Informe de Laboratorio: Estructura de Computadores

Nombre del Estudiante: [Geri Britney Guerrero Lizcano]
Fecha: [02/03/2026]
Asignatura: Estructura de Computadores

Enlace del repositorio en GitHub: [(https://github.com/geribgl/lab-pipeline-mips)]

---

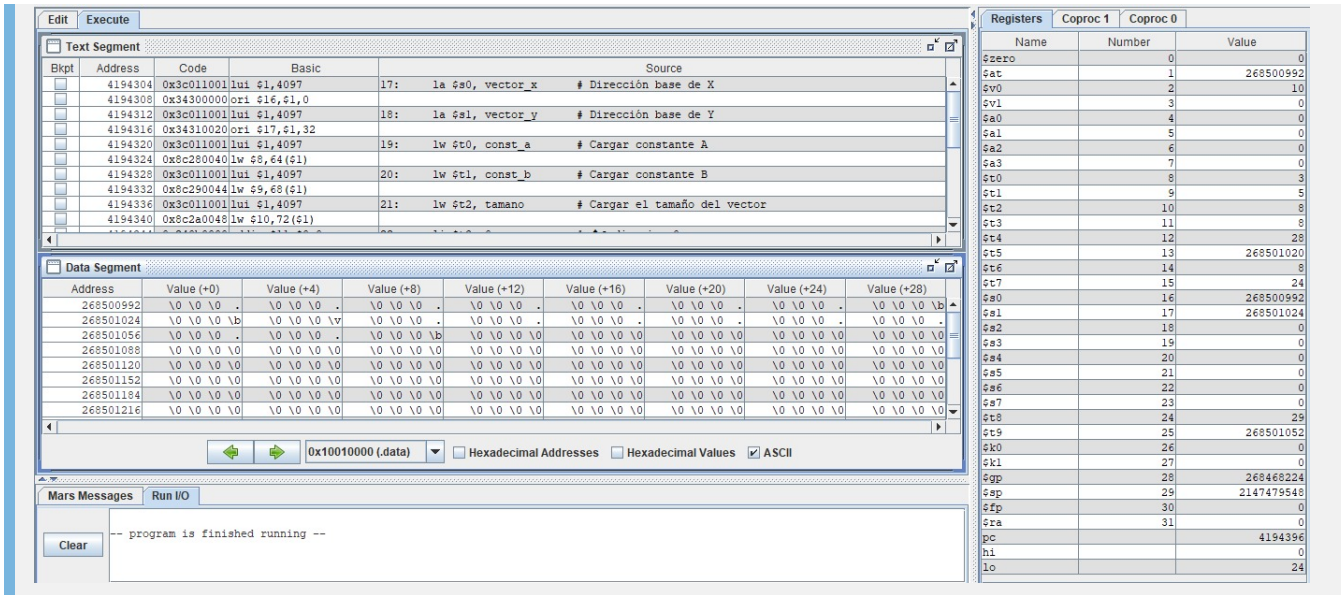
1. Análisis del Código Base

1.1. Evidencia de Ejecución

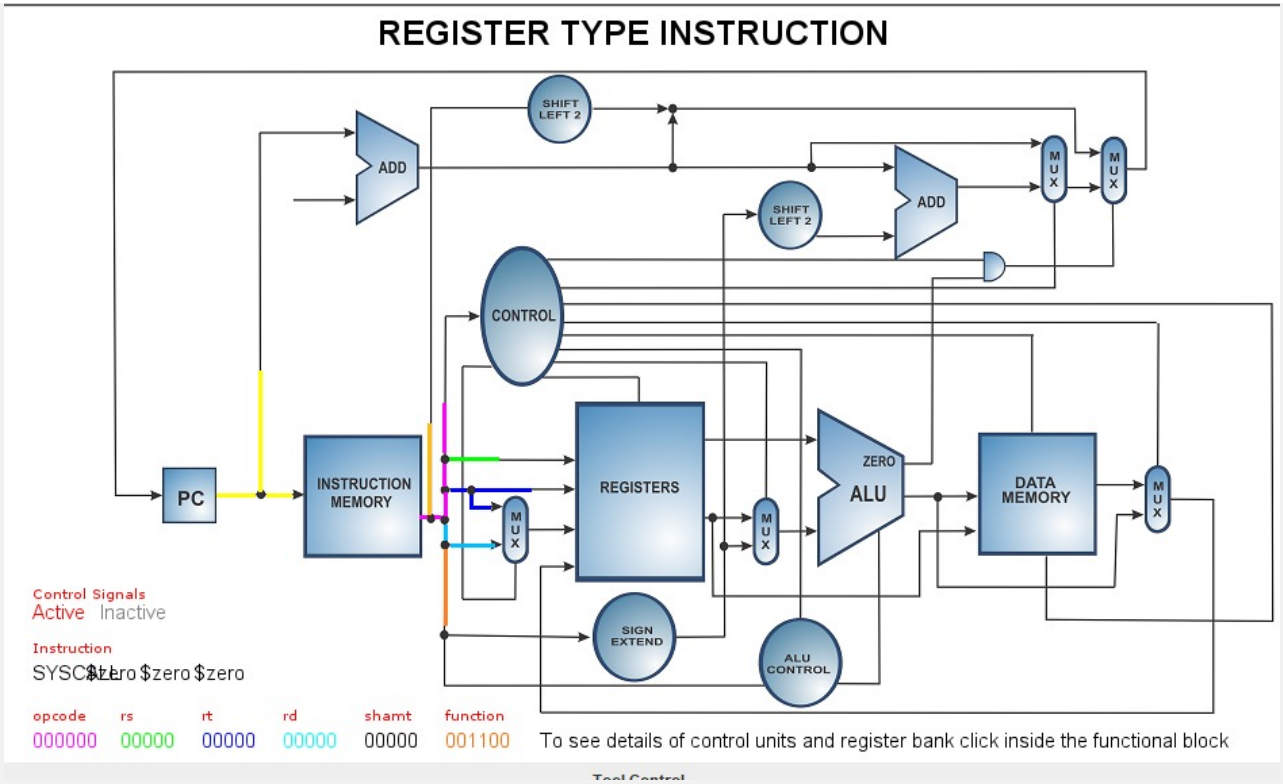
Adjunte aquí las capturas de pantalla de la ejecución del `programa_base.asm` utilizando las siguientes herramientas de MARS:

- **MIPS X-Ray** (Ventana con el Datapath animado).
- **Instruction Counter** (Contador de instrucciones totales).
- **Instruction Statistics** (Desglose por tipo de instrucción).

1. MIPS X-Ray



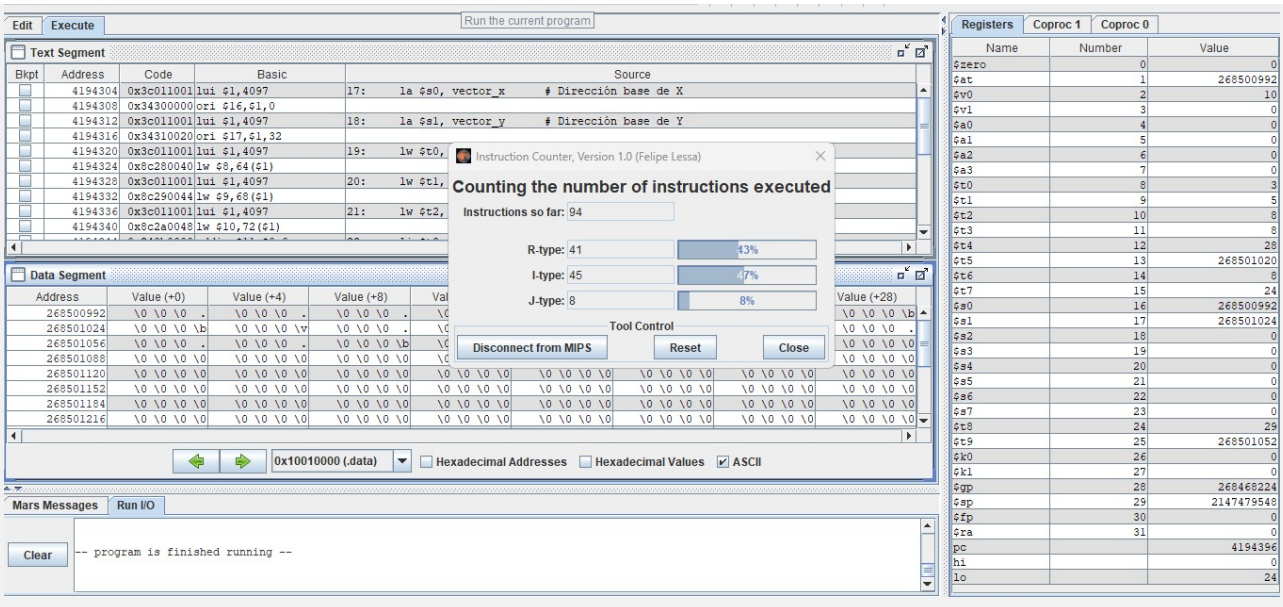
1.1. MIPS X-Ray del código base



** Figura 1. Ejecución del programa base en MIPS X-Ray.**

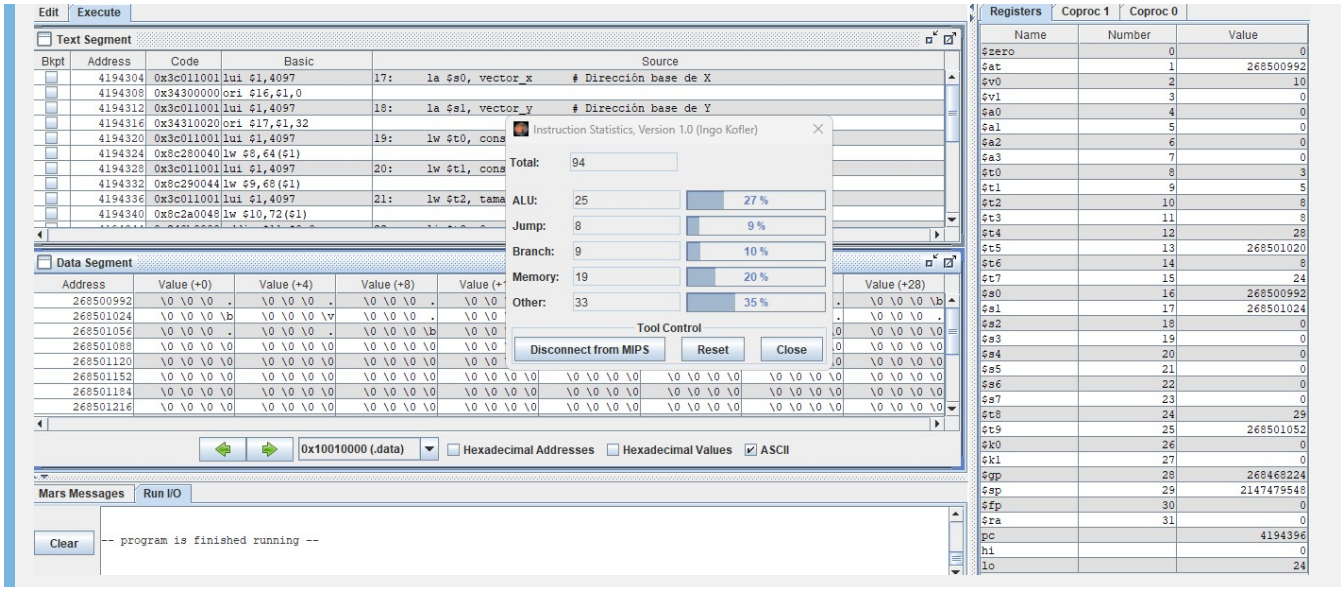
En esta visualización se observa el flujo de datos a través del datapath del procesador MIPS durante la ejecución del programa. Aquí se puede identificar cómo las instrucciones utilizan los registros, la ALU y la memoria antes de aplicar la optimización

2. Instruction Counter



---

3. Instruction Statistics



1.2. Identificación de Riesgos (Hazards)

Completa la siguiente tabla identificando las instrucciones que causan paradas en el pipeline:

Instrucción Causante	Instrucción Afectada	Tipo de Riesgo (Load-Use, etc.)	Ciclos de Parada
`lw $t6, 0($t5)`	`mul $t7, $t6, $t0`	Load-Use	1
`mul $t7, $t6,$t0`	`addu $t8, $t7, $t1`	Dependencia RAW	0

```
**La segunda no genera stall porque el pipeline puede manejarla con forwarding.**
→ Al no haber un adelanto de esre que cubra este caso especifico en un pipeline
estandar, se debe insertar 1 ciclo de burbuja para asi esperar el dato.
lw → mul
esto genera
1 stall * porque el dato cargado de memoria aún no está listo.

** El riesgo ocurre cuando la instrucción `mul` intenta usar el registro `$t6` en
la estapa Instruction Decode, pero el dato que esa cargado por `lw` solo esta
disponible de la etapa MEMORY ACCESS
```

1.2. Estadísticas y Análisis Teórico

Dado que MARS es un simulador funcional, el número de instrucciones ejecutadas será igual en ambas versiones. Sin embargo, en un procesador real, el tiempo de ejecución (ciclos) varía. Completa la siguiente tabla de análisis teórico:

```
    > **Se obtuvo en MARS (94 instrucciones)**
```

Métrica	Código Base	Código Optimizado

Métrica	Código Base	Código Optimizado
Instrucciones Totales (según MARS)	94	94
Stalls (Paradas) por iteración	1	0
Total de Stalls (8 iteraciones)	8	0
Ciclos Totales Estimados (Inst + Stalls)	102	94
CPI Estimado (Ciclos / Inst)	1.08	1.00

- Instrucciones totales: 94,

---

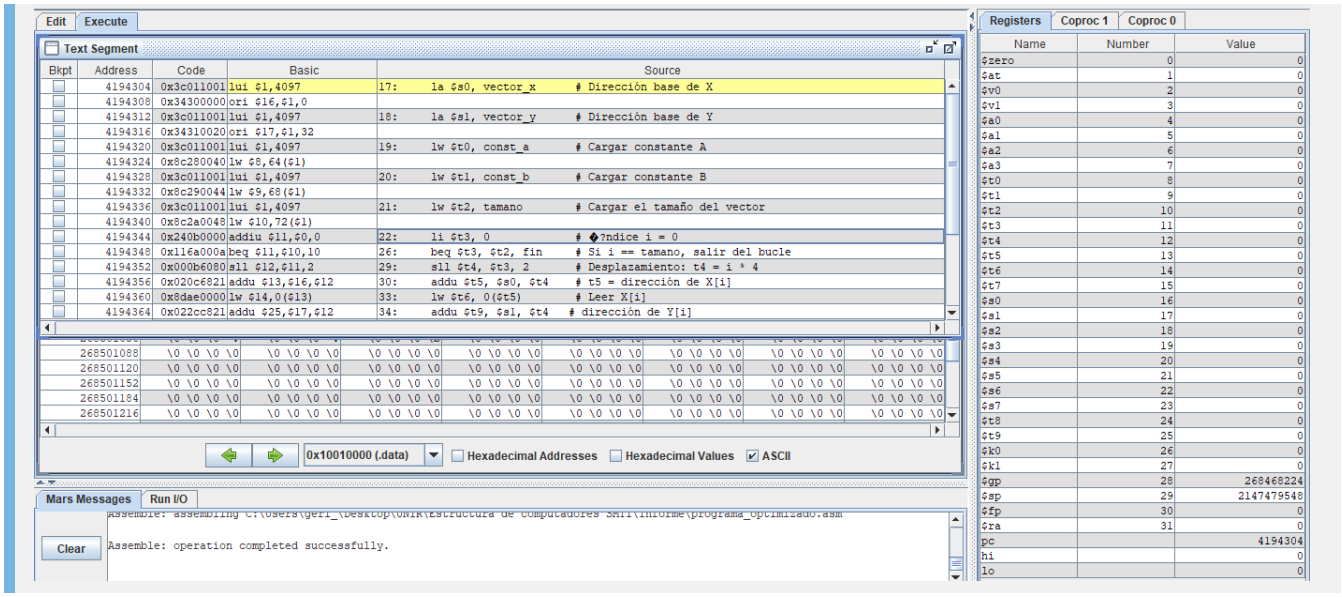
2. Optimización Propuesta

2.1. Evidencia de Ejecución (Código Optimizado)

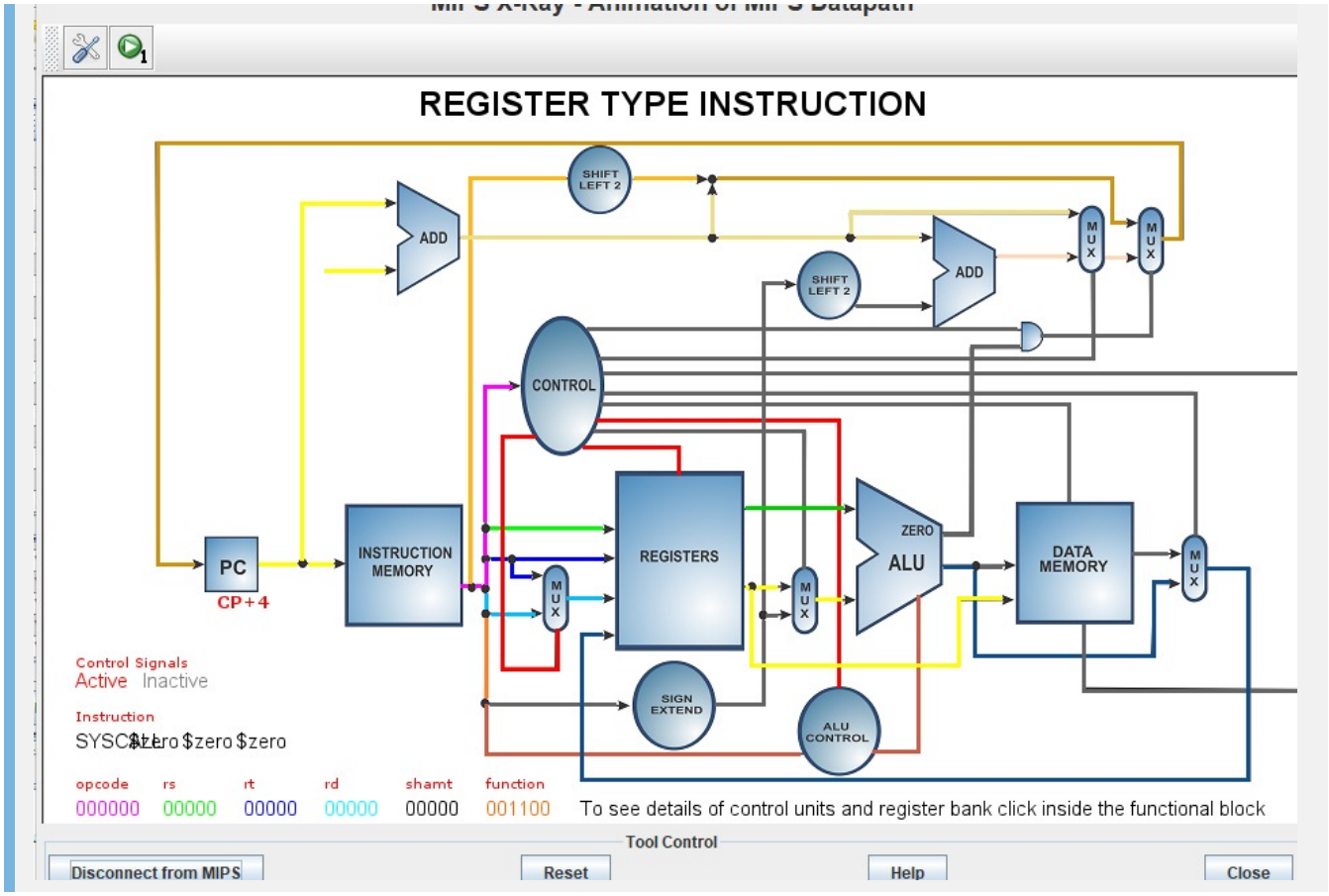
Adjunte aquí las capturas de pantalla de la ejecución del `programa_optimizado.asm` utilizando las mismas herramientas que en el punto 1.1:

- **MIPS X-Ray**.
- **Instruction Counter**.
- **Instruction Statistics**.

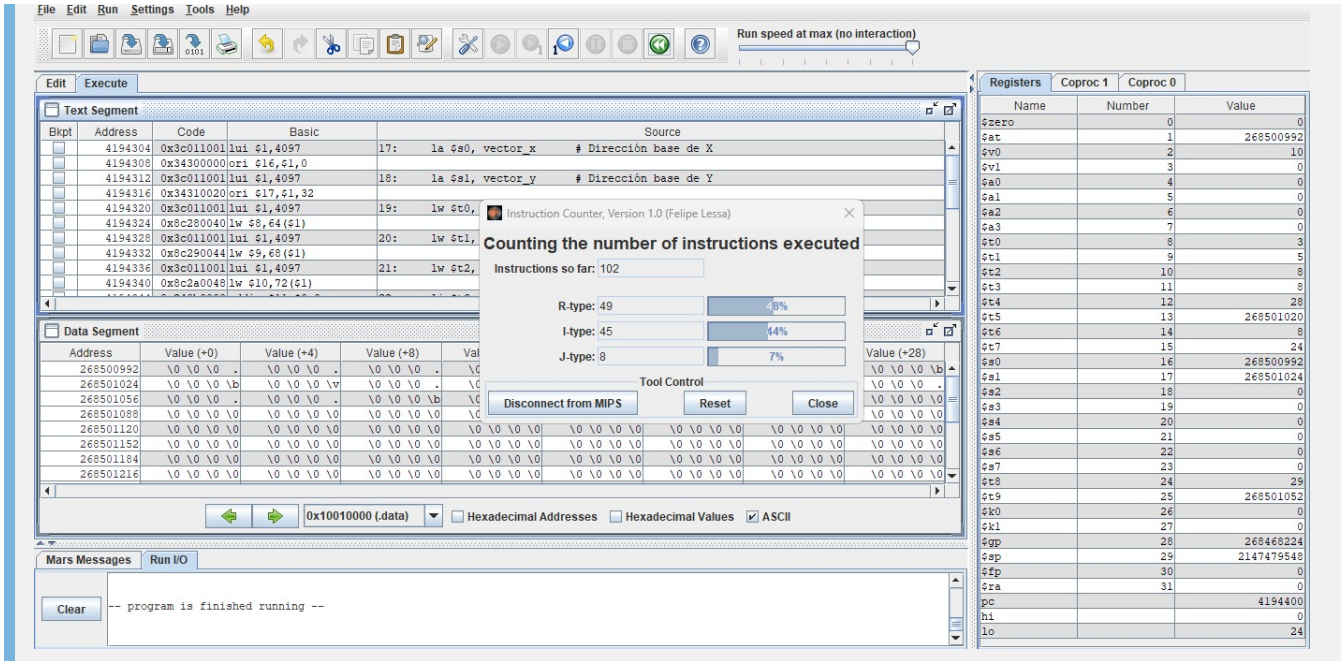
1. MIPS X-Ray. Optimizado



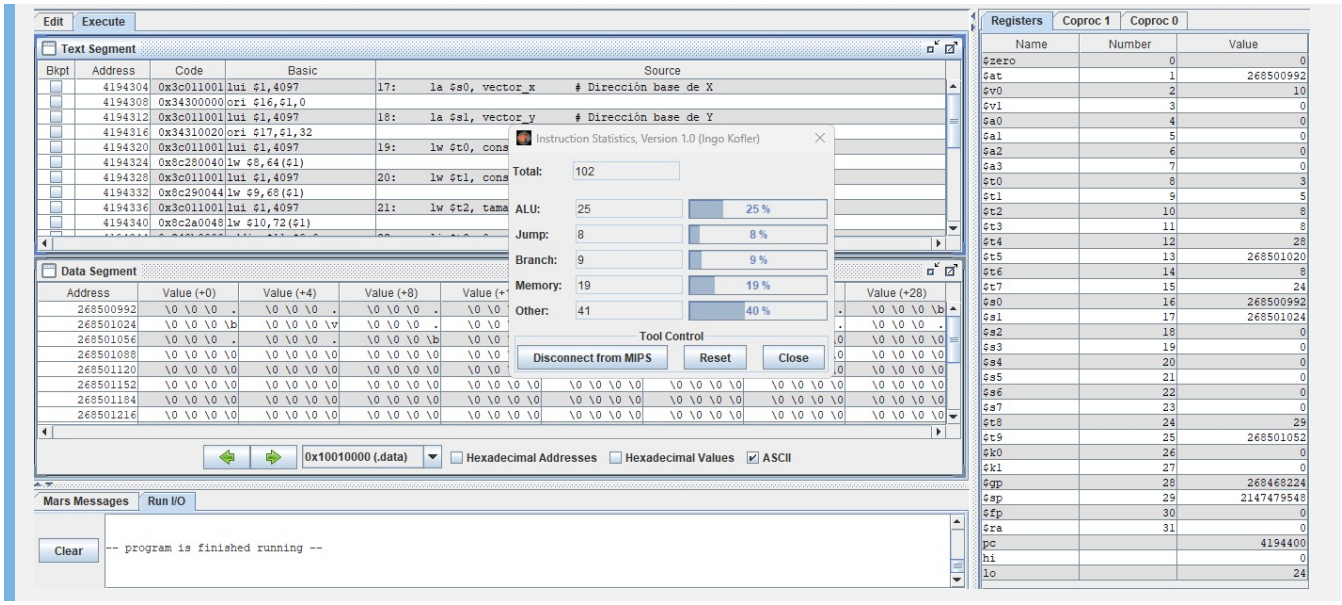
1.1. MIPS X-Ray del código base - Optimizado



2. Instruction Counter



3. Instruction Statistics



2.2. Código Optimizado

Pega aquí el fragmento de tu bucle `loop` reordenado:

```
# loop:
beq $t3, $t2, fin

sll  $t4, $t3, 2
addu $t5, $s0, $t4

lw   $t6, 0($t5)
addu $t9, $s1, $t4
mul  $t7, $t6, $t0
addu $t8, $t7, $t1
sw   $t8, 0($t9)
```

> *Instrucciones totales 94*

Codigo base

```
Stalls por iteración = 1
Iteraciones = 8
Total stalls = 8
Ciclos = 94 + 8 = 102
CPI = 102 / 94 = 1.08
```

Codigo Optimizado

```
Stalls = 0
Ciclos = 94
```

```
CPI = 94 / 94 = 1.00
```

> Las instrucciones totales se obtuvieron utilizando la herramienta Instruction Counter del simulador MARS. Este contador muestra cuántas instrucciones se ejecutan durante la ejecución completa del programa.

> Dado que el programa realiza un bucle de 8 iteraciones, el número de instrucciones ejecutadas se mantiene igual tanto en el código base como en el optimizado, ya que solo se reorganizaron instrucciones y no se agregaron ni eliminaron.

2.2. Justificación Técnica de la Mejora

Explica qué instrucción moviste y por qué colocarla entre el `lw` y el `mul` elimina el riesgo de datos:

> [En el código original, la instrucción `mul` utilizaba inmediatamente el resultado de la instrucción `lw`, lo que generaba un riesgo de datos tipo Load-Use en el pipeline. Para optimizar el programa, se movió la instrucción que calcula la dirección de memoria de Y[i] (addu $t9, $s1, $t4) entre el `lw` y la `mul`. Esto permite que el dato cargado desde memoria esté disponible antes de ser utilizado, eliminando el stall en el pipeline y mejorando el rendimiento del programa.]

---

3. Comparativa de Resultados

Métrica	**Código Base**	**Código Optimizado**	**Mejora (%)**
Ciclos Totales	102	94	7.8%
Stalls (Paradas)	8	0	100%
CPI	1.08	1.00	7.4%

---

4. Conclusiones

¿Qué impacto tiene la segmentación en el diseño de software de bajo nivel? ¿Es siempre posible eliminar todas las paradas?

> La segmentación permite mejorar el rendimiento del procesador al ejecutar múltiples instrucciones simultáneamente. Sin embargo, cuando existen dependencias de datos entre instrucciones consecutivas, se producen riesgos que generan paradas en el pipeline.

> En este laboratorio se observó que, mediante la reordenación de instrucciones, es posible reducir estos riesgos y mejorar el CPI del programa. No siempre es posible eliminar todas las paradas, pero una correcta organización del código permite optimizar el rendimiento del procesador.

Conclusion sobre el rendimiento:

> Se identifica en el pipeline presenta paradas principalmene por dependencias de datos tipo Load-Use. Se aplica reordenamiento de codigo para asi reducir el número de ciclo totales, separando la carga de memorira (lw) de la operacion aritmética que usa ese registro, asi permite que el procesador trabaje sin insertar las burbujas innecesarias.