Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE0424 - Laboratorio de Circuitos Digitales
I ciclo 2025

Reporte de Laboratorio #1

Gabriel Siles Chaves C17530
Jorge Loría Chaves C04406

Grupo 01

Profesor:
Marco Villalta Fallas

Índice

1. Introducción **2**

2. Práctica 2 RVfpga: Programación en C **2**
2.1. Ejercicio 1: FlashSwitchesToLEDs . . . 2
2.1.1. Descripción del problema [1] . . . 2
2.1.2. Desarrollo de la solución . . . 2
2.1.3. Diseño Final . . . 3
2.2. Ejercicio 3: ScrollLEDs . . . 4
2.2.1. Descripción del problema [1] . . . 4
2.2.2. Desarrollo de la solución . . . 5
2.2.3. Diseño Final . . . 6

3. Práctica 2 RVfpga: Lenguaje ensamblador de RISC-V **7**
3.1. Ejercicio 2: DisplayInverse . . . 7
3.1.1. Descripción del problema [2] . . . 7
3.1.2. Desarrollo de la solución . . . 8
3.1.3. Diseño Final . . . 9
3.2. Ejercicio 4: 4bitAdder . . . 10
3.2.1. Descripción del problema [2] . . . 10
3.2.2. Desarrollo de la solución . . . 10
3.2.3. Diseño Final . . . 11

4. Práctica 3 RVfpga: Llamadas a función **13**
4.1. Ejercicio 7: Triplets . . . 13
4.1.1. Descripción del problema [3] . . . 13
4.1.2. Desarrollo de la solución . . . 14
4.1.3. Diseño Final . . . 15
4.2. Ejercicio 8: Filters . . . 18
4.2.1. Descripción del problema [3] . . . 18
4.2.2. Desarrollo de la solución . . . 18
4.2.3. Diseño Final . . . 20

5. Problemas y dificultades **20**

6. Conclusiones y recomendaciones **21**

1. Introducción

Con el propósito de aplicar los conocimientos adquiridos sobre la tarjeta *RVfpga Nexys 4*, se seleccionaron diversos ejercicios pertenecientes a las prácticas 2, 3 y 4 del Laboratorio 1 del curso. Dichos ejercicios implicaron la implementación de múltiples codificaciones, tanto en lenguaje ensamblador *RISC-V* como en lenguaje *C*. Cada actividad fue desarrollada y registrada en ramas específicas dentro de un repositorio *Git* compartido por ambos integrantes del equipo, lo que permitió una colaboración eficiente y un seguimiento claro del progreso. El código fuente correspondiente a los ejercicios puede consultarse en el siguiente enlace: `https://git.ucr.ac.cr/GABRIEL.SILES/ie0424-gj`.

2. Práctica 2 RVfpga: Programación en C

2.1. Ejercicio 1: FlashSwitchesToLEDs

2.1.1. Descripción del problema [1]

El objetivo de este ejercicio es desarrollar un programa en lenguaje C que permita visualizar el valor de los switches en los LED de la tarjeta *RVfpga Nexys 4*. Para lograr esto, el valor de los switches debe reflejarse en los LED con un parpadeo visible, es decir, con una velocidad lo suficientemente lenta como para que el ojo humano pueda percibir el encendido y apagado.

2.1.2. Desarrollo de la solución

Al ser el primer ejercicio por desarrollar en el laboratorio se revisaron los ejemplos disponibles para comprender las direcciones de memoria para acceder al GPIO de los LEDs y los Switches. Siguiendo la guía de labo 1.

Una vez se comprendió el funcionamiento general hicimos el siguiente diagrama de flujo, con el cuál nos guiamos para realizar la solución del ejercicio:

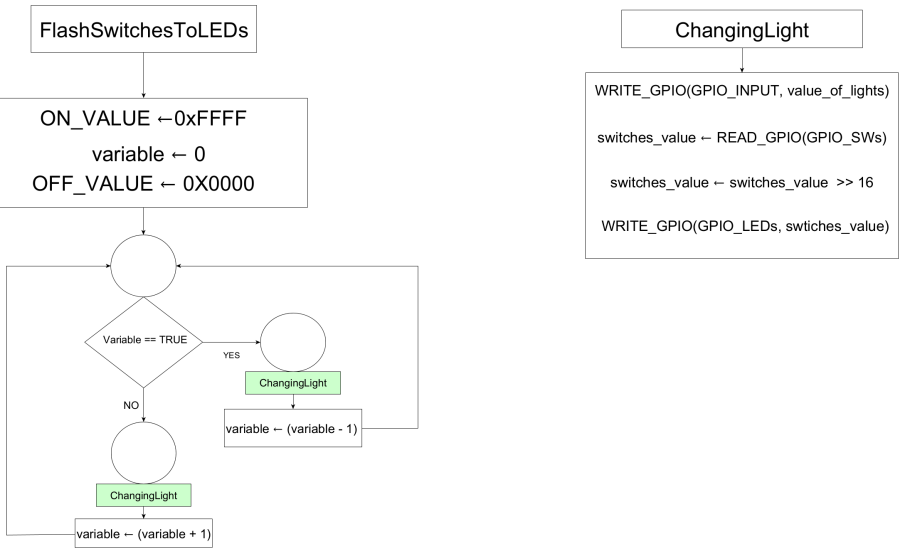


Figura 1: Diagrama de flujo del diseño FlashSwitchesToLEDs

2.1.3. Diseño Final

El siguiente programa en lenguaje *C* permite visualizar el estado de los switches en los LEDs de la tarjeta *RVfpga Nexys 4* con un efecto de parpadeo controlado. Para lograrlo, se utiliza una variable que alterna entre dos estados, simulando el comportamiento de una señal de reloj. El valor de los switches se muestra cuando la variable vale 1, y los LEDs se apagan cuando es 0. Se incluye un retardo para que sea visible al ojo humano.

```
#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); }

#if defined(D_NEXYS_A7)
   #include <bsp_printf.h>
   #include <bsp_mem_map.h>
   #include <bsp_version.h>
#else
   PRE_COMPILED_MSG("no platform was defined")
#endif
#include <psp_api.h>
#define DELAY 100000000

int main ( void )
{
    int ON_Value=0xFFFF, OFF_Value = 0x0000, switches_value;
    int variable = 0;
    int i;

    uartInit();

    while (1) {
        if (variable) {
             ChangingLight(switches_value, ON_Value);
             for (i=0; i < DELAY; i++) ;
             variable = variable - 1;
        }
        else {
             ChangingLight(switches_value, OFF_Value);
             printfNexys("FLASH OFF\n");
             for (i=0; i < DELAY; i++) ;
             printfNexys("CHANGING variable TO 1");
             variable++;
        }
    }
    return(0);
```

```
}

int ChangingLight(int switches_value, int value_of_lights) {
    WRITE_GPIO(GPIO_INOUT, value_of_lights);
    switches_value = READ_GPIO(GPIO_SWs);
    switches_value = switches_value >> 16;
    WRITE_GPIO(GPIO_LEDs, switches_value);
    return (0);
}
```

Explicación del diseño

El diseño consiste en lo siguiente:

- Se define una función `ChangingLight` que configura el valor de los LEDs con base en el valor de los switches (desplazado 16 bits).
- La función principal contiene un bucle infinito que alterna el valor de una variable (`variable`) entre 0 y 1.
- Cuando la variable es 1, se encienden los LEDs con el valor de los switches; cuando es 0, se apagan.
- Se incorpora una pausa entre los estados mediante un bucle de retardo artificial.
- Se imprime el estado actual mediante `printfNexys`, permitiendo una depuración visual a través de la consola UART.

Se puede visualizar la demostración en el siguiente video: https://youtu.be/FlashSwitchesToLEDs

2.2. Ejercicio 3: ScrollLEDs

2.2.1. Descripción del problema [1]

Este ejercicio consiste en escribir un programa en lenguaje C que despliegue un patrón animado en los LED de la tarjeta *RVfpga Nexys 4*, simulando un efecto de desplazamiento de luces encendidas.

El comportamiento esperado del programa es el siguiente:

1. Inicialmente, un solo LED encendido debe desplazarse de derecha a izquierda y luego de izquierda a derecha.
2. Luego, dos LEDs encendidos deben desplazarse en la misma forma: de derecha a izquierda y luego de regreso.
3. Posteriormente, tres LEDs encendidos deben hacer el mismo recorrido.
4. Este patrón debe continuar incrementando la cantidad de LEDs encendidos hasta que todos estén iluminados.
5. Una vez que todos los LEDs estén encendidos, el patrón debe reiniciarse desde un solo LED encendido.

2.2.2. Desarrollo de la solución

Al completar el primer ejercicio y comprender el funcionamiento de general, usamos el siguiente código llamado LedsSwitches como punto de partida.

```
#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); }

int main ( void )
{
    int En_Value=0xFFFF, switches_value;

    WRITE_GPIO(GPIO_INOUT, En_Value);

    while (1) {
        switches_value = READ_GPIO(GPIO_SWs);
        switches_value = switches_value >> 16;
        WRITE_GPIO(GPIO_LEDs, switches_value);
    }

    return(0);
}
```

A partir de este código empezamos a realizar modificaciones y notamos que podíamos emplear la función realizada para el ejercicio anterior llamada *ChangingLight* y desarrollamos el siguiente diagrama de flujo:

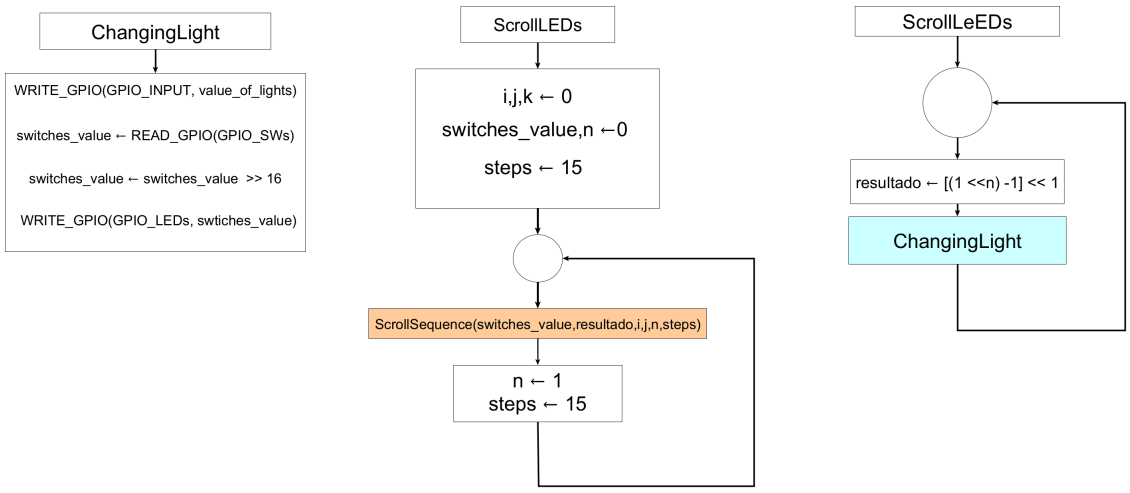


Figura 2: Diagrama de flujo de ScrollLEDs

Durante las primeras pruebas notamos que cumplía con el barrido de los LEDs de ida, pero nos faltaba implementar el barrido de retorno. Por lo que, en la función de ScrollLEDs agregamos otro ciclo for que iniciara en la última posición denomada *steps* y su valor se fuera reduciendo.

2.2.3. Diseño Final

El siguiente programa en lenguaje *C* implementa un patrón visual animado sobre los LEDs de la tarjeta *RVfpga Nexys 4*. Se trata de un desplazamiento hacia la derecha e izquierda, primero con un solo bit encendido, luego con dos, y así sucesivamente hasta encender todos los bits. Una vez alcanzado el máximo, el patrón se reinicia.

```
#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); }

#if defined(D_NEXYS_A7)
   #include <bsp_printf.h>
   #include <bsp_mem_map.h>
   #include <bsp_version.h>
#else
   PRE_COMPILED_MSG("no platform was defined")
#endif
#include <psp_api.h>
#define DELAY 1000000

int ChangingLight(int switches_value, int value_of_lights) {
    WRITE_GPIO(GPIO_INOUT, value_of_lights);
    switches_value = READ_GPIO(GPIO_SWs);
    switches_value = switches_value >> 16;
    WRITE_GPIO(GPIO_LEDs, switches_value);
    return (0);
}

int ScrollSequence(int switches_value, int resultado, int i, int j, int n, int steps) {
    for(i=0; i < steps; i++){
        resultado = (((1 << n) - 1) << i);
        for (j=0; j < DELAY; j++) ;
        ChangingLight(switches_value, resultado);
        for (j=0; j < DELAY; j++) ;
    }

    for(i = steps; i >= 0; i--) {
        resultado = (((1 << n) - 1) << i);
        for (j = 0; j < DELAY; j++) ;
        ChangingLight(switches_value, resultado);
        for (j=0; j < DELAY; j++) ;
    }
    return(0);
}
```

```
int main ( void ) {
    int i, j, k, resultado = 0;
    int switches_value, n = 1;
    int steps = 15;

    while(1){
        for(k = 0; k < 16; k++){
            ScrollSequence(switches_value, resultado, i, j, n, steps);
            n++;
            steps--;
        }
        n = 1;
        steps = 15;
    }
    return(0);
}
```

Explicación del diseño

El programa realiza un desplazamiento dinámico de bits en los LEDs basado en el siguiente diseño:

- Se utiliza una función `ScrollSequence` para realizar el recorrido de izquierda a derecha y viceversa, aumentando progresivamente el número de bits encendidos.
- El patrón de bits encendidos se genera mediante la expresión `(((1 << n) - 1) << i)`, donde n indica cuántos bits consecutivos estarán encendidos y i determina su posición de inicio.
- La función `ChangingLight` se encarga de actualizar el estado de los LEDs según el valor calculado, y además mantiene sincronizados los valores de los switches.
- Se utiliza un doble retardo para hacer visible la animación, y se reinicia el proceso una vez que todos los LEDs se han encendido.

Se puede visualizar la demostración en el siguiente video: https://youtu.be/ScrollLEDs

3. Práctica 2 RVfpga: Lenguaje ensamblador de RISC-V

3.1. Ejercicio 2: DisplayInverse

3.1.1. Descripción del problema [2]

En este ejercicio se debe desarrollar un programa en lenguaje ensamblador *RISC-V* que muestre en los LED el valor inverso del estado actual de los switches de la tarjeta *RVfpga Nexys 4*. La operación consiste en aplicar una inversión bit a bit al valor de entrada proveniente de los switches y reflejar el resultado en los LED.

Por ejemplo:

- Si los switches están en el estado binario `0101010101010101`, los LEDs deben mostrar `1010101010101010`.
- Si los switches están en el estado `1111000011110000`, los LEDs deben mostrar `0000111100001111`.

3.1.2. Desarrollo de la solución

Se empleo de base el código de LEDSwitches que es el siguiente:

```
#define GPIO_SWs 0x80001400
#define GPIO_LEDs 0x80001404
#define GPIO_INOUT 0x80001408

.globl main
main:
    li x28, 0x5555
    li x29, GPIO_INOUT
    sw x28, 0(x29)          # Write the Enable Register

next:
    li a1, GPIO_SWs         # Read the Switches
    lw t0, 0(a1)
    li a0, GPIO_LEDs
    srl t0, t0, 16
    sw t0, 0(a0)            # Write the LEDs
    beq zero, zero, next

.end
```

Notamos que para realizar la inversión de los LEDs necesitábamos aplicar una máscara XOR que cuenta con la siguiente tabla de verdad:

A	**B**	**A** $\oplus$ **B**
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 1: Tabla de verdad de la compuerta XOR (A $\oplus$ B)

A partir de esto, se procedió a realizar el siguiente diagrama de flujo:

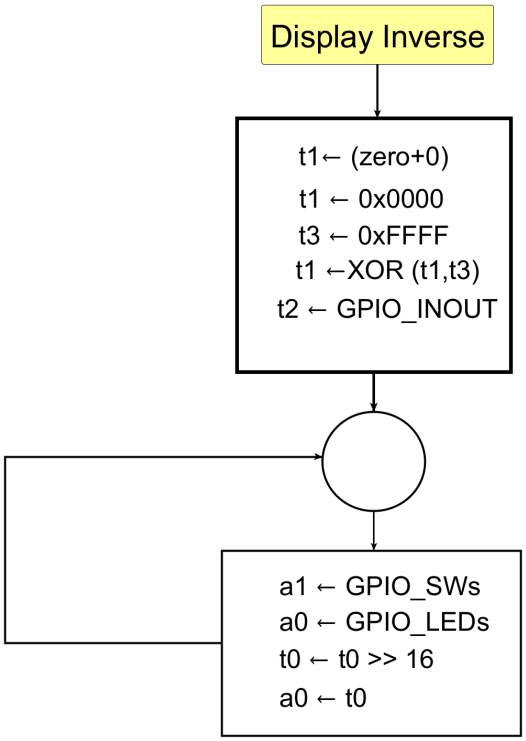


Figura 3: Diagrama de Flujo del diseño de DisplayInverse

3.1.3. Diseño Final

Finalmente, mediante el uso de la máscara XOR se logró tener el siguiente código en ensamblador *RISC-V* implementa una funcionalidad simple pero efectiva: mostrar el valor invertido de los switches en los LEDs de la tarjeta *RVfpga Nexys 4*. Esto se logra aplicando una operación bit a bit de XOR entre el valor leído y un patrón de todos los bits en alto (0xFFFF).

```
#define GPIO_SWs 0x80001400
#define GPIO_LEDs 0x80001404
#define GPIO_INOUT 0x80001408

.globl main
main:
    addi t1,zero,0
    li t1, 0x0000
    li t3,0xffff
    xor t1, t1,t3
    li t2, GPIO_INOUT
    sw t1, 0(t2)

next:
    li a1, GPIO_SWs
    lw t0, 0(a1)
    li a0, GPIO_LEDs
    srl t0, t0, 16
```

```
    sw t0, 0(a0)
    beq zero, zero, next

.end
```

Explicación del diseño

El programa realiza los siguientes pasos:

- Inicializa el valor de entrada/salida en el registro correspondiente, permitiendo que los LEDs sean manipulados por el software.
- Aplica una máscara XOR con `0xFFFF` para obtener el valor invertido de los switches.
- En un bucle infinito, lee continuamente el valor actual de los switches desde la dirección `GPIO_SWs`.
- Realiza un corrimiento a la derecha de 16 bits (`srl`) para alinear los bits de interés (si es necesario según el diseño del hardware).
- Escribe el valor resultante en los LEDs a través del puerto `GPIO_LEDs`.

Se puede visualizar la demostración en el siguiente video: https://youtu.be/DisplayInverse

3.2. Ejercicio 4: 4bitAdder

3.2.1. Descripción del problema [2]

El objetivo de este ejercicio es escribir un programa en ensamblador *RISC-V* que realice la suma sin signo (*unsigned*) de dos valores de 4 bits extraídos de los switches de la tarjeta *RVfpga Nexys 4*. En particular, se deben sumar:

- Los 4 bits menos significativos (LSB) de los switches.
- Los 4 bits más significativos (MSB) de los switches.

El resultado de esta suma debe mostrarse en los 4 bits menos significativos (más a la derecha) de los LEDs. Además, si la suma genera un *carry out* (desbordamiento sin signo), entonces el quinto LED desde la derecha debe encenderse para indicar esta condición de overflow.

3.2.2. Desarrollo de la solución

Se basó en el código Blinky para el diseño del del presente ejercicio:

```
#define GPIO_LEDs 0x80001404
#define GPIO_INOUT 0x80001408
#define DELAY 0x100000              /* Define the DELAY */

.globl main
main:
    li x28, 0xFFFF
```

```
    li a0, GPIO_INOUT
    sw x28, 0(a0)                   # Write the Enable Register

    li t1, DELAY                    # Set timer value to control blink speed
    li t0, 0

bl1:
    li a0, GPIO_LEDs
    sb t0, 0(a0)                    # Write to LEDs
    xori t0, t0, 1                  # Invert LED
    and t2, zero, zero              # Reset timer

time1:                              # Delay loop
    addi t2, t2, 1
    bne t1, t2, time1
    j bl1
```

De igual forma, se creo el siguiente diagrama de flujo para poder solucionar el ejercicio y se procedió a escribir el código sobre la base del ejemplo brindado por el material del curso.

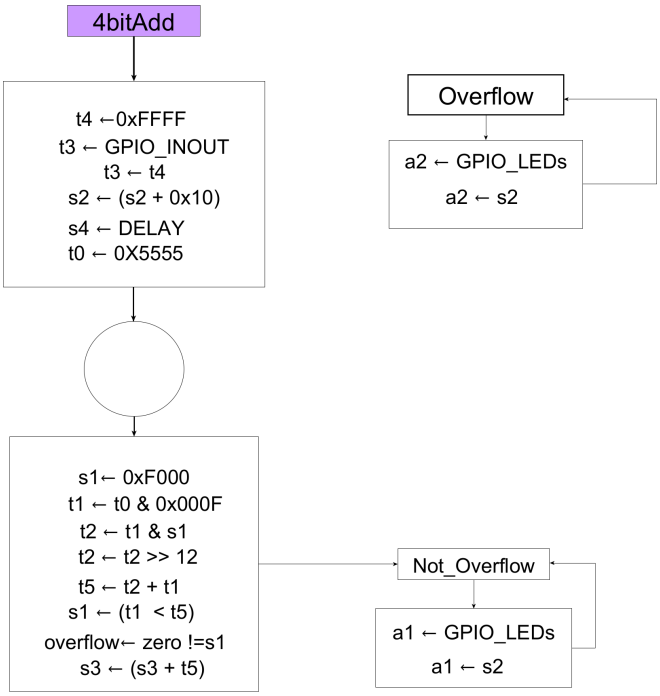


Figura 4: Diagrama de Flujo del 4bitAdder

3.2.3. Diseño Final

El siguiente código en ensamblador RISC-V implementa un sumador de 4 bits a partir de la lectura de un valor fijo (representando los switches), separando los 4 bits menos significativos y los 4 más significativos, y sumándolos de forma no signada. El resultado se muestra en los 4 LEDs menos significativos, y se enciende un LED adicional (el quinto) en caso de que ocurra un desbordamiento (*unsigned overflow*).

```
#define GPIO_SWs 0x80001400
#define GPIO_LEDs 0x80001404
#define GPIO_INOUT 0x80001408
#define DELAY 0x100000
.globl main

main:
    addi t1,zero,0
    addi t0,zero,0

    li x29, 0xFFFF
    li x28, GPIO_INOUT
    sw x29, 0(x28)

    addi s2,s2,0x10
    li s4, DELAY

    # t0 valor a cambiar para suma
    li t0, 0x5555

loop:
    addi sp, sp, -12
    sw s1, 0(sp)
    sw t1, 4(sp)
    sw t2, 8(sp)

    li s1, 0xF000
    andi t1, t0, 0x000F      # extrae los 4 LSB
    and t2, t0, s1          # extrae los 4 MSB
    srli t2, t2, 12         # los alinea a la derecha
    add t5, t2, t1          # suma sin signo

    lw s1, 0(sp)
    lw t1, 4(sp)
    lw t2, 8(sp)
    addi sp,sp,12

    li t1, 0xF              # límite sin overflow
    sltu s1, t1, t5
    bne s1, zero, overflow
    add s3, s3, t5

not_overlow:
    li a1, GPIO_LEDs
    sw s3, 0(a1)
    addi t2,zero,0
    j wait_loop_1
```

```
overflow:
    li a2, GPIO_LEDs
    sw s2, 0(a2)
    addi t2,zero,0
    j wait_loop_2

wait_loop_1:
    addi t2,t2,1
    bne s4, t2, wait_loop_1
    j not_overlow

wait_loop_2:
    addi t2,t2,1
    bne s4, t2, wait_loop_2
    j overflow
.end
```

Explicación del diseño

El programa realiza las siguientes acciones:

- Configura el puerto de entrada/salida para permitir la escritura en los LEDs.
- Extrae los 4 bits menos significativos (LSB) y los 4 más significativos (MSB) del valor cargado en `t0`.
- Suma ambos valores de forma no signada.
- Si la suma excede los 4 bits (mayor a 15), se detecta un overflow y se enciende el quinto LED.
- En ambos casos, se incluye una pausa con un bucle de espera (`DELAY`) para que los resultados sean visibles.

Se puede visualizar la demostración en el siguiente video: https://youtu.be/4bitAdder

4. Práctica 3 RVfpga: Llamadas a función

4.1. Ejercicio 7: Triplets

4.1.1. Descripción del problema [3]

Este ejercicio tiene como objetivo implementar un programa en lenguaje ensamblador *RISC-V* que procese un vector de entrada A, compuesto por $3 \times N$ elementos, y genere un nuevo vector B de tamaño N. Cada elemento de B debe ser el valor absoluto de la suma de una terna de elementos consecutivos en A.

Formalmente, la operación que se debe realizar es la siguiente:

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

El programa debe cumplir con las convenciones de llamada del estándar *RISC-V* y seguir el siguiente esquema de alto nivel:

- El programa principal itera sobre las ternas de A y llena el vector B utilizando la función `res_triplet`.
- La función `res_triplet` toma como entrada un vector y una posición inicial, y retorna el valor absoluto de la suma de tres elementos consecutivos a partir de esa posición.
- La función `abs` retorna el valor absoluto de un número entero.

4.1.2. Desarrollo de la solución

La estructura del código se diseño de manera para que existieran 3 bloques, el primero la función main en donde se guardarán en los registros a las variables creadas en .data y luego dar inicio al for. En este loop principal se llama la función de res_triplet. A partir del código en C que se visualiza a continuación se desarrolla en RISC-V.

```
#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;
    }
}

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;
}
```

res_tiplet: En esta función con base a la dirección del lista con los valores a trabajar se crea el array y dando inicio a el for que daría inicio a la operación. Su funcionamiento se basa en los siguientes pasos:

- ***res_triplet_loop***: inicia el contador i que terminará en 3 debido al tamaño de N que es 4, obteniendo el valor *i* e *i + 1* de A para empezar a sumarlos, obteniendolos uno a uno.

- ***res_triplet_abs***: intermediario para conectar *res_triplet_loop* y la función abs encargada de obtener el valor absoluto, transfiere el resultado de la suma total.
- ***abs_function***: obtiene el valor absoluto de la suma total, se asegura que el resultado obtenido sea positivo, salta a *abs_end* para regresar al loop de *res_triplet_abs* o realiza el cambio de signo de valores negativos.
- ***res_triplet_end***: Se encarga de terminar el loop guardando en *a0* abs(sum) devolviendo los valores de pila y continuando el for principal.

Una vez esta función cumple estos pasos se asocia en el loop principal el valor obtenido para *B[i]* el cual a diferencia del contador i de *res_triplet* este es un contador global que cambia al acceder a memoria. A continuación se puede apreciar el diagrama de flujo de código:

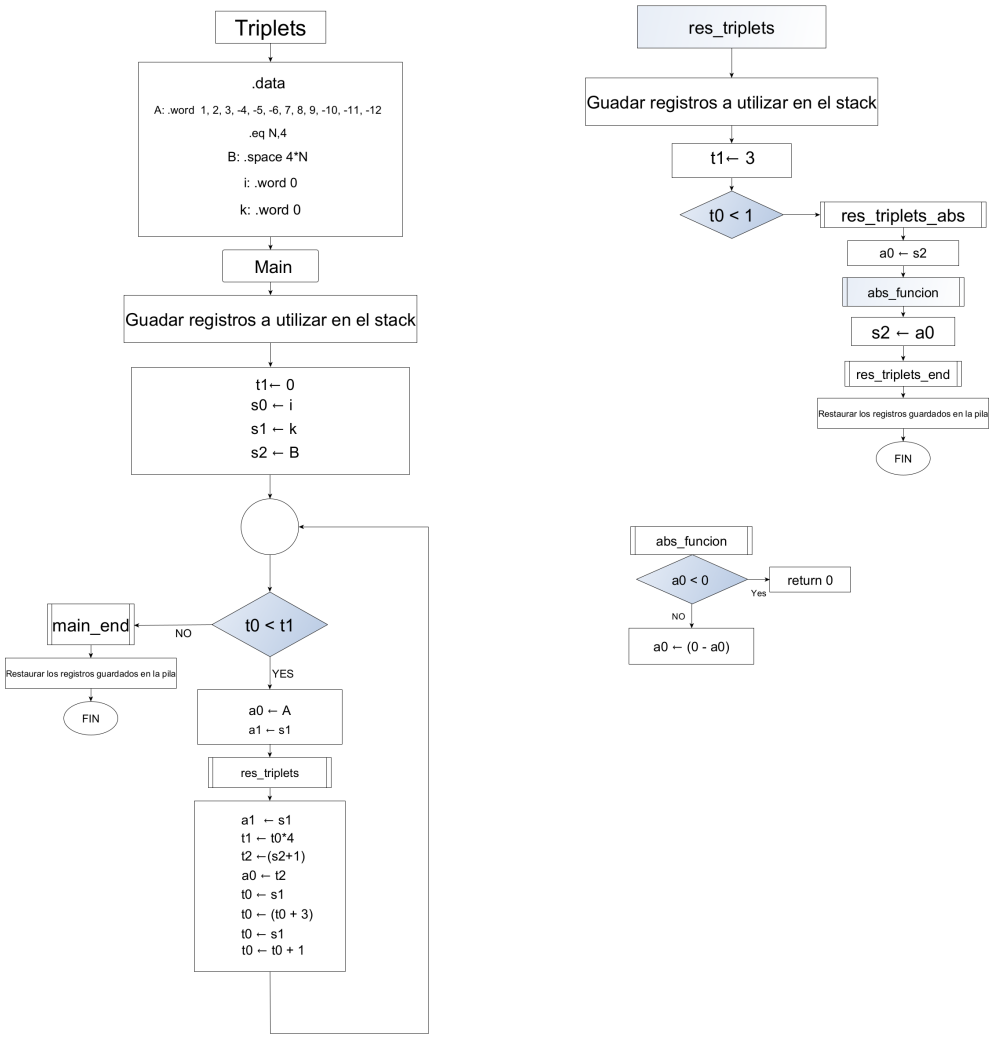
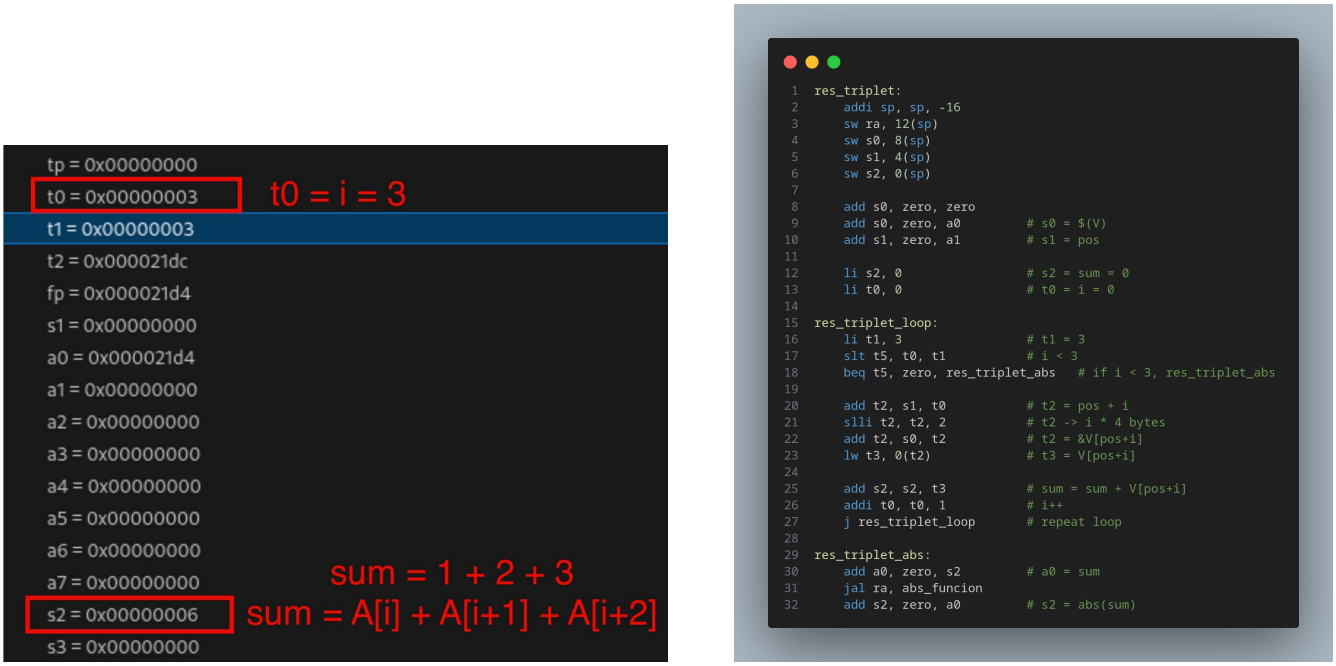


Figura 5: Diagrama de flujo triplets

4.1.3. Diseño Final

Para el diseño se define N como 4, el array de $A = [1, 2, 3, -4, -5, -6, 7, 8, 9, -10, -11, -12]$, se utilizan dos contadores globales i y k, permitiendo aumentarlos en memoria directamente. La

principal función es *res_triplet* la cual a continuación se puede observar realizando la primera iteración para asignar a *B[0]*, en este caso siendo la suma de los primeros tres elementos de A. Teniendo como resultado 6, dentro de la función *res_triplets* se realiza en un for 3 veces como se muestra en la Figura 6a.



(a) Valores de registros

(b) Código de res_triplets

Figura 6: Funcionamiento de la función *res_triplets*

En esta misma función se llama la función *abs(x)* que obtiene el valor absoluto de la suma, por lo que el caso donde *sum* sea un valor negativo la función realizará una resta para cambiar el signo, tal como se muestra en la Figura 7.

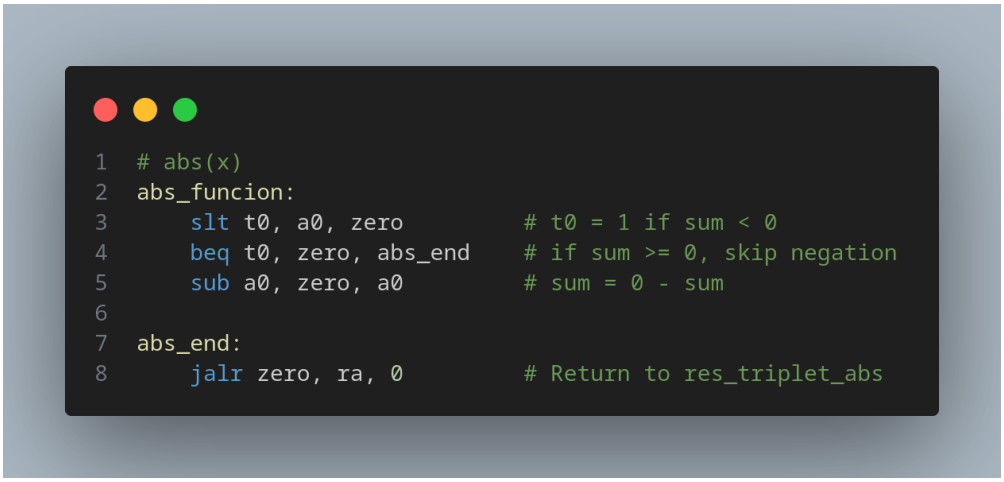


Figura 7: Función *abs(x)*

El ejemplo en donde se hace uso de la función se puede visualizar en la Figura 8, para llegar al resultado que se guarda en *s2* se realizó lo siguiente:

$$-4 + (-5) = -9 \rightarrow 0xFC + 0xFB = 0xF7$$
$$-9 + (-6) = -15 \rightarrow 0xF7 + 0xFA = 0xF1$$

Por lo que al finalizar el loop de *res_triplets* y se finaliza la suma se guarda en *s2* el número negativo, saltando a la función *abs* convirtiendo -15 a 15 guardándolo en a0 como *0x0F*.

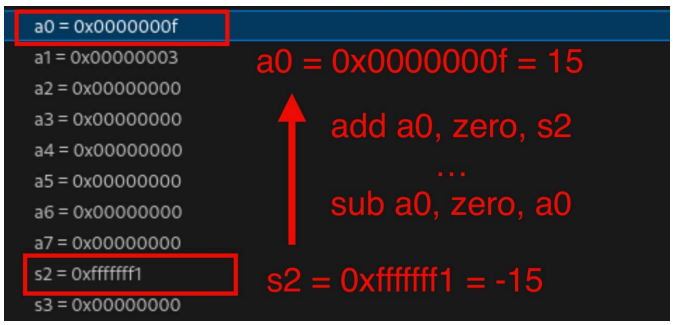


Figura 8: Ejemplo de funcionamiento de la función abs

Por último, para poder asociar cada valor obtenido a B se vuelve al llamado del *loop* y se asocia a *B[i]*. En la Figura 9 se observa que en la línea 58 se asocia el resultado. Posteriormente se aumenta el contador para volver al for principal de la función.

```
1  loop:
2      lw t0, 0(s0)            # t0 = i
3      li t1, N                # t1 = N = 3
4      slt t4, t0, t1          # i < N
5      beq t4, zero, main_end  # Sale del ciclo For
6
7      # a0 -> $(A)
8      lui a0, %hi(A)
9      addi a0, a0, %lo(A)
10
11     lw a1, 0(s1)            # a1 = k
12     jal ra, res_triplet
13
14     lw t0, 0(s0)            # t0 = i
15     slli t1, t0, 2          # t1 -> i * 4 bytes
16     add t2, s2, t1          # B[i]
17     sw a0, 0(t2)            # B[i] = resultado
18
19     lw t0, 0(s1)            # t0 = k
20     addi t0, t0, 3          # k = k+3
21     sw t0, 0(s1)
22
23     lw t0, 0(s0)            # t0 = i
24     addi t0, t0, 1          # i++
25     sw t0, 0(s0)            # save i -> word
26
27     j loop
```

Figura 9: Loop for principal

El código se puede encontrar en el repositorio git.ucr.ac.cr/Triplets.S

4.2. Ejercicio 8: Filters

4.2.1. Descripción del problema [3]

El objetivo de este ejercicio es implementar un programa en lenguaje ensamblador *RISC-V*, llamado `Filter.S`, que procese un arreglo de enteros conforme a un criterio de filtrado específico. El programa debe ser compatible con las convenciones estándar de manejo de funciones establecidas para la arquitectura *RISC-V*.

A partir del arreglo A de tamaño $N = 6$, definido como:

$$A = \{48, 64, 56, 80, 96, 48\}$$

se debe generar un nuevo arreglo B, también de tamaño N, donde se almacenan resultados únicamente si se cumple cierta condición. El proceso puede representarse mediante 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++;
    }
}
```

La lógica del filtro está definida por la función `myFilter`, que retorna 1 si el primer argumento es múltiplo de 16 y el segundo argumento es mayor que el primero. En caso contrario, retorna 0.

El programa en ensamblador debe incluir las siguientes secciones:

- `.data`: para inicializar el arreglo de entrada.
- `.bss`: para reservar espacio para el arreglo de salida.
- `.text`: para el código de la función principal y la función `myFilter`.

4.2.2. Desarrollo de la solución

Con referencia al ejercicio resuelto anteriormente, se comenzó primero pasando el código de C a RISC-V. Y se realiza el siguiente diagrama de flujo con el fin de diseñar una base para implementar la función *MyFilter*.

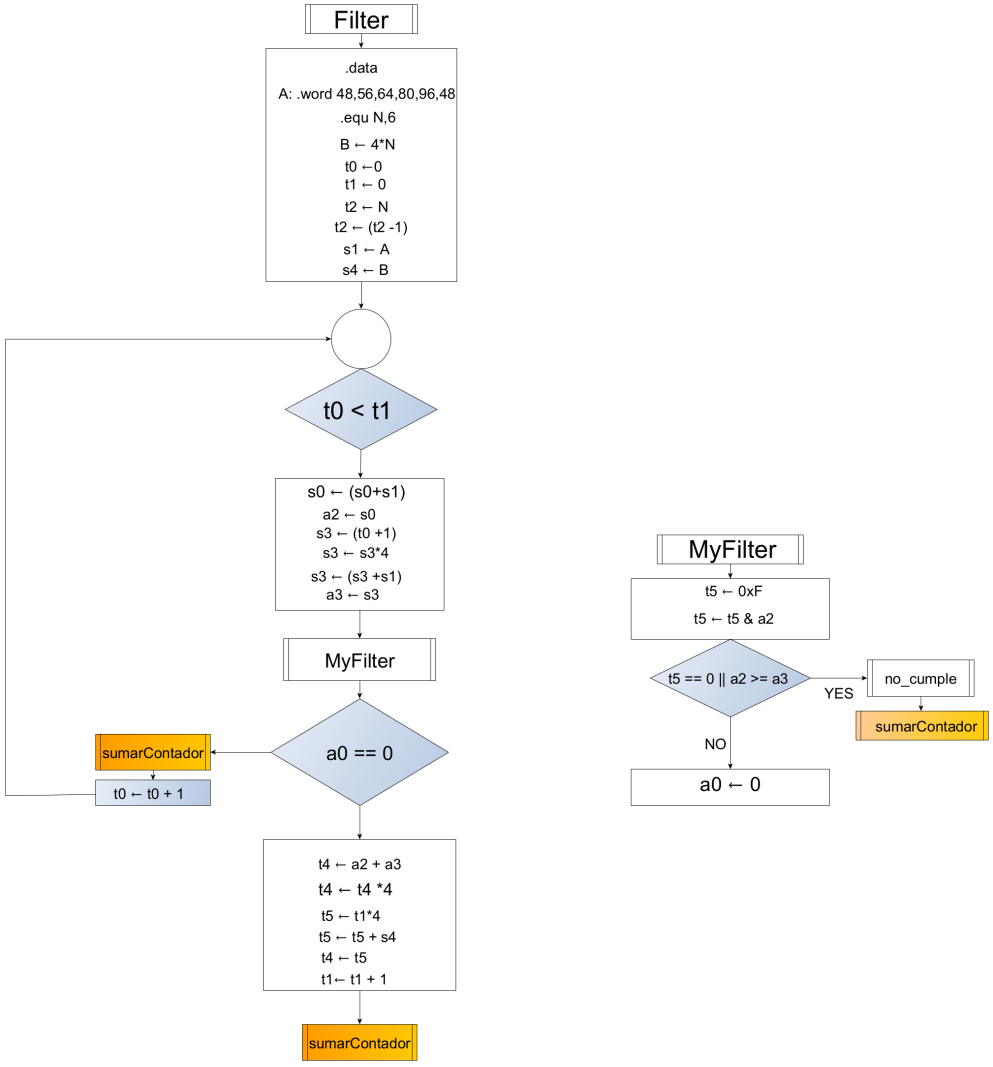


Figura 10: Diagrama de flujo para Filter

Subrutina `MyFilter`

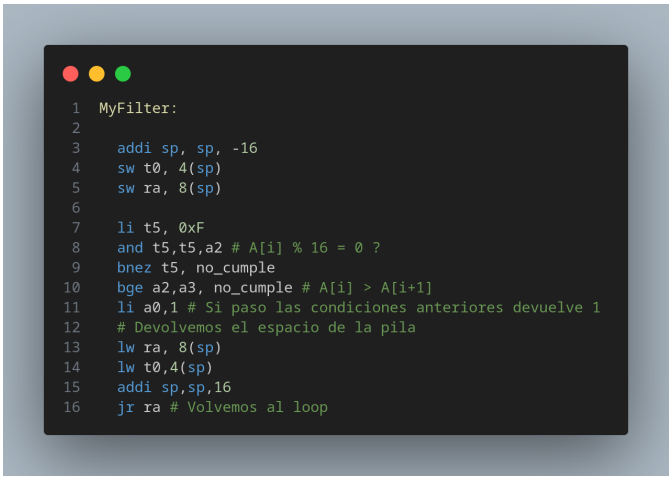
La subrutina `MyFilter` comienza con el respaldo en la pila de los registros utilizados. Luego, se procede a evaluar los siguientes criterios:

- Se aplica una máscara con `0x000F` usando una operación `AND` al valor de $A[i]$, lo que permite verificar si es múltiplo de 16. Esta condición se cumple únicamente si los cuatro bits menos significativos son cero.
- Seguidamente, se verifica si $A[i+1] > A[i]$, lo cual se implementa con una instrucción `bge` en sentido contrario: si $A[i] \geq A[i+1]$, se considera que no cumple la condición.

Si ambas condiciones se cumplen, se asigna 1 al registro `a0`, que actúa como valor de retorno. En caso contrario, se asigna 0. Finalmente, se restauran los registros previamente guardados en la pila, se ajusta el puntero de pila y se retorna al programa principal usando la instrucción `jr ra`.

4.2.3. Diseño Final

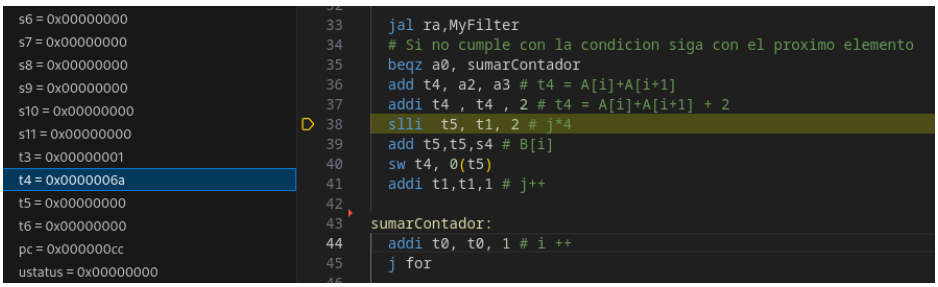
Para el diseño final se implementa el programa en C a RISC-V. Para esta función, teniendo el array A y un tamaño N, se va a generar un nuevo array B. Para el primer elemento del array se cumple la condición de divisibilidad entre 16 y que $A[i] < A[i+1]$. Al momento de entrar al for y se genera el array se llama la función $myFilter$ para verificar la divisibilidad, en a2 se encuentra el valor de $A[i]$ y al aplicar la máscara se guarda en $t5$ si se cumple la condición, si es 1 continúa al loop para realizar la suma y asignar a $B[i]$, si es 0 salta a aumentar contadores y revisar la condición de divisibilidad guardando el valor en a0.



```
1  MyFilter:
2
3    addi sp, sp, -16
4    sw t0, 4(sp)
5    sw ra, 8(sp)
6
7    li t5, 0xF
8    and t5,t5,a2 # A[i] % 16 = 0 ?
9    bnez t5, no_cumple
10   bge a2,a3, no_cumple # A[i] > A[i+1]
11   li a0,1 # Si paso las condiciones anteriores devuelve 1
12   # Devolvemos el espacio de la pila
13   lw ra, 8(sp)
14   lw t0,4(sp)
15   addi sp,sp,16
16   jr ra # Volvemos al loop
```

Figura 11: Función myFilter

Ahora al cumplirse la condición para este caso donde 48 es múltiplo de 16 y el siguiente valor es 56, siendo número mayor, se procede a sumar los valores guardando en ***a2=0x30*** (48) y ***a3=0x38*** (56), para después en $t4$ guardar $A[i]+A[i+1]+2$ por lo que se obtiene ***t4 = 0x6a*** (106) la suma de los primeros 2 elementos más el valor de 2. Se puede observar a continuación el código específico donde se realizan las sumas, asociándose a B con respecto a la dirección de $B[i]$ y posteriormente el aumentando los contadores j e i. Este proceso se repite hasta N-1.



```
s6 = 0x00000000
s7 = 0x00000000
s8 = 0x00000000
s9 = 0x00000000
s10 = 0x00000000
s11 = 0x00000000
t3 = 0x00000001
t4 = 0x0000006a
t5 = 0x00000000
t6 = 0x00000000
pc = 0x000000cc
ustatus = 0x00000000

33   jal ra,MyFilter
34   # Si no cumple con la condicion siga con el proximo elemento
35   beqz a0, sumarContador
36   add t4, a2, a3 # t4 = A[i]+A[i+1]
37   addi t4 , t4 , 2 # t4 = A[i]+A[i+1] + 2
38   slli  t5, t1, 2 # j*4
39   add t5,t5,s4 # B[i]
40   sw t4, 0(t5)
41   addi t1,t1,1 # j++
42
43 sumarContador:
44   addi t0, t0, 1 # i ++
45   j for
```

Figura 12: Bloque de código de suma de valores en A y asociación de resultado a B

El código se puede encontrar en el repositorio git.ucr.ac.cr/Filter.S

5. Problemas y dificultades

A la hora de desarrollar las diferentes prácticas se presentaron diferentes dificultades y casos que provocaron fallos en el código los cuales fueron los siguientes:

- **FlashSwitchesToLEDs**: a la hora de realizar el código no se podía conocer el estado de las luces LED debido a la velocidad, por lo que se agregó un delay mayor para poder observar el cambio.
- **ScrollLEDs**: de igual manera a la práctica anterior la velocidad provocando que se visualizara un LED de más debido al desplazamiento veloz creando la duda si existía un error a nivel funcional, lo que conllevo a revisar exhaustivamente la situación concluyendo en la velocidad de desplazamiento.
- **DisplayInverse**: el código no funcionaba como debía teniendo en cuenta lo simple que era no se encontraba error en la lógico por lo que se tuvo que revisar paso a paso con el Debugger para observar el cálculo correcto de los registros observando que un registro iniciaba con un valor predeterminado, por lo que se solucionó limpiando el registro antes de realizar los cálculos.
- **Triplets**: se tuvo dificultades con contadores y accesos a memoria por lo que se tuvo que repasar el tema, debido que existía errores al no manejar correctamente los registros y datos se traslapaban unos sobre otros, por lo que se utilizó un método de contadores globales para diferenciarlo entre la función de res_triplets

6. Conclusiones y recomendaciones

- Se recomienda revisar paso a paso el contenido de los registros y verificar que la información deseada se esté almacenando correctamente. En ocasiones, el código puede parecer funcional desde el punto de vista teórico, pero errores como valores residuales en registros o un mal manejo de la información pueden alterar completamente el resultado final.
- A lo largo de las prácticas se logró una familiarización progresiva con la plataforma de desarrollo Nexys 4, así como con herramientas como PlatformIO y extensiones para Visual Studio Code, aplicando un método de trabajo ordenado y eficiente.
- Se desarrolló e implementó código funcional tanto en lenguaje C como en ensamblador RISC-V, en conjunto con el hardware de la placa Nexys 4. Esto permitió comprender el acceso a direcciones específicas mediante estructuras de código bien definidas, así como el uso de herramientas de depuración integradas.
- En conclusión, se lograron completar satisfactoriamente todos los ejercicios propuestos en lenguaje ensamblador RISC-V. A pesar de ser la primera experiencia con este lenguaje, se alcanzaron los resultados esperados gracias a una adecuada comprensión del funcionamiento de la arquitectura y a una correcta aplicación de las convenciones de programación.

Referencias

[1] Imagination University Programme, *RVfpga Lab 1: C Programming*, ver. 2.2, Available from the Imagination University Programme, Imagination Technologies, 2022.

[2] Imagination University Programme, *RVfpga Lab 2: RISC-V Assembly Language*, ver. 2.2, Available from the Imagination University Programme, Imagination Technologies, 2022.

[3] Imagination University Programme, *RVfpga Lab 3: Function Calls*, ver. 2.2, Available from the Imagination University Programme, Imagination Technologies, 2022.