Laboratorio 02: Manipulación de Bytes, Bits y Convención de Llamadas (ABI)

Table of Contents

Fecha límite de entrega: lunes 23 de febrero.

En este laboratorio trabajaremos con la manipulación de datos a nivel de byte y bit, y aprenderás a gestionar la pila (**stack**) para llamadas a funciones anidadas. Aunque quizás es redudante decirlo, recuerda que las entradas de estos programas son solo ilustrativas: tú programa será validado empleando diversas entradas.

1. Ejercicio 1: Longitud de Cadena

Implementa la función `my_strlen`. Debe contar el número de caracteres en una cadena hasta encontrar el terminador nulo (`0`).

Archivo: `ex1_strlen.s`

```
.data
test_string: .string "Arquitectura 2025"

.text
.globl main

main:
    # Preparar argumento
```

```
    la a0, test_string

    # Llamar a la función
    jal ra, my_strlen

    # Imprimir resultado (Entero)
    mv a1, a0       # El resultado estaba en a0, lo movemos a a1 para ecall
    li a0, 1        # syscall 1: print_int
    ecall

    # Salir
    li a0, 10       # syscall 10: exit
    ecall

# ------------------------------------------------
# FUNCTION: my_strlen
# Entrada: a0 = dirección de inicio del string
# Salida:  a0 = longitud del string
# ------------------------------------------------
my_strlen:
    # TU CÓDIGO AQUÍ

    # --- INSTRUCCIÓN: BORRAR LAS SIGUIENTES DOS LÍNEAS ---
    li a0, 0    # Código temporal solo para evitar errores al compilar este código
incompleto
    ret         # Reemplaza esto con tu lógica de retorno real
    # ----------------------------------------------------
```

2. Ejercicio 2: Manipulación de Bits

Recibes un registro de 32 bits que contiene información de color en formato RGBA. Bits [23:16] corresponden al canal Verde (Green). Queremos aislar ese valor.

Archivo: `ex2_bitwise.s`

```
.text
.globl main

main:
    li a0, 0x12345678   # Color de prueba (R=12, G=34, B=56, A=78 hex)

    jal ra, get_green

    # Imprimir resultado (debería ser 0x34 o 52 decimal)
    mv a1, a0       # El resultado estaba en a0, lo movemos a a1 para ecall
    li a0, 1        # syscall 1: print_int
    ecall

    li a0, 10       # syscall 10: exit
```

```
    ecall

# ------------------------------------------------
# FUNCTION: get_green
# Entrada: a0 = color RGBA (32 bits)
# Salida:  a0 = valor del canal verde (8 bits)
# ------------------------------------------------
get_green:
    # TU CÓDIGO AQUÍ

    ret
```

3. Ejercicio 3: La ABI y Funciones Anidadas

Implementa `capitalize_string`. Esta función recorre un string y, para cada carácter, llama a una función auxiliar `to_upper`.

Reglas:

- `capitalize_string` **DEBE** llamar a `to_upper`. No puedes hacer la conversión ahí mismo.
- Debes gestionar el puntero de pila (`sp`) para guardar `ra` y cualquier registro `s` que uses.

Archivo: `ex3_abi.s`

```
.data
lower_str: .string "hola mundo riscv"

.text
.globl main

main:
    la a0, lower_str
    jal ra, capitalize_string

    # Imprimir string modificado
    la a1, lower_str # Cargar dirección para print_string
    li a0, 4         # syscall 4: print_string
    ecall

    li a0, 10        # syscall 10: exit
    ecall

# ------------------------------------------------
# FUNCTION: to_upper
# Entrada: a0 = carácter (byte)
# Salida:  a0 = carácter en mayúscula (si era minúscula)
# NOTA: Esta es una función hoja. No necesita stack.
# ------------------------------------------------
to_upper:
```

```
    li t0, 'a'
    li t1, 'z'
    blt a0, t0, is_not_lower   # Si es menor que 'a', no hacer nada
    bgt a0, t1, is_not_lower   # Si es mayor que 'z', no hacer nada
    addi a0, a0, -32           # Convertir a mayúscula (ASCII offset)
is_not_lower:
    ret

# ------------------------------------------------
# FUNCTION: capitalize_string
# Entrada: a0 = dirección del string
# Salida:  void (modifica el string en memoria)
# ------------------------------------------------
capitalize_string:
    # PRÓLOGO: Guardar registros necesarios en el Stack
    # ¿Qué registros necesitas mantener a través de la llamada a to_upper?

    # TU CÓDIGO AQUÍ (Prólogo)

loop_cap:
    # 1. Cargar byte actual
    # 2. Si es 0, ir a fin
    # 3. Mover byte a a0 (argumento)
    # 4. Llamar a to_upper
    # 5. Guardar el resultado (a0) de vuelta en memoria
    # 6. Avanzar puntero
    # 7. Repetir

    # TU CÓDIGO AQUÍ (Cuerpo)

end_cap:
    # EPÍLOGO: Restaurar registros y stack

    # TU CÓDIGO AQUÍ (Epílogo)

    ret
```

4. Ejercicio 4: El Código Roto

En la industria, rara vez escribes código desde cero; casi siempre arreglas el código de alguien más.

El siguiente programa, `broken_sum.s`, intenta sumar los elementos de un arreglo. Para "optimizar" la velocidad, el programador anterior intentó usar una técnica llamada **Loop Unrolling**: sumar 2 números por iteración en lugar de 1.

Sin embargo, el código tiene un **bug crítico**.

1. Funciona si el arreglo tiene longitud par (ej: 4).
2. Falla catastróficamente (bucle infinito o acceso ilegal de memoria) si el arreglo tiene longitud

impar (ej: 5).

Tu Tarea:

1. Copia el código en Venus.
2. Ejecútalo tal cual. Verás que falla (el arreglo de prueba tiene 5 elementos).
3. Usa el depurador paso a paso ("Step") para encontrar por qué el bucle no termina.
4. Corrige el código para que funcione con longitudes pares e impares.

Archivo: `ex4_broken.s`

```
.data
array: .word 1, 2, 3, 4, 5
len:   .word 5

.text
.globl main

main:
    la a0, array
    lw a1, len
    jal ra, sum_array_broken

    # Imprimir resultado
    # Debería ser 1+2+3+4+5 = 15
    mv a1, a0
    li a0, 1
    ecall

    li a0, 10
    ecall

# ------------------------------------------------
# FUNCTION: sum_array_broken
# Entrada: a0 = dirección array
# Entrada: a1 = longitud (número de palabras)
# Salida:  a0 = suma total
# ------------------------------------------------
sum_array_broken:
    li t0, 0          # Acumulador

loop:
    beq a1, zero, end # Si len == 0, terminar

    # Procesa elemento i
    lw t1, 0(a0)
    add t0, t0, t1

    # Procesa elemento i+1 (Optimización)
    lw t2, 4(a0)
```

```
    add t0, t0, t2

    # Avanzar punteros
    addi a0, a0, 8    # Avanzamos 2 palabras (8 bytes)
    addi a1, a1, -2   # Restamos 2 a la longitud

    j loop

end:
    mv a0, t0
    ret
```

5. Ejercicio 5: Decodificador de Mensajes Empaquetados

Has interceptado un flujo de datos binario "comprimido". El mensaje está oculto dentro de un arreglo de palabras de 32 bits (`.word`), pero no se lee secuencialmente. Debes escribir un decodificador que reconstruya el mensaje original.

5.1. Especificación del Formato de Instrucción

Cada palabra de 32 bits contiene una "Instrucción" empaquetada con la siguiente estructura:

- **Bits [7:0] → PAYLOAD:** Un carácter ASCII o un valor numérico.
- **Bits [10:8] → OPCODE:** El código de operación (0-7).
- **Bits [31:11] → BASURA:** Datos irrelevantes que deben ser ignorados.

5.2. Tabla de Opcodes

Tu programa debe leer una palabra, extraer el OPCODE y ejecutar la acción:

Opcode	Acción
`0` (FIN)	Detener la decodificación.
`1` (COPY)	Guardar el carácter `PAYLOAD` en el buffer de salida. Avanzar a la **siguiente** palabra.
`2` (SKIP)	Saltar hacia adelante `N` palabras, donde `N = PAYLOAD`. (Ej: si Payload es 2, saltas 2 palabras, ignorando las intermedias).
`3` (XOR)	Guardar en la salida el resultado de: `PAYLOAD XOR 0x2A`. Avanzar a la siguiente palabra.

Cualquier otro Opcode debe ser tratado como un salto simple a la siguiente palabra (sin guardar nada).

Archivo: `ex5_decoder.s`

```
.data
# Mensaje codificado (Entrada)
# Analiza manualmente el primero: 0x00000148
# -> Payload (7:0) = 0x48 ('H')
# -> Opcode (10:8) = 1 (COPY)
# -> Acción: Guardar 'H' y avanzar 1 palabra.
packed_data:
    .word 0x00000148 # Op:1, Val:'H' -> Save 'H'
    .word 0x0000016F # Op:1, Val:'o' -> Save 'o'
    .word 0x00000202 # Op:2, Val:2   -> Skip 2 words
    .word 0xFFFFFFFF # GARBAGE (Skipped)
    .word 0x12345678 # GARBAGE (Skipped)
    .word 0x00000346 # Op:3, Val:0x46 -> 0x46 XOR 0x2A = 'l' -> Save 'l'
    .word 0x00000161 # Op:1, Val:'a' -> Save 'a'
    .word 0x00000000 # Op:0 -> FIN

# Buffer para guardar el resultado (Salida)
output_buffer: .space 256

.text
.globl main

main:
    la a0, packed_data      # Input pointer
    la a1, output_buffer    # Output pointer

    jal ra, decode_message

    # Imprimir resultado descifrado
    la a1, output_buffer
    li a0, 4                # print_string
    ecall

    li a0, 10
    ecall

# ------------------------------------------------
# FUNCTION: decode_message
# Entrada: a0 = dirección input (packed_data)
# Entrada: a1 = dirección output (output_buffer)
# ------------------------------------------------
decode_message:
    # TU CÓDIGO AQUÍ

    ret
```

Salida esperada: Si lo haces bien, debería imprimir "Hola".

6. Entrega

Enviar el laboratorio resuelto a la dirección de correo electrónico jaime.parada.courses@gmail.com. Entrega un archivo `.zip` con:

1. Los 5 archivos `.s` corregidos.
2. Capturas de pantalla de Venus mostrando los detalles de cada ejercicio.