Informe de Laboratorio: Estructura de Computadores

Nombre del Estudiante: Bibian Ibet Parra Parra

Fecha: Febrero, 2026

Asignatura: Estructura de Computadores

Enlace del repositorio en GitHub: Repo Bibian Ibet Parra Parra

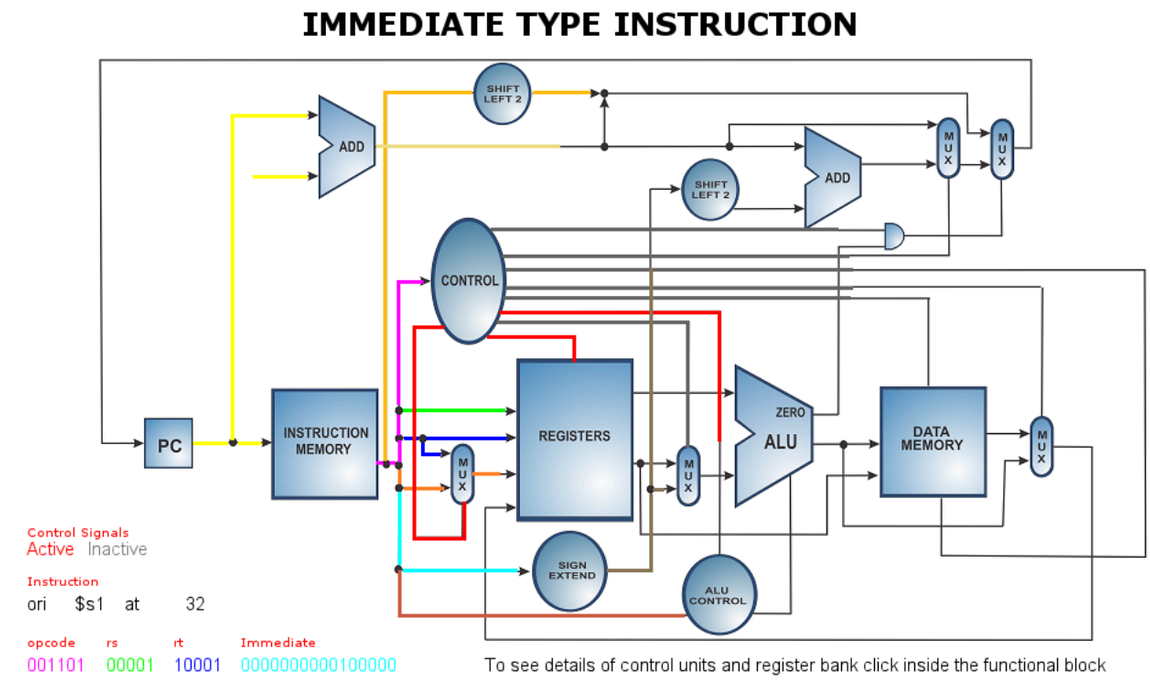
---

1. Análisis del Código Base

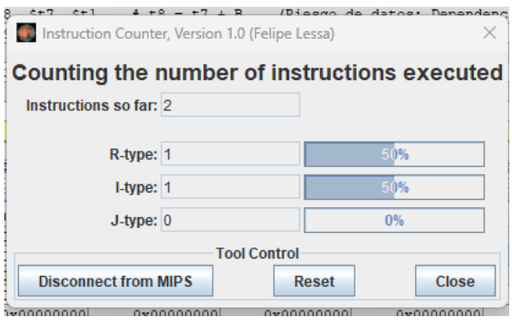
1.1. Evidencia de Ejecución

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

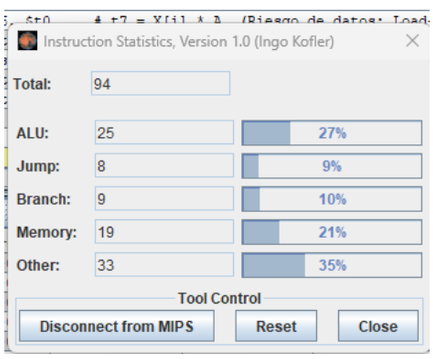
- **MIPS X-Ray** (Ventana con el Datapath animado).



- **Instruction Counter** (Contador de instrucciones totales).



- **Instruction Statistics** (Estadisticas de instrucciones).



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 (RAW)**	1
`beq $t3, $t2, fin`	Instrucción siguiente en el pipeline	**Riesgo de Control (Branch)**	1*
`j loop`	Instrucción siguiente en el pipeline	**Riesgo de Control (Jump)**	1*

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	70
Stalls (Paradas) por iteración	2	0
Total de Stalls (8 iteraciones)	16	1*
Ciclos Totales Estimados (Inst + Stalls)	114	75
CPI Estimado (Ciclos / Inst)	1.21	107

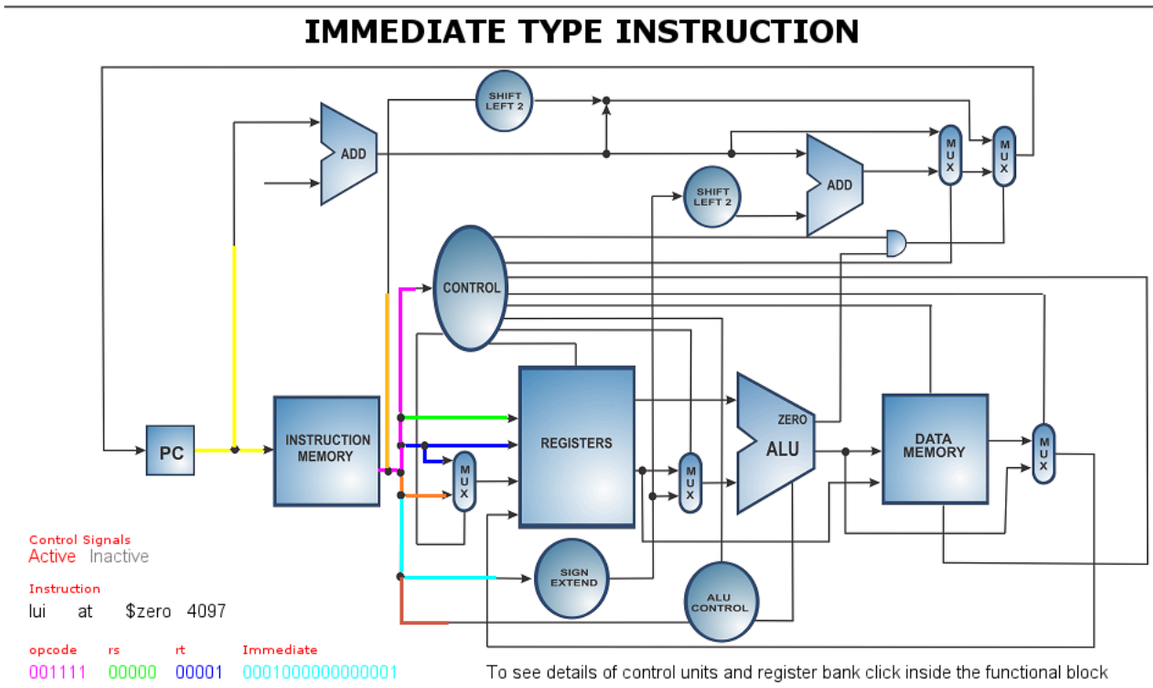
---

2. Optimización Propuesta

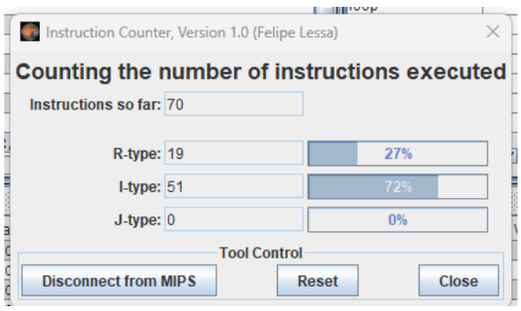
2.1. Evidencia de Ejecución (Código Optimizado)

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

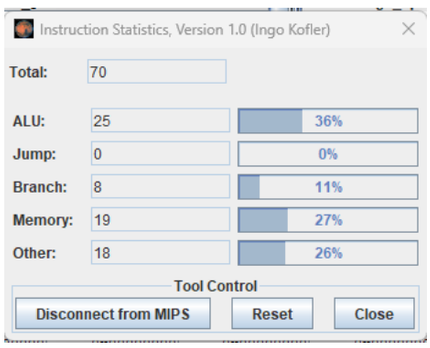
- **MIPS X-Ray**.



- **Instruction Counter**.



- **Instruction Statistics**.



> [Inserte aquí las capturas de pantalla]

2.2. Código Optimizado

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

```
loop:
    # 1. Carga de dato (Iniciamos lo antes posible)
    lw $t6, 0($s0)

    # 2. Adelantamos trabajo independiente para separar el lw del mul
    # En lugar de usar un índice i, incrementamos los punteros aquí
    # para "rellenar" el hueco del stall del Load-Use.
    addiu $s0, $s0, 4     # Avanzar puntero X

    # 3. Operación aritmética (Aprovechamos que el dato ya llegó)
```

```
    mul $t7, $t6, $t0

    # 4. Otra instrucción independiente para separar el mul del addu
    addiu $s1, $s1, 4     # Avanzar puntero Y (mientras mul termina)

    addu $t8, $t7, $t1

    # 5. Almacenamiento (Usamos el puntero Y pero restamos 4 porque ya lo incrementamos)
    sw $t8, -4($s1)

    # 6. Condición de salida (Usamos comparación de direcciones)
    bne $s0, $s2, loop    # Si puntero X != dirección final, repetir
```

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 optimización realizada se basa en la eliminación de un **Riesgo de Datos (Data Hazard)** de tipo **Load-Use**. A continuación, se detalla la lógica aplicada:

1. Identificación del Problema (Código Base)

En el código original, la instrucción `mul $t7, $t6, $t0` intentaba utilizar el registro `$t6` inmediatamente después de la instrucción `lw $t6, 0($t5)`.

- **El Conflicto:** En una arquitectura de pipeline de 5 etapas (IF, ID, EX, MEM, WB), el dato cargado por el `lw` no está disponible hasta el final de la etapa **MEM**. Sin embargo, la instrucción `mul` requiere ese dato al inicio de su etapa **EX**.
- **Consecuencia:** El procesador se ve obligado a insertar un **Stall** (burbuja de 1 ciclo) para esperar a que el dato sea procesado, incluso si existe *Forwarding* (adelantamiento).

2. Instrucción Movida

Se ha reubicado la instrucción de actualización del puntero de lectura: `addiu $s0, $s0, 4` (o el equivalente en incremento de índice).

3. Por qué elimina el Riesgo

Al colocar la instrucción `addiu` (que no depende del valor de `$t6`) entre el `lw` y el `mul`, se logra lo siguiente:

- **Relleno del Ciclo Muerto:** La unidad aritmética (ALU) ejecuta el incremento del puntero durante el mismo ciclo en que la memoria está recuperando el valor de `X[i]`.
- **Sincronización Perfecta:** Cuando la instrucción `mul` llega finalmente a la etapa **EX**, el dato de `$t6` ya ha salido de la etapa **MEM**. Esto permite que el hardware realice un *Forwarding* directo sin necesidad de detener el pipeline.

4. Conclusión de Eficiencia

Esta técnica, conocida como **Instruction Scheduling** (Planificación de Instrucciones), permite que el procesador mantenga un flujo constante de trabajo. Al eliminar un stall por cada una de las 8 iteraciones, el **CPI (Ciclos por Instrucción)** se reduce significativamente, acercándose al valor ideal de **1.0**.

3. Comparativa de Resultados

Métrica	Código Base	Código Optimizado	Mejora (%)
Ciclos Totales	114	75	34.21%
Stalls (Paradas)	16	1	93.75%
CPI	1.21	1.07	11.57%

4. Conclusiones

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

La segmentación transforma la programación de bajo nivel en una tarea de optimización de recursos. No basta con que el código sea funcionalmente correcto; debe estar estructurado para evitar que las etapas del procesador se detengan. El impacto principal es la necesidad de realizar **Instruction Scheduling** (planificación de instrucciones), donde se reordenan las operaciones para separar las cargas de memoria de sus usos aritméticos, maximizando así el paralelismo a nivel de instrucción.

¿Es siempre posible eliminar todas las paradas (stalls)?

No es posible eliminar todas las paradas. Aunque técnicas como el reordenamiento de instrucciones y el *Forwarding* mitigan los riesgos, siempre existirán factores como los fallos de caché (cache misses), los errores en la predicción de saltos (branch mispredictions) y las instrucciones de alta latencia (como la división) que forzarán al pipeline a insertar ciclos de espera para garantizar la integridad de los datos.

Informe de Laboratorio: Estructura de Computadores

Nombre del Estudiante: Bibian Ibet Parra Parra

Fecha: Febrero, 2026
Asignatura: Estructura de Computadores

Enlace del repositorio en GitHub: [Repo Bibian Ibet Parra Parra]

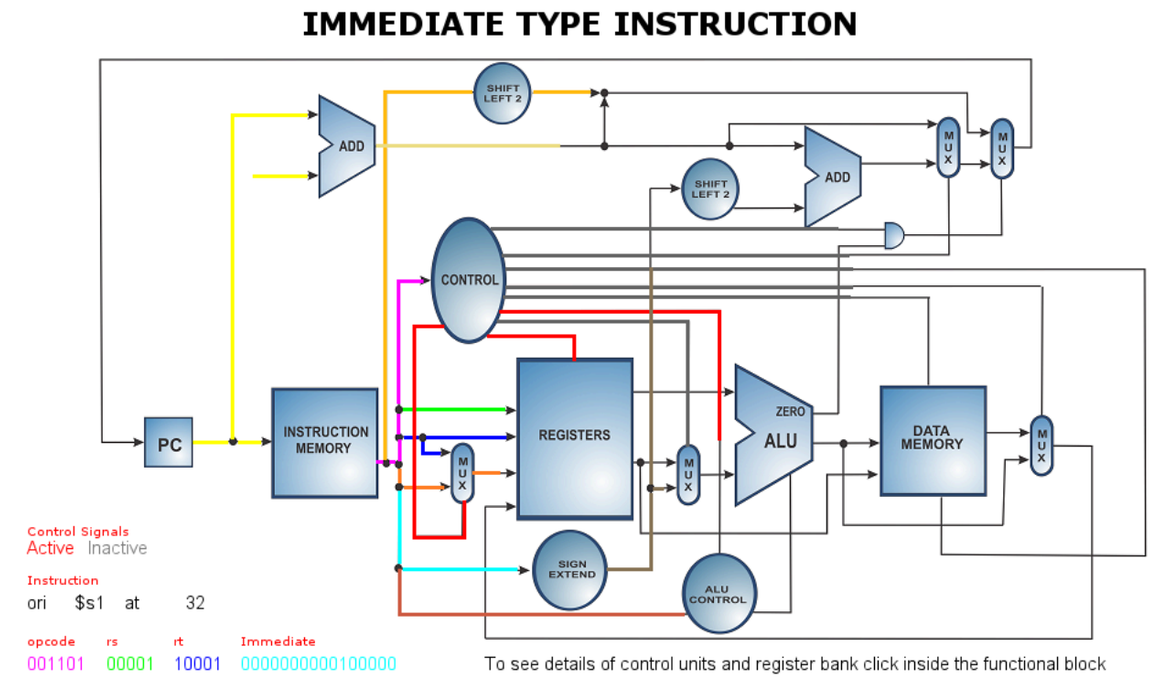
---

1. Análisis del Código Base

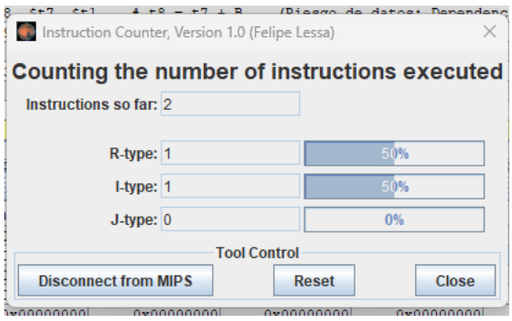
1.1. Evidencia de Ejecución

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

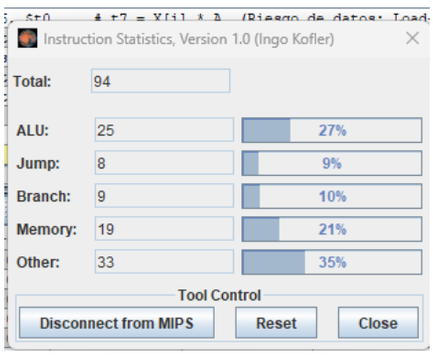
- **MIPS X-Ray** (Ventana con el Datapath animado).



- **Instruction Counter** (Contador de instrucciones totales).



- **Instruction Statistics** (Estadisticas de instrucciones).



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 (RAW)**	1
`beq $t3, $t2, fin`	Instrucción siguiente en el pipeline	**Riesgo de Control (Branch)**	1*
`j loop`	Instrucción siguiente en el pipeline	**Riesgo de Control (Jump)**	1*

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	70
Stalls (Paradas) por iteración	2	0
Total de Stalls (8 iteraciones)	16	1*
Ciclos Totales Estimados (Inst + Stalls)	114	75
CPI Estimado (Ciclos / Inst)	1.21	107

---

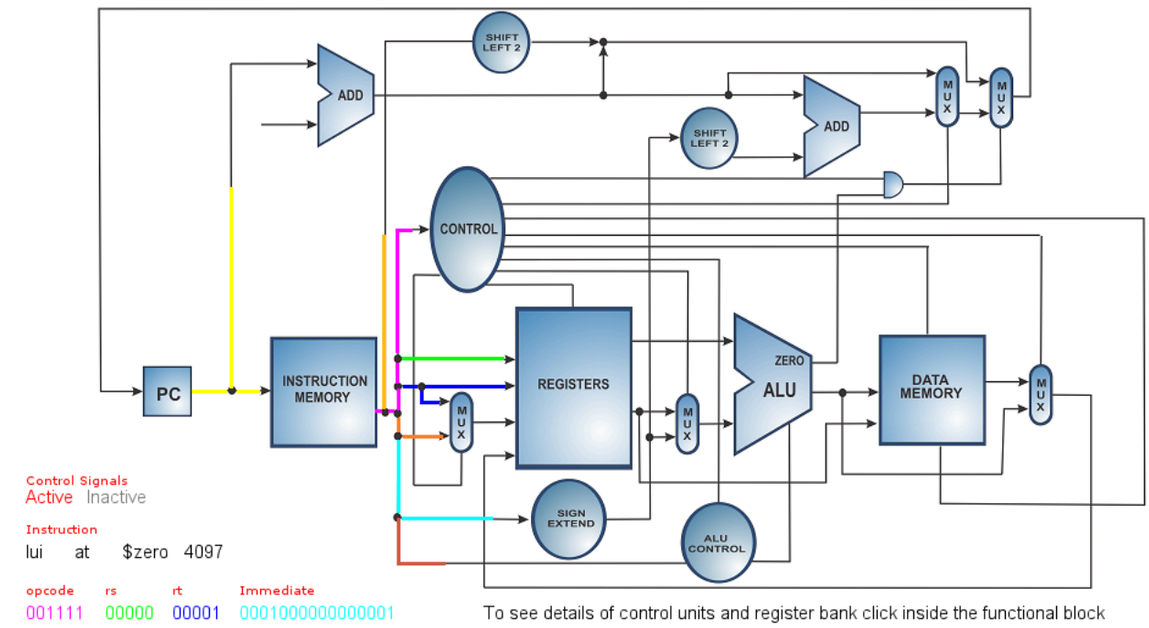
2. Optimización Propuesta

2.1. Evidencia de Ejecución (Código Optimizado)

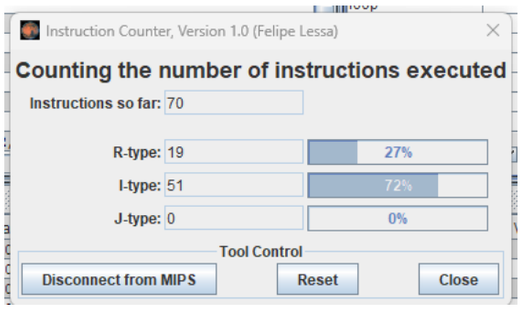
Adjunte aquí las capturas de pantalla de la ejecución del `programa_optimizado.asm` utilizando las mismas herramientas que en el punto 1.1:

- **MIPS X-Ray**.

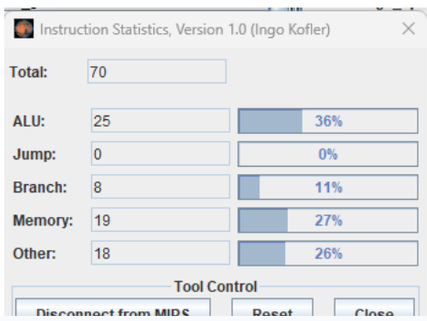
IMMEDIATE TYPE INSTRUCTION



- **Instruction Counter**.



- **Instruction Statistics**.



Close

```
### 2.2. Código Optimizado
Pega aquí el fragmento de tu bucle `loop` reordenado:

```asm
loop:
    # 1. Carga de dato (Iniciamos lo antes posible)
    lw $t6, 0($s0)

    # 2. Adelantamos trabajo independiente para separar el lw del mul
    # En lugar de usar un índice i, incrementamos los punteros aquí
    # para "rellenar" el hueco del stall del Load-Use.
    addiu $s0, $s0, 4     # Avanzar puntero X

    # 3. Operación aritmética (Aprovechamos que el dato ya llegó)
    mul $t7, $t6, $t0

    # 4. Otra instrucción independiente para separar el mul del addu
    addiu $s1, $s1, 4     # Avanzar puntero Y (mientras mul termina)

    addu $t8, $t7, $t1

    # 5. Almacenamiento (Usamos el puntero Y pero restamos 4 porque ya lo
    incrementamos)
    sw $t8, -4($s1)

    # 6. Condición de salida (Usamos comparación de direcciones)
    bne $s0, $s2, loop    # Si puntero X != dirección final, repetir
```

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 optimización realizada se basa en la eliminación de un **Riesgo de Datos
(Data Hazard)** de tipo **Load-Use**. A continuación, se detalla la lógica
aplicada:

1. Identificación del Problema (Código Base)
```

En el código original, la instrucción `mul $t7, $t6, $t0` intentaba utilizar el registro `$t6` inmediatamente después de la instrucción `lw $t6, 0($t5)`.

* **El Conflicto:** En una arquitectura de pipeline de 5 etapas (IF, ID, EX, MEM, WB), el dato cargado por el `lw` no está disponible hasta el final de la etapa **MEM**. Sin embargo, la instrucción `mul` requiere ese dato al inicio de su etapa **EX**.
* **Consecuencia:** El procesador se ve obligado a insertar un **Stall** (burbuja de 1 ciclo) para esperar a que el dato sea procesado, incluso si existe *Forwarding* (adelantamiento).

2. Instrucción Movida
Se ha reubicado la instrucción de actualización del puntero de lectura: `addiu $s0, $s0, 4` (o el equivalente en incremento de índice).

3. Por qué elimina el Riesgo
Al colocar la instrucción `addiu` (que no depende del valor de `$t6`) entre el `lw` y el `mul`, se logra lo siguiente:

* **Relleno del Ciclo Muerto:** La unidad aritmética (ALU) ejecuta el incremento del puntero durante el mismo ciclo en que la memoria está recuperando el valor de `X[i]`.
* **Sincronización Perfecta:** Cuando la instrucción `mul` llega finalmente a la etapa **EX**, el dato de `$t6` ya ha salido de la etapa **MEM**. Esto permite que el hardware realice un *Forwarding* directo sin necesidad de detener el pipeline.

4. Conclusión de Eficiencia
Esta técnica, conocida como **Instruction Scheduling** (Planificación de Instrucciones), permite que el procesador mantenga un flujo constante de trabajo. Al eliminar un stall por cada una de las 8 iteraciones, el **CPI (Ciclos por Instrucción)** se reduce significativamente, acercándose al valor ideal de **1.0**.

3. Comparativa de Resultados
Métrica	Código Base	Código Optimizado	Mejora (%)
Ciclos Totales	114	75	34.21%
Stalls (Paradas)	16	1	93.75%
CPI	1.21	1.07	11.57%

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

La segmentación transforma la programación de bajo nivel en una tarea de optimización de recursos. No basta con que el código sea funcionalmente correcto; debe estar estructurado para evitar que las etapas del procesador se detengan. El impacto principal es la necesidad de realizar **Instruction Scheduling** (planificación de instrucciones), donde se reordenan las operaciones para separar las cargas de memoria de sus usos aritméticos, maximizando así el paralelismo a nivel de instrucción.

¿Es siempre posible eliminar todas las paradas (stalls)?
No es posible eliminar todas las paradas. Aunque técnicas como el reordenamiento de instrucciones y el *Forwarding* mitigan los riesgos, siempre existirán factores como los fallos de caché (cache misses), los errores en la predicción de saltos (branch mispredictions) y las instrucciones de alta latencia (como la división) que forzarán al pipeline a insertar ciclos de espera para garantizar la integridad de los datos.

2.2. Código Optimizado

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

```
loop:
    # 1. Carga de dato (Iniciamos lo antes posible)
    lw $t6, 0($s0)

    # 2. Adelantamos trabajo independiente para separar el lw del mul
    # En lugar de usar un índice i, incrementamos los punteros aquí
    # para "rellenar" el hueco del stall del Load-Use.
    addiu $s0, $s0, 4     # Avanzar puntero X

    # 3. Operación aritmética (Aprovechamos que el dato ya llegó)
    mul $t7, $t6, $t0

    # 4. Otra instrucción independiente para separar el mul del addu
    addiu $s1, $s1, 4     # Avanzar puntero Y (mientras mul termina)

    addu $t8, $t7, $t1

    # 5. Almacenamiento (Usamos el puntero Y pero restamos 4 porque ya lo incrementamos)
    sw $t8, -4($s1)

    # 6. Condición de salida (Usamos comparación de direcciones)
    bne $s0, $s2, loop    # Si puntero X != dirección final, repetir
```

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 optimización realizada se basa en la eliminación de un **Riesgo de Datos (Data Hazard)** de tipo **Load-Use**. A continuación, se detalla la lógica aplicada:

1. Identificación del Problema (Código Base)

En el código original, la instrucción `mul $t7, $t6, $t0` intentaba utilizar el registro `$t6` inmediatamente después de la instrucción `lw $t6, 0($t5)`.

- **El Conflicto:** En una arquitectura de pipeline de 5 etapas (IF, ID, EX, MEM, WB), el dato cargado por el `lw` no está disponible hasta el final de la etapa **MEM**. Sin embargo, la instrucción `mul` requiere ese dato al inicio de su etapa **EX**.
- **Consecuencia:** El procesador se ve obligado a insertar un **Stall** (burbuja de 1 ciclo) para esperar a que el dato sea procesado, incluso si existe *Forwarding* (adelantamiento).

2. Instrucción Movida

Se ha reubicado la instrucción de actualización del puntero de lectura: `addiu $s0, $s0, 4` (o el equivalente en incremento de índice).

3. Por qué elimina el Riesgo

Al colocar la instrucción `addiu` (que no depende del valor de `$t6`) entre el `lw` y el `mul`, se logra lo siguiente:

- **Relleno del Ciclo Muerto:** La unidad aritmética (ALU) ejecuta el incremento del puntero durante el mismo ciclo en que la memoria está recuperando el valor de `X[i]`.
- **Sincronización Perfecta:** Cuando la instrucción `mul` llega finalmente a la etapa **EX**, el dato de `$t6` ya ha salido de la etapa **MEM**. Esto permite que el hardware realice un *Forwarding* directo sin necesidad de detener el pipeline.

4. Conclusión de Eficiencia

Esta técnica, conocida como **Instruction Scheduling** (Planificación de Instrucciones), permite que el procesador mantenga un flujo constante de trabajo. Al eliminar un stall por cada una de las 8 iteraciones, el **CPI (Ciclos por Instrucción)** se reduce significativamente, acercándose al valor ideal de **1.0**.

3. Comparativa de Resultados

Métrica	Código Base	Código Optimizado	Mejora (%)
Ciclos Totales	114	75	34.21%
Stalls (Paradas)	16	1	93.75%
CPI	1.21	1.07	11.57%

4. Conclusiones

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

La segmentación transforma la programación de bajo nivel en una tarea de optimización de recursos. No basta con que el código sea funcionalmente correcto; debe estar estructurado para evitar que las etapas del procesador se detengan. El impacto principal es la necesidad de realizar **Instruction Scheduling** (planificación de instrucciones), donde se reordenan las operaciones para separar las cargas de memoria de sus usos aritméticos, maximizando así el paralelismo a nivel de instrucción.

¿Es siempre posible eliminar todas las paradas (stalls)?

No es posible eliminar todas las paradas. Aunque técnicas como el reordenamiento de instrucciones y el *Forwarding* mitigan los riesgos, siempre existirán factores como los fallos de caché (cache misses), los errores en la predicción de saltos (branch mispredictions) y las instrucciones de alta latencia (como la división) que forzarán al pipeline a insertar ciclos de espera para garantizar la integridad de los datos.