



IIC2343 - Arquitectura de Computadores (II/2025)

## Ayudantía 8

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

### Pregunta 1: De Assembly a RISC-V

Convierta el siguiente código en Assembly básico, que calcula el término  $n$  de la [sucesión de Fibonacci](#), a instrucciones de la arquitectura RISC-V. Maneje todas las variables del problema mediante *inputs* y registros, es decir, no debe utilizar el tag `.data`.

**Hint:** Para recibir la variable `num` como input, utilice la instrucción `ecall` con código 5.

```
DATA:
    num      ?    // Debe recibirse como input (? representa esto, no se puede utilizar en código real)
    counter  1    // Contador (i = 1)
    fn       1    // Fibonacci de n, número a calcular
    fn_1     0    // Fibonacci de n-1 (inicia con el valor de f(0))
    fn_2     1    // Fibonacci de n-2 (inicia con el valor de f(1))

CODE:
    check_base_cases: // Revisamos si estamos en un caso base
        MOV A, (num)    // A <- Número
        CMP A, 0
        JEQ end_0       // Si Número = 0, salto al caso base del 0 (end_0)
        CMP A, 1
        JEQ end_1       // Si Número = 1, salto al caso base del 1 (end_1)
        JMP while       // En caso contrario, entro en bucle para calcular Fibonacci (while)

    while:
        MOV A, (counter) // A <- Contador
        MOV B, (num)     // B <- Número
        CMP A, B
        JEQ end          // Si i = Número, salgo del bucle y salto al final del programa (end).

        INC (counter)    // Aumento en 1 el contador
        MOV A, (fn_1)    // A <- f(n-1)
        MOV B, (fn_2)    // B <- f(n-2)
        ADD A, B         // A <- f(n-1) + f(n-2)
        MOV (fn), A      // f(n) <- f(n-1) + f(n-2)

        // Actualizo los valores de f(n-1) y f(n-2) para la siguiente iteración
        MOV B, (fn_1)    // B <- f(n-1)
        MOV (fn_2), B    // f(n-2) <- f(n-1)
        MOV (fn_1), A    // f(n-1) <- f(n)

        JMP while

    end_0: // Caso base: f(0) = 0
        MOV A, 0
        MOV (fn), A
        JMP end

    end_1: // Caso base: f(1) = 1
        MOV A, 1
        MOV (fn), A
        JMP end
```

```
end:
    JMP end
```

### Solución:

```
.text
.globl check_base_cases

check_base_cases:
    li a7, 5          # a7 <- 5 (Código de ecall para recibir un input enter)
    ecall             # a0 <- input
    li t1, 1          # t1 <- 1
    beq a0, zero, end_0 # Si a0 = 0, salto al caso base del 0 (end_0)
    beq a1, t1, end_1  # Si a0 = 1, salto al caso base del 1 (end_1)

    # Guardamos variables en registros
    mv s0, a0 # s0 <- Input
    li s1, 1  # s1 <- Contador (i=1)
    li t0, 0  # t0 <- Fibonacci de n-2
    li t1, 1  # t1 <- Fibonacci de n-1

while:
    beq s0, s1, end # Si Número = i, salgo del bucle y salto al final del programa (end)

    addi s1, s1, 1 # Aumento en 1 el contador
    add t2, t0, t1 # f(n) <- f(n-1) + f(n-2)

    mv a0, t2 # a0 <- f(n)
    mv t0, t1 # f(n-2) <- f(n-1)
    mv t1, t2 # f(n-1) <- f(n)
    J while

end_0: # Caso base: f(0) = 0
    li a0, 0
    J end

end_1: # Caso base: f(1) = 1
    li a0, 1
    J end

end:
    li a7, 1 # a7 <- 1 (Código de input para imprimir a0)
    ecall    # Imprimir a0
    li a7, 10 # a7 <- 10 (Código de input para terminar programa)
    ecall    # Fin del programa
```

## Pregunta 2: Convención de llamadas (AP2 2023-2)

A continuación se presentarán varios fragmentos de código donde se realizan llamados a subrutinas. Deberá indicar, con argumentos, si los fragmentos respetan, o no, la convención de llamadas de RISC-V.

### (a) Primer fragmento:

```
.data
```

```

N:          .word 17
N_is_prime: .word -1
.text
main:
    lw a0, N
    li s0, 1
    la s1, N_is_prime
    addi sp, sp, -4
    sw ra, 0(sp)
    call check_is_prime
    lw ra, 0(sp)
    addi sp, sp, 4
    sw a0, 0(s1)
    li a7, 10
    ecall

check_is_prime:
    ble a0, s0, is_not_prime
    addi t1, a0, -1
    li t0, 2

division_loop:
    rem t2, a0, t0
    beqz t2, is_not_prime
    addi t0, t0, 1
    ble t0, t1, division_loop

is_prime:
    li a0, 1
    j end

is_not_prime:
    li a0, 0

end:
    ret

```

**Solución:** El fragmento anterior **sí respeta** la convención de llamada: Se respalda el registro **ra** antes de cada llamada; se utilizan los argumentos en los registros **a\***; se entrega el valor de retorno en **a0**; y siempre se restaura el valor de **sp**. No es necesario respaldar los registros **t\*** dado que no se utilizan fuera de la subrutina ni guardan un valor previo que se desee recuperar.

(b) Segundo fragmento:

```
.data
N:          .word 17
N_is_prime: .word -1
.text

main:
    lw a0, N
    li s0, 1
    la s1, N_is_prime
    call check_is_prime
    sw a0, 0(s1)
    li a7, 10
    ecall

check_is_prime:
    ble a0, s0, is_not_prime
    addi t1, a0, -1
    li t0, 2

division_loop:
    rem t2, a0, t0
    beqz t2, is_not_prime
    addi t0, t0, 1
    ble t0, t1, division_loop

is_prime:
    li a0, 1
    j end

is_not_prime:
    li a0, 0

end:
    ret
```

**Solución:** El fragmento anterior **no respeta la convención de llamadas de RISC-V** dado que no se respalda el registro **ra** antes de llevar a cabo el llamado de una subrutina.

(c) Tercer fragmento:

```
.data
N:          .word 17
N_is_prime: .word -1
.text

main:
    lw a0, N
    li s0, 1
    la s1, N_is_prime
    addi sp, sp, -4
    sw ra, 0(sp)
    call check_is_prime
    lw ra, 0(sp)
    addi sp, sp, 4
    sw t3, 0(s1)
    li a7, 10
    ecall

check_is_prime:
    ble a0, s0, is_not_prime
    addi t1, a0, -1
    li t0, 2

division_loop:
    rem t2, a0, t0
    beqz t2, is_not_prime
    addi t0, t0, 1
    ble t0, t1, division_loop

is_prime:
    li t3, 1
    j end

is_not_prime:
    li t3, 0

end:
    ret
```

**Solución:** El fragmento anterior **no respeta la convención de llamadas de RISC-V** dado que el valor de retorno no se encuentra en ninguno de los registros a0, a1.

(d) Cuarto fragmento:

```
.data
N:          .word 17
N_is_prime: .word -1
.text

main:
    lw a0, N           # Carga el valor de N en a0
    addi t0, zero, 1   # Carga el literal 1 en t0
    la t1, N_is_prime  # Carga la dirección de N_is_prime en t1
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, check_is_prime # Salto a check_is_prime y guarda retorno en ra
    lw ra, 0(sp)
    addi sp, sp, 4
    la t1, N_is_prime  # Carga la dirección de N_is_prime en t1
    sw a0, 0(t1)
    addi a7, zero, 10   # Carga el literal 10 en a7
    ecall               # Exit. Solo termina el programa

check_is_prime:
    ble a0, t0, is_not_prime # Salto condicional a is_not_prime si a0 <= t0
    addi t1, a0, -1
    addi t0, zero, 2         # Carga el literal 2 en t0

division_loop:
    rem t2, a0, t0
    beq t2, zero, is_not_prime # Salto condicional a is_not_prime si t2 == 0
    addi t0, t0, 1
    ble t0, t1, division_loop # Salto condicional a division_loop si t0 <= t1

is_prime:
    addi a0, zero, 1   # Carga el literal 1 en a0
    beq zero, zero, end # Salto incondicional a end

is_not_prime:
    addi a0, zero, 0   # Carga el literal 0 en a0

end:
    jalr zero, 0(ra)   # Retorno usando dirección almacenada en ra
```

**Solución:** El fragmento anterior **no respeta la convención de llamadas de RISC-V** dado que el valor del registro **t1** es sobrescrito por **check\_is\_prime** y no se respalda a pesar de ser *caller saved*.

### Pregunta 3: Corrección de código en RISC-V (AP2 2023-2)

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, arréglo 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.

