DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

**IIC2343 - Arquitectura de Computadores (II/2025)**

# Actividad de programación

**Sección 1 - Pauta de evaluación**

## Pregunta 1: Explique el código (3 ptos.)

En el siguiente fragmento de código se realiza el llamado de una subrutina `func_n_m`:

```
.data
n:   .word 37
m:   .word 5
q:   .word 0

.text
main:
  addi sp, sp, -16
  sw   ra, 12(sp)

  la   t0, n
  lw   a0, 0(t0)
  la   t1, m
  lw   a1, 0(t1)

  jal  ra, func_n_m

  la   t2, q
  sw   a0, 0(t2)

  lw   ra, 12(sp)
  addi sp, sp, 16
  addi a7, zero, 10
  ecall

func_n_m:
  blt  a0, a1, ccc

  addi sp, sp, -16
  sw   ra, 12(sp)

  sub  a0, a0, a1
  jal  ra, func_n_m
  addi a0, a0, 1

  lw   ra, 12(sp)
  addi sp, sp, 16
  jalr x0, 0(ra)

ccc:
  addi a0, x0, 0
  jalr x0, 0(ra)
```

Este fragmento representa el cómputo de una función $f(n, m)$. A partir de este:

1. (1.5 ptos.) Indique, con argumentos y en términos de $n$ y $m$, lo que retorna la función $f(n, m)$. Por ejemplo, $f(n, m) = n + m$. Se otorgan **0.75 ptos.** por la correctitud de la descripción del retorno y **0.75 ptos.** por justificación.

   **Solución:** La función implementa la **división entera por restas sucesivas**, por lo que:

   $$f(n, m) = n//m$$

   En el código, el registro `a0` contiene el argumento $n$ y el registro `a1` contiene el argumento $m$. La subrutina `func_n_m` resta repetidamente $m$ a $n$ hasta que el valor de $a0$ es menor que $a1$, alcanzando el caso base. En cada retorno desde la recursión se incrementa en una unidad el valor acumulado en `a0`, representando la cantidad de veces que $m$ cabe en $n$. De esta forma, el resultado final (el cociente entero de la división) queda almacenado en `a0`, y posteriormente se guarda en la etiqueta `q`.

2. (1.5 ptos.) Indique, con argumentos, si el fragmento anterior respeta o no la convención de llamadas de RISC-V. Se otorgan **0.75 ptos.** si indica de forma correcta si se respeta o no la convención y **0.75 ptos.** por entregar una justificación válida respecto a su respuesta.

   **Solución:** El fragmento anterior **NO respeta** la convención de llamadas de RISC-V. Aunque el programa ejecuta correctamente la división entera y maneja el `stack` sin errores de ejecución, desde el punto de vista de la convención formal incumple reglas esenciales. En particular:

   - El registro `a1` es de tipo *caller-saved*. La función `func_n_m` realiza un llamado recursivo sin respaldar previamente este registro, lo que constituye una violación directa a la convención. Al ser una función recursiva, actúa simultáneamente como *caller* y *callee*, por lo que debe preservar el valor de sus argumentos antes del llamado.
   - Los registros `a0` y `a1` no se respaldan en el programa principal (`main`) antes de realizar el llamado a la subrutina, a pesar de que ambos pueden ser modificados por esta. Esto también infringe la responsabilidad del *caller*.
   - Los registros `t0`, `t1` y `t2` son de tipo *caller-saved* y se utilizan en el programa principal sin respaldo previo. Aunque en este caso no generan un error funcional, formalmente representan una falta a la convención.
   - Además, el código emplea los nombres `x0` en lugar de los nombres simbólicos `zero`. Según la convención de llamadas de RISC-V, se exige el uso de nombres simbólicos estandarizados (`zero`, `ra`, `sp`, `a0-a7`, etc.) en lugar de los nombres numéricos (`x0`, `x1`, ...). Este uso constituye una infracción formal de la convención, aunque no afecte la ejecución.

## Pregunta 2: Elabore el código (3 ptos.)

Elabore, utilizando el Assembly RISC-V, un programa que, dado un arreglo `arr` de largo `len` cuyos elementos son enteros en el rango $[0, \texttt{len} - 1]$ sin repetición (es decir, una permutación), calcule la longitud del camino más largo sin ciclos partiendo desde la posición `0` y guarde el resultado en la variable `path_len`. En cada paso, si está en el índice `i`, debe ir a la posición `arr[i]` del mismo arreglo. El conteo se detiene al intentar visitar una posición ya visitada.

Puede utilizar el siguiente fragmento de código como base:

```
.data
    arr:       .word 3,0,1,2  # Arreglo
    len:       .word 0        # Largo del arreglo
    path_len:  .word -1       # Camino más largo
.text
  # Su código aquí
```

**Solución:** El programa analiza el contenido del arreglo `arr`, donde cada posición indica el índice siguiente a visitar. El propósito es determinar la longitud máxima del recorrido antes de que una posición ya visitada se repita, almacenando el resultado en la variable `path_len`.

- Inicializa `path_len = -1` y crea en el stack un arreglo auxiliar `visitados[len]` para marcar las posiciones ya visitadas.
- Recorre el arreglo desde la posición inicial, marcando cada índice como visitado.
- Desde cada posición, avanza a la siguiente indicada por el valor en `arr[i]` mientras no se repita una posición.
- Cuando se detecta que la siguiente posición ya fue visitada, se compara la longitud actual del recorrido con `path_len` y se actualiza si es mayor.

La subrutina principal usa el stack para guardar los registros necesarios durante la llamada y restaurarlos correctamente al terminar. Los registros `s0`, `s1` y `s2` mantienen, respectivamente, el largo del arreglo, su dirección base y la dirección base del arreglo de visitados.

```
.data
    arr:      .word 3,0,1,2
    len:      .word 4
    path_len: .word -1
.text
main:
    la t0, len
    lw s0, 0(t0)
    la s1, arr
    la t0, path_len
    addi t1, zero, -1
    sw t1, 0(t0)
    slli t0, s0, 2
    sub sp, sp, t0
    add s2, sp, zero
    addi t1, zero, 0
L_init:
    bge t1, s0, L_init_done
    slli t2, t1, 2
    add t3, s2, t2
    sw zero, 0(t3)
    addi t1, t1, 1
    jal zero, L_init
L_init_done:
    addi a0, zero, 0
    addi a1, zero, 0
    jal ra, L_call
L_return:
    slli t0, s0, 2
    add sp, sp, t0
    addi a7, zero, 10
    ecall
L_call:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw a0, 8(sp)
    sw a1, 4(sp)
    sw s3, 0(sp)
    slli t0, a0, 2
    add t1, s2, t0
    addi t2, zero, 1
    sw t2, 0(t1)
    add t1, s1, t0
    lw s3, 0(t1)
    slli t0, s3, 2
    add t1, s2, t0
    lw t2, 0(t1)
    addi t3, zero, 1
    beq t2, t3, L_cycle
    add a0, s3, zero
    addi a1, a1, 1
    jal ra, L_call
    jal zero, L_end
L_cycle:
    la t0, path_len
    lw t1, 0(t0)
    bge t1, a1, L_end
    sw a1, 0(t0)
L_end:
    lw a0, 8(sp)
    slli t0, a0, 2
    add t1, s2, t0
    sw zero, 0(t1)
    lw s3, 0(sp)
    lw a1, 4(sp)
```

```
    lw a0, 8(sp)
    lw ra, 12(sp)
    addi sp, sp, 16
    jalr zero, 0(ra)
```

La distribución sugerida del puntaje es la siguiente:

- **1.0 pto. — Estructura general y manejo de stack:** reserva, guardado y restauración correctos de los registros.
- **1.0 pto. — Lógica de recorrido y detección de repetición:** marcación y comparación adecuadas de posiciones visitadas.
- **1.0 pto. — Actualización del valor máximo:** comparación y almacenamiento correctos en `path_len`.

Si el código presenta la estructura correcta de recorrido con marcación de visitados pero tiene errores menores en saltos o restauración del stack, se podrá otorgar **hasta 0.75 ptos.**