



**PROGRAMA UNIVERSITARIO DE IMAGINATION**

# **Práctica 4 RVfpga**

## **Llamadas a función**

## 1. INTRODUCCIÓN

Las llamadas a funciones son una parte crítica de cualquier programa porque permiten modularidad y la reutilización de código y, por lo tanto, facilitan la escritura y la depuración del código. El lenguaje de programación C también incluye librerías estándar, así como librerías específicas de procesador/placa, de funciones C de uso común, como generadores de números aleatorios y funciones matemáticas típicas. Las funciones de alto nivel se traducen en ensamblador siguiendo un *convenio de llamadas*. Esta práctica describe cómo escribir y utilizar funciones en programas en C, tanto funciones escritas por el programador como funciones incluidas en las librerías C. También ilustra cómo se implementan las funciones en lenguaje ensamblador. Al final de la práctica se proporcionan ejercicios de escritura de programas que usan llamadas a funciones y librerías.

## 2. Escribiendo un programa C que utiliza funciones

Una función (también llamada subrutina o procedimiento) es código que está empaquetado en un bloque de código que tiene un funcionamiento y una interfaz (entradas y salidas) definidas. Esta modularidad aumenta la eficiencia al disminuir la complejidad y soportar reutilización del código. Se puede llamar a una función desde cualquier punto del programa de tal manera que, cuando la función finaliza, la ejecución del programa se reanuda justo a continuación de la llamada a la función. Las funciones pueden ser llamadas desde otra función (funciones *anidadas*), o incluso por la misma función (llamadas *recursivas*).

Para escribir un programa RISC-V con funciones, se siguen los mismos pasos generales descritos en las Prácticas 2 y 3:

1. Crear un proyecto RVfpga
2. Escribir un programa en C
3. Descargar RVfpgaNexys en la placa FPGA Nexys A7 (recuerde que también puede ejecutar los programas en simulación, tanto en Verilator como en Whisper)
4. Compilar, descargar y ejecutar/depurar el programa

Consulte la Práctica 2 para obtener las instrucciones detalladas sobre estos pasos. A continuación, se proporciona una breve descripción de cada paso.

### Paso 1. Crear un proyecto RVfpga

Cree un proyecto llamado project1 en la siguiente carpeta:

```
[RVfpgaPath]/RVfpga/Labs/Lab4
```

### Paso 2. Escribir un programa C

Ahora añadirá un programa C al proyecto. Cree un nuevo archivo y escriba o copie/pegue el siguiente código C, que también está disponible en el siguiente archivo:

```
[RVfpgaPath]/RVfpga/Labs/Lab4/LedsSwitches_functions.c
```

```
// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) =
(value); }
```

```
void IOsetup();
unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main ( void )
{
    unsigned int switches_val;

    IOsetup();
    while (1) {
        switches_val = getSwitchVal();
        writeValtoLEDs(switches_val);
    }

    return(0);
}

void IOsetup()
{
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal()
{
    unsigned int val;

    val = READ_GPIO(GPIO_SWs);    // read value on switches
    val = val >> 16;    // shift into lower 16 bits

    return val;
}

void writeValtoLEDs(unsigned int val)
{
    WRITE_GPIO(GPIO_LEDs, val);    // display val on LEDs
}
```





Guarde el archivo en el directorio `src` de su proyecto con el nombre `LedsSwitches_Functions.c`.

### Paso 3. Descargar RVfpgaNexys en la placa FPGA Nexys A7

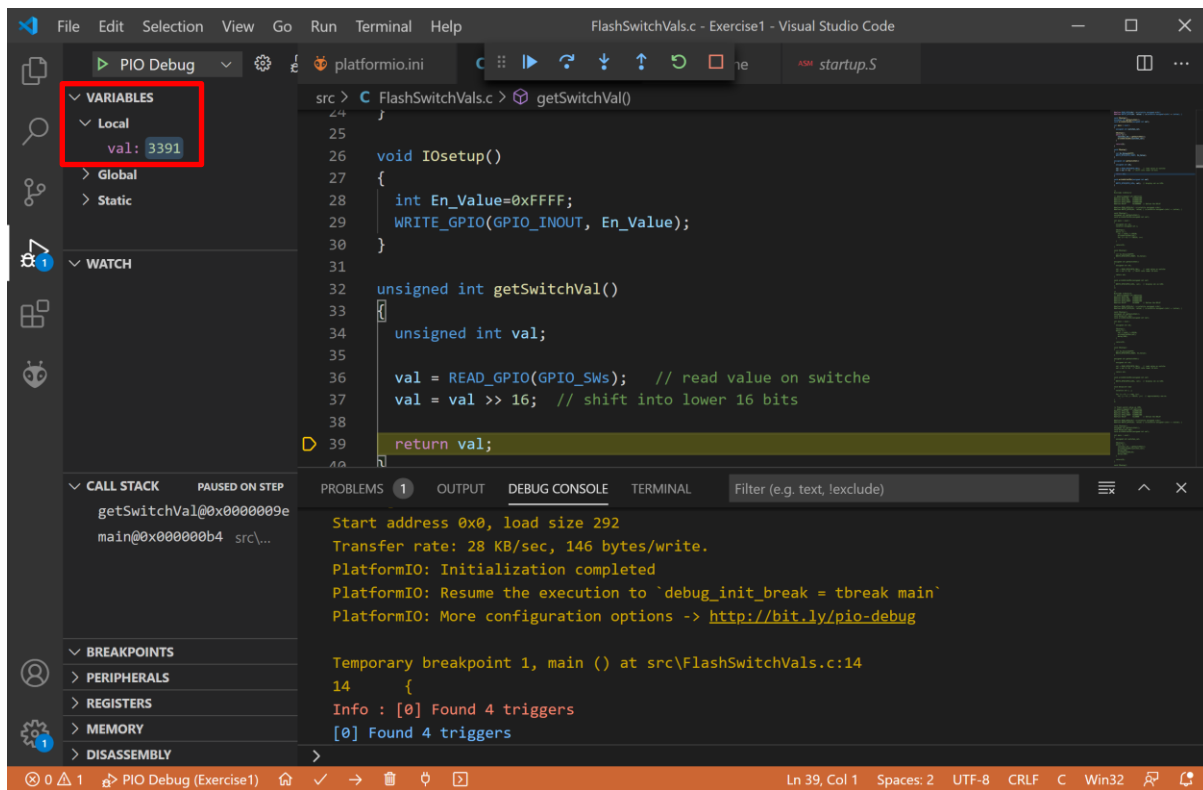
Descargue RVfpgaNexys en la placa Nexys A7 del mismo modo que en las Prácticas 2 y 3.

### Paso 4. Compilar, descargar y ejecutar el programa

Ahora el usuario y a puede compilar, descargar y ejecutar/depurar el programa en RVfpgaNexys.

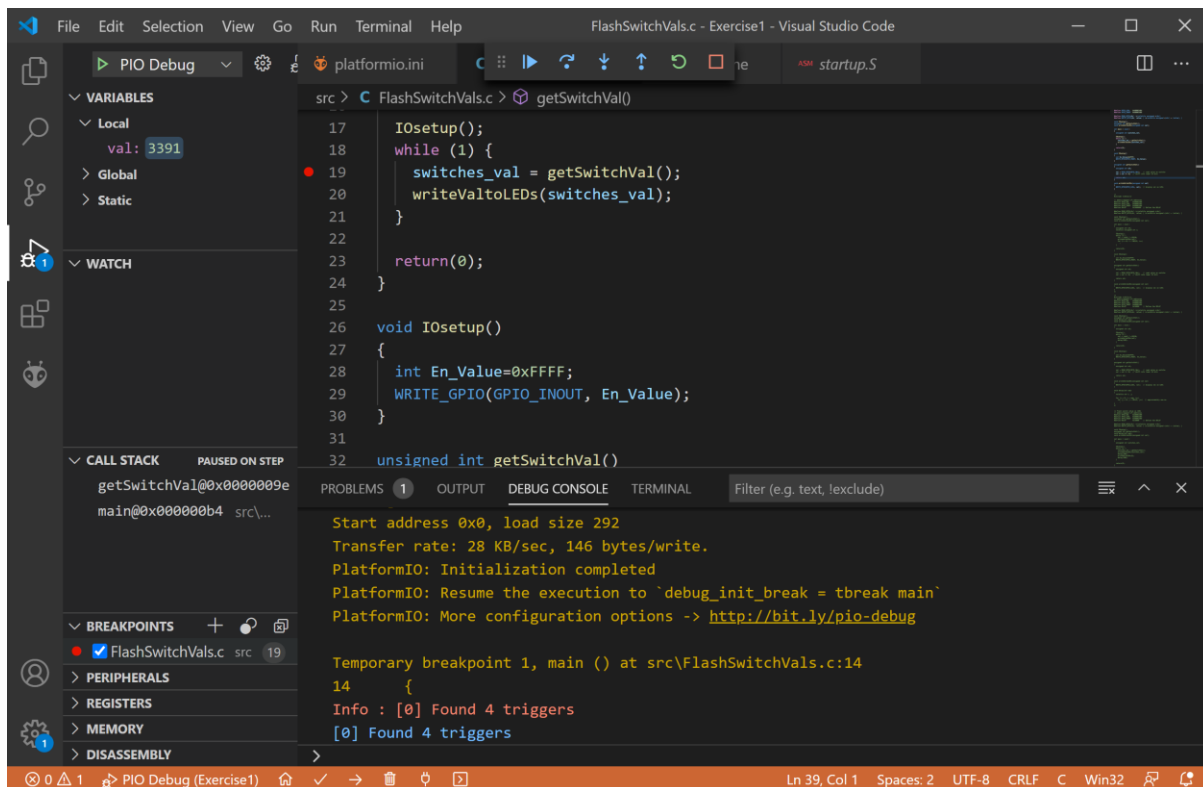
Después de pulsar los botones Run  y Start Debugging  PIO Debug, haga clic en el botón Step Over  (situado en la barra de herramientas superior) o presione F10 dos veces hasta que llegue a la línea 19 en la que se llama a la función `getSwitchVal()`. Luego presiona el botón Step Into  (o presione F11). Así accederá al código de la función `getSwitchVal()`. Si aún no es visible, expanda el campo VARIABLES → Local en la barra de herramientas de la izquierda para visualizar la variable `val`. Esta variable puede aparecer como "optimizada" en este punto del programa. Haga clic en Step (ya sea Step Over o Step

Into) una vez y vea el cambio de la variable val al valor de los interruptores, como se muestra en la Figura 1.





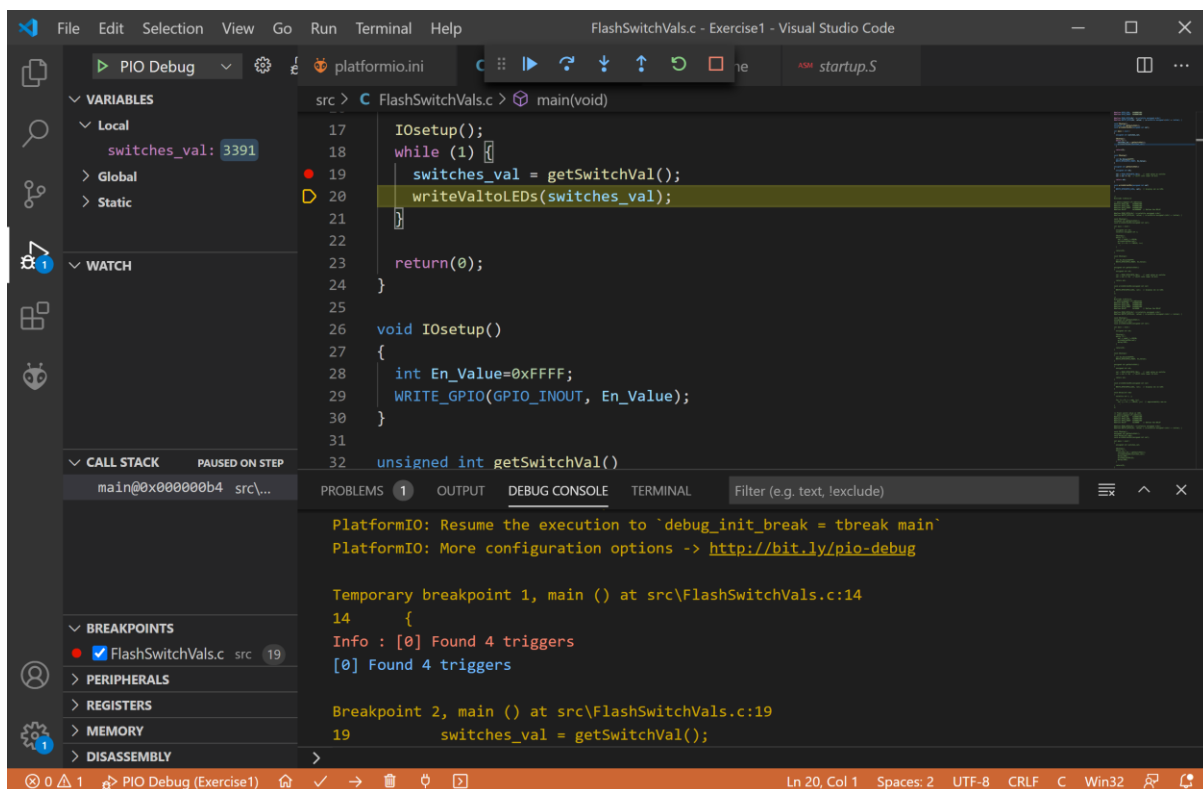
**Figura 1. Ejecución de la función getSwitchVal()**

Establezca un punto de interrupción en la línea 19 haciendo clic a la izquierda de la misma. Aparecerá un punto rojo a la izquierda de la línea, como se muestra en la Figura 2.



**Figura 2. Creación de un punto de interrupción**

Presione el botón Continúe  (o F5). El programa se detendrá en la línea 19 una vez que se alcance el punto de interrupción. Esta vez, presione el botón Step Over  (o presione F10). La función se ejecutará, pero el depurador no entrará en la función. Sólo se muestran los efectos de la función. En particular, la variable switches\_val toma el valor de los interruptores, como se muestra en la Figura 3.



**Figura 3. Ejecución de la función getSwitchVal() sin detenerse la depuración**

### 3. Escritura de un programa en C con llamadas a funciones de librería

Los lenguajes de programación de alto nivel como C incluyen librerías de funciones que los programadores usan comúnmente. Si busca en Google "Librerías estándar de C" puede encontrar una lista de las librerías de C más utilizadas, que se pueden utilizar incluyendo el archivo de cabecera que proporciona la declaración de las funciones incluidas en la librería. Esto se hace añadiendo la siguiente línea al principio del archivo del programa C:

```
#include <nombre_de_la_librería>
```

en donde "nombre\_de\_la\_librería" se sustituye por el nombre de la librería en cuestión. Por ejemplo, la librería matemática (math.h) proporciona funciones comunes como fabs(), que calcula el valor absoluto de un número en coma flotante, fmax(), que devuelve el mayor de dos números en coma flotante, etc.

Otra librería común es la librería estándar de C (stdlib.h). Algunas de las funciones incluidas en esta librería generan números aleatorios. Por ejemplo, el programa que se muestra a continuación muestra un número aleatorio en los LEDs al incluir el archivo de cabecera stdlib.h (#include <stdlib.h>) y llamar a la función rand() que devuelve un número aleatorio. Copie y pegue el siguiente programa en un proyecto RVfpga en PlatformIO y ejecútelo sobre RVfpgaNexys en la placa FPGA Nexys A7.

```
#include <stdlib.h>

// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
```

```
#define GPIO_LEDS    0x80001404
#define GPIO_INOUT   0x80001408
#define DELAY        0x1000000 // Define the DELAY

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) =
(value); }

void IOsetup();
unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main(void)
{
    unsigned int val;
    volatile unsigned int i;

    IOsetup();
    while (1) {
        val = rand() % 65536;
        writeValtoLEDs(val);
        for (i = 0; i < DELAY; i++)
            ;
    }
    return(0);
}

void IOsetup() {
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal() {
    unsigned int val;

    val = READ_GPIO(GPIO_SWs); // read value on switches
    val = val >> 16; // shift into lower 16 bits

    return val;
}

void writeValtoLEDs(unsigned int val) {
    WRITE_GPIO(GPIO_LEDS, val); // display val on LEDs
}
```

Este programa también está disponible en el siguiente archivo:

[RVfpgaPath]/RVfpga/Labs/Lab4/RandomNumberLEDs.c

Además de estas librerías estándar de C, Western Digital (WD) proporciona, dentro de su paquete de firmware (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>), librerías específicas para el procesador SweRV EH1 (PSP, que puede encontrar en su sistema en `~/.platformio/packages/framework-wd-riscv-sdk/psp/`) y para la placa Nexys A7 (BSP, que puede encontrar en su sistema en

```
~/platformio/packages/framework-wd-riscv  
sdk/board/nexys_a7_eh1/bsp/). Como se explica en la Guía de inicio (Sección 6.F -  
Programa HelloWorld_C-Lang), estas librerías se incluyen en el proyecto añadiendo la línea  
adecuada en platformio.ini e incluyendo los archivos adecuados al principio del programa C.
```

Estas librerías proporcionan funciones y macros que permiten a los programadores utilizar interrupciones, imprimir una cadena de caracteres, leer/escribir registros individuales, entre otras cosas. En la Guía de inicio de RVfpga y en estas prácticas, el usuario utilizará muchas de estas funciones en los ejemplos y ejercicios.

#### 4. Convenio de Llamadas en RISC-V

Esta sección describe el Convenio de Llamadas de RISC-V, que define cómo se traducen las funciones de alto nivel a lenguaje ensamblador de RISC-V. Este convenio es parte de la **Interfaz Binaria de Aplicación** (ABI, por sus siglas en inglés). Al definir un convenio, las funciones escritas por diferentes programadores o contenidas en librerías pueden ser usadas en todos los programas. En RISC-V, la instrucción de **salto y enlace** (`jal`) invoca una llamada a una función. Por ejemplo, el siguiente código llama a la función `func1`:

```
jal func1
```

Esta instrucción salta a la etiqueta `func1` y guarda la dirección de la instrucción posterior a `jal` en el registro de direcciones de retorno (`ra = x1`). Posteriormente la función regresa usando la pseudo-instrucción de retorno (`ret`) (o instrucción de salto a registro: `jr ra`), que salta a la dirección almacenada en `ra`.

Las funciones pueden ser invocadas con argumentos de entrada y también pueden devolver un valor a la función invocante. Por el convenio de RISC-V, los argumentos de entrada se pasan a la función en los registros `a0-a7`. Si se necesitan argumentos adicionales, se colocan en la pila. De nuevo por el convenio, los valores de retorno se colocan en los registros `a0` y `a1`. El acuerdo sobre qué registros **se utilizan para pasar los argumentos y los valores de retorno está definido por el convenio de llamadas de RISC-V**.

Para poder invocar con seguridad una función desde cualquier lugar del programa, es esencial que la función preserve el estado arquitectónico de la máquina (es decir, el contenido de los registros que puede ver el programador). Suponga un programa con una función `main` que tiene un bucle que utiliza el registro `t0` para almacenar el índice del bucle. En el cuerpo del bucle se invoca una función llamada `SortVector`, y esta función `SortVector` utiliza el registro `t0` para almacenar la dirección del vector A (ver Figura 4). Así, el registro `t0` se sobrescribe en la función `SortVector`, lo que tiene el indeseable efecto colateral de modificar el índice del bucle y hacer que su ejecución sea incorrecta.



```

main:
    add t2, zero, M
    add t0, zero, zero
    ...
loop1:
    bge t0, t2, endloop1
    ...
    jal SortVector
    ...
    add t0, t0, 1
    j loop1
endloop1:
    ...
    ret

SortVector:
    ...
    la t0, A
    ...
    ret

```

**Figura 4. Ejemplo del conflicto en el uso de un registro entre el programa principal y una función**

Obviamente, esto no habría ocurrido si al escribir el programa `main` se hubiera seleccionado otro registro para implementar el índice del bucle (por ejemplo, `t1`). Sin embargo, no es razonable (y en algunos casos, ni siquiera posible) obligar al programador a conocer todos los detalles internos de la implementación de una función antes de llamarla.

Una solución más práctica es que cada función cree una copia temporal en la memoria de todos aquellos registros que serán modificados, y que se restauren sus valores originales antes de volver al programa que lo llamó. Esta solución se implementa mediante la **Pila de Llamadas**, que es una región de memoria a la que se accede mediante una política LIFO (Last-In-First-Out). Esta región se utiliza para almacenar toda la información relacionada con las funciones activas del programa (es decir, aquellas funciones que han comenzado su ejecución, pero no han terminado todavía), y comienza al final de la memoria disponible (es decir, en las direcciones más altas), y crece hacia las direcciones más bajas.

Una función normalmente se estructura en tres partes:

- ➔ Código de entrada (**Prólogo**)
- ➔ Función **Cuerpo**
- ➔ Código de salida (**Epílogo**)

El *Prólogo* debe crear el **marco de pila** de la función y almacenar registros en la pila, si es necesario. El *marco de pila* es la región de memoria utilizada por una función durante su ejecución. El *Epílogo* restaura el estado arquitectónico del programa que llama a la función y libera el espacio de memoria ocupado por el *marco de pila*, dejando así la pila exactamente como estaba antes de ejecutar el *Prólogo*.

Los accesos a la pila se gestionan mediante un puntero, llamado *puntero de pila* (`sp = x2`), que almacena la dirección de la última posición ocupada de la pila. Antes de iniciar un programa, `sp` debe ser inicializado con la dirección de la base de la pila (es decir, la dirección más alta de la región de la pila). En el Sistema RVfpga, el registro `sp` se inicializa mediante la función `_start`, que se implementa en el archivo `~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/startup.S`. En la inicialización, la pila está vacía. Un segundo puntero, el *puntero de marco* (`fp = x8`),

apunta a la dirección base (es decir, la dirección más alta) del marco de pila de la función activa.

Las funciones utilizan el **marco de pila** como una región de memoria privada, a la que sólo se puede acceder desde la propia función. Una parte del **marco de pila** se dedica a guardar una copia de los registros arquitectónicos que han de ser modificados por la función y, en algunos casos, también puede utilizarse como una forma de pasar parámetros a la función a través de posiciones de la memoria.

Tabla 1 describe el papel que el convenio de RISC-V asigna a cada registro entero. Como también se ilustra en la misma, una función llamada debe preservar algunos registros, mientras que otros pueden ser sobrescritos por la función (es decir, no se preservan).

- Si la función necesita sobrescribir algún registro preservado, primero debe hacer una copia de dicho registro en su *marco de pila* y restaurar el valor antes de volver a la función que la llamó. Además del puntero de pila (*sp*) y del registro de dirección de retorno (*ra*), se preservan a través de las llamadas doce registros enteros *s0-s11*, que deben ser guardados por la función llamada si ésta los utiliza.
- Por otra parte, la función que llama debe ser consciente de que algunos registros no necesitan ser preservados por *la función llamada* y, por lo tanto, podrían perderse después de la llamada. Obsérvese que, además de los registros de argumentos y de valores de retorno (*a0-a7*), siete registros enteros *t0-t6* son registros temporales que son volátiles entre llamadas y que deben ser guardados por la función que llama si se vuelven a utilizar después invocar a la función.

**Tabla 1. Registros enteros de RISC-V**

Nombre	Número de registro	Utilice	Preservado
<b>zero</b>	<b>x0</b>	Valor constante 0	-
<b>ra</b>	<b>x1</b>	Dirección de retorno	Sí
<b>sp</b>	<b>x2</b>	Puntero de pila	Sí
<b>gp</b>	<b>x3</b>	Puntero global	-
<b>tp</b>	<b>x4</b>	Puntero de thread	-
<b>t0-2</b>	<b>x5-7</b>	Variables temporales	No
<b>s0/fp</b>	<b>x8</b>	Variable conservada / Puntero de marco	Sí
<b>s1</b>	<b>x9</b>	Variable conservada	Sí
<b>a0-1</b>	<b>x10-11</b>	Argumentos de función / Valores de retorno	No
<b>a2-7</b>	<b>x12-17</b>	Argumentos de la función	No
<b>s2-11</b>	<b>x18-27</b>	Variables conservadas	Sí
<b>t3-6</b>	<b>x28-31</b>	Variables temporales	No

En el ejemplo de la Figura 4, habría dos soluciones conforme a este convenio:

- El programa `main` podría usar, en lugar de `t0`, un registro para el índice de bucle cuya conservación por la función `SortVector` estuviese garantizada (como `s0`).
- La función `main` podría seguir usando `t0`, pero en ese caso tendría que preservar su contenido en la pila antes de llamar a `SortVector`, y restaurarlo después de regresar de `SortVector`.

La pila se expande a medida que se necesita más memoria para los marcos de pila de las funciones y se contrae según esas funciones finalizan. La pila crece hacia abajo (hacia

direcciones inferiores) y el puntero de pila se alineará a un límite de 16 bytes al entrar en el procedimiento. En la ABI estándar, el puntero de la pila debe permanecer alineado durante la ejecución del procedimiento.

## Ejemplo

El siguiente ejemplo implementa un algoritmo de ordenación, primero en C (Figura 5) y luego en el lenguaje ensamblador de RISC-V (Figura 6). La entrada es un array A de N elementos, cada uno de los cuales es un entero mayor que 0. La salida es otro array, B, que almacena los elementos de A en orden decreciente.

En C, la función `main` llama a la función `SortVector`, que recibe las direcciones de los arrays A y B, y su tamaño (N), y almacena los elementos de A en B, elemento por elemento, en orden decreciente. Esta función `SortVector` llama a otra función, `MaxVector`, que recibe la dirección del array A y su tamaño, y devuelve el valor máximo del array A y resetea ese valor, de modo que éste ya no se considera en las siguientes iteraciones.

```
#define N 8

int MaxVector(int A[], int size)
{
    int max=0, ind=0, j;
    for(j=0; j<size; j++){
        if(A[j]>max){
            max=A[j];
            ind=j;
        }
    }
    A[ind]=0;
    return(max);
}

int SortVector(int A[], int B[], int size)
{
    int max, j;
    for(j=0; j<size; j++){
        max=MaxVector(A, size);
        B[j]=max;
    }
    return(0);
}

int main ( void )
{
    int A[N]={7,3,25,4,75,2,1,1}, B[N];
    SortVector(A, B, N);
    return(0);
}
```

**Figura 5. Algoritmo de ordenación en lenguaje C**

Figura 6 ilustra el mismo algoritmo escrito en ensamblador. A continuación, se analiza el programa teniendo en cuenta los conceptos explicados en las secciones anteriores.

### - función `main`

#### o Prólogo

- En primer lugar, se reserva espacio en la pila para almacenar los registros preservados que se utilizan en la función: `add sp, sp, -16`. Nótese que, de acuerdo con el convenio, el registro `sp` debe mantenerse siempre alineado en 16 bytes para mantener la

- compatibilidad con la versión de 128 bits de RISC-V, RV128I.
    - Dado que esta función no utiliza ningún registro preservado, no es necesario que los registros `s0-s11` se almacenen en la pila. Sin embargo, el registro `ra` debe ser guardado, dado que la función `main` llama a la función `SortVector`, que actualiza el valor almacenado en `ra`.
  - Cuerpo de la función
    - La función `SortVector` se invoca usando la instrucción `jal SortVector`. Antes de llamar a la función, según el Convenio de Llamadas, los 3 parámetros de entrada se colocan en los registros `a0` (dirección de A), `a1` (dirección de B), y `a2` (tamaño de los arrays A y B).
  - Epílogo
    - El registro que se guardó en la pila en el prólogo (`ra`) se restaura en este punto.
    - El puntero de pila (`sp`) también se restaura a su posición inicial: `add sp, sp, 16`.
- **Función `SortVector`**
  - Prólogo
    - En primer lugar, se reserva espacio en la pila para almacenar los registros preservados que se utilizan en la función: `add sp, sp, -32`.
    - Luego, los registros guardados que utiliza la función (`s1-s3`) se almacenan en la pila, uno por uno.
    - El registro `ra` también debe guardarse, porque `SortVector` llama a la función `MaxVector`, que sobrescribe el valor almacenado en `ra`.
  - Cuerpo de la función
    - En primer lugar, los parámetros de entrada (`a0`, `a1` y `a2`) se mueven a registros preservados (`s1`, `s2` y `s3`), para que puedan ser utilizados después de la ejecución de la función `MaxVector`.
    - Para calcular el vector B, se implementa un bucle que, en cada iteración, calcula el valor máximo de A y lo almacena en B. Para calcular el valor máximo de A, se invoca la función `MaxVector` en cada iteración del bucle: `jal MaxVector`. Antes de llamar a la función, según el Convenio de Llamadas, los parámetros de entrada de esta función se trasladan a los registros `a0` y `a1`. Cuando la función termina de ejecutarse, devuelve el valor máximo de A en el registro `a0`.
    - Tenga en cuenta que el bucle usa los registros guardados sobre todo para almacenar variables. El Convenio de Llamadas de RISC-V garantiza que estos registros preservan su valor después de la ejecución del `MaxVector` (es decir, la función debe preservar sus valores).
    - Los registros `a0` y `a1` pueden ser modificados por la función. Por lo tanto, deben ser preparados antes de cada invocación.
    - El registro `t1` necesita ser reutilizado después de que `MaxVector` regrese. Por lo tanto, debe ser preservado en la pila de `SortVector` antes de llamar a la función (`sw t1, 16(sp)`) y restaurado después de ejecutarla (`lw t1, 16(sp)`).
  - Epílogo

- Los registros que se guardaron en la pila durante el prólogo, se deben restaurar en este punto.
- El puntero de la pila (*sp*) también se devuelve a su posición inicial:  
`add sp, sp, 32.`

## - Función **MaxVector**

### ○ Prólogo

- Primero, se reserva espacio en la pila para almacenar los registros preservados que se utilizan en la función: `add sp, sp, -16.`
- Después, el registro preservado que utiliza la función (es decir, el registro *s1*) se almacena en la pila: `sw s1, 0(sp)`. Obsérvese que si este registro no fuera preservado por esta función, la ejecución de la función invocante (*SortVector*) fallaría, ya que también está utilizando este registro para almacenar la dirección del vector A.
- Dado que esta función no invoca a ninguna otra (es una función *hoja*), no es necesario guardar *ra* en este caso.

### ○ Función Cuerpo

- La función utiliza *s1* y algunos registros temporales para calcular el valor máximo del array A.

### ○ Epílogo

- La función debe preparar el valor de retorno antes de volver a la función invocante: `mv a0, t2.`
- El registro que se guardó en la pila durante el prólogo (*s1*), se restaura en este punto.
- El puntero de la pila (*sp*) también se devuelve a su posición inicial:  
`add sp, sp, 16.`

```
.globl main

.equ N, 8

.data
A: .word 7,3,25,4,75,2,1,1

.bss
B: .space 4*N

.text

MaxVector:
    add sp, sp, -16
    sw s1, 0(sp)

    mv s1, zero
    mv t2, zero
loop2:
    beq s1, a1, endloop2
    lw t1, (a0)
    ble t1, t2, else2
    mv t2, t1
    mv t3, a0
else2:
    add a0, a0, 4
    add s1, s1, 1
    j loop2
endloop2:
    sw zero, (t3)
```

```

mv a0, t2
lw s1, 0(sp)
add sp, sp, 16
ret

SortVector:
    add sp, sp, -32
    sw s1, 0(sp)
    sw s2, 4(sp)
    sw s3, 8(sp)
    sw ra, 12(sp)

    mv s1, a0          # Address of vector A
    mv s2, a1          # Address of vector B
    mv s3, a2          # Size of vectors A and B
    mv t1, zero

loop1:
    beq t1, s3, endloop1
    mv a0, s1
    mv a1, s3
    sw t1, 16(sp)
    jal MaxVector
    lw t1, 16(sp)
    sw a0, (s2)
    add s2, s2, 4
    add t1, t1, 1
    j loop1
endloop1:

    lw s1, 0(sp)
    lw s2, 4(sp)
    lw s3, 8(sp)
    lw ra, 12(sp)
    add sp, sp, 32
    ret

main:
    add sp, sp, -16
    sw ra, 0(sp)

    la a0, A
    la a1, B
    add a2, zero, N
    jal SortVector

    lw ra, 0(sp)
    add sp, sp, 16
    ret

.end

```

**Figura 6. Algoritmo de ordenación en lenguaje ensamblador**

Figura 7 ilustra el estado de la pila cuando se ejecuta el cuerpo de la función `MaxVector`.

- El *marco de pila* de la función `main` se muestra en azul, e incluye la dirección de retorno (`ra`) para esa función.
- El *marco de pila* de la función `SortVector` se muestra en verde, e incluye los registros guardados que utiliza esta función (`s1-s3`), el registro `t1` y `ra`.
- Por último, el *marco de pila* de la función `MaxVector`, que es el *marco de pila activo* (el *marco de pila* de la función que se está ejecutando), se muestra en amarillo, e

incluye el registro guardado que utiliza esta función (s1).



**Figura 7. Estado de la pila en el cuerpo de la función `MaxVector` para el programa en lenguaje ensamblador de la Figura 6.**

**TAREA:** El programa en lenguaje ensamblador de la Figura 6 se proporciona en un proyecto de PlatformIO disponible en:

`[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions`. Ejecute este programa en la placa (o en el simulador del ISS) utilizando la opción de depuración paso a paso para analizar los valores almacenados en los distintos registros (s, ra, a, etc.), así como los valores almacenados en la pila, conforme al Convenio de Llamadas de RISC-V.

- El archivo `.pio/build/swervolf_nexys/firmware.dis`, generado por PlatformIO después de la compilación del programa, puede ser útil para conocer las direcciones de cada instrucción de su programa.

- Puede usar la consola de memoria para analizar la evolución de la pila, así como el contenido de los arrays A y B.

- En este proyecto se usa un script `link.ld` modificado para que el registro `sp` esté alineado en 16 bytes. Puede encontrar el script en `[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions/ld/link.ld`. La alineación del registro `sp` se fuerza usando el comando `ALIGN()`:

```
.stack :
{
    _heap_end = .;
    . = . + __stack_size;
    /* Force 16-B alignment of SP register */
    . = ALIGN(16);
}
```

```
_sp = .;  
} > ram : ram_load
```

## 5. Ejercicios

Ahora el usuario podrá crear sus propios programas de C/Ensamblador que incluyan llamadas a función realizando los siguientes ejercicios.

Recuerde que, si deja la placa Nexys A7 conectada a su computadora y encendida, no necesita recargar RVfpgaNexys en la placa entre diferentes programas. Sin embargo, si usted apaga la placa Nexys A7, tendrá que recargar RVfpgaNexys en la placa usando PlatformIO.

Recuerde que también puede ejecutar los programas en simulación, tanto en Verilator como en Whisper

**Ejercicio 1.** Escriba un programa en C que muestre en los LEDs valor inverso de los interruptores. Guarde el programa como **DisplayInverse\_Functions.c**.

Por ejemplo, si los interruptores son (en binario): 01010101010101, entonces los LEDs deben mostrar: 1010101010101010; si los interruptores son: 1111000011110000, los LEDs mostrarán: 0000111100001111; y así sucesivamente. Incluya una función `getSwitchesInvert()` que devuelva el valor invertido de los interruptores. La declaración de la función es:

```
unsigned int getSwitchesInvert();
```

**Ejercicio 2.** Escriba un programa C que proyecte el valor de los interruptores sobre los LEDs. Guarde el programa como **FlashSwitchesToLEDs\_Functions.c**

Los LEDs deberían encenderse y apagarse cada dos segundos. Incluya una función denominada `delay()` que provoque un retardo de *num* milisegundos. Esto puede hacerse empíricamente y no necesita ser exacto. La declaración de la función sería como sigue:

```
void delay(int num);
```

**Ejercicio 3.** Escriba un programa en C que mida el tiempo de reacción. Su programa debe medir el tiempo que tarda una persona en activar el interruptor de la derecha (SW[0]) después de que todos los LEDs se enciendan. Usará la función `rand()` de la librería `stdlib.h` para generar una cantidad aleatoria de tiempo de retardo entre cada vez que el usuario intente probar su tiempo de reacción. Guarde el programa como **ReactionTime.c**.

El programa debería funcionar de la siguiente manera:

1. El usuario apaga el interruptor de la derecha (hacia abajo) para indicar que le gustaría empezar.
2. El programa apaga todos los LEDs, y luego espera un tiempo aleatorio (pero no más de 3 segundos). Podrá utilizar para ello la función `delay()` del ejercicio 2.



3. A continuación todos los LEDs se encienden y el programa comienza a contar el número de milisegundos que transcurren hasta que el usuario enciende el interruptor de la derecha (hacia arriba).
4. Cuando el usuario activa el interruptor de la derecha (SW[0]), el número de milisegundos que invirtió en ello se muestra en binario en los LEDs y en decimal en la consola serie.
5. El juego se repite cuando el usuario apaga el interruptor de la derecha (hacia abajo).

**Ejercicio 4.** Un problema con la función `rand()` es que utiliza una secuencia de números aleatoria predecible. Es decir, cada vez que ejecute el programa comenzará con el mismo número aleatorio y seguirá la misma secuencia de números aleatorios. Ejecute el programa del Ejercicio 3 varias veces para comprobar que comienza con el mismo número aleatorio y sigue la misma secuencia aleatoria.

Sin embargo, si primero usa la función `srand()`, está establecerá en la función `rand()` una semilla aleatoria. El único problema es que a `srand()` se le debe dar un argumento de entrada, un entero sin signo, que en sí mismo es aleatorio. Proporcione a `srand()` un número aleatorio, por ejemplo, el número de milisegundos hasta que el usuario apague el interruptor para comenzar el juego.

Haga de nuevo el ejercicio 3 para producir una secuencia verdaderamente aleatoria de tiempos antes de que los LEDs se enciendan. Utilice funciones cuando sea posible. Guarde el programa como **ReactionTimeTrulyRandom.c**.

**Ejercicio 5.** Reescriba el ejercicio 4 de modo que los LEDs muestren una barra creciente de LEDs, proporcional al tiempo de reacción. De esta manera, la persona que ve su tiempo de reacción puede saber más fácilmente si se está mejorando su tiempo, sin tener que interpretar la representación binaria del número de milisegundos. Puede elegir el rango de tiempos de reacción correspondiente a cada rango de LEDs encendidos. Por ejemplo, para los tiempos de reacción reducidos, sólo unos pocos LEDs a la derecha deberían encenderse. Un número creciente de LEDs a la izquierda debería iluminarse a medida que los tiempos de reacción aumentan. Un tiempo de reacción muy elevado encendería todos los LEDs. Guarde el programa como **ReactionTimeBar.c**.

**Ejercicio 6.** Escriba un programa en C que implemente el juego "Simón dice". Debería suceder lo siguiente:

1. El programa establece un patrón de parpadeo en los tres LEDs de la derecha y espera a que el usuario pulse la secuencia correspondiente a dicho patrón en los tres interruptores de la derecha. Los interruptores [2:0] corresponden a los LED[2:0], siendo el LED[0] el de más a la derecha y el interruptor [0] también el de más a la derecha.
2. Los patrones aleatorios deben comenzar encendiendo 1 LED, luego 2 LED, luego 3, etc.
3. El usuario entonces intenta repetir la secuencia usando los tres interruptores de la derecha. El LED correspondiente debería iluminarse cuando el usuario cambia el interruptor hacia arriba (y apagarse cuando vuelve a cambiar el interruptor hacia abajo).
4. Si el usuario introduce la secuencia correcta, después de una pausa debería aparecer el siguiente patrón, con un LED más en la secuencia.
5. Si el usuario introduce una secuencia incorrecta, los LEDs permanecen encendidos y no se reproduce ninguna secuencia nueva.
6. El juego se reinicia pulsando el interruptor más a la izquierda (Switches[15]) hacia arriba (on) y luego hacia abajo (off).

Utilice las funciones que desee para modularizar el programa y facilitar su escritura, depuración y comprensión. Recuerde usar las librerías estándar de C como desee para escribir su programa. Guarde el programa como **SimonSays.c**.

**Ejercicio 7.** Dado un vector, A, de  $3*N$  elementos, se desea obtener un nuevo vector, B, de N elementos, de manera que cada elemento de B sea el valor absoluto de la suma de un trío de elementos consecutivos de A. Por ejemplo:

$$B[0] = |A[0]+A[1]+A[2]|, B[1] = |A[3]+A[4]+A[5]|, \dots$$

Escriba un programa en lenguaje ensamblador de RISC-V llamado **Triplets.S** (el programa debe ajustarse a la convenio de llamadas de RISC-V):

- El programa `main` implementa el cálculo de B, de acuerdo con el siguiente pseudo-código de alto nivel:

```
#define N 4

int A[3*N] = {a list of 3*N values};
int B[N];
int i, j=0;

void main (void)
{
    for (i=0; i<N; i++){
        B[i] = res_triplet(A,j);
        j=j+3;
    }
}
```

- La función `res_triplet` devuelve el valor absoluto de la suma de 3 elementos consecutivos del vector V, comenzando en la posición p. Se implementa conforme a la especificación dada por el siguiente pseudo-código de alto nivel:

```
int res_triplet(int V[ ], int pos)
{
    int i, sum=0;
    for (i=0; i<3; i++)
        sum = sum + V[pos+i];
    sum=abs(sum);
    return sum;
}
```

- La función `abs(int x)` devuelve el valor absoluto de su argumento de entrada.

**Ejercicio 8.** Escriba un programa en lenguaje ensamblador de RISC-V llamado **Filter.S** (el programa debe cumplir con el estándar de gestión de funciones estudiado anteriormente). Puede utilizar el siguiente pseudocódigo:

```
#define N 6
int i, j=0, A[N]={48,64,56,80,96,48}, B[N];
```

```

for (i=0; i<(N-1); i++){
    if( (myFilter(A[i],A[i+1])) == 1){
        B[j]=A[i]+ A[i+1] + 2;
        j++;
    }
}

```

- Escriba el código equivalente en lenguaje ensamblador de RISC-V, incluyendo las directivas necesarias para reservar el espacio de memoria, y declarando las secciones correspondientes (.data, .bss y .text). La función `myFilter` devuelve el valor 1 si el primer argumento es un múltiplo de 16 y el segundo es mayor que el primero; en caso contrario, devuelve un 0.
- Escriba el código ensamblador de la función `myFilter`.

**Ejercicio 9.** Se desea realizar un programa en ensamblador de RISC-V llamado **Coprimes.S** (el programa debe cumplir con el estándar de gestión de funciones estudiado anteriormente), de tal forma que dada una lista de pares de números enteros ( $> 0$ ) encuentre qué pares están compuestos de números coprimos (o primos mutuos). Se entiende que dos números son coprimos si el único divisor común que tienen es el 1.

Se supone que los datos de entrada están contenidos en un array, D, de la forma:

$$D = (x_0, y_0, c_0, x_1, y_1, c_1, \dots, x_{N-1}, y_{N-1}, c_{N-1})$$

Cada terna  $(x_i, y_i, c_i)$  se interpreta como sigue:  $x_i$  e  $y_i$  representan un par de números, y  $c_i$  es inicialmente 0. Después de ejecutar el programa, el valor de cada  $c_i$  debe haber sido modificado de tal manera que  $c_i = 2$ , si  $x_i$  e  $y_i$  son coprimos; y  $c_i = 1$ , en caso contrario.

Por ejemplo:

Para el vector de entrada:  $D = (3,5,0, 6,18,0, 15,45,0, 13,10,0, 24,3,0, 24,35,0)$   
 el resultado final debería ser:  $D = (3,5,2, 6,18,1, 15,45,1, 13,10,2, 24,3,1, 24,35,2)$

- Escriba un programa en lenguaje ensamblador de RISC-V que recorra el array D y genere el resultado de acuerdo con la especificación dada en el cuadro inferior a la izquierda. El programa llama a la función `check_coprime(int D [], int i)`, cuyos argumentos son la dirección inicial de D y el número del par que se desea comprobar (de 0 a M-1). La función comprueba si los números del par i-ésimo del array D son coprimos y almacena el resultado en la posición de memoria correspondiente.
- Escriba el código de la función `check_coprime`, de acuerdo con la especificación dada en el cuadro inferior a la derecha. Recuerde que la función `gcd(int a, int b)` se implementó en la Práctica 3 según el algoritmo euclidian, y devuelve el mayor divisor común (gcd) de los dos argumentos de entrada. Si el gcd es 1, entonces los números son coprimos.

<pre> #define M 6 int D[] = {a list de M*3 int values}  void main ( ) {     int i;     for (i=0; i&lt;M; i++)         check_coprime(D,i); } </pre>	<pre> void check_coprime (int A[ ], int pos) {     int res;     res= gcd( A[3*pos], A[(3*pos)+1] );     if (res == 1)         A[(3*pos)+2]=2;     else         A[(3*pos)+2]=1; } </pre>
--	---