Boletín de problemas RISC V

Problema 1.
Suponiendo que el registro x7 contiene la dirección 0x10000000 y que en dicha dirección de memoria se ubica el dato de tamaño palabra 0x1020d040, indique la palabra almacenada en la dirección 0x10000004 tras ejecutar los siguientes pares de instrucciones:

a)
```
lb x6, 0(x7)
sw x6, 4(x7)
```

b)
```
lh x6, 0(x7)
sw x6, 4(x7)
```

c)
```
lhu x6, 0(x7)
sw x6, 4(x7)
```

Solución :

```
lb x6,0(x7) → x6=0x00000040
sw x6,4(x7) → memdat(0x10000004) = 0x00000040

lh x6,0(x7) → x6=0xffffd040
sw x6,4(x7) → memdat(0x10000004) = 0xffffd040

lhu x6,0(x7) → x6=0x0000d040
sw x6,4(x7) → memdat(0x10000004) = 0x0000d040
```

Problema 2.
Para el trozo de código que se muestra indique el contenido final de t0, t1 y t2.

```
li   t0, 4
slli t1, t0, 3
addi t2, t1, -5
add  t0, t2, t0
```

Solución:

```
li   t0, 4       →    t0=4
slli t1, t0, 3   → desplazamiento a izquierda de 3 bits (multiplicar por 8)
                   t1 = 4 x 8 = 32
addi t2, t1, -5  → t2 = 32 – 5 = 27
add  t0, t2, t0  → t0 = 27 + 4 = 31
```

Problema 3.
Para el trozo de código que se muestra y suponiendo los valores de a0 y memoria que se dan, indique el contenido final de t0 y de la memoria.

a0 = 0x2000
Memoria:
[0x2000] = 5
[0x2004] = 8
[0x2008] = 12

Código:

```
lw   t0, 4(a0)
addi t0, t0, 2
sw   t0, 8(a0)
lw   t0, 8(a0)
```

Solución:

```
lw   t0, 4(a0)  → dirección: 0x2000 + 4 = 0x2004
                  t0 = datmem[0x2004] = 8

addi t0, t0, 2  → t0 = 8 + 2 = 10

sw   t0, 8(a0)  → dirección: 0x2000 + 8 = 0x2008
                  datmem[0x2008] = 10

lw   t0, 8(a0) → dirección: 0x2000 + 8 = 0x2008
                 t0 = datmem[0x2008] = 10
```

Problema 4.
Explique qué hace este código

```
lw t0, 0(a0)
lw t1, 4(a0)
sw t0, 4(a0)
sw t1, 0(a0)
```

Solución:

Intercambia dos palabras en la memoria de datos, primero lee una palabra de la posición indicada por a0 y la almacena en t0, luego lee la siguiente palabra y la almacena en t1, posteriormente las escribe en orden inverso: t1 en la posición indicada por a0 y t0 en la siguiente posición.

Problema 5.
Siempre que no de error de ensamblado, indique el contenido del registro x10 después de ejecutar las siguientes parejas de instrucciones e indique una pseudoinstrucción equivalente:

```
a)lui x10,0x12345
  addi x10,x10,0x678
b)lui x10,0x12345
  addi x10,x10,0x876
c)lui x10,1
  addi x10,x10,-0x544
d)lui x10,0xffed8
  addi x10,x10,-0x679
```

Solución:

a)

```
lui x10,0x12345
addi x10,x10,0x678   →  x10=0x12345678
```

la pseudoinstrucción sería li x10,0x12345678

b)

```
lui x10,0x12345
addi x10,x10,0x876   → error, 0x876 excede los límites de addi [-0x7ff,+0x800]
```

podríamos hacerlo expresándolo como negativo: 0x876 = -0x78a que si está en rango, entonces:

```
lui x10,0x12345
addi x10,x10,-0x78a
```

x10 = 0x12345000 – 0x0000078a = 0x12344876
o bien x10 = 0x12345000 + 0xfffff876 = 0x12344876
en cualquier caso la pseudoinstrucción será li x10,0x12344876

c)

```
lui x10,1
addi x10,x10,-0x544
```

-0x544 = 0xfffffabc → x10 = 0x00001000 + 0xfffffabc = 0x00000abc
pseudoinstrucción: li x10,0xabc

d)

```
lui x10,0xffed8
addi x10,x10,-0x679
```

-0x679 = 0xfffff987 → x10 = 0xffed8000 + 0xfffff987 = 0xffed7987
pseudoinstrucción: li x10,0xffed7987

Problema 6.
Proponga una secuencia de instrucciones nativas de la ISA RISC-V, equivalentes a las siguientes pseudoinstrucciones:

```
li x10, 0xcaba56ff
li x10, 0x1abc
li x10, 0x56789abc
li x10, 0x1d3
li x10, 0x800
```

Solución:

```
li x10, 0xcaba56ff   →     lui x10, 0xcaba5
                           addi x10, x10, 0x6ff

li x10, 0x1abc       →     lui x10, 0x2
                           addi x10, x10, -0x544

li x10, 0x56789abc   →     lui x10, 0x5678a
                           add x10, x10, -0x544

li x10, 0x1d3        →     add x10, zero, 0x1d3

li x10, 0x800        →     lui x10, 0x1
                           addi x10, x10, -0x800
```

Problema 7.
Proponga una secuencia de instrucciones nativas de la ISA RISC-V, equivalentes a las siguientes pseudoinstrucciones:

mv t0, t1
li t0, 12
j etiqueta
ret
neg t0, t1
not t0, t1
call funcion
nop (no operación)

Solución:

```
mv t0, t1   →    add t0, t1, zero
li t0, 12   →    addi t0, zero, 12
j etiqueta  →    jal zero, etiqueta   o beq zero, zero, etiqueta
ret         →    jalr zero, x1, 0    (x1 es ra)
neg t0, t1  →    sub t0, zero, t1
```

```
not t0, t1    →    xori t0, t1, -1
call funcion  →  jal ra, funcion
nop           →    addi zero, zero, 0
```

Problema 8.
Analice el siguiente código y diga la función realizada.

```
    .data
X: .word 7
Y: .word 8
Z: .word 0

.text
la t0, X
lw t1, 0(t0)
lw t2, 4(t0)
add a0, t1, t2
sw a0, 8(t0)
```

Solución:
El programa inicializa dos datos en la memoria de datos, los suma y almacena el resultado en otra dirección de la memoria de datos

`.data` → determina el inicio del segmento de datos
(por defecto en la dir. 0x0. – no es así en rars debido al include hr1bs3.s)

```
X: .word 7
Y: .word 8
Z: .word 0
```
→ se inicializan tres valores tamaño palabra (en 0x0 el valor 7, en 0x4 el valor 8, en 0x8 el valor 0)

```
.text

la t0, X          → pseudoinstrucción "la" : almacena la dirección de X en t0 (x5)
lw t1, 0(t0)      → carga el primer dato en t1(x6)
lw t2, 4(t0)      → carga el segundo dato en t2(x7)
add a0, t1, t2    → suma ambos datos y almacena el resultado en a0 (x10)
sw a0, 8(t0)      → carga el resultado en la siguente dirección (Z)
```

Problema 9.
Escriba un programa en lenguaje ensamblador del RISC-V que implemente el siguiente bloque de código.
Use la sección .data para asignar el valor inicial a las variables.

```
int x = 10, y = 5;
if (x >= y) {
        x = x + 2;
        y = y - 2;
        }
```

Solución:

```
    .data
X:  .word   10
Y:  .word   5

    .text
    la      t0, X
    lw      s1, 0(t0) # X
    lw      s2, 4(t0) # Y
If:
    bge     s1, s2, Then        # alternativa:          blt  s1, s2, End
Else:                                                 addi s1, s1, 2
    j       End                                       addi s2, s2, -2
Then:                                           End :  sw   s1, 0(t0) # X
    addi    s1, s1, 2                                  sw   s2, 4(t0) # Y
    addi    s2, s2, -2
End:
    sw      s1, 0(t0) # X
    sw      s2, 4(t0) # Y

fin: j fin
```

Problema 10.

Escriba un programa en lenguaje ensamblador del RISC-V que implemente el siguiente bloque de código.

Use la sección .data para asignar el valor inicial a las variables.

```
int x = 5, y = 10;
if (x >= y) {
        x = x + 2;
        y = y + 2;
        }
else {
        x = x - 2;
        y = y - 2;
        }
```

```
Solución:
    .data
X:  .word   5
Y:  .word   10

    .text
    la      t0, X
    lw      s1, 0(t0) # X
    lw      s2, 4(t0) # Y
If:
    bge     s1, s2, Then
Else:
    addi    s1, s1, -2
    addi    s2, s2, -2
    j       End
Then:
    addi    s1, s1, 2
    addi    s2, s2, 2
End:
    sw      s1, 0(t0) # X
    sw      s2, 4(t0) # Y

fin: j fin
```

Problema 11.
Escriba un programa en lenguaje ensamblador del RISC-V que implemente el siguiente bloque de código. Use la sección .data para reservar espacio en memoria para las variables.

```
int a, b;
a = 81;
b = 18;
do {
    a = a - b;
    } while (a > 0);
```

```
Solución:
    .data
A:  .space  4
B:  .space  4

    .text
    la      t0, A
    li      s1, 81
    li      s2, 18
    sw      s1, 0(t0) # A
    sw      s2, 4(t0) # B
```

```
Do:
    sub     s1, s1, s2
    bgt     s1, zero, Do

End:
    sw      s1, 0(t0) # A

fin: j fin
```

Problema 12.
Escriba un programa en lenguaje ensamblador del RISC-V que implemente el siguiente bloque de código. Use la sección .data para reservar espacio en memoria para las variables.

```
int n, fprev, f, i;
n = 5;
fprev = 1;
f = 1;
i = 2;
while (i <= n) {
        faux = f;
        f = f + fprev;
        fprev = faux;
        i = i + 1;
        }
```

Solución:

```
       .data
N:     .space  4
Fprev: .space  4
F:     .space  4
I:     .space  4

    .text
    la      t0, N
    li      s1, 5
    li      s2, 1
    li      s3, 1
    li      s4, 2

    sw      s1, 0(t0) # N
    sw      s2, 4(t0) # Fprev
    sw      s3, 8(t0) # F
    sw      s4, 12(t0) # I
```

```
While:
    bgt     s4, s1, End       # fin de bucle cuando i>n
    mv      t1, s3     # Faux
    add     s3, s3, s2
    mv      s2, t1
    addi    s4, s4, 1
    j       While
End:

    sw      s1, 0(t0) # N
    sw      s2, 4(t0) # Fprev
    sw      s3, 8(t0) # F
    sw      s4, 12(t0) # I

fin: j fin
```

Problema 13.
Escriba un programa en lenguaje ensamblador del RISC-V que implemente el siguiente bloque de código.

```
int f = 2, n = 5;
int i;
for (i = 2; i <= n; i++)
      f = f + f;
```

Solución:

```
    .data
F:  .word   2
N:  .word   5
I:  .space  4       #no necesario

    .text
    la      t0, F
    lw      s1, 0(t0) # F
    lw      s2, 4(t0) # N

For:
    li      t1, 2
Loop:
    bgt     t1, s2, End
    add     s1, s1, s1
    addi    t1, t1, 1
    j       Loop
End:
```

```
    sw      s1, 0(t0) # F
    sw      t1, 8(t0) # I   #no necesario

fin: j fin
```

Problema 14.
Una tabla de 100 números enteros está almacenada a partir de la dirección 0x0. Escriba un programa que almacene en x5 el número de datos positivos que hay en dicha tabla, en x6 el número de datos negativos y en x7 el número de elementos iguales a cero.

Solución:

```
        .text
main:
        # Inicialización
        li t3, 0x0        # t3 (x28) → puntero a la tabla
        li t4, 100        # t4 (x29) → contador de elementos

        li x5, 0          # x5 → positivos
        li x6, 0          # x6 → negativos
        li x7, 0          # x7 → ceros

loop:
        lw t5, 0(t3)      # t5 (x30) = dato actual

        beq t5, x0, es_cero
        blt t5, x0, es_negativo

        # positivo
        addi x5, x5, 1
        j siguiente      # pseudoinstrucción j ⇔  jal x0,siguiente

es_negativo:
        addi x6, x6, 1
        j siguiente

es_cero:
        addi x7, x7, 1

siguiente:
        addi t3, t3, 4    # avanzar puntero
        addi t4, t4, -1   # decrementar contador
        bne t4, x0, loop

        li a7, 10    # Fin del programa
        ecall
```

Problema 15.
Escriba una subrutina que busque el menor de una tabla de números enteros. La subrutina recibirá como argumento de entrada la dirección de comienzo de la tabla y el número de elementos de esta y devolverá el valor del elemento menor de la tabla y el número de orden donde se encuentra.
Compruebe el correcto funcionamiento de la subrutina con una tabla de 16 números enteros definida en tiempo de compilación en la sección de datos.

Solución:
Llamaremos a la subrutina mínimo_tabla, los argumentos, según enunciado, son:
Argumentos de entrada:
- dirección base de la tabla, consideraremos que se pasa en a0
- número de elementos, consideraremos que se pasa en a1

La salida se devolverá en a0 y a1:
a0 = valor mínimo
a1 = índice del mínimo

```
minimo_tabla:
    mv t0, a0       # t0: puntero al elemento a analizar
    mv t1, a1       # t1: contador de elementos

    lw a0, 0(t0)    # el primer elemento es el mínimo actual
    li a1, 0        # índice del mínimo actual
    li t3, 0        # índice del elemento analizado

bucle:
    addi t1,t1,-1        # decremento contador (hasta que llegue a 0)
    beq t1,zero,fin

    addi t0,t0,4        # incremento puntero
    addi t3,t3,1        # incremento índice

    lw t2,0(t0)
    bgt a0,t2,cambia
    j bucle

cambia:
    mv a1,t3
    mv a0,t2
    j bucle

fin:
    ret


ejemplo de uso con 6 elementos (igual sería para 16)
```

```
.data
tabla:        .word 5, -3, 12, 0, -7, 9
resultado:    .space 4

.text
       la a0, tabla   # dirección de la tabla
       li a1, 6       # número de elementos
       call minimo_tabla
       la t0, resultado
       sw a0,0(t0)

       li a7, 10     # Fin del programa
       ecall
```

Problema 16.
Escriba una subrutina que traslade una tabla de 20 datos tamaño word. La subrutina recibirá como argumentos (en este orden) la dirección de comienzo de la tabla y la dirección a la que se la quiere trasladar.
Compruebe el correcto funcionamiento de la subrutina anterior con una tabla de 16 números enteros almacenada en la sección de datos.

Solución:

```
traslada_tabla:
        li   t0, 20          # contador = 20 elementos

bucle:
        beq  t0, zero, fin

        lw   t1, 0(a0)       # leer word origen
        sw   t1, 0(a1)       # escribir word destino

        addi a0, a0, 4       # avanzar origen
        addi a1, a1, 4       # avanzar destino
        addi t0, t0, -1      # contador--

        j    bucle

fin: ret

ejemplo para tabla de 16 elementos

        .data
tabla_origen:
        .word 1, 2, 3, 4, 5, 6, 7, 8
        .word 9,10,11,12,13,14,15,16
```

```
tabla_destino:
        .space 64            # 16 words = 64 bytes

        .text

start:
        la   a0, tabla_origen
        la   a1, tabla_destino

        call  traslada_tabla

fin:   li a7, 10     # Fin del programa
       ecall
```

Problema 17.
Escriba una subrutina que invierta el orden de datos de una tabla datos tamaño word de longitud arbitraria. La subrutina recibirá como argumentos (en este orden) la dirección de comienzo de la tabla y el número de elementos.
Compruebe el correcto funcionamiento de la subrutina anterior con una tabla de 16 números enteros almacenada en la sección de datos.

Solución:
El enunciado dice que

- a0 = dirección base de la tabla
- a1 = número de elementos

Se quiere invertir la tabla:

```
     tabla[0]  ⇔ tabla[n-1]
     tabla[1]  ⇔ tabla[n-2]
     ...
```

Solo hace falta recorrer la mitad de la tabla.

```
invertir_tabla:
        mv t0, a0            #pseudoinstrucción para t0 <- a0
        addi t1, a1, -1      #t1 = N-1
        slli t1, t1, 2       #t1 = 4*(N-1) (ya que son tamaño word)
        add t1, t1, a0       # dirección del último elemento

bucle:
        bge t0, t1, fin     # mientras inicio < final
        # intercambiar elementos
        lw   t4, 0(t0)       # temp4 = tabla[i]
        lw   t5, 0(t1)       # temp5 = tabla[j]
        sw   t5, 0(t0)       # tabla[i] = temp5 = tabla[j]
        sw   t4, 0(t1)       # tabla[j] = temp4 = tabla[i]
```

```
        addi t0, t0, 4       # i++
        addi t1, t1, -4      # j--
        j    bucle

fin:
        ret


ejemplo de uso con tabla de 16 elementos:

        .data
tabla:
        .word 1, 2, 3, 4, 5, 6, 7, 8
        .word 9,10,11,12,13,14,15,16

        .text

start:
        la   a0, tabla       # a0 = dirección base de la tabla
        li   a1, 16          # a1 = número de elementos

        jal invertir_tabla   # llamada a la subrutina

fin:
        li a7, 10    # Fin del programa
        ecall
```

Problema 18.
Escriba una subrutina que calcule el valor absoluto de su argumento. Utilice esta subrutina para calcular mediante otra subrutina la suma de los valores absolutos de los elementos de una tabla. Esta segunda subrutina recibirá como argumentos (en este orden) la dirección de comienzo de la tabla y el número de elementos.

Solución:

1)Subrutina abs

```
abs:
        bge a0, zero, fin_abs     # zero es x0 (= 0), se comprueba si X > 0
        sub a0, zero, a0          # si X<0 lo cambiamos de signo haciendo 0-X
fin_abs:
        ret
```

2)Subrutina suma_abs

Se trata de una subrutina que va a llamar a otra subrutina, es decir es una subrutina no hoja, y por tanto habrá que utilizar la pila para preservar la dirección de retorno de *suma_abs*, que va a ser sobreescrita al llamar a la subrutina *abs*.
También al ser una subrutina no hoja, se debe usar registros salvados o guardar los t en la pila antes de llamar a la subrutina abs pues no se debe asumir que los registros t no se modifican.
La solución más limpia es usar registros s en la subrutina no hoja (suma_abs).

El enunciado dice que

- a0 = dirección base de la tabla
- a1 = número de elementos

```
suma_abs:
        addi sp, sp, -16      # reservar pila
        sw   ra, 12(sp)       # salvar ra  (dirección de retorno de suma_abs)
        sw   s0, 8(sp)        # salvar s0
        sw   s1, 4(sp)        # salvar s1
        sw   s2, 0(sp)        # salvar s2

        mv   s0, a0          # s0 = puntero tabla
        li   s1, 0           # s1 = suma acumulada
        li   s2, 0           # s2 = contador de elementos

bucle:
        beq  s2, a1, fin     #se repite hasta que s2 = N

        lw   a0, 0(s0)       # a0 = tabla[i] (a0 es el argumento de abs)
        jal  abs             # llamada → ra protegido

        add  s1, s1, a0      # suma = suma + |tabla[i]|
        addi s0, s0, 4       # avanzar puntero
        addi s2, s2, 1       # i++
        j    bucle

fin:
        mv   a0, s1          # valor retorno

        lw   s2, 0(sp)        # restaurar registros
        lw   s1, 4(sp)
        lw   s0, 8(sp)
        lw   ra, 12(sp)      # restaurar ra
        addi sp, sp, 16
        ret
```

Problema 19.
Escriba un programa en ensamblador del RISC-V que llame a una subrutina swap

encargada de intercambiar el contenido de dos posiciones de memoria. La subrutina recibirá como parámetros de entrada las posiciones de memoria correspondiente a las variables a y b y deberá preservar el contenido de todos los registros que obligue el estándar de llamadas estudiado en clase.

Solución:

- Entrada:
- a0 = dirección de a
- a1 = dirección de b

La subrutina es hoja y utiliza solo registros temporales, por tanto, no es necesario preservar nada en la pila.

```
swap:
    lw   t0, 0(a0)      # t0 = *a
    lw   t1, 0(a1)      # t1 = *b

    sw   t1, 0(a0)      # *a = antiguo *b
    sw   t0, 0(a1)      # *b = antiguo *a
    ret
```

Problema 20.
Utilizando la subrutina del problema anterior realice otra subrutina llamada swapTable que intercambie dos tablas que se encuentran en memoria. Esta subrutina recibe como argumentos el tamaño de las tablas en a0, y la dirección de comienzo de las dos tablas en a1 y a2.

Solución:
Como se trata de una subrutina no hoja, se usarán registros saved, que hay que preservar en la pila al comenzar.

```
swapTable:
        addi sp, sp, -16      # reservar pila
        sw   ra, 12(sp)       # salvar ra  (dirección de retorno de swapTable)
        sw   s0, 8(sp)        # salvar s0
        sw   s1, 4(sp)        # salvar s1
        sw   s2, 0(sp)        # salvar s2

        li s0,0             # contador
        mv s1,a1            # puntero tabla 1
        mv s2,a2            # puntero tabla 2

buc_swT:
        bge s0,a0,fin
        mv a0,s1          # argumento para swap (dirección elemento tabla 1)
        mv a1,s2          # argumento para swap (dirección elemento tabla 2)
```

```
        call swap

        addi s1,s1,4
        addi s2,s2,4
        addi s0,s0,1
        j buc_swT

fin:    lw   s2, 0(sp)        # restaurar registros
        lw   s1, 4(sp)
        lw   s0, 8(sp)
        lw   ra, 12(sp)      # restaurar ra
        addi sp, sp, 16
        ret
```

Problema 21.
Escriba un programa para el ensamblador del RISC-V que cuente el número de 0 de un vector de longitud arbitraria. Emplee para ello una subrutina llamada cuenta_ceros que reciba como parámetros de entrada toda la información necesaria para llevar a cabo la tarea.

```
En primer lugar escribimos la subrutina que cuenta el número de elementos que
son 0 en un vector de una determinada longitud: cuenta_ceros
Argumentos:
      a0 -> dirección base del vector
      a1 -> longitud del vector
Resultado:
      a0 -> número de ceros encontrados

Al ser una subrutina hoja se usan los registros t.

cuenta_ceros:
       li t0, 0      # contador de ceros
       li t1, 0      # contador de elementos
       mv t2, a0     # puntero al vector

bucle:
       bge t1, a1, fin # si t1 >= longitud del vector -> terminar
       lw t3,0(t2)
       bne t3,zero,siguiente
       addi t0,t0,1

siguiente:
       addi t2,t2,4
       addi t1,t1,1
       j bucle
```

```
fin:    mv a0,t0
        ret
```

Ahora escribimos el programa principal que hará uso de la subrutina

```
.data
vector:         .word 3, 0, 5, 0, 0, 8, 2
longitud:       .word 7
resultado:      .space 4

.text

main:
        la a0, vector           # Cargar dirección del vector en a0
        lw a1, longitud         # Cargar longitud del vector en a1

        call cuenta_ceros       # Llamar a la subrutina
        la t0, resultado        # Guardar el resultado
        sw a0, 0(t0)

        li a7, 10       # Fin del programa
        ecall
```

Problema 22.
Realice una subrutina llamada prod_escalar que calcule el producto escalar de dos vectores. Para ello, se utilizará otra subrutina (Muls) que debe realizar la multiplicación de números enteros con signo.
Compruebe el correcto funcionamiento de la subrutina prod_escalar con dos vectores de 10 elementos almacenados en la sección de datos.

Solución:

El producto escalar de dos vectores de n componentes se calculará así:
$P = v1 \cdot v2 = v1[0]v2[0] + \cdots + v1[n-1]v2[n-1]$

```
Argumentos para la subrutina prod_escalar:
      a0 -> longitud de los vectores
      a1 -> dirección base del vector v1
      a2 -> dirección base del vector v2
Resultado:
      a0 -> resultado
```

Como es una subrutina no hoja, dado que usará llamadas a la subrutina Muls, se usarán los registros s.

```
Argumentos para la subrutina Muls:
      a0 -> factor
      a1 -> factor
Resultado:
      a0 -> producto
Como es una subrutina hoja usará los registros t.


prod_escalar:
       addi sp, sp, -32      # reservar pila
       sw   ra, 28(sp)       # salvar ra (dirección de retorno de prod_escalar)
       sw   s0, 24(sp)       # salvar s0
       sw   s1, 20(sp)       # salvar s1
       sw   s2, 16(sp)       # salvar s2
       sw   s3, 12(sp)       # salvar s3

       li s0,0       # para acumular las sumas v1[i]v2[i]
       mv s1,a1      # punteros a v1 y v2
       mv s2,a2
       mv s3,a0      # número de componentes

bucle:
       beq s3,x0,fin
       lw a0,0(s1)          #la subrutina Muls espera sus argumentos en a0 y a1
       lw a1,0(s2)

       call Muls

       add s0,s0,a0         #la subrutina Muls da su resultado en a0

       addi s1,s1,4 #actualizo puntero a v1
       addi s2,s2,4 #actualizo puntero a v2
       addi s3,s3,-1 #decremento contador
       j bucle

fin:   mv a0,s0
       lw ra, 28(sp)
       lw s0, 24(sp)
       lw s1, 20(sp)
       lw s2, 16(sp)
       lw s3, 12(sp)
       addi sp, sp, 32
       ret


Muls:
       li t0,0    # para acumular el res
       li t1,0    # información de signo (0=positivo, 1=negativo)
```

```
       bge a0, x0, check_a1        # si a0 ≥ 0 saltamos a ver el signo de a1
       sub a0, x0, a0              # a0 <0, lo cambio de signo (↔ neg a0,a0)
       addi t1, t1, 1              # almaceno información de signo

check_a1: # vemos si a1 < 0
      bge a1, x0, bucle       # los dos son positivos, salto a bucle de suma
       sub a1, x0, a1          # a1 <0, lo cambio de signo (↔ neg a0,a0)
       xori t1, t1, 1          # cambio información de signo al valor contrario

bucle:                         #sumamos a0, a1 veces
       beq a1, x0, fin
       add t0, t0, a0
       addi a1, a1, -1
       j bucle
fin:
       beq t1, x0, devolver    #los dos signos eran iguales (t1 == 0)
       sub t0, x0, t0   #los dos signos eran distintos (t1 == 1)(↔ neg t0,t0)

devolver:
       mv a0, t0
      ret
```

Prueba para dos vectores de 10 elementos almacenados en la sección de datos:

```
    .data
v1:         .word 1, 5, -2, -5, 1, -1, 0, 3, 1, 4
v2:         .word -3, 2, -2, 0, 1, -2, 0, -5, 1, 2
resultado: .space 4

    .text
       li a0,10
       la a1,v1 #cargo dirección de v1
       la a2,v2 #cargo dirección de v2
       la t0,resultado

       call prod_escalar

       sw a0,0(t0)

       li a7, 10     # Fin del programa
       ecall
```

Problema 23.
Realice la subrutina Muls del programa anterior usando una subrutina llamada mul que multiplica dos números enteros positivos.

```
Argumentos para la subrutina Muls:
      a0 -> factor
      a1 -> factor
Resultado:
      a0 -> producto
Como ahora es una subrutina no hoja usará los registros s.

Argumentos para la subrutina mul:
      a0 -> factor
      a1 -> factor
Resultado:
      a0 -> producto
Como es una subrutina hoja usará los registros t.


Muls:
        addi sp, sp, -16      # reservar pila
        sw   ra, 12(sp)       # salvar ra  (dirección de retorno de Muls)
        sw   s1, 8(sp)        # salvar s1

       li s1,0    # información de signo (0=positivo, 1=negativo)

       bge a0, x0, check_a1        # si a0 ≥ 0 saltamos a ver el signo de a1
       sub a0, x0, a0              # a0 <0, lo cambio de signo
       addi s1, s1, 1              # almaceno información de signo

check_a1: # vemos si a1 < 0
       bge a1, x0, multiplica    # los dos son positivos
       sub a1, x0, a1          # a1 <0, lo cambio de signo
       xori s1, s1, 1          # cambio información de signo

multiplica:
       call mul

       beq s1, x0, fin        #los dos signos eran iguales
        sub a0, x0, a0         #los dos signos eran distintos

fin:
        lw   s1, 8(sp)      # restaurar registros
        lw   ra, 12(sp)      # restaurar ra
        addi sp, sp, 16
        ret
```

```
mul:
        mv t0,zero
loopm:
        beq a0,zero,endm
        add t0,t0,a1
        addi a0,a0,-1
        j loopm
endm:
        mv a0,t0
        ret
```