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

IIC2343 - Arquitectura de Computadores (I/2025)

Actividad de programación

Sección 2 - Pauta de evaluación

Pregunta 1: Explique el código (3 ptos.)

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

```
.data
  x: .word 4
.text
  lw t0, x
  addi s0, zero, 1
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, func_x
  lw ra, 0(sp)
  add a0, zero, t0
  addi a7, zero, 1
  ecall
  addi a7, zero, 10
  ecall
  func_x:
    beq t0, s0, end_func_x
    addi sp, sp, -8
    sw ra, 0(sp)
    sw t0, 4(sp)
    addi t0, t0, -1
    jal ra, func_x
    lw ra, 0(sp)
    lw t1, 4(sp)
    addi sp, sp, 8
    mul t0, t0, t1
    end_func_x:
      jalr zero, 0(ra)
```

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

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

   Solución: La función computa el factorial de x, *i.e.* $f(x) = x!$. Esto se evidencia en el hecho de que el valor de retorno, almacenado en el registro `t0`, es igual a $x \times f(x-1)$, con caso base $f(1) = 1$. De esta forma, se tiene que:

   $f(x) = x \times f(x-1) = x \times (x-1) \times f(x-2) = \cdots = x \times (x-1) \times (x-2) \times \cdots \times 1 = x!$

   No es necesario que se haga un detalle completo del código, pero sí que se demuestre que se entiende el cómputo recursivo del factorial de x.

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 ptos.** si su respuesta es incorrecta. Por otra parte, se otorgan **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. El motivo radica en el almacenamiento del argumento x y del valor de retorno de la subrutina `func_x`: si se siguiera la convención, se habría usado el registro `a0` tanto para almacenar x como para retornar $x!$. No obstante, se hace uso del registro `t0`, el que está pensando en el almacenamiento de valores temporales que pueden ser sobreescritos por llamados a subrutinas.

Pregunta 2: Elabore el código (3 ptos.)

Elabore, utilizando el Assembly RISC-V, un programa que **realice la búsqueda binaria de un elemento `x` sobre un arreglo ordenado `arr` y guarde su índice en `element_index`**. Si el elemento `x` no se encuentra en `arr`, entonces se debe mantener por defecto el valor `element_index` $= -1$. El largo del arreglo se deduce de la variable `right_bound`, correspondiente al índice del último elemento de este.

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 resultado de la búsqueda.
- Si su valor no es igual al buscado:
  - Si el elemento central es mayor al elemento buscado, se realiza la búsqueda nuevamente con `right_bound` igual a `c` - 1.
  - Si el elemento central es menor al elemento buscado, se realiza la búsqueda nuevamente con `left_bound` igual a `c` + 1.
- La búsqueda termina cuando `left_bound` es mayor a `right_bound`.

Puede utilizar el siguiente fragmento de código como base:

```
.data
    arr:           .word -1000, -255, -7, 0, 10, 11, 27, 255, 1000, 10000  # Arreglo
    left_bound:    .word 0                                                 # Límite izquierdo
    right_bound:   .word 9                                                 # Límite derecho
    x:             .word -7                                                # Elemento a buscar
    element_index: .word -1                                                # Índice del elemento
.text
  # Su código aquí
```

La asignación de puntaje se distribuirá de la siguiente forma:

- **1 pto.** por resolver correctamente el caso donde el elemento **no se encuentra** en el arreglo. Este punto se asigna si, y solo si se recorre el arreglo, independiente de si se hace con búsqueda binaria o no.
- **2 ptos.** por resolver correctamente el caso donde el elemento **se encuentra** en el arreglo. Se descuentan:
  - **0.5 ptos.** si en `element_index` se almacena la dirección de memoria del elemento y no su índice.
  - **1 pto.** si la búsqueda falla **en solo un caso**.
  - **2 ptos.** si la búsqueda falla en más de un caso; o bien **no se realiza con búsqueda binaria**.

IMPORTANTE: No es necesario que respete la convención en este ejercicio.

Solución: A continuación, un código que cumple lo pedido:

```
.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:
    la a0, arr
    lw a1, left_bound
    lw a2, right_bound
    lw a3, x
    addi s0, zero, 2
    addi s1, zero, 4
    addi sp, sp, -20
    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
    sw a0, 0(t0)
    lw ra, 0(sp)
    lw a0, 4(sp)
    lw a1, 8(sp)
    lw a2, 12(sp)
    lw a3, 16(sp)
    addi sp, sp, 20
    addi a7, zero, 10
    ecall
  bin_search:
    blt a2, a1, element_not_found
    add t0, a1, a2
    div t0, t0, s0
    mv t2, t0
    slli t0, t0, 2
    add t0, t0, a0
    lw t0, 0(t0)
    beq t0, a3, element_found
    addi sp, sp, -28
    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)
    bgt t0, a3, left_array_search
    right_array_search:
      add a1, zero, t2
      addi a1, a1, 1
      beq zero, zero, next_search
    left_array_search:
      add a2, zero, t2
      addi a2, a2, -1
    next_search:
      jal ra, bin_search
      lw ra, 0(sp)
      lw a1, 8(sp)
      lw a2, 12(sp)
      lw a3, 16(sp)
      lw t0, 20(sp)
      lw t2, 24(sp)
      addi sp, sp, 28
      beq zero, zero, bin_search_end
    element_found:
      add a0, zero, t2
      beq zero, zero, bin_search_end
    element_not_found:
      addi a0, zero, -1
    bin_search_end:
      jalr zero, 0(ra)
```

Para evaluar su correctitud, se modificará el valor de `x` para verificar el cómputo correcto del índice según sea el caso. Si se detecta *hardcodeo*, **se otorgarán 0 puntos.**