Para el aprendizaje de la programación, he instalado UBUNTU en el portátil y he descargado el simulador Rars. Con este aprenderemos la programación en ensamblador.

Fuente: Seguiremos el tutorial de ObiJuan

<https://github.com/myTeachingURJC/2019-20-LAB-AO/wiki/L1:-Practica-1>

Lo primero que aprendemos es que en un programa, después del .text, viene el código. Luego profundizaremos mas en ello, pero por ahora sabemos que:

Con li a7, 10

Ecall

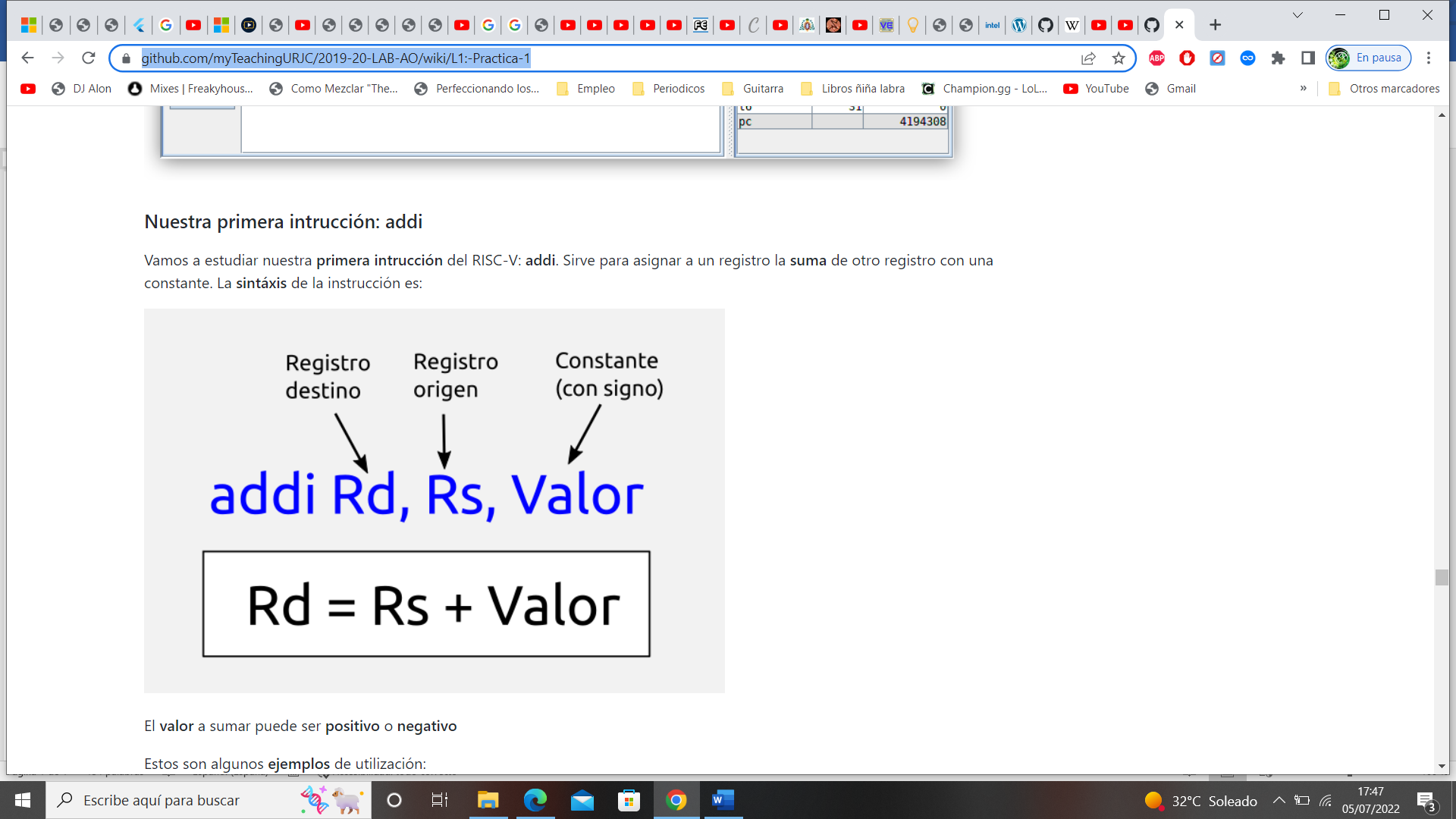
Le decimos al programa que finalice

El chip el cuál se simula es de 32 bits, lo que quiere decir que tiene 32 registros, numerados del 0 al 31. El cero es un registro especial, pues su valor siempre será cero y NO se le puede asignar otro valor.

El registro PC (Contador de programa) es otro registro especial que guarda la dirección de la siguiente instrucción a ejecutar.

**Primera instrucción: Addi**

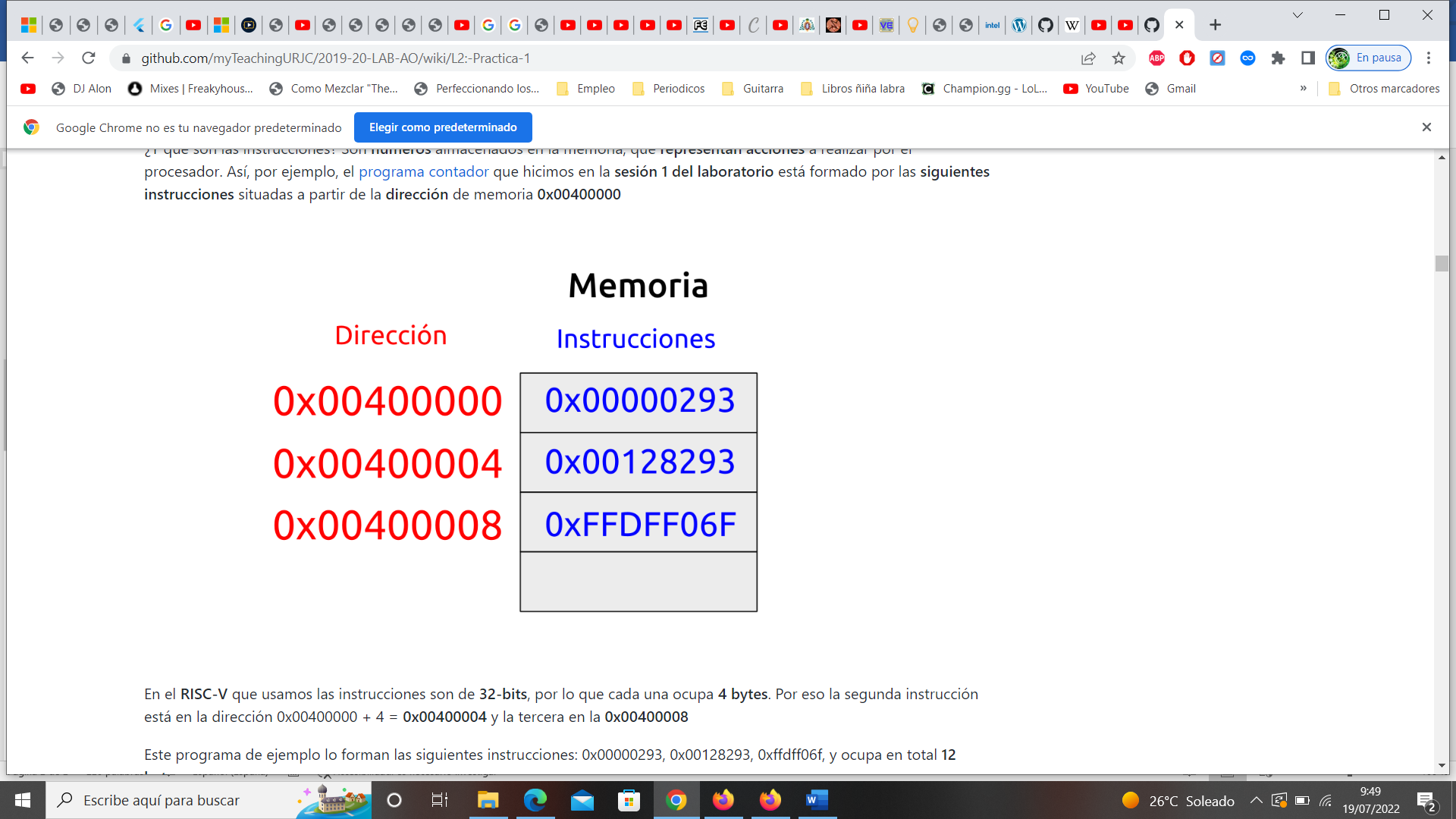
Asigna a un registro la suma de otro registro con una constante.



El valor a sumar puede ser positivo o negativo. Ej: Addi x2,x5,30

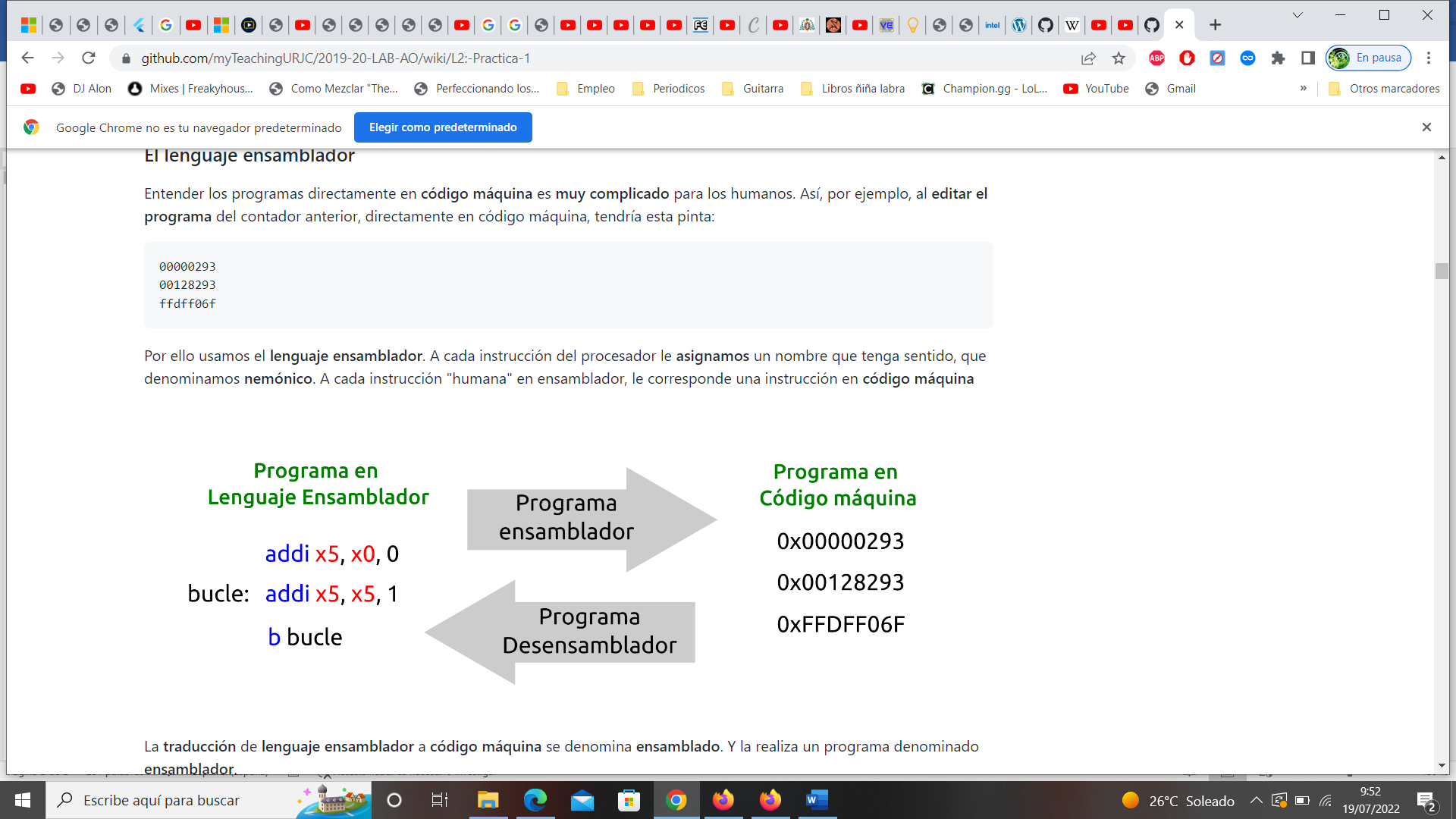
Para continuar debemos de saber cómo funciona un procesador. Lo que hace un procesador básicamente es ejecutar instrucciones, tú le das instrucciones y el las ejecuta, todo el tiempo está ejecutando. Él va leyendo las instrucciones de una memoria en la que se encuentran ordenadas en direcciones consecutivas.

¿Y que son las instrucciones? Son números almacenados en la memoria y que representan acciones.



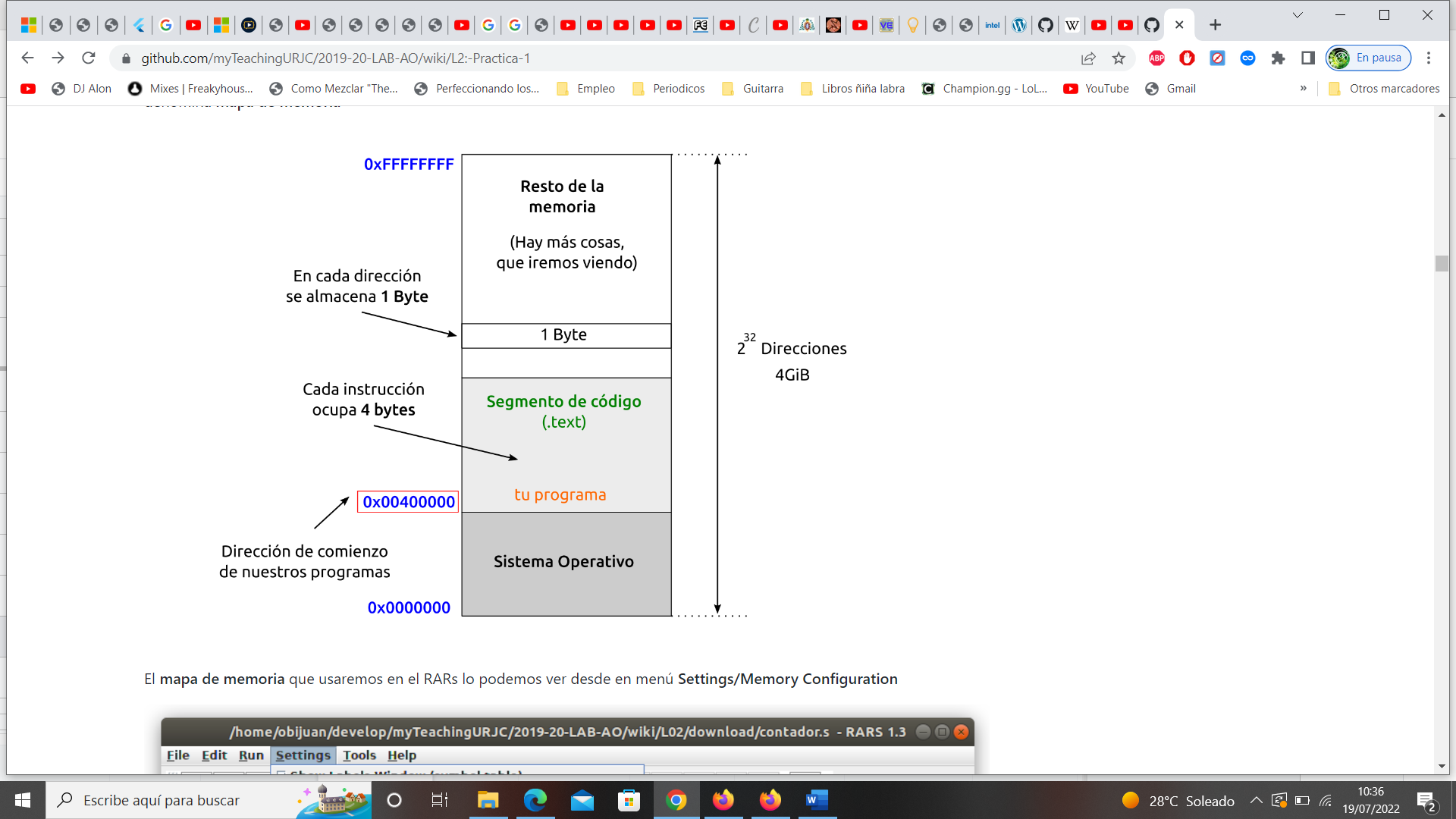
En el Risc-V que usamos, las instrucciones son de 32 bits (4 bytes) por eso las direcciones van de 4 en 4.

A cada instrucción en ensamblador se le denomina nemónico.

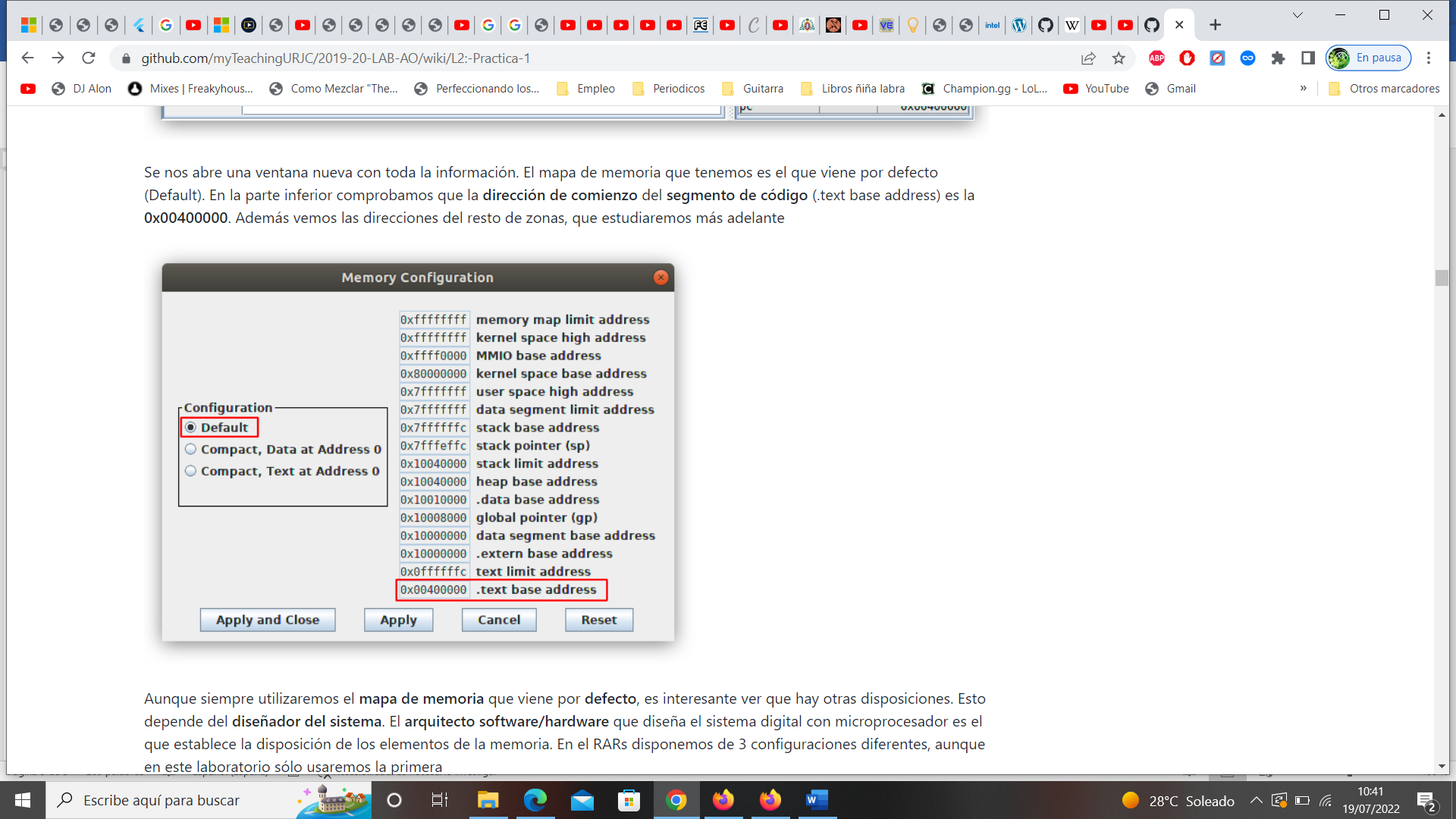


Mapa de memoria.

En el Risc-V tenemos 2 elevado a 32 direcciones en la memoria. Las direcciones donde se almacenará el programa se llama segmento de código, que en el caso de Risc-V empieza en la dirección 0x00400000.



Así se verían las direcciones de comienzo de los distintos sectores de memoria del RISC-V



Para verlo vamos a settings->Memory configuration.

El funcionamiento del procesador es sencillo. Las instrucciones son números que se encuentran almacenados en una memoria en direcciones consecutivas. Pues el procesador lo que hace es ir en orden, leyendo y ejecutando las instrucciones. Su comportamiento es leer y ejecutar instrucciones sin parar.

Nota: Hay instrucciones que modifican el contador de programa, en este caso la siguiente instrucción podría no ser la consecutiva.

**Instrucciones**

Las instrucciones en el RISC-V son de 32 bits (4 bytes) es por ello que las direcciones van de 4 en 4, puesto que cada instrucción ocupa 4 direcciones consecutivas. Es importante que estén alineadas. Veremos que es esto más adelante.

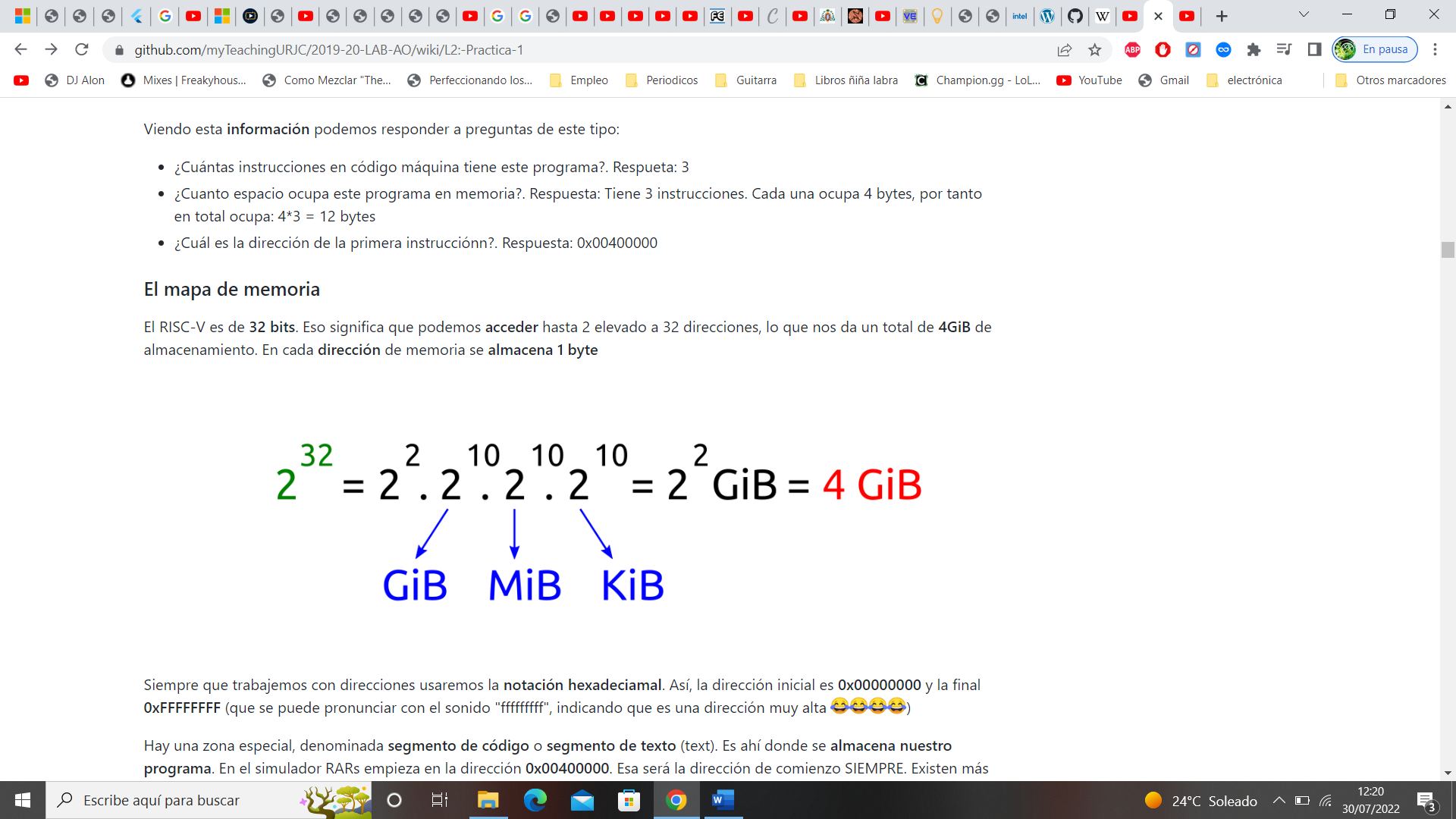
**Nemónico:** Nombre asignado a una instrucción de máquina para nuestro entendimiento. Esto es el lenguaje ensamblador.

**Procesador de 32 bits:**  Esto es que tiene 32 registros y sus instrucciones son de 32 bits. Entonces si por ejemplo tenemos 5 instrucciones y cada instrucción son 32 bits, que son 4bytes, 5x4=20 bytes de memoria para 5 instrucciones.

En los procesadores Risc las instrucciones suelen tener la misma longitud, en cambio de los Cisc no.

**Mapa de memoria**

Para saber el número total de Bytes que podemos almacenar en memoria, es decir, el número de posiciones de memoria a las que podemos acceder, para saber ese espacio, hacemos el siguiente cálculo.



32 es porque es un procesador en este caso de 32 bits. 2 elevado a 10 es 1024, que, para ingeniería, es lo mismo que el kilo para el sistema métrico internacional.

Con lo cual tendríamos 4GB de memoria.



De las diferentes partes que tiene en la memoria nuestro procesador, existe una (.text) que es el segmento de código, donde se almacena el programa. En este caso para el RISC-V empieza en la dirección 0x00400000. Cada procesador tiene su manual donde te dirá como está organizado su mapa de memoria.

Si nos fijamos en el último dígito de cada dirección que hay por cada instrucción, **SIEMPRE** es múltiplo de 4, a esto se le llama estar alineada. Por lo cuál siempre terminará en 0,4,8 o C.

**La directiva .text**

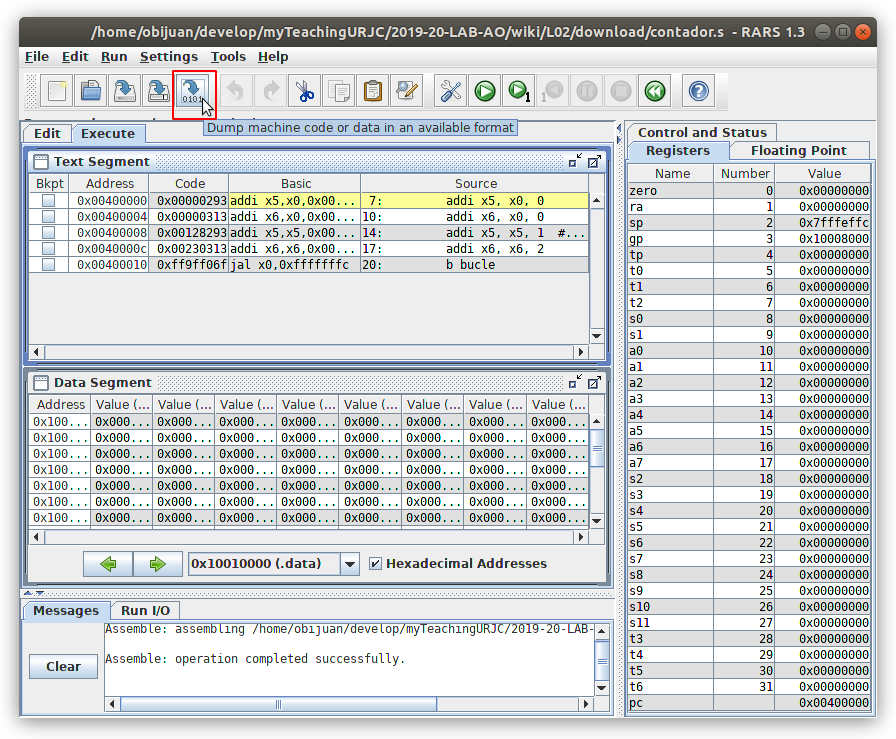
Las directivas no son instrucciones, si tu las ejecutas no generan código máquina ni ocupan espacio. Son comandos para indicar al ensamblador donde tiene que escribir lo que va a continuación. Por lo general, actualmente, aunque no lo pongas, el ensamblador es lo suficientemente listo para saber a que sección debe ir, pero como aprendizaje nos viene bien ponerlo.

La directiva .text, indica al ensamblador, que lo que se escribe a continuación va en el segmento de código.

El **codigo máquina** de nuestros programas lo podemos **exportar** a un **fichero**. Esto es muy útil al trabajar con **sistemas reales**, a los que hay que cargarles el código máquina (el programa ejecutable) para que lo ejecuten

Si tenemos **una placa con un procesador RISC-V** podremos ejecutar nuestros programas en ella. Primero **simulamos** con el RARs para comprobar que todo funciona bien, luego **exportamos** el código máquina y lo **cargamos** en la placa

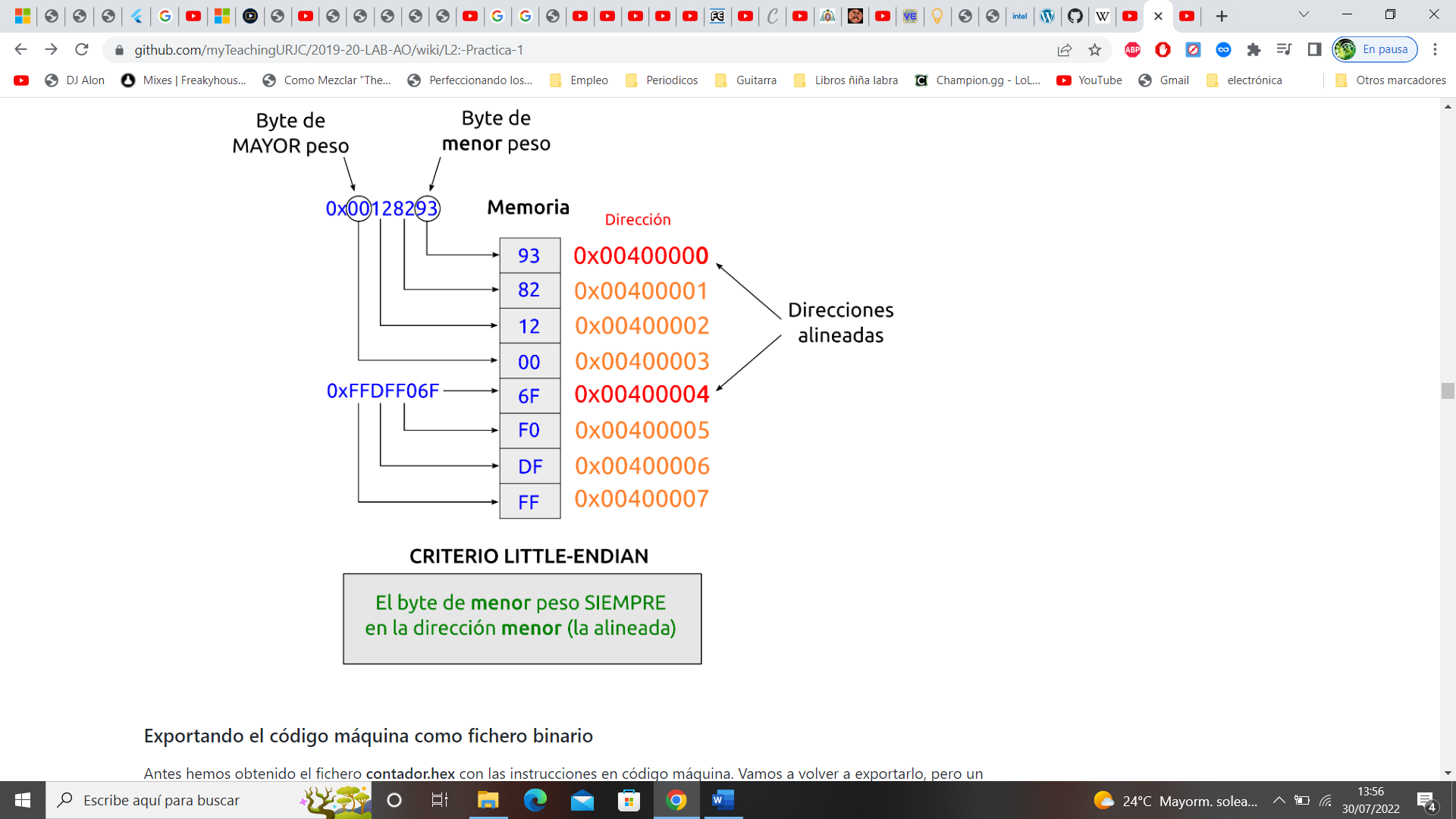
Para **exportar el código máquina** primero es necesario **ensamblar** el programa y luego pinchar en el botón **Dump machine code** en la barra superior



Nos aparecerá una ventana donde elegimos el formato para el fichero ejecutable. Vamos a seleccionar en el desplegable de la derecha el formato **Hexadecimal Text**. Son las instrucciones en código máquina en caracteres ASCII para poder abrirlos con un **editor de textos** cualquiera

**LITTLE ENDIAN**

Pues bien, sabiendo que para el RISC-V cada instrucción ocupa 32 bits, lo cual son 4 bytes, con lo cual son 4 direcciones. ¿Dentro de esas 4 direcciones, como se ordena la instrucción? Esto depende del fabricante y del convenio que tenga. Nuestro dispositivo utiliza el convenio Little Endian. Este convenio nos dice que en la primera dirección irá el byte de menor peso y en la ultima dirección el byte de mayor peso.



Hay un convenio que se llama big endian que es al revés, el primero es el de mayor peso.

Nota: Para leer archivos .bin, es necesario un programa especial, en Linux por ejemplo está el GHex, o el hd en la consola.

**Etiquetas**

Para definir una etiqueta, ponemos un identificador (un nombre) seguido de dos puntos. Ej; bucle:

Una etiqueta no es más que una dirección de memoria. Por ejemplo, tenemos una dirección de memoria 0x4000008 que contiene una instrucción addi x5,x5,1, entonces, para acceder a esa dirección de memoria, como recordar el número es más difícil, lo que hacemos es asignarle una etiqueta a esa dirección, de manera que cada vez que llamemos a la etiqueta, nos direccionará a la dirección que tenga asignada.

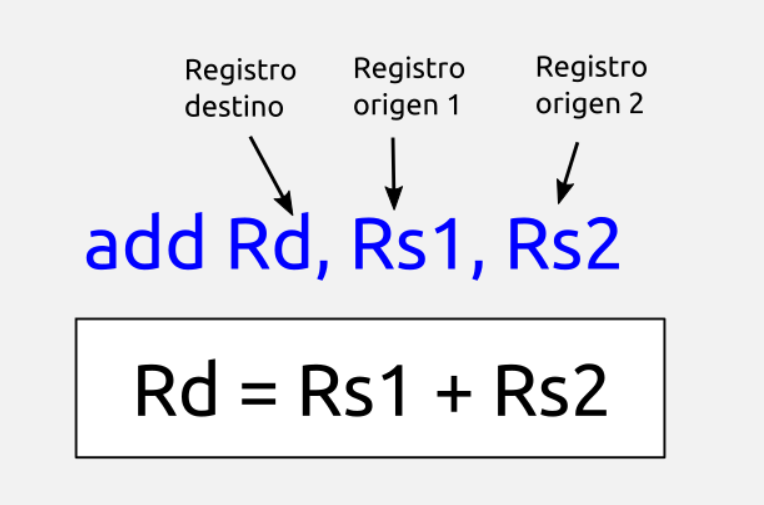
Si necesitásemos referirnos a esta dirección en alguna parte de nuestro programa, sólo hay que colocar la etiqueta sin los dos puntos al final. El ensamblador la sustituirá por la dirección correcta. Por ejemplo: b buble. Donde la b es de Branch, lo que le indica que salte a la dirección indicada por la etiqueta bucle. Es decir que salte a ejecutar la instrucción situada en la dirección 0x4000008 que es addi x5,x5,1.

Podemos definir tantas etiquetas como queramos.

**Instrucción addi**

La instrucción addi es siempre necesaria al principio del código para inicializar las variables que vayamos a usar. SIEMPRE QUE VAYAMOS A USAR UNA VARIABLE LA TENEMOS QUE INICIALIZAR PRIMERO.

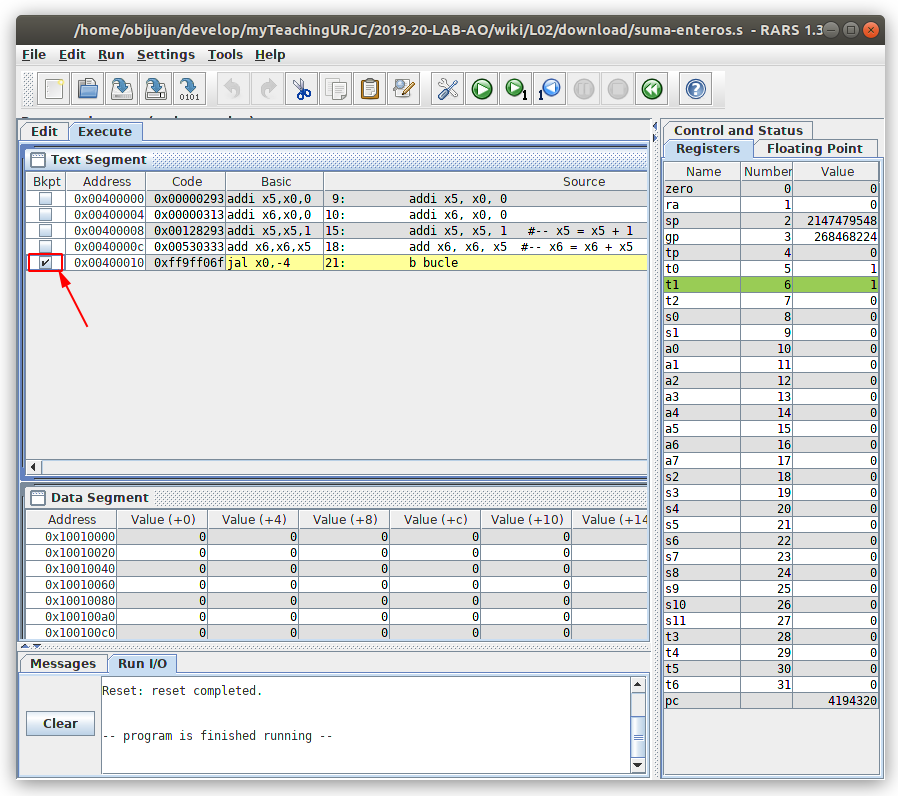
**Instrucción add, sumar registros.**

****

* **Almacenar** en el registro **x5** la suma de los registros **x3** y **x4**:

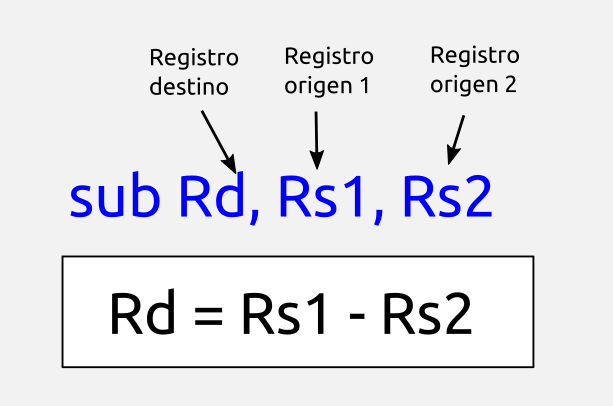
***BreakPoin***

Para facilitar la **depuración del código**, vamos establecer un **punto de rupturo** (Breakpoint) en el lugar en el que estamos parados ahora (la instrucción b bucle). De esta forma podemos ejecutar el programa, y cada vez que llegue a ese punto la ejecución se detiene, permitiéndonos analizar la situación. Para **activar el Breakpoin**t hay que pulsar sobre la **casilla de verificación** de la izquierda de la **instrucción b bucle**



**Restando registros. Instrucción sub**

La instrucción para **restar registros** tiene la **misma sintáxis** que la instrucción add:



### Calculando expresiones con sumas y restas

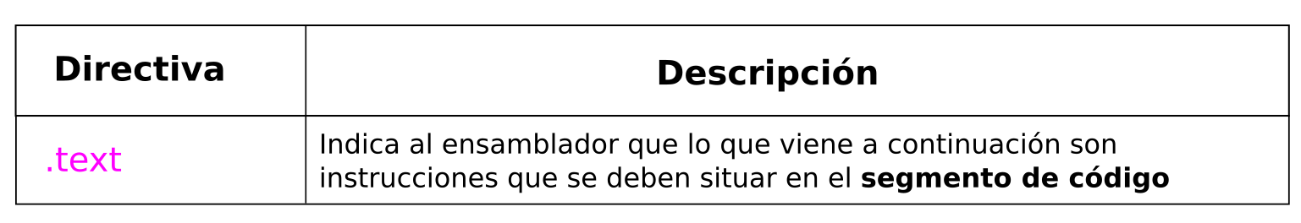
Con las tres instrucciones que ya conocemos: **addi**, **add** y **sub**, podemos implementar **programas en ensamblador** que calculen el valor de **expresiones genéricas** con **sumas** y **restas**, como por ejemplo esta:

## Recopilación de instrucciones hasta el momento

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



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



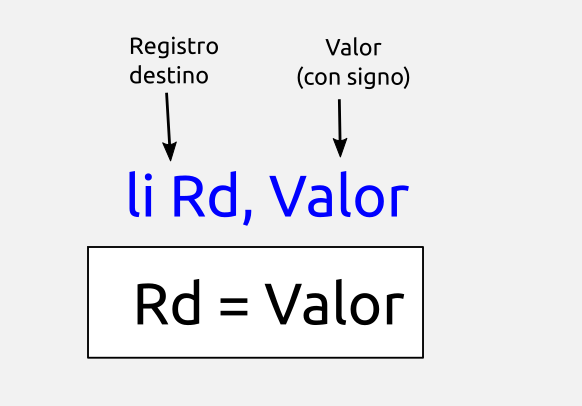
## **Pseudoinstrucciones li y mv**

Para **facilitar** la labor del programador, además de las **instruccions básicas**, se definen **pseudoinstrucciones**. Se trata de instrucciones que el **programa ensamblador**, es decir que no las entiende el RISC-V, pero el programa ensamblador que usemos si, y es este el que se encarga de traducirlas para que las entienda el procesador, convierte a instrucciones reales del RISC-V. Hay dos que utilizaremos mucho: **li** y **mv**

### li: Carga de un dato inmediato

Hemos aprendido que para **cargar un dato** en cualquier registro lo hacemos con la instrucción **addi**. Así por ejemplo, si queremos inicializar el registro **x5** con el valor **30**, hacemos:

Esta operación de cargar datos en un registro es **MUY HABITUAL**. Por eso se ha creado una **pseudoinstrucción** para ello: **li** (load inmediate). Tiene el siguiente formato:



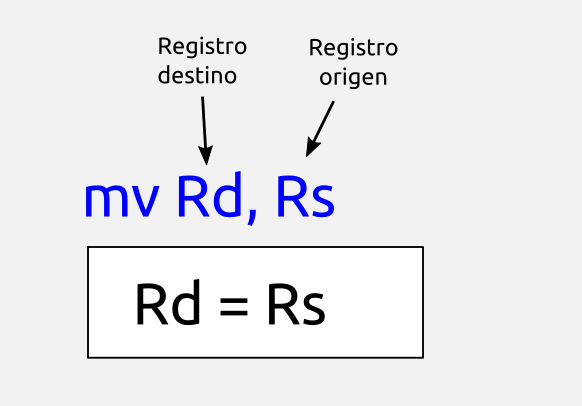
El **ejemplo** de la asignación anterior se haría así:

El programa **ensamblador** detecta que se trata de una pseudoinstrucción y la traduce a la **instrucción real**: addi x5, x0, 30

### mv: Mover valores entre registros

La operación de **transferir** un datos de un registro a otro la realizamos con la instrucción básica **add**, haciendo que uno de los registros fuente sea el **x0**. Así, para mover el contenido del registro **x5** al registro **x10** haríamos:

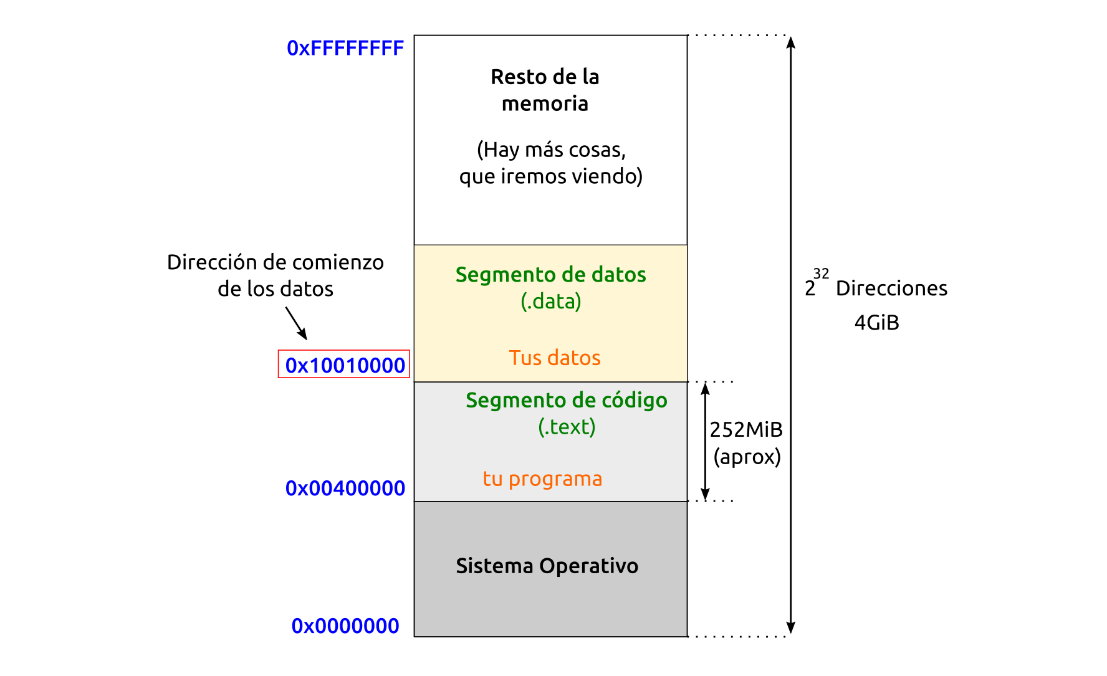
Esta operación es tan **frecuente**, que se ha creado la **pseudoinstrucción mv** con ese mismo fin. Tiene este **formato**:



La operación anterior se realiza así:

## **El segmento de datos (data)**

Los programas, además del código, tienen **datos**. Se almacenan en una región de la memoria que se llama **segmento de datos** (data) que comienza a partir de la dirección **0x10010000**, justo después del segmento de código. El **mapa de memoria** es así:



Fíjate que los datos se almacenan de forma **separada al código**, en zonas diferentes de la memoria. El espacio asignado para el código es de aproximadamente 252MiB (Mebibytes)

### Inicializando el segmento de datos con palabras

Para **colocar datos** en el segmento de datos, de forma que estén almacenados para usarlos después desde nuestros programas, usamos primero la **directiva .data**. Con esto le decimos al **ensamblador** que lo que viene a continuación son **datos**, y que por tanto se deben almacenar en el **segmento de datos**

Usando la directiva **.word** indicamos el **dato** (una palabra de 32 bits) que se almacenará en esa posición

En este **programa de ejemplo** se inicializa la **primera posición** del **segmento de datos** con el valor hexadecimal **0xBEBECAFE**

#-- Programa de ejemplo que inicializa la primera palabra del

#-- segmento de datos a un valor

#-- Indicar al ensamblador que lo que viene a

#-- continuacion esta en el segmento de datos

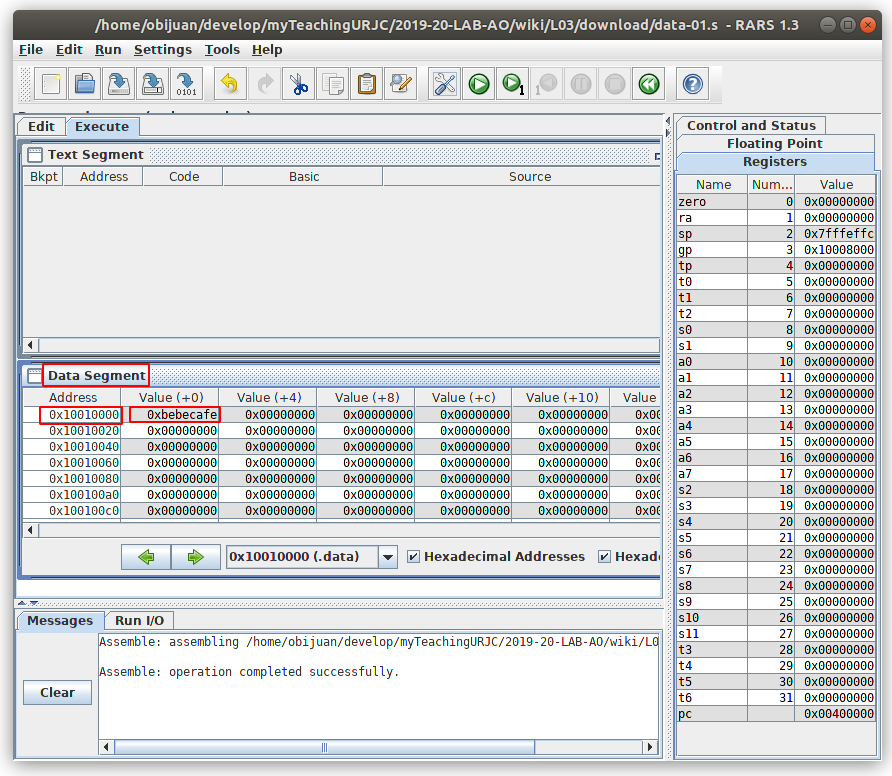
.data

#-- Dato a colocar en el segmento de datos

#-- (en la primera posicion)

.word 0xBEBECAFE

Este programa **NO** tiene código, por lo que **NO** se generará ningún código máquina. Pero **SI** tiene datos, que deberán estar en el segmento de datos. Cargamos el programa en el **simulador** y le damos a **ensamblar**

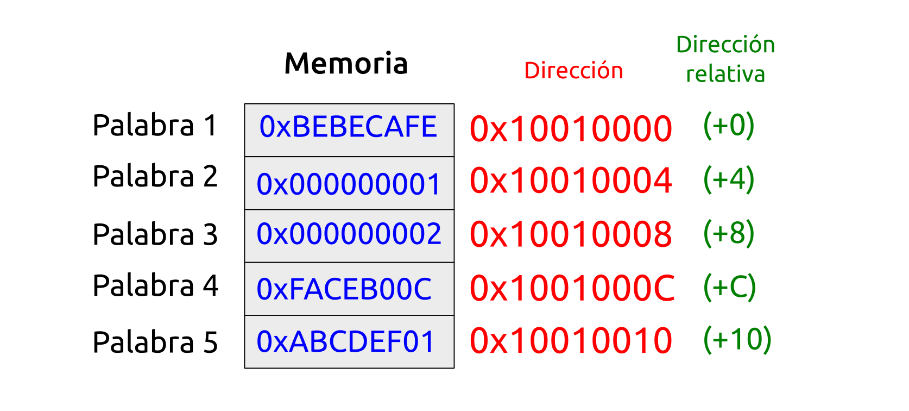


En la ventana con la información del **segmento de código** vemos que efectivamente **NO hay nada**. Está vacío. Nos fijamos en la ventana que está debajo: es la **ventana del segmento de datos**. La identificamos porque tiene el título **Data segment**

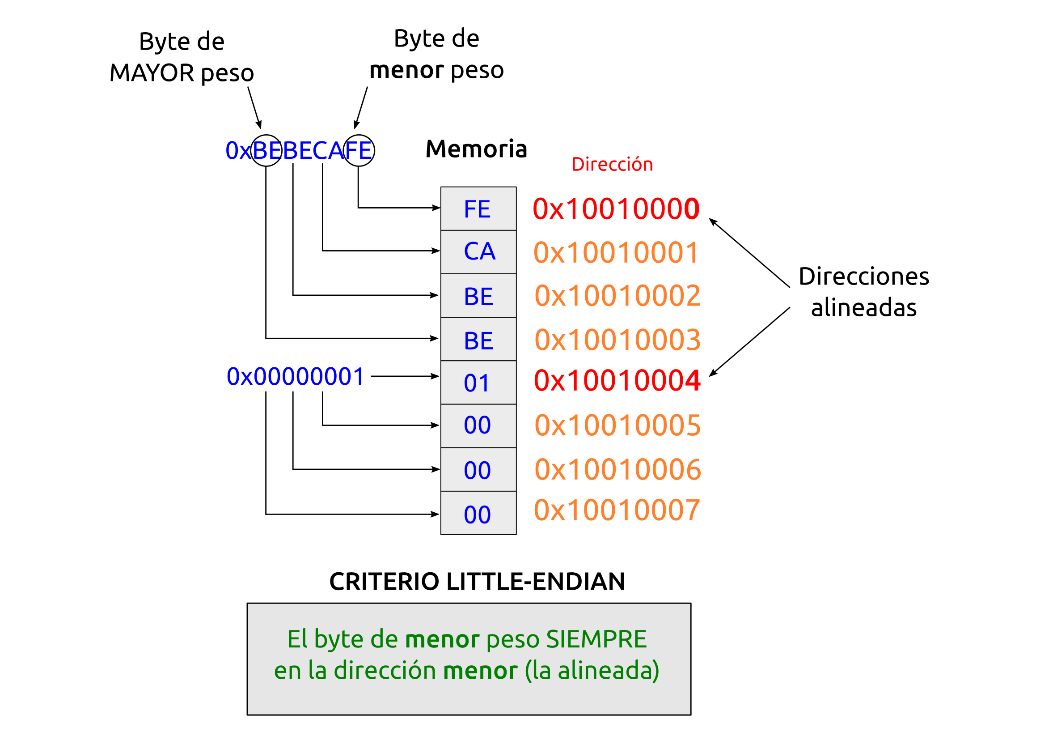
En la **izquierda** están las **direcciones**, empezando en la **0x10010000**, que ya sabemos que es la **dirección inicial** del **segmento de datos**. En la derecha se encuentran las palabras (de 32 bits) almacenadas en esas posiciones de memoria. Identificamos la que hemos colocado: **0xBEBECAFE**

Para introducir **más palabras** se pueden añadir en la misma directiva .word separas con comas o bien colocar nuevas directivas .word. Utilizaremos ambos métodos para situar **cinco palabras** en el segmento de datos

En memoria las palabras SIEMPRE se guardan en direcciones alineadas. Si no es así te va a saltar un error.



**Little Endian**: los bytes de menor peso siempre en la dirección menor



### Usando etiquetas con los datos

Para **acceder a los datos** necesitamos conocer la **dirección** en la que se encuentran. Pero en vez de utilizar números, usamos **etiquetas**. El ensamblador se encarga de **asignar** a cada etiqueta su dirección correspondiente

En el siguiente programa se declaran **5 variables**, inicializadas con los valores 1,2,3,4 y 5. Para referirnos a ellas usaremos las etiquetas a, b, c, d y e respectivamente

#-- Ejemplo de uso de etiquetas en el

#-- segmento de datos

.data

#-- Almacenamos 5 variables en memoria

#-- inicializadas con los valores 1,2,3,4 y 5

#-- Cada variable es una palabra, almacenada en una

#-- direccion de memoria (alineada)

a: .word 1

b: .word 2

c: .word 3

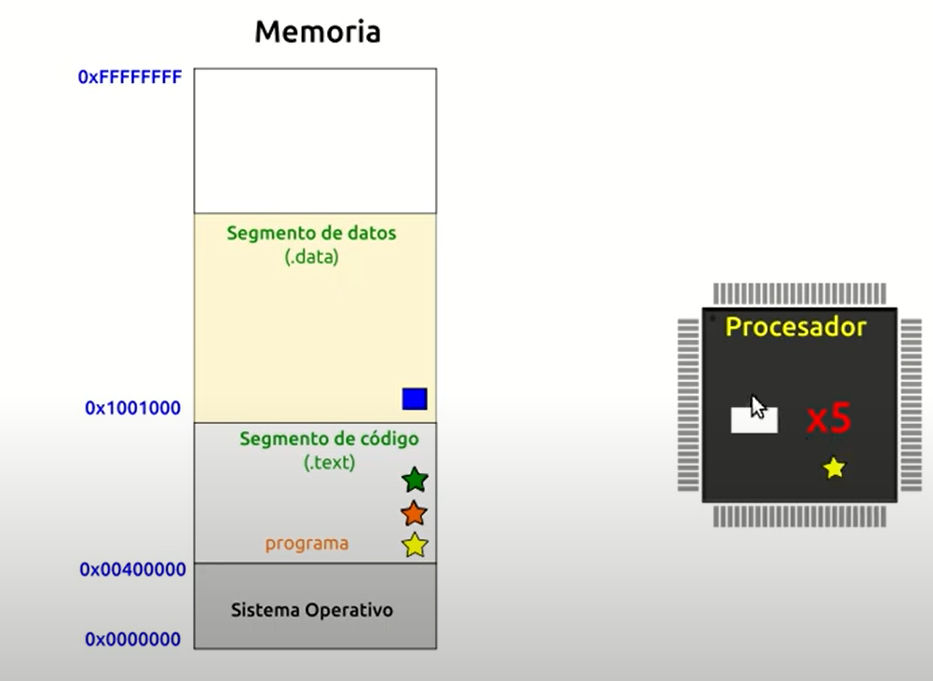
d: .word 4

e: .word 5

Activando la **tabla de símbolos**, comprobamos qué direcciones le corresponde a cada una



Es importante recordad que la memoria está en un lugar físico distinto al procesador. Tendríamos por un lado el procesador y por otro lado la Memoria, que podría estar en otro chip o en varios chips.



## Instrucciones de acceso a memoria: load y store

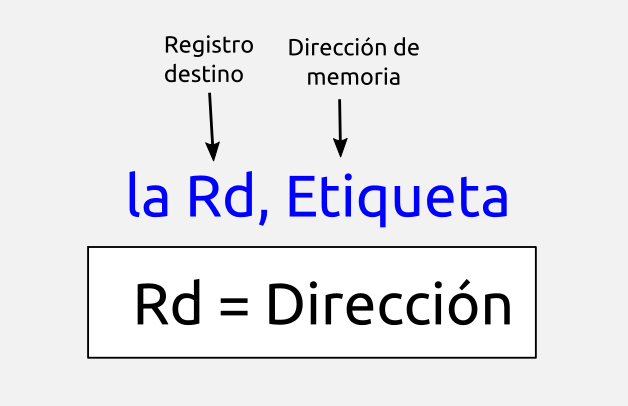
Nuestros programas usan el **segmento de datos** para **leer datos** y **almacenar resultados**. Sólo se puede acceder a memoria usando dos tipos de instrucciones, denominadas **load** y **store**. Los **valores transferidos** pueden ser **palabras** (32 bits), **medias palabras** (16 bits) o **bytes**

* **Instrucciones de load**: Se usan para leer de la memoria un valor y guardarlo en un registroes
  + **lw** (load word): Leer una palabra
  + **lh** (load half-word): Leer una media palabra
  + **lb** (load byte): Leer un byte
* **Instrucciones store**: Almacenar un registro en una posición de memoria
  + **sw** (store word): Almacenar una palabra
  + **sh** (store half-world): Almacenar una media palabra
  + **sb** (store byte): Almacenar un byte

La **dirección a la que se accede** tiene que estar **situada en un registro**, al que se le **suma** un **desplazamiento** (offset), que puede ser 0. Sólo se puede acceder a la memoria de esta forma (no hay más modos de direccionamiento)

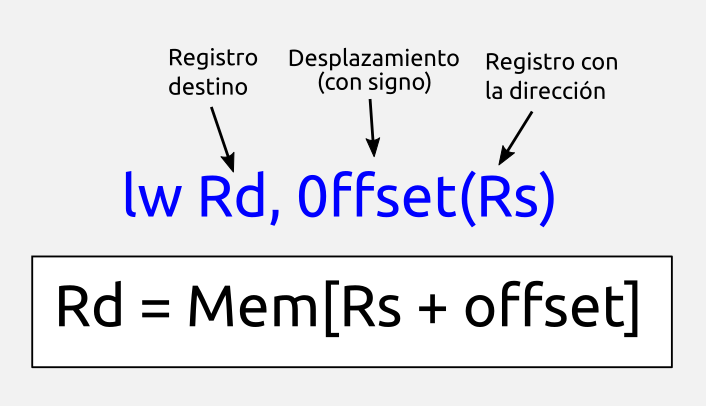
### Cargando la dirección en un registro: la

Tanto para **leer** datos de memoria (load) como para **escribirlos** (store) necesitamos tener en **un registro** la **dirección de memoria** a la que queremos acceder. Por ello, la primera acción será siempre **cargar** esta **dirección** en un registro. Esto lo hacemos con la **pseudo-instrucción la** (load address)



### La instrucción lw

Con esta instrucción **leemos** una palabra de la **memoria** y se almacena en el registro destino indicado. El **formato** es el siguiente:



Se accede a la dirección de memoria indicada en el **registro fuente** (RS) a la que se le suma un **número con signo**, denominado **desplazamiento** (offset)

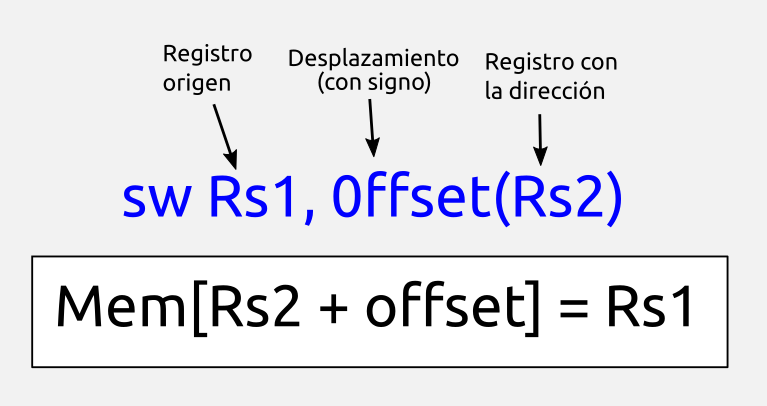
**Ejemplos** de uso:

##-- Cargar en el registro x3 la palabra situada en la dirección x2 + 0

##-- Cargar en el registro x4 la palabra situada en la dirección x2 + 4

### La instrucción sw

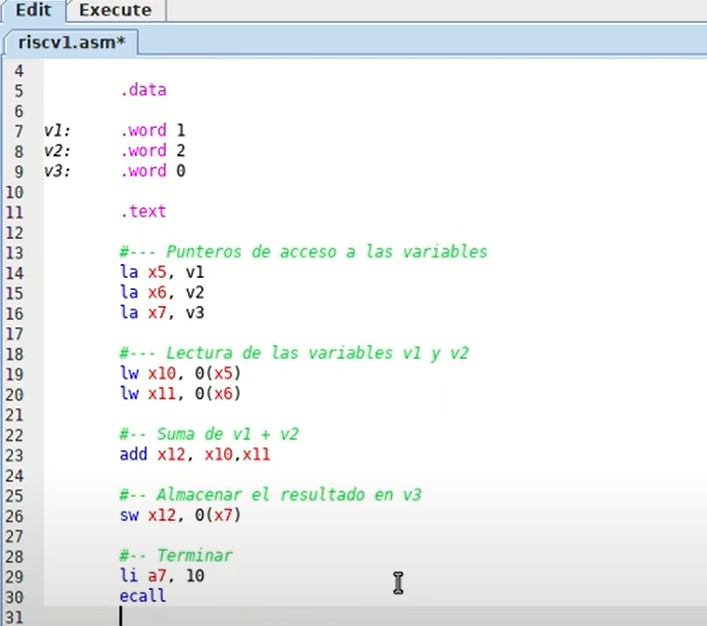
Con la instrucción sw **almacenamos** un **registro** en la dirección de **memoria** indicada. El **formato** es el siguiente:

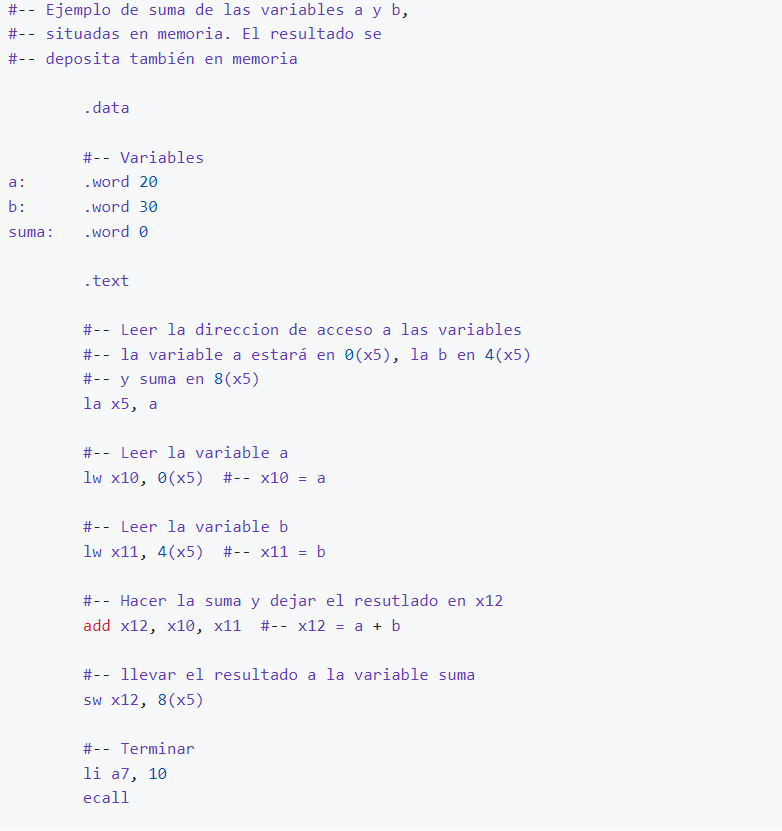


Se accede a la dirección de memoria indicada en el **registro fuente** (RS2) al que se le suma el desplazamiento (offset)

### Almacenando la suma de dos variables en otra variable

Completaremos el ejemplo de la suma anterior, pero almacenando el resultado en la **variable suma**. Así, estamos calculando la expresión: **suma = a + b**, donde a, b y suma son **variables** que están situadas en memoria

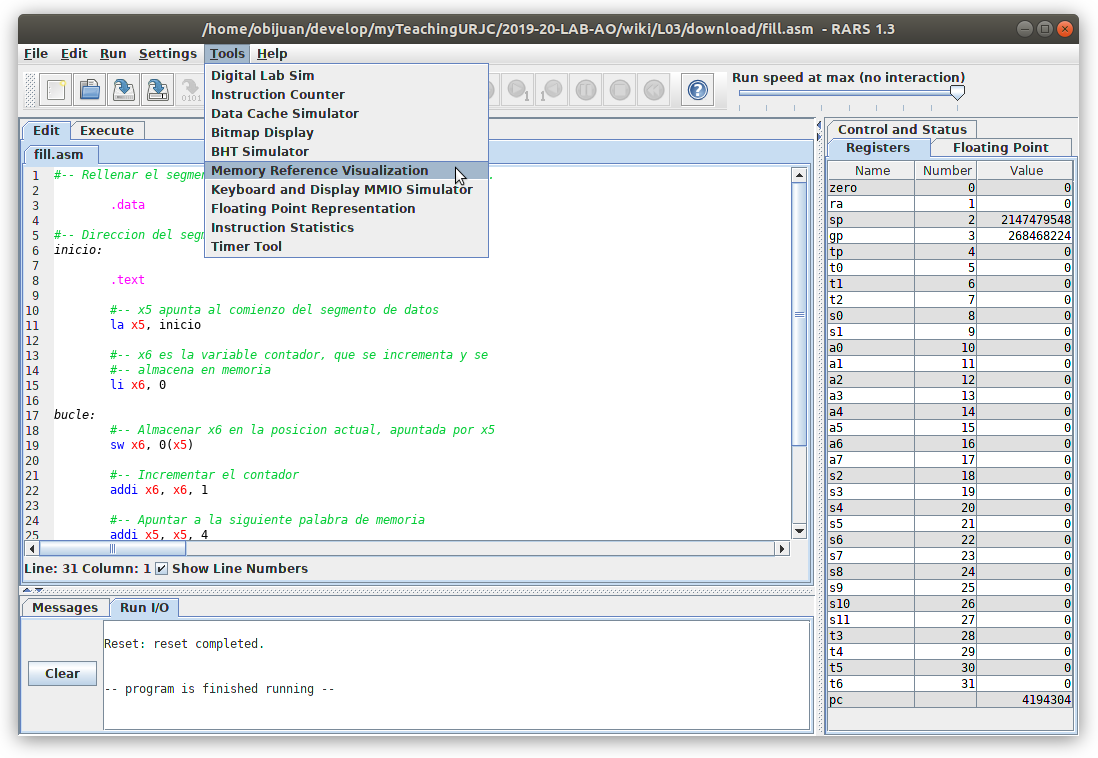




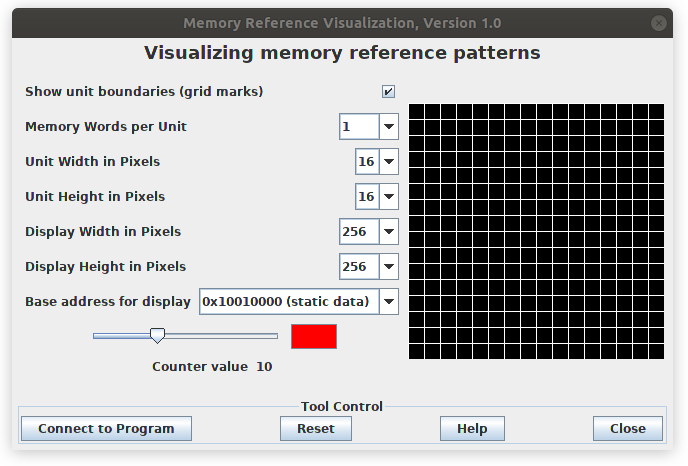
Los dos ejemplos dan el mismo resultado.

## Usando el monitor de memoria

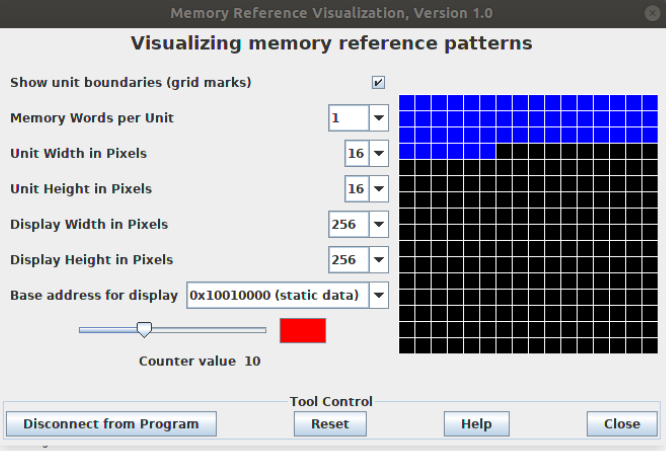
El **monitor de memoria** es una herramienta que se encuentra en el menú **Tools/Memory Reference visualization**



Nos permite ver los **accesos** que se realizan a la **memoria** (lecturas y escrituras). Es muy útil para comprobar si estamos accediendo bien a la memoria, o si hay regiones de la memoria a las que no se accede. Al arrancarlo aparece esta **pantalla**:

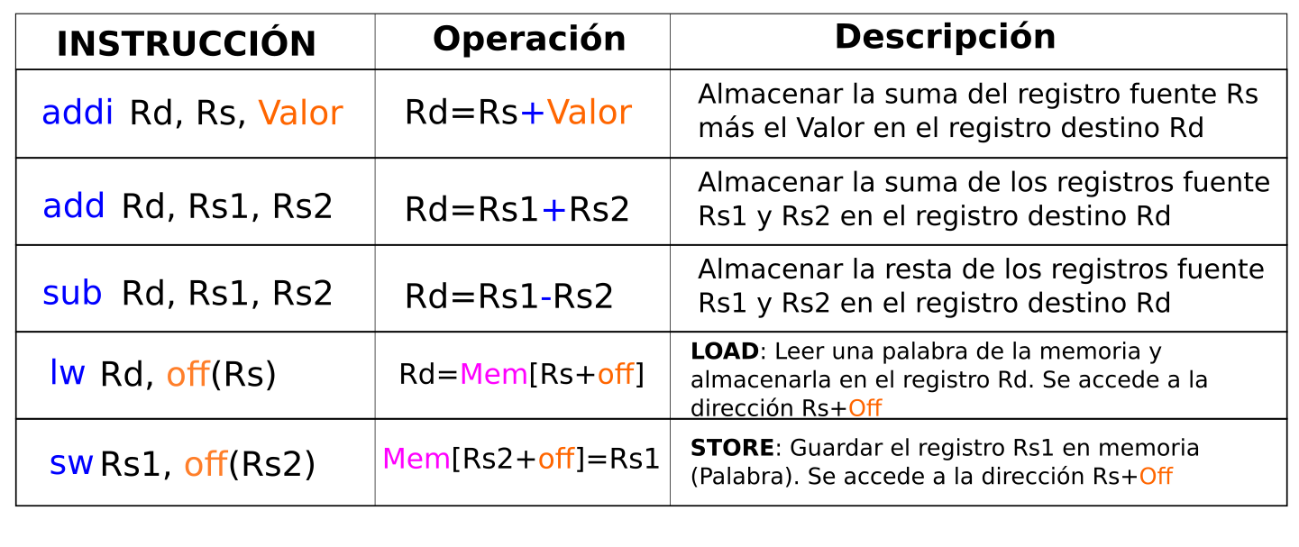


A medida que se va rellenando la memoria se irían poniendo los cuadraditos negros en azul.

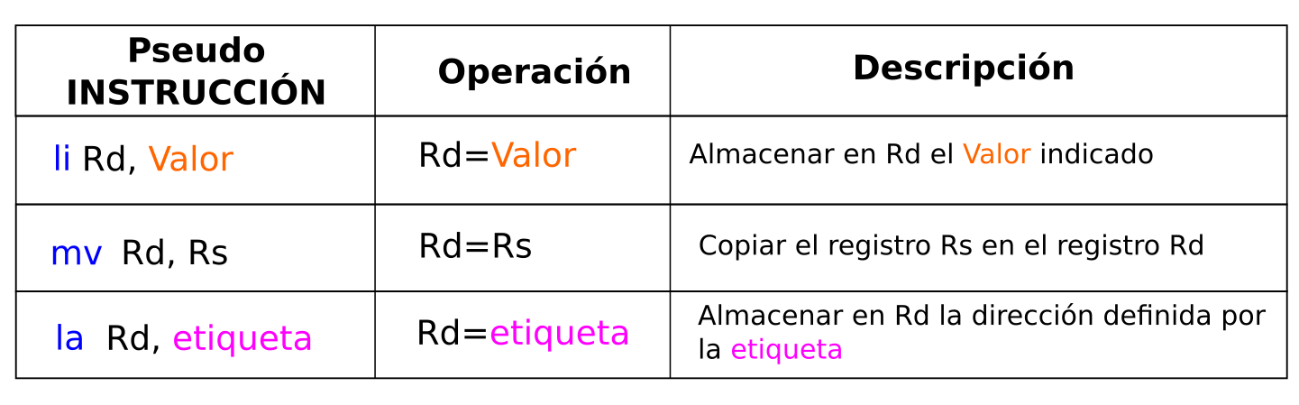


## Recopilación de instrucciones hasta el momento

* **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

