EdC-TI Grupo 1 **Prueba 2** **Curso 25/26**

ALUMNO:_____

Ejercicio 1 (5 puntos)

Para el procesador RISC V 32I:

1) Si x2= 0x10010020, ¿a qué dirección de memoria se accede con sw x8,-12(x2)?

x2 = 0x10010020, offset = -12 (decimal) = 0xFFFFFFF4 en complemento a 2, pero lo que importa es:
Dirección = 0x10010020 + (-12) = 0x10010020 - 0xC = 0x10010014

2) Suponga que x3 = -1 y x4 =1, explique si las siguientes instrucciones producen salto o no.
   a. blt x3,x4,etiqueta

   comparación con **signo**
   x3 = -1, x4 = 1 → -1 < 1 → **SÍ produce salto**

   b. bltu x3,x4,etiqueta

   comparación **sin signo**
   x3 = -1 → en binario 32 bits = 0xFFFFFFFF =
   x4 = 1
   2^{32}-1 < 1 → **falso** → **NO produce salto**

3) ¿Qué tipo de estructura de control típica de los lenguajes de alto nivel realizan las siguientes líneas de ensamblador? Escriba el código fuente asociado.

```
loop:
        add x1, x1, x2
        blt x1, x3, loop
```

Es un bucle do-while, porque el cuerpo se ejecuta al menos una vez antes de comprobar la condición.
Por ejemplo, en C:

```
do {
   x1 = x1 + x2;
} while (x1 < x3);
```

4) Si x10=0x1000 y se ejecutan las siguientes líneas de programa, ¿qué bytes de la memoria se modifican? Exprese la dirección de cada uno de dichos bytes y su contenido.

```
li x2,0x12345678
sw x2,0(x10)
```

sw escribe una palabra (4 bytes) en little-endian a partir de 0x1000:

0x1000 → 0x78
0x1001 → 0x56
0x1002 → 0x34
0x1003 → 0x12

5) Suponga las siguientes direcciones y contenidos de la memoria de datos y, para cada apartado, conteste qué valor toman los registros implicados:

0x1000 → 0xAA
0x1001 → 0xBB
0x1002 → 0xCC
0x1003 → 0xDD

a. lw x1,0x1000(x0)
carga 4 bytes en little-endian:
Bytes: DD CC BB AA → x1 = 0xDDCCBBAA

b. lh x2,0x1002(x0)
carga 2 bytes con extensión de signo:
Bytes en 0x1002-0x1003: CC DD → valor = 0xDDCC
0xDDCC en 16 bits tiene el bit de signo a 1 → extensión de signo
x2 = 0xFFFFDDCC

c. lbu x3,0x1001(x0)
carga 1 byte sin extensión de signo:
Byte en 0x1001: 0xBB
x3 = 0x000000BB

6) Suponga las siguientes direcciones y contenidos de la memoria de datos y el valor de los registros que se muestran y, para cada apartado, conteste qué posiciones de dicha memoria se modifican y qué valor toman.

0x1000 → 0x78
0x1001 → 0x56
0x1002 → 0x34
0x1003 → 0x12

x1=0x01002
x2=0xbadcfe21
x3=0x789abcde

a. sw x1,-2(x1)

Dirección = 0x1002 - 2 = 0x1000
Escribe x1 = 0x00001002 en 4 bytes little-endian:
0x1000 → 0x02
0x1001 → 0x10
0x1002 → 0x00
0x1003 → 0x00

b. sh x2,0(x1)

Dirección = 0x1002 + 0 = 0x1002
Escribe los 2 bytes bajos de x2 = 0xFE21 en little-endian:
0x1002 → 0x21
0x1003 → 0xFE

c. sb x3,1(x1)

Dirección = 0x1002 + 1 = **0x1003**
Escribe el byte bajo de x3 = 0xDE:
0x1003 → 0xDE

7) Explique qué es el alineamiento en memoria. ¿Qué alineamiento necesitan los bytes, las medias palabras y las palabras?

El alineamiento es la restricción por la que un dato de N bytes debe almacenarse en una dirección múltiplo de N. Esto permite al hardware acceder eficientemente sin tener que leer varias líneas de memoria.

Byte (1 byte): cualquier dirección → sin restricción
Media palabra / halfword (2 bytes): dirección múltiplo de 2 (bit 0 = 0)
Palabra / word (4 bytes): dirección múltiplo de 4 (bits 1:0 = 00)

8) Considere la pseudoinstrucción li x2, 0xbadcfe21 y dé su equivalente con instrucciones nativas del RISCV32I

li con una constante de 32 bits se descompone en lui + addi. Una cuestión importante es que addi extiende el signo de los 12 bits bajos, así que hay que ajustar la parte alta si la parte baja tiene el bit 11 a 1.

Parte baja: 0xE21 = 1110 0010 0001 → bit 11 = 1 → hay que sumar 1 a la parte alta para compensar
Parte alta original: 0xBADCF → ajustada: 0xBADD0

```
lui x2, 0xBADD0       # x2 = 0xBADD0000
addi x2, x2, -0x1DF   # -0x1DF=0xE21 se extiende como negativo: 0xFFFFF E21
```

Verificación: 0xBADD0000 + 0xFFFFFE21 = 0xBADCFE21 ✓
Verificación: 0xBADD0000 - 0x1DF = 0xBADCFE21 ✓

Ejercicio 2 (5 puntos)

a. Escriba una subrutina (mul) que realice la multiplicación de números enteros sin signo. Utilice esta subrutina para realizar otra, llamada Nfactorial, que calcule el factorial de un número entero positivo.

La idea para el factorial es:
```
resultado = 1
mientras contador != 0:
resultado = contador * resultado
contador—
```

Subrutina mul:
es una subrutina hoja, no llamará a ninguna otra subrutina, usará registros t.

Argumentos: a0, a1 (factores)
Resultado: a0 (producto)
Sumamos a1 tantas veces como indica a0

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

Subrutina Nfactorial:
es una subrutina no-hoja, llamará a la subrutina mul, usará registros s.

Argumentos: a0 = N (factor)
Resultado: a0 = N! (factorial)

```
Nfactorial:
        addi sp, sp, -16
        sw   ra, 12(sp)
        sw   s0, 8(sp)

        mv   s0, a0         # s0 = N (contador, empieza en N)
        li   a0, 1          # a0 = resultado 0! = 1

Nfact_loop:
        beq  s0, zero, Nfact_end  # si contador == 0, fin
        mv   a1, a0         # a1 = resultado acumulado hasta ahora, argumento para mul
        mv   a0, s0         # a0 = contador actual, argumento para mul

        call mul            # a0 =nuevo resultado hasta ahora

        addi s0, s0, -1     # contador--
        j    Nfact_loop

Nfact_end:
        lw   s0, 8(sp)
        lw   ra, 12(sp)
        addi sp, sp, 16
        ret
```

b. Escriba un programa que utilice la subrutina Nfactorial y calcule el factorial de un número almacenado en la sección de datos de la memoria y almacene a continuación el resultado.

```
    .data
N:        .word 5     # número del que calcular el factorial
result:   .space 4    # aquí se guardará el resultado

.text
   la   s0, N         # dirección de N
   lw   a0, 0(s0)     # a0 = N

   call Nfactorial    # a0 = N!

   la   s0, result    # dirección de result
   sw   a0, 0(s0)     # almacena el resultado

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