**UNIVERSIDAD DE SANTIAGO DE CHILE**

**FACULTAD DE INGENIERÍA**

**Departamento de Ingeniería Informática**


# LABORATORIO 1: INSTRUCCIONES MIPS Y PROGRAMACIÓN EN LENGUAJE ENSAMBLADOR

**Organización de Computadores**

**Isaac Alejandro Espinoza Barría 21.278.828-7**

**Sección:**

13275-0-C-3 Diurno

**Profesor:**

Viktor Tapia

**Ayudante:**

Nicolás Henríquez

**Fecha:**

18 de octubre de 2024

# TABLA DE CONTENIDO

TABLA DE CONTENIDO ..... 2
1. INTRODUCCIÓN ..... 3
2. MARCO TEÓRICO ..... 4
2.1. ARQUITECTURA MIPS ..... 4
2.2. MIPS Y LOS REGISTROS ..... 5
2.3. INSTRUCCIONES MIPS ..... 5
2.4. INSTRUCCIONES PARA SUMA ..... 6
2.5. INSTRUCCIONES DE TRANSFERENCIA DE DATOS ..... 7
2.6. INSTRUCCIONES CONDICIONALES DE FLUJO ..... 7
2.7. MODOS DE DIRECCIONAMIENTO ..... 8
3. DESARROLLO Y RESULTADOS ..... 9
3.1. PREDECIR EL FUNCIONAMIENTO DE PROGRAMAS MIPS ..... 9
3.2. ESCRIBIR PROGRAMAS MIPS ..... 13
4. CONCLUSIONES ..... 15
REFERENCIAS ..... 16

# 1. INTRODUCCIÓN

Sin duda, la creación del computador ha sido un antes y un después en la historia de la humanidad. En la actualidad, no podemos imaginar la vida sin computadores, tanto así que casi todas las personas tienen disponibilidad de un computador móvil como un celular, pero muchas veces no comprendemos su funcionamiento, solo aceptamos que lo hacen.

Los computadores ejecutan programas, los que son un set de instrucciones en lenguaje máquina que se alojan en la memoria, sin embargo, estas instrucciones son difíciles de entender ya que están compuestas por números binarios. Ante esto, para facilitar la programación a nivel de hardware, se utiliza el lenguaje ensamblador de bajo nivel, el que permite a los programadores interpretar de manera más fácil las instrucciones en lenguaje máquina.

El presente informe aborda un primer acercamiento a la programación en lenguaje ensamblador, enfocado en la arquitectura MIPS en su versión de 32 bits. Esta arquitectura es mayormente utilizada para el ámbito educativo, en la enseñanza del funcionamiento de los procesadores de los computadores, siendo famosa por su simpleza y facilidad de uso.

El problema existente es la dificultad de traducir programas en lenguajes de programación de alto nivel a lenguaje ensamblador, posteriormente a instrucciones en lenguaje máquina que el procesador pueda entender. La solución es desarrollar pequeños programas en lenguaje ensamblador MIPS para entender el funcionamiento de instrucciones. De esta manera, se comprende la relación entre un programa en alto nivel y su equivalente en lenguaje máquina.

Los objetivos de la experiencia son: predecir el funcionamiento de un programa MIPS; traducir programas escritos en un lenguaje de alto nivel a MIPS; escribir programas MIPS que usan instrucciones aritméticas, de salto y memoria; usar MARS (un IDE para MIPS) para escribir, ensamblar y depurar programas MIPS; implementar algoritmos en MIPS para resolver problemas matemáticos de baja complejidad.

Con esto, no solo se busca entender el lenguaje ensamblador, sino también considerar la importancia de la programación en bajo nivel para el desarrollo de software.

# 2. MARCO TEÓRICO

Un computador se comprende como una “máquina capaz de aceptar unos datos de entrada, efectuar con ellos operaciones lógicas y aritméticas, y proporcionar los datos resultantes a través de medios de salida; todo ello … bajo el control de un programa de instrucciones” (Prieto, Lloris, & Torres, 1989).

Estas órdenes son conocidas como lenguaje máquina, el cual, según Castillo y Claver (2001):

> Está constituido por instrucciones máquina que indican al computador lo que tiene que hacer, es decir, son las órdenes que el computador es capaz de comprender. Cada instrucción máquina está constituida por un conjunto ordenado de unos y ceros, organizados en diferentes campos. Cada uno de estos campos contiene información que se complementa para indicar al procesador la acción a realizar… está constituido por instrucciones máquina que indican al computador lo que tiene que hacer, es decir, son las órdenes que el computador es capaz de comprender. Cada instrucción máquina está constituida por un conjunto ordenado de unos y ceros, organizados en diferentes campos. Cada uno de estos campos contiene información que se complementa para indicar al procesador la acción a realizar.

Cabe destacar que el lenguaje ensamblador es la representación simbólica de instrucciones en lenguaje máquina. Donde este

> ofrece una representación más próxima al programador y simplifica la lectura y escritura de los programas. Las principales herramientas que proporciona la programación en ensamblador son: la utilización de nombres simbólicos para operaciones y posiciones de memoria, y unos determinados recursos de programación que permiten aumentar la claridad (legibilidad) de los programas. (Castillo Catalán & Claver Iborra, 2001)

## 2.1. ARQUITECTURA MIPS

Dentro de un computador, el procesador se entiende como el “cerebro” que ejecuta las instrucciones según el tipo de arquitectura implementada. “Una arquitectura son los atributos del sistema computacional que son visibles a un programador, ya sean, set de instrucciones, número de bits usados para representar un dato, mecanismos de entrada y salida, etc.” (Barriga, 2005).

MIPS es una arquitectura de procesadores que implementa la filosofía RISC (Reduced Instruction Set Computer), por lo que busca ser simple y regular, y en consecuencia, fácil de aprender y entender (Castillo Catalán & Claver Iborra, 2001). Por esta razón, MIPS implementa 4 principios clave: lo

simple favorece la regularidad, hacer que el caso común sea rápido, pequeño es rápido, y un buen diseño demanda buenos compromisos (Money Harris & Harris, 2007).

## 2.2. MIPS Y LOS REGISTROS

“La unidad entera del MIPS (la CPU) cuenta con 32 registros de propósito general que almacenan valores enteros de 32 bits. Los nombres de los registros van desde $0 hasta $31, y tienen nombres alternativos” (Ortega, 2002). En la siguiente imagen se observan los nombres, números y usos para todos los registros:

| Name | Number | Use |
|---|---|---|
| $0 | 0 | the constant value 0 |
| $at | 1 | assembler temporary |
| $v0-$v1 | 2–3 | procedure return values |
| $a0-$a3 | 4–7 | procedure arguments |
| $t0-$t7 | 8–15 | temporary variables |
| $s0-$s7 | 16–23 | saved variables |
| $t8-$t9 | 24–25 | temporary variables |
| $k0-$k1 | 26–27 | operating system (OS) temporaries |
| $gp | 28 | global pointer |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | procedure return address |

Imagen 1: Registros de MIPS (Money Harris & Harris, 2007).

## 2.3. INSTRUCCIONES MIPS

MIPS utiliza instrucciones de 32 bits, las que pueden ser de 3 tipos y, según esto, cambia el modo de codificación a lenguaje máquina. “Las instrucciones de tipo R operan en tres registros. Las instrucciones de tipo I operan en dos registros y un registro inmediato de 16 bits. Las instrucciones de tipo J (jump/salto) operan en un registro inmediato de 26 bits” (Money Harris & Harris, 2007). Cada tipo de instrucciones se codifica de distinta forma, donde cada instrucción y registro en particular tiene un código, los que combinados en un orden especifico, generan el código binario de la instrucción. Por ejemplo, las instrucciones tipo I siguen el siguiente formato:

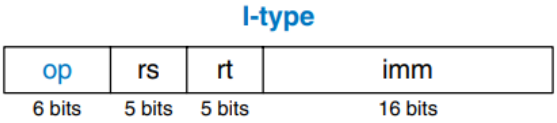
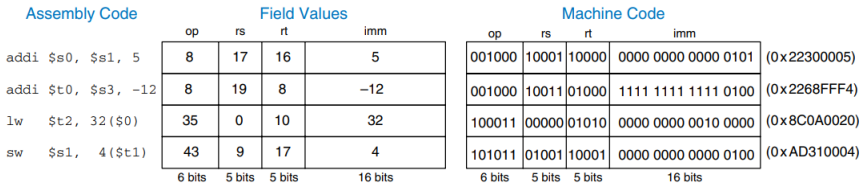



Imagen 2: Formato de instrucciones I en MIPS (Money Harris & Harris, 2007).

Donde “op” es el código de la instrucción específica, “rs” es el número del registro fuente, “rt” es el número del registro destino, y “imm” es el valor de la constante. Todos los números en código binario.

A continuación, se observan ejemplos de transformaciones de lenguaje ensamblador a lenguaje máquina:



| Assembly Code | Field Values | | | | Machine Code | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | op | rs | rt | imm | op | rs | rt | imm | |
| addi $s0, $s1, 5 | 8 | 17 | 16 | 5 | 001000 | 10001 | 10000 | 0000 0000 0000 0101 | (0x22300005) |
| addi $t0, $s3, -12 | 8 | 19 | 8 | −12 | 001000 | 10011 | 01000 | 1111 1111 1111 0100 | (0x2268FFF4) |
| lw $t2, 32($0) | 35 | 0 | 10 | 32 | 100011 | 00000 | 01010 | 0000 0000 0010 0000 | (0x8C0A0020) |
| sw $s1, 4($t1) | 43 | 9 | 17 | 4 | 101011 | 01001 | 10001 | 0000 0000 0000 0100 | (0xAD310004) |
| | 6 bits | 5 bits | 5 bits | 16 bits | 6 bits | 5 bits | 5 bits | 16 bits | |

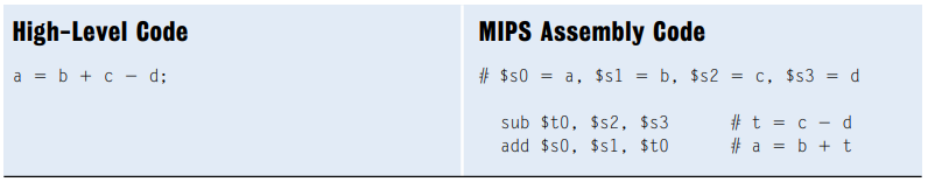
Imagen 3: Transformaciones de instrucciones I a lenguaje máquina (Money Harris & Harris, 2007).

## 2.4. INSTRUCCIONES PARA SUMA

Para realizar una suma común, se necesitan tres operandos: dos números a sumar y un lugar donde recibir la suma. De este modo, con el propósito de ser simple, la mayoría de las instrucciones aritméticas y lógicas que implementa MIPS son de tres operandos.

A diferencia de los programas en lenguajes de alto nivel, en las instrucciones aritméticas de MIPS los operandos no pueden ser variables. Solo pueden ser registros, o cómo máximo, dos registros y una constante. Es tarea de los compiladores el asociar eficientemente las numerosas variables de los programas con los pocos registros que ofrece cada arquitectura (Ortega, 2002).

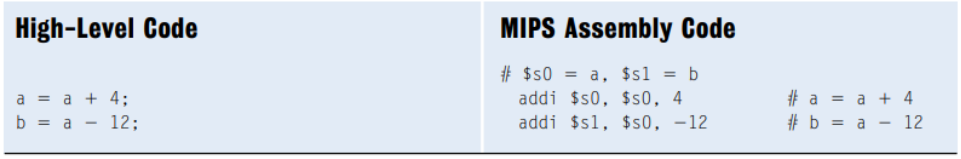
En la siguiente imagen se muestran las instrucciones de MIPS para sumar y restar:



| High-Level Code | MIPS Assembly Code |
|---|---|
| `a = b + c - d;` | `# $s0 = a, $s1 = b, $s2 = c, $s3 = d`<br>`sub $t0, $s2, $s3 # t = c - d`<br>`add $s0, $s1, $t0 # a = b + t` |

Imagen 4: Ejemplo de instrucciones de suma (Money Harris & Harris, 2007).

De modo similar, podemos trabajar con constantes, también conocidas como inmediatos. Esto considerando instrucciones tipo I, donde el último operando es la constante.



| High-Level Code | MIPS Assembly Code |
|---|---|
| `a = a + 4;`<br>`b = a - 12;` | `# $s0 = a, $s1 = b`<br>`addi $s0, $s0, 4 # a = a + 4`<br>`addi $s1, $s0, -12 # b = a - 12` |

Imagen 5: Ejemplo de instrucciones de suma con constante (Money Harris & Harris, 2007).

## 2.5. INSTRUCCIONES DE TRANSFERENCIA DE DATOS

Según Ortega (2002):

> El lenguaje ensamblador incluye instrucciones que transfieren datos entre la memoria y los registros. Para referirse a una palabra en memoria, una instrucción debe proveer su dirección. La memoria puede considerarse como un enorme arreglo unidimensional y las direcciones (empezando en cero) son simplemente los índices a este arreglo. Como no sólo las palabras, sino también los bytes son útiles en muchos programas, la arquitectura MIPS direcciona a los bytes individuales; cuando se dice "Memory[i]", se está haciendo referencia al i-ésimo byte de la memoria, y no a la i-ésima palabra de la memoria.

La instrucción que mueve datos de la memoria a los registros se llama “load word” (lw), y la que transfiere datos de los registros a la memoria, se llama “store word” (sw) (Ortega, 2002).

A continuación, el formato de ambas instrucciones:

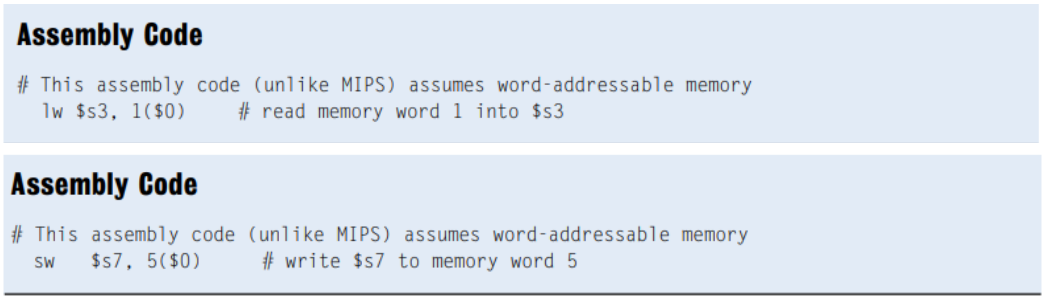



Imagen 6: Ejemplo de instrucciones lw y sw en MIPS (Money Harris & Harris, 2007).

## 2.6. INSTRUCCIONES CONDICIONALES DE FLUJO

Lo que distingue a las computadoras de las calculadoras simples es la habilidad para tomar decisiones. Basándose en los datos de entrada y en los valores que se van calculando, las computadoras pueden ejecutar distintos grupos de instrucciones.

En los lenguajes de programación de más alto nivel la toma de decisiones se representa comúnmente a través de los enunciados (statements) if combinados, que generan condicionales (Ortega, 2002). En este caso, MIPS posee dos tipos de instrucciones condicionales de tipo I, beq (Branch if equal) y, bne (Branch if not equal). Ambas comparan los valores de registros, y dependiendo si se cumple la condición, la ejecución se dirige una instrucción que tenga la etiqueta especificada, en el caso contrario, continua la ejecución secuencial de instrucciones (Money Harris

& Harris, 2007). Una etiqueta, también conocida como label, “indica la locación de una instrucción en el programa” (Money Harris & Harris, 2007).

A continuación, un programa en MIPS que utiliza la instrucción condicional beq:

**MIPS Assembly Code**

```
  addi  $s0, $0, 4         # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1         # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2        # $s1 = 1 << 2 = 4
  beq   $s0, $s1, target   # $s0 == $s1, so branch is taken
  addi  $s1, $s1, 1        # not executed
  sub   $s1, $s1, $s0      # not executed

target:
  add   $s1, $s1, $s0      # $s1 = 4 + 4 = 8
```

Imagen 7: Ejemplo de programa condicional en MIPS (Money Harris & Harris, 2007).

## 2.7. MODOS DE DIRECCIONAMIENTO

Según Ortega (2002), la máquina MIPS sólo provee el tipo de direccionamiento de memoria c(rx) que vimos en las instrucciones “lw” y “sw”, y que usa la suma del valor inmediato c y el almacenado en el registro rx como la dirección. MIPS nos provee los modos de direccionamiento para instrucciones de carga y almacenamiento que se presenta a continuación:

| Formato | Cálculo de la dirección |
|---|---|
| (registro) | contenido del registro |
| inm | inmendiato |
| inm(registro) | inmediato + contenido del registro |
| símbolo | dirección del símbolo (definido por una etiqueta) |
| símbolo ± inm | dirección del símbolo ± inmendiato |
| símbolo ± inm(registro) | dirección del símbolo ± (inmendiato + contenido del registro) |

Imagen 8: Modos de direccionamiento en MIPS (Ortega, 2002).

# 3. DESARROLLO Y RESULTADOS

## 3.1. PREDECIR EL FUNCIONAMIENTO DE PROGRAMAS MIPS

Dentro de este apartado, ambos códigos fueron abordados de modo similar. En primer lugar, se analiza cada instrucción secuencial y se determina lo que hace. Una vez ya clarificado el funcionamiento de cada una, se procede a realizar la traza de las instrucciones considerando valores específicos para los registros usados.

Cabe destacar que, para el segundo código, fue necesario un mayor esfuerzo en el análisis de instrucciones. Esto debido a que, además de incluir operaciones aritméticas, también se realiza la lectura y escritura de datos en la memoria. Lo que obligó la comprensión del cálculo de direcciones de memoria y el acceso a los datos guardados en ellas.

1 ¿Cuál es el valor en los registros `$t1` y `$t0` al terminar el programa cuando $t2 almacena un 2? ¿Y cuándo $t2 almacena un 0?

`addi $t0, $zero, 2` → $t0 almacena la suma del valor de $zero = 0 con la cte 2. ∴ $t0 = 0 + 2 = 2

`add $t1, $t0, $t2` → $t1 contiene la suma de los valores de $t0 y $t2 ∴ $t1 = $t0 + $t2 = 2 + $t2

`beq $t0, $t1, A` → beq compara los valores de $t0 y $t1, si son iguales, salta a la instrucción del label A; sino, continua con la siguiente instrucción.

`addi $t1, $zero, 1` → ∴ $t1 = $zero + 1 = 0 + 1 = 1

Luego, tenemos 2 casos posibles:

- Si $t2 = 2.
  - ⇒ `addi $t0, $zero, 2` → $t0 = 0 + 2 = 2
    `add $t1, $t0, $t2` → $t1 = 2 + 2 = 4
    `beq $t0, $t1, A` → 2 ≠ 4 ⇒ $t0 ≠ $t1. No va a A
    `addi $t1, $zero, 1` → $t1 = 0 + 1 = 1

  ∴ $t0 = 2 ∧ $t1 = 1

- Si $t2 = 0
  - ⇒ `addi $t0, $zero, 2` → $t0 = 0 + 2 = 2
    `add $t1, $t0, $t2` → $t1 = 2 + 0 = 2
    `beq $t0, $t1, A` → 2 = 2 ⇒ $t0 = $t1. Va a A

  ∴ $t0 = 2 ∧ $t1 = 2

1.1 Considere que $t2 almacena un 0 y la instrucción A contiene lo siguiente:

Indique los nuevos valores de los registros $t0, $t1 y $t2

bgez $t2, B → Analiza si el valor de $t2 es mayor o igual a cero, si esto es cierto, va al label B; sino, continua normalmente.

# Notar que, en este caso, aun que no se cumpla la condición, de igual manera se ejecuta la siguiente instrucción porque es la misma de etiqueta B.

B:

addi $t2, $t1, -5 → $t2 = $t1 + (-5)

add $t2, $t1, $t2 → $t2 = $t1 + $t2

Luego, tenemos 2 casos:

- Caso 1, donde $t0 = 2, $t1 = 1, $t2 = 2

  ⇒ bgez $t2, B → $t2 = 2 ⇒ 2 ≥ 0. Voy a B.

  B: addi $t2, $t1, -5 → $t2 = $t1 + (-5) = 1 - 5 = -4

  add $t2, $t1, $t2 → $t2 = $t1 + $t2 = 1 - 4 = -3

  ∴ $t0 = 2 ∧ $t1 = 1 ∧ $t2 = -3

- Caso 1.1, donde $t0 = 2, $t1 = 2, $t2 = 0

  ⇒ bgez $t2, B → $t2 = 0 ⇒ 0 ≥ 0. Voy a B.

  B: addi $t2, $t1, -5 → $t2 = $t1 + (-5) = 2 - 5 = -3

  add $t2, $t1, $t2 → $t2 = $t1 + $t2 = 2 - 3 = -1

  ∴ $t0 = 2 ∧ $t1 = 2 ∧ $t2 = -1

2 ¿Cuál es el valor en los registros `$t2`, `$t1` y `$t0` y las direcciones de memoria `0x10010000` y `0x10010004` al terminar el programa?

```
addiu $t0, $zero, 0x10010000
# La instrucción suma el valor de un registro con un inmediato, aceptando
# hexadecimal, y lo almacena en el registro base.
#  ∴ $t0 = $zero + 0x10010000 = 0x10010000
addi $t1, $zero, 5
#  ∴ $t1 = $zero + 5 = 0+5 = 5 = 0x00000005
sw $t1, 0($t0)
# Escribo el valor de $t1 en la dirección de memoria ($t0 + 0),
# el valor de $t0 es el registro base y el 0 es el offset.
#  ∴ Dirección = $t0 + 0 = 0x10010000 + 0x00000000 = 0x10010000.
#  ∴ En la dirección 0x10010000 escribo $t1 = 0x00000005
lw $t2, 0($t0)
# Guardo en $t2 el valor dentro de la dirección ($t0 + 0)
#  ∴ Dirección = $t0 + 0 = 0x10010000, contiene → 0x00000005
#  ∴ $t2 = 0x00000005
addiu $t0, $t0, 4
#  ∴ $t0 = $t0 + 4 = 0x10010000 + 0x00000004 = 0x10010004
sw $t2, 0($t0)
#  ∴ Dirección = $t0 + 0 = 0x10010004 + 0x00000000 = 0x10010004
#  ∴ En la dirección 0x10010004 escribo $t2 = 0x00000005.
```

## 3.2. ESCRIBIR PROGRAMAS MIPS

Para el desarrollo del primer código, lo más difícil era cómo transformar un ciclo while de un lenguaje de alto nivel a un código de bajo nivel, considerando que no tenemos estructuras de iteración. Sin embargo, para lograrlo fue necesario utilizar una instrucción condicional de igualdad llamada “beq”, que actúa como condición de salida del búcle cuando se cumple. Además, se usó la instrucción jump para saltar al inicio del búcle una vez terminadas las instrucciones, y así volver a consultar la condición de salida con los nuevos valores.

Notamos que el código representa la suma de los primeros 9 números naturales.

```
.data

.text

        #considerando  a = $s0, z = $s1

        #inicializacines
        add $s0, $zero, $zero    #Inicializacion a=0
        addi $s1, $zero, 1       #Inicializacion z=1
        addi $t0, $zero, 10      #Incializacion var temp con 10 para las iteraciones

while:
        beq $s1, $t0, salida     #Consideramos el caso contrario de z <> 10,  luego,
                                 #  solo si z==10, es decir, $s1==$t0, voy al label salida,
                                 #  en caso contrario sigo con las instrucciones siguientes
        add $s0, $s0, $s1        #Al registro de a se le suma el valor de z. $s0 = $s0+$s1
        addi $s1, $s1, 1         #Se suma 1 al registro del valor de z. $s1 = $s1+1
        j while                  #Salto al inicio de la iteracion, al label while,
                                 #  porque no se ha cumplido la condicion de salida


salida:
        li $v0, 10               #Llamada al sistema para indicar el termino de ejecucion
        syscall                  #  del programa
```

La suma de los primeros 9 números enteros es 45, valor que una vez finalizado el programa se almacena en la variable “a”. Lo que se verifica al observar el valor del registro $s0 es el 45 en hexadecimal, es decir, 0x0000002d.

| | | |
|---|---|---|
| $t7 | 15 | 0x00000000 |
| $s0 | 16 | 0x0000002d |
| $s1 | 17 | 0x0000000a |

En cuanto al segundo código, el ciclo while funciona de manera similar al código anterior, pero utiliza una desigualdad para determinar la condición de salida. Ante esto, se emplea la instrucción “slti”, la cual considera el valor de un registro, lo resta con una constante, y guarda el valor del signo en un registro destino; valor que permite analizar la condición de salida.

Para escribir valores en el arreglo, es necesario calcular la dirección de memoria correspondiente en cada iteración, y así acceder a cada elemento del arreglo. Esto implica calcular el valor de la variable “a” multiplicado por 4 y sumarlo a la dirección base; ya que multiplicar por 4 permite acceder a palabras completas de la memoria, en vez de a bytes individuales.

Finalmente, se utiliza “sw” para almacenar el valor calculado de “a+b” en cada elemento de arreglo, se actualiza el valor de “a” y se salta (jump) al inicio de la iteración.

En el código, suponemos valores iniciales para “a”, “b” y la dirección base del arreglo; donde a=0, b=10, y la dirección base es 0x10010000. Esto para observar el funcionamiento del programa.

```
.data

.text

        #Suponiendo  a = $s0, b = $s1, y que la base del arrehlo esta en el registro $s2
        #En este caso damos valores cualquiera a los registro y a la direccion base, para ver el funcionamiento del codigo

        addi $s0, $zero, 0      #Inicializamos el registro $s0 (a) como 0
        addi $s1, $zero, 10     #Inicializamos el registro $s1 (b) como 10
        addiu $s2, $zero, 0x10010000    #Inicializamos el registro $s2 con una direccion de memoria existente

while:
        slti $t1, $s0, 10       #Guarda en $t1 el signo de la resta entre "a" y la cte 10
                                # Luego, si a<10 -> $t1=1  . Si a>=10 -> $t1=0
        beq $t1, $zero, salida  #En el caso en que el signo sea posito (0), significa que como es igual a cero ($zero), voy al label salida
                                #   En caso contrario sigo con la siguiente instruccion
        sll $t1, $s0, 2         #En $t1, almacenamos el valor de "a" multiplicado por 4, para acceder al elemento siguiente de array
                                #   Esto se hace, realizando un movimiento de dos bits hacia la izquierda y agregando ceros
        add $t1, $t1, $s2       #Se suma en $t1, la direccion base del array ($s2) mas "a" por 4 (que ya existia en $t1).
                                #   Asi obtenemos la direccion de memoria del elemento actual de array
        add $t2, $s1, $s0       #En $t2 guado la suma entre "a" y "b", para usarla en la proxima instruccion
                                #   $t2 = b+a
        sw $t2, 0($t1)          #Escribimos en la direccion obtenida de (0 + $t1), el valor de $t2, que es b+a
                                #   D[a] = b+a
        addi $s0, $s0, 1        # Aumentamos en 1 el valor de a para evitar ciclo infinito
                                #   a += 1
        j while                 #Saltamos al inicio de la iteracion


salida:
        li $v0, 10              #Llamada al sistema para indicar el termino de ejecucion del programa
        syscall
```

Como resultado de la ejecución del ciclo while con los valores iniciales propuestos, el arreglo debería contener los números ordenados desde el 10 al 19. Lo cual se confirma al revisar los valores de la memoria desde la dirección base del arreglo, donde se obtuvieron los números en hexadecimal desde el 10 al 19.

Data Segment

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) | Value (+1c) |
|---|---|---|---|---|---|---|---|---|
| 0x10010000 | 0x0000000a | 0x0000000b | 0x0000000c | 0x0000000d | 0x0000000e | 0x0000000f | 0x00000010 | 0x00000011 |
| 0x10010020 | 0x00000012 | 0x00000013 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |

# 4. CONCLUSIONES

En esta primera experiencia de laboratorio, se logró realizar con éxito el 100% de los ejercicios propuestos, con lo que estoy bastante satisfecho considerando que es mi primer acercamiento a un lenguaje ensamblador. Se cumplieron todos los objetivos, se logró predecir el funcionamiento de programas MIPS a través de una traza realizada manualmente; traducir programas escritos en lenguaje de alto nivel a MIPS, donde transformamos ciclos while y operaciones de arreglos a instrucciones en MIPS; escribir programas MIPS que usan instrucciones aritméticas, de salto y memoria; usar el software MARS para escribir, ensamblar y depurar programas MIPS; e implementar algoritmos en MIPS para resolver problemas matemáticos de baja complejidad, como lo fue el cálculo de la suma de los primeros 9 números naturales.

En general, disfruté mucho realizar el laboratorio, ya que me brindó la oportunidad de aplicar prácticamente el lenguaje ensamblador y la arquitectura MIPS. Aunque la teoría la había obtenido en clases, no había tenido la oportunidad de implementarla en el software. Creo que realizar la experiencia me ayudó a prepararme mejor para enfrentar los próximos laboratorios del curso, que implicarán un mayor nivel de dificultad.

Me parece fascinante que un computador solo entienda valores binarios y que las instrucciones de un lenguaje de alto nivel deban convertirse a lenguaje ensamblador y posteriormente a lenguaje máquina. Me gusta saber que puedo entender cómo se realiza esta transformación, y tener la capacidad de trabajar instrucciones tan cercanas al funcionamiento interno del hardware.

# REFERENCIAS


Barriga, J. (2005). *Historia y evolución del computador y microprocesador.* Universidad Peruana de Ciencias Aplicadas - UPC.

Castillo Catalán, M., & Claver Iborra, J. M. (2001). *Prácticas guiadas para el Ensamblador del MIPS R2000.*

Money Harris, D., & Harris, S. L. (2007). *Digital Design and Computer Architecture.* Morgan Kaufmann.

Ortega, J. (Noviembre de 2002). *Manual de Laboratorio para Arquitectura y Organización de Computadoras Usando XSPIM.* Obtenido de http://hp.fciencias.unam.mx/~jloa/AC/index.html

Prieto, A., Lloris, A., & Torres, J. C. (1989). *Introducción a la Informática.* McGraw-Hill.