Informe de Laboratorio: Estructura de Computadores

Nombre del Estudiante: Maria Angelica Simoneg Flores
Fecha: 28 de febrero de 2026
Asignatura: Estructura de Computadores

Enlace del repositorio en GitHub: https://github.com/mariaangelicafloressimoneg/maria-angelica-flores-estructura-computadores-act01.git

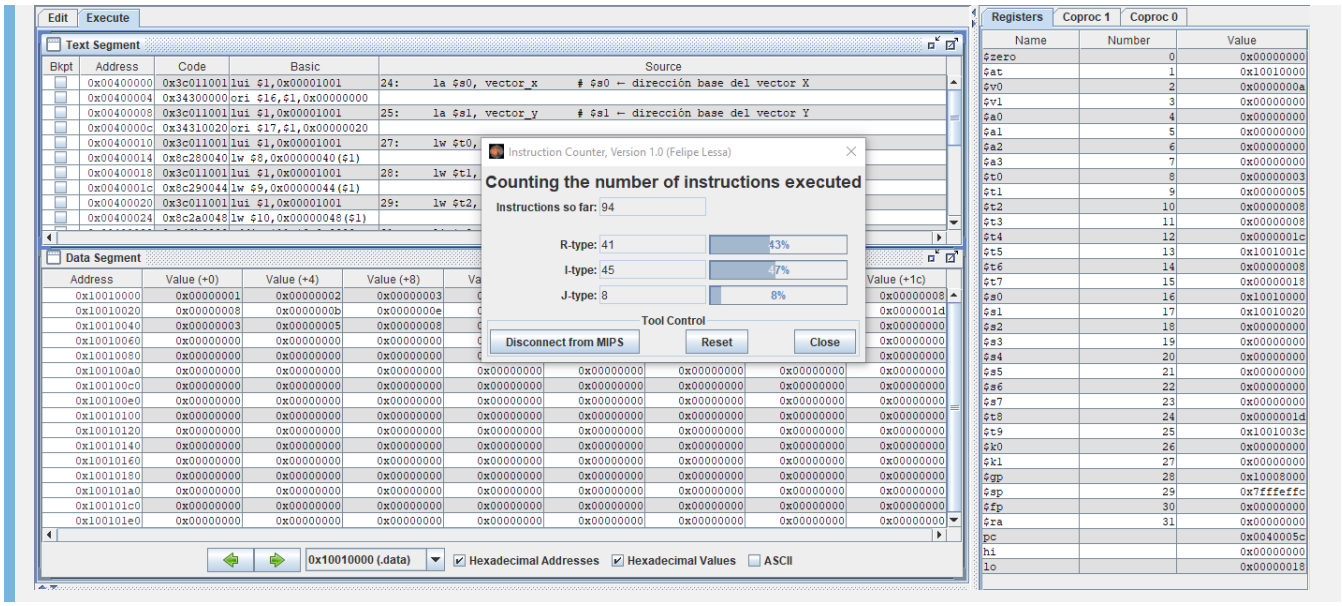
---

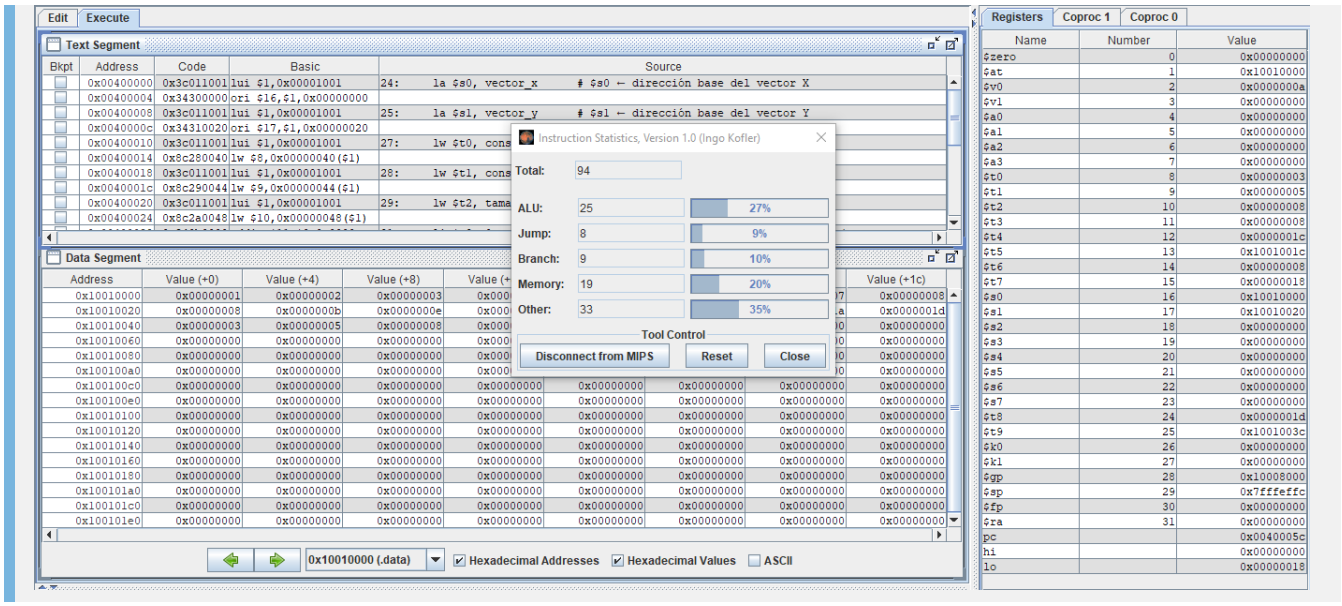
1. Análisis del Código Base

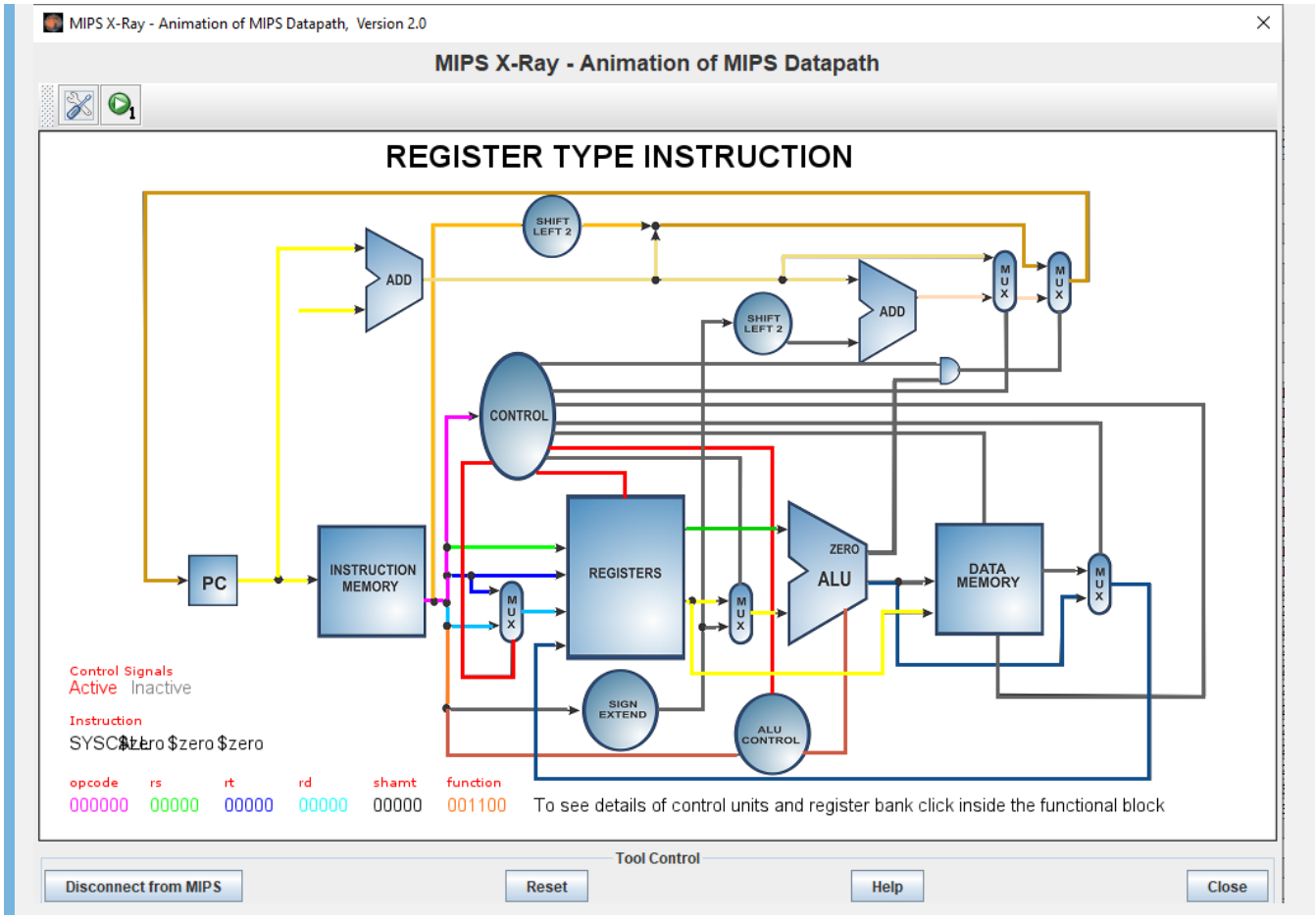
1.1. Evidencia de Ejecución

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).







1.2. 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 ciclo
`mul $t7, $t6, $t0`	`addu $t8, $t7, $t1`	Dependencia RAW aritmética	0 ciclos
`sll $t4, $t3, 2`	`addu $t5, $s0, $t4`	RAW dirección	0 ciclo

Instrucción Causante	Instrucción Afectada	Tipo de Riesgo (Load-Use, etc.)	Ciclos de Parada
`addu $t5, $s0, $t4`	`lw $t6, 0($t5)`	RAW dirección	0 ciclos
`addu $t9, $s1, $t4`	`sw $t8, 0($t9)`	RAW dirección	0 ciclos

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:

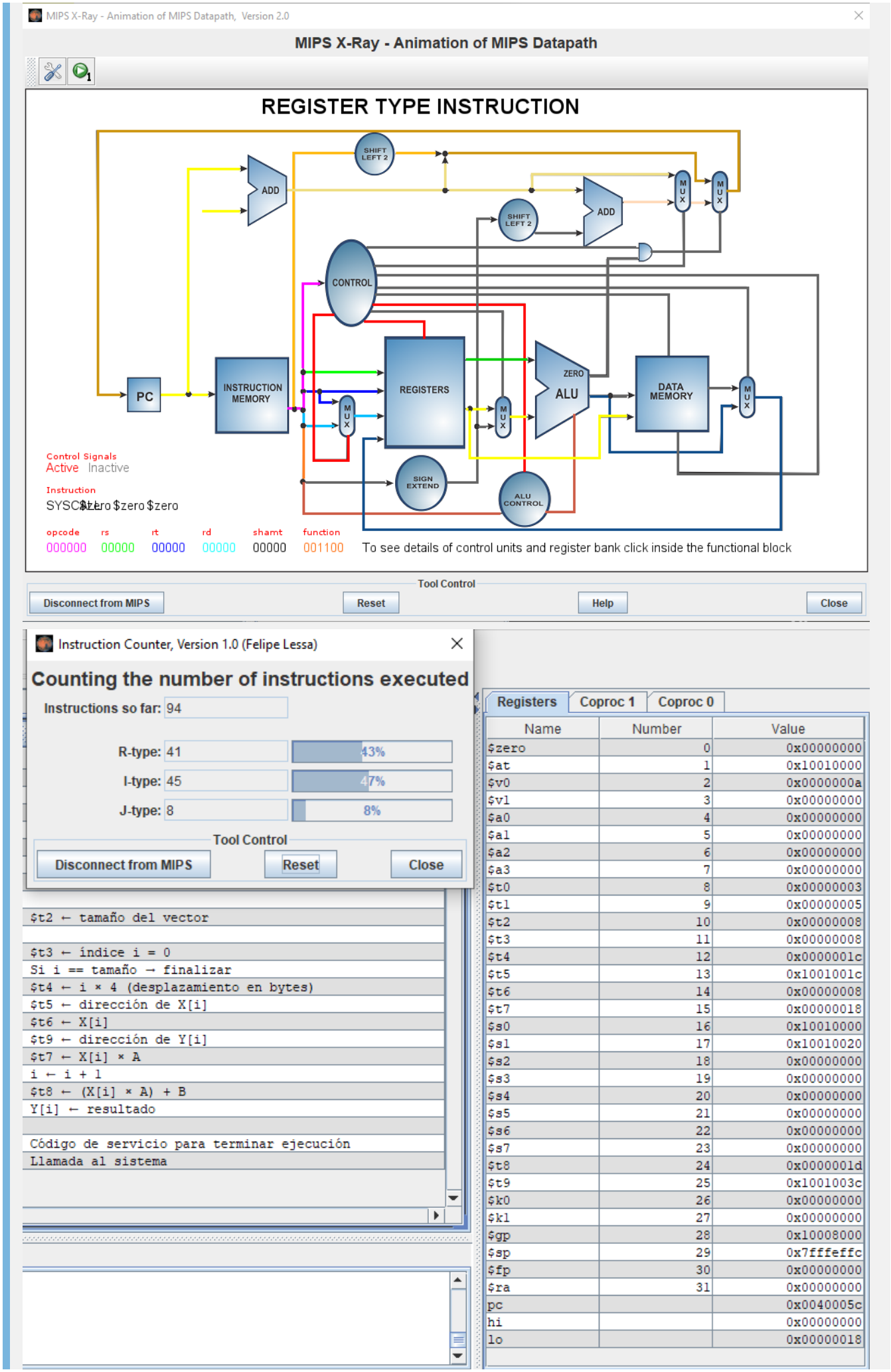
Métrica	Código Base	Código Optimizado
Instrucciones Totales (según MARS)	94	94
Stalls (Paradas) por iteración	1	0
Total de Stalls (8 iteraciones)	8	0
Ciclos Totales Estimados (Inst + Stalls)	102	94
CPI Estimado (Ciclos / Inst)	1.085	1

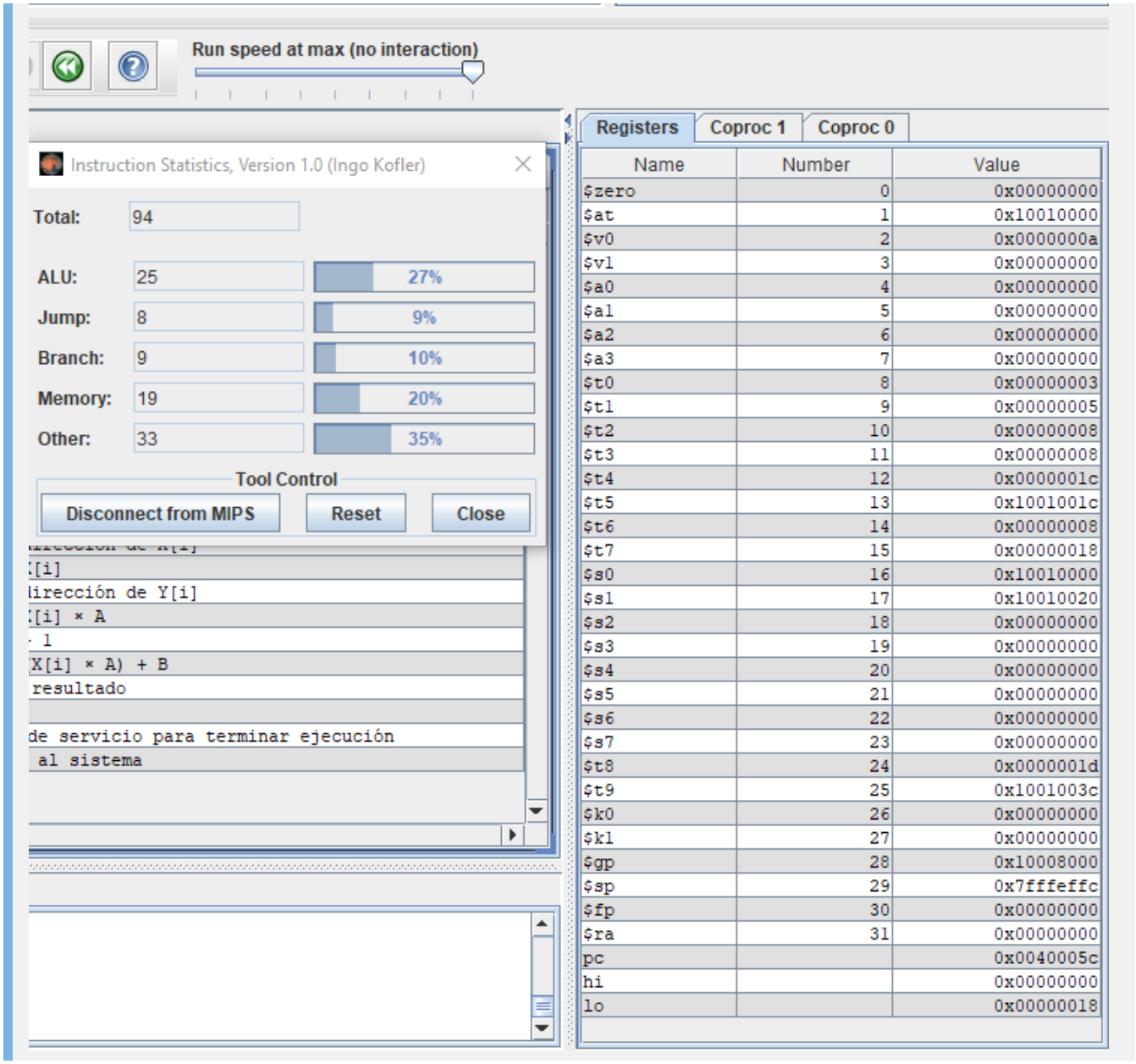
2. Optimización Propuesta

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**.


MIPS X-Ray - Animation of MIPS Datapath, Version 2.0
MIPS X-Ray - Animation of MIPS Datapath
REGISTER TYPE INSTRUCTION
SHIFT LEFT 2
ADD
SHIFT LEFT 2
ADD
MUX
MUX
CONTROL
PC
INSTRUCTION MEMORY
REGISTERS
ZERO
ALU
DATA MEMORY
MUX
MUX
MUX
SIGN EXTEND
ALU CONTROL
Control Signals
Active Inactive
Instruction
SYSCALL $zero $zero $zero
opcode rs rt rd shamt function
000000 00000 00000 00000 00000 001100
To see details of control units and register bank click inside the functional block
Tool Control
Disconnect from MIPS
Reset
Help
Close
Instruction Counter, Version 1.0 (Felipe Lessa)
Counting the number of instructions executed
Instructions so far: 94
R-type: 41 43%
I-type: 45 47%
J-type: 8 8%
Tool Control
Disconnect from MIPS
Reset
Close
$t2 ← tamaño del vector
$t3 ← índice i = 0
Si i == tamaño → finalizar
$t4 ← i × 4 (desplazamiento en bytes)
$t5 ← dirección de X[i]
$t6 ← X[i]
$t9 ← dirección de Y[i]
$t7 ← X[i] × A
i ← i + 1
$t8 ← (X[i] × A) + B
Y[i] ← resultado
Código de servicio para terminar ejecución
Llamada al sistema
Registers
Coproc 1
Coproc 0
Name Number Value
$zero 0 0x00000000
$at 1 0x10010000
$v0 2 0x0000000a
$v1 3 0x00000000
$a0 4 0x00000000
$a1 5 0x00000000
$a2 6 0x00000000
$a3 7 0x00000000
$t0 8 0x00000003
$t1 9 0x00000005
$t2 10 0x00000008
$t3 11 0x00000008
$t4 12 0x0000001c
$t5 13 0x1001001c
$t6 14 0x00000008
$t7 15 0x00000018
$s0 16 0x10010000
$s1 17 0x10010020
$s2 18 0x00000000
$s3 19 0x00000000
$s4 20 0x00000000
$s5 21 0x00000000
$s6 22 0x00000000
$s7 23 0x00000000
$t8 24 0x0000001d
$t9 25 0x1001003c
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x10008000
$sp 29 0x7fffeffc
$fp 30 0x00000000
$ra 31 0x00000000
pc 0x0040005c
hi 0x00000000
lo 0x00000018



2.2. Código Optimizado

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

```
# loop:
    # ------------------------------------------------------
    # 2. CONDICIÓN DE TERMINACIÓN DEL BUCLE
    # ------------------------------------------------------

    beq $t3, $t2, fin     # Si i == tamaño → terminar programa

    # ------------------------------------------------------
    # 3. CÁLCULO DE DIRECCIÓN DE X[i]
    # Cada entero ocupa 4 bytes en memoria
    # ------------------------------------------------------

    sll $t4, $t3, 2       # $t4 ← i * 4 (desplazamiento en memoria)
    addu $t5, $s0, $t4    # $t5 ← dirección de X[i]
```

```
    # ------------------------------------------------------
    # 4. LECTURA DEL ELEMENTO X[i]
    # ------------------------------------------------------

    lw $t6, 0($t5)        # $t6 ← X[i]

    # ------------------------------------------------------
    # 5. OPERACIÓN MATEMÁTICA
    # Y[i] = A * X[i] + B
    # ------------------------------------------------------

    mul $t7, $t6, $t0     # $t7 ← X[i] * A
    addu $t8, $t7, $t1    # $t8 ← (X[i] * A) + B

    # ------------------------------------------------------
    # 6. GUARDAR RESULTADO EN Y[i]
    # ------------------------------------------------------

    addu $t9, $s1, $t4    # $t9 ← dirección de Y[i]
    sw $t8, 0($t9)        # Y[i] ← resultado calculado

    # ------------------------------------------------------
    # 7. ACTUALIZACIÓN DEL ÍNDICE Y REPETICIÓN DEL BUCLE
    # ------------------------------------------------------

    addi $t3, $t3, 1      # i ← i + 1
    j loop                # Volver al inicio del bucle
```

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:

> La instrucción lw $t6, 0($t5) carga el valor X[i] desde memoria al registro $t6. En un pipeline MIPS de 5 etapas, el dato cargado no está disponible hasta después de la etapa MEM. Si la siguiente instrucción (mul $t7, $t6, $t0) intenta usar $t6 inmediatamente, se produce un riesgo Load-Use, obligando al procesador a insertar un ciclo de parada (stall).

> Al colocar addu $t9, $s1, $t4 entre lw y mul, el pipeline dispone de un ciclo adicional para que el dato cargado llegue correctamente al registro. Esta instrucción es independiente de $t6, por lo que puede ejecutarse sin conflicto y ocupa el ciclo que antes era una burbuja.

> La reordenación elimina la parada del pipeline al sustituir un ciclo inactivo por trabajo útil, manteniendo el flujo continuo de instrucciones y mejorando el rendimiento sin alterar el resultado del programa.

---

3. Comparativa de Resultados

Métrica	Código Base	Código Optimizado	Mejora (%)
Ciclos Totales	102	94	7.84%

Métrica	Código Base	Código Optimizado	Mejora (%)
Stalls (Paradas)	8	0	100%
CPI	1.085	1	7.83%

4. Conclusiones

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

> La segmentación del pipeline influye directamente en el diseño del software de bajo nivel, ya que el orden de las instrucciones puede afectar el rendimiento del procesador. En arquitecturas segmentadas, el programador deben considerar las dependencias de datos para evitar ciclos de inactividad. Mediante técnicas como la reordenación de instrucciones, es posible mantener ocupadas las etapas del pipeline y reducir el coste por instrucción CPI, mejorando la eficiencia sin modificar la funcionalidad del programa.

> En el caso analizado, el programa base presenta un riesgo de datos tipo Load-Use entre la instrucción de carga (lw) y la multiplicación (mul), lo que provoca una parada de un ciclo por iteración. La optimización aplicada inserta una instrucción independiente entre ambas, eliminando el ciclo de espera. Asimismo, la dependencia entre la multiplicación y la suma no genera paradas gracias al mecanismo de adelantamiento de datos - forwarding -, que permite reutilizar resultados antes de su escritura definitiva en el banco de registros.

> Sin embargo, no siempre es posible eliminar todas las paradas. Cuando no existen instrucciones independientes disponibles para reordenar, el hardware debe insertar stalls obligatorios para preservar la correcta ejecución del programa. Por tanto, la optimización por software puede reducir significativamente los riesgos de datos, pero su efectividad depende de la estructura del algoritmo y de las dependencias inherentes a la computación realizada.