



**Grado en Ingeniería Informática**

**Estructura de Computadores**  
**Prácticas de laboratorio**

## Índice

Práctica 1: Operaciones aritméticas y de acceso a memoria.....	3
Objetivo.....	3
El simulador.....	3
Memoria.....	5
Normas de estilo .....	6
Pseudoinstrucciones .....	7
Impresión por pantalla.....	7
Ejemplo de instrucciones .....	7
Entregables.....	8
Práctica 2: Saltos .....	9
Objetivo.....	9
Entregables.....	9
Práctica 3: Procedimientos.....	11
Objetivo.....	11
Salvar el contexto.....	12
La pila del sistema .....	12
Estructura de los procedimientos .....	13
Entregables.....	13
Práctica 4: Recursividad .....	15
Objetivo.....	15
Entregables.....	15

# Práctica 1: Operaciones aritméticas y de acceso a memoria

## Objetivo

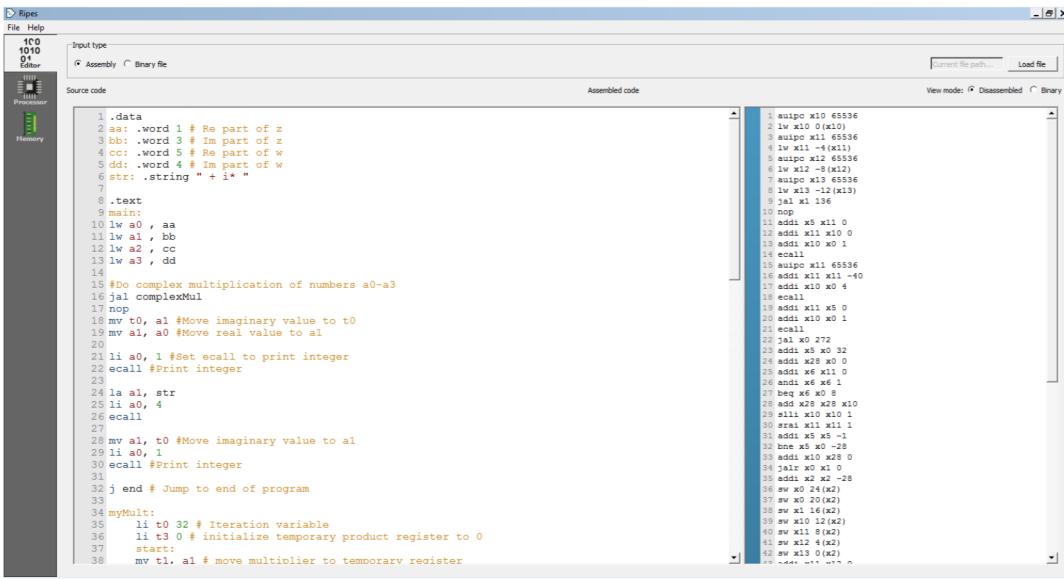
En esta práctica se aprenderán los aspectos generales de la programación en ensamblador así como el uso de las instrucciones aritméticas básicas y de acceso a memoria.

Las presentes prácticas están diseñadas para utilizar el lenguaje ensamblador de RISC-V, sobre un entorno de simulación denominado RIPES.

## El simulador

El simulador utilizado tiene tres ventanas: "Editor", "Processor" y "Memory".

La ventana "Editor" es donde se escribe el programa en ensamblador (pestaña "Source code"). También es posible cargar un fichero de texto con el programa. En caso de programar dentro del editor de RIPES, se recomienda guardar con frecuencia el código. En la pestaña "Source code" se puede ver el texto fuente con distintos códigos de colores. Por ejemplo los comentarios y etiquetas se marcan en ámbar, los registros en rojo, las instrucciones en azul, las constantes en verde, etc. En caso de un error sintáctico, el compilador marcará la línea en rojo con el objetivo de que sea corregida por el programador. Por último, en la pestaña "Assembled code" se puede ver el mismo código ensamblador del programa. Esto es útil si el código de entrada no está escrito en ensamblador.



The screenshot shows the RIPES simulation interface with three main windows: 'Processor', 'Memory', and 'Editor'. The 'Editor' window is active, displaying assembly code for a RISC-V program. The code includes sections for .data and .text, with various instructions like lw, mv, and jal. The processor window shows the execution flow, and the memory window shows the memory state.

```
1 .data
2 li t0, word 1 # Re part of z
3 bhi .word 3 # Im part of z
4 cci .word 5 # Re part of w
5 dd .word 4 # Im part of w
6 str .string " + i* "
7
8 .text
9 main:
10 lw a0 , aa
11 li a1 , bb
12 lw a2 , cc
13 li a3 , dd
14
15 #Do complex multiplication of numbers a0-a3
16 jal complexMul
17 nop
18 mv t0, a1 #Move imaginary value to t0
19 mv a1, a0 #Move real value to a1
20
21 li a0, 1 #Set ecall to print integer
22 ecall #Print integer
23
24 la a1, str
25 li a0, 4
26 ecall
27
28 mv a1, t0 #Move imaginary value to a1
29 li a0, 1
30 ecall #Print integer
31
32 j end # Jump to end of program
33
34 myMult:
35 li t0 32 # Iteration variable
36 li t3 0 # initialize temporary product register to 0
37 start:
38     mv t1, a1 # move multiplier to temporary register
```

Ilustración 1 Vista por defecto del Editor

En la ventana "Processor" es donde se ejecuta el programa. La ejecución se lanza pulsando en el botón "Run". También es posible ejecutarlo en modo paso a paso manual ("Step"), paso a paso automático usando el botón "Start autosteping" (la velocidad

puede ser modificada con el control deslizante “Autostep speed”) o resetearlo (“Reset”). En el centro de la ventana se puede ver el cauce de ejecución de un procesador de cinco etapas RISC-V, permitiendo comprobar cómo las instrucciones van fluyendo a través de cada una de dichas etapas (para esto es recomendable ejecutar el programa en modo paso a paso). En la parte derecha se ve el contenido de los registros y así como las instrucciones que se ejecutan. En la parte de abajo (“Application Output”) se puede ver los mensajes y resultados que muestra el programa.

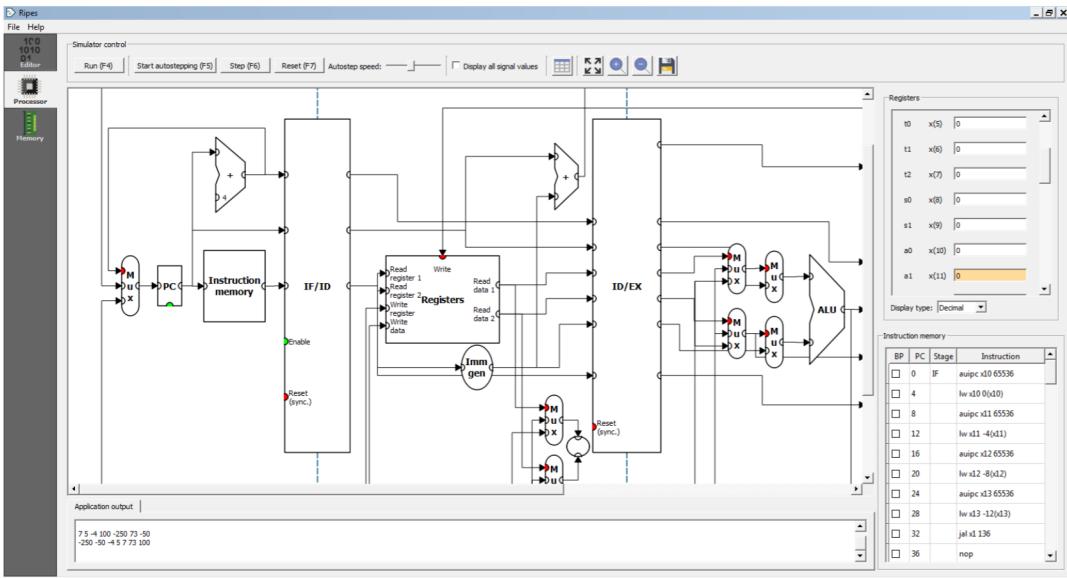


Ilustración 2 Vista por defecto del entorno de ejecución

Finalmente, en la ventana “Memory” se puede ver el contenido de los registros (parte izquierda) y de la memoria principal (parte derecha). RIPES también permite elegir el formato de representación de los datos mediante la función “Display type”.

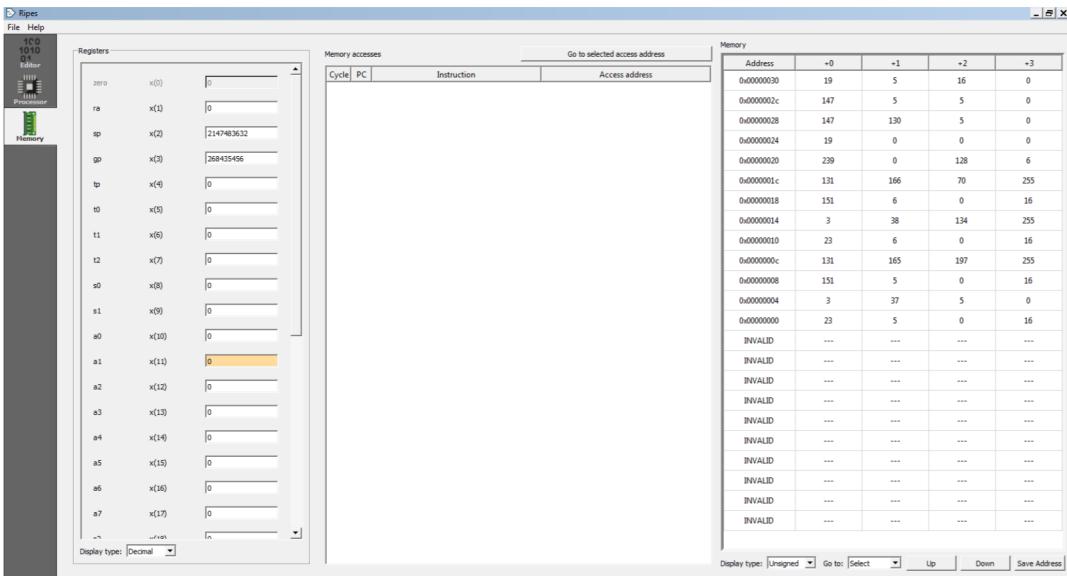


Ilustración 3 Vista por defecto de la memoria (Memory)

## Memoria

Una característica importante es que RIPES trabaja con un ancho de palabra de 32 bits, por tanto, en una palabra de memoria caben 4 Bytes. Cada uno de los bytes puede ser direccionado de manera independiente. Como la mayoría de las arquitecturas RISC, RISC-V es una arquitectura “little endian”, lo cual significa que la dirección de cada palabra corresponde a la dirección de su bit menos significativo.

El hecho de que el ancho sea de 32 bits hace que las operaciones de lectura (“load”) y escritura (“store”) en memoria se refieran con respecto a la palabra, usando las instrucciones “**lw**” (“load word”) y “**sw**” (“store word”). De esta manera, no se usarán las lecturas y cargas de 64 bits (“double word”), es decir “**ld**” y “**sd**”. Cuando se trabaje con caracteres ASCII, hay que tener en cuenta que cada uno de estos caracteres ocupa un byte. Por lo tanto, para leer y escribir caracteres de memoria se usarán las instrucciones “**lb**” (load byte) y “**sb**” (store byte).

Los datos se almacenan en memoria en tres zonas, atendiendo al tipo de cada uno:

- Zona de instrucciones (“Text”). Es el área donde el sistema almacena las instrucciones del programa a ejecutar. Empieza en la dirección 0x00000000 y va creciendo hacia direcciones más altas.
- Zona de datos estáticos (“Static data”). Es el área donde se almacenan los datos estáticos del programa (el equivalente a las constantes y variables). Empieza en la dirección 0x10000000 y va creciendo hacia direcciones más altas
- Zona de pila (“Stack”). Es el área donde se guardan datos dinámicos, indexados mediante el puntero de pila (**sp**). Empieza en la dirección 0x7fffffff0 y va decreciendo hacia direcciones más bajas.

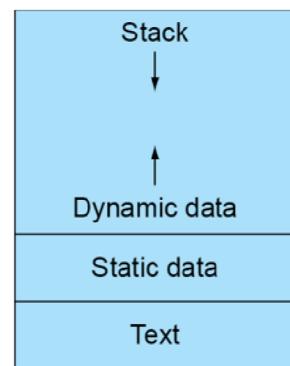


Ilustración 4 Estructura de memoria

Los programas empiezan con la declaración de la zona de datos estáticos, mediante el uso de la expresión “.data”.

Las etiquetas son símbolos que se asocian a la dirección de una determinada palabra de memoria, y se representan por cadenas alfanuméricas acabadas en “：“, por ejemplo “data：“.

Para reservar una palabra de memoria y guardar un valor entero, se usa “.word”, seguido por el valor.

Para reservar una cadena de caracteres, se usa “.string”, seguido por su valor entre comillas.

```
1 .data
2 data:      .word 5
3 sentence:  .string "This is an example"
4           .word 0
5
```

Ilustración 5 Ejemplo de reserva de datos

En este ejemplo, se ha reservado una palabra de memoria y se ha almacenado en ella el número 5. Esta palabra se referenciará con el nombre “data” (en otras palabras, “data” contendrá un puntero con la dirección de esa palabra).

De la misma manera, se ha reservado espacio en la memoria para almacenar la cadena “This is an example” (donde cada carácter ASCII ocupa un byte). El identificador “sentence” será un puntero con la dirección al primer carácter de la cadena. Después de cada cadena, es conveniente utilizar “.word 0”, ya que esto indica que la cadena anterior acaba en ese punto (el código ASCII 0, llamado null, se usa como terminador de cadena).

A continuación de la sección de datos, se encuentra la sección de instrucciones, identificada con “.text”. A partir de este punto, se escriben todas las instrucciones del programa. En esta sección se pueden escribir tanto instrucciones como etiquetas.

```
1 .data
2 data:      .word 5
3 sentence:  .string "This is an example"
4          .word 0
5
6 .text
7
8 begin:
9 lw a0,data    # load an integer in a0
10 la a1,sentence # load an address in a1
11
12 end:
13 nop
14
```

Ilustración 6 Ejemplo de zona de datos e instrucciones

En este programa de ejemplo, se han usado dos instrucciones.

La primera instrucción, “**lw a0,data**” (“load word”), carga el contenido de la palabra referenciada por “data” en el registro a0. Es decir, carga el valor 5.

En cambio, la segunda instrucción, “**la a1,sentence**” (“load address”), carga la dirección de la cadena “sentence” en el registro a1. En este caso, esta la dirección es 0x10000004, como se puede ver en el contenido del registro a1, en la sección “Memory” del simulador.

Las etiquetas “**begin:**” y “**end:**” no son obligatorias, pero son recomendables por cuestiones de claridad.

## Normas de estilo

Los programas en ensamblador suelen tener poca legibilidad. Por ello, es preciso mejorar su claridad con un buen estilo de escritura y organización del código. El uso de la sangría cuando sea preciso, dejar líneas en blanco, usar espacios, y en general utilizar cualquier recurso que permita que el código sea fácil de seguir e interpretar, ayuda a mejorarla claridad y documentación del programa.

Especial atención merecen los comentarios. Es preciso añadirlos a cada parte del código para indicar qué es lo que se está realizando en cada momento, así como identificar las secciones y aclarar las instrucciones que sean pertinentes. Los comentarios en el entorno se añaden con el símbolo "#". Todos los caracteres desde ese símbolo hasta el final de línea son interpretados como un comentario, y por tanto ignorados por el compilador. El entorno no reconoce caracteres especiales como la letra "ñ" o los acentos. Por este motivo, y para aumentar la portabilidad del código, las etiquetas y comentarios de los programas han de ser escritos en inglés.

Otro tema importante es usar los registros con sus nombres significativos. Aunque el sistema reconoce los registros de forma secuencial, de x0 a x31, es preferible utilizar sus alias. Por ejemplo, usar **sp** (stack pointer) en lugar de x2, a0 en lugar de x10 o s5 en lugar de x21. En general, los registros "a" se usan para pasar argumentos a funciones, los registros "t" son para valores temporales y los registros "s" son registros guardados en las llamadas a procedimientos.

## Pseudoinstrucciones

Aparte de las instrucciones estándar del repertorio del RISC-V, el compilador entiende otras, llamadas "*pseudoinstrucciones*". Estas instrucciones se usan para simplificar la escritura del código, bien sea agrupando un conjunto de instrucciones o "redefiniendo" el nombre de una de ellas. Como es obvio, es el compilador quien se encarga de traducirlas a instrucciones reales del ISA.

Por ejemplo, en RISC-V no existe la instrucción de movimiento entre registros, pero sí la pseudoinstrucción "**mv**". De esta manera, "**mv a1,t1**" copia el contenido del registro t1 en el registro a1. Esto se traduce como "**add a1,t1,zero**", que suma t1 con cero y guarda el resultado en a1. Como se ve, el resultado es el mismo, pero la primera forma es más intuitiva que la segunda.

Aparte de "**mv**", otras pseudoinstrucciones son por ejemplo "**la**", "**li**", "**j**".

## Impresión por pantalla

Se usa "**ecall**" para imprimir por pantalla.

Para imprimir un valor entero hay que seguir tres pasos: cargar en a1 el valor a imprimir, en a0 el valor 1 y llamar a la función **ecall**.

Para imprimir una cadena de caracteres hay que seguir tres pasos: cargar en a1 la dirección (puntero) de la cadena a imprimir, en a0 el valor 4 y llamar a la función **ecall**.

## Ejemplo de instrucciones

Algunos **ejemplos** de instrucciones sencillas:

Instrucción	Resultado
<b>li s4,5</b>	Carga en s4 el valor 5
<b>la a2,var</b>	Carga en a2 la dirección de la variable "var" reservada en memoria
<b>lw a0,var</b>	Carga en a0 el contenido de la variable "var" reservada en memoria.
<b>lw a0,4(a1)</b>	Carga en a0 la palabra de memoria cuya dirección es a1+4
<b>lb a0,4(a1)</b>	Carga en a0 el byte de memoria cuya dirección es a1+4

<b>sw a3,8(a2)</b>	Almacena el valor de a3 en la palabra de memoria cuya dirección es a2+8
<b>sb a3,8(a2)</b>	Almacena el valor de a3 en el byte de memoria cuya dirección es a2+8
<b>add a0,a1,a2</b>	Suma a1 más a2, y guarda el resultado en a0
<b>addi a0,a1,6</b>	Suma a1 más 6, y guarda el resultado en a0
<b>sub a0,a1,a2</b>	Resta a1 menos a2, y guarda el resultado en a0
<b>addi a0,a1,-1</b>	Resta a1 menos 1, y guarda el resultado en a0
<b>mul a0,a1,a2</b>	Multiplica a1 por a2, y guarda el resultado en a0
<b>div a0,a1,a2</b>	Divide a1 entre a2, y guarda el cociente de la división en a0
<b>rem a0,a1,a2</b>	Divide a1 entre a2, y guarda el resto de la división en a0

## Entregables

Se entregará un programa en ensamblador para cada uno de los apartados siguientes. A la hora de puntuar la práctica se tendrán en cuenta tres aspectos:

1. La corrección del programa en base a las especificaciones descritas.
2. La calidad del código.
3. El cumplimiento de las normas de estilo.

### 1) Impresión por pantalla

- a) Mostrar por pantalla el texto “39steps”
- b) Mostrar por pantalla (en líneas distintas) el texto “39  
steps”

Para escribir en distinta línea, considerar que el código ASCII para nueva línea es el 10.

### 2) Operaciones aritméticas simples

Realizar la operación matemática  $F = (A+B) - (C+D)$ . Para ello, definir las variables A, B, C, D y F en memoria. Dar los valores iniciales A=5, B=3, C=2, D=2. Escribir el resultado de la operación en la variable F. Mostrar también el valor de F por pantalla.

### 3) Resto de la división

- a) Definir en memoria dos variables enteras positivas, A y B (valores iniciales a elegir), y calcular el resto de dividir A entre B. Almacenar el resultado en la variable R y mostrar el resultado por pantalla. Para realizar este apartado, utilizar la instrucción “rem”.
- b) Repetir el punto anterior, pero esta vez sin utilizar la instrucción “rem”. Para ello, utilizar la propiedad de la división de que el dividendo es el divisor multiplicado por el cociente más el resto.

## Práctica 2: Saltos

### Objetivo

El objetivo de esta práctica es comprender los saltos en lenguaje ensamblador y utilizarlos para modificar el flujo de control de un programa.

En programación, tanto las sentencias condicionales (*if-then-else*) como los bucles (*for, while*) se implementan mediante saltos. Un salto es una modificación del flujo de control de un programa, mediante el cual la próxima instrucción que se ejecuta no es la siguiente en orden secuencial, sino que el contador del programa (**PC**) realiza un salto a otra posición.

Los saltos más habituales son los condicionales, que se producen cuando se cumple una condición que relaciona dos registros.

Instrucción	Condición de salto
<b>beq a1,a2,label</b>	$a1=a2$
<b>bne a1,a2,label</b>	$a1 \neq a2$
<b>bge a1,a2,label</b>	$a1 \geq a2$
<b>blt a1,a2,label</b>	$a1 < a2$

En todos los casos, si se cumple la condición, el programa saltará a la instrucción marcada con la etiqueta "**label**".

Los saltos incondicionales son aquellos que se producen siempre, sin tener en cuenta ninguna condición, como por ejemplo:

**beq a0,a0,label**

En este caso, da igual el valor de registro que se use, ya que la condición de igualdad se cumplirá siempre.

También se puede usar la pseudoinstrucción de salto:

**j label**

### Entregables

Se entregará un programa en ensamblador para cada uno de los apartados siguientes. A la hora de puntuar la práctica se tendrán en cuenta tres aspectos:

1. La corrección del programa en base a las especificaciones descritas.
2. La calidad del código.
3. El cumplimiento de las normas de estilo.

#### 1) Conversión de minúsculas en mayúsculas – versión simple

Definir una variable de cadena en memoria e inicializarla con cualquier palabra de 6 caracteres, utilizando únicamente letras minúsculas. Realizar un programa que cambie todas las letras de la palabra a mayúsculas. Para este caso, no utilizar ningún salto. Como resultado, mostrar en pantalla la palabra inicial en minúsculas, y en la siguiente línea la misma palabra convertida a mayúsculas. Para realizar el cambio, aprovechar la propiedad de que los códigos ASCII de una letra en mayúsculas y la

misma letra en minúsculas están siempre a una distancia fija. Recordar que cada letra almacenada en memoria ocupa un byte y que por lo tanto hay que utilizar “**lb**” y “**sb**”.

**2) Conversión de minúsculas en mayúsculas – versión avanzada**

Definir una variable de cadena en memoria, de longitud indeterminada, e inicializarla con una frase. La frase puede contener, únicamente, cualquier combinación de letras minúsculas y espacios. La frase debe acabar en punto. Repetir el apartado anterior, pero esta vez utilizando un bucle. La condición de parada del bucle se producirá al encontrar el punto final de la frase.

## Práctica 3: Procedimientos

### Objetivo

Aprender a utilizar procedimientos en ensamblador, como una manera de estructurar, organizar y reusar código.

Los procedimientos son secciones de código que realizan una función determinada. Los procedimientos suelen recibir parámetros de entrada desde el programa principal y devuelven un resultado. Mediante el uso de procedimientos se simplifica la programación, ya que una vez programado un procedimiento, este se puede utilizar en distintos programas.

Para llamar a un procedimiento, es preciso que el sistema guarde la dirección de retorno, para que el flujo de control vuelva al sitio adecuado tras la ejecución del mismo. La dirección de retorno se corresponde con la instrucción siguiente a la que realizó la llamada al procedimiento.

Para llamar a un procedimiento, hay que utilizar la instrucción “*jump and link*” (saltar y enlazar), descrita como “**jal**”:

```
8 #####
9 jal ra,dummy    #procedure call
10 addi a0,a0,-1   #next instruction after call
11 #####
12
13 #-----
14 dummy: #this is an example of procedure
15 begin_dummy:
16 mul a0,a2,a4  #the procedure just does a multiplication
17 end_dummy:
18 jalr zero,ra,0 #return to main program
19 #-----
```

Ilustración 7 Ejemplo de procedimiento

En este ejemplo, la instrucción “**jal ra,dummy**” salta al procedimiento marcado con la etiqueta “**dummy**”, y guarda en el registro **ra** la dirección de retorno (“return address”). De esta manera, cuando el procedimiento acabe, el flujo de control volverá a la siguiente instrucción, en este caso “**addi a0,a0,-1**”.

Para volver del procedimiento, una vez que este acabe, al programa principal, hay que usar la instrucción “**jalr zero,ra,0**”.

Esta instrucción retornará a la dirección contenida en **ra**, que previamente se guardó con la instrucción **jal** de llamada.

## Salvar el contexto

El contexto de un microprocesador es el conjunto de registros que está utilizando, junto con sus valores. Cuando se llama a un procedimiento, es preciso guardar al principio todos los registros que se vayan a modificar en la pila del sistema. A esta operación se le denomina guardar el contexto.

Análogamente, es preciso recuperar todos los valores guardados en la pila justo antes de salir del procedimiento. A esta operación se le denomina restaurar el contexto.

En RISC-V, el convenio es que cada procedimiento ha de guardar todos los registros “**s**” que modifique. Para simplificar, se sugiere que los procedimientos guarden y restauren todos los registros que utilicen (independientemente de su tipo), excepto los registros con parámetros de salida (que son los que devuelven los resultados al programa principal).

**Importante:** si un procedimiento llama a su vez a otro procedimiento, o a sí mismo (recursividad), es obligatorio guardar y restaurar el registro ra. Si no se hace, se perdería la dirección de retorno de la primera llamada, y se rompería el flujo de ejecución. Si un procedimiento no llama a ningún otro, entonces no es necesario guardar el registro **ra**.

## La pila del sistema

La pila del sistema es una zona de memoria que sirve para guardar datos de manera dinámica. Sigue una estructura *FIFO* (*First-In-First-Out*), y se gestiona mediante el denominado puntero de pila (**sp**). El puntero de pila apunta al último dato (final de la pila) que se ha almacenado en la estructura. La pila crece hacia abajo, es decir, los datos se guardan en direcciones decrecientes. Por la estructura FIFO de la pila, es necesario que los registros se guarden y se restauren en orden inverso. En la figura se ve un ejemplo donde se guardan y restauran los registros **ra**, **s0**, **s1**, **a0** y **a1**. Recordar que cada palabra en memoria ocupa 4 bytes, y por eso las referencias son todas múltiplos de 4.

```
2 #save context
3 addi sp,sp,-20
4 sw ra,16(sp)
5 sw s0,12(sp)
6 sw s1,8(sp)
7 sw a0,4(sp)
8 sw a1,0(sp)
9 -----
10 |
11 -----
12 #restore context
13 lw a1,0(sp)
14 lw a0,4(sp)
15 lw s1,8(sp)
16 lw s0,12(sp)
17 lw ra,16(sp)
18 addi sp,sp,20
19 -----
20
```

Ilustración 8 Ejemplo de grabación y restauración del contexto

## Estructura de los procedimientos

Para mejorar la portabilidad de los procedimientos, es necesario seguir las siguientes normas de estilo:

- Los procedimientos vendrán separados, al comienzo y fin, por comentarios separadores.
- Al principio, después de la etiqueta que da nombre al procedimiento, es preciso describir con comentarios todos sus parámetros de entrada y salida, así como cualquier explicación necesaria para comprender su funcionamiento.
- Añadir a las etiquetas de “**begin**” y “**end**” el nombre del procedimiento.
- Marcar claramente las zonas de grabación y restauración del contexto.
- Añadir cuantos comentarios sean necesarios.

En la figura se muestra un ejemplo de cómo estructurar un procedimiento:

```
12 #-----
13 compare: # a1,a2 (in): Values to compare
14      # a0 (out): Greatest value
15      # Function: This procedure receives two integer values (in a1 and a2)
16      #           and returns the greatest of the two in a0
17 begin_compare:
18 -----
19 #save context  #a0 is not saved because it is an output parameter
20 addi sp,sp,-12
21 sw ra,8(sp)   #saving ra is not needed in this case
22 sw a1,4(sp)
23 sw a2,0(sp)
24 -----
25 bge a1,a2,first_greatest
26 mv a0,a2      # a2>a1
27 j end_compare
28 first_greatest:
29 mv a0,a1      # a1>a2
30 -----
31 end_compare:
32 #restore context
33 lw a2,0(sp)
34 lw a1,4(sp)
35 lw ra,8(sp)
36 addi sp,sp,12
37 -----
38 jalr zero,ra,0 #return
39 #-----
```

Ilustración 9 Ejemplo de procedimiento

## Entregables

Se entregará un programa en ensamblador para cada uno de los apartados siguientes. A la hora de puntuar la práctica se tendrán en cuenta tres aspectos:

1. La corrección del programa en base a las especificaciones descritas.
2. La calidad del código.
3. El cumplimiento de las normas de estilo.

### 1) Años bisiestos – versión simple

Diseñar un programa que determine si un año es bisiesto o no. Para este primer apartado, no utilizar ninguna llamada a procedimiento. Definir una variable en memoria denominada “year”, que almacene el año a analizar. Mostrar por pantalla si el año elegido es bisiesto o no, por ejemplo:

Ejemplo 1: “2018 is not a leap year”

Ejemplo 2: "2020 is a leap year"

**2) Creación de procedimientos**

Diseñar los siguientes procedimientos. Diseñar también un programa principal que llame a estos procedimientos y compruebe su funcionamiento.

- a) Procedimiento de imprimir enteros: El procedimiento recibe un valor numérico en a1 y lo muestra por pantalla.
- b) Procedimiento de imprimir cadenas de caracteres. El procedimiento recibe un puntero al comienzo de una cadena, en a1, y lo muestra por pantalla.
- c) Procedimiento de nueva línea. El procedimiento genera una nueva línea en pantalla.

**3) Años bisiestos – versión avanzada**

Repetir el primer apartado, pero ahora estructurando el código que calcula si un año es bisiesto o no en un procedimiento. El procedimiento recibirá un año en el registro a1. Si el año es bisiesto, escribirá un 1 en a0. Si no es bisiesto escribirá un 0 en a0. Guardar y restaurar el contexto en el procedimiento y seguir las normas de estilo descritas anteriormente.

Diseñar un programa principal que demuestre el procedimiento creado, y que además aproveche los procedimientos diseñados en el apartado 2). La salida por pantalla deberá ser la misma que la descrita en el apartado 1).

## Práctica 4: Recursividad

### Objetivo

Realizar un programa en ensamblador que realice la ordenación de un array mediante el algoritmo de la burbuja (Bubblesort), estructurándolo mediante un proceso recursivo.

En esta práctica se utilizarán los conceptos aprendidos en prácticas anteriores para diseñar un algoritmo de ordenación.

### Entregables

Se entregará un programa en ensamblador para cada uno de los apartados siguientes. A la hora de puntuar la práctica se tendrán en cuenta tres aspectos:

1. La corrección del programa en base a las especificaciones descritas.
2. La calidad del código.
3. El cumplimiento de las normas de estilo.

#### 1) Procedimiento de impresión de un array.

Diseñar un procedimiento que imprima un array de números enteros por pantalla. El procedimiento recibirá en a1 un puntero al comienzo del array, y en a0 la longitud de dicho array. Se mostrará en pantalla los elementos que forman dicho array, separados por un espacio.

Ejemplo: 7 5 -4 100 -250 73 -50

Este procedimiento debe llamar a su vez a los procedimientos de imprimir entero y nueva línea de la práctica anterior. Recordar que hay que guardar el registro ra como parte del contexto.

#### 2) Algoritmo de la burbuja mediante procedimiento recursivo.

Diseñar un programa que ordene un array de enteros mediante el procedimiento del algoritmo de la burbuja ("bubblesort"). El programa definirá en memoria una lista de enteros y una variable con la longitud de dicha lista. Por ejemplo:

```
1 .data
2
3 length: .word 7
4
5 list:   .word 7
6     .word 5
7     .word -4
8     .word 100
9     .word -250
10    .word 73
11    .word -50
```

Ilustración 10 Ejemplo de lista inicial

El programa mostrará en pantalla el array inicial, y en una nueva línea el array ordenado de menor a mayor:

Ejemplo: 7 5 -4 100 -250 73 -50

-250 -50 -4 5 7 73 100

Es obligatorio implementar el algoritmo de ordenación usando un procedimiento recursivo. Igualmente, utilizar también el procedimiento de impresión creado en el apartado anterior, así como cualquier otro procedimiento que sea preciso para una mejor organización del código.