

Lab02 - Programación bajo nivel

Arquitectura de Computadores

Jhon Alexander Botero Gomez
Jhonatan Andrés Granda Orrego

Objetivo

- Estudiar la arquitectura del conjunto de instrucciones MIPS de 32 bits.
- Diseñar, codificar, ensamblar, simular y depurar programas escritos en lenguaje ensamblador MIPS.
- Familiarizarse con el uso de un entorno de desarrollo de software de bajo nivel.

Descripción

En esta práctica cada equipo de trabajo debe diseñar, codificar, simular y verificar un programa escrito en lenguaje ensamblador MIPS32 que esté en capacidad de organizar un listado de números.

Procedimiento

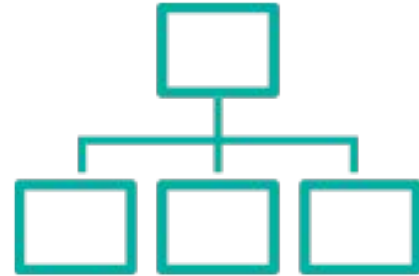
Cada equipo de trabajo debe desarrollar el programa en lenguaje ensamblador MIPS empleando la herramienta MARS1. Las condiciones a las que se debe ajustar el desarrollo son las siguientes.

- a. Se ordenará el vector implementando un algoritmo de ordenamiento en particular, de acuerdo con el identificador de cada equipo de trabajo, como se establece en la Sección 4 de esta guía.
- b. La lista de números desordenados debe estar en un archivo de texto llamado numeros.txt
- c. El listado de números ordenado debe ser volcado a un archivo de texto plano con nombre equipoXX.txt. Los números estarán separadas por un separador que el equipo elija.
- d. La programación en bajo nivel debe ser estructurada, es decir, debe estar basada en el uso de procedimientos, de manera que la organización modular del programa facilite su comprensión y favorezca la reutilización de código a lo largo del mismo. Se debe usar por lo menos 1 vez el uso de la Pila.
- e. El código desarrollado debe seguir la convención para el uso de registros MIPS.
- f. El código fuente en ensamblador debe estar comentado de manera asertiva.

Consideraciones

Son diversos los aspectos específicos de la implementación de la solución al problema que tendrán que ser definidos por el equipo de trabajo. En todos los casos, las decisiones de diseño deben estar ampliamente explicadas y justificadas. La calidad de estas decisiones será valorada como parte de la calificación del laboratorio.

También serán valoradas la creatividad y elegancia en la programación, así como la calidad de la interacción con el usuario mediante el uso de los servicios SYSCALL. El rendimiento del programa también será tenido en cuenta, entendiendo este como el número de instrucciones ejecutadas para completarlo.



Asignaciones

El equipo correspondiente es el grupo 17, dado que son 6 posibles asignaciones $17\%6 \Rightarrow 5$.

La asignación correspondiente es la número 5, algoritmo de ordenamiento **MergeSort** Descendente

Tabla 1. Asignaciones para los equipos de trabajo

# de asignación	Algoritmo de ordenamiento	Ordenamiento
0	Quick	Ascendente
1	Heap	Ascendente
2	Merge	Ascendente
3	Quick	Descendente
4	Heap	Descendente
5	Merge	Descendente

Implementación en C

El algoritmo implementado en el lenguaje C, nos da una idea de como es el funcionamiento general de este método de ordenamiento y un esquema amplio de las partes del algoritmo en alto nivel.

Este algoritmo implica la técnica de programación “Divide y vencerás”

Primeramente tenemos la instancia principal o el método main en donde se realizan 3 procesos importantes:

1. Carga del vector
2. Llamado a la función mergeSort()
3. Mostrar el arreglo ordenado

```
// Clase main
int main()
{
    // tengo un arreglo de 6 elementos
    int arreglo[] = {25, 65, 85, 156, 3, 7};
    int n = sizeof(arreglo) / sizeof(arreglo[0]);

    // Llamar implementación de la función MergeSort
    mergeSort(arreglo, 0, n - 1);

    for (int i = 0; i < n; i++)
        printf("%d ", arreglo[i]);
    return 0;
}
```

Implementación en C

La función mergeSort recibe 3 parametros para empezar con el ordenamiento, el primero es el arreglo a ordenar, segundo el inicio (índice base) y el final (índice final).

Valida que el arreglo aun pueda dividirse usando recursión, y luego calcula el umbral de división del arreglo en la función. Una vez calculado este punto medio, el algoritmo hace nuevamente 2 llamadas, para realizar el ordenamiento de la primera mitad y de la segunda mitad, para por último llamar el método merge y unir las listas que se generaron.

```
void mergeSort(int arreglo[], int inicio, int final)
{
    if (inicio < final)
    {
        // Calcular el punto medio
        int mitad = inicio + (final - inicio) / 2;
        // Ordenar la primera y la segunda mitad
        mergeSort(arreglo, inicio, mitad);
        mergeSort(arreglo, mitad + 1, final);
        // Fusionar las mitades ordenadas
        merge(arreglo, inicio, mitad, final);
    }
}
```

Implementación en C

En esta primera parte se tienen la declaración de los contadores necesarios en la función, los umbrales de la división del arreglo.

Luego se tienen dos subarreglos para la primera mitad y la segunda mitad.

Se rellenan los arreglos temporales, se usa i para el arreglo izquierdo, y la variable j para el arreglo derecho.

Por último inicializamos nuevamente las variables que apuntan los índices de los arreglos, y en la siguiente parte se procede con la mezcla

```
void merge(int arreglo[], int inicio, int mitad, int final){
    int i, j, k;
    int n1 = mitad - inicio + 1;
    int n2 = final - mitad;
    // Crear los arreglos temporales
    int arregloIzq[n1], arregloDer[n2];
    // Mapear los datos del arreglo original en los temporales
    for (i = 0; i < n1; i++)
        arregloIzq[i] = arreglo[inicio + i];
    for (j = 0; j < n2; j++)
        arregloDer[j] = arreglo[mitad + 1 + j];
    // Mezclar los arreglos temporales en el orden{asc, desc}
    i = 0;
    j = 0;
    k = inicio;
    .
    .
    .
```


Implementación en C

En esta última sección como primero el fragmento de código es la que ordena de manera descendente o ascendente.

```
if (arreglozq[i] >= arregloDer[j])
```

Para este caso es descendente.
//Si utilizo <= este operador cambiaria a ascendente

Y luego los elementos restantes se mapean en el arreglo general

```
while (i < n1 && j < n2){
    if (arreglozq[i] >= arregloDer[j]) {
        arreglo[k] = arreglozq[i];
        i++;
    }else {
        arreglo[k] = arregloDer[j];
        j++;
    } k++;
}
...
// Copia los elementos restantes de arreglozq[], si los hay
while (i < n1){
    arreglo[k] = arreglozq[i];
    i++;
    k++;
}
// Copiar los elementos restantes de arregloDer[], si los hay
while (j < n2){
    arreglo[k] = arregloDer[j];
    j++;
    k++;
}
}
```

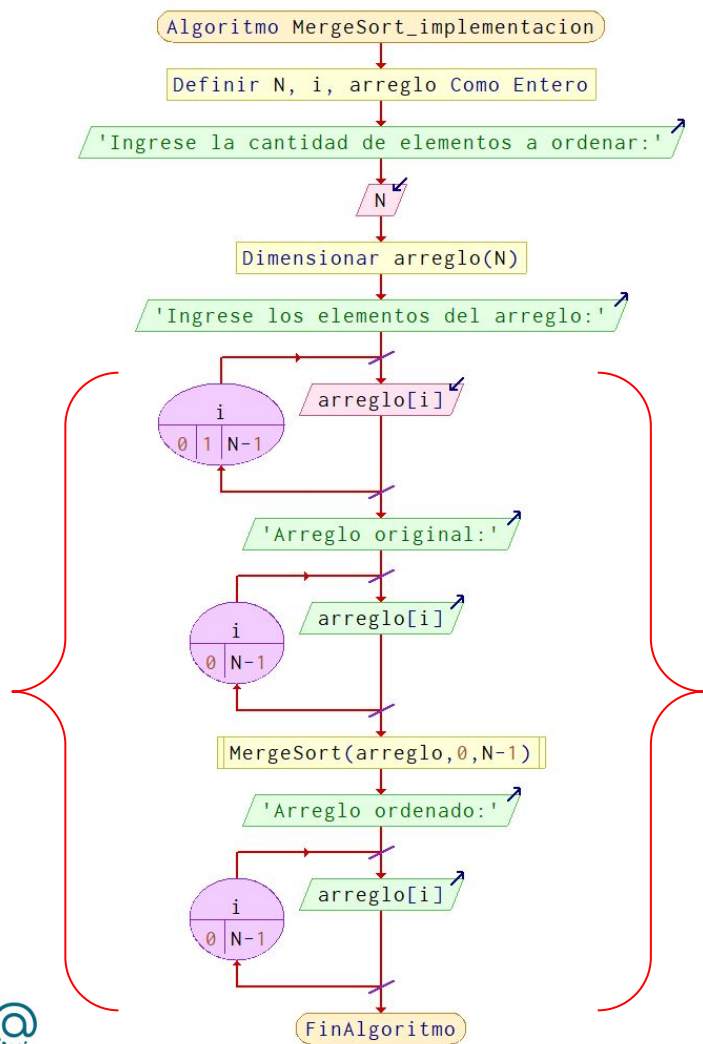


Diagrama flujo

En el diagrama de flujo se puede ver una línea principalmente secuencial, y solo dos estructuras repetitivas para mapear el arreglo ordenado y desordenado.

La parte señalada es la vital y la que tiene las esencia de la ejecución, pues en esta se quiere ilustrar cómo llenamos el arreglo, ordenamos, y luego lo mostramos arreglado.

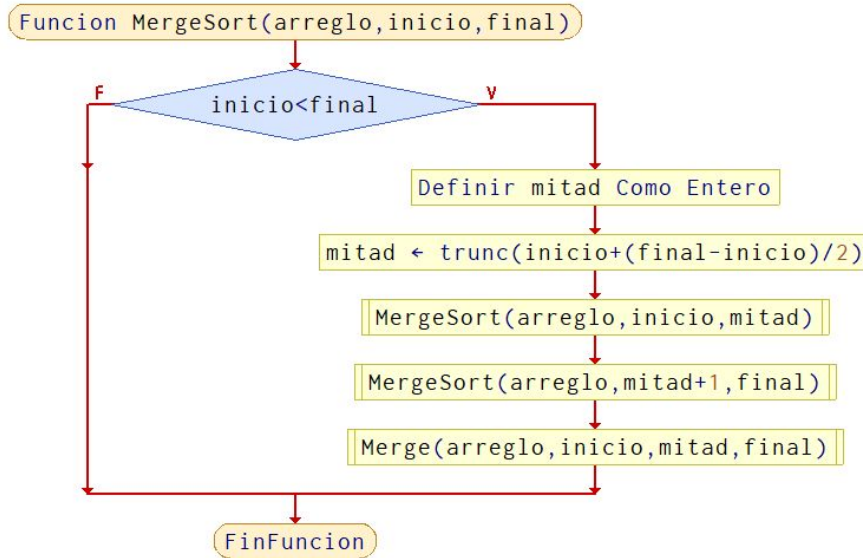
Pseudocódigo

```
Algoritmo MergeSort_implementacion
  Definir N,i,arreglo Como Entero
    Escribir "Ingrese la cantidad de elementos a
ordenar:"
    Leer N
    Dimension arreglo[N]
    Escribir "Ingrese los elementos del arreglo:"
    Para i <- 0 Hasta N-1 Con Paso 1 Hacer
      Leer arreglo[i]
    Fin Para
    Escribir "Arreglo original:"
    Para i <- 0 Hasta N-1 Hacer
      Escribir arreglo[i]
    Fin Para
    MergeSort(arreglo, 0, N-1)
    Escribir "Arreglo ordenado:"
    Para i <- 0 Hasta N-1 Hacer
      Escribir arreglo[i]
    Fin Para
FinAlgoritmo
```

El pseudocódigo en la clase principal puede ingresar el número de elementos que desea mapear en el arreglo. También es importante ver la definición de las variables iniciales en el programa. N, arreglo[] <= indispensables

La parte más relevante destaca en este fragmento de código, donde se muestra el arreglo, se llama a la función mergeSort, y luego se imprime el arreglo ordenado

Diagrama flujo



En esta sección del programa se puede observar directamente la recursividad del algoritmo y la condición base se da cuando se tengan sublistas con 1 elemento. `inicio = final` finaliza la recursividad.

Se hacen 2 llamados recursivos a la misma función, el primero aplica ordenamiento a la primera mitad, y el segundo para el arreglo desde la mitad hasta el final, y luego se llama a la función **merge**.

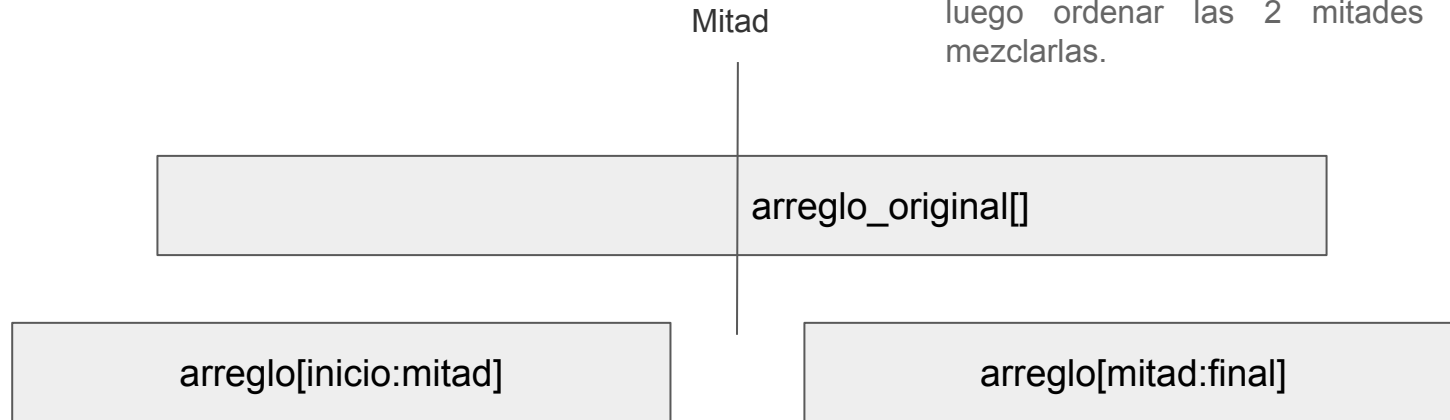
Pseudocódigo

```
Funcion MergeSort(arreglo, inicio, final)
  Si inicio < final Entonces
    Definir mitad Como Entero
    mitad <- trunc((inicio + final) / 2)
    MergeSort(arreglo, inicio, mitad)
    MergeSort(arreglo, mitad + 1, final)
    Merge(arreglo, inicio, mitad, final)
  Fin Si
FinFuncion
```

Para esta instrucción se recibe los tres parámetros enviados desde el main al llamado de la función:

- arreglo[]
- inicio
- final

Y la función tiene la tarea de calcular la mitad del arreglo original, para luego ordenar las 2 mitades y mezclarlas.



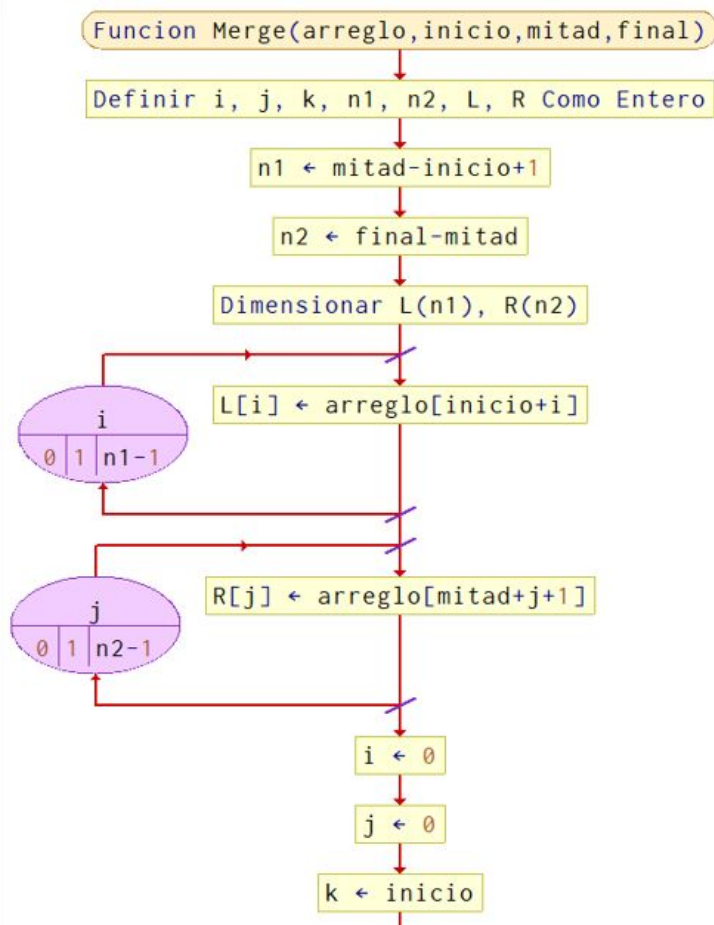


Diagrama flujo

Por último la función **merge**, en esta sección del programa se presenta una secuencialidad muy marcada, ya que recibe 4 parámetros enviados desde **mergeSort**: **arreglo, inicio, mitad, final**.

Se definen las contadoras y las variables para recorrer hasta el umbral que parte el arreglo en subarreglos.

Y para cada subarreglo se le asigna la longitud y se mapean los elementos correspondientes.

Después de haber llenado cada vector, se inicializan nuevamente las variables contadoras, para el proceso de mezclado.

Pseudocódigo

Funcion Merge(arreglo, inicio, mitad, final)

Definir i, j, k, n1, n2, L, R Como Entero

n1 <- mitad - inicio + 1

n2 <- final - mitad

Dimension L[n1], R[n2]

Para i <- 0 Hasta n1-1 Con Paso 1 Hacer

L[i] <- arreglo[inicio + i]

Fin Para

Para j <- 0 Hasta n2-1 Con Paso 1 Hacer

R[j] <- arreglo[mitad + j + 1]

Fin Para

i <- 0

j <- 0

k <- inicio

...

El pseudocódigo implica la esencia importante del algoritmo por mezcla, en este caso declaramos los arreglos temporales para las particiones recursivas que genera el algoritmo y además de las variables contadoras y de ubicación en los punteros necesarias para el algoritmo

En primer lugar las 2 sublistas se mapean los elementos del arreglo que llega por argumento

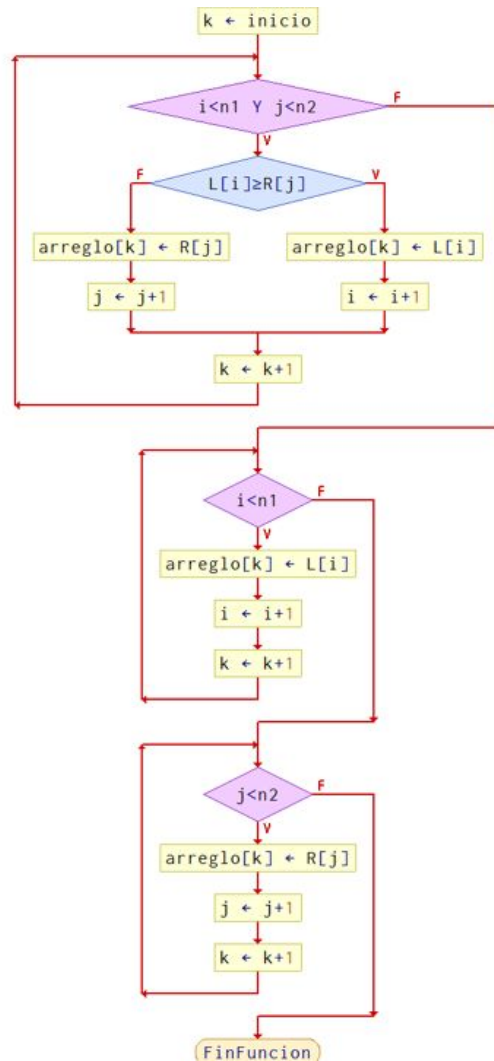
Diagrama flujo

La última sección de la función **merge**, presenta una secuencia de ciclos, en la que se ordenan los subarreglos, y luego se terminan de mezclar.

Se definen las contadoras y las variables para recorrer hasta el umbral que parte el arreglo en subarreglos.

Y para cada subarreglo se le asigna la longitud y se mapean los elementos correspondientes.

Después de haber llenado cada vector, se inicializan nuevamente las variables contadoras, para el proceso de mezclado.



Pseudocódigo

El pseudocódigo implica la esencia importante del algoritmo por mezcla

```
...
Mientras i < n1 Y j < n2 Hacer
    //Implementación para que el algoritmo sea
    descendente Left debe ser mayor que Righth L[i] >= R[j]
    Si L[i] >= R[j] Entonces
        arreglo[k] <- L[i]
        i <- i + 1
    Sino
        arreglo[k] <- R[j]
        j <- j + 1
    Fin Si
    k <- k + 1
Fin Mientras
```

```
Mientras i < n1 Hacer
    arreglo[k] <- L[i]
    i <- i + 1
    k <- k + 1
Fin Mientras
```

```
Mientras j < n2 Hacer
    arreglo[k] <- R[j]
    j <- j + 1
    k <- k + 1
Fin Mientras
```

FinFuncion

Proceso de compilación

Lenguaje de
alto nivel



Lenguaje
ensamblador

Pseudocódigo

Algoritmo MergeSort_implementacion

Definir N,i,arreglo Como Entero

Escribir "Arreglo original:"

Para i <- 0 Hasta N-1 Hacer

Escribir arreglo[i]

Fin Para

MergeSort(arreglo, 0, N-1)

Escribir "Arreglo ordenado:"

Para i <- 0 Hasta N-1 Hacer

Escribir arreglo[i]

Fin Para

FinAlgoritmo

MIPS Assembler

main:

Preparar argumentos para mergeSort

la \$a0, array # Dirección base del arreglo #Arreglo

li \$a1, 0 # inicio = 0 #inicio #load immediate

lw \$a2, size # Cargar tamaño #n, length

addi \$a2, \$a2, -1 # final = tamaño - 1 #n-1

Llamar a mergeSort

jal mergeSort

Imprimir el arreglo ordenado

jal printArray

Terminar programa

li \$v0, 10 #finaliza la ejecución

syscall

Pseudocódigo

```
Funcion MergeSort(arreglo, inicio, final)
  Si inicio < final Entonces
    Definir mitad Como Entero
    mitad <- trunc((inicio + final) / 2)
    MergeSort(arreglo, inicio, mitad)
    MergeSort(arreglo, mitad + 1, final)
  Fin Si
FinFuncion
```

MIPS Assembler

mergeSort:

```
# Verificar si inicio < final
slt $t0, $a1, $a2 #a1-inicio a2-final(n-1)
#si t0 == zero entonces a1 no es menor, entonces finaliza lleva
cero
```

```
beq $t0, $zero, mergeSortReturn #Incumplir la condición
```

```
# Guardar registros en la pila
```

```
#Debo separar 4 registros por eso debo moverme a la siguiente
posición
```

```
#Debo correr 4 registros
```

```
addi $sp, $sp, -16
```

```
sw $ra, 12($sp) #guardar ra
```

```
sw $a1, 8($sp) #guardar a1 -> inicio en la pila
```

```
sw $a2, 4($sp) #guardar a2 -> final en la pila
```

```
# Calcular mitad = (inicio + final) / 2
```

```
add $t0, $a1, $a2 #t0 = inicio + final
```

```
srl $t0, $t0, 1 # mitad = (inicio + final) / 2
```

```
sw $t0, 0($sp) # Guardar mitad
```

```
#mitad es un argumento para el siguiente llamado de la función
```

```
# Primera llamada recursiva: mergeSort(arreglo, inicio, mitad)
```

```
add $a2, $t0, $zero # final = mitad
```

```
jal mergeSort
```

...

Pseudocódigo

```
Funcion MergeSort(arreglo, inicio, final)
  Si inicio < final Entonces
    Definir mitad Como Entero
    mitad <- trunc((inicio + final) / 2)
    MergeSort(arreglo, inicio, mitad)
    MergeSort(arreglo, mitad + 1, final)
    Merge(arreglo, inicio, mitad, final)
  Fin Si
FinFuncion
```

MIPS Assembler

```
...
# Segunda llamada recursiva: mergeSort(arreglo, mitad + 1, final)
lw $t0, 0($sp)      # Pop mitad de la pila, usar en este llamado
#no afectar a $a2
addi $a1, $t0, 1     # inicio = mitad + 1 #a1 -> nuevo inicio
lw $a2, 4($sp)       # Pop final original
jal mergeSort

# Preparar argumentos para merge
lw $a1, 8($sp)       # Pop inicio original
lw $a2, 4($sp)       # Pop final original
lw $a3, 0($sp)       # mitad
jal merge

#Debo traer $ra -- para realzar el jr
lw $ra, 12($sp)      #liberar la pila

# liberar registros y retornar
addi $sp, $sp, 16
```

mergeSortReturn:

```
jr $ra
#no cumplimiento del condicional en la funcion mergeSort()
#Entregue el control a main
```

Pseudocódigo

Funcion Merge(arreglo, inicio, mitad, final)

Definir i, j, k, n1, n2, L, R Como Entero

$n1 \leftarrow \text{mitad} - \text{inicio} + 1$

$n2 \leftarrow \text{final} - \text{mitad}$

Dimension $L[n1]$, $R[n2]$

Para $i \leftarrow 0$ Hasta $n1-1$ Con Paso 1 Hacer

$L[i] \leftarrow \text{arreglo}[\text{inicio} + i]$

Fin Para

Para $j \leftarrow 0$ Hasta $n2-1$ Con Paso 1 Hacer

$R[j] \leftarrow \text{arreglo}[\text{mitad} + j + 1]$

Fin Para

$i \leftarrow 0$

$j \leftarrow 0$

$k \leftarrow \text{inicio}$

...

MIPS Assembler

merge:

Guardar registros

Separar el espacio en la pila

Para este caso necesito guardar 7 registros

addi \$sp, \$sp, -28

sw \$ra, 24(\$sp) #registro ra

sw \$s0, 20(\$sp) #guardar inicio

sw \$s1, 16(\$sp) #guardar final

sw \$s2, 12(\$sp) #umbral para la primera mitad

sw \$s3, 8(\$sp) #umbral para la segunda mitad

sw \$s4, 4(\$sp) #Tomar el puntero base del vector Izq

sw \$s5, 0(\$sp) #Tomar el puntero base del vector Der

Calcular tamaños umbrales

sub \$s2, \$a3, \$a1 # $n1 = \text{mitad} - \text{inicio}$

addi \$s2, \$s2, 1 # $n1 = \text{mitad} - \text{inicio} + 1$

sub \$s3, \$a2, \$a3 # $n2 = \text{final} - \text{mitad}$

Guardar índices originales

add \$s0, \$a1, \$zero # $s0 = \text{inicio}$

add \$s1, \$a2, \$zero # $s1 = \text{final}$

Copiar elementos al arreglo izquierdo

la \$s4, Left # $s4 = \text{dirección base de la sublista izq}$

add \$t0, \$zero, \$zero # $i = 0$

add \$t1, \$s0, \$zero # $k = \text{inicio}$

Pseudocódigo

Funcion Merge(arreglo, inicio, mitad, final)

Definir i, j, k, n1, n2, L, R Como Entero

$n1 \leftarrow \text{mitad} - \text{inicio} + 1$

$n2 \leftarrow \text{final} - \text{mitad}$

Dimension L[n1], R[n2]

Para i <- 0 Hasta n1-1 Con Paso 1 Hacer

L[i] <- arreglo[inicio + i]

Fin Para

Para j <- 0 Hasta n2-1 Con Paso 1 Hacer

R[j] <- arreglo[mitad + j + 1]

Fin Para

i <- 0

j <- 0

k <- inicio

...

copiarArregloLzq:

```
beq $t0, $s2, copiarArregloDerP # beq i == n1, si es igual salta
# Dado que ya debe seguir con la otra mitad del arreglo
sll $t2, $t1, 2 # a $t2 le lleva la i * 4
add $t2, $a0, $t2 # puntero del arreglo general lo reasigno
lw $t3, 0($t2) # traer a un registro el valor de la lista
sll $t2, $t0, 2 # luego cambio a la siguiente dirección
add $t2, $s4, $t2 # y debo situarme en la sublista
sw $t3, 0($t2) # para escribir ese valor en la sublista
addi $t0, $t0, 1 # aumentar i en 1
addi $t1, $t1, 1 # Aumentar k en 1
j copiarArregloLzq
```

copiarArregloDerP:

```
la $s5, Right # s5 = Dirección base de la sublista derecha
add $t0, $zero, $zero # i = 0
addi $t1, $a3, 1 # k = mitad + 1
```

Pseudocódigo

...

Mientras $i < n1$ Y $j < n2$ Hacer

Si $L[i] \geq R[j]$ Entonces

arreglo[k] \leftarrow L[i]

$i \leftarrow i + 1$

Sino

arreglo[k] \leftarrow R[j]

$j \leftarrow j + 1$

Fin Si

$k \leftarrow k + 1$

Fin Mientras

Mientras $i < n1$ Hacer

arreglo[k] \leftarrow L[i]

$i \leftarrow i + 1$

$k \leftarrow k + 1$

Fin Mientras

Mientras $j < n2$ Hacer

arreglo[k] \leftarrow R[j]

$j \leftarrow j + 1$

$k \leftarrow k + 1$

Fin Mientras

MIPS Assembler

mergeArrays:

Verificar si se termino algun subarreglo

beq \$t0, \$s2, copiarRestanteDer # si $i == n1$ (umbral izquierdo)

beq \$t1, \$s3, copiarRestanteIzq # j == n2 (umbral derecho)

Cargar y comparar elementos

Proceso comparativo del algoritmo

sll \$t3, \$t0, 2

add \$t3, \$s4, \$t3

muevo el puntero para el vector izquierdo

lw \$t3, 0(\$t3)

t3 = arregloIzquierdo[i]

sll \$t4, \$t1, 2

add \$t4, \$s5, \$t4

muevo el puntero para el vector derecho

lw \$t4, 0(\$t4)

t4 = arregloDerecho[j]

Comparar y copiar el elemento mayor (orden descendente)

slt \$t5, \$t3, \$t4

Cambiado para orden descendente \$t4, \$t3

beq \$t5, \$zero, copiarIzquierdo

Llamar a la etiqueta izquierda

Pseudocódigo

...
Mientras $i < n1$ Y $j < n2$ Hacer

Si $L[i] \geq R[j]$ Entonces

arreglo[k] <- L[i]

i <- i + 1

Sino

arreglo[k] <- R[j]

j <- j + 1

Fin Si

k <- k + 1

Fin Mientras

Mientras $i < n1$ Hacer

arreglo[k] <- L[i]

i <- i + 1

k <- k + 1

Fin Mientras

Mientras $j < n2$ Hacer

arreglo[k] <- R[j]

j <- j + 1

k <- k + 1

Fin Mientras

FinFuncion

MIPS Assembler

copiarDerecho:

Inicialmente k = mitad + 1

sll \$t5, \$t2, 2 #temporal5 con el valor inicial de k *4

add \$t5, \$a0, \$t5 #muevo el puntero del arreglo

sw \$t4, 0(\$t5) #guardar el registro del puntero a la sublista

addi \$t1, \$t1, 1 #aumentar la variable contadora j

j mergeContinue

copiarIzquierdo:

sll \$t5, \$t2, 2 #temporal5 con el valor inicial de k *4

add \$t5, \$a0, \$t5 #muevo el puntero del arreglo

\$t3 puntero izquierdo

sw \$t3, 0(\$t5) #guardar el registro del puntero a la sublista

addi \$t0, \$t0, 1 #aumentar i

mergeContinue:

addi \$t2, \$t2, 1 #aumento en K

j mergeArrays

Pseudocódigo

...

Mientras $i < n1$ Y $j < n2$ Hacer

Si $L[i] \geq R[j]$ Entonces

arreglo[k] \leftarrow L[i]

$i \leftarrow i + 1$

Sino

arreglo[k] \leftarrow R[j]

$j \leftarrow j + 1$

Fin Si

$k \leftarrow k + 1$

Fin Mientras

Mientras $i < n1$ Hacer

arreglo[k] \leftarrow L[i]

$i \leftarrow i + 1$

$k \leftarrow k + 1$

Fin Mientras

Mientras $j < n2$ Hacer

arreglo[k] \leftarrow R[j]

$j \leftarrow j + 1$

$k \leftarrow k + 1$

Fin Mientras

FinFuncion

MIPS Assembler

copiarRestanteDer:

beq \$t1, \$s3, mergeEnd # j == n2

sll \$t3, \$t1, 2 # Al \$t3 le llevo el $i * 4$

add \$t3, \$s5, \$t3 # \$t3, la dirección del vector derecho

lw \$t3, 0(\$t3) # Traer el valor al banco de registros

sll \$t4, \$t2, 2 # aumentar a la siguiente dirección dado K

add \$t4, \$a0, \$t4 # Llevar el puntero a la siguiente posición

sw \$t3, 0(\$t4) # Guardar en la sublista

addi \$t1, \$t1, 1 # aumentar j

addi \$t2, \$t2, 1 # aumentar k

j copiarRestanteDer

mergeEnd:

Restaurar registros

lw \$ra, 24(\$sp)

lw \$s0, 20(\$sp)

lw \$s1, 16(\$sp)

lw \$s2, 12(\$sp)

lw \$s3, 8(\$sp)

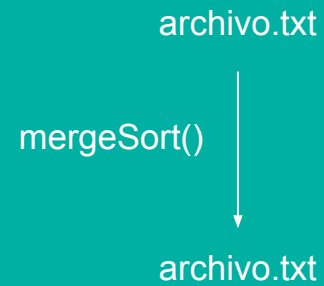
lw \$s4, 4(\$sp)

lw \$s5, 0(\$sp)

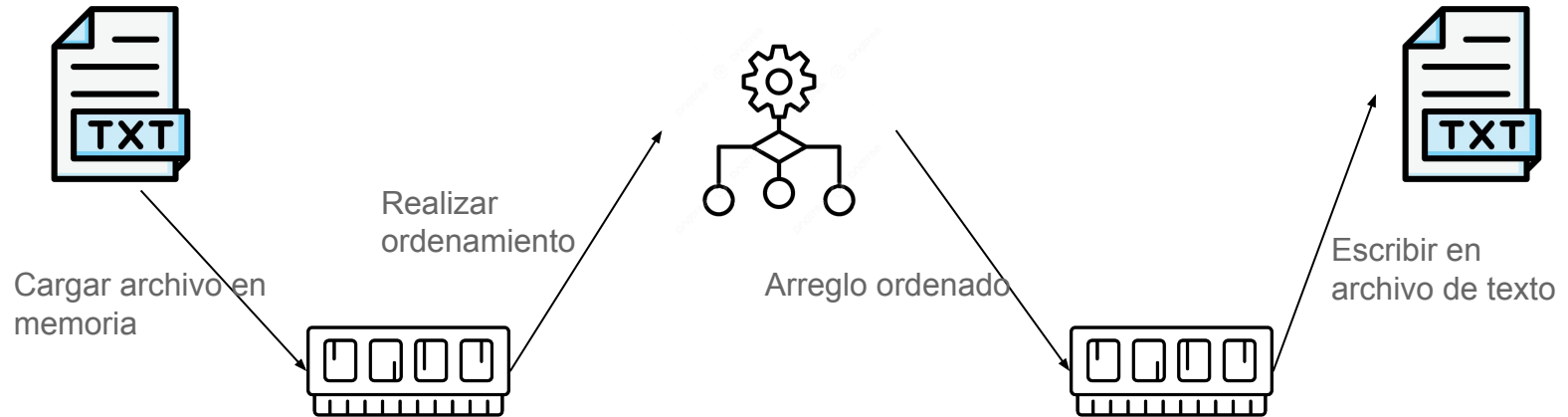
addi \$sp, \$sp, 28 # liberar los 7 registros necesarios

jr \$ra

Esquema general



Entradas y salidas



Funciones extras

Se hizo necesaria la creación de funciones para imprimir, el arreglo en la consola separado por espacios en una nueva línea

Luego imprimir un salto de línea., por si se hace necesaria nuevas impresiones en consola.

MIPS Assembler

printArray:

la \$t0, array	# Dirección base del array
lw \$t1, size	# Tamaño del array
li \$t2, 0	# Contador

printLoop:

```
beq $t2, $t1, printEnd
# Imprimir número
li $v0, 1
lw $a0, 0($t0)
syscall
# Imprimir espacio
li $v0, 4
la $a0, space
syscall
addi $t0, $t0, 4
addi $t2, $t2, 1
j printLoop
```

printEnd:

```
# Imprimir nueva línea
li $v0, 4
la $a0, newline
syscall
jr $ra
```

Funciones extras

Calcular longitud del arreglo, abrir, leer y cerrar el archivo, además de convertir los caracteres del archivo a memoria.

MIPS Assembler

#Calcular la longitud del arreglo

longitudArreglo:

```
addi $sp, $sp, -8 # Reservamos espacio en la pila
sw $ra, 0($sp) # Guardamos en la pila el retorno
sw $a0, 4($sp) # Guardamos en la pila nuestro arreglo
addi $t1, $zero, 0 # Definimos el contador
```

Abrir archivo de entrada

```
addi $v0, $zero, 13 # Syscall para abrir archivo
la $a0, filename # Dirección del nombre del archivo
addi $a1, $zero, 0 # Modo de lectura (read-only)
addi $a2, $zero, 0 # Permisos por defecto
syscall
add $t0, $zero, $v0 # Guardar descriptor del archivo en $t0
```

Leer el contenido del archivo

```
addi $v0, $zero, 14 # Syscall para leer archivo
add $a0, $zero, $t0 # Descriptor de archivo en $a0
la $a1, buffer # Dirección de almacenamiento del buffer
addi $a2, $zero, 100 # Tamaño máximo de lectura en bytes
syscall
add $t1, $zero, $v0 # Almacenar número de bytes en $t1
```

Cerrar el archivo

```
addi $v0, $zero, 16 # Syscall para cerrar archivo
add $a0, $zero, $t0 # Descriptor de archivo
syscall
```

Funciones extras

Para este caso, toca convertir el arreglo antes de almacenarlo, por lo que se tiene una condición para el arreglo y la lectura de archivos, y es que si se quieren leer todos los números, nuestro último carácter debe ser una coma (',').

La etiqueta **store** nos almacena el vector.

MIPS Assembler

parse_loop:

```
# Comprobar fin del buffer
beq $t2, $t3, end_parse # Si llegamos al final, salir del bucle
lb $t4, 0($t2) # Cargar el siguiente byte en $t4
beq $t4, 44, store_number # Si es una coma (','), guardar número
beq $t4, 10, end_parse # Si es salto de línea, terminar
beq $t4, 13, end_parse # Si es retorno de carro, terminar
sub $t4, $t4, 48 # Convertir de ASCII a número (0-9)
sll $t6, $t5, 3 # $t6 = $t5 * 8 (desplaza 3 bits a la izquierda)
sll $t7, $t5, 1 # $t7 = $t5 * 2 (desplaza 1 bit a la izquierda)
add $t5, $t6, $t7 # $t5 = ($t5 * 8) + ($t5 * 2) = $t5 * 10
add $t6, $zero, $zero # Resetear $t6 a 0
add $t7, $zero, $zero # Resetear $t7 a 0
add $t5, $t5, $t4 # Añadir el dígito actual
addi $t2, $t2, 1 # Avanzar al siguiente byte
j parse_loop
```

store_number:

```
# Almacenar el número en la pila y preparar para el siguiente
sw $t5, 0($sp) # Guardar número en la pila
addi $sp, $sp, -4 # Reservar espacio para el siguiente número
addi $t5, $zero, 0 # Reiniciar $t5 para el próximo número
addi $t2, $t2, 1 # Avanzar al siguiente byte
j parse_loop
```

end_parse:

```
# Ajustar la pila para iniciar la impresión
addi $sp, $sp, 4
```

Funciones extras

Una vez almacenado debemos empezar a procesar el mapeo de la memoria desde la pila, y además de eso hacer el llamado a la función de ordenamiento del programa **mergeSort**

MIPS Assembler

llenarVector:

```
# Llenar el array desde la pila
lw $t6, 0($sp) # Cargar número desde la pila
beq $t6, 0, ordenar_e_imprimir # Salir si hay un cero (fin de pila)
bne $t7, $zero, indiceInicializado
add $t7, $zero, 0 # Definir índice $t7 = 0
```

indiceInicializado:

```
sw $t6, arreglo($t7) # Guardar el número en vector[i]
# Imprimir el número actual
addi $v0, $zero, 1 # Syscall para imprimir entero
add $a0, $zero, $t6 # Número a imprimir
syscall
# Avanzar en la pila y aumentar el índice del vector
addi $sp, $sp, 4 # Moverse al siguiente número en la pila
addi $t7, $t7, 4 # Incrementar índice
j llenarVector
```

ordenar_e_imprimir:

```
la $a0, arreglo
jal longitudArreglo # a0 = dirección de vector[]
addi $a1, $v0, 0 # a1 = n (longitud del array)
addi $s0, $v0, 0 # a1 = n (longitud del array)
jal mergeSort
jal guardar_archivo
```


Funciones extras

Para este caso, toca convertir el arreglo antes de almacenarlo, por lo que se tiene una condición para el arreglo y la lectura de archivos, y es que si se quieren leer todos los números, nuestro último carácter debe ser una coma (',').

La etiqueta **store** nos almacena el vector.

MIPS Assembler

guardar_archivo:

```
# Abrir o crear el archivo para escritura
li $v0, 13      # syscall para abrir/crear archivo
la $a0, nuevoArchivo # nombre del archivo
li $a1, 1       # modo de escritura
li $a2, 0       # permisos por defecto
syscall
add $s6, $zero, $v0 # guardar el descriptor del archivo
# Verificar si hubo error al abrir el archivo
bltz $s6, salir    # si es negativo, hubo error
# Inicializar variables
la $t1, arreglo    # puntero al vector
li $t2, 0          # índice
la $s1, nuevo_espacio # buffer para conversión
```

Funciones extras

Manejar arreglo y escritura del archivo, y conversiones y manejo de los datos ASCII

MIPS Assembler

escribir_bucle:

```
lw $t3, ($t1)      # cargar número actual
beq $t3, $zero, cerrar_archivo # si es 0, terminamos
# Convertir número a string
add $t4, $zero, $t3 # copiar número para conversión
li $t5, 0           # contador de dígitos
li $t6, 10          # divisor
la $s1, nuevo_espacio # reiniciar puntero del buffer
# Si el número es negativo, manejarlo
bgez $t4, bucle_conversion
li $t7, 45          # ASCII del signo menos
sb $t7, ($s1)       # guardar el signo
addiu $s1, $s1, 1   # avanzar puntero
sub $t4, $zero, $t4 # hacer positivo el número
```

bucle_conversion:

```
divu $t4, $t6       # dividir por 10
mfhi $t7            # obtener residuo (último dígito)
mflo $t4            # obtener cociente
addiu $t7, $t7, 48  # convertir a ASCII
sb $t7, ($s1)       # guardar dígito
addiu $s1, $s1, 1   # avanzar puntero
addiu $t5, $t5, 1   # incrementar contador
bne $t4, $zero, bucle_conversion # si quedan dígitos, continuar
# Invertir la cadena de caracteres
la $s1, nuevo_espacio # reiniciar puntero
add $t7, $s1, $zero # guardar inicio
add $t8, $s1, $t5   # apuntar al final
addi $t8, $t8, -1   # ajustar al último carácter
```



UNIVERSIDAD
DE ANTIOQUIA

Vicerrectoría de Docencia

Funciones extras

Intercambio de números, retroceso para trabajar con la memoria y escribir de manera descendente. Acá hacemos uso del buffer, y se verifica la condición definida, de separar los números por comas

MIPS Assembler

invertir_bucle:

```
slt $at, $t7, $t8      # $at = 1 si $t7 < $t8, sino $at = 0
beq $at, $zero, escribir_numero # salta si $at es 0 (si $t7 >= $t8)
lb $t4, ($t7)          # cargar carácter del inicio
lb $t6, ($t8)          # cargar carácter del final
sb $t6, ($t7)          # intercambiar caracteres
sb $t4, ($t8)
addiu $t7, $t7, 1      # avanzar puntero inicio
addiu $t8, $t8, -1     # retroceder puntero final
j invertir_bucle
```

escribir_numero:

```
# Escribir el número en el archivo
li $v0, 15             # syscall para escribir
add $a0, $zero, $s6    # descriptor del archivo
la $a1, nuevo_espacio  # buffer con el número
add $a2, $zero, $t5    # longitud del número
syscall

# Avanzar al siguiente número
addiu $t1, $t1, 4      # siguiente elemento del vector
addiu $t2, $t2, 1      # incrementar índice
# Verificar si hay más números para escribir la coma
lw $t3, ($t1)          # cargar el siguiente número
bne $t3, $zero, escribir_coma # si no es cero, escribir la coma
j escribir_bucle       # si es cero, terminar el bucle
```

Funciones extras

Por último tenemos la escritura en el archivo de texto

MIPS Assembler

invertir_bucle:

```
slt $at, $t7, $t8      # $at = 1 si $t7 < $t8, sino $at = 0
beq $at, $zero, escribir_numero # salta si $at es 0 (si $t7 >= $t8)
lb $t4, ($t7)          # cargar carácter del inicio
lb $t6, ($t8)          # cargar carácter del final
sb $t6, ($t7)          # intercambiar caracteres
sb $t4, ($t8)
addiu $t7, $t7, 1      # avanzar puntero inicio
addiu $t8, $t8, -1     # retroceder puntero final
j invertir_bucle
```

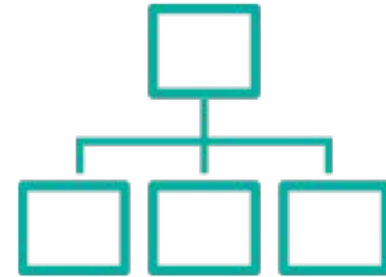
escribir_numero:

```
# Escribir el número en el archivo
li $v0, 15             # syscall para escribir
add $a0, $zero, $s6    # descriptor del archivo
la $a1, nuevo_espacio  # buffer con el número
add $a2, $zero, $t5    # longitud del número
syscall
# Avanzar al siguiente número
addiu $t1, $t1, 4      # siguiente elemento del vector
addiu $t2, $t2, 1      # incrementar índice
# Verificar si hay más números para escribir la coma
lw $t3, ($t1)          # cargar el siguiente número
bne $t3, $zero, escribir_coma # si no es cero, escribir la coma
j escribir_bucle       # si es cero, terminar el bucle
```

Conclusiones

La implementación del algoritmo fue una práctica bastante desafiante, en la que se reconoce la importancia de la localidad y el manejo de la memoria, se logra obtener una habilidad más marcada para definir instrucciones básicas de flujo en programación de bajo nivel. Por otro lado el desarrollo del pensamiento algorítmico, tuvo un papel fundamental dada la importancia que tenía entender el algoritmo en alto nivel, para luego plantear una abstracción más detallada.

Por último destacar el amplio proceso investigativo que se tuvo para definir la gestión de archivos, en especial, la conversión y manejo de datos **ASCII** para el posterior procesamiento de esos datos en un arreglo



Anexos

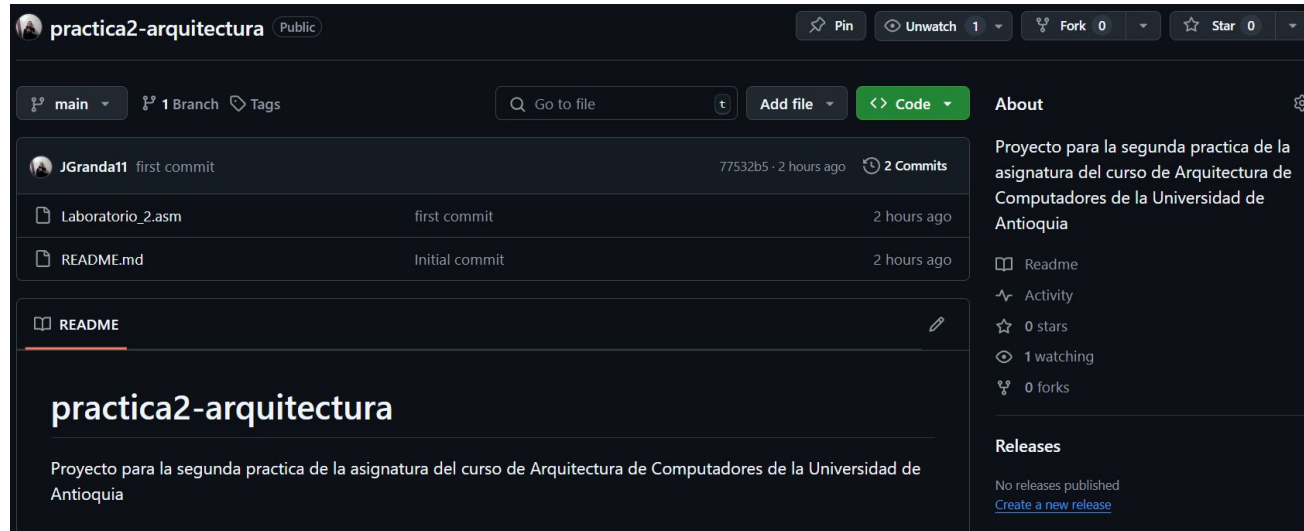


Video YouTube



[Enlace al video](#)

Repositorio GitHub



The screenshot shows a GitHub repository page for 'practica2-arquitectura'. The repository is public and has 1 branch (main), 0 forks, and 0 stars. The repository description is 'Proyecto para la segunda practica de la asignatura del curso de Arquitectura de Computadores de la Universidad de Antioquia'. The repository contains two files: 'Laboratorio_2.asm' and 'README.md'. The 'README' file is selected and shows the repository name 'practica2-arquitectura' and the description 'Proyecto para la segunda practica de la asignatura del curso de Arquitectura de Computadores de la Universidad de Antioquia'. The repository was created by JGranda11 and has 2 commits.

practica2-arquitectura Public

main 1 Branch Tags

Go to file Add file Code

JGranda11 first commit 77532b5 · 2 hours ago 2 Commits

File	Commit	Time
Laboratorio_2.asm	first commit	2 hours ago
README.md	Initial commit	2 hours ago

README

practica2-arquitectura

Proyecto para la segunda practica de la asignatura del curso de Arquitectura de Computadores de la Universidad de Antioquia

About

Proyecto para la segunda practica de la asignatura del curso de Arquitectura de Computadores de la Universidad de Antioquia

- Readme
- Activity
- 0 stars
- 1 watching
- 0 forks

Releases

No releases published

[Create a new release](#)

[Enlace repositorio](#)