

Índice

1. Suma consecutiva de enteros	2
1.1. Código en C	2
1.2. Transformación a Assembly	2
1.2.1. main	2
1.2.2. consecutiveAdd	4
1.2.3. Loop	4
1.3. Prueba de funcionalidad	4
2. Suma de Diferencias Absolutas (SAD)	5
2.1. Código en C	5
2.2. TODO1: Inicializando datos en memoria	6
2.3. TODO2: Función abs_diff()	7
2.4. TODO3: recursive_sum()	7
2.5. TODO4: Función Main	8
2.6. TODO5: Sección <i>endloop</i> :	9
2.7. Ejecución	9

Todo el código puede ser encontra el siguiente repositorio de GitHub

1. Suma consecutiva de enteros

El objetivo del programa es encontrar la suma de enteros positivos desde A hacia B:
 $S = A + (A + 1) + (A + 2) + \dots (B + 1) + B$. Para ello implementaremos un programa en Assembly usando solo las instrucciones que han sido implementadas en el Lab05 en conjunto de J, ADDI y BEQ.

1.1. Código en C

En primer lugar, creamos un código que realice la suma de enteros consecutivos que pueda ser utilizada con las funciones implementadas en nuestro ALU.

```
#include <stdio.h>
int consecutiveAdd(int a, int b){
    int sum = 0;
    for (int i = a ; i <= b ; i++){
        sum = sum + i;
    }
    return sum;
}
int main(void){
    int A = 5;
    int B = 10;
    consecutiveAdd(A,B)
    return 0;
}
```

1.2. Transformación a Assembly

Procedemos a crear nuestro archivo `ConsecutiveAdd.asm` donde transformaremos el código creado en C a Assembly.

1.2.1. main

Primero, crearemos el módulo main donde asignaremos los dos números enteros que serán calculados (A y B) y saltaremos a la función creada (`consecutiveAdd`). Vease a continuación:

```
.text
main:
    addi $s1, $s1,5          #A
```

```
addi $s2, $s2, 10      #B
```

```
j consecutiveAdd
```

1.2.2. consecutiveAdd

Para crear nuestra función `consecutiveAdd`, usamos una variable temporal (`$t2`) para almacenar el resultado de nuestra suma de enteros consecutivos. Luego asignamos que `i`, el valor por el que contara el número de veces que el loop se repetira, sea igual a `A`. Al valor de `B`, le agregamos 1 para poder hacer la comparación en el Loop. Por último hacemos un `j` al módulo mencionado con anterioridad.

```
consecutiveAdd:
    addi $t2, $t2, 0      #sum
    addi $t1, $s1, 0      #i = a
    addi $t3, $s2, 1      #b+1
    j Loop                #Initialazing loop
```

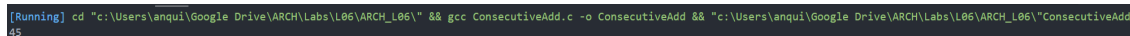
1.2.3. Loop

Una vez en el módulo `Loop`, procedemos a crear nuestro condicional `beq` dónde preguntaremos en cada repetición si la variable `i == (b+1)`. Luego procedemos a realizar a la suma consecutiva del valor de `i` (`A`) en `sum` hasta llegar al entero `B+1` dónde parara la repetición. Para ello, usaremos la función `addi` para aumentar el valor de nuestra var `i`. Por último agregamos el `j` para crear la repetición de la misma.

```
Loop:
    beq $t1, $t3, end     # if i == b+1
    add $t2, $t2, $t1     # sum = sum + i
    addi $t1, $t1, 1      # i = i + 1
    j Loop
end:
    j end
```

1.3. Prueba de funcionalidad

Ejecutamos el código en C para tener referencia del resultado obtenido por la función realizada.



```
[Running] cd "c:\Users\anqui\Google Drive\ARCH\Labs\L06\ARCH_L06\" && gcc ConsecutiveAdd.c -o ConsecutiveAdd && "c:\Users\anqui\Google Drive\ARCH\Labs\L06\ARCH_L06\ConsecutiveAdd
45
```

Figura 1: Compilamos y ejecutamos el código `consecutiveAdd.c`

Finalmente, ensamblamos y ejecutamos el código `.ASM` en el programa `MARS` y obtenemos el resultado calculado en el programa inicial, que se encuentra en `sum` (`$t2`).

Registers	Coproc 1	Coproc 0
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	0
\$t1	9	11
\$t2	10	45
\$t3	11	11
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	5
\$s2	18	10
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268469224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194348
hi		0
lo		0

Figura 2: Asamblaje y ejecución del código `consecutiveAdd.asm` en MARS

2. Suma de Diferencias Absolutas (SAD)

Implementación del algoritmo SAD

2.1. Código en C

En primer lugar, se analiza el código realizado en lenguaje C del algoritmo SAD.

```
#include <stdio.h>
#include <stdlib.h>

int abs_diff(int pixel_left, int pixel_right){
    int abs_diff = abs(pixel_left - pixel_right);
    return abs_diff;
}

int recursive_sum(int arr[],int size){
    if (size==0)
        return 0;
    else
        return recursive_sum(arr,size-1) + arr[size-1];
}

int main(){
    int sad_array [9];
```

```
int image_size = 9;
int sad_value;
int left_image[9] = {5 , 16 , 7 , 1 , 1 , 13 , 2 , 8 , 10};
int right_image[9] = {4 , 15 , 8 , 0 , 2 , 12 , 3 , 7 , 11};
for (int i = 0 ; i < image_size ; i++)
    sad_array[i] = abs_diff(left_image[i],right_image[i]);
sad_value = recursive_sum (sad_array,image_size);
}
```

2.2. TODO1: Inicializando datos en memoria

Establecemos que el **data segment** comienza con el address **0x10010000**. Asimismo, se guarda los datos del arreglo **left_image** a partir de ese address. Luego, se guarda los elementos del **right_image** en la siguiente posición memoria. Finalmente, se guarda el base address del **sad_array**, el cual es la siguiente posición de memoria del anterior.

```
lui    $s0,0x00000000 # Address of first element in the vector
ori    $s0,0x10010000
addi   $t0,$0,5        # left_image[0]
sw     $t0,0($s0)
addi   $t0,$0,16       # left_image[1]
sw     $t0,4($s0)
# TODO1: initilize the rest of the memory.
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)

lui    $s1, 0x00000000 #Addres of firt element in the vector
ori    $s1, 0x10010024

addi   $t0, $0, 4      # right_image[0]
```

```
sw      $t0, 0($s1)
addi    $t0, $0, 15      # right_image[1]
sw      $t0, 4($s1)
addi    $t0, $0, 8       # right_image[2]
sw      $t0, 8($s1)
addi    $t0, $0, 0       # right_image[3]
sw      $t0, 12($s1)
addi    $t0, $0, 2       # right_image[4]
sw      $t0, 16($s1)
addi    $t0, $0, 12      # right_image[5]
sw      $t0, 20($s1)
addi    $t0, $0, 3       # right_image[6]
sw      $t0, 24($s1)
addi    $t0, $0, 7       # right_image[7]
sw      $t0, 28($s1)
addi    $t0, $0, 11      # right_image[8]
sw      $t0, 32($s1)

lui     $s2, 0x00000000   #sad array
ori     $s2, 0x10010048
```

2.3. TODO2: Función `abs_diff()`

Para la implementación en assembly de la función `abs_diff()` nos apoyamos de la instrucción implementada `abs`. Función propia.

```
abs_diff:
sub     $t1, $a0, $a1
abs     $v0, $t1          #v0 = abs(pixel_left - pixel_right)
jr      $ra
```

2.4. TODO3: `recursive_sum()`

La implementación del `recursive_sum()` es la siguiente:

```
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 = lv0 + arr[size - 1]
    lw  $ra, 4($sp)           # restore return address from stack
    addi $sp, $sp, 8          # Adjust sp
    jr $ra                    # Return
```

2.5. TODO4: Función Main

Antes de completar la sección "loop:" se define lo siguiente:

```
addi $s3, $0, 0 # i = 0
addi $s4, $0, 9 # image_size = 9
j loop
```

Luego de ello, se completa la sección "loop:" correctamente:

```
loop:
    # Check if we have traverse all the elements
    # of the loop. If so, jump to end_loop:
    slt $t0,$s3,$s4
    beq $t0,$0,end_loop # i < image_size
    # Load left_image[i] and put the value in the corresponding register
    # before doing the function call
    sll $t4,$s3,2
    add $t1,$s0,$t4
    lw $a0,0($t1)        #a0 = left_image[i]
    # Load right_image[i] and put the value in the corresponding register
    add $t2,$s1,$t4
    lw $a1,0($t2)        #a1 = right_image[i]
    # Call abs_diff
    jal abs_diff
    #Store the returned value in sad_array[i]
    add $t3,$s2,$t4
    sw $v0,0($t3)        #sad_array[i] = v0
    # Increment variable i and repeat loop:
```



```
addi $s3,$s3,1
j loop
```

2.6. TODO5: Sección *endloop*:

Finalmente, se completa la sección *endloop*: en donde se prepara los argumentos para llamar la función **recursive_sum()** y guardar el resultado en **\$t2**.

```
end_loop:
    addi $a0,$s2,0                # Parametro base address : sad_array

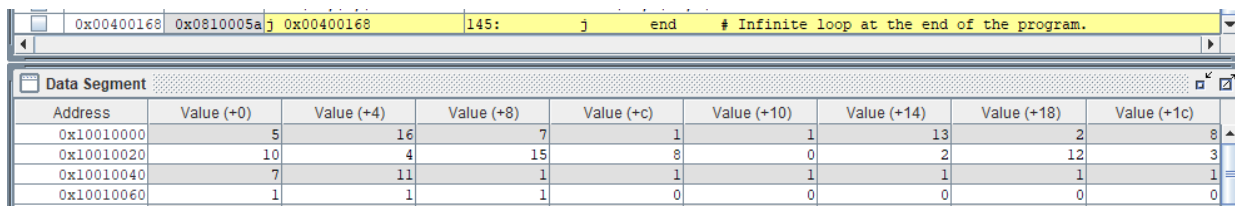
    # Prepare the second argument of the function call: the size of the array
    add $a1,$s4,$0                # Parametro image_size

    # Call to funtion
    jal recursive_sum

    #Store the returned value in t2
    add $t2, $v0, $0
```

2.7. Ejecución

Luego de definir las instrucciones, se ejecuta el programa y estos son los resultados obtenidos:



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	5	16	7	1	1	13	2	8
0x10010020	10	4	15	8	0	2	12	3
0x10010040	7	11	1	1	1	1	1	1
0x10010060	1	1	1	0	0	0	0	0

Figura 3



Register	Value
\$v0	2

Figura 4