



PROGRAMA UNIVERSITARIO DE IMAGINATION

Práctica 3 RVfpga

Lenguaje ensamblador de RISC-V

1. INTRODUCCIÓN

La programación en lenguajes de alto nivel como C, Java y Python es eficiente para el programador. Estos lenguajes de alto nivel se traducen a lenguaje ensamblador, que es un conjunto de instrucciones simples. A veces las secciones de código críticas para el rendimiento o el tiempo de ejecución se escriben en ensamblador para garantizar un tiempo específico o reducir el tiempo de cálculo. En esta práctica se describe cómo crear un programa en lenguaje ensamblador de RISC-V que pueda ser ejecutado en el Sistema RVfpga usando PlatformIO. Para ello, en primer lugar se proporciona una breve visión general del lenguaje ensamblador de RISC-V y a continuación se indica cómo crear y ejecutar un programa en lenguaje ensamblador sobre RVfpgaNexys (recuerde que también se pueden ejecutar los programas en simulación, tanto en Verilator como en Whisper). Finalmente se proporcionan ejercicios para que el usuario practique mediante la escritura de sus propios programas en lenguaje ensamblador de RISC-V.

2. Visión global del lenguaje Ensamblador de RISC-V

El lenguaje ensamblador de RISC-V incluye instrucciones simples que se utilizan para implementar código de alto nivel. Por ejemplo, algunas instrucciones comunes de RISC-V son las instrucciones `add`, `sub` y `mul`, que suman, restan o multiplican dos operandos, respectivamente.

Los tipos básicos de instrucciones RISC-V son: instrucciones de cálculo (aritméticas, lógicas y de desplazamiento), operaciones de memoria y saltos condicionales e incondicionales. Las instrucciones RISC-V más comunes se muestran en la Tabla 1. Las instrucciones utilizan operandos que se encuentran en registros o memoria o que están codificados como una constante (es decir, *inmediatos*). RISC-V incluye 32 registros de 32 bits. En la Tabla 2 se listan los nombres de los 32 registros de RISC-V. Pueden especificarse por su nombre (por ejemplo, `zero`, `s0`, `t5`, etc.) o por su número de registro (es decir, `x0`, `x8`, `x30`). Los programadores suelen utilizar nombres de registro que conllevan cierta información sobre el propósito típico del registro. Por ejemplo, los registros preservados (*saved*), `s0-s11`, se utilizan típicamente para las variables del programa, mientras que los registros temporales, `t0-t6` se utilizan para los cálculos temporales. El registro cero (`x0`) siempre contiene el valor 0, ya que es un valor comúnmente necesario en los programas. Los otros registros también tienen usos específicos, como se muestra en la Tabla 2, pero en esta práctica, el usuario sólo necesita usar el registro `zero` y los registros temporales y preservados.

Tabla 1. Instrucciones comunes del lenguaje ensamblador de RISC-V

	Ensamblador RISC-V	Descripción	Operación
Cálculo	<code>add s0, s1, s2</code>	Suma	$s0 = s1 + s2$
	<code>sub s0, s1, s2</code>	Resta	$s0 = s1 - s2$
	<code>addi t3, t1, -10</code>	Suma inmediata	$t3 = t1 - 10$
	<code>mul t0, t2, t3</code>	Multiplicación de 32 bits	$t0 = t2 * t3$
	<code>div s9, t5, t6</code>	División	$t9 = t5 / t6$
	<code>rem s4, s1, s2</code>	Resto	$s4 = s1 \% s2$
	<code>and t0, t1, t2</code>	Y lógica bit a bit	$t0 = t1 \text{ y } t2$
	<code>or t0, t1, t5</code>	O lógica bit a bit	$t0 = t1 t5$
	<code>xor s3, s4, s5</code>	O-exclusiva bit a bit	$s3 = s4 \wedge s5$
	<code>andi t1, t2, 0xFFB</code>	Y lógica bit a bit inmediata	$t1 = t2 \& 0xFFFFFFFFB$
	<code>ori t0, t1, 0x2C</code>	O lógica bit a bit inmediata	$t0 = t1 0x2C$

	xori s3, s4, 0xABC	O-exclusiva inmediata	s3 = s4 ^ 0xFFFFFABC
	sll t0, t1, t2	Desplazamiento lógico a la izquierda	t0 = t1 << t2
	srl t0, t1, t5	Desplazamiento lógico a la derecha	t0 = t1 >> t5
	sra s3, s4, s5	Desplazamiento aritmético a la derecha	s3 = s4 >>> s5
	slli t1, t2, 30	Desplazamiento lógico a la izquierda inmediato	t1 = t2 << 30
	srli t0, t1, 5	Desplazamiento lógico a la derecha inmediato	t0 = t1 >> 5
	srai s3, s4, 31	Desplazamiento aritmético a la derecha inmediato	s3 = s4 >> > 31
Memoria	lw s7, 0x2C(t1)	Leer palabra	s7 = memory[t1+0x2C]
	lh s5, 0x5A(s3)	Leer media palabra	s5 = SignExt(memory[s3+0x5A] _{15:0})
	lb s1, -3(t4)	Leer byte	s1 = SignExt(memory[t4-3] _{7:0})
	sw t2, 0x7C(t1)	Almacenar palabra	memory[t1+0x7C] = t2
	sh t3, 22(s3)	Almacenar media palabra	memory[s3+22] _{15:0} = t3 _{15:0}
	sb t4, 5(s4)	Almacenar Byte	memory[s4+5] _{7:0} = t4 _{7:0}
Salto	beq s1, s2, L1	Saltar si igual	si (s1==s2), PC = L1
	bne t3, t4, Loop	Saltar si no igual	si (s1!=s2), PC = Loop
	blt t4, t5, L3	Saltar si menor que	si (t4 < t5), PC = L3
	bge s8, s9, Done	Saltar si mayor o igual que	si (s8>=s9), PC = Done
Pseudoinstrucciones	li s1, 0xABCDEF12	Carga de inmediato	s1 = 0xABCDEF12
	la s1, A	Cargar Dirección	s1 = Dirección de memoria donde se almacena la variable A
	nop	No operación	no hay operación
	mv s3, s7	Mover	s3 = s7
	no t1, t2	No (Invertir)	t1 = ~t2
	neg s1, s3	Negar	s1 = -s3
	j Label	Saltar	PC = Label
	jal L7	Saltar y guardar	PC = L7; ra = PC + 4
	jr s1	Saltar a registro	PC = s1

Además de las instrucciones reales de RISC-V, RISC-V incluye pseudoinstrucciones (como se muestra en la parte inferior de la Tabla 1), instrucciones que no son realmente instrucciones de RISC-V pero que los programadores utilizan frecuentemente. Las pseudoinstrucciones se implementan usando una o más instrucciones reales de RISC-V. Por ejemplo, la pseudoinstrucción de movimiento (mv s1, s2) copia el contenido de s2 y lo pone en s1. Se implementa usando la instrucción real de RISC-V: addi s1, s2, 0.

Tabla 2. Los registros de RISC-V

Nombre	Número de registro	Útilice
zero	x0	Valor constante 0
ra	x1	Dirección de retorno
sp	x2	Puntero de pila
gp	x3	Puntero global
tp	x4	Puntero de hilo
t0-2	x5-7	Variables temporales
s0/fp	x8	Variable guardada / Puntero de frame

s1	x9	Variable guardada
a0-1	x10-11	Argumentos de función / Valores de retorno
a2-7	x12-17	Argumentos de función
s2-11	x18-27	Variables guardadas
t3-6	x28-31	Variables temporales

Los comandos que comienzan con un punto son directivas de ensamblador. Se trata de comandos para el ensamblador en lugar de código para ser traducido por el mismo. Le dicen al ensamblador dónde ubicar el código y los datos, especifican las constantes de texto y datos para su uso en el programa, etc. Tabla 3 muestra las principales directivas del ensamblador de RISC-V (*The RISC-V Reader: An Open Architecture Atlas*, Patterson & Waterman, © 2017).

Tabla 3. Principales directivas de RISC-V

Directiva	Descripción
.text	Los elementos que la siguen se almacenan en la sección <code>text</code> (código máquina).
.data	Los elementos que la siguen se almacenan en la sección <code>data</code> (variables globales).
.bss	Los elementos que la siguen se almacenan en la sección <code>bss</code> (variables globales inicializadas a 0).
.section .foo	Los elementos que la siguen se almacenan en la sección llamada <code>.foo</code> .
.align n	Alinea el siguiente dato en un límite de 2^n bytes. Por ejemplo, <code>.align 2</code> alinea el siguiente valor en un límite de palabra.
.balign n	Alinear el siguiente dato en un límite de n bytes. Por ejemplo, <code>.balign 4</code> alinea el siguiente valor en un límite de palabra.
.globl sym	Declara que la etiqueta <code>sym</code> es global y puede ser referenciada desde otros archivos
.string "str"	Almacena el string <code>str</code> en memoria y lo finaliza en NULL.
.word w1,..., wn	Almacena las n cantidades de 32 bits en palabras de memoria sucesivas.
.byte b1,..., bn	Almacena las n cantidades de 8 bits en bytes de memoria sucesivos.
.space	Reserva espacio de memoria para almacenar variables sin un valor inicial. Se utiliza comúnmente para declarar las variables de salida, cuando no se están utilizando también como variables de entrada. El espacio que queremos reservar debe expresarse siempre como un número de bytes. Por ejemplo, la directiva <code>RES: .space 4</code> reserva cuatro bytes (es decir, una palabra) que no están inicializados.
.equ name, constant	Define el símbolo <code>name</code> con el valor <code>constant</code> . Por ejemplo, <code>.equ N, 12</code> , define el símbolo <code>N</code> con el valor 12.
.end	El ensamblador concluirá su trabajo cuando llegue a la directiva <code>.end</code> . Se ignorará cualquier código situado después de esta directiva.

Los siguientes ejemplos (ver Tabla 4 y Tabla 5) muestran cómo codificar en ensamblador de RISC-V algunas estructuras típicas de alto nivel. Observe que las instrucciones de salto condicional (`bge`, `bne`, `blt` y `bge`) saltan condicionalmente a una etiqueta; mientras que la instrucción de salto incondicional (`j`) salta incondicionalmente a una etiqueta. Los comentarios de una sola línea se indican en C con `“//”` y con `“#”` en lenguaje ensamblador de RISC-V.

En el primer ejemplo (en el que se implementa una estructura `if/else`, ver Tabla 4), observe que el código C y el código en lenguaje ensamblador de RISC-V comprueban casos opuestos: el código C comprueba la condición menor que (`<`) y el equivalente de ensamblador chequea la condición mayor o igual (`>=`).

Tabla 4. Ejemplo 1 de Ensamblador de RISC-V: construcción `if/else`

// Código C	# Ensamblador de RISC-V
<code>int a, b, c;</code>	<code># s0 = a, s1 = b, s2 = c</code>
<code>if (a < b)</code>	<code>bge s0, s1, L1 # if (a >= b) goto L1</code>
<code> c = 5;</code>	<code>addi s2, zero, 5 # c = 5</code>
<code>else</code>	<code>j L2 # jump over else block</code>
<code> c = a + b;</code>	<code>L1: add s2, s0, s1 # c = a + b</code>
	<code>L2:</code>

En el segundo ejemplo (manipulación de un array de números enteros, véase Tabla 5), el código en lenguaje ensamblador de RISC-V utiliza registros temporales (`t0-t3`) para almacenar valores temporales, como la constante 100 y la dirección base del array de datos. Después de inicializar los registros en las tres primeras instrucciones, el código en lenguaje ensamblador de RISC-V comprueba si `i >= 100` usando la instrucción `bge` (salto si es mayor o igual que); de nuevo, este es el caso opuesto al código C. Si se cumple esa condición, se ejecuta el bucle `for`. Si **no** se toma el salto, `i` es menor que 100 y se ejecuta el resto del código. Observe que el índice `i` se multiplica por 4 (usando la instrucción `slli t2, s0, 2`) antes de sumarse a la dirección base porque los números enteros (números en complemento a dos de 32 bits) ocupan 4 bytes de memoria. En RISC-V, la memoria es direccionable por bytes (es decir, cada byte tiene su propia dirección). Si se tratase de un array de caracteres (es decir, `char data[100];`), entonces cada elemento del array sólo ocuparía un byte y podría añadirse directamente `i` a la dirección base para formar la dirección del elemento `i` del array, es decir, `array[i]`. Después de que se haya leído el elemento del array, se le haya restado diez y se haya escrito con el nuevo valor (mediante las instrucciones `lw`, `addi` y `sw`, respectivamente), el índice `i` del array (es decir, `s0`) se incrementa y el programa salta de nuevo al principio del bucle `for` (usando la instrucción `j L5`).

Tabla 5. Ejemplo 2 de Ensamblador de RISC-V: manipulación de un array de enteros

// Código C	# Ensamblador de RISC-V
<code>int i;</code>	<code># s0 = i, t1 = base address of data (assumed</code>
<code>int data[100];</code>	<code># to be at 0x300)</code>
	<code>addi s0, zero, 0 # i = 0</code>
	<code>addi t0, zero, 100 # t0 = 100</code>
	<code>li t1, 0x300 # base address of array</code>
<code>for (i=0; i<100; i++)</code>	<code>L5: bge s0, t0, L7 # if (i>=100) exit loop</code>
	<code>slli t2, s0, 2 # t2 = i*4</code>
	<code>add t2, t1, t2 # address of data[i]</code>

<pre>array[i] = array[i]-10;</pre>	<pre>lw t3, 0(t2) # t3 = array[i] addi t3, t3, -10 # t3 = array[i]-10 sw t3, 0(t2) # array[i] = array[i]-10 addi s0, s0, 1 # i++ j L5 # loop L7:</pre>
------------------------------------	---

Para más detalles sobre el lenguaje ensamblador de RISC-V, consulte el Manual del Repertorio de Instrucciones de RISC-V (disponible aquí: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>) o un libro de texto como *Digital Design and Computer Architecture*, Harris & Harris, Elsevier, © 2021 (cuya publicación se prevee para verano de 2021) o *The RISC-V Reader: An Open Architecture Atlas*, Patterson & Waterman, © 2017.

3. Escritura de un programa en ensamblador de RISC-V para RVfpga

En este momento el usuario está preparado para explorar y practicar por su cuenta la codificación de programas en lenguaje ensamblador de RISC-V. Antes de escribir sus propios programas, siga estos pasos para configurar un proyecto de PlatformIO y crear y ejecutar un programa en ensamblador sobre RVfpgaNexys (recuerde que también se pueden ejecutar los programas en simulación, tanto en Verilator como en Whisper):

1. Crear un proyecto RVfpga
2. Escribir un programa en lenguaje ensamblador de RISC-V
3. Descargar RVfpgaNexys en la placa FPGA Nexys A7
4. Compilar, descargar y ejecutar el programa en ensamblador

Paso 1. Crear un proyecto RVfpga

Siga el paso 1 de la práctica 2 de RVfpga, que se repite a continuación para su comodidad. Abra VSCode haciendo clic en el botón de inicio, escribiendo a continuación VSCode y luego haciendo clic en Virtual Studio Code (ver Figura 1).

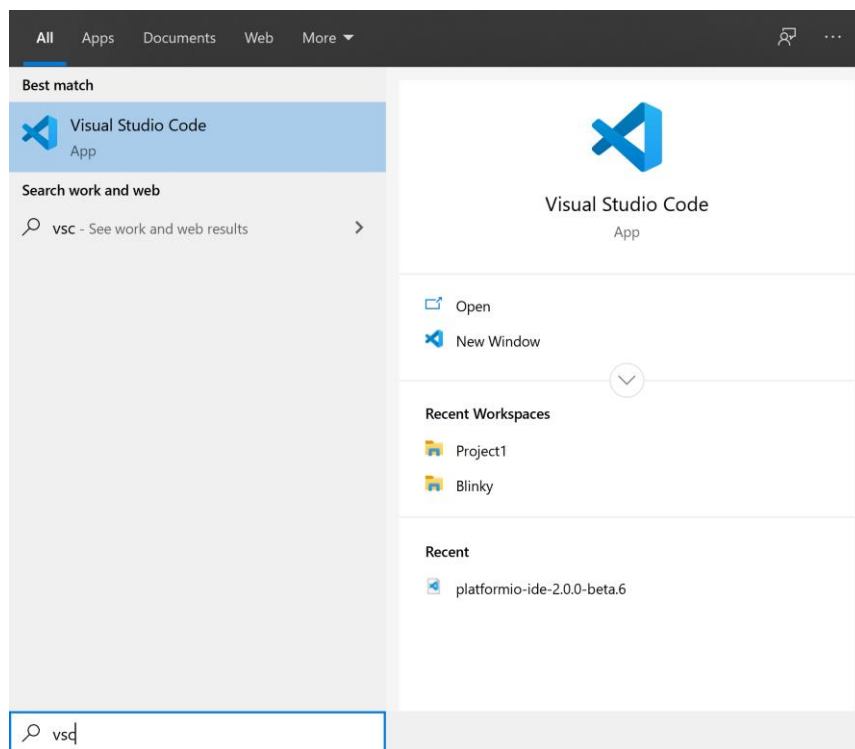


Figura 1. Abrir VSCode

Si PlatformIO no se abre automáticamente al iniciar VSCode, haga clic en el icono de PlatformIO en la barra del menú de la izquierda y luego haga clic en PIO Home → Open (ver Figura 2).

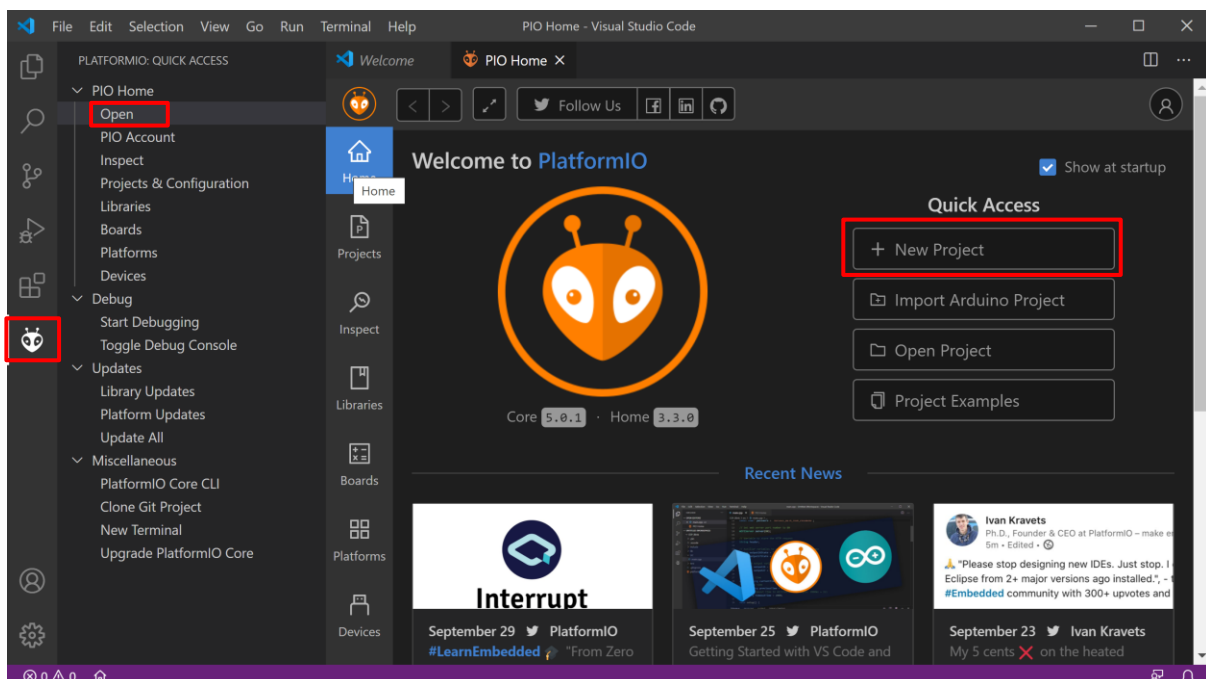


Figura 2. Abrir PlatformIO y crear un nuevo proyecto

En la ventana de bienvenida de PIO Home, haga clic en New Project (ver Figura 2).

Como se muestra en la Figura 3, nombre el proyecto como Project1 y elija como placa “RVfpga: Digilent Nexys A7” (comience a escribir RVfpga y aparecerá el nombre completo). Deje como entorno por defecto WD-framework (framework de Western Digital, que incluye el gcc y gdb de Freedom-E SDK). Haciendo clic desactive la opción Usar la ubicación predeterminada (Use default location) y guarde su programa en:

[RVfpgaPath] /RVfpga/Labs/Lab3

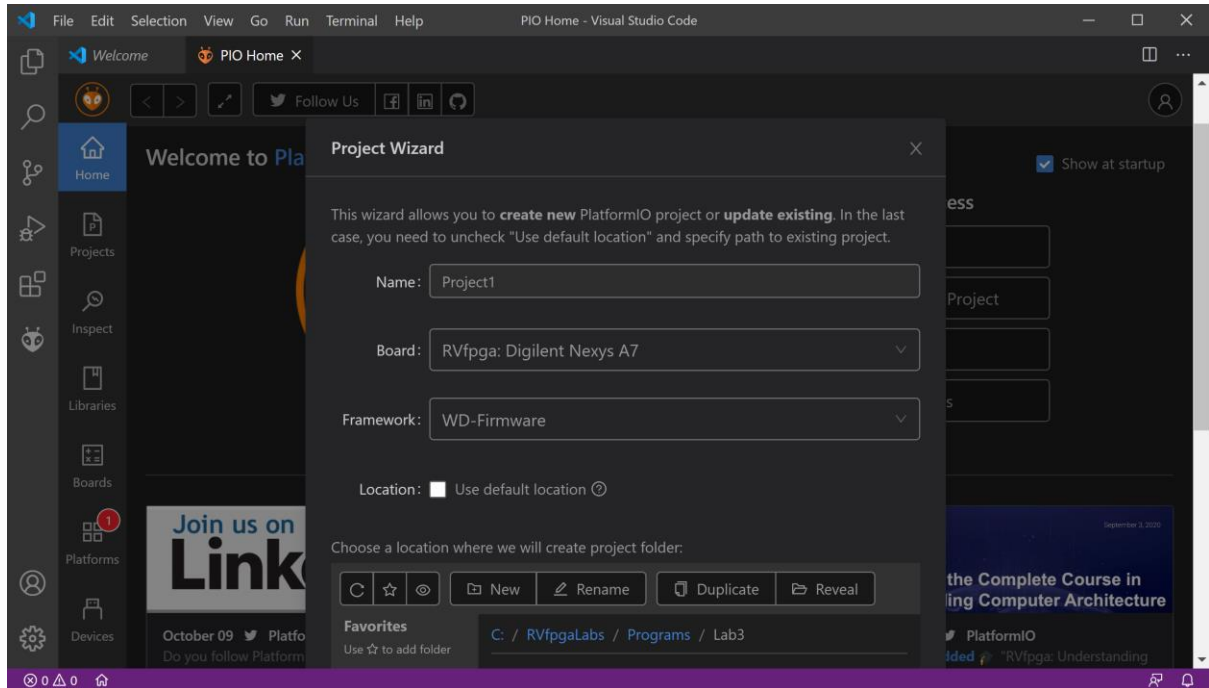


Figura 3. Nombre del proyecto y selección de placa y ubicación del proyecto

A continuación, haga clic en Finalizar en la parte inferior de la ventana (ver Figura 4).

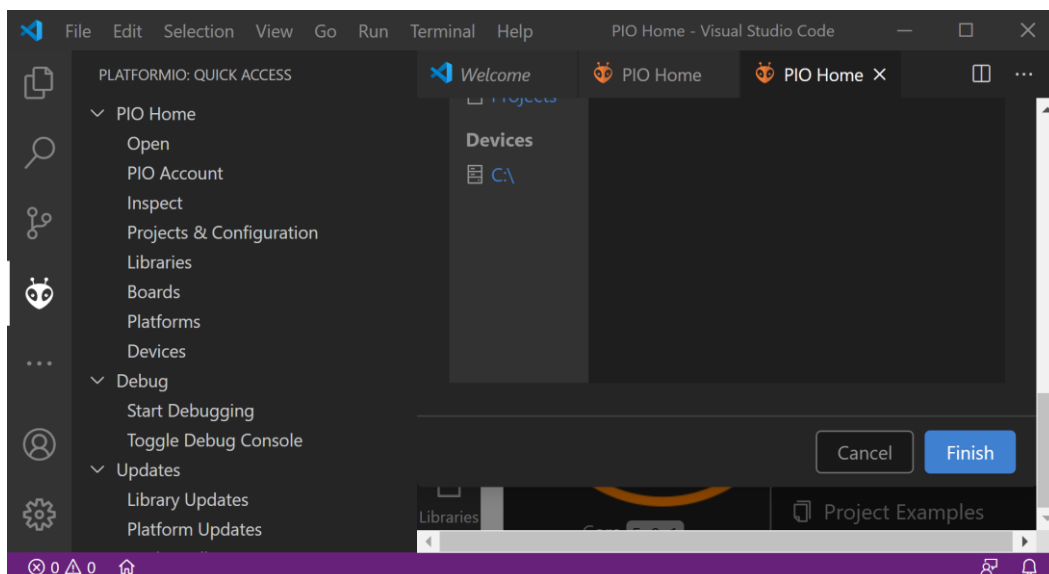


Figura 4. Finalizar la creación del proyecto

En el panel del explorador de la izquierda, en PROJECT1 (que tal vez necesite expandir), haga doble clic en `platformio.ini` para abrirlo (véase la Figura 5). Este es el archivo de inicialización de PlatformIO.

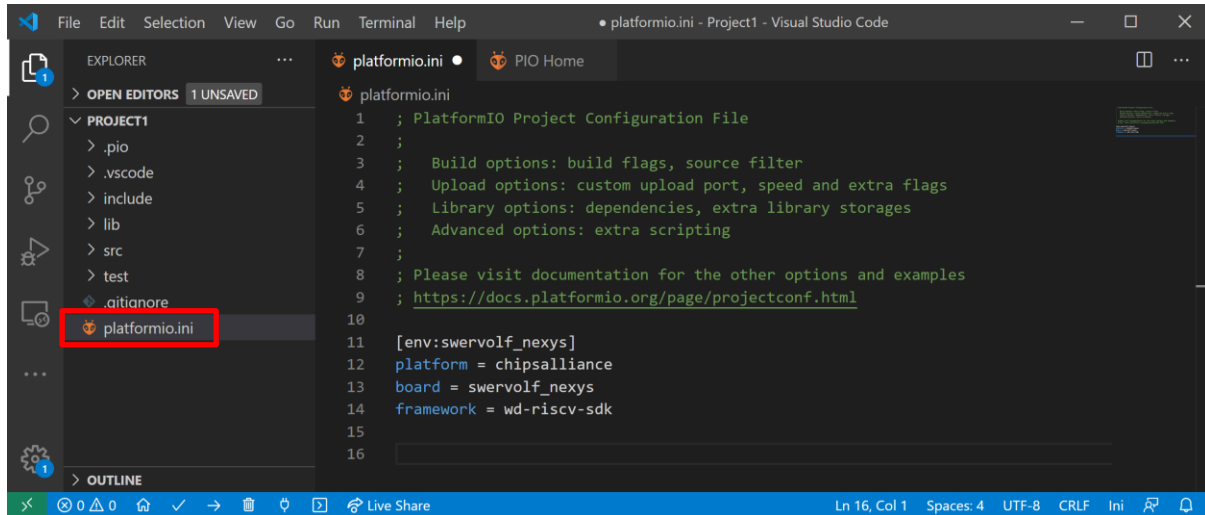


Figura 5. Archivo de inicialización de PlatformIO: `platformio.ini`

Añada la siguiente línea al archivo `platformio.ini`, como se muestra en la Figura 6:

```
board_build.bitstream_file =
[RVfpgaPath]/RVfpga/Labs/Lab1/Project1/Project1.runs/impl_1/rvfpganexys.bit
```

Esta línea le indica a PlatformIO en dónde debe buscar el archivo bitstream para cargar en la FPGA. La ruta incluida en la mencionada línea es la ubicación del archivo bitstream que se creó en la Práctica 1. (Si no ha completado la Práctica 1, puede usar el archivo bitstream de RVfpgaNexys distribuido con la Guía de inicio en: `[RVfpgaPath]/RVfpga/src/rvfpganexys.bit`). Presione Ctrl-s para guardar el archivo `platformio.ini`.

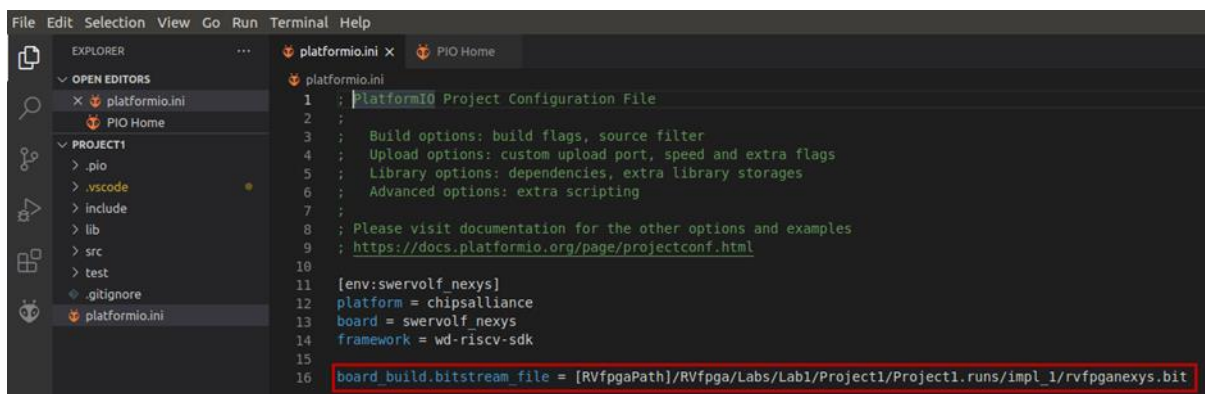


Figura 6. Agregar la ubicación del archivo bitstream de RVfpgaNexys (`rvfpganexys.bit`)

Recuerde que se utilizó un archivo `platformio.ini` más completo en los ejemplos utilizados en la Guía de inicio. Si desea utilizar alguna funcionalidad que requiera comandos adicionales (como la ruta al simulador Verilator, la configuración de la consola serie, la herramienta de depuración *whisper*, etc.), puede utilizar el archivo `platformio.ini` de esos ejemplos.

Paso 2. Escribir un programa en lenguaje ensamblador de RISC-V

Ahora el usuario escribirá un programa en lenguaje ensamblador de RISC-V. Haga clic en File → New File (ver Figura 7).

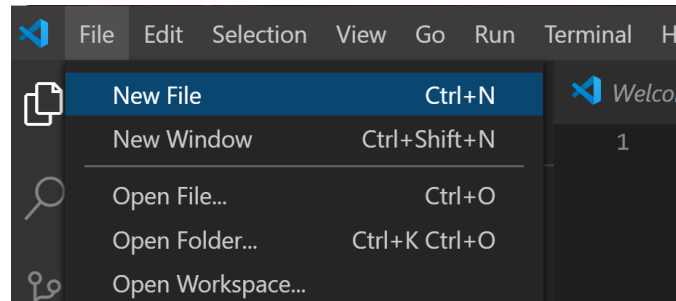


Figura 7. Añadir un archivo al proyecto

Se abrirá una ventana en blanco. Escriba (o copie/pegue) el siguiente programa en lenguaje ensamblador de RISC-V en esa ventana (vea la Figura 8). Este programa también está disponible en:

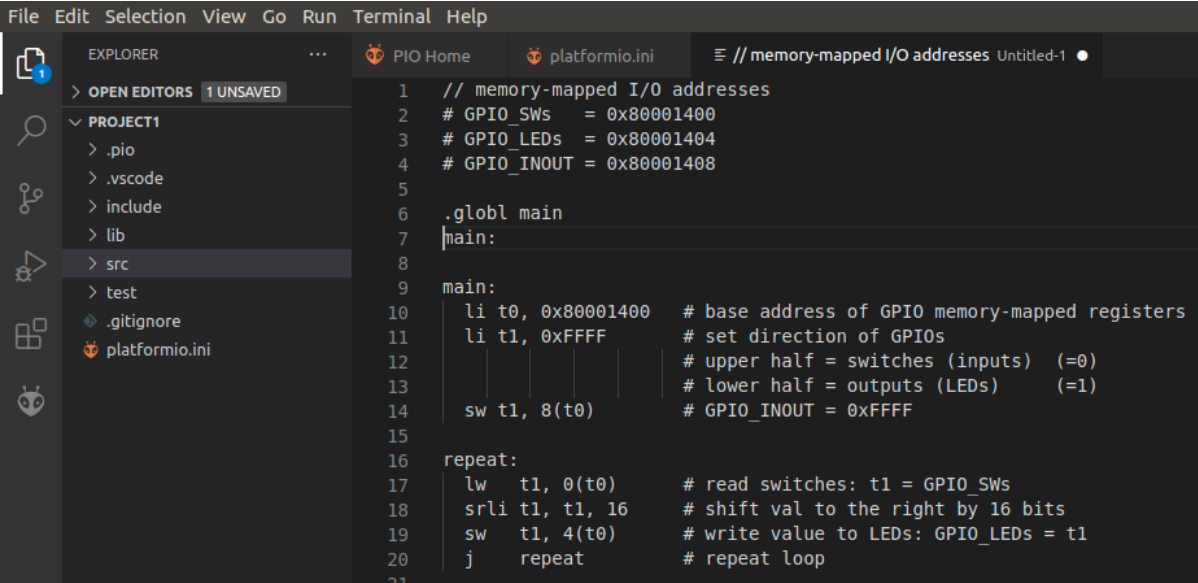
```
[RVfpgaPath]/RVfpga/Labs/Lab3/ReadSwitches.S
```

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

.globl main
main:

main:
    li t0, 0x80001400    # base address of GPIO memory-mapped registers
    li t1, 0xFFFF       # set direction of GPIOs
                        # upper half = switches (inputs)    (=0)
                        # lower half = outputs (LEDs)       (=1)
    sw t1, 8(t0)         # GPIO_INOUT = 0xFFFF

repeat:
    lw  t1, 0(t0)        # read switches: t1 = GPIO_SWs
    srli t1, t1, 16      # shift val to the right by 16 bits
    sw  t1, 4(t0)        # write value to LEDs: GPIO_LEDs = t1
    j   repeat           # repeat loop
```



```

1 // memory-mapped I/O addresses
2 # GPIO_SWs = 0x80001400
3 # GPIO_LEDS = 0x80001404
4 # GPIO_INOUT = 0x80001408
5
6 .globl main
7 main:
8
9 main:
10     li t0, 0x80001400 # base address of GPIO memory-mapped registers
11     li t1, 0xFFFF    # set direction of GPIOs
12                     # upper half = switches (inputs) (=0)
13                     # lower half = outputs (LEDs) (=1)
14     sw t1, 8(t0)      # GPIO_INOUT = 0xFFFF
15
16 repeat:
17     lw t1, 0(t0)      # read switches: t1 = GPIO_SWs
18     srli t1, t1, 16   # shift val to the right by 16 bits
19     sw t1, 4(t0)      # write value to LEDs: GPIO_LEDS = t1
20     j repeat          # repeat loop
21

```

Figura 8. Introducción del programa en lenguaje ensamblador de RISC-V

El código ensamblador debe contener las siguientes líneas al principio del código:

```
.globl main
main:
```

La directiva de ensamblador `.globl` hace que la etiqueta sea visible en todos los archivos enlazados. El código de arranque (`~/platformio/packages/framework-riscv-sdk/board/hexys_a7_eh1/startup.S`) configurará el sistema y saltará a esta etiqueta (`main`). El depurador establecerá un punto de interrupción temporal allí cuando comience.

Este programa en ensamblador de RISC-V es el mismo programa de ejemplo que en la Práctica 2, pero esta vez escrito en lenguaje ensamblador de RISC-V. El programa establece la dirección de las entradas y salidas de la E/S de propósito general (GPIO) y a continuación lee repetidamente el valor de los interruptores y escribe ese valor en los LEDs.

Después de introducir el programa en el editor de texto, presione Ctrl-s para guardar el archivo. Nómbrelo como `ReadSwitches.S` y guárdelo en la carpeta `src` del directorio `Project1` (ver Figura 9).

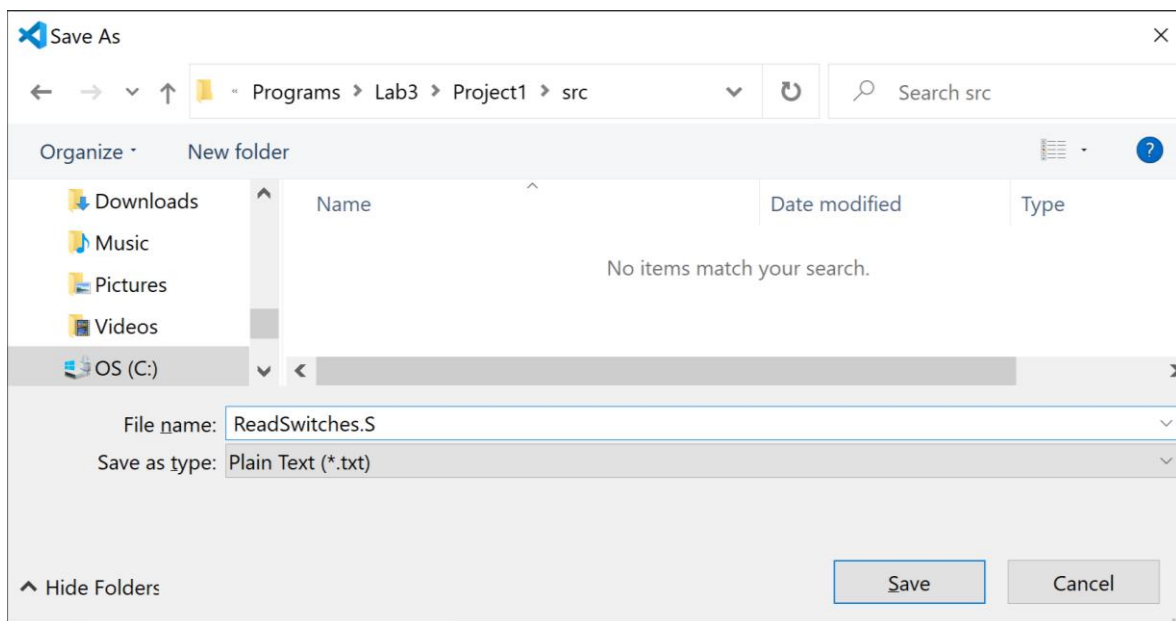




Figura 9. Guardado de archivo como ReadSwitches.S

Paso 3. Descargar RVfpgaNexys en la placa FPGA Nexys A7

Ahora se procederá a la descarga de RVfpgaNexys en la placa FPGA Nexys A7. Siga las instrucciones para descargar RVfpgaNexys como se describe en la Guía de inicio y en la práctica 2, que se repiten aquí para mayor comodidad.

Descargue RVfpgaNexys en la placa Nexys A7 haciendo clic en el icono de PlatformIO en la barra de menú de la izquierda , luego expanda Project Tasks → env:swervolf_nexys → Platform y haga clic en Upload Bitstream.

Como alternativa, puede descargar RVfpgaNexys desde una ventana de terminal de PlatformIO haciendo clic en el botón PlatformIO: New Terminal () en la parte inferior de la ventana PlatformIO, y luego escribiendo (o copiando) lo siguiente en el terminal de PlatformIO:

```
pio run -t program_fpga
```

Paso 4. Compilar, descargar y ejecutar el programa de ensamblador de RISC-V

Ahora que RVfpgaNexys está cargado en la placa, debe compilar el programa, descargarlo en RVfpgaNexys y ejecutar/depurar el programa. Si VSCode no está ya abierto, ábralo. Su último proyecto, Project1, debería abrirse automáticamente. Si no, asegúrese de que la extensión de PlatformIO esté abierta y haga clic en File → Open Folder y seleccione (pero no abra) Project1, que creó anteriormente en esta práctica.

Haga clic en el botón Run en la barra del menú de la izquierda y luego haga clic en el botón Iniciar depuración (ver Figura 10).

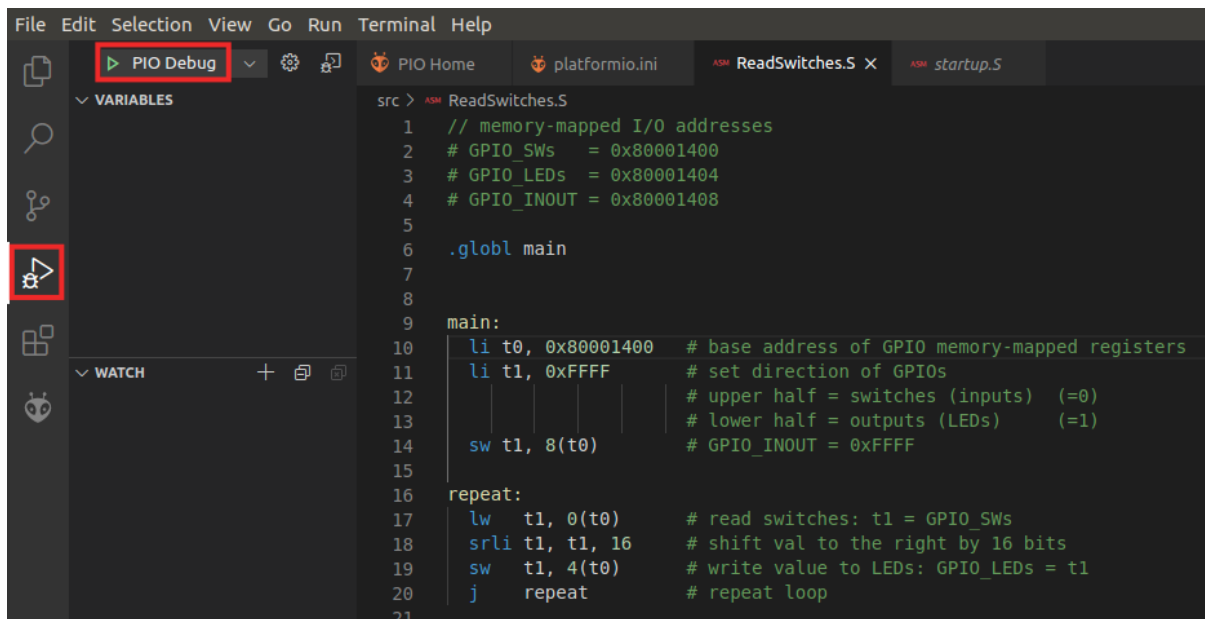


Figura 10. Ejecución del programa en RVfpgaNexys

El programa se descargará en RVfpgaNexys, que se ejecuta en la FPGA de la placa Nexys A7. Ahora puede empezar a ejecutar y depurar el programa (ver Figura 11).

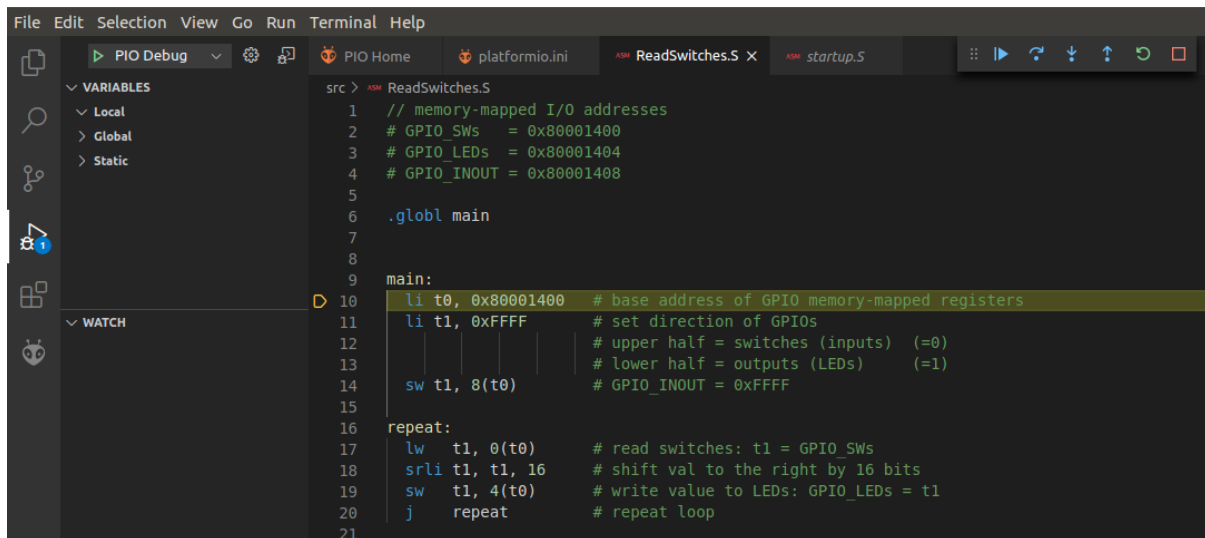



Figura 11. Programa ejecutándose en RVfpgaNexys

Como se describe en la Guía de inicio de RVfpga y en la Práctica 2, utilice la barra de herramientas de depuración y las opciones del Depurador para ejecutar y gestionar el programa. Por ejemplo, puede establecer un punto de interrupción en la línea 17 (haciendo clic justo a la izquierda del número de línea) y luego visualizar el registro `t1`, que es en el que se carga el valor de los interruptores. Cuando se detiene la sesión de depuración

pulsando el botón Stop  (o Shift - F5), la sesión de depuración termina, pero el programa sigue ejecutándose en RVfpgaNexys.

4. Ejercicios

Ahora creará sus propios programas en lenguaje ensamblador de RISC-V realizando los mismos ejercicios que en la Práctica 2, pero esta vez en ensamblador RISC-V en lugar de C. Los enunciados de los ejercicios se repiten a continuación para su comodidad.

Recuerde que si deja la placa Nexys A7 conectada a su computadora y encendida, no necesita volver a cargar RVfpgaNexys en la placa entre la ejecución de diferentes programas. Sin embargo, si apaga la placa Nexys A7, es necesario recargar RVfpgaNexys en la placa usando PlatformIO.

Recuerde que también se pueden ejecutar los programas en simulación, tanto en Verilator como en Whisper.

Ejercicio 1. Escriba un programa en ensamblador de RISC-V que proyecte el valor de los interruptores en los LEDs. Ese valor se mantiene durante un tiempo tras el cual los LEDs se apagan, y se repite el proceso. El valor debe encenderse y apagarse lo suficientemente lento como para que una persona pueda ver el parpadeo. Guarde el programa como **FlashSwitchesToLEDs.S**.

Ejercicio 2. Escriba un programa en ensamblador de RISC-V que muestre el valor inverso de los interruptores en los LEDs. Por ejemplo, si los interruptores son (en binario): 01010101010101, entonces los LEDs deberían mostrar: 1010101010101010; si los interruptores son: 1111000011110000, entonces en los LEDs debería aparecer: 0000111100001111; y así sucesivamente. Use **DisplayInverse.S** como nombre para el programa.

Ejercicio 3. Escriba un programa en ensamblador de RISC-V que desplace un número creciente de LEDs encendidos hacia adelante y hacia atrás hasta que todos los LEDs estén encendidos. Entonces el patrón debe repetirse. Guarde el programa como **ScrollLEDs.S**.

El programa debería realizar lo siguiente:

1. Primero, un LED encendido debe desplazarse de derecha a izquierda y luego de izquierda a derecha.
2. A continuación dos LEDs encendidos deben desplazarse de derecha a izquierda y luego de izquierda a derecha.
3. Después tres LEDs encendidos deben desplazarse de derecha a izquierda y luego de izquierda a derecha.
4. Y así sucesivamente, hasta que todos los LEDs se encienden.
5. A continuación el patrón debe repetirse.

Ejercicio 4. Escriba un programa en ensamblador de RISC-V que muestre el resultado de la suma (4 bits sin signo) de los 4 bits menos significativos de los interruptores con los 4 bits más significativos de los mismos. El resultado se debe mostrar en los 4 bits menos significativos (más a la derecha) de los LEDs. Guarde el programa como **4bitAdd.S**. El quinto bit de los LEDs debería encenderse cuando se produzca un desborde sin signo (es decir, cuando el acarreo de salida es 1).

Ejercicio 5. Escriba un programa en ensamblador de RISC-V que encuentre el *máximo común divisor* de dos números, *a* y *b*, según el algoritmo de Euclides. Los valores *a* y *b* deben ser variables definidas estáticamente en el programa. Guarde el programa como **GCD.S**. En el siguiente enlace se proporciona información adicional sobre el algoritmo de

Euclides: <https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm>. También puede simplemente buscar en Google "Algoritmo de Euclides".

Ejercicio 6. Escriba un programa en ensamblador de RISC-V que calcule los primeros 12 números de la serie de Fibonacci y almacene el resultado en un vector finito (es decir, un array), V , de longitud 12. La secuencia infinita de números de Fibonacci se define del siguiente modo:

$$V(0)=0, \quad V(1)=1, \quad V(i)=V(i-1)+V(i-2) \quad (\text{donde } i=0,1,2,\dots)$$

Expresado en palabras, el número de Fibonacci correspondiente al elemento i es la suma de los dos números previos de la serie. Tabla 6 muestra los números de Fibonacci para $i = 0$ hasta 8.

Tabla 6. Serie Fibonacci

i	0	1	2	3	4	5	6	7	8
V	0	1	1	2	3	5	8	13	21

La dimensión del vector, N , debe ser definida en el programa como una constante. Guarde el programa como **Fibonacci.S**.

Ejercicio 7. Dado un vector de N elementos (es decir, un array), A , genere otro vector, B , de manera que B sólo contenga aquellos elementos de A que son números pares mayores que 0. Por ejemplo: supongamos que $N=12$ y $A = [0,1,2,7,-8,4,5,12,11,-2,6,3]$, entonces B sería: $B = [2,4,12,6]$. Guarde el programa como **EvenPositiveNumbers.c**.

Ejercicio 8. Dados dos vectores de N elementos (es decir, arrays), A y B , crear otro vector, C , definido como:

$$C(i) = |A[i] + B[N-i-1]|, \quad i = 0, \dots, N-1.$$

Escriba un programa en ensamblador de RISC-V que calcule el nuevo vector. Utilice arrays de 12 elementos en su programa. Guarde el programa como **AddVectors.S**.

Ejercicio 9. Implementar el algoritmo de ordenación burbuja (*bubble sort*) en C. Este algoritmo ordena los elementos de un vector en orden ascendente mediante el siguiente procedimiento:

1. Recorrer el vector repetidamente hasta terminar.
2. Intercambiar cualquier par de componentes adyacentes si $V(i) > V(i+1)$.
3. El algoritmo se detiene cuando cada par de componentes consecutivos está ordenado.

Use arrays de 12 elementos para probar su programa. Guarde el programa como **BubbleSort.S**.

Ejercicio 10. Escriba un programa en ensamblador de RISC-V que calcule el factorial de un número no negativo, n , mediante multiplicaciones iterativas. Aunque debe probar el correcto funcionamiento de su programa para múltiples valores de n , la entrega final del mismo debe ser para $n = 7$. El programa debe imprimir el valor del factorial(n) al final del programa. n debe ser una variable definida estáticamente dentro del programa. Guarde el programa como **Factorial.c**.