Capítulo 3

**Introducción**

Ya vimos que el **comportamiento** por defecto del procesador es **leer** una instrucción de memoria, **ejecutarla**, leer la siguiente, ejecutarla... Las instrucciones se van ejecutando **secuencialmente**, una detrás de otra

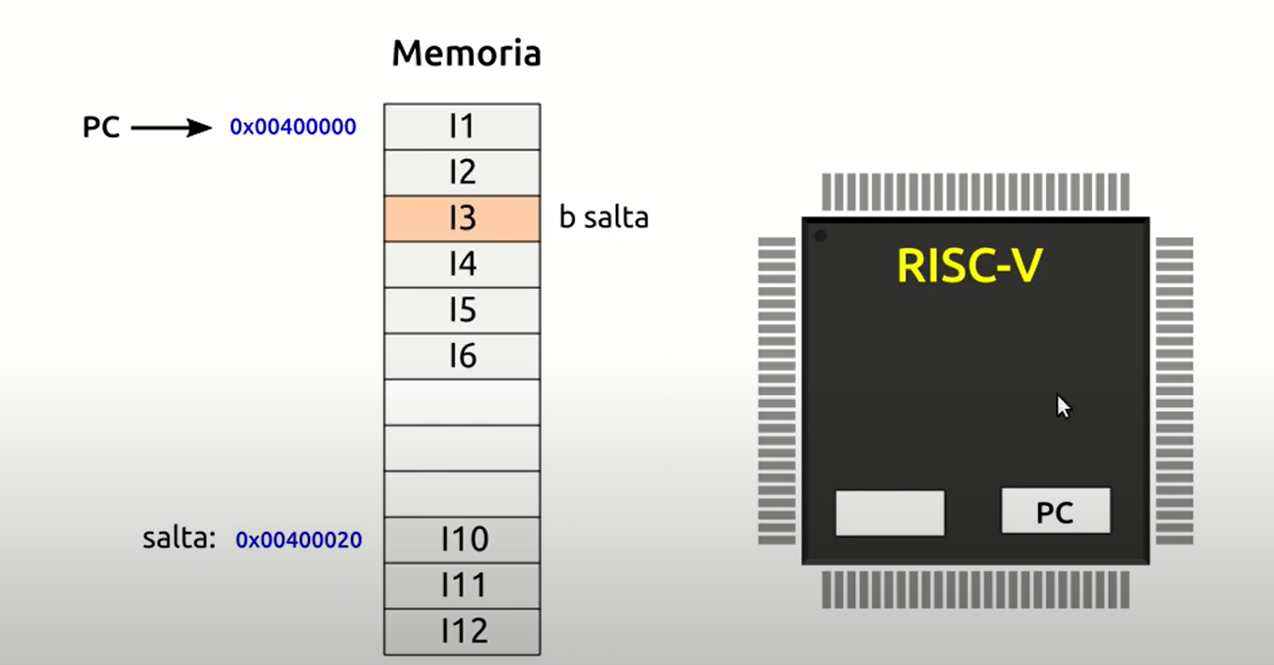
Sin embargo, ese comportamiento lo podemos **alterar** para que se ejecuten **instrucciones situadas** en **otros lugares** de la memoria. Es lo que se denominan **saltos**

Hasta ahora hemos visto como el procesador hace operaciones y se comunica con el SO, la memoria y los periféricos. ¿Pero cómo hace para tomar decisiones? Esto es lo que hace del procesador diferenciarse de otros. La toma de decisiones. Esta se hace con comparación y saltos.

## Saltos incondicionales

Mediante las **instrucciones de salto incondicional** modificamos el comportamiento del microprocesador para que siga **ejecutando** instrucciones situadas en **otro punto de la memoria**. En vez de continuar con la siguiente instrucción, podemos indicarle que ejecute las que se encuentran a partir de una **dirección de memoria**. La **instrucción** que utilizaremos para hacer estos **saltos** es **b** (branch)

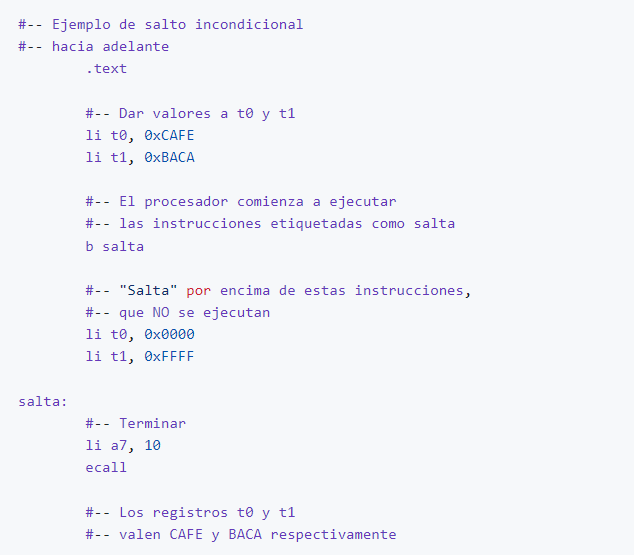
Primero hemos de saber que el procesador tiene un registro llamado PC que es el que le indica al procesador cual es la siguiente instrucción a ejecutar. Cuando un procesador se inicia, este siempre ha guardado previamente en el registro PC la primera dirección a ejecutar para cuando se vuelva a iniciar.



En esta imagen, el procesador ejecutaría las tres primeras instrucciones de manera ordenada y de ahí saltaría a la instrucción 10 y desde ahí seguiría ejecutando las instrucciones de forma secuencial (11,12…).

Como podemos observar, este salto lo hace si o si, no depende de una condición. De ahí su nombre.

Ejemplo 1

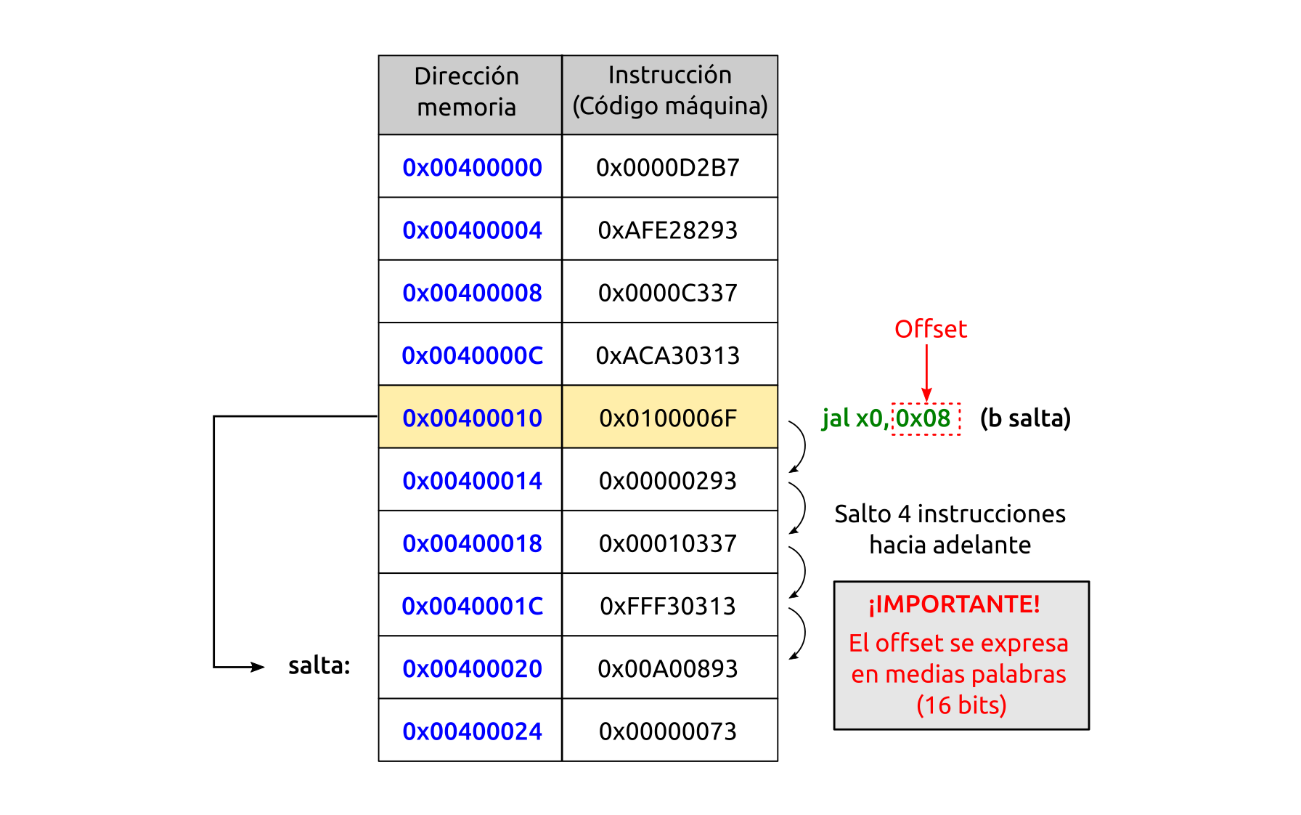


### **Instrucción b**

La **instrucción b** es en realidad una **pseudoinstrucción**, que en el ejemplo 1 se traduce a la siguiente **instrucción básica**:

Esta instrucción tiene dos parámetros. X0 es el registro donde queremos guardar la que sería la siguiente instrucción si no se ejecutase el salto y 0x08 esta relacionado con cuanta distancia (en medias palabras) tenemos que movernos hacia adelante o hacia atrás para llegar a la dirección. El programa pone de forma automática X0, puesto que, al ser incondicional, no necesitamos saber que instrucción era la siguiente en caso de no saltar.

El **funcionamiento** de la **instrucción b** se muestra en este dibujo, donde se ha representado el **mapa de memoria** correspondiente al programa anterior (ejemplo 1)



El microprocesador ejecuta las instrucciones desde arriba hasta abajo, comenzando en la dirección **0x00400000**. Al llegar a la dirección **0x00400010** ejecuta el **salto**, por lo que continúa con la ejecución de la instrucción que está en la dirección **0x00400020** (que se corresponde con la etiqueta salta)

Las instrucciones de **salto** se realizan de forma relativa a tu posición. Donde tienen un campo **offset** que indica **cuánto** moverse **hacia adelante**, o **hacia atras** (offset negativo). En el ejemplo 1, el procesador **ha saltado 4 instrucciones** hacia adelante. Como cada instrucción es una **palabra**, se ha hecho un salto hacia adelante de **4 palabras**. Sin embargo, en la **instrucción jal** (que es la que hace realmente el salto), se ha utilizado el offset con valor 0x08. En realidad, la cuenta del offset en el RISC-V se hace en medias-palabras (16 bits). De esta forma, el salto de **4 palabras hacia adelante**, es el mismo que un **salto de 8 medias-palabras**

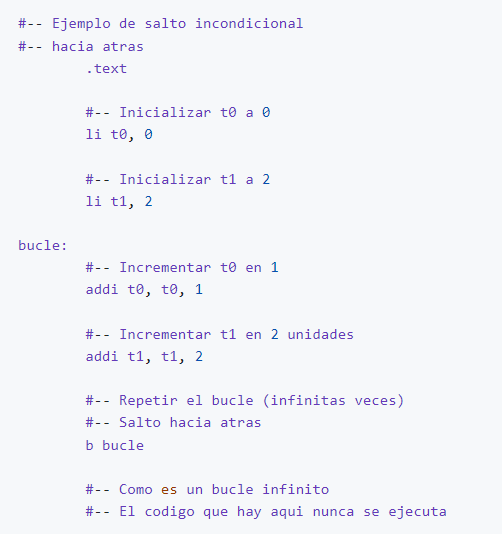
### **Juego de instrucciones comprimido**

La razón por la que el **offset** de los saltos se mide en **medias palabras** es porque el RISC-V tiene una **expansión**, conocida como **RV32C (el juego de instrucciones original es el RV32i)**, que implementa un **juego de instrucciones comprimido**. Algunas instrucciones comunes se codifican de forma que ocupen **16 bits**, en vez de 32. De esta forma, se pueden **comprimir** dos instrucciones en lo que antes ocupaba una, con lo que se consiguen ahorros de hasta un 25% en el tamaño de los programas. Puedes encontrar **más información** en este documento: [riscv-compressed-spec-v1.7.pdf](https://riscv.org/wp-content/uploads/2015/05/riscv-compressed-spec-v1.7.pdf)

En esta asignatura **NO veremos** las instrucciones comprimidas

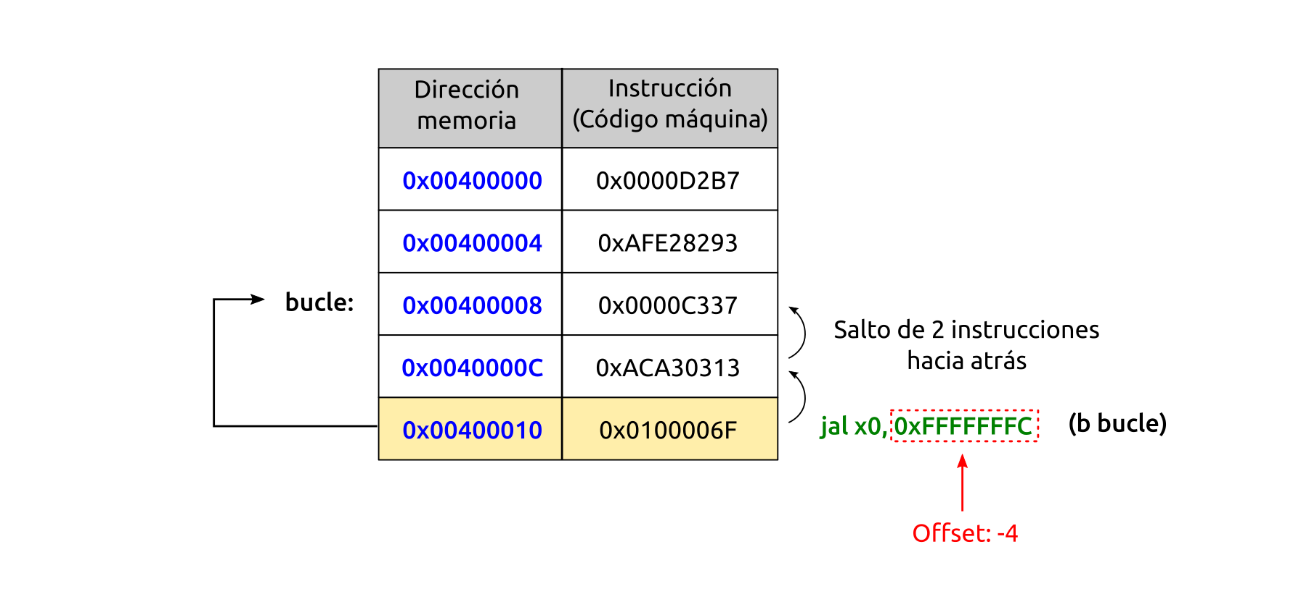
### **Salto hacia atrás (bucle infinito)**

También se pueden realizar **saltos hacia atrás**, retrocediendo hacia direcciones más bajas de memoria, como se muestra en este **ejemplo**. El programa principal inicializa los registros **t0** y **t1** con los valores **0** y **2** respectivamente. Incrementa t0 en uno, t1 en dos y **salta** dos instrucciones **hacia atrás** para repetir el proceso. El ciclo se repite indefinidamente hasta que paramos el simulador o apagamos el microprocesador real. Se denomina **bucle infinito** (y es algo que se debe evitar. El control siempre hay que devolvérselo al sistema operativo). Es algo que no se suele hacer si tenemos un SO.



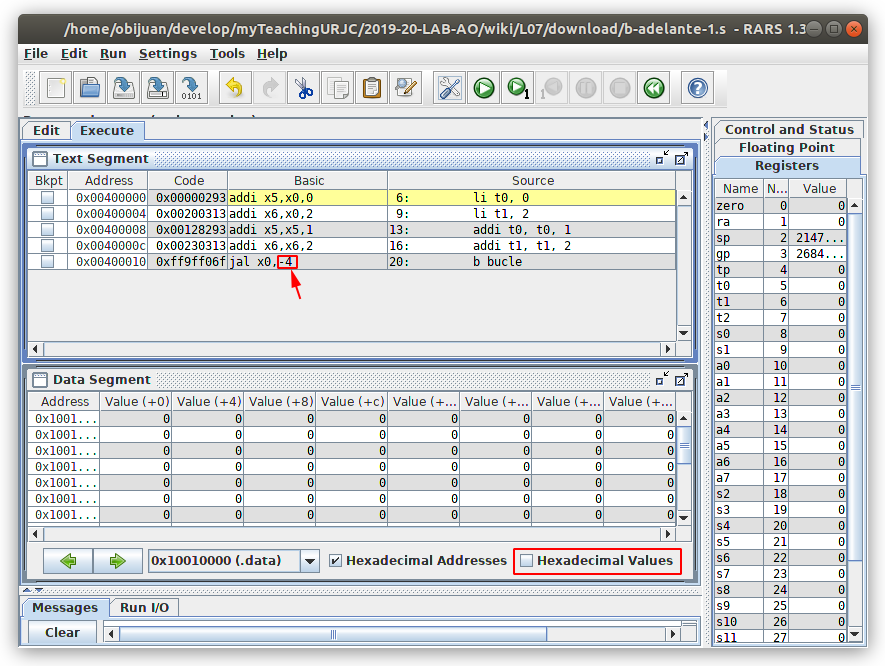
Si observamos la **instrucción de salto** que se ha generado, es:

El offset se interpreta como un **número con signo**. Como el salto es **hacia atrás**, esto se interpreta como **negativo** (la dirección de memoria se decrementa). El salto es ahora **2 instrucciones** hacia atrás



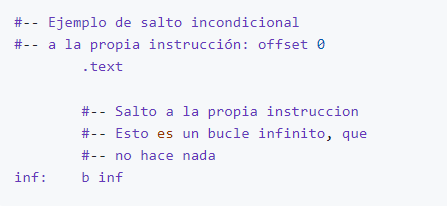
Como el offset se especifica en **medias palabras**, se está realizando un **salto de 4 medias palabras hacia atrás**. Es decir, el **offset es de -4**. Que puesto en **complemento a dos** de 32 bits es el número **0xFFFFFFFC**

Los **offsets** de las instrucciones de salto los podemos convertir fácilmente de hexadecimal a decimal con signo con la opción **hexadecimal Values**



### Salto con offset 0: Bucle infinito de una instrucción

Este es un ejemplo de una **instrucción que salta a sí misma**, y por tanto se entra en un bucle infinito que no hace nada



La instrucción de salto generada tiene **offset 0**:

Esto hace que la siguiente instrucción a ejecutar sea la que **está en la dirección actual más 0**. Es decir, otra vez la que se acaba de ejecutar. Así hasta el infinito... o hasta que se quite la alimentación del microprocesador

## Saltos condicionales

Las instrucciones de **salto condicional** permiten al procesador **tomar decisiones**. Son muy importantes. Con ellas podemos **comparar** los valores de los registros y realizar unas **acciones** u otras en función los resultados. También nos sirven para hacer **bucles** que terminen al cumplirse alguna **condición**

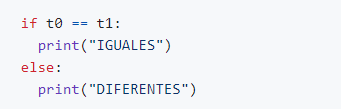
Las instrucciones básicas de **salto condicional** son: **beq**, **bne**, **bge** y **blt**. Realizan las operaciones de **comparación de igualdad** (=), de **desigualdad** (!=), **mayor o igual** (>=) ó **menor que** (<). Como los saltos condicionales se utilizan mucho, se definen más **pseudo-instrucciones** para facilitarnos la vida: beqz, bgez, bgt, ble...

### **Ejemplo: Comprobar si dos registros son iguales**

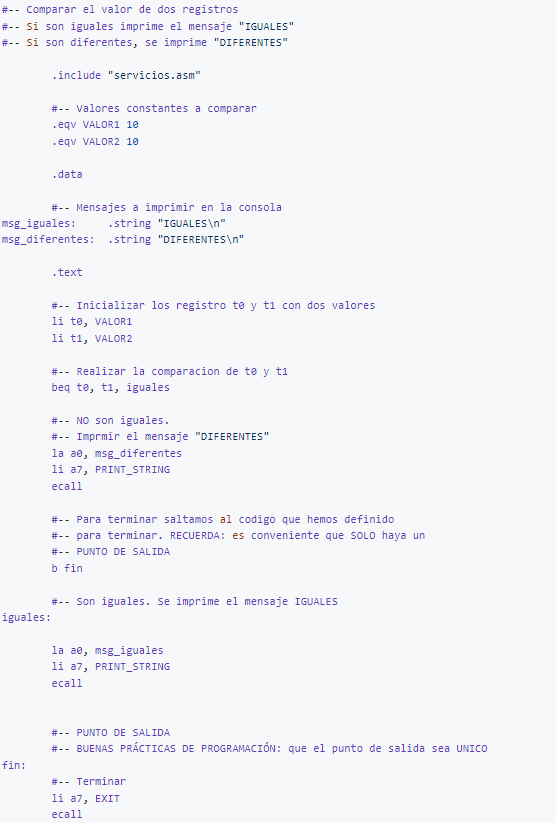
Empezaremos practicando con la **instrucción beq** que nos permite saber si **dos registros son iguales** o no SOLO sirve con dos registros. En caso de ser iguales se realiza un **salto**. Si no son iguales se **continúa ejecutando** el código que sigue a beq

En este **ejemplo** damos dos valores a los registros **t0** y **t1**, y los comparamos. Si son iguales, se imprime la cadena "IGUALES", pero si son diferentes se imprimirá "DIFERENTES"

La **lógica** que se está implementando en el programa es el equivalente al **if** de los lenguajes de **alto nivel**:



Este es el programa en **ensamblador**



La clave está en la **instrucción beq**:

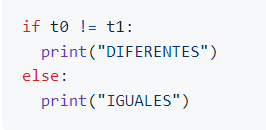
Cuando **t0** y **t1** son **iguales**, el micro **ejecuta** las instrucciones que están a partir de la **etiqueta iguales**. En caso contrario, continúa ejecutando las que están a continuación del beq

### **Comprobando si dos registros son diferentes**

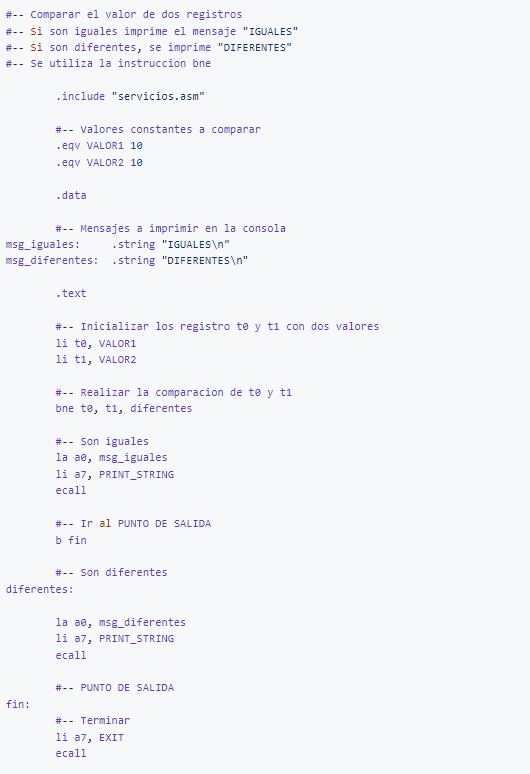
La condición de que **dos registros sean diferentes** se comprueba con la **instrucción bne** que realiza un salto si NO son iguales. Es justo la **contraria** de beq

Como **ejemplo** haremos el **mismo ejemplo anterio**r de comparación de dos números, pero usando **bne** en lugar de beq. Si los números son **diferentes**, se imprime el mensaje "DIFERENTES" y si son iguales se imprime "IGUALES"

Si lo expresásemos en un **lenguaje de alto nivel**, este sería el código:



Este es el **programa en ensamblador**:



La **instrucción clave** es:

Si los registros **t0** y **t1** son diferentes, salta a ejecutar el código situado a partir de la **etiqueta no\_iguales**. En caso contrario continúa ejecutando las instrucciones que están a **continuación de bne**

Por lo demás, este código es **prácticamente igual** al del ejemplo anterior. El mismo programa se puede implementar de múltiples maneras

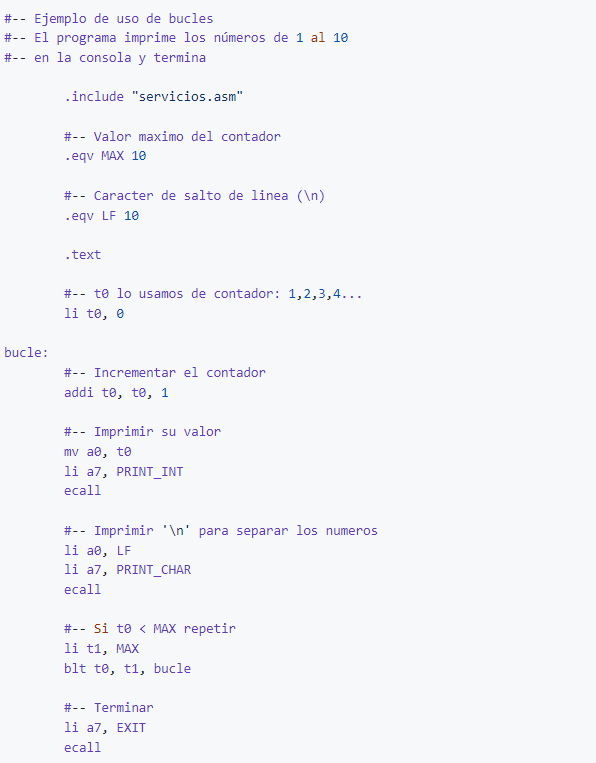
## Haciendo bucles

Las instrucciones de salto condicional nos permiten hacer **bucles** que terminen cuando se **cumpla una condición de salida**

### Ejemplo: Imprimiendo los números de 1 al 10

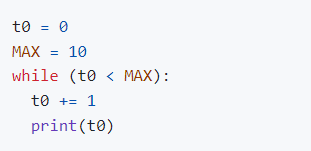
La **instrucción blt** realiza un **salto** si el primer registro es **estrictamente menor** que el segundo. Esto es muy útil para hacer, por ejemplo, **bucles** que terminen cuando se alcanza un **valor máximo**

En este ejemplo se utiliza el **registro t0** para **contar** de **1** a **10**. Mientras que t0 sea **menor a 10**, se repite el bucle. En cuanto es 10, se termina



La **instrucción clave** es:

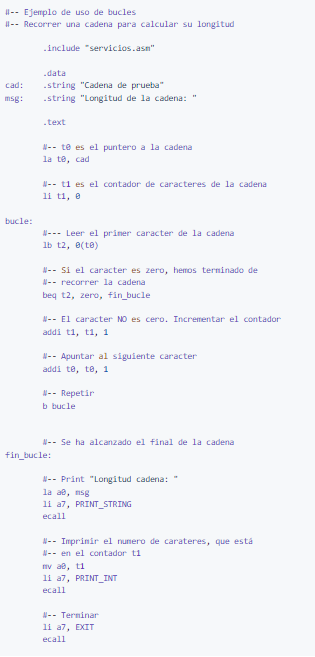
Mientras que **t0 < t1**, se **repite** el bucle. A t1 se le ha asignado el valor constante MAX. El código equivalente en un lenguaje de **alto nivel** sería el siguiente:



### **Contando los caracteres de una cadena**

Los bucles nos sirven para **recorrer cadenas**, detectando cuándo se llega al final. Esto lo sabemos porque su último elemento es el '\0'

Este programa calcula el **número de caracteres** que hay en una **cadena**. El programa usa un puntero, **t0**, para acceder a todos los elementos de la cadena hasta que alguno de ellos es **0**. En ese momento se imprime su **longitud**



## Cadenas

Las **cadenas de caracteres** son muy importantes. Ahora que ya sabemos hacer **bucles** y tomar **decisiones** podemos implementar **algoritmos sencillos** para **manipular cadenas**. Haremos un **repaso** de los conceptos implicados, muchos de los cuales ya se han expuesto antes y se conocen, pero se incluyen aquí como referencia

Lo más importante es **analizar los ejemplos** y **practicar** intentando **resolver los ejercicios**

## Caracteres alfanuméricos: Tabla ASCII

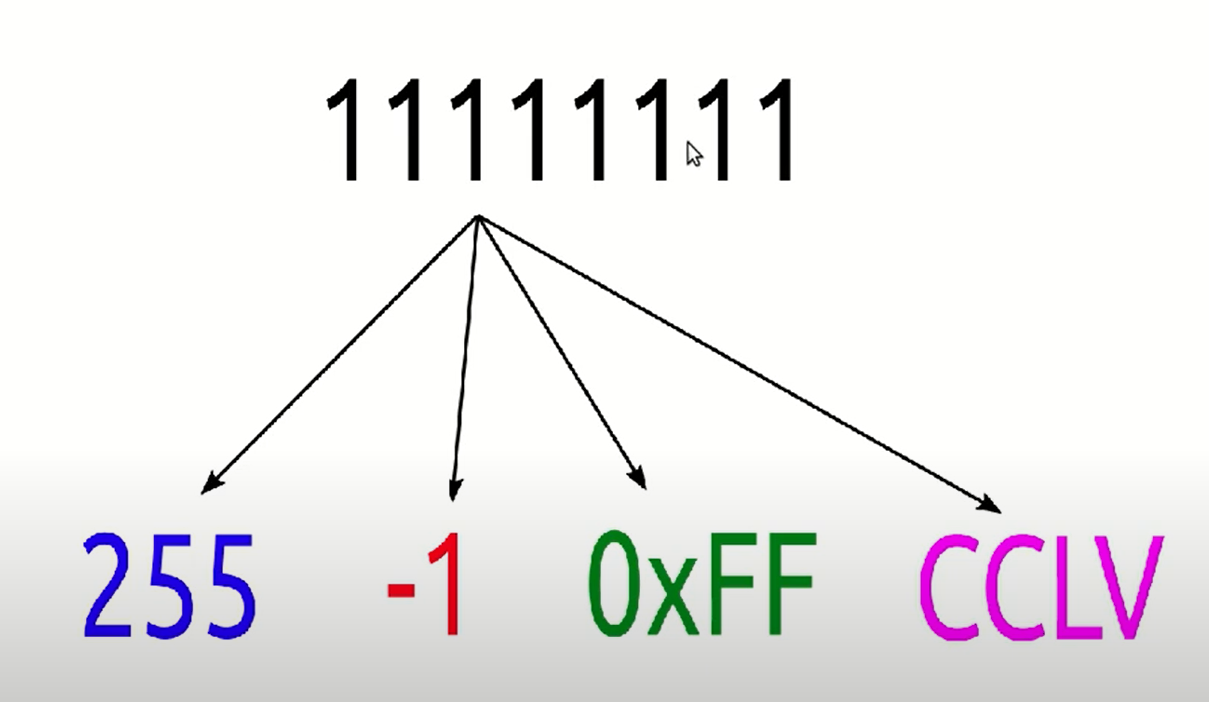
Los procesadores son **sistemas digitales**. Esto significa que funcionan con **números**. Son capaces de **almacenar**, **transportar** y **transformar números**. Estos números representan **instrucciones** o **datos**. Los números usados como datos pueden a su vez representar una **imagen**, un **vídeo**, un **sonido**, un **carácter**...

Los caracteres son **letras** (a,b,c...), **dígitos** (0,1,2...) o **signos de puntuación** (espacios, comas, paréntesis...). Para representar caracteres se usa una **codificación**: se asignan números para representar cada caracter. Una codificación muy extendida es el [Código ASCII](https://es.wikipedia.org/wiki/ASCII). Es el que usaremos

Así, por ejemplo, el código ASCII correspondiente a la **letra 'G'** es el **71** (0x47), y para la **letra 'g'** el **103** (0x67)

Cuando escribimos un caracter con sus **comillas**: 'g', en realidad nos estamos refiriendo a su **código ASCII**. En este caso sería el número 71

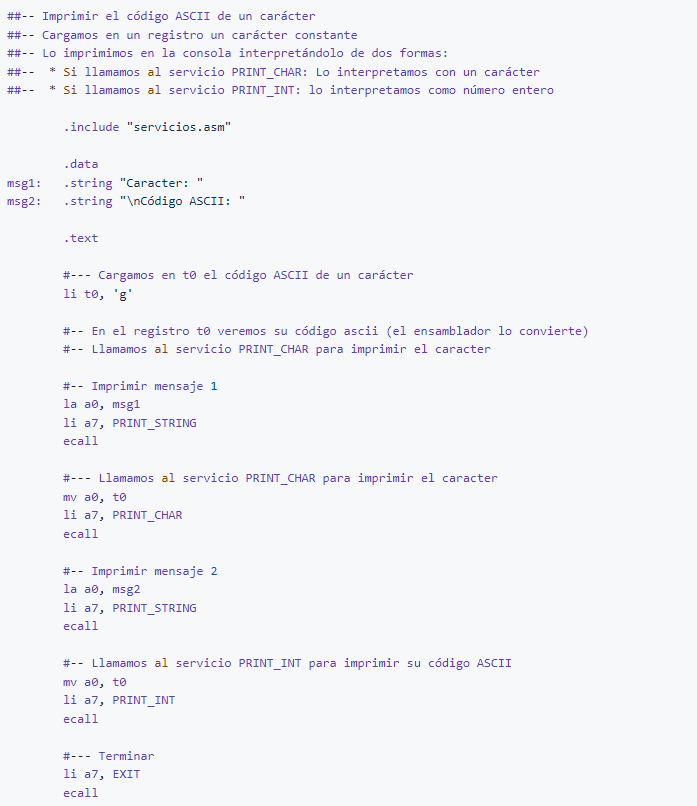
Es importante entender que el procesador sólo opera binario. Pero no sabe que significa lo que hace. Si por ejemplo tenemos 8 bits a uno. Según la interpretación que le queramos dar puede ser una cosa u otra. Lo importante es el significado que nosotros queramos dar a esos bits. Por ejemplo eso puede ser un determinado color, un determinado sonido, etc…



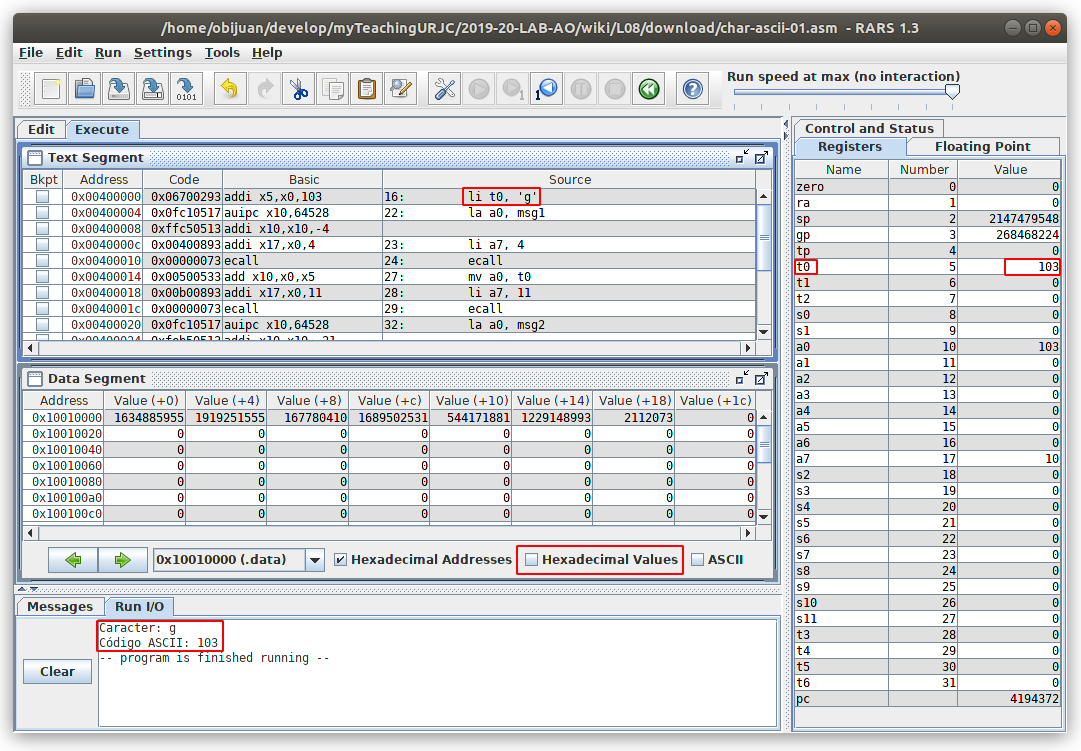
### **Ejemplo: Codificación en ASCII**

Desde nuestros programas en ensamblador podemos **referirnos** al código ASCII de cualquier carácter colocando las **comillas simples**. Así por ejemplo, si cargamos en el registro **t0** el valor 'g', el ensamblador sustituye 'g' por su valor ASCII (103 en decimal, 0x67 en hexa)

En este **ejemplo** lo comprobamos. Cargamos en **registro t0** el carácter 'g' y luego llamamos al servicio **PRINT\_CHAR** para imprimir el carácter y a **PRINT\_INT** para imprimir su **código ASCII**



**Ensamblamos** y **ejecutamos** el programa:



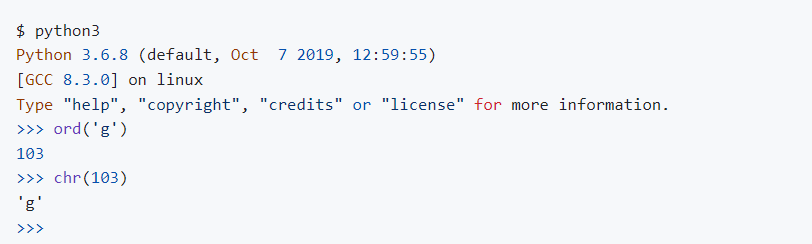
En el programa fuente indicamos que queremos cargar en el registro t0 el carácter 'g'. El ensamblador lo convierte a ASCII, y lo que se **almacena en t0** es **103**. Según al servicio que se llame, este número se interpreta de una forma u otra. El servicio **PRINT\_CHAR** lo **interpreta como ASCII**, e imprime su carácter equivalente: **'g'**. El servicio **PRINT\_INT** lo **interpreta como un número enter**o, imprimiendo en la consola **103**. Pero en todo momento, lo que está almacenado en t0 es el número 103 en binario **01100111** (0x67)

En resumen, un mismo número binario puede tener diferentes significados según la interpretación que le demos.

### **Conversión carácter - código ASCII en python**

El **intérprete de python** se pueden usar para obtener los códigos ASCII de un carácter, o lo contrarios: el carácter que se corresponde con un código ASCII. Simplemente abrimos el intérprete de python desde un terminal y ejecutamos las siguientes órdenes:

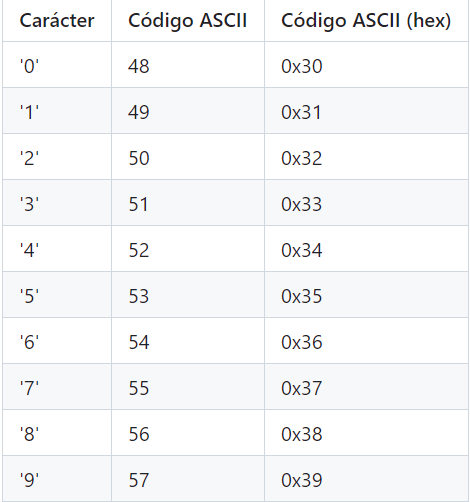
* **ord('g')** : Obtener el código ASCII del carácter g
* **chr(103)** : Obtener el carácter cuyo código ascii es 103



### **Códigos ASCII de los dígitos decimales**

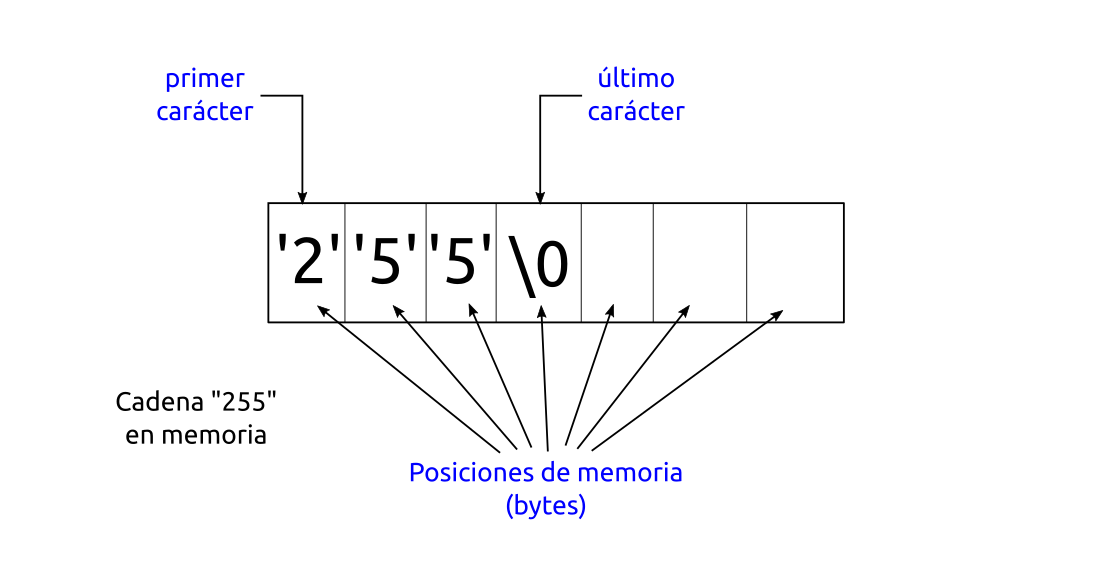
Los **dígitos decimales** 0, 1, 2 ... 9 también son **caracteres**. Y por tanto tienen un **código ASCII** asociado. Así, el número 2, es **diferente** del carácter '2'. El primero es un número (0x02) y el segundo es el código ascii asociado al carácter 2: 0x32

Los **códigos ASCII** asociados los caracteres '0' al '9' se muestran en esta **tabla**:

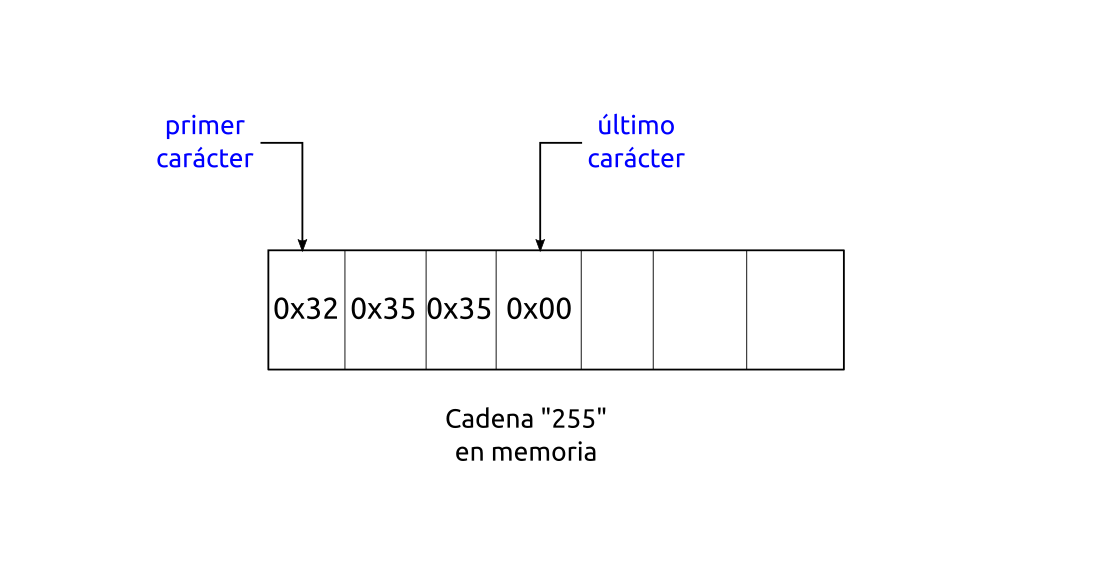


## Conversiones entre cadenas y números

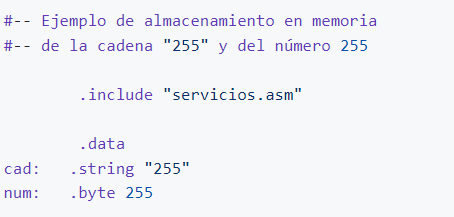
Las **cadenas** pueden contener cualquier carácter. En muchas ocasiones **contienen dígitos decimales**. Así, por ejemplo, la **cadena "255"** está formada por **3 caracteres** más el '\0' final. NO ES UN NUMERO. En memoria está almacenada de esta forma:



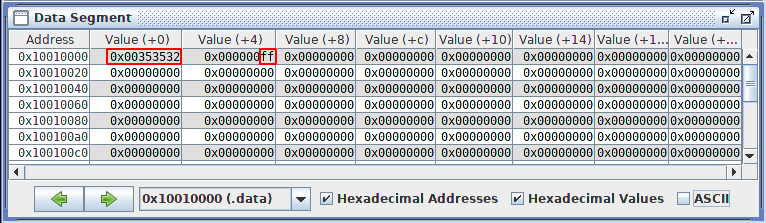
En ese dibujo se han representado los **caracteres entrecomillados**, pero en realidad lo que está almacenado es su **código ASCII**:



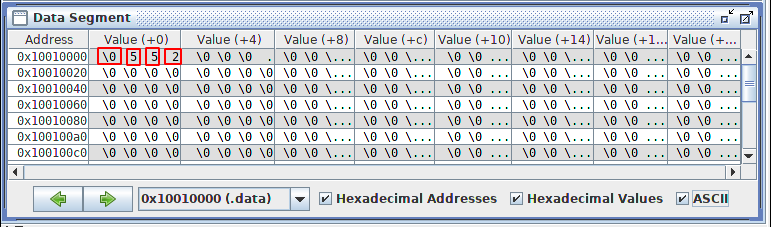
En este **programa de ejemplo** vemos la diferencia. En el **segmento de datos** almacenamos la cadena "255" y a continuación el número 255, que ocupa un byte



**Ensamblamos** el programa y miramos lo que hay en el **segmento de datos**:



Si lo miramos en **ASCII**, veremos los dígitos '2', '5' y '5'



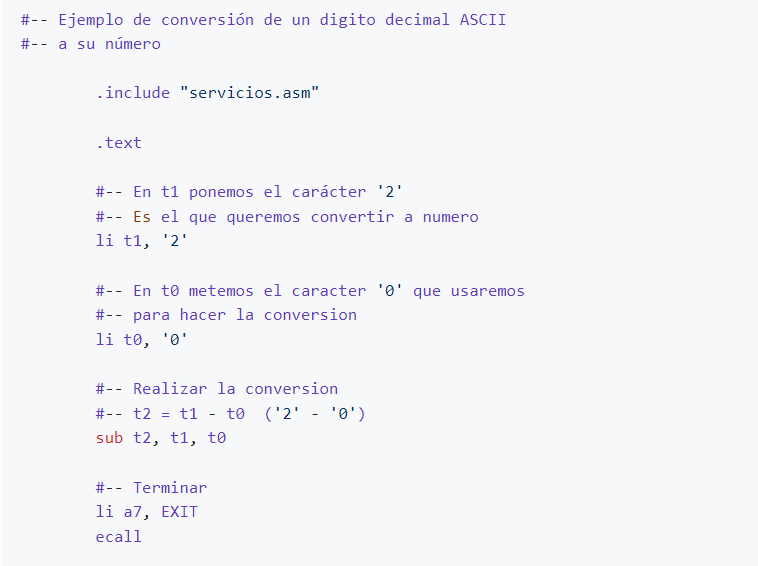
Para practicar la programación del RISC-V, veremos cómo hacer **conversiones** entre **cadenas con dígitos numéricos** y su **número correspondiente**. ¿Cómo transformar la cadena "255" en el número 255?

### **Números de un dígito**

Empezaremos primero con la **conversión de dígitos aislados** en sus **números** correspondientes

Teniendo en cuenta la tabla ASCII de la página 18. Para convertir un **carácter ASCII numérico** en el número que representa sólo hay que restar el valor **0x30**, que se corresponde con el carácter ASCII '0'. Así, si queremos convertir el **carácter '1'** en el **número 1**, hay que hacer esta operación: **'1' - '0'**, que representa la resta de sus códigos ascii: **49 - 48 = 1**

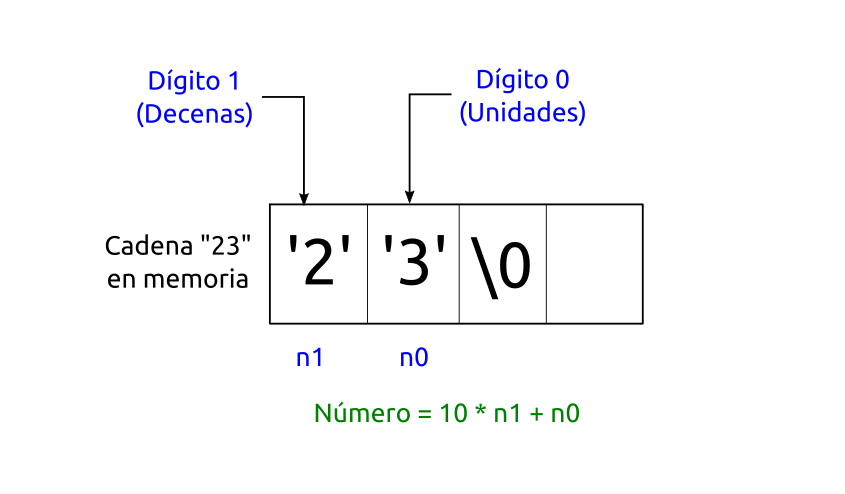
En este programa en ensamblador se muestra un **ejemplo** de conversión del carácter '2' al número 2. El carácter '2' se introduce en el registro **t1** y en el registro **t2** se deja el resultado de la conversión:



### **Números de 2 dígitos**

Para **convertir a número** una cadena con **dos dígitos** hay que multiplicar el dígito de **mayor peso** por **10** (decenas) y sumar el de **menor peso** (unidades)

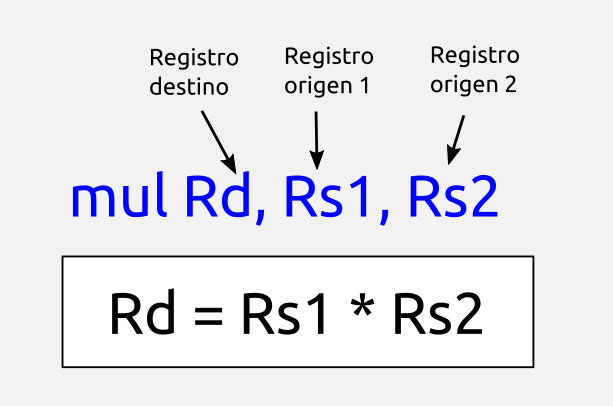
Por ejemplo, si tenemos la **cadena "23"**, en memoria se representaría así:



Como sabemos que tiene **2 dígitos**, directamente podemos calcular la fórmula:

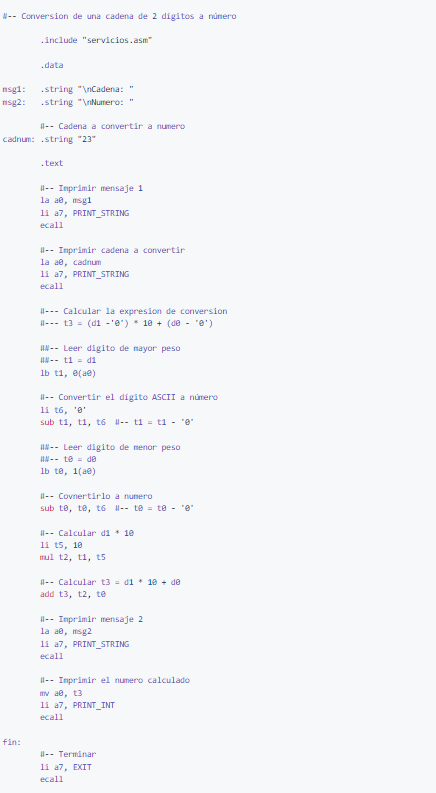
Donde **n1** es el dígito de las decenas convertido a número y **n0** dígito de las unidades también convertido a número. Por el apartado anterior ya sabemos convertir un dígito a su número

La **operación de multiplicación** en el RISCV es **mul**, y tiene la **misma sintaxis** que la suma add



Puede pasar que el juego de instrucciones de tu RISC-V no tenga esta operación. ¿Qué pasa si tengo un código ya hecho con esta operación y lo paso a un RISC que no tenga implementada la operación? No pasaría nada, el diseño de este procesador haría la operación con su juego de instrucciones de forma automática.

En este **ejemplo** se implementa la conversión de la cadena "23" a su número



### **Conversión de cualquier cadena a número**

Para convertir una **cadena genérica**, de cualquier longitud, a un **número** tenemos que **recorrer** la cadena empezando por su **primer dígito** hasta llegar al final ('\0'). El **algoritmo** es el siguiente:

* Inicializar el resultado parcial a cero: num = 0
* Repetir las siguientes acciones (bucle)
  + Leer el carácter actual de la cadena (car)
  + Si car es un salto de línea '\n', entonces hemos terminado. El resultado está en num
  + Multiplicar num por 10: num = num \* 10
  + Convertir el dígito car a su número: n = car - '0'
  + Calcular: num = num + n
  + Apuntar al siguiente carácter
  + Repetir el bucle

### **Manipulación de caracteres**

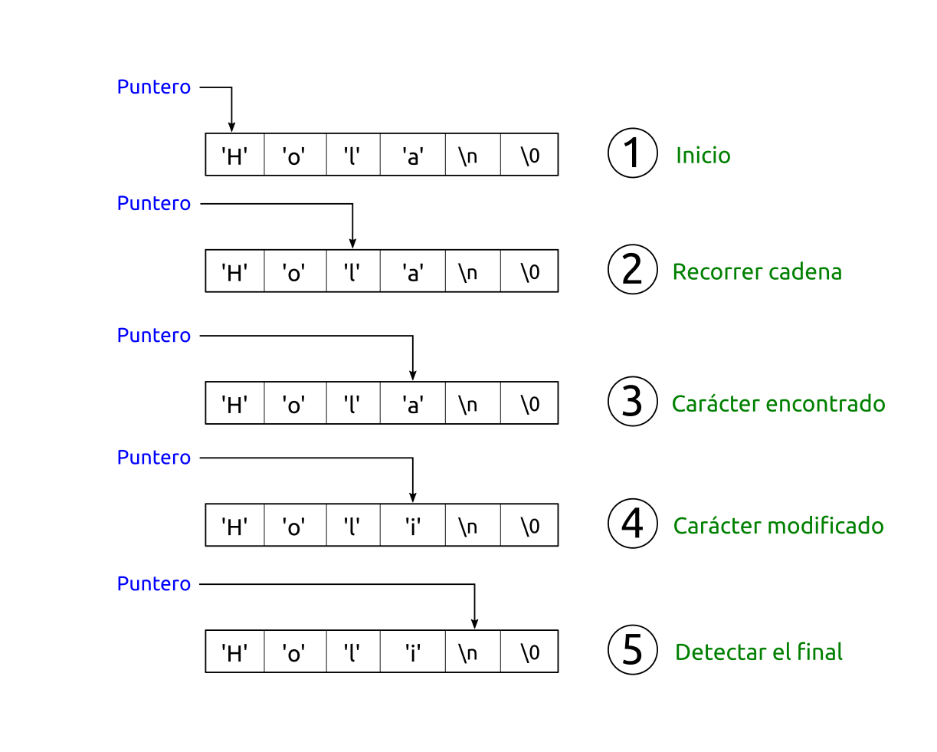
Los caracteres de una cadena se pueden **modificar**, para sustituirlos por otros, o **encontrar** su posición dentro de la cadena. En todos los casos necesitamos un **puntero** que contenga la dirección del primer carácter. Este puntero se va **incrementando** para leer los caracteres (bytes) uno a uno y realizar la operación requerida

Así, si estamos usando el **registro t0** como puntero, las operaciones que típicamente se realizan son:

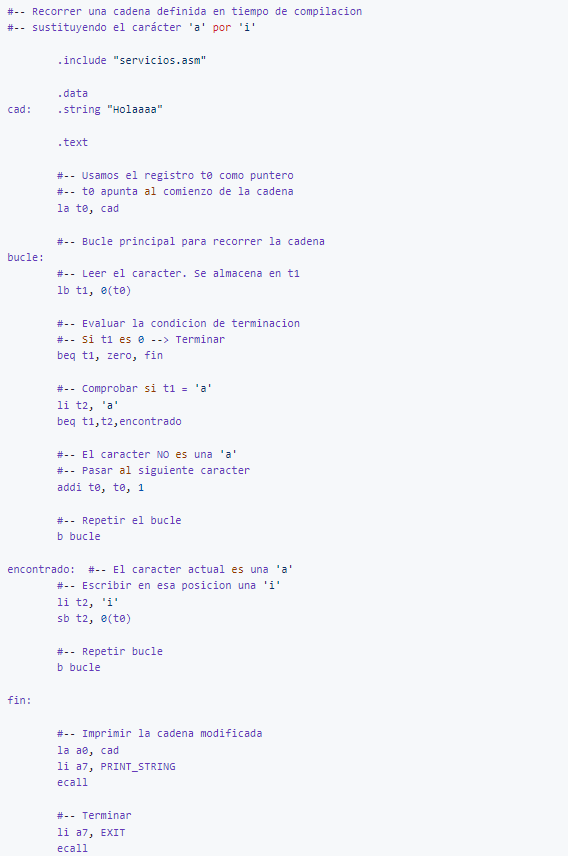
1. **Inicialización**: Asignar a t0 la dirección del primer carácter de la cadena
2. **Recorrer** la cadena: Para pasar al siguiente carácter hay que incrementar el puntero en 1 byte
3. **Leer carácter**: Para leer el carácter usamos **load-byte** (lb). Una vez leído en un registro ya podemos tomar decisiones con las instrucciones de **salto condicional** (beq, bne, bgt...)
4. **Modificación del carácter**: Para cambiar el carácter actual hay que hacer un store-byte (sb). Si en el registro t1 tenemos el nuevo carácter a escribir, usamos la instrucción:
5. **Detección del final**: El final de la cadena lo detectamos bien usando el carácter de terminación '\0' o bien el carácter '\n' que se introduce siempre al final de la cadena (antes del '\0') al invocar el servicio READ\_STRING. Hay que leer el carácter actual, y si es alguno de esos, hemos terminado de recorrer la cadena

En este **ejemplo** se **recorre** una cadena definida en **tiempo de compilación** y se sustituyen todos los caracteres 'a' por 'i'

En este dibujo se muestra el funcionamiento gráficamente:



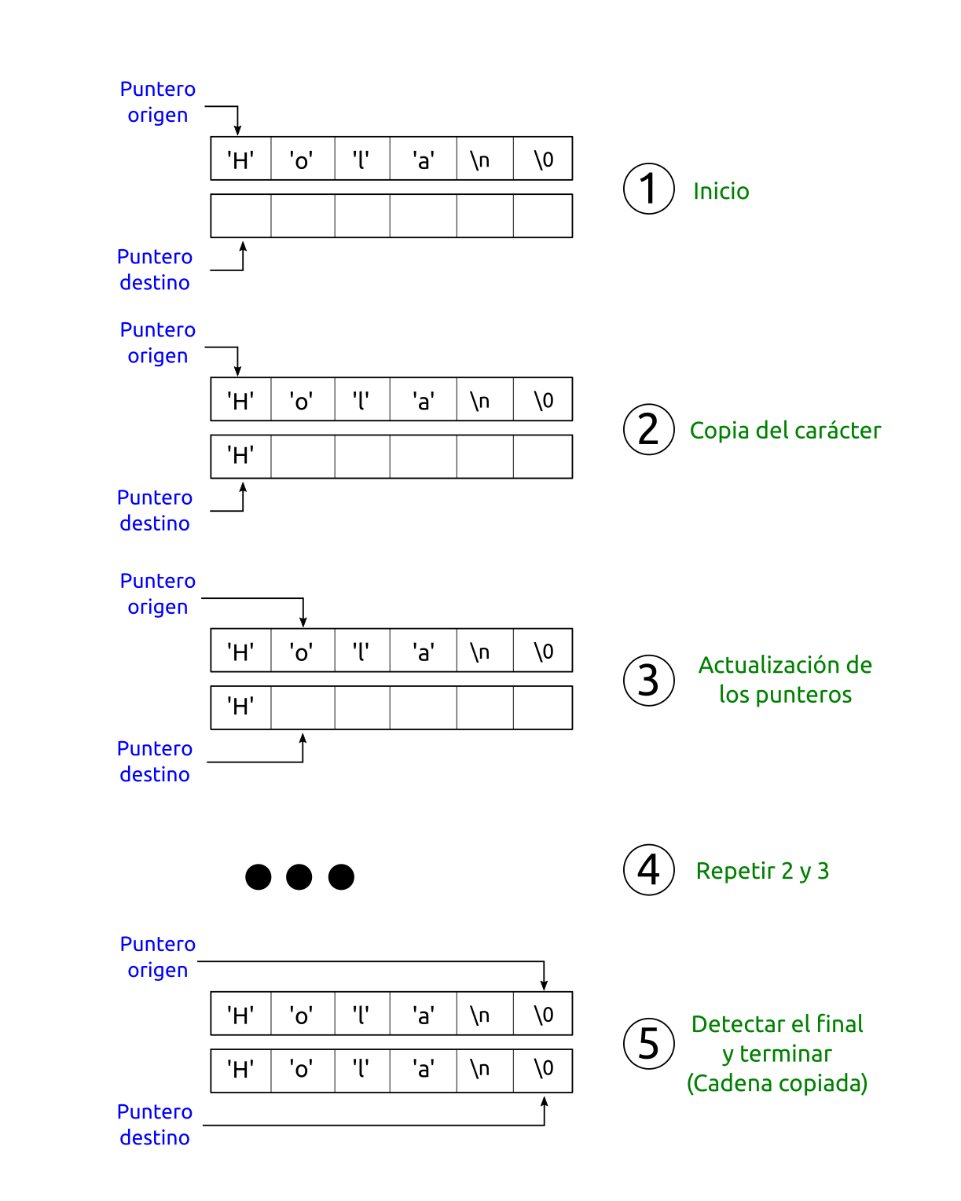
Y este es el código:



### **Copias**

La **copia** o duplicado de cadenas es una operación **muy frecuente**. Para realizarla necesitamos **dos punteros**: uno para la **cadena origen** y otro para la **cadena destino**. Inicialmente apuntan a la posición del **primer carácter**

El proceso de copia se **resume** en este dibujo



Es **muy importante** copiar también el **carácter de terminación** '\0'

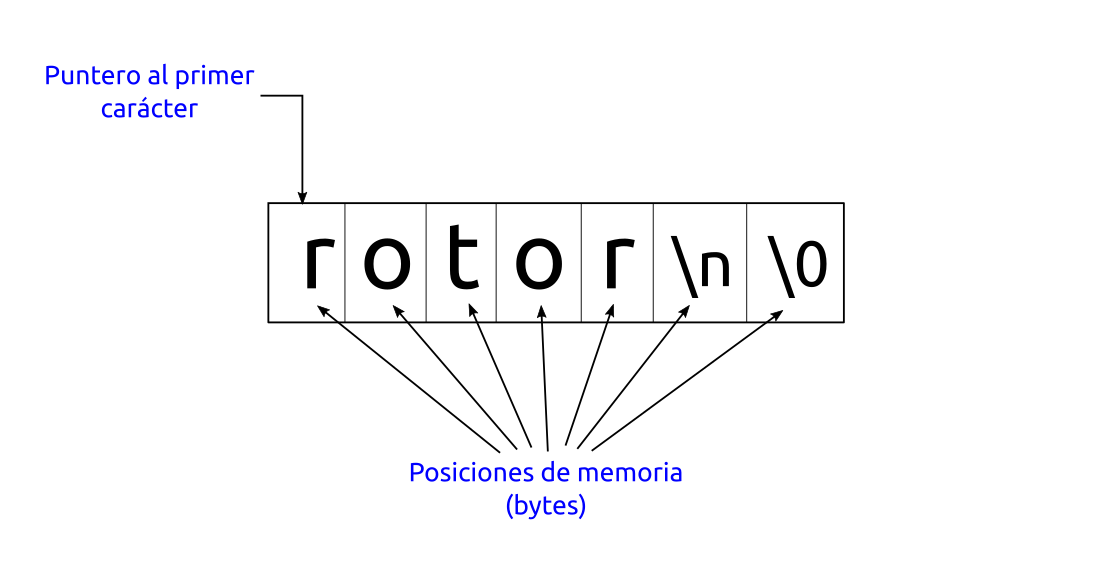
### **Palíndromos**

Un [palíndromo](https://es.wikipedia.org/wiki/Pal%C3%ADndromo) es una palabra que se lee igual al derecho que al revés. Por ejemplo, **rotor** es un palíndromo

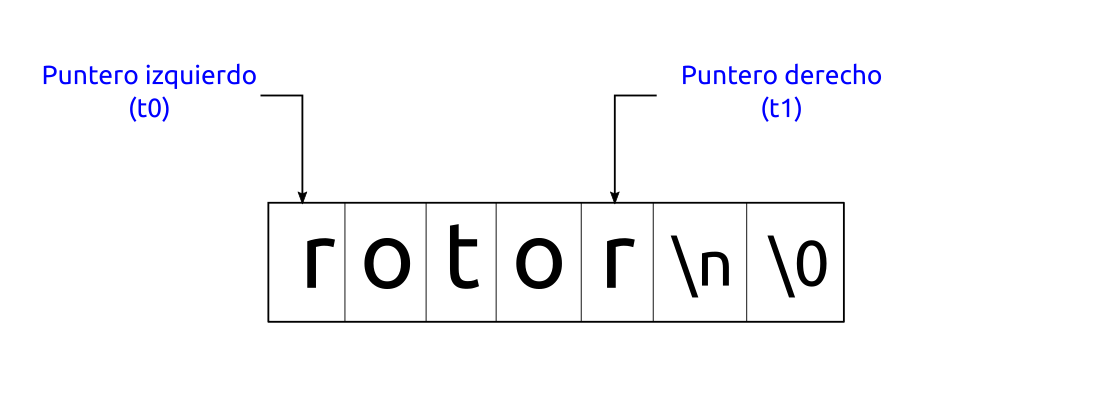
Para **practicar** los bucles y el manejo de cadenas de caracteres, vamos a hacer un programa que **pida al usuario** una cadena y que detecte si es un **palíndromo** o no

Antes de empezar a programar, hay que pensar **cómo resolver el problema** (el algoritmo) y luego se implementa. El **algoritmo** que usaremos se explica a continuación

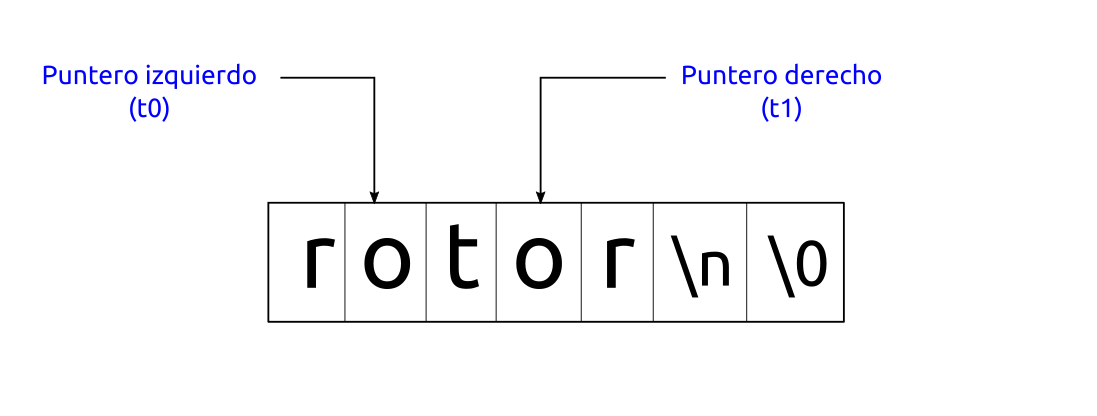
* Una vez que el usuario **ha introducido la cadena**, quedará **almacenada en memoria** y tendremos un puntero que apunta al primer carácter



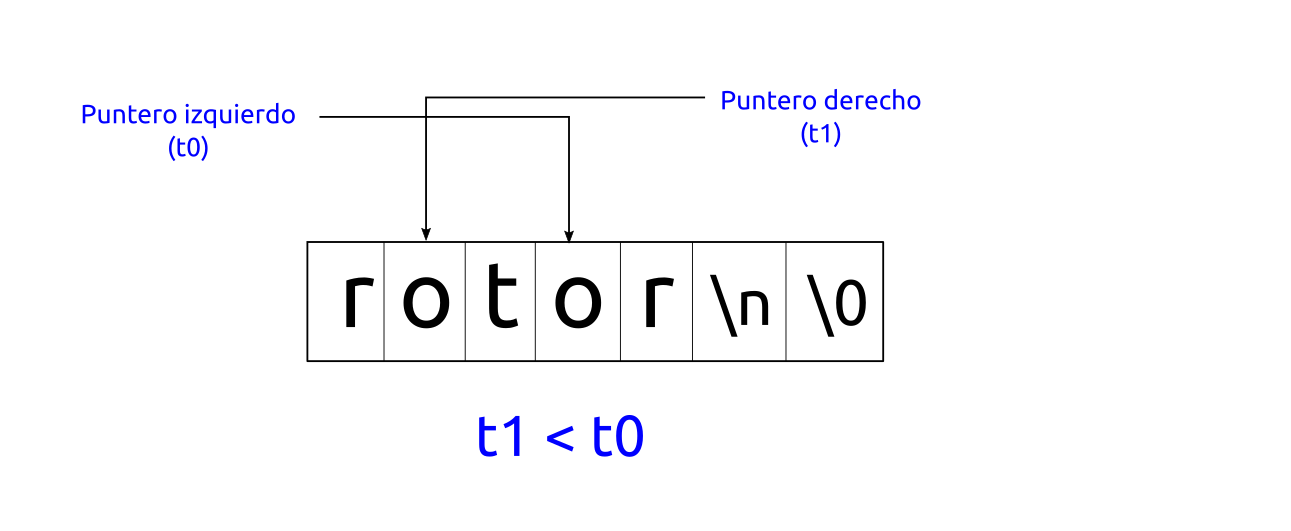
* Utilizaremos **dos punteros**, uno que apunte al **carácter izquierdo** y otro al **derecho**. Para obtener el puntero derecho, copiamos el puntero inicial y lo incrementamos hasta llegar al '\0' (detectar el final de la cadena) y luego retrocedemos 2 caracteres, para apuntar al carácter de la derecha



* **Leemos** los **caracteres** apuntados por ambos punteros. Si son **diferentes**, la palabra **NO es un palíndromo**, por lo que terminamos. Pero si son **iguales**, hay que **actualizar los punteros** para apuntar a los siguientes caracteres. El puntero izquierdo lo **incrementamos**, y el derecho lo **decrementamos**



* **Repetimos** la operación anterior. Cuando llegamos a la situación en la que los punteros "se cruzan" (el derecho es menor o igual que el izquierdo) entonces sabemos que la palabra **SÍ es un palíndromo**, y terminamos



Este es el **código**

