Informe de Laboratorio: Estructura de Computadores

Nombre del Estudiante: Felipe Jimenez Melguizo
Fecha: 2026.02.26
Asignatura: Estructura de Computadores

Enlace del repositorio en GitHub: https://github.com/fjmelg97/Felipe-Jimenez-Melguizo-estructura-computadores-act01/

---

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).

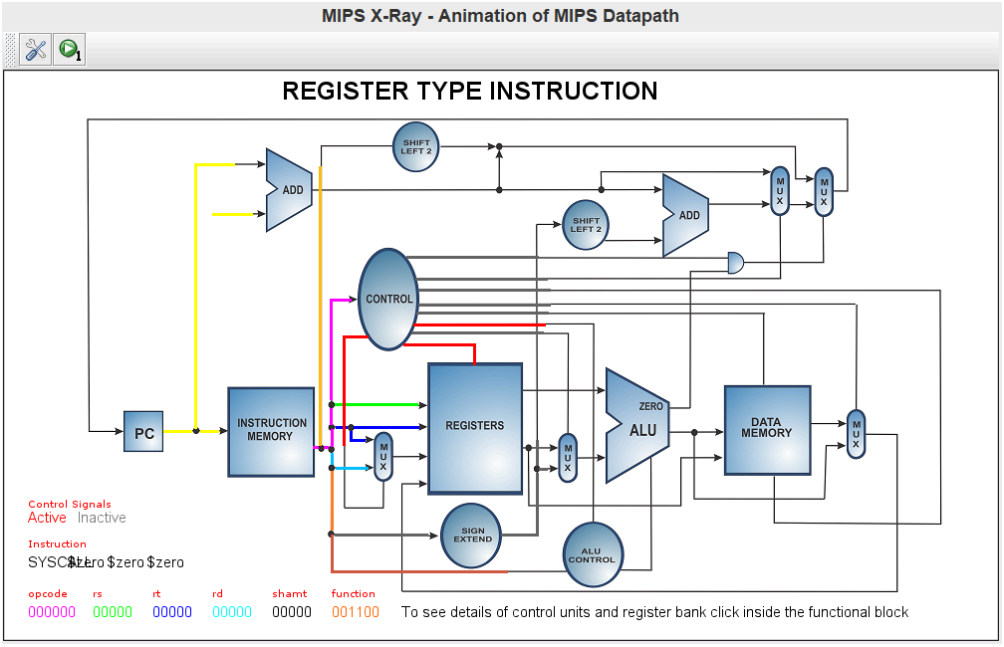

Figura 1. MIPS X-Ray del Programa Base

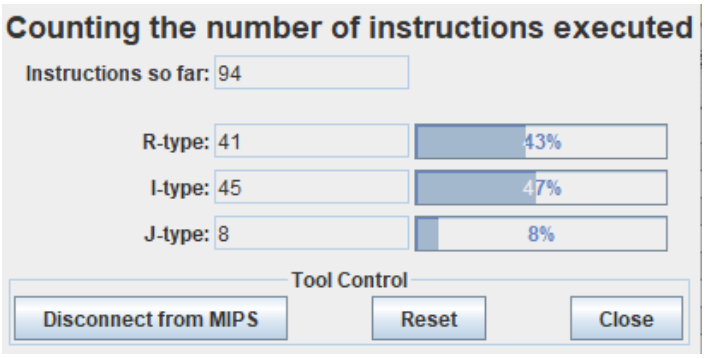

Figura 2. Instruction Counter del Programa Base

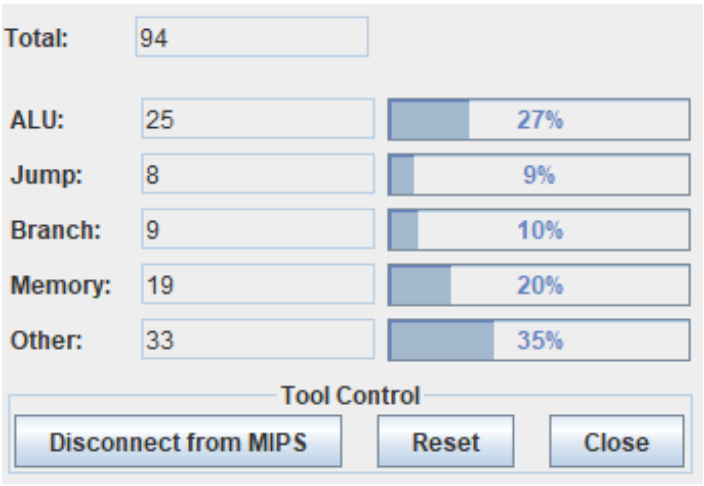

Figura 3. Instruction Statistics del Programa Base

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	2
`mul $t7, $t6, $t0`	`addu $t8, $t7, $t1`	Read After Write (RAW)	2

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:

Métrica	Código Base	Código Optimizado
Instrucciones Totales (según MARS)	94	94
Stalls (Paradas) por iteración	4	1
Total de Stalls (8 iteraciones)	32	8
Ciclos Totales Estimados (Inst + Stalls)	126	102
CPI Estimado (Ciclos / Inst)	~1.34	~1.09

---

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**.

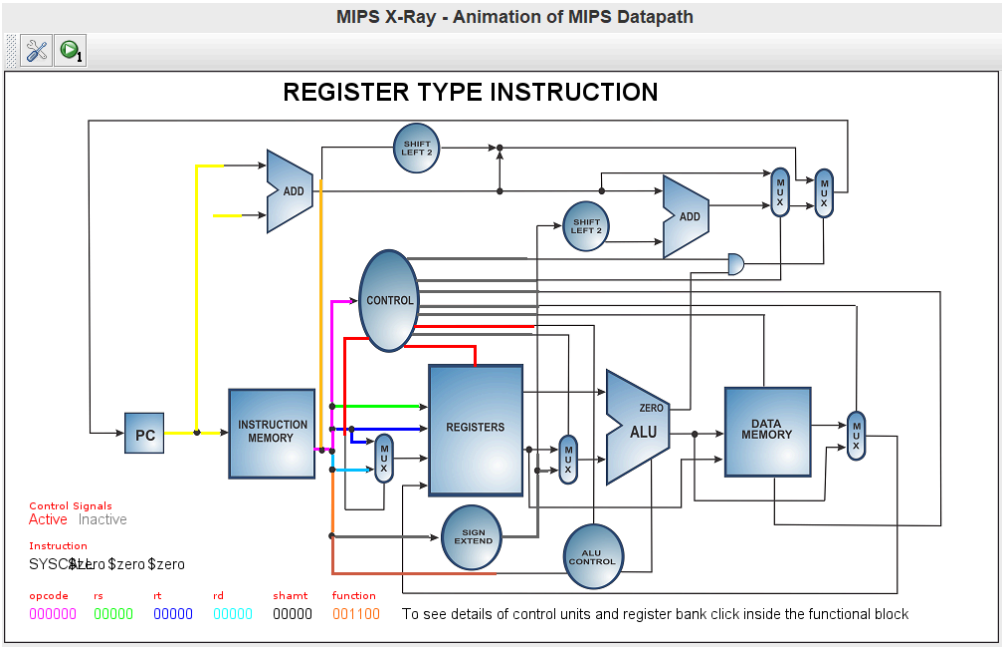

Figura 4. MIPS X-Ray del Programa Optimizado

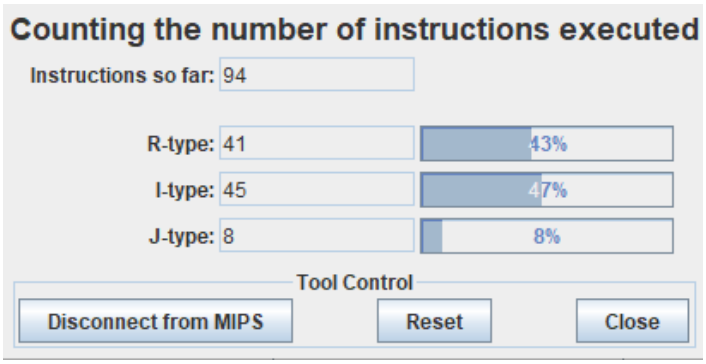

Figura 5. Instruction Counter del Programa Optimizado

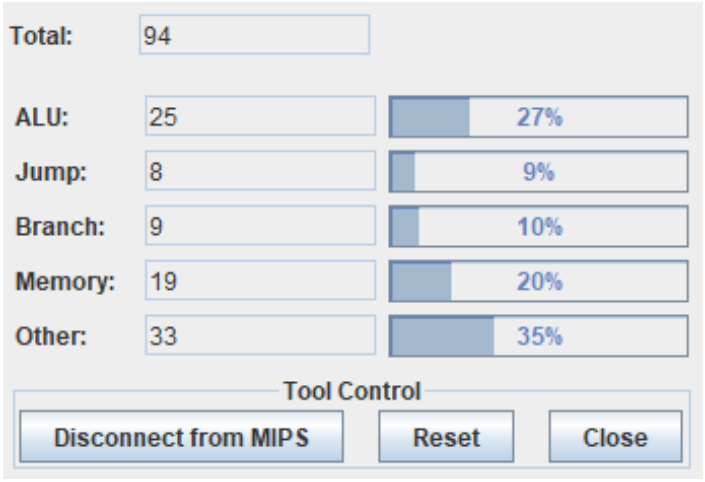

Figura 6. Instruction Statistics del Programa Optimizado

2.2. Código Optimizado

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

```
# Laboratorio: Estructura de Computadores
# Actividad: Optimizaci�n de Pipeline en Procesadores MIPS
# Objetivo: Calcular Y[i] = A * X[i] + B e identificar riesgos de datos.

# Optimizado por: Felipe Jimenez Melguizo
# 2025.02.26

.data
    vector_x: .word 1, 2, 3, 4, 5, 6, 7, 8
    vector_y: .space 32          # Espacio para 8 enteros (8 * 4 bytes)
    const_a:  .word 3
    const_b:  .word 5
    tamano:   .word 8

.text
.globl main

main:
    # --- Inicialización ---
    la $s0, vector_x      # Dirección base de X
    la $s1, vector_y      # Dirección base de Y
    lw $t0, const_a       # Cargar constante A
    lw $t1, const_b       # Cargar constante B
    lw $t2, tamano        # Cargar el tamaño del vector
    li $t3, 0             # �?ndice i = 0

loop:
    # --- Condición de salida ---
    beq $t3, $t2, fin     # Si i == tamano, salir del bucle

    # --- Cálculo de dirección de memoria ---
    sll $t4, $t3, 2       # Desplazamiento: t4 = i * 4
    addu $t5, $s0, $t4    # t5 = dirección de X[i]

    # --- Carga de dato ---
    lw $t6, 0($t5)        # Leer X[i]
    # NOTA: En un pipeline, la siguiente instrucción 'mul' depende de este 'lw'

    ### Instrucción reubicada para eliminar Riesgo de Datos: Load-Use
    addu $t9, $s1, $t4    # t9 = dirección de Y[i] - Se calcula la dirección de

    # --- Operación aritmética ---
    mul $t7, $t6, $t0     # t7 = X[i] * A  (ELIMINADO Riesgo de datos: Load-Use

    ### Instrucción reubicada para eliminar Riesgo de Datos: Dependencia mul-ac
    addi $t3, $t3, 1      # i = i + 1 - Se calcula i + 1 entre el cálculo de X[

    # --- Cálculo del valor total de Y[i]
    addu $t8, $t7, $t1    # t8 = t7 + B    (REDUCIDO Riesgo de datos: Dependenc

    # --- Almacenamiento de resultado ---
    sw $t8, 0($t9)        # Guardar resultado en Y[i]
```

```
    # --- salto ---
    j loop

fin:
    # --- Finalización del programa ---
    li $v0, 10           # Syscall para terminar ejecución
    syscall
```

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:

> Para el caso del hazard entre `lw` y `mul`, se tiene que, asumiendo desde hardware no se realice Forwarding, requiere de 2 stalls para que el valor del registro `$t6` se cargue desde memoria y esté disponible para ser utilizado por `mul`. En el pipeline de 5 etapas, para la instrucción `lw`, el valor cargado se obtiene en la **etapa 4 (MEM)** y es escrita en el respectivo registro para la **etapa 5 (WB)**. Únicamente despues de esta el valor cargado estará disponible para que `mul` disponga de esta (lo cual requiere en la **etapa 3 (EX)**, momento en el que la ALU realiza la operación aritmética de la multiplicación.)
>
> La optimización por software en este caso implica introducir una instrucción independiente de `$t6` y que a su vez la siguiente instrucción (`mul`) no dependa de esta. Considerando esto, se decide tomar la instrucción `addu $t9, $s1, $t4` (Cálculo de la dirección de memoria de `Y[i]`) para cubrir este propósito, lo cual realiza trabajo util para los dos ciclos que `mul` debe esperar a que `lw` cargue el valor.
>
> Para el caso de `mul $t7, $t6, $t0` y `addu $t8, $t7, $t1`, se tiene un riesgo de tipo **Read After Write (RAW)**, dado que addu intenta acceder al `$t7` en la **etapa 3 (EX)** sobre el que `mul` apenas almacena el dato en **etapa 5 (WB)**, dos ciclos después de que addu lo requiere, por lo que se introducen 2 stalls.
>
> Como optimización parcial, se propone insertar `addi $t3, $t3, 1` (Cálculo de `i = i + 1`) entre las dos mencionadas, de forma que aunque sigue persistiendo un stall, se elimina otro stall, lo cual optimiza el programa respecto al programa base propuesto.

---

3. Comparativa de Resultados

Métrica	Código Base	Código Optimizado	Mejora (%)
Ciclos Totales	94	94	0
Stalls (Paradas)	32	8	75
CPI	~1.34	~1.09	18

4. Conclusiones

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

> La segmentación tiene implicaciones muy positivas como lo es la velocidad de ejecución de un programa (es posible tener cierto nivel de paralelismo en la ejecución de instrucciones), pero de la misma forma conlleva la necesidad de verificar otras implicaciones como lo son los Hazard, donde es posible se introduzcan ineficiencias inherentes al programa, requiriendo de un análicis juicioso de la relación que pueedan tener instrucciones sucesivas en el rendimiento y la continuidad de la ejecución.
>
> No siempre es posible eliminar las paradas, puesto que esto depende en buena medida de la implementación realizada y la lógica del algoritmo trabajado. Algunas veces es viable reordenar las instrucciones para optimizar la ejecución de un programa, pero en otras ocasiones tiene prelación cierto orden de ejecución para la conservación de la lógica del programa mismo.