Challenge: ISA Functions

CS3051 - Computer Architecture

cwilliams, jgonzalez@utec.edu.pe
TAs: lcarranza, mchincha@utec.edu.pe

2025-1

Convertir String a Minúsculas

Problema

Descripción

Cree una función en ARM que cambie las letras mayúsculas de una cadena a minúsculas. La cadena termina con un carácter nulo (ASCII 0). Modifique la cadena original directamente en memoria (in-place). Los caracteres que no sean letras mayúsculas (A-Z) no deben cambiar.

Entrada y Salida

Prototipo en C

```c
void string_to_lowercase(char *str);
```

- `str` (en R0): Puntero al inicio de la cadena. La cadena termina con un byte de valor 0x00.

La función no retorna un valor; modifica la cadena. Si usa registros R4-R11, debe restaurarlos.

Detalles

- Recorra la cadena byte por byte. Use LDRB para leer y STRB para escribir.
- Una letra es mayúscula si su ASCII está entre 'A' (0x41) y 'Z' (0x5A).
- Para convertir una mayúscula a minúscula, sume 0x20 (la diferencia entre 'a' y 'A').
- Deténgase al encontrar el carácter nulo (0x00).
- Ejemplo: `"Hola MUNDO 123!"` se convierte en `"hola mundo 123!"`.
- ASCII: 'A'=0x41, 'Z'=0x5A, 'a'=0x61, '0'=0x00.

Entregables

1. **Código C:** Implementación de `string_to_lowercase`.
2. **Código ARM:** Traducción a ensamblador ARM.
3. **Memoria/Stack:** Diagrama de la memoria (y stack si lo usa) antes y después, para el caso de CPUlator.
4. **Prueba en CPUlator:**
   - **Entrada:** Cadena "CS3051z\0". Bytes: `0x43, 0x53, 0x33, 0x30, 0x35, 0x31, 0x7A, 0x00`. Si la cadena inicia en `0x1000` (memoria little-endian de 32 bits):

     ```
     0x1000: 0x30335343  ("03SC")
     0x1004: 0x007A3135  ("\0z15")
     ```

Salida esperada: ”cs3051z\0”. Bytes: `0x63, 0x73, 0x33, 0x30, 0x35, 0x31, 0x7A, 0x00`. En memoria:

```
0x1000: 0x30337363
0x1004: 0x007A3135
```

Muestre capturas de pantalla de la memoria.

División Entera Sin Signo (Recursiva)

Problema

Descripción

Cree una función **recursiva** en ARM para dividir dos enteros de 32 bits sin signo usando restas sucesivas. Calcule el cociente y el resto.

Entrada y Salida

- `dividendo` (en R0): Número a dividir.
- `divisor` (en R1): Número por el cual se divide.

Al retornar, R0 debe tener el cociente y R1 el resto. Para la recursión, guarde LR y otros registros necesarios en el stack. Restaure R4-R11 si los usa.

Algoritmo Recursivo

Pseudocódigo para División Recursiva

```
function UnsignedDivisionRecursive(dividendo, divisor):
    // División por cero
    if divisor == 0:
        return (cociente=0, resto=dividendo)

    // Caso base: no se puede restar más
    if dividendo < divisor:
        return (cociente=0, resto=dividendo)
    else:
        // Paso recursivo
        nuevoDividendo = dividendo - divisor
        (cocienteParcial, restoParcial) =
            UnsignedDivisionRecursive(nuevoDividendo, divisor)

        cociente = cocienteParcial + 1
        resto = restoParcial
        return (cociente, resto)
```

Puntos clave para ARM:

- **Stack Frame:** Antes de una llamada recursiva (BL), guarde LR y registros que necesite después. Restáurelos al volver.
- El cociente se forma sumando 1 al volver de cada llamada.
- El resto es el `dividendo` del caso base.
- Ejemplo: Dividir 7 por 3.

1. Div(7,3) llama Div(4,3).
2. Div(4,3) llama Div(1,3).
3. Div(1,3) retorna (cociente=0, resto=1) (caso base).
4. Div(4,3) recibe (0,1), calcula cociente=0+1=1. Retorna (1,1).
5. Div(7,3) recibe (1,1), calcula cociente=1+1=2. Retorna (2,1).

Resultado: Cociente = 2, Resto = 1.

Entregables

1. **Código C:** Implementación recursiva.
2. **Código ARM:** Implementación recursiva en ARM, manejando el stack.
3. **Memoria/Stack:** Diagrama del stack para dos niveles de recursión en un caso de prueba.
4. **Prueba en CPUlator:** Muestre R0, R1 (y SP) antes y después para:
   - Caso 1: dividendo = 23, divisor = 4 (Esperado: R0=5, R1=3)
   - Caso 2: dividendo = 10, divisor = 3 (Esperado: R0=3, R1=1)
   - Caso 3: dividendo = 5, divisor = 15 (Esperado: R0=0, R1=5)
   - Caso 4: dividendo = 42, divisor = 0 (Esperado: R0=0, R1=42)

Ordenamiento de Array

Problema

Descripción

Cree una función en ARM que ordene un array de enteros de 32 bits con signo de menor a mayor. Use el algoritmo de **Ordenamiento por Selección (Selection Sort)**. Modifique el array original (in-place).

Entrada y Salida

Prototipo en C

```c
void selection_sort(int *array, int size);
```

- `array` (en R0): Puntero al inicio del array.
- `size` (en R1): Número de elementos.

La función no retorna un valor. Restaure R4-R11 si los usa.

Algoritmo y Detalles

Selection Sort:

1. Para cada posición `i` del array (desde 0 hasta `size-2`):
2. Encuentre el elemento más pequeño en el sub-array `array[i...size-1]`.
3. Intercambie este mínimo con `array[i]`.

Notas:

- Use bucles anidados.
- Para acceder a `array[k]`, la dirección es `base_puntero + (k * 4)`. Use LDR/STR.
- Intercambiar dos elementos requiere un registro temporal (o el stack).
- Si `size` es 0 o 1, no haga nada.

Ejemplo: `[5, 1, 4, 2, 8]` se ordena a `[1, 2, 4, 5, 8]`.

Ejemplo de ejecución (size = 4): Array inicial `[6, 1, 8, 3]`

- **Iteración i = 0:** (Parte no ordenada: `[6, 1, 8, 3]`)
  - Mínimo en `[6, 1, 8, 3]` es **1** (en índice `minIndex`=1).
  - Intercambiar `array[0]` (6) con `array[1]` (1).
  - Array ahora: `[1, 6, 8, 3]`
- **Iteración i = 1:** (Parte no ordenada: `[6, 8, 3]`, que es `array[1..3]`)
  - Mínimo en `[6, 8, 3]` es **3** (en índice original `minIndex`=3).
  - Intercambiar `array[1]` (6) con `array[3]` (3).
  - Array ahora: `[1, 3, 8, 6]`
- **Iteración i = 2:** (Parte no ordenada: `[8, 6]`, que es `array[2..3]`)
  - Mínimo en `[8, 6]` es **6** (en índice original `minIndex`=3).
  - Intercambiar `array[2]` (8) con `array[3]` (6).
  - Array ahora: `[1, 3, 6, 8]`
- El bucle externo termina (se ejecuta hasta `size-2`, que es 2 para `size`=4). Array ordenado: `[1, 3, 6, 8]`.

Entregables

1. **Código C:** Implementación de `selection_sort`.
2. **Código ARM:** Traducción a ensamblador ARM.
3. **Memoria/Stack:** Diagrama del array en memoria antes y después para un caso de prueba. Incluya el stack si lo usa.
4. **Prueba en CPUlator:** Muestre la memoria del array antes y después para:
   - **Caso 1 (Positivos, duplicados):** Array: `[7, 3, 5, 1, 5, 2]` (size=6). Inicia en `0x2000`. Memoria inicial (little-endian):

     ```
     0x2000: 0x00000007
     0x2004: 0x00000003
     0x2008: 0x00000005
     0x200C: 0x00000001
     0x2010: 0x00000005
     0x2014: 0x00000002
     ```

     Esperado: `[1, 2, 3, 5, 5, 7]`
   - **Caso 2 (Con negativos):** Array: `[4, -2, 0, 9, -5]` (size=5). Inicia en `0x2000`. Memoria inicial (little-endian, negativos en complemento a 2):

     ```
     0x2000: 0x00000004
     0x2004: 0xFFFFFFFE  (-2)
     0x2008: 0x00000000
     0x200C: 0x00000009
     0x2010: 0xFFFFFFFB  (-5)
     ```

     Esperado: `[-5, -2, 0, 4, 9]`