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

IIC2343 - Arquitectura de Computadores (II/2026)

Ayudantía 8

Ayudantes: Ignacio Gajardo (gajardo.ignacio@uc.cl), Mario Rojas (mario.denzel@estudiante.uc.cl), Nicolás Romo Aubele (nroma@uc.cl), Carlos Canales (carlos.canales@estudiante.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:** $x!$. Tu compañero te entrega la siguiente propuesta en RISC-V para calcularla:

```
.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
```

A partir del código presentado y considerando la **convención de llamadas de RISC-V** vista en clases:

a) ¿Cumple este código con la convención de llamadas de RISC-V?

b) Justifique detalladamente su respuesta indicando el uso de registros y el manejo de contexto.

Solución:

a) **No cumple** con la convención de llamadas estándar de RISC-V.

b) **Justificación:**

- **Pase de parámetros y valor de retorno:** La subrutina `factorial_x` no recibe el argumento a través del registro `a0` ni retorna el resultado en él. En su lugar, opera directamente sobre el registro `t0`.
- **Uso de registros temporales (*caller-saved*):** Se utiliza el registro `t0` dentro de la subrutina sin ser respaldado debidamente. Al ser un registro *caller-saved*, las llamadas recursivas sobreescriben su valor si no se guarda explícitamente en el *stack* previo a la llamada.



Pregunta 2: Análisis de Código: La Función Misteriosa

El siguiente codigo recursivo fue escrito en **Assembly RISC-V**, pero los nombres de las funciones y variables que participan en la subrutina han sido ofuscados.

```
.data
    a: .word 2
    b:  .word 5
.text
    main:
        la t0, a
        lw a0, 0(t0)
        la t0, b
        lw a1, 0(t0)
        call fun_mistery
        mv s0, a0
        j fin
    fun_mistery:
        addi sp, sp, -8
        sw ra, 4(sp)
        sw a0, 0(sp)
        blez a1, caso_a
        addi a1, a1, -1
        call fun_mistery
        lw t1, 0(sp)
        mul a0, a0, t1
        j retorno
    caso_a:
        li   a0, 1
    retorno:
        lw   ra, 4(sp)
        addi sp, sp, 8
        ret
    fin:
        li a7, 1
        ecall
        li a7, 10
        ecall
```

Responda a las siguientes preguntas basándose exclusivamente en el análisis del código:

a) ¿Qué hace la subrutina `fun_mistery` en función de las variables `a` y `b`?

b) Al ejecutar el bloque `fin`, ¿Cuál será el valor numérico que se imprimirá en la consola?

c) En el prólogo de `fun_mistery`, se observa que el registro `a0` es respaldado en el *stack*, pero `a1` no lo es. Justifique esta decisión basándose en la **Convención de llamadas** y el proposito general del algoritmo.

Solución: A continuación, se muestra una solución a las preguntas y un analisis del código presentado:

a) **Operación matemática:** La subrutina calcula una **potencia**. Toma el valor de `a` como base y `b` como exponente.

b) **Valor impreso:** Las variables iniciales son $a = 2$ y $b = 5$. La función calcula 2^5. Por lo tanto, el sistema imprimirá el valor **32**.

c) **Justificación del Stack:** Ambos son registros temporales (*Caller-saved*), pero su necesidad de persistencia a la recursión (`call fun_mistery`) es distinta.

- El registro `a0` (la base) **debe** respaldarse porque la instrucción `mul a0, a0, t1` lo necesita para operar **después** de que la llamada recursiva retorna. Como la subrutina hija utiliza `a0` para traer el valor de retorno, la Base original sería destruida si no se guarda en el **stack**.
- El registro `a1` (el exponente) **no** se respalda porque solo se utiliza para evaluar el caso base y decrementar (`addi a1, a1, -1`) **antes** del `call`. Una vez que la subrutina retorna de la recursión, el valor pasado del exponente ya no tiene utilidad.

```
.data
    base: .word 2                       # Base = 2
    exp:  .word 5                       # Exponente = 5

.text
    main:
        la t0, base                     # t0 = base
        lw a0, 0(t0)                    # a0 = t0 = base
        la t0, exp                      # t0 = exp
        lw a1, 0(t0)                    # a1 = t0 = exp
        call potencia                   # El resultado quedara en a0
        mv s0, a0                       # Guardar el resultado final de la potencia en s0
        j fin

    potencia:
        addi sp, sp, -8                 # Reservamos espacio en el stack pointer
        sw ra, 4(sp)                    # Respaldamos la return address
        sw a0, 0(sp)                    # Respaldamos a0
        blez a1, caso_base              # Si exponente <= 0
        addi a1, a1, -1                 # exponente = exponente - 1
        call potencia                   # a0 = potencia(Base, exponente-1)
        lw t1, 0(sp)                    # Recuperamos la Base original hacia t1
        mul a0, a0, t1                  # a0 = Resultado parcial * Base
        j retorno

    caso_base:
        li a0, 1                        # Cualquier numero elevado a 0 es 1

    retorno:
        lw ra, 4(sp)                    # Restaura la direccion de retorno original (ra)
        addi sp, sp, 8                  # Libera los 8 bytes del stack (mueve el SP hacia arriba)
        ret                             # Salta a la direccion almacenada en ra

    fin:
        li a7, 1                        # Cargar el codigo 1 de ecall (Imprimir)
        ecall                           # Llamada al sistema imprimir
        li a7, 10                       # Cargar el codigo 10 de ecall (Terminar programa)
        ecall                           # Llamada al sistema salir
```

Pregunta 3: Corrección de código en RISC-V

El siguiente programa, desarrollado en el Assembly del computador básico visto en clases, **debiese verificar si el número `n` es múltiplo de `m`**:

```
.data
    x_value: .word 7
    n:       .word 3
    x_pow_n: .word 0
.text

main:
    lw a0, x_value
    lw a1, n
    la s0, x_pow_n
    addi s1, zero, 1
    jal ra, pow
    sw a0, 0(s0)
    addi a7, zero, 10
    ecall

pow:
    beq a1, s1, end
    addi sp, sp, -4
    sw a0, 0(sp)
    addi a1, a1, -1
    jal ra, pow
    lw t0, 0(sp)
    addi sp, sp, 4
    mul a0, a0, t0

end:
    jr ra
```

Sin embargo, no lo hace correctamente. Si compila y ejecuta el código, se dará cuenta que no entrega el resultado esperado. **Busque** el error y, una vez encontrado, **arréglelo** para que el fragmento anterior entregue el resultado esperado.

Solución: El problema radica en que **no se respalda la dirección de retorno en cada llamado recursivo de `pow`**. Basta con asegurar el respaldo del registro `ra` antes del llamado recursivo y su recuperación antes del retorno, como se muestra a continuación:

```
.data
    x_value: .word 7
    n:       .word 3
    x_pow_n: .word 0
.text

main:
    lw a0, x_value
    lw a1, n
    la s0, x_pow_n
    addi s1, zero, 1
    addi sp, sp, -4   # MODIFICACIÓN: Restamos 4 al Stack Pointer para hacer espacio para ra (dirección
     de retorno)
    sw ra, 0(sp)      # MODIFICACIÓN: Respaldamos ra (caller-saved)
    jal ra, pow
    lw ra, 0(sp)      # MODIFICACIÓN: Recuperamos ra desde el Stack Pointer
    addi sp, sp, 4    # MODIFICACIÓN: Reestablecemos el valor del Stack Pointer
    sw a0, 0(s0)
    addi a7, zero, 10
    ecall

pow:
    beq a1, s1, end
    addi sp, sp, -8   # MODIFICACIÓN: Restamos 8 al Stack Pointer para hacer espacio para ra y para a0
    sw ra, 0(sp)      # MODIFICACIÓN: Respaldamos ra (caller-saved), ya que en la recursión se vuelve a
     llamar a pow
    sw a0, 4(sp)
    addi a1, a1, -1
    jal ra, pow
    lw ra, 0(sp)      # MODIFICACIÓN: Recuperamos ra desde el Stack Pointer
    lw t0, 4(sp)
    addi sp, sp, 8    # MODIFICACIÓN: Reestablecemos el valor del Stack Pointer, considerando que también
      respaldamos ra
    mul a0, a0, t0

end:
    jr ra
```



Feedback ayudantía

Escanee el QR para entregar feedback sobre la ayudantía.

