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

IIC2343 - Arquitectura de Computadores (I/2026)

Ayudantía 10: RISC-V

Ayudantes: Alberto Maturana (alberto.maturana@uc.cl), Mario Rojas (mario.denzel@estudiante.uc.cl), José Mendoza (jfmendoza@uc.cl)

Pregunta 1: Elabore el código I

Elabore, utilizando el Assembly RISC-V, un programa que a partir de un arreglo `arr` de largo `len`, determine si este posee un par de números **en posiciones distintas** tal que su suma sea igual a un valor `pair_sum`. Si se encuentra un par de números que cumple lo pedido, se deben guardar los índices del par en las variables `x_pair_sum` e `y_pair_sum`; en otro caso, sus valores deben ser iguales a -1. Si existe más de un par de valores que cumple lo pedido, puede escoger arbitrariamente los primeros que encuentre.

A continuación, tres ejemplos:

$$\texttt{arr} = [1, 3, 5, 7, -5] \quad \texttt{pair_sum} = 10 \rightarrow (\texttt{x_pair_sum}, \texttt{y_pair_sum}) = \quad (1, 3)$$
$$\texttt{arr} = [1, 3, 5, 7, -5] \quad \texttt{pair_sum} = 11 \rightarrow (\texttt{x_pair_sum}, \texttt{y_pair_sum}) = \quad (-1, -1)$$
$$\texttt{arr} = [1, 3, 5, 7, -5] \quad \texttt{pair_sum} = 14 \rightarrow (\texttt{x_pair_sum}, \texttt{y_pair_sum}) = \quad (-1, -1)$$

Puede utilizar el siguiente fragmento de código como base:

```
.data
  arr:          .word 1, 3, 5, 7, -5
  len:          .word 5
  pair_sum:     .word 10
  x_pair_sum:   .word -1
  y_pair_sum:   .word -1
.text
  # Su código aquí.
```

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

```
.data
  arr:         .word 1, 3, 5, 7, -5
  len:         .word 5
  pair_sum:    .word 10
  x_pair_sum: .word -1
  y_pair_sum: .word -1
.text
  la s0, arr                        # Dirección inicial del arreglo.
  addi s1, zero, 4                  # Constante para multiplicar índices por 4 para obtener direcciones.
  lw s2, len                        # Largo del arreglo.
  addi s3, s2, -1                   # len - 1 para terminar antes el loop de i.
  lw s4, pair_sum                   # Suma a buscar.
  la s5, x_pair_sum                 # Dirección para almacenar el índice del primer número del par.
  la s6, y_pair_sum                 # Dirección para almacenar el índice del segundo número del par.
  addi t0, zero, 0                  # Índice i
  find_pair_sum:
    i_loop:                         # Iteración respecto al índice i
      addi t1, t0, 1                # j = i + 1, no vemos los elementos hacia atrás, ya se revisaron.
      j_loop:
        mul t2, t0, s1              # t2 = 4 * i
        mul t3, t1, s1              # t3 = 4 * j
        add t2, s0, t2              # t2 = dir(arr) + 4 * i
        add t3, s0, t3              # t3 = dir(arr) + 4 * j
        lw t2, 0(t2)                # t2 = arr[i]
        lw t3, 0(t3)                # t3 = arr[j]
        add t4, t2, t3              # t4 = arr[i] + arr[j]
        beq t4, s4, pair_sum_found  # arr[i] + arr[j] == pair_sum
      j_loop_continue:
        addi t1, t1, 1
        blt t1, s2, j_loop          # if j >= n, end j_loop
    i_loop_continue:
      addi t0, t0, 1
      bge t0, s3, end               # if i >= n - 1, end i_loop
      jal zero, i_loop
    pair_sum_found:
      sw t0, 0(s5)                  # x_pair_sum = i
      sw t1, 0(s6)                  # y_pair_sum = j
  end:
    addi a7, zero, 10
    ecall
```

Pregunta 2: Explique el código

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

```
.data
  x:            .word 7
  n:            .word 3
  result:       .word 0
.text
    lw a0, x
    lw a1, n
    la s0, result
    addi s1, zero, 1
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, func_x_n
    lw ra, 0(sp)
    addi sp, sp, 4
    sw a0, 0(s0)
    li a7, 10
    ecall
  func_x_n:
    beq a1, s1, func_x_n_end
    addi sp, sp, -8
    sw ra, 0(sp)
    sw a0, 4(sp)
    addi a1, a1, -1
    jal ra, func_x_n
    lw ra, 0(sp)
    lw t0, 4(sp)
    addi sp, sp, 8
    mul a0, a0, t0
    func_x_n_end:
      jalr zero, 0(ra)
```

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

1. Indique, con argumentos y en términos de x y n, lo que retorna la función $f(x, n)$. Por ejemplo, $f(x, n) = x + n$.

   Solución: La función computa la potencia de x a la n, *i.e.* $f(x, n) = x^n$. Esto se evidencia en el hecho de que el valor de retorno, almacenado en el registro `a0`, es igual a $x \times f(x, n-1)$, con caso base $f(x, 1) = x$. De esta forma, se tiene que:

   $$f(x, n) = x \times f(x, n-1) = x \times x \times f(x, n-2) = \cdots = x \times x \times x \times \cdots \times x = x^n$$

2. Indique, con argumentos, si el fragmento anterior respeta o no la convención de llamadas de RISC-V.

   Solución: El fragmento anterior **sí respeta** la convención de llamadas de RISC-V. Los motivos son los siguientes:

   - Se respalda `ra` antes de cada llamado.

- Se usan los registros `a0` y `a1` para los argumentos de `func_x_n`, y `a0` para su retorno.
- Se respalda el registro `a0` antes de cada llamado recursivo, considerando su sobre-escritura; su uso posterior al llamado; y el hecho de ser *caller-saved*.

Pregunta 3: Elabore el código II

Estamos en el año 2084. Eres el ingeniero principal de la Resistencia y has recibido un cargamento de suministros médicos vitales. El sistema de transporte automático ha dejado las cajas en el muelle de carga, pero están completamente desordenadas según su número de prioridad.

Para que los drones de distribución puedan recoger las cajas de forma eficiente, el protocolo exige que los suministros estén ordenados de **menor a mayor prioridad** en la memoria central del sistema. Debido a que el procesador del almacén es un modelo antiguo basado en la arquitectura **RISC-V** y que por la antigüedad se ha estropeado una parte del procesador, dando error al ejecutar loops, deberas programar una rutina totalmente recursiva.

El Algoritmo: Bubble Sort con Doble Recursión

En esta versión avanzada para el procesador del almacén, utilizaremos una **doble recursión** para garantizar que el sistema opere de manera puramente funcional. El problema se divide en dos procesos recursivos anidados:

1. **Recursión Externa (Control de Pasadas):** Esta función se encarga de determinar cuántas veces debemos revisar el arreglo.
   - **Caso Base:** Si el tamaño del fragmento a ordenar es $n \leq 1$, el proceso termina.
   - **Llamada Recursiva:** En cada paso, invoca a la función de la pasada interna y luego se llama a sí misma con un tamaño de $n - 1$.
2. **Recursión Interna (Pasada de Comparación):** Sustituye al bucle de recorrido tradicional. Se encarga de comparar el elemento en el índice i con el de la posición $i + 1$.
   - **Caso Base:** Si el índice i alcanza el límite superior $(n - 1)$, la pasada termina y el control regresa a la función externa.
   - **Acción:** Si $arr[i] > arr[i + 1]$, se realiza el intercambio en memoria.
   - **Llamada Recursiva:** La función se llama a sí misma incrementando el índice $(i + 1)$ para comparar la siguiente pareja.

Esta arquitectura permite que el procesador gestione el ordenamiento exclusivamente mediante la manipulación del **Stack Pointer** (`sp`) y el registro de retorno (`ra`).

Código Base

Utiliza la siguiente estructura de memoria para comenzar tu implementación:

```
.data
arr:    .word 64, 34, 25, 12, 22, 11, 90  # Prioridades de las cajas
size:   .word 7                           # Cantidad de cajas
.text
main:
  # Su código aquí
```

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

```
.data
arr:    .word 64, 34, 25, 12, 22, 11, 90  # Arreglo desordenado
size:   .word 7                           # Tamano del arreglo
.text
main:
    la   a0, arr               # a0 = Direccion base del arreglo
    lw   a1, size              # a1 = n (elementos totales)

    jal  ra, bubble_sort       # Llamada a la recursion externa

    addi a7, zero, 10
    ecall

bubble_sort:
    addi s0, zero, 1           # s0 = 1 (para comparacion)
    ble  a1, s0, sort_end      # Si n <= 1, terminar

    addi sp, sp, -4            # Reservar espacio en Stack
    sw   ra, 0(sp)             # Guardar ra

    addi a2, zero, 0           # a2 = indice i = 0
    addi a1, a1, -1            # a3 = limite = n - 1
    jal  ra, bubble_pass       # Llamada a la recursion interna

    lw   ra, 0(sp)             # Restaurar ra
    addi sp, sp, 4             # Liberar Stack

    addi sp, sp, -4            # Reservar espacio en Stack
    sw   ra, 0(sp)             # Guardar ra
    jal  ra, bubble_sort       # Siguiente pasada recursiva
    lw   ra, 0(sp)             # Restaurar ra
    addi sp, sp, 4             # Liberar Stack

sort_end:
    jalr zero, 0(ra)

bubble_pass:
    bge  a2, a1, pass_end      # Si i >= limite, terminar pasada

    slli t0, a2, 2             # t0 = i * 4 (offset)
    add  t0, t0, a0            # t0 = &arr[i]
    lw   t1, 0(t0)             # t1 = arr[i]
    lw   t2, 4(t0)             # t2 = arr[i+1]

    ble  t1, t2, no_swap       # Si arr[i] <= arr[i+1], no swap
    sw   t2, 0(t0)             # Intercambiar: arr[i] = t2
    sw   t1, 4(t0)             # Intercambiar: arr[i+1] = t1
no_swap:
    addi sp, sp, -4            # Guardar ra para bajar nivel
    sw   ra, 0(sp)             # de recursion interna

    addi a2, a2, 1             # i = i + 1
    jal  ra, bubble_pass       # Llamada recursiva interna

    lw   ra, 0(sp)             # Restaurar ra
    addi sp, sp, 4             # Liberar Stack
pass_end:
    jalr zero, 0(ra)
```

Feedback ayudantía

Escanee el QR para entregar feedback sobre la ayudantía.

