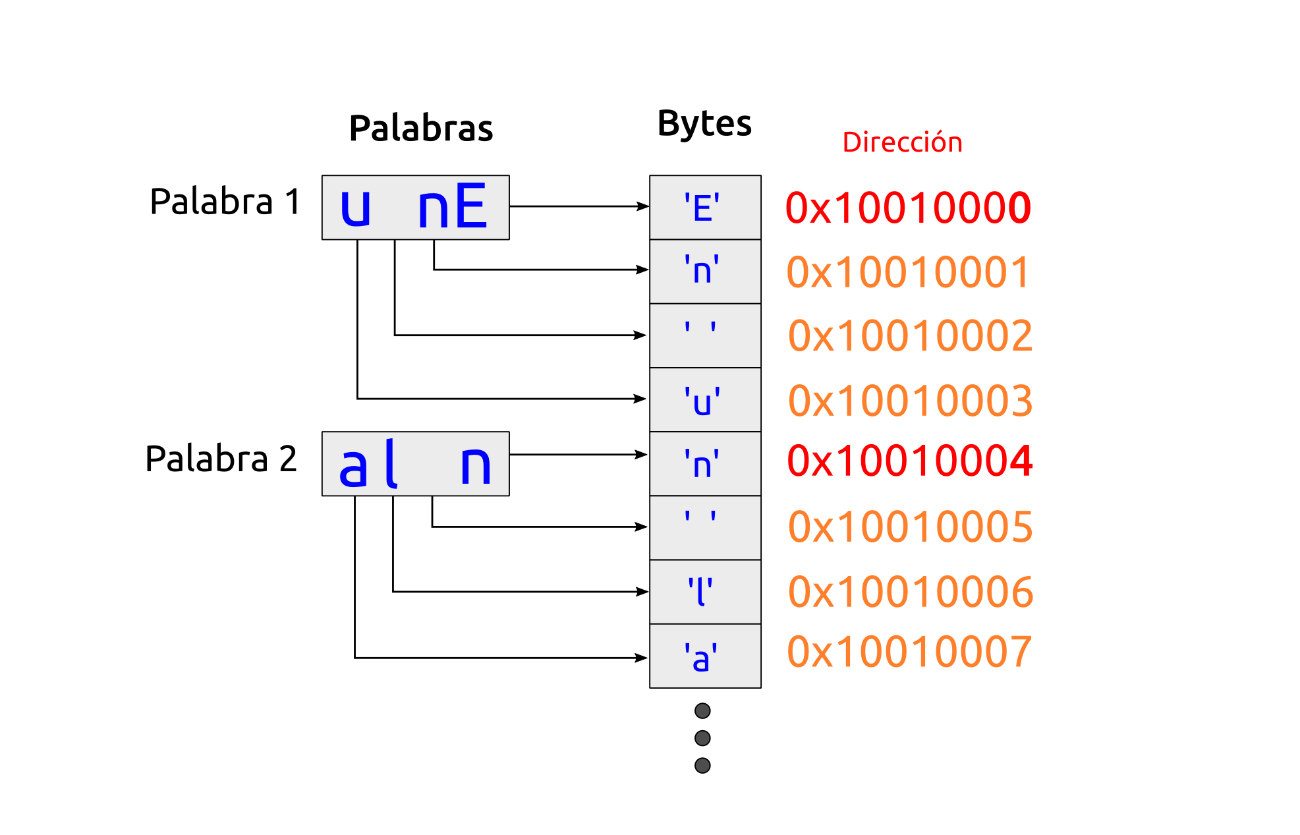
#-- Usamos la directiva .string

.string "En un lab de la URJC"

.string "de cuyo numero no quiero acordarme"

## Almacenamiento de las cadenas en memoria

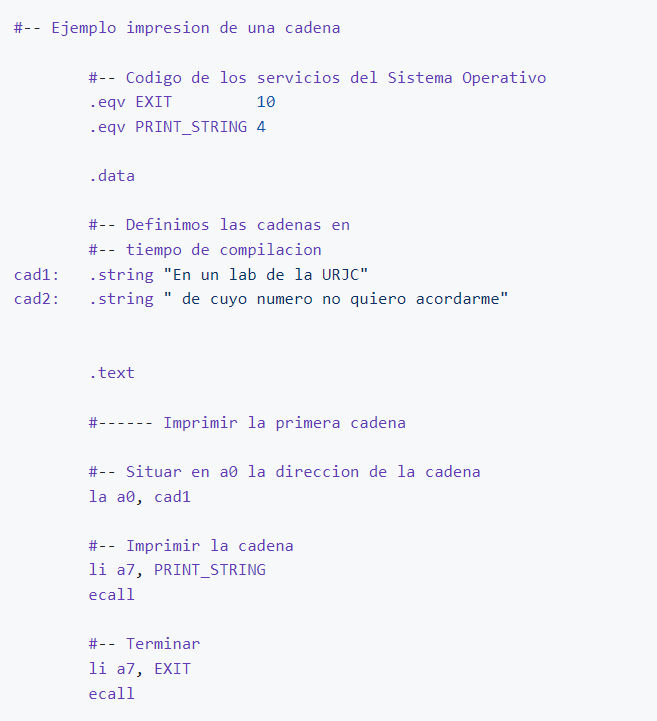
Las cadenas son **secuencias de bytes**, que se almacenan consecutivamente. Los 8 primeros bytes de la cadena definida en el apartado anterior está almacenada en memoria así:



Cuando lo vemos como **palabras** es cuando el orden nos parece extraño, ya que se utiliza la organización little-endian. Pero esto no es un problema ya que... ¡Las cadenas siempre las trataremos como bytes independientes!

## **Imprimiendo cadenas: Servicio PrintString**

Podemos **imprimir** cualquier **cadena** invocando el servicio **PrintString**, cuyo **código** es el **4**. Hay que colocar en **a0** la dirección donde está el primer carácter de la cadena



Las etiquetas cad1 y cad2 nos guardan la dirección del primer byte de cada cadena.

## **Leyendo cadenas introducidas por el usuario**

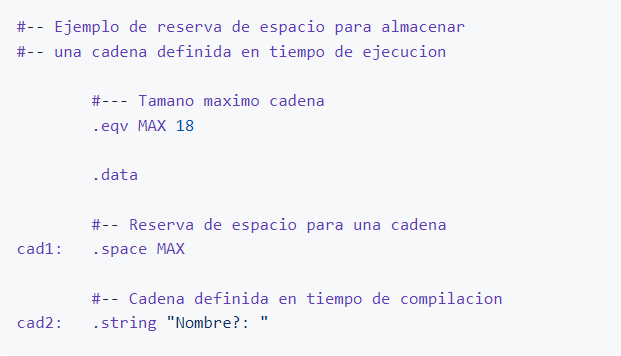
El **sistema operativo** nos ofrece un **servicio** para solicitar al **usuario** que introduzca una cadena. Este tipo de cadenas se dice que se define en **tiempo de ejecución**, ya que antes de ejecutar el programa **desconocemos** cuál es la **cadena**

Lo que sí que debemos definir es el **tamaño máximo** que reservamos en memoria para el **almacenamiento** de la cadena

### Reservando bytes: directiva .space

La **directiva .space** sirve para especificar el **número de bytes** que queremos **reservar**. Los usaremos para **almacenar la cadena pedida al usuario**, pero en general nos sirve para cualquier uso (no sólo cadenas)

En este ejemplo **reservamos 18 bytes**, y a continuación definimos una cadena en **tiempo de compilación.**



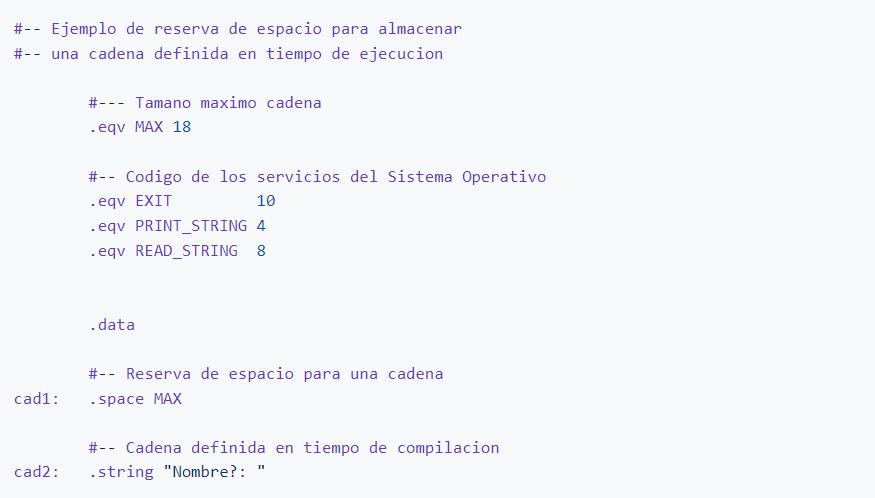
Como cada cadena tiene que terminar en 0. Ese carácter ya se incluye en los 18 reservados, es decir como máximo el usuario podrá poner 17 carácteres.

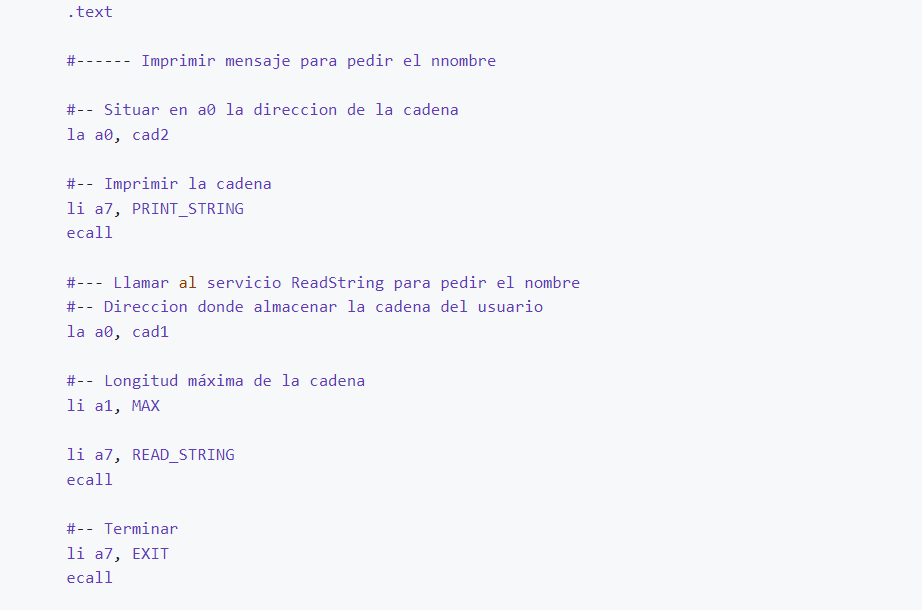
**Servicio ReadString**

Para **solicitar una cadena al usuario** invocamos el **servicio ReadString** del sistema operativo, cuyo **código** es el **8**. Los **parámetros** a pasar son los siguientes:

* **a0**: Dirección de memoria donde situar el primer carácter de la cadena
* **a1**: Tamaño máximo de la cadena que el usuario puede introducir

Al **invocarlo**, se le solicita la entrada de datos al usuario. Si está activada la opción **Settings/Pop up dialog for syscals** aparecerá una ventana emergente pidiendo explícitamente la cadena. Si no, aparecerá un cursor parpadeando en la consola. Al devolvernos el control, podremos ver en la memoria lo que ha introducido el usuario

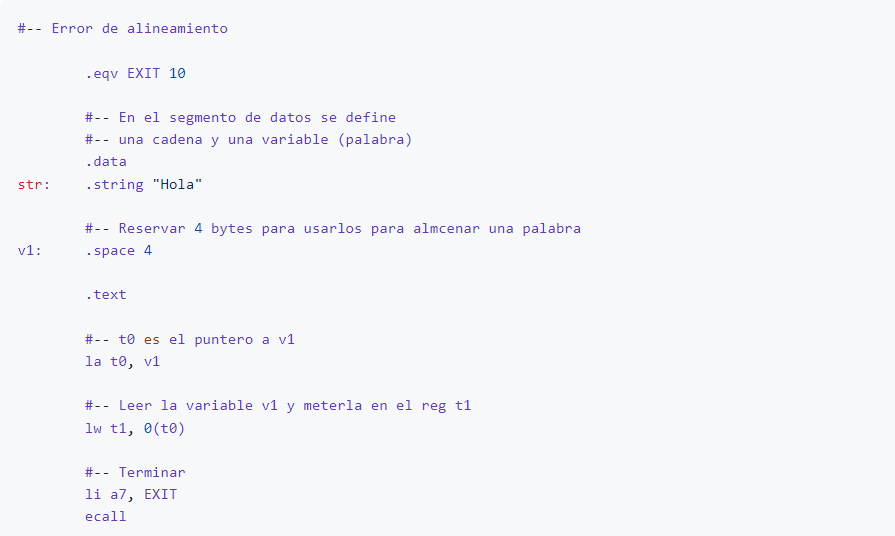




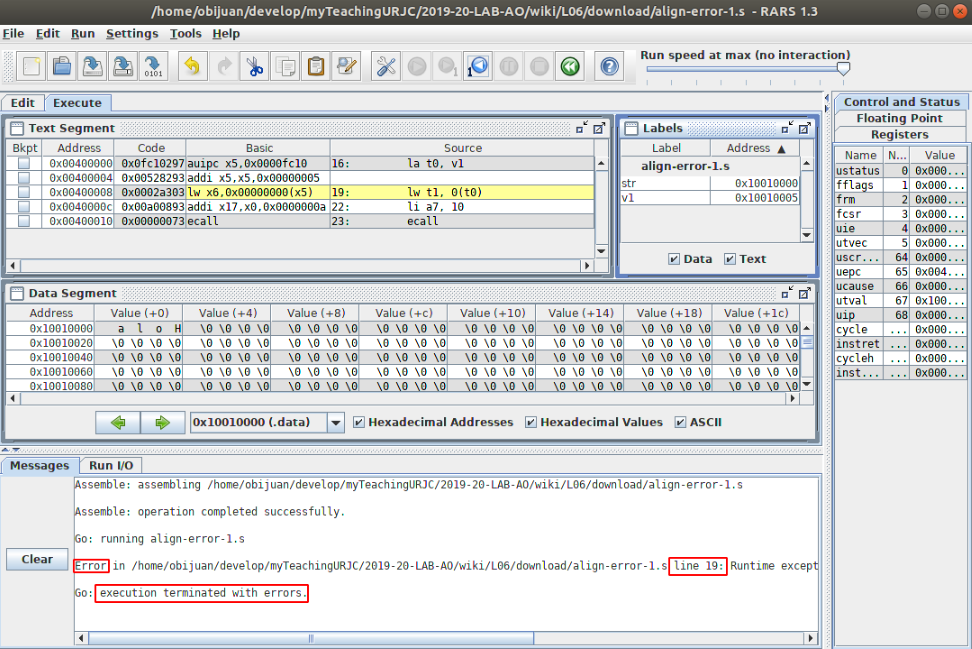
A tener en cuenta que el String que se pide al usuario se guardará en la dirección donde se guardó espacio para dicho string, esto es cad1. Por eso cuando queremos imprimir ese String que ha introducido el usuario, vamos a la dirección cad1.

## **Errores de alineamiento de palabras**

Como ya hemos visto, para trabajar con **variables** que son **palabras**, usamos las instrucciones lw/sw. Estas palabras, como ya dijimos también, SIEMPRE deben estar situadas en **direcciones alineadas**, o de lo contrario se produce un **error de acceso a memoria**



Lo **ensamblamos**. Al **ejecutarlo** observamos que se produce un **error**



El **mensaje de error** tiene esta pinta. ¿Qué ha pasado?

Si nos fijamos en la **tabla de símbolos**, vemos que la **etiqueta v1** se corresponde con la **dirección 0x10010005**, que NO es múltiplo de 4, y por tanto NO está alineada

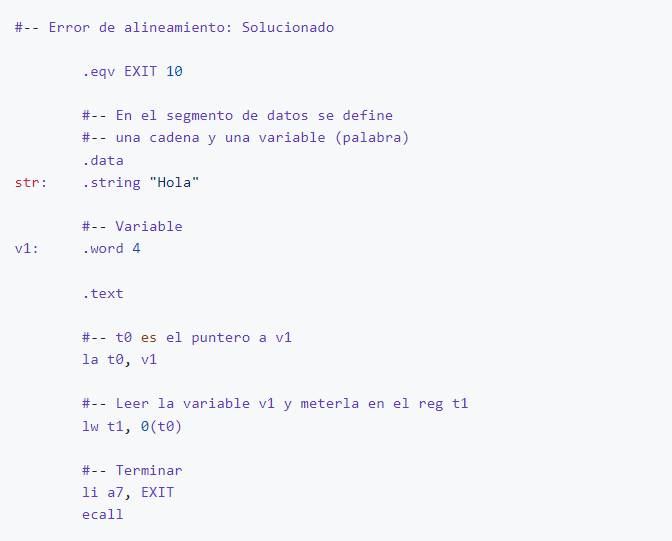
En la **línea 19** estamos usando la **instrucción lw** para **leer una palabra** de una dirección **NO alineada**. Esto **no es posible**, y por eso sale el mensaje de error

**Solucionando problemas de alineamiento**

Estos **problemas de alineamiento** se solucionas de manera sencilla. Tenemos **varias soluciones**:

* **Solución 1**: Usar **.word** para definir las palabras

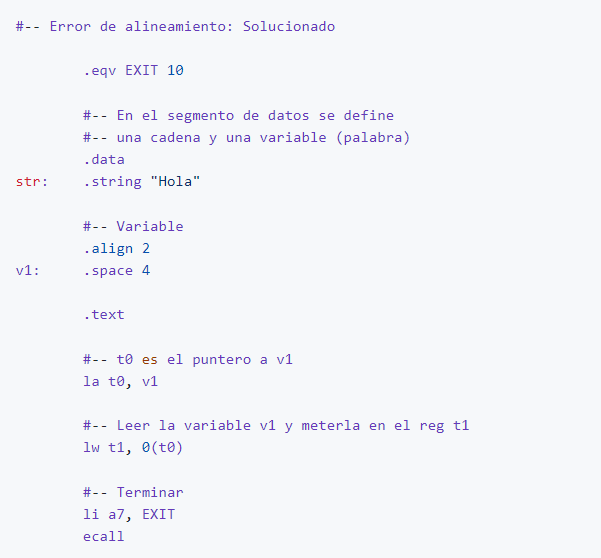
Siempre que usemos la **directiva .word** para definir una palabra en memoria, el **ensamblador** se encargará de que se encuentre en **una dirección alineada**, y que **NO** se produzcan estos errores. El ejemplo anterior quedaría así:



* **Solución 2**: Usar **.align 2** antes de definir la palabra

La directiva .align le dice al ensamblador que la siguiente dirección esté alineada. Los distintos tipos de alineamiento se especifican mediante el parámetro de .align (consultar el manual). Así, con

se indica que se quiere un **alineamiento de palabra**. Esto lo colocamos justo antes de definir el espacio para la palabra. El **ejemplo modificado** quedaría así:

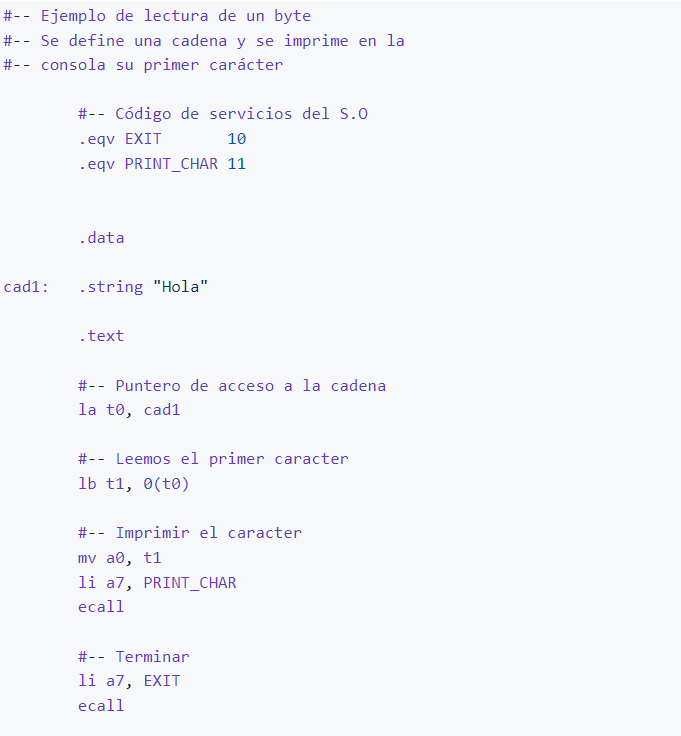


## **Trabajando con bytes**

Para trabajar con **cadenas** tenemos que leer **bytes** de memoria, y escribir **bytes** en ella. Los **accesos a memoria** de **tipo byte** los hacemos con las instrucciones **lb** y **sb** (load-byte, store-byte). Son iguales que lw y sw, pero los accesos son sólo a bytes

Cuando trabajamos con bytes, las **direcciones puedes ser cualesquiera**, no tienen que estar alineadas

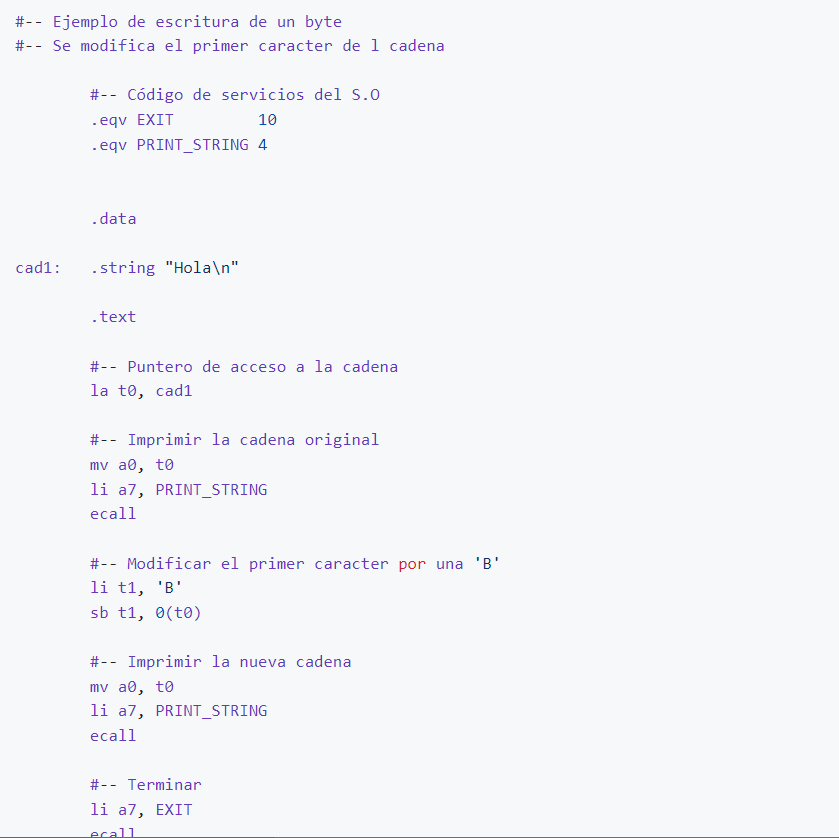
Donde t1 es donde queremos guardar el carácter que queremos manipular y t0 es la dirección donde se encuentra este carácter.



### Escritura de bytes: Instrucción sb

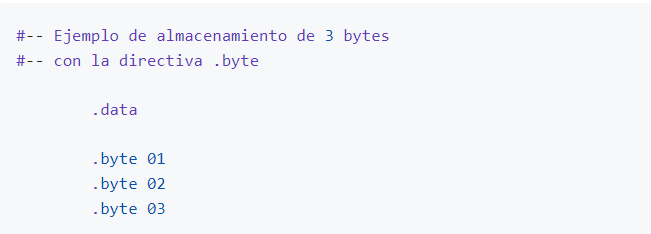
La **escritura de un byte** en memoria se hace con la **instrucción sb** (store-byte).

Donde t1 es el registro donde se encuentra el carácter nuevo que vamos a usar y t0 es el registro de destino, donde vamos a depositar ese nuevo carácter.



### Directiva .byte

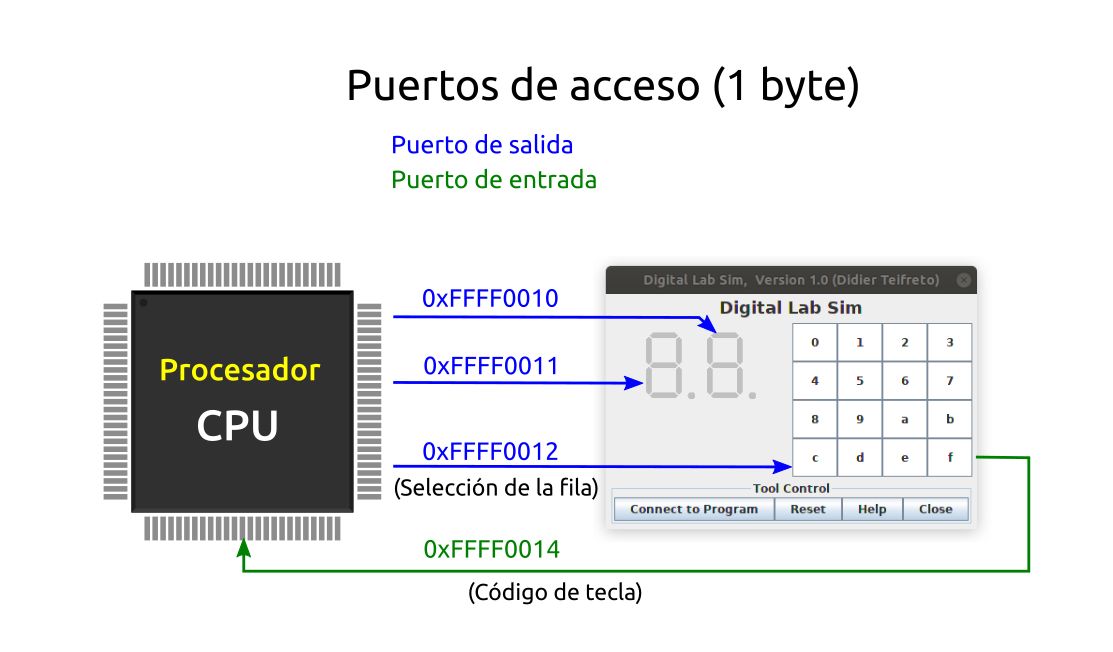
Podemos **almacenar bytes** aislados en la memoria usando la **directiva .byte**. En este ejemplo declaramos 3 bytes, inicializados con los valores 01, 02 y 03:



## **Accediendo a los Displays y al teclado**

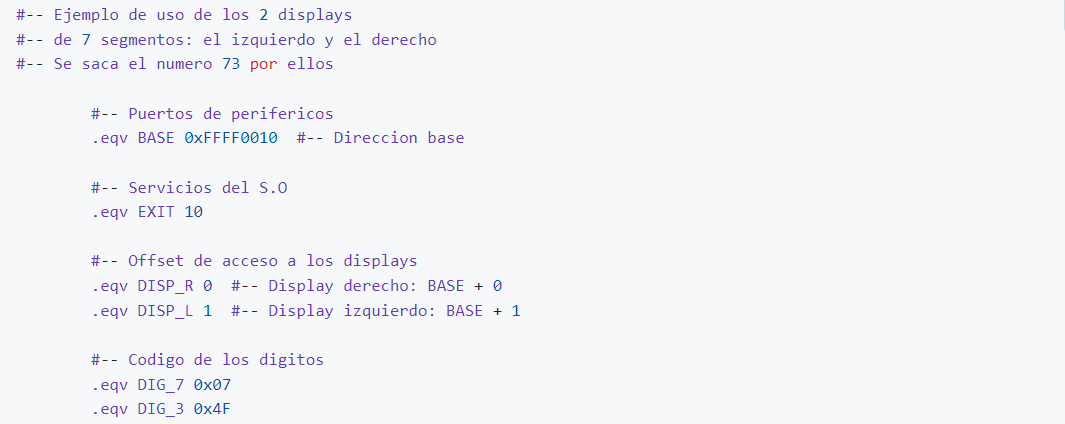
En la página 5 ya vimos como podemos acceder a un display con la instrucción SW. En realidad, para acceder a los periféricos, tantos a los displays como al teclado se usan 4 puertos de tipo byte. Cada byte tiene una función diferente.

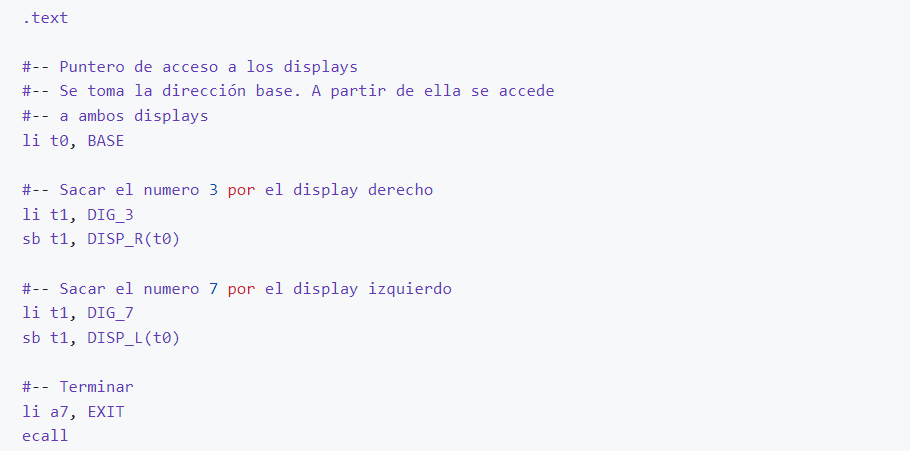
En esta figura se muestra el **esquema general**. Hay **3 puertos de salida**, y uno de **entrada**. Todos son de **tipo byte**, accesibles en las direcciones indicadas



### Acceso a los dos displays

Los **displays de 7 segmentos** se encuentran en las direcciones **0xFFFF0010** y **0xFFFF0011**. Para escribir lo podemos hacer con la **instrucción sb que hemos aprendido anteriormente.**

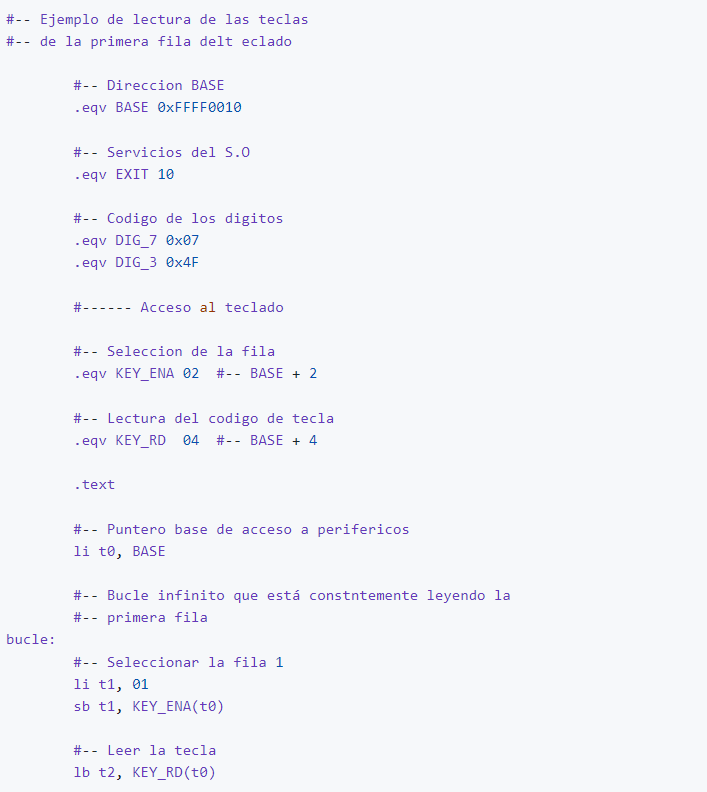




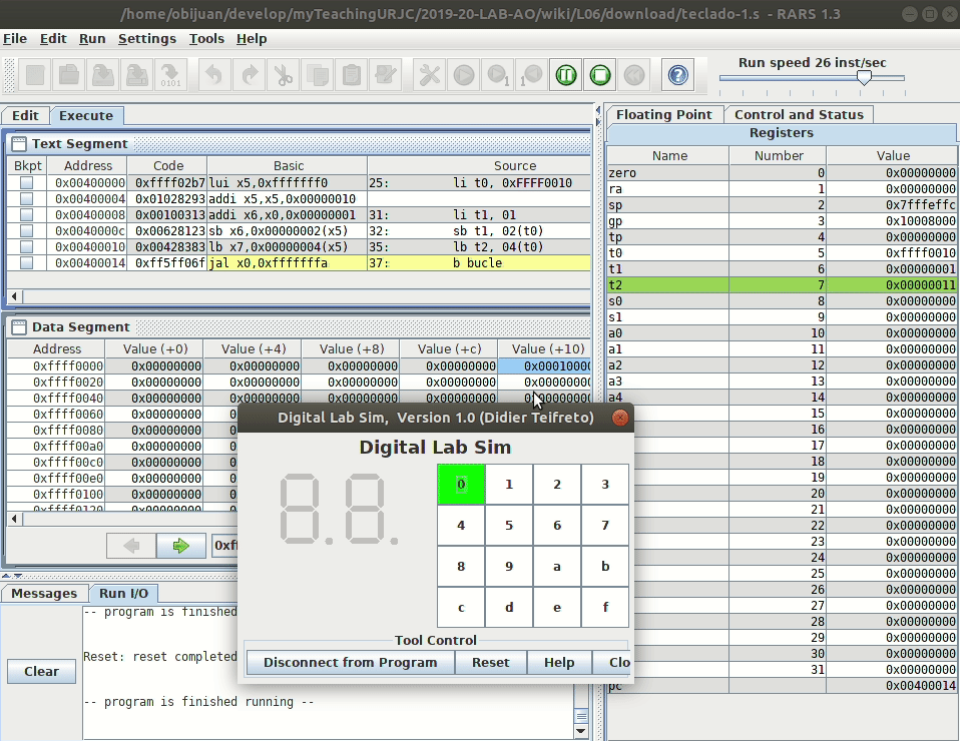
### Lectura del teclado

El **teclado** dispone de **dos puertos** de acceso (de 1 byte). Con el puerto de salida (0xFFFF0012) se selecciona la **fila** del teclado que se quiere comprobar. Leyendo del **puerto de entrada** (0xFFFF0014) se obtiene el **código de la tecla** pulsada, ó **0** si no hay ninguna apretada

Para **seleccionar** la fila a comprobar, se usan los **códigos** 1,2,4,y 8, esto viene dado por el fabricante y el **código** se introduciría en la dirección de salida donde esté el teclado. En este caso, como ya dijimos antes, está en la 0xFFF0012. Entonces si queremos la primera fila, tendríamos que poner un uno en dicha dirección.



Para **probarlo** lo **ensamblamos**, y **bajamos la velocidad** (Si está a su máxima velocidad el teclado NO funcionará). Le damos al **play** y nos fijamos en el **registro t2**. Cuando no hay ninguna tecla apretada, su valor es **0**. Cuando hay una tecla pulsada (la tecla está en verde) veremos su código



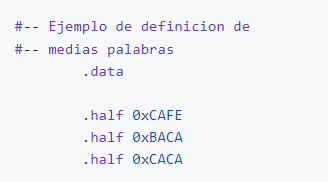
## **Trabajando con medias palabras**

En el **RISC-V** también podemos trabajar con **medias palabras** (16 bits, 2 bytes): usamos las instrucciones **lh** y **sh** (la h significa half-word)

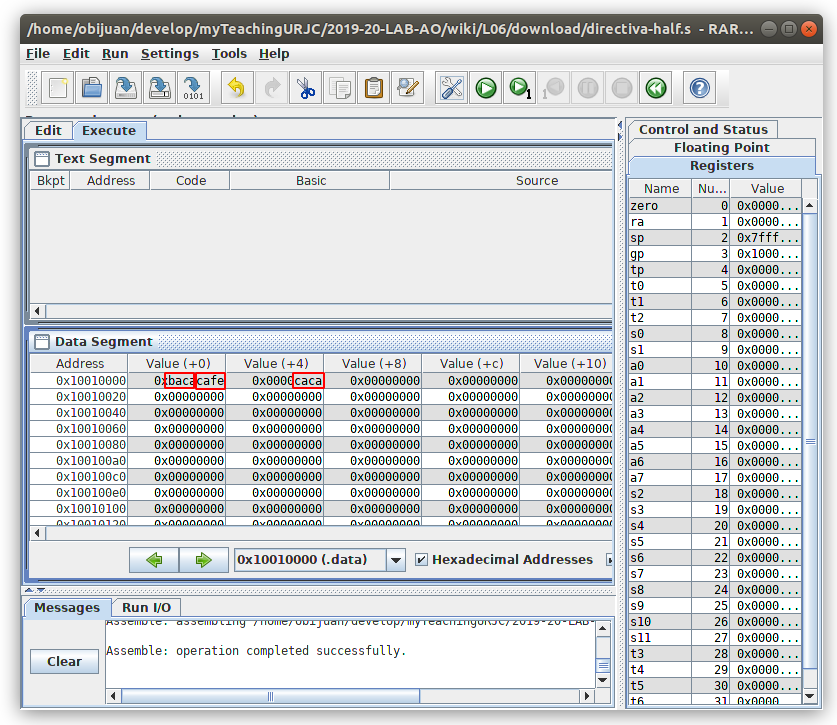
Es necesario que las **medias palabras** estén almacenadas en **direcciones múltiplos de 2**: deben acabar siempre en 0,2,4,6,8,A,C,E. Si no es así se producirá un **error de acceso** al usar lh o sh

### La directiva .half

Para **definir variables** de tipo **media-palabra** usamos la directiva **.half**. El ensamblador se encarga de que estén **correctamente alineadas**. En este ejemplo definimos **3 medias palabras**:



Lo **ensamblamos** y miramos el **segmento de datos**. Ahí están nuestras **medias palabras**



### Lectura de medias palabras: Instrucción lh

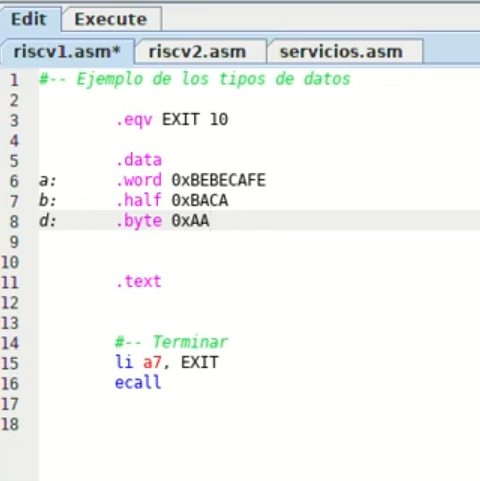
Para leer una media palabra en un registro usamos la **insrucción lh**.

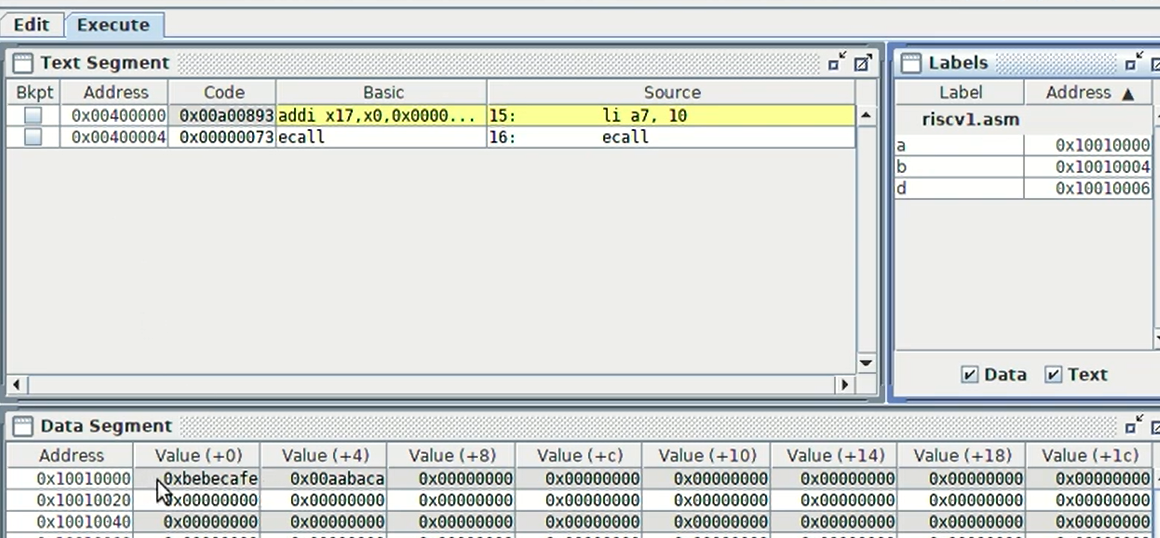
### Escritura de medias palabras: Instrucción sh

Para **almacenar** una **media-palabra** se usa la **instrucción sh**.

**Probando como guarda en memoria el RISC-V los distintos datos:**

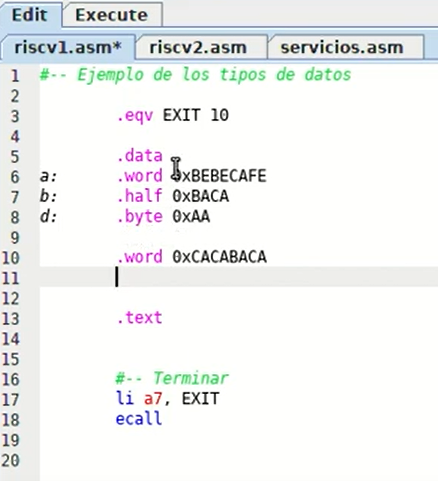
Guardando primero una palabra, una media palabra y un byte.

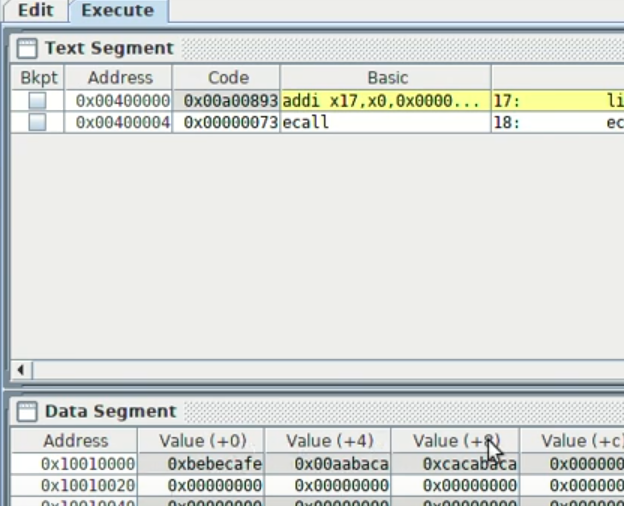




Vemos como la palabra **BEBECAFE** se me guarda en los primeros 4 bytes. La media palabra **BACA** en los siguientes 2 bytes y el byte **AA** en el siguiente byte del mismo registro de 4 bytes.

Ahora, ¿qué pasa si guardo después una palabra que ha de empezar en una dirección alineada?



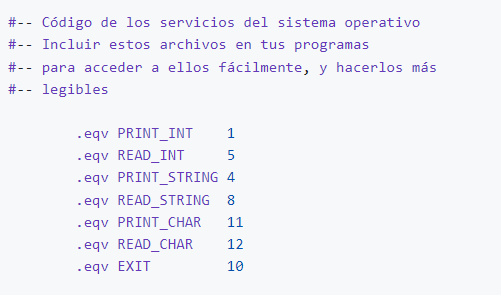


Vemos que la palabra la ubica en la siguiente dirección alineada y se queda el byte que no se estaba usando a 00. Pues bien, este byte, si seguimos guardando palabras ya no lo usaría el procesador para guardar datos. Es por eso que hay que usar el espacio de forma sabia.

## **La directiva .include**

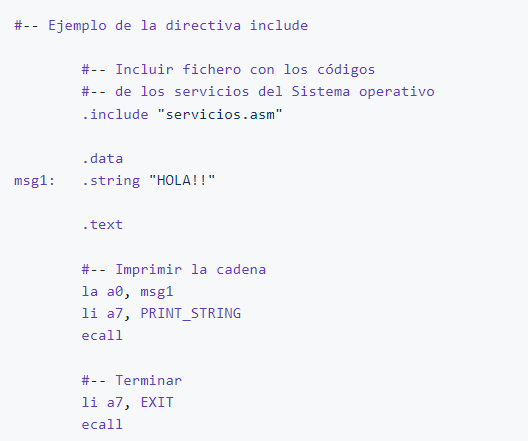
Para **facilitarnos** la programación, el ensamblador del RARs incluye una **directiva** que permite **incluir** otro **fichero en ensamblador** dentro de nuestro fichero. Esto es muy útil, por ejemplo, para tener almacenadas las **constantes** en un fichero, y poder compartirlas con varios programas

Un ejemplo muy útil es el de tener los **códigos** de los **servicios** del **sistema operativo** en el fichero **servicios.asm**:



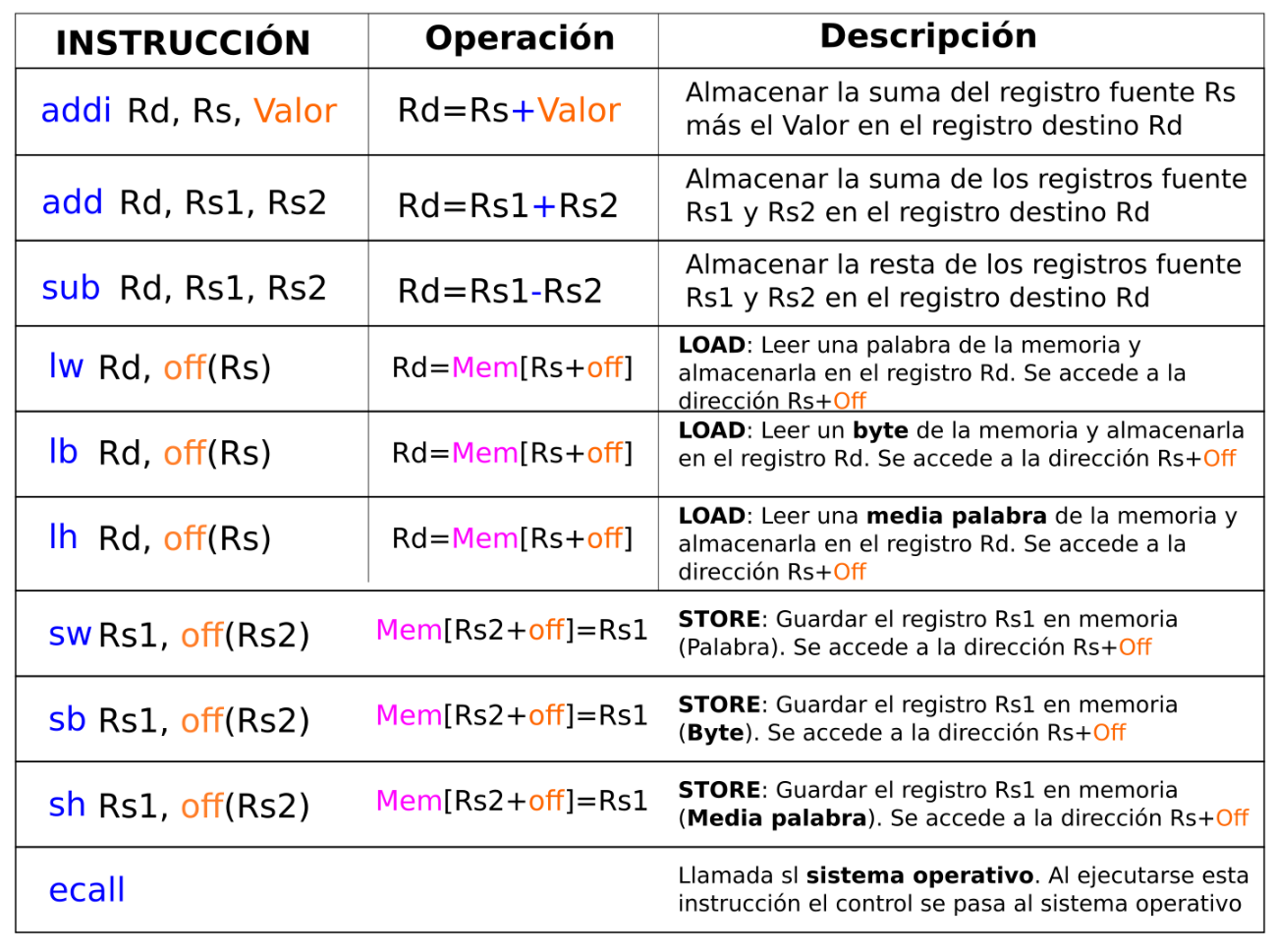
Ahora desde cualquier otro programa, que esté en el **mismo directorio** que el fichero servicios.asm, simplemente **añadimos** esta directiva, y el **fichero se incluirá** automáticamente:

Ahora solo necesitaremos poner esta instrucción en cualquier otro fichero para que nos incluya en ese programa lo que tenemos en servicios.asm.

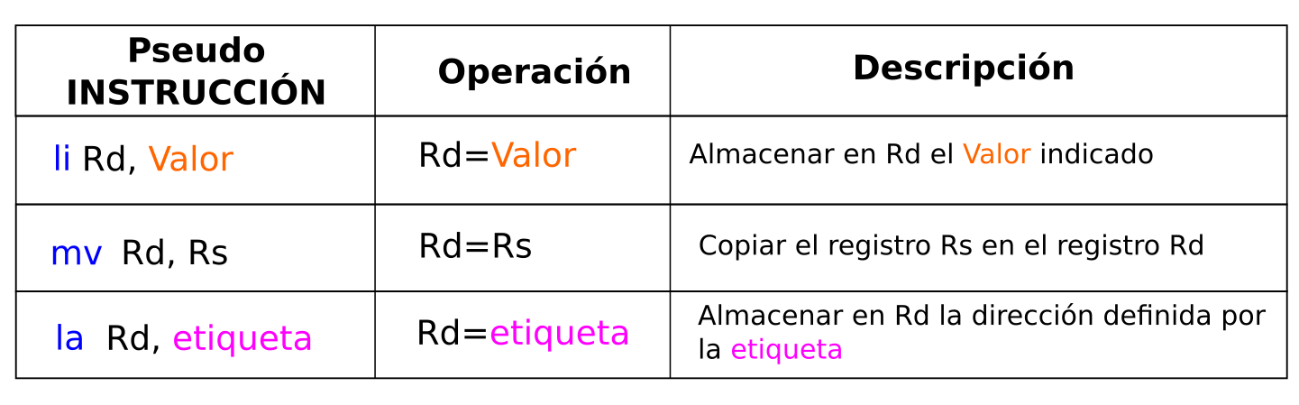


## **Recopilación de instrucciones hasta el momento**

* **Instrucciones básicas**: Son las que se transforman a código máquina y que ejecuta el procesador



* **Pseudo-instrucciones**: No existen realmente como instrucciones. El **ensamblador** las **transforma** en instrucciones básicas. Una pseudo-instrucción puede dar lugar a 1 ó varias instrucciones básicas



* **Directivas**: Dar información al programa ensamblador. No generan código máquina

