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

IIC2343 - Arquitectura de Computadores (II/2025)

Guía de ejercicios: RISC-V

Ayudantes: Daniela Ríos (danielaarp@uc.cl), Alberto Maturana (alberto.maturana@uc.cl), Ignacio Gajardo (gajardo.ignacio@uc.cl)

Pregunta 1: DCCálculo - RISC-V al rescate

Te encuentras estudiando para un ramo de la Universidad, cuando en un ejercicio de ayudantía te aparece la temible función **factorial:** `x!`, la cual te toma mucho tiempo calcular y no alcanzarás a terminar tu estudio a tiempo.

Por suerte, también te encuentras cursando Arquitectura de Computadores y, como expert@ en RISC-V, decides elaborar un código capaz de calcular esta función al instante y así agilizar tu estudio. Para esto, puedes utilizar el siguiente código base:

```
.data
  x: .word 4

.text
  # Completa tu código acá,
```

IMPORTANTE: No es necesario que respetes la convención de llamadas en este ejercicio.

Solución: A continuación se presenta una alternativa de código que implementa la función f(x) = x!:

```
.data
        x: .word 8

.text
    lw t0, x              # Contador de cuántas multiplicaciones nos faltan
    addi s0, zero, 1      # Constante = 1
    addi sp, sp, -4       # Respaldar ra
    sw ra, 0(sp)
    jal ra, factorial_x   # Llamada a subrutina
    lw ra, 0(sp)

    # ecall para imprimir el resultado (código 1)
    addi a7, zero, 1      # a7 -> Código de la operación a realizar
    add a0, zero, t0      # a0 -> (Si es necesario) Argumentos de la operación a realizar
    ecall

    # ecall para terminar el programa (código 10)
    addi a7, zero, 10
    ecall

    factorial_x:
        beq t0, s0, end_factorial_x
        addi sp, sp, -8
        sw ra, 0(sp)          # Respaldo de ra
        sw t0, 4(sp)          # Guardar qué números tengo que multiplicar
        addi t0,t0, -1        # Decrezco el contador
        jal ra, factorial_x   # Vuelvo a llamar a la subrutina
        lw ra, 0(sp)          # Recuperamos ra

        # Multiplicación
        lw t1, 4(sp)           # Recuperamos el número que toca multiplicar ahora
        addi sp, sp, 8         # Reestablecemos el SP
        mul t0, t0, t1         # t0 = t0 * t1

        end_factorial_x:
            jalr zero, 0(ra)  # Equivalente a RET, sale de la función
```

Puede ver el desarrollo de este código en la siguiente cápsula.

Nota: Si bien el código funciona correctamente, este **NO** cumple con la convención de llamadas de RISC-V, dado que se debería haber hecho uso del registro `a0` para almacenar el argumento x y el valor de retorno de la subrutina `factorial_x`. Además, dentro de la subrutina se utilizó el registro `t0` (*caller-saved*), sin hacer el respaldo correspondiente.

Pregunta 2: DCCálculo - Resumen perdido

Llegó el día de dar la prueba para la que tanto has estudiado y, al mirar Canvas, ¡te enteras que podrás llevar una hoja de resumen! Sin embargo, entre todo el caos del estudio no logras encontrar la hoja en tu cajón de apuntes. Por suerte, tienes todas tus hojas marcadas con un identificador, por lo que decides utilizar tus habilidades de RISC-V y con ellas crear un programa que te ayude a encontrar el resumen.

Para esto, deberás elaborar un programa que realice la **búsqueda binaria** de un elemento `x` sobre un arreglo ordenado `arr` y guardar su índice en `element_index`. Si el elemento `x` no se encuentra en `arr`, entonces se debe mantener por defecto el valor `element_index` = -1.

La búsqueda binaria se realiza a través del siguiente procedimiento:

- Se revisa el elemento central del arreglo `arr[left_bound : right_bound]`. Llamaremos `c` al índice del elemento central y se computa así: `c = (left_bound + right_bound) // 2`.
- Si el valor del elemento central es igual al valor buscado, *i.e.* `arr[c] == x`, se retorna su posición como el restulado de la búsqueda.
- Si su valor no es igual al buscado:
  - Si el elemento central es menor al elemento buscado, se realiza la búsqueda nuevamente con `left_bound` igual a `c + 1`.
  - Si el elemento central es mayor al elemento buscado, se realiza la búsqueda nuevamente con `right_bound` igual a `c - 1`.
- La búsqueda termina cuando `left_bound` es mayor a `right_bound`.

Puedes utilizar el siguiente fragmento de código como base:

```
.data
    arr:            .word -1000, -255, -7, -1, 0, 10, 11, 27, 255, 1000, 10000  # Arreglo
    left_bound:     .word 0                                                     # Límite izquierdo
    right_bound:    .word 10                                                    # Límite derecho
    x:              .word -7                                                    # Elemento a buscar
    element_index:  .word -1                                                    # Índice del elemento

.text
    # Tu código aquí
```

IMPORTANTE: No es necesario que respetes la convención de llamadas en este ejercicio.

Solución: A continuación se presenta un código que implementa la búsqueda binaria:

```
.data
    arr:            .word -1000, -255, -7, -1, 0, 10, 11, 27, 255, 1000, 10000  # Arreglo
    left_bound:     .word 0                                                     # Límite izquierdo
    right_bound:    .word 10                                                    # Límite derecho
    x:              .word -7                                                    # Elemento a buscar
    element_index:  .word -1                                                    # Índice del elemento

.text
    main:
        # Almacenar elementos en registros
```

```
    # Datos en memoria (.data)
    la a0, arr  # Dirección de memoria del primer elemento del arreglo
    lw a1, left_bound
    lw a2, right_bound
    lw a3, x

    # Constantes útiles
    addi s0, zero, 2  # s0 = 2
    addi s1, zero, 4  # s1 = 4


    # Llamado inicial a la subrutina
    # respaldo de registros
    addi sp, sp, -20  # hacemos espacio en el stack
    sw ra, 0(sp)
    sw a0, 4(sp)
    sw a1, 8(sp)
    sw a2, 12(sp)
    sw a3, 16(sp)

    jal ra, bin_search
    la t0, element_index  # t0 = dirección de memoria de element_index
    sw a0, 0(t0)  # guardamos el índice encontrado (a0) en element_index

    # Terminar el programa -> ecalls (terminar el programa)
    # a7 -> codigo de la operación a realizar
    # a0 -> argumentos de la operación/retornos de la operación
    addi a7, zero, 10  # código 10 = terminar el programa
    ecall

bin_search:
    # 0) Condición de salida: Si left_bound > right_bound, no encontré el elemento
    blt a2, a1, element_not_found

    # 1) Revisar el elemento central
    # Calcular el índice c del elemento central
    add t0, a1, a2  # t0 = left_bound + right_bound
    div t0, t0, s0  # t0 = t0 // 2 = c
    mv t2, t0       # t2 = t0 = c

    # Revisar el elemento asociado al índice c
    # En este punto [t0 = c (índice del arreglo)] y [a0 = dirección de memoria de arr[0]]
    # Recordar que en RISC-V, cada bloque de memoria usa 4 bytes de memoria
    # En otras palabras, si el elemento 0 del array (arr[0]) está en la dirección 0=4*0, entonces
    # el elemento 1 (arr[1]) está en la dirección 4=4*1, el elemento 2 está en la dirección 8=4*2...

    mul t0, t0, s1  # t0= t0*4 (dir. de memoria que tengo que avanzar desde arr[0] hasta arr[c])
    add t0, t0, a0  # t0 = dirección de memoria del elemento arr[c]
    lw t0, 0(t0)    # t0 = arr[c] = 10

    # 2) Si arr[c] == x, encontre el elemento y retorno el índice
    beq t0, a3, element_found

    bgt t0, a3, left_array_search
    # 3) Si arr[c] < x, busco hacia la derecha
    right_array_search:
        add a1, zero, t2  # left_bound = c
        addi a1, a1, 1    # left_bound = c + 1
        beq zero, zero, next_search

    # 4) Si arr[c] > x, busco hacia la izquierda
    left_array_search:
        add a2, zero, t2  # right_bound = c
        addi a2, a2, -1   # right_bound = c - 1

    next_search:  # llamado recursivo a la subrutina bin_search
```

```
            # respaldar antes del siguiente llamado
            addi sp, sp, -28  # generar el espacio en el stack
            sw ra, 0(sp)
            sw a0, 4(sp)
            sw a1, 8(sp)
            sw a2, 12(sp)
            sw a3, 16(sp)
            sw t0, 20(sp)
            sw t2, 24(sp)

            jal ra, bin_search  # llamado recursivo

            # recuperar después del llamado recursivo
            lw ra, 0(sp)
            # a0 NO se recupera, porque el llamado recursivo encontró el resultado
            lw a1, 8(sp)
            lw a2, 12(sp)
            lw a3, 16(sp)
            lw t0, 20(sp)
            lw t2, 24(sp)
            addi sp, sp, 28  # liberamos el espacio en el stack

            beq zero, zero, bin_search_end

        element_found:
            add a0, zero, t2  # a0 = c (el indice del valor que buscabamos)
            beq zero, zero, bin_search_end

        element_not_found:
            addi a0, zero, -1  # a0 = -1 (no se encontró el elemento x)

        bin_search_end:
            jalr zero, 0(ra) # RET, termina la subrutina
```

Puede ver el desarrollo de este código en la siguiente cápsula.

Nota: Si bien el código funciona correctamente, este **NO** cumple con la convención de llamadas de RISC-V, dado que no se hacen los respaldos *caller-saved* y *callee-saved* que propone la convención.

1. *Feedback* cápsulas

Escanee el QR para entregar *feedback* sobre las cápsulas.