Practica 3

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.

Here's the code in assembly language for RISC-V architecture:

```
.global _start      # Inicio del programa

.equ GPIO_BASE, 0x10012000    # Dirección base del puerto GPIO

_start:
    la x10, GPIO_BASE   # Cargamos la dirección base del puerto GPIO en el registro x1
0

loop:
    lw x11, 0(x10)      # Leemos el valor de los interruptores desde el puerto GPIO
    sw x11, 4(x10)      # Escribimos el valor de los interruptores en el puerto GPIO p
ara encender los LEDs
    addi x12, x0, 1000000   # Esperamos 1 segundo (1 millón de ciclos)
delay:
    addi x12, x12, -1   # Decrementamos el contador de espera
    bnez x12, delay     # Si el contador no ha llegado a cero, repetimos el ciclo de e
spera
    sw x0, 4(x10)       # Apagamos los LEDs
    j loop              # Vuelta al inicio del bucle principal
```

Este código utiliza los registros x10, x11 y x12 para manipular el puerto GPIO y generar el parpadeo de los LEDs. El registro x10 se carga con la dirección base del puerto GPIO, el registro x11 se utiliza para almacenar el valor de los interruptores leído desde el puerto GPIO y el registro x12 se utiliza para contar el tiempo de espera.

El programa se ejecuta en un bucle infinito, en el que se lee el valor de los interruptores desde el puerto GPIO, se escribe ese valor en el puerto GPIO para encender los LEDs, se espera un segundo y se apagan los LEDs, para luego volver al inicio del bucle. El tiempo de espera se cuenta utilizando un contador que se

decrementa en cada ciclo, y se repite el ciclo de espera mientras el contador no ha llegado a cero.

Espero que este ejemplo te sea útil para entender cómo se puede programar en ensamblador RISC-V para controlar dispositivos como los interruptores y los LEDs.

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.

```
.global _start      # Inicio del programa

.equ GPIO_BASE, 0x10012000    # Dirección base del puerto GPIO

_start:
    la x10, GPIO_BASE   # Cargamos la dirección base del puerto GPIO en el registro x1
0

loop:
    lw x11, 0(x10)      # Leemos el valor de los interruptores desde el puerto GPIO
    not x11, x11        # Invertimos el valor de los interruptores
    sw x11, 4(x10)      # Escribimos el valor invertido en el puerto GPIO para encende
r los LEDs
    addi x12, x0, 1000000   # Esperamos 1 segundo (1 millón de ciclos)
delay:
    addi x12, x12, -1   # Decrementamos el contador de espera
    bnez x12, delay     # Si el contador no ha llegado a cero, repetimos el ciclo de e
spera
    sw x0, 4(x10)       # Apagamos los LEDs
    j loop              # Vuelta al inicio del bucle principal
```

En este caso, se utiliza la instrucción `not` para invertir el valor de los interruptores leído desde el puerto GPIO. Luego, se escribe ese valor invertido en el puerto GPIO para encender los LEDs. El resto del código es similar al del ejemplo anterior, en el que se espera un segundo y se apagan los LEDs antes de volver al inicio del bucle principal.

Espero que este ejemplo te sea útil para entender cómo se puede programar en ensamblador RISC-V para invertir los valores de los interruptores y controlar los LEDs en consecuencia.

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.

```
.global _start      # Inicio del programa

.equ GPIO_BASE, 0x10012000    # Dirección base del puerto GPIO
.equ LEDS, 0xFF      # Valor de todos los LEDs encendidos

_start:
    la x10, GPIO_BASE   # Cargamos la dirección base del puerto GPIO en el registro x1
0

loop:
    addi x11, x0, 0x01  # Empezamos con un LED encendido
    shift_left:
    slli x12, x11, 8    # Desplazamos el valor del LED encendido hacia la izquierda en
el registro x12
    sw x12, 4(x10)      # Escribimos el valor del LED encendido en el puerto GPIO para
encender los LEDs correspondientes
    addi x11, x11, 1    # Añadimos un LED encendido
    bnez x11, shift_left   # Si no hemos encendido todos los LEDs, repetimos el ciclo
 de desplazamiento hacia la izquierda
shift_right:
    srai x12, x12, 2    # Desplazamos el valor de los LEDs encendidos hacia la derecha
en el registro x12
    sw x12, 4(x10)      # Escribimos el valor de los LEDs encendidos en el puerto GPIO
para encender los LEDs correspondientes
    bnez x11, shift_right  # Si no hemos apagado todos los LEDs, repetimos el ciclo de
```

```
desplazamiento hacia la derecha
    beq x12, LEDS, loop   # Si hemos encendido todos los LEDs, volvemos al inicio del
 bucle principal para repetir el patrón
```

En este caso, el programa utiliza las instrucciones `slli` y `srai` para desplazar los valores de los LEDs encendidos hacia la izquierda y hacia la derecha, respectivamente. El ciclo principal del programa consiste en encender un número creciente de LEDs, desplazarlos hacia la izquierda, luego hacia la derecha, y repetir el proceso hasta encender todos los LEDs. Una vez que todos los LEDs están encendidos, se vuelve al inicio del bucle principal para repetir el patrón desde el principio.

Espero que este ejemplo te sea útil para entender cómo se puede programar en ensamblador RISC-V para controlar el desplazamiento de LEDs de forma dinámica y repetitiva.

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).

```
# Inicializar los registros

.data
.align 4

num1:   .word 0x000F    # 4 bits menos significativos de los interruptores
num2:   .word 0xFFF0    # 4 bits más significativos de los interruptores
result: .word 0x0000    # Resultado de la suma
mask:   .word 0x001F    # Máscara para los 5 bits menos significativos de los LEDs

.text
.align 2
.globl _start

_start:
    # Cargar los valores iniciales en los registros
```

```
    lw t0, num1
    lw t1, num2
    lw t2, result
    lw t3, mask

    # Realizar la suma
    add t2, t0, t1

    # Comprobar si hay desbordamiento sin signo
    sltu t4, t2, t0
    beqz t4, skip
    ori t2, t2, 0x0010

skip:
    # Aplicar la máscara para mostrar el resultado en los 4 bits menos
    signifitivos de los LEDs
    and t2, t2, t3

    # Mostrar el resultado en los LEDs
    sw t2, LEDS

    # Bucle infinito

loop:
    j loop

    .align 2

    LEDS:   .word 0x0000    # Registro de LEDs
```

Este programa carga los valores iniciales en los registros `t0`, `t1`, `t2` y `t3`, y luego realiza la suma de los dos primeros. Después, comprueba si hay un desbordamiento sin signo y establece el quinto bit en el resultado si es así. A continuación, aplica una máscara para mostrar solo los 4 bits menos significativos de los LEDs, y luego almacena el resultado en el registro de LEDs. Finalmente, el programa entra en un bucle infinito para que los LEDs sigan mostrando el resultado.

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.

```
.section .data
a: .word 24
b: .word 36
MCD: .word 0

.section .text
```

```
.globl _start

_start:
    # cargar los valores de a y b en los registros
    lw a0, a
    lw a1, b

loop:
    # calcular el resto de a / b
    remu a0, a0, a1

    # si el resto es 0, entonces b es el MCD
    beqz a0, done
    # de lo contrario, intercambiar a y b y continuar
    mv a0, a1
    mv a1, a0
    j loop

done:
    # almacenar el MCD en la memoria y salir del programa
    sw a0, MCD
    li a7, 10
    ecall
```

En este ejemplo, los valores de a y b están definidos estáticamente en la sección .data del programa. Primero, cargamos los valores de a y b en los registros a0 y a1, respectivamente. Luego, entramos en un bucle donde se calcula el resto de la división de a por b utilizando la instrucción remu. Si el resto es cero, significa que b es el MCD y salimos del bucle. De lo contrario, intercambiamos los valores de a y b y continuamos el bucle.

Finalmente, almacenamos el valor del MCD en la memoria en la dirección MCD y salimos del programa.

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, V(1)=1, V(i)=V(i-1)+V(i-2) (donde i=0,1,2...)
Expresado en palabras, el número de Fibonacci correspondiente al elemento i es la suma
de los dos números previos de la serie.

```
# Definir el vector para almacenar los resultados
.data
V: .word 12 dup(0)

# Definir los valores iniciales para la serie de Fibonacci
fib0: .word 0
fib1: .word 1

.text
.globl _start
_start:
    # Inicializar los registros
    li t0, 0     # contador i
    la t1, V     # dirección de inicio del vector
    lw t2, fib0  # cargar el valor de V(i-1)
    lw t3, fib1  # cargar el valor de V(i-2)

fibonacci_loop:
    # Calcular el siguiente número de Fibonacci
    add t4, t2, t3   # t4 = V(i-1) + V(i-2)

    # Almacenar el resultado en el vector
    sw t4, 0(t1)     # V(i) = t4

    # Actualizar los registros para la siguiente iteración
    addi t0, t0, 1   # i++
    addi t1, t1, 4   # avanzar la dirección del vector
    mv t2, t3        # V(i-1) = V(i-2)
    mv t3, t4        # V(i-2) = t4

    # Salir del ciclo si hemos calculado los primeros 12 números
    bne t0, 12, fibonacci_loop

    # Terminar el programa
    li a7, 10        # sys_exit
    ecall
```

Este programa utiliza un ciclo para calcular los primeros 12 números de la serie de Fibonacci. En cada iteración, se calcula el siguiente número sumando los dos números previos y se almacena en el vector `v`. Luego, se actualizan los registros para la siguiente iteración y se verifica si se han calculado los primeros 12 números. Cuando se alcanza el límite, el programa termina con la llamada al sistema `sys_exit`.

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].

```
# Definir los vectores A y B
        .data
A:      .word   0, 1, 2, 7, -8, 4, 5, 12, 11, -2, 6, 3
B:      .word   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

        .text
        .align  2
        .globl  main
main:
        # Configurar el tamaño de los vectores
        li      a0, 12                  # a0 = tamaño del vector A
        li      t0, 12                  # t0 = tamaño del vector B

        # Inicializar índices y punteros
        li      t1, 0                   # t1 = índice del vector A
        li      t2, 0                   # t2 = índice del vector B
        la      t3, A                   # t3 = puntero al vector A
        la      t4, B                   # t4 = puntero al vector B

loop:
        # Salir del ciclo si se ha llegado al final del vector A
        bge     t1, a0, done

        # Cargar el valor actual de A en t5
        lw      t5, 0(t3)

        # Verificar si el valor es un número par mayor que 0
        andi    t6, t5, 1               # t6 = t5 % 2
        bnez    t6, skip                # Saltar si t5 no es par
        bge     t5, zero, store          # Saltar si t5 es negativo
skip:
        # Incrementar el índice del vector A
        addi    t1, t1, 1

        # Continuar con el siguiente valor de A
        j       loop

store:
        # Almacenar el valor en B y aumentar el índice de B
        sw      t5, 0(t4)
        addi    t2, t2, 1

        # Incrementar el índice del vector A
        addi    t1, t1, 1

        # Continuar con el siguiente valor de A
        j       loop

done:
        # Terminar el programa
```

```
        li      a7, 10                  # a7 = código de salida de programa
        ecall                           # Salir del programa
```

Este programa carga el vector A en la memoria y crea un vector B vacío de la misma longitud. Luego, itera sobre cada elemento de A y verifica si es un número par mayor que 0. Si lo es, almacena el valor en el vector B y aumenta el índice de B. Una vez que se ha recorrido todo el vector A, el programa termina. El resultado final se puede encontrar en el vector B.

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]|, i = 0,..,N-1.
Escriba un programa en ensamblador de RISC-V que calcule el nuevo vector. Utilice arrays
de 12 elementos en su programa

```
# Definición de los vectores A, B y C
    .data
A:  .word  1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23
B:  .word  2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24
C:  .word  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

    .text
    .globl  main

main:
    # Inicializar registros
    li      t0, 0               # i = 0
    li      t1, 11              # N - 1
    la      t2, A               # dirección de inicio de A
    la      t3, B               # dirección de inicio de B
    la      t4, C               # dirección de inicio de C

loop:
    # Calcular C(i) = |A[i] + B[N-i-1]|
    lw      a0, 0(t2)           # A[i]
    lw      a1, 4(t3)           # B[N-i-1]
    addi    t0, t0, 1           # i = i + 1
    addi    t2, t2, 4           # avanzar la dirección de A
    addi    t3, t3, -4          # retroceder la dirección de B
    sub     a0, a0, a1           # A[i] - B[N-i-1]
    abs     a0, a0              # |A[i] - B[N-i-1]|
    sw      a0, 0(t4)           # almacenar C(i) en su dirección correspondiente
    addi    t4, t4, 4           # avanzar la dirección de C

    # Comprobar si se ha llegado al final del vector
    bne     t0, t1, loop        # repetir si i != N - 1
```

```
    # Fin del programa
    li      a0, 10              # código de salida
    ecall                       # salir del programa
```

El programa utiliza los registros t0 a t4 para el cálculo y la iteración del vector. Primero, se inicializan los registros con los valores iniciales necesarios. Luego, se utiliza un bucle para recorrer el vector y calcular cada elemento de C. El bucle termina cuando se ha llegado al final del vector (i.e., i = N-1). Finalmente, se utiliza el código de salida 10 para terminar el programa.

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.

```
#include <stdio.h>

void bubble_sort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Intercambia los elementos adyacentes
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int arr[12] = {12, 10, 4, 6, 7, 8, 11, 1, 9, 5, 2, 3};
    int n = 12;

    printf("Array original: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
```

```
    }
    printf("\n");

    bubble_sort(arr, n);

    printf("Array ordenado: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

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.

```
# programa para calcular el factorial de n
# por multiplicaciones iterativas

.text
.globl main

main:
    # definir n
    li t0, 7       # cambiar este valor para probar otros n

    # inicializar el resultado
    li a0, 1

    # iterar a través de los números de 1 a n
    addi t1, zero, 1   # t1 = 1
loop:
    beq t1, t0, done   # salir del loop cuando t1 = n

    # multiplicar el resultado actual por el número actual
    mul a0, a0, t1

    # incrementar el número actual
    addi t1, t1, 1

    # repetir el loop
```

```
    j loop

done:
    # imprimir el resultado final
    li a7, 1     # system call para imprimir entero
    ecall

    # salir del programa
    li a7, 10    # system call para salir
    ecall
```