Laboratorio 6

Victor Ostolaza
Macarena Oyague

Arquitectura de Computadores

June 22, 2020

Suma de enteros consecutivos

Se ha desarrollado un programa en assembly que realice la suma de enteros consecutivos existentes entre un número A y un número B. Las instrucciones realizadas a este bajo nivel se asemejan a aquellas correspondistes a un lenguaje de programación a un nivel más alto. Las operaciones para MIPS son las siguientes:

```
.text
main:
    #
    # bool $t3, i $t4, acum $ t5
    #
    addi $t0, $t0, 2 # A = 2
    addi $t1, $t1, 5 # B = 5
loop:
    add $t5, $t0, $t4 # X = A + i
    slt $t3, $t1, $t5 # B < X ? 1 : 0
    bgtz $t3, exit    # bool > 0 ? exit : pass
    add $t2, $t2, $t5 # sum = sum + X
    addi $t4, $t4, 1  # i += 1
    j loop
exit:

    # Put your sum S into register $t2
end:
    j   end # Infinite loop at the end of the program.
```

Bkpt	Address	Code	Basic	Source
	0x00003000	0x21080002	addi $8,$8,2	6: addi $t0, $t0, 2 # A = 2
	0x00003004	0x21290005	addi $9,$9,5	7: addi $t1, $t1, 5 # B = 5
	0x00003008	0x010c6820	add $13,$8,$12	9: add $t5, $t0, $t4 # X = A + i
	0x0000300c	0x012d582a	slt $11,$9,$13	10: slt $t3, $t1, $t5 # B < X ? 1 : 0
	0x00003010	0x1d600003	bgtz $11,3	11: bgtz $t3, exit # bool > 0 ? exit : pass
	0x00003014	0x014d5020	add $10,$10,$13	12: add $t2, $t2, $t5 # sum = sum + X
	0x00003018	0x218c0001	addi $12,$12,1	13: addi $t4, $t4, 1 # i += 1
	0x0000301c	0x08000c02	j 0x00003008	14: j loop
	0x00003020	0x08000c08	j 0x00003020	19: j end # Infinite loop at the end of the program.

Estas operaciones pueden ser traducidas matemáticamente a la fórmula:

$$S = A + (A + 1) + (A + 2) + ... + (B1) + B$$

El resultado de las operaciones que realiza en la sección 'main' se pueden ver a continuación, que son el setteo inicial de las variables. Se puede observar que el valor 2 es guardado en $to y el valor 5 en $t1.

Registers

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	2
$t1	9	5
$t2	10	0

La suma de los enteros consecutivos entre A y B se realiza la sección 'loop'. Se settea $t5 como la suma del valor de A ($t0) más el valor del contador ($t4). Luego, se hace una verificación de si el valor del registro $t1 (B) es menor a esté número almacenado en $t5, almacenándose de manera booleana en $t3. Si $t3 es mayor a cero, significa que el número por sumar guardado en $t5 es mayor a B, lo cual significa que el valor de B ya ha sido aumentado en la suma, por lo que sale del loop y se dirige a exit y a end. De lo contrario, el valor se aumentará en el registro $t2, donde se acumula la suma de los enteros consecutivos, y se aumenta en 1 el valor del contador ($t4). Luego realiza un jump hacia loop hasta que se de la condición bgtz que indique la salida.

Con el ejemplo realizado de A = 2 y B = 5 se puede visualizar el comportamiento final por medio del resultado en los registros, lo cual tiene como valor final 12 en la suma almacenada en $t2:

Registers | Coproc 1

Name	Num...	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	2
$t1	9	5
$t2	10	14
$t3	11	1
$t4	12	4
$t5	13	6

Suma de Diferencias Absolutas

El algoritmo de suma de diferencias absolutas sirve para medir la similtúd entre dos imágenes. Este nivel de similitúd se llega a obtener comparando cada pixel de la imágen original con su pixel correspondiente en la imágen a comparar. A cada pixel se le asigna un valor 0 - 16 dependiendo de su valor grayscale. La diferencia absoluta de estos valores de pixeles entre imágenes acumuladas y este número es el nivel de similitúd encontrado.

En la implementación, se asigna al registro $s0 el espacio en memoria 0x00000000 que indica el comienzo del primer vector que va a contener los valores grayscale de left_image. Se procede a insertar estos valores llenando los primeros 36 bits, correspondiendo 4 bits para cada pixel de la imágen:

```
# Inicializamos left_image
lui     $s0, 0x0000 # Address of first element in the vector
ori     $s0, 0x0000
addi   $t0, $0, 5   # left_image[0]
sw      $t0, 0($s0)
addi   $t0, $0, 16 # left_image[1]
sw      $t0, 4($s0)
addi   $t0, $0, 7   # left_image[2]
sw      $t0, 8($s0)
addi   $t0, $0, 1   # left_image[3]
sw      $t0, 12($s0)
addi   $t0, $0, 1   # left_image[4]
sw      $t0, 16($s0)
addi   $t0, $0, 13 # left_image[5]
sw      $t0, 20($s0)
addi   $t0, $0, 2   # left_image[6]
sw      $t0, 24($s0)
addi   $t0, $0, 8   # left_image[7]
sw      $t0, 28($s0)
addi   $t0, $0, 10 # left_image[8]
sw      $t0, 32($s0)
```

Se hace lo mismo para los siguientes 36 llenando los valores de right_image. Se realiza de la manera siguiente porque, como cada imágen ocupa más de un address de memoria, se busca que todos los datos de ambas imágenes sean contiguos, se asignan haciendo referencia nuevamente a $s0 por motivos de simplicidad, pero se almacena el inicio de los datos a guardar de este vector en $s1 mediante un load. Las operaciones de almacenamiento se pueden ver de esta manera:

```
# Inicializamos right_image
lw      $s1, 36($s0) # Address of first element in the vector
addi    $t0, $0, 4          # right_image[0]
sw       $t0, 36($s0)
addi    $t0, $0, 15          # right_image[1]
sw       $t0, 40($s0)
addi    $t0, $0, 8          # rigt_image[2]
   sw          $t0, 44($s0)
addi    $t0, $0, 0          # right_image[3]
   sw          $t0, 48($s0)
addi    $t0, $0, 2          # right_image[4]
   sw          $t0, 52($s0)
addi    $t0, $0, 12          # right_image[5]
   sw          $t0, 56($s0)
addi    $t0, $0, 3          # right_image[6]
   sw          $t0, 60($s0)
addi    $t0, $0, 7          # right_image[7]
   sw          $t0, 64($s0)
addi    $t0, $0, 11          # right_image[8]
   sw          $t0, 68($s0)
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	5	16	7	1	1	13	2	8
0x00000020	10	4	15	8	0	2	12	3
0x00000040	7	11	0	0	0	0	0	0

Se realiza el mismo bucle del código C traduciéndolo a Assembler. Primero se almacena el valor 0 en $s2, y va a ser este registro el que usemos como un iterador hasta que alcance el tamaño del arreglo a iterar, valor que almacenamos en el registro $s3. Una vez asignados estos valores, se inicia el bucle que pase por todos los valores de cada vector, en la posición designada por el iterador en $s2, y llamamos a la función abs_diff pasandole cada pixel junto a su correspondiente en la otra imágen para obtener la diferencia absoluta entre ellos y almacenando este resultado en el arreglo sad_array que hemos inicializado en el espacio en memoria siguiente al arreglo right_image.

```
    # TODO4: Loop over the elements of left_image and right_image
    addi $s2, $0, 0 # $s2 = i = 0
    addi $s3, $0, 9 # $s3 = image_size = 9
loop:
    # Check if we have traversed all the elements
    # of the loop. If so, jump to end_loop:
    beq $s2, $s3, end_loop
    # Load left_image[i] and put the value in the corresponding register
    # before doing the function call
    sll $t0, $s2, 2
    add $t0, $s0, $t0
    lw $a0, 0($t0)
    # Load right_image[i] and put the value in the corresponding register
    lui $s4, 0x0000
    ori $s4, 0x0024
    sll $t0, $s2, 2
    add $t0, $s4, $t0
    lw $a1, 0($t0)
    # Call function
    jal abs_diff
    #Store the returned value in sad_array[i]
    lui $s5, 0x0000
    ori $s5, 0x0048
    sll $t0, $s2, 2
    addu $t0, $s5, $t0
    sw $v0, 0($t0)
    # Increment variable i and repeat loop:
    addi $s2, $s2, 1
    j loop
end_loop:
```

La función abs_diff toma dos números y devuelve el valor absoluto de la diferencia entre ellos.

```
# TODO2: Implement the abs_diff routine here, or use the one provided
# FUNCION PROPIA
abs_diff:
    sub $t1, $a0, $a1
    abs $v0, $t1
    jr $ra
```

Finalmente, se llama a la función recursive_sum para que sume todos los valores del arreglo sad_array y guarde el resultado final en el registro $t2.

En el caso de la función recursive_sum, recibe como argumento $a0 el arreglo con las diferencias absolutas y en $a1 el tamaño del arreglo. Realiza un bucle con una duración igual al valor de $a1 y en cada iteración, le suma al registo

```
    #TODO5: Call recursive_sum and store the result in $t2
    #Calculate the base address of sad_array (first argument
    #of the function call)and store in the corresponding register
    addi $a0, $s5, 0
    # Prepare the second argument of the function call: the size of the array
    addi $a1, $s3, 0
    # Call function
    jal recursive_sum
    #Store the returned value in $t2
    addi $t2, $v0, 0
end:
    j   end # Infinite loop at the end of the program.
```

$v0, el valor almacenado en ese índice del arreglo y una vez terminado devuelve la suma de todos los valores.

```
# TODO3: Implement the recursive_sum routine here, or use the one provided
recursive_sum:
        addi $sp, $sp, -8       # Adjust sp
        addi $t0, $a1, -1       # Compute size - 1
        sw   $t0, 0($sp)        # Save size - 1 to stack
        sw   $ra, 4($sp)        # Save return address
        bne  $a1, $zero, else   # size == 0 ?
        addi  $v0, $0, 0        # If size == 0, set return value to 0
        addi $sp, $sp, 8        # Adjust sp
        jr $ra                  # Return
else:
        add  $a1, $t0, $0       #update the second argument
        jal   recursive_sum
        lw    $t0, 0($sp)       # Restore size - 1 from stack
        sll  $t1, $t0, 2        # Multiply size by 4
        add   $t1, $t1, $a0     # Compute & arr[ size - 1 ]
        lw    $t2, 0($t1)       # t2 = arr[ size - 1 ]
        add   $v0, $v0, $t2     # retval = $v0 + arr[size - 1]
        lw    $ra, 4($sp)       # restore return address from stack
        addi $sp, $sp, 8        # Adjust sp
        jr $ra                  # Return
```