

Universidad de Costa Rica

Facultad de Ingeniería

Escuela de Ingeniería Eléctrica

IE0424 - Laboratorio de Circuitos Digitales

I ciclo 2025

## Reporte de Laboratorio #3

Gabriel Siles Chaves C17530

Jorge Loría Chaves C04406

Grupo 01

Profesor:

Marco Villalta Fallas

# Índice

<b>1. Resumen</b>	<b>2</b>
<b>2. Práctica 9: Entrada/salida dirigida por interrupciones [1]</b>	<b>2</b>
2.1. Ejercicio 1: . . . . .	2
2.1.1. Descripción del problema . . . . .	2
2.1.2. Desarrollo de la solución . . . . .	2
2.1.3. Diseño Final . . . . .	4
2.2. Ejercicio 2: . . . . .	4
2.2.1. Descripción del problema . . . . .	4
2.2.2. Desarrollo de la solución . . . . .	5
2.2.3. Diseño Final . . . . .	6
2.3. Ejercicio 3: . . . . .	7
2.3.1. Descripción del problema . . . . .	7
2.3.2. Desarrollo de la solución . . . . .	8
2.3.3. Diseño Final . . . . .	9
<b>3. Práctica 10: Buses en serie [2]</b>	<b>9</b>
3.1. Ejercicio 1: . . . . .	9
3.1.1. Descripción del problema . . . . .	9
3.1.2. Desarrollo de la solución . . . . .	9
3.1.3. Diseño Final . . . . .	12
<b>4. Complicaciones</b>	<b>12</b>
<b>5. Conclusiones y recomendaciones</b>	<b>12</b>

# 1. Resumen

En la primera parte del laboratorio 9, se exploraron las capacidades del sistema para gestionar interrupciones externas provenientes de diferentes módulos periféricos. Se implementaron rutinas de atención a interrupciones (ISR) tanto para el temporizador (PTC) como para los módulos GPIO encargados de detectar acciones del usuario a través de interruptores y botones. Además, se estudió el uso de prioridades y el manejo de múltiples fuentes de interrupción, permitiendo responder de forma eficiente.

En la segunda parte (Lab 10), se trabajó con la interfaz SPI integrada en el sistema, con el objetivo de establecer comunicación entre el procesador SweRV EH1 y el acelerómetro ADXL362. Se implementó un programa en ensamblador RISC-V que permite leer los datos de aceleración en los tres ejes y desplegarlos en los displays de 7 segmentos, aprovechando subrutinas para el control del SPI y la selección de dispositivos. Para facilitar su revisión, todos los archivos se encuentran disponibles en el repositorio de GitHub en el siguiente enlace: <https://git.ucr.ac.cr/GABRIEL.SILES/ie0424-gj>.

## 2. Práctica 9: Entrada/salida dirigida por interrupciones [1]

### 2.1. Ejercicio 1:

#### 2.1.1. Descripción del problema

El objetivo de este ejercicio es modificar el programa `LED-Switch_7SegDispl_Interrupts_C-Lang` para incorporar una segunda fuente de interrupción, esta vez generada por el temporizador del sistema, también conocido como unidad PTC (Pulse Timer/Counter). Esta modificación busca ampliar el manejo de interrupciones externas en el sistema RVfpga.

En particular, se debe habilitar la interrupción generada por el temporizador conectada a la línea `IRQ3`, lo cual se logra activando el bit correspondiente (`irq_ptc_enable`) en la dirección de memoria `0x80001018`. Para ello, se requiere implementar una función de inicialización de interrupciones del PTC, análoga a la función `GPIO_Initialization` del programa original.

Asimismo, se debe desarrollar una segunda rutina de atención a interrupciones (ISR) denominada `PTC_ISR`, similar en estructura y comportamiento a la `GPIO_ISR`, pero destinada exclusivamente al manejo y limpieza de la interrupción del temporizador.

Una vez implementado y verificado el programa, se podrán utilizar funciones del PSP como `pspExtInterruptsSetThreshold()` y `pspExtInterruptSetPriority(interrupt_source, priority)` para analizar combinaciones de prioridades y umbrales. Estas funciones permiten modificar las prioridades dinámicamente durante la ejecución, posibilitando por ejemplo que los displays de 7 segmentos cuenten hasta cierto número y se detengan al modificar la prioridad de la interrupción correspondiente.

#### 2.1.2. Desarrollo de la solución

La idea del ejercicio es generar la nueva fuente de interrupción llamada `PTC_ISR`, por lo que la definimos de la siguiente forma:

```

1 // Se agrega el modulo de PTC_ISR
2 void PTC_ISR(void)
3 {
4     M_PSP_WRITE_REGISTER_32(SegDig_ADDR, SegDisplCount);
5     SegDisplCount++;
6     M_PSP_WRITE_REGISTER_32(RPTC_CNTR, 0x0);
7     M_PSP_WRITE_REGISTER_32(RPTC_CTRL, 0x31); //Limpiamos el GPIO
8
9     bspClearExtInterrupt(3); //Detenemos la generacion de una interrupcion externa
10 }

```

Figura 1: Definición del módulo de textttPTC\_ISR

Seguidamente, es necesario generar una función de inicialización de nuevo módulo de PTC, por lo que se sigue de forma general la lógica de como se implementó en `GPIO_Initialization`:

```

1
2 // Siguiendo la logica de GPIO_Initialization
3 void PTC_Initialization(void)
4 {
5     M_PSP_WRITE_REGISTER_32(RPTC_LRC, 0xFFFF); //Configuramos el LRC
6     ////Configuracion del counter y control
7     M_PSP_WRITE_REGISTER_32(RPTC_CNTR, 0x0);
8     M_PSP_WRITE_REGISTER_32(RPTC_CTRL, 0x21);
9 }

```

Figura 2: Definición de PTC\_Initialization

Finalmente realizamos las modificaciones correspondientes en el main, en el cuál se hace la inicialización de la línea de interrupción IRQ3 y va a ser ocupada por la función `PTC_ISR`. De igual forma, se hace un llamado a la función de inicialización anteriormente creada para así poder configurar el módulo PTC y que al entrar al while pueda ejecutar correctamente las interrupciones:

```

1 int main(void){
2     int count=0, i;
3
4     /* INITIALIZE THE INTERRUPT SYSTEM */
5     DefaultInitialization(); /* Default initialization
6     */
7     pspExtInterruptsSetThreshold(5); /* Set interrupts
8     threshold to 5 */
9
10    /* INITIALIZE INTERRUPT LINE IRQ4 */
11    ExternalIntLine_Initialization(4, 6, GPIO_ISR); /* Initialize line IRQ4
12    with a priority of 6. Set GPIO_ISR as the Interrupt Service Routine */
13    ExternalIntLine_Initialization(3, 6, PTC_ISR); /* Initialize line IRQ3
14    for PTC */

```

```

11  M_PSP_WRITE_REGISTER_32(Select_INT, 0x3);          /* Connect the GPIO and
    PTC interrupt to the IRQ lines */
12
13  /* INITIALIZE THE PERIPHERALS */
14  GPIO_Initialization();                             /* Initialize the GPIO */
15  PTC_Initialization();                             /* Initialize the timer
    peripheral (PTC) */
16  M_PSP_WRITE_REGISTER_32(SegEn_ADDR, 0x0);         /* Initialize the 7-Seg
    Displays */
17
18  /* ENABLE INTERRUPTS */
19  pspInterruptsEnable();                             /* Enable all interrupts
    in mstatus CSR */
20  M_PSP_SET_CSR(D_PSP_MIE_NUM, D_PSP_MIE_MEIE_MASK); /* Enable external
    interrupts in mie CSR */
21
22  while (1) {
23      if (SegDisplCount == 0xFF) {
24          pspExtInterruptSetPriority(3, 0);
25          pspExtInterruptsSetThreshold(1);
26      }
27  }
28 }

```

Listing 1: Función `main()` con configuración de interrupciones y periféricos

### 2.1.3. Diseño Final

Como resultado final ya con la implementación del código se puede observar en la Figura 3 como se construye correctamente el debug mode generando que se pueda realizar la prueba de la interrupción. En el este caso se utilizó el valor de `0xFF5` como `SegDisplCount` y que en el momento de llegar a este valor se realizara la interrupción, teniendo en consideración la prioridad. Se puede observar el video de la demostración en el siguiente link: <https://youtu.be/2hGSN7sJEtg?si=cfoLhNNzitOs69hY>

```

Linking .pio/build/swervolf_nexys/firmware.elf
Generating disassembly
riscv64-unknown-elf-objdump -d ".pio/build/swervolf_nexys/firmware.elf" > ".pio/build/swervolf_nexys/firmware.dis"
Checking size .pio/build/swervolf_nexys/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM: [ ] 1.1% (used 13464 bytes from 1216512 bytes)
Flash: [ ] 0.0% (used 3524 bytes from 16777216 bytes)
Building .pio/build/swervolf_nexys/firmware.bin
===== [SUCCESS] Took 0.45 seconds =====

```

Figura 3: Debug mode del ejercicio 1

## 2.2. Ejercicio 2:

### 2.2.1. Descripción del problema

En este ejercicio se busca modificar el diseño `RVfpgaNexys` para incluir una tercera fuente de interrupción, esta vez proveniente del segundo módulo GPIO (`GPIO2`) diseñado previamente para controlar los botones integrados en la placa (`BTNU`, `BTND`, `BTNL`, `BTNR`, `BTNC`). Esta extensión permite detectar eventos de los botones mediante interrupciones, en lugar de emplear sondeo (polling).

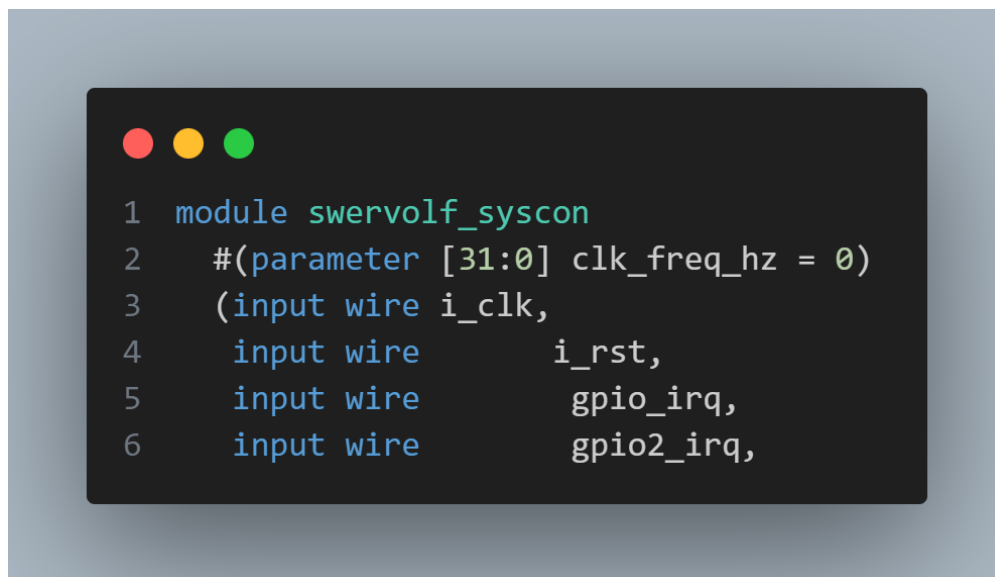
Existen dos enfoques posibles: usar una línea de interrupción externa no utilizada del SweRV EH1 (requiriendo modificación de las bibliotecas de WD) o reutilizar la línea IRQ4 para conectar tanto el módulo GPIO original como GPIO2, aprovechando el modo de interrupción de vector único. Este último enfoque es más simple y compatible con la BSP provista.

Como parte de este ejercicio, el estudiante debe modificar la lógica en Verilog del módulo `swervolf_core` para permitir que IRQ4 también reciba interrupciones del módulo GPIO2. Para ello, se debe ajustar la asignación de señales y utilizar los registros de control mapeados en memoria, como `0x80001018`, que permite habilitar o redirigir las fuentes de interrupción.

Esta tarea fomenta la comprensión de cómo múltiples fuentes de interrupción pueden compartir una línea IRQ y cómo configurar un sistema embebido para gestionarlas adecuadamente, incluyendo su identificación dentro de una rutina común de atención a interrupciones (ISR).

### 2.2.2. Desarrollo de la solución

Partiendo del segundo módulo GPIO desarrollado en la práctica del Laboratorio 6, en este ejercicio se opta por reutilizar la línea de interrupción IRQ4, en lugar de asignar una nueva entre las 255 disponibles en la Nexys. Dicha línea ya era utilizada por el módulo GPIO1, encargado del control de los *LEDs* y *Switches*. Para permitir que ambas fuentes de interrupción compartan esta línea, se implementa una compuerta lógica OR que unifica las señales de interrupción y las direcciona hacia el módulo `swervolf_core`.

A screenshot of a code editor with a dark background and light-colored text. The code is Verilog, defining a module named `swervolf_syscon`. It includes a parameter `clk_freq_hz` and declares three input wires: `i_clk`, `i_rst`, `gpio_irq`, and `gpio2_irq`. The code is numbered from 1 to 6.

```
1 module swervolf_syscon
2   #(parameter [31:0] clk_freq_hz = 0)
3   (input wire i_clk,
4     input wire      i_rst,
5     input wire      gpio_irq,
6     input wire      gpio2_irq,
```

Figura 4: Declaración del wire en `swervolf_syscon`

```

1 // GPIO Interrupt through IRQ4 and IRQ3. Enable by setting bit 0 of word 0x80001018
2 if (irq_gpio_enable & (gpio_irq | gpio2_irq)) begin
3     sw_irq4 <= 1'b1;
4 end
5

```

Figura 5: Asignación del OR

Finalmente se instancia en `swervolf_core.v`.

```

1 swervolf_syscon
2     #(.clk_freq_hz (clk_freq_hz))
3     syscon
4     (.i_clk          (clk),
5      .i_rst          (wb_rst),
6      .gpio_irq       (gpio_irq),
7      .gpio2_irq      (gpio2_irq),

```

Figura 6: Instancia de GPIO2\_IRQ

### 2.2.3. Diseño Final

El diseño final se implementó y sintetizó en Vivado, integrando correctamente el segundo módulo GPIO (GPIO2) a través de la línea de interrupción compartida IRQ4 y conectando al swervolf. En la Figura 7 se muestra el resultado exitoso del proceso de síntesis, mientras que en la Figura 8 se observa la implementación final, confirmando la integración funcional del diseño propuesto.

Name	Constraints	Status	TNS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
✓ synth_1	constrs_1	synth_design Complete!				3520	197	44	0	4	5/21/25, 00:02:45	
✓ impl_1	constrs_1	write_bitstream Complete!	0 0.00C	0 0.936	0 1	3466	197	44	0	4	5/21/25, 00:03:46	

Figura 7: Síntesis en Vivado

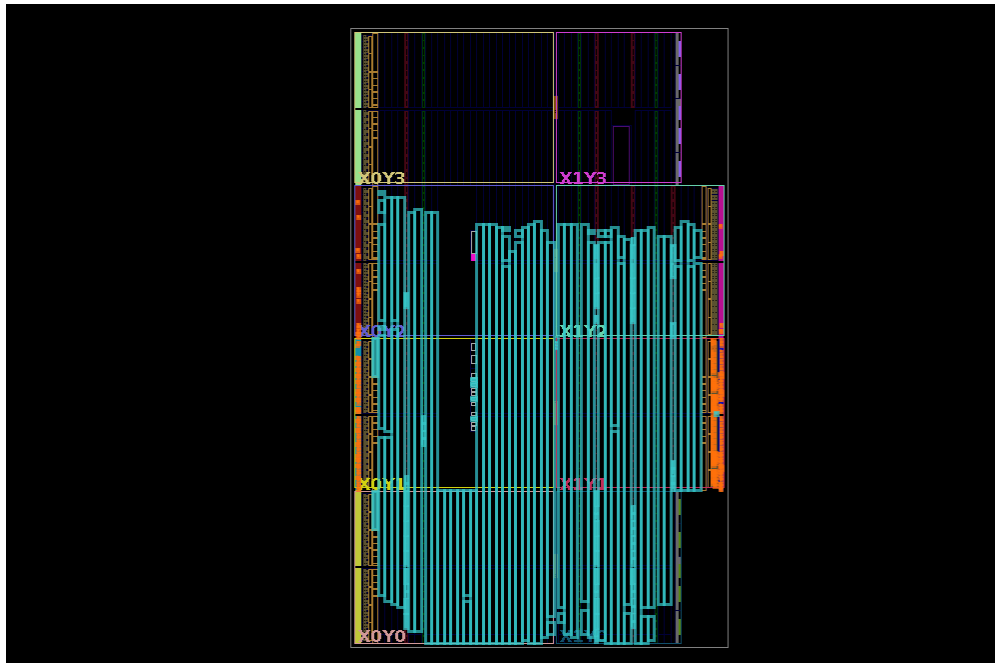


Figura 8: Implementación en Vivado

## 2.3. Ejercicio 3:

### 2.3.1. Descripción del problema

Este ejercicio consiste en desarrollar un programa en lenguaje C que aproveche el diseño extendido del sistema `RVfpgaNexys`, con tres fuentes de interrupción activas: GPIO (interruptores), GPIO2 (botones) y PTC (temporizador). El programa debe mostrar una cuenta binaria creciente en los LEDs, comenzando desde 1.

Para mejorar la interacción, se implementa un retardo entre cada incremento usando el temporizador (PTC) y su respectiva interrupción. Adicionalmente, el botón central (BTNC) debe permitir cambiar la velocidad del conteo, y el interruptor `Switch[0]` debe reiniciar la cuenta a su valor inicial.

Debido a que las interrupciones de los módulos GPIO y GPIO2 comparten la línea `IRQ4`, la rutina de atención a interrupciones debe identificar de cuál de los dos módulos proviene el evento. Esta identificación puede realizarse mediante la lectura de registros internos del GPIO.



Este ejercicio integra programación de interrupciones múltiples, manipulación de periféricos y control del flujo del programa en tiempo real, siendo un ejemplo representativo de sistemas embebidos reactivos.

### 2.3.2. Desarrollo de la solución

Inicialmente, se trabajó con una versión preliminar del código que realizaba el conteo dentro del bucle principal utilizando retardos por software y lectura directa de entradas a través de instrucciones `READ_GPIO`. Para mejorar el diseño, se reorganizó el sistema en torno al uso de interrupciones. El temporizador PTC fue configurado para generar interrupciones periódicas. Esto permitió trasladar la lógica de incremento del contador a una rutina de servicio dedicada (`PTC_ISR`), lo que liberó al procesador del retardo activo y optimizó el uso de recursos. La inicialización del temporizador se realizó con valores programables del registro `LRC`, facilitando el control dinámico de la frecuencia de interrupción.

```
1 void PTC_ISR(void)
2 {
3     count++;
4     M_PSP_WRITE_REGISTER_32(GPIO_LEDs, count);
5
6     M_PSP_WRITE_REGISTER_32(RPTC_LRC, delay);
7     M_PSP_WRITE_REGISTER_32(RPTC_CNTR, 0x0);
8     M_PSP_WRITE_REGISTER_32(RPTC_CTRL, 0x40);
9     M_PSP_WRITE_REGISTER_32(RPTC_CTRL, 0x31);
10
11     bspClearExtInterrupt(3);
12 }
```

Listing 2: Fragmento de `PTC_ISR` para actualizar los LEDs

Luego, se extendió el sistema mediante la incorporación del módulo `GPIO2`, destinado a la gestión de los botones. Esto permitió separar físicamente los eventos generados por los botones de aquellos producidos por los switches. Se configuraron ambos módulos `GPIO` para generar interrupciones en la línea compartida `IRQ4`, y se diseñó una rutina de servicio común (`GPIO_ISR`) que identifica el origen de la interrupción mediante la lectura de los registros `RGPIO_INTS` y `RGPIO2_INTS`.

En esta rutina se implementó la lógica de interacción del usuario: el botón `BTNC` alterna entre dos valores predefinidos para `delay`, permitiendo modificar la velocidad del conteo; mientras que el interruptor `Switch[0]` reinicia el valor del contador y limpia los LEDs. La identificación del evento se realiza mediante operaciones de enmascaramiento bit a bit, tal como se observa en el siguiente fragmento:

```
1 void GPIO_ISR(void)
2 {
3     u32_t gpio_status = M_PSP_READ_REGISTER_32(RGPIO_INTS);
4     u32_t btn_status  = M_PSP_READ_REGISTER_32(RGPIO2_INTS);
5
6     if ((gpio_status >> 16) & 0x1) {
7         count = 0;
8         M_PSP_WRITE_REGISTER_32(GPIO_LEDs, 0);
9         M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0);
10    }
11
12    if (btn_status & 0x1) {
13        delay = (delay == 0x5FFFF) ? 0x5FFFFFF : 0x5FFFF;
```

```

14     M_PSP_WRITE_REGISTER_32(RGPIO2_INTS, 0x0);
15 }
16
17     bspClearExtInterrupt(4);
18 }

```

Listing 3: Identificación de fuente en GPIO\_ISR

Finalmente, se modularizó el código utilizando funciones específicas de inicialización para cada periférico, como `GPIO_Initialization`, `GPIO2_Initialization` y `PTC_Initialization`, con el objetivo de mejorar la organización y la reutilización del código.

### 2.3.3. Diseño Final

Con la síntesis, el diseño e implementación del ejercicio 2, se logró establecer la base funcional necesaria para desarrollar el ejercicio 3. La integración de las interrupciones provenientes de los módulos GPIO, GPIO2 y PTC permitió el desarrollo de un sistema de conteo binario controlado por interrupciones. Esta estructura, permite la utilización del temporizador (PTC) y la capacidad de gestión de eventos por interrupciones. Se puede observar el video de la demostración en el siguiente link: <https://youtu.be/1l7Dhu-7Gb8?si=AxBsJ3foSH-NtZDp>

## 3. Práctica 10: Buses en serie [2]

### 3.1. Ejercicio 1:

#### 3.1.1. Descripción del problema

En este ejercicio se propone implementar un programa en lenguaje ensamblador RISC-V que permita leer los 8 bits más significativos de los datos de aceleración en los ejes X, Y y Z, utilizando el acelerómetro ADXL362 integrado en la placa Nexys A7. La información recolectada se debe visualizar en los displays de 7 segmentos disponibles en la FPGA, permitiendo observar en tiempo real el comportamiento del sensor.

Para llevar a cabo esta tarea, se debe hacer uso de la interfaz SPI del sistema RVfpga, configurando adecuadamente el controlador SPI y empleando las rutinas de apoyo proporcionadas: `spiInit`, `spiCSDown`, `spiCSUp` y `spiSendGetData`. Estas subrutinas permiten inicializar el módulo SPI, controlar la señal de selección de chip (CS) y realizar la comunicación bidireccional con el periférico.

El ejercicio permite afianzar conceptos como la comunicación serial síncrona mediante SPI, la manipulación directa de registros de control y estado del periférico, así como la programación de dispositivos embebidos en lenguaje ensamblador. Además, sienta las bases para futuras aplicaciones con sensores más complejos o múltiples periféricos SPI.

#### 3.1.2. Desarrollo de la solución

El desarrollo de este ejercicio comenzó a partir del objetivo de crear un programa en lenguaje ensamblador para la arquitectura RISC-V que leyera los 8 bits más significativos de los datos de aceleración en los ejes X, Y y Z, y los desplegara en los 8 dígitos del display de 7 segmentos. Para ello, se utilizó un acelerómetro conectado mediante el módulo SPI, cuya interacción se gestiona utilizando un conjunto de subrutinas provistas por el laboratorio.

En la fase inicial, se analizó el comportamiento del módulo SPI, así como el contenido de los registros relacionados: **SPCR**, **SPSR**, **SPDR**, **SPER** y **SPCS**. Además, se estudió el propósito y funcionamiento de las subrutinas auxiliares proporcionadas, como **spiInit**, **spiCSUp**, **spiCSDown** y **spiSendGetData**. Estas funciones permiten configurar el periférico, controlar la línea **CS** y realizar transmisiones SPI de forma modular.

La primera tarea consistió en inicializar correctamente el periférico SPI. Esto se logró mediante la invocación de la subrutina **spiInit**, que configura el registro de control y el registro de extensión del módulo SPI con valores adecuados para habilitar la comunicación con el acelerómetro.

```

1 spiInit:
2     li t1, SPCR
3     li t0, 0x53
4     sb t0, 0(t1)
5     li t1, SPER
6     li t0, 0x02
7     sb t0, 0(t1)
8 ret

```

Listing 4: Inicialización del módulo SPI

Una vez habilitado el módulo SPI, se implementó la secuencia de lectura para los tres ejes. Para cada eje, se realiza un intercambio SPI que comienza levantando la línea **CS** con **spiCSUp**, enviando el comando de lectura (**0x0B**), especificando la dirección del registro deseado (por ejemplo, **0x08** para el eje **X**), y luego realizando múltiples lecturas hasta recibir los datos válidos. Finalmente, se baja la línea **CS** con **spiCSDown**. Este procedimiento se repite para los ejes **Y** y **Z**, accediendo a las direcciones correspondientes en el periférico.

A continuación, los valores obtenidos se almacenan en registros auxiliares (**a2**, **a3**, **a4**) y se extraen los 8 bits menos significativos de cada muestra aplicando una operación **AND** con la máscara **0xFF**. Cada valor se desplaza a una posición distinta para ser mostrado en los 8 dígitos del display de 7 segmentos.

```

1 # Extrae los 8 bits menos significativos en a4
2     li t0, 0xFF
3     and a4, a1, t0
4
5 # ===== WORD DE 7 SEGMENTOS =====
6 # FORMATO [a2][a3][a4]
7
8 # Se desplaza valor de X
9     slli a2, a2, 24
10 # Se desplaza valor de Y
11     slli a3, a3, 12
12
13 # Se suma valores
14     add a2, a2, a3
15     add a2, a2, a4
16
17     li t0, DISPLAY_SEG
18     sw a2, 0(t0)
19
20     jal delay
21     j RefreshValues

```

Listing 5: Cálculo y despliegue de la palabra en el display de 7 segmentos

Para controlar la tasa de actualización, se incluyó una rutina de retardo por software (**delay**) que introduce una pausa perceptible entre cada ciclo de lectura y escritura. De esta manera, se asegura que los datos mostrados en el display sean estables y legibles.

```

1 delay:
2     addi t2, t2, 1
3     blt t2, s0, delay
4     jr ra

```

Listing 6: Rutina de retardo

Finalmente, todo el proceso se encapsuló en un bucle infinito llamado **ResfreshValues**, donde se repiten las lecturas de los tres ejes y la actualización del display. Esta estructura garantiza una visualización continua y en tiempo real de los datos de aceleración del sensor.

```

1 ResfreshValues:
2
3 # ===== EJE X =====
4 # INICIO Comunicaci n SPI
5     jal spiCSUp
6 # Se lee datos en X
7     li a0, 0x0B
8     jal spiSendGetData
9 # se obtienen datos en X
10    li a0, 0x08
11    jal spiSendGetData
12    jal spiSendGetData
13 # Se carga valor de relleno
14    li a0, 0xFF
15    jal spiSendGetData
16 # FIN comunicacion SPI
17    jal spiCSDown
18
19 # Reinicia el contador
20    li t2, 0x00
21    jal delay
22
23 # Extrae los 8 bits menos significativos en a2
24    li t0, 0xFF
25    and a2, a1, t0
26
27 # ===== EJE Y =====
28
29 # INICIO Comunicacion SPI
30    jal spiCSUp
31 # Se lee datos en Y
32    li a0, 0x0B
33    jal spiSendGetData
34 # se obtienen datos en Y
35    li a0, 0x09
36    jal spiSendGetData
37    jal spiSendGetData
38 # Se carga valor de relleno
39    li a0, 0xFF
40    jal spiSendGetData
41 # FIN comunicacion SPI
42    jal spiCSDown

```

```

43
44 # Reinicia el contador
45     li t2, 0
46     jal delay

```

Listing 7: Bucle RefreshVlaues

### 3.1.3. Diseño Final

Como ejercicio final, se analizó el funcionamiento del módulo SPI y los registros asociados, como SPCR, SPSR y SPDR. Se implementaron las subrutinas auxiliares como `spiInit`, `spiCSUp`, `spiCSDown` y `spiSendGetData`. Con esto se estableció una secuencia de lectura para los tres ejes (X, Y, Z) del acelerómetro. Para cada eje, se levantó la línea CS, se envió el comando de lectura, se obtuvieron los datos y finalmente se bajó la línea CS. Se puede observar el video de la demostración en el siguiente link: <https://youtu.be/7eKSI6td02c?si=4OGmJSzVmoIc1Hrz>

## 4. Complicaciones

Las principales complicaciones en el diseño fueron las siguientes:

- **Prioridades de interrupciones:** Al reutilizar la línea de interrupción IRQ4 para los módulos GPIO1 y GPIO2, fue necesario implementar una lógica para identificar correctamente la fuente de cada interrupción.
- **Conexión SPI para el acelerómetro:** Se presentó la dificultad de configurar correctamente el módulo SPI para la comunicación con el acelerómetro. Se tuvo que agregar 2 veces la función de `spiSendGetData` para que funcionara correctamente para obtener los datos

## 5. Conclusiones y recomendaciones

- Se trabajó en la estructura del sistema de interrupciones de RVfpga, por medio del ejercicio 1 de la guía 9 se pudo comprender cómo funcionan las interrupciones y cómo se gestionan.
- Se implementó un sistema de interrupciones utilizando código en C, lo que permitió desarrollar la habilidad de escribir y configurar rutinas de servicio de interrupción (ISR) y gestionar eventos en tiempo real en un sistema embebido.
- Se introdujo el protocolo SPI, trabajando en la configuración de registros y la implementación de la comunicación con periféricos como el acelerómetro. Esto facilitó la familiarización con la configuración y control de dispositivos mediante SPI.
- Como recomendación siempre es importante leer el funcionamiento de los módulos y elementos a la hora de querer implementar a la FPGA proyectos para poder facilitar la implementación.

## Referencias

- [1] Imagination University Programme, *RVfpga Lab 9: Interrupt-Driven I/O*, Version 2.2, Available from <https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>, Imagination Technologies, mayo de 2022.
- [2] Imagination University Programme, *RVfpga Lab 10: Serial Buses*, Version 2.2, Available from <https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>, Imagination Technologies, mayo de 2022.