Practica 1.1

Ejercicio 1. Configura el simulador para que el forwarding esté desactivado y vuelve a ejecutar el programa del ejemplo.

```
.text
addi x12, x0, 2
addi x10, gp, 8
loop:
beq x10, gp, end
lw x5, 100(x10)
add x5, x5, x12
sw x5, 200(x10)
j loop
addi x10, x10, -4
end:
addi x0, x0, 0
```

1.1. Utilizando el diagrama de la tabla de ejecución, haz una comparación entre la ejecución con forwarding y sin él. Explica las diferencias que encuentres.

Con forwarding

EXECUTION TABLE

FULL LOOPS

Instruction	CPU Cycles																								
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
addi a2, x0, 2	F	D	X	M	W																					
addi a0, gp, 8		F	D	X	M	W																				
beq a0, gp, 48			F	-	D	X	M	W																		
lw t0, 100(a0)					F	D	X	M	W																	
add t0, t0, a2						F	-	D	X	M	W															
sw t0, 200(a0)								F	D	X	M	W														
jal x0, -32									F	D	X	M	W													
addi a0, a0, -4										F	D	X	M	W												
beq a0, gp, 48											F	-	D	X	M	W										
lw t0, 100(a0)													F	D	X	M	W									
add t0, t0, a2														F	-	D	X	M	W							
sw t0, 200(a0)																F	D	X	M	W						
jal x0, -32																	F	D	X	M	W					
addi a0, a0, -4																		F	D	X	M	W				
beq a0, gp, 48																			F	-	D	X	M	W		
lw t0, 100(a0)																					F	D	X	M	W	
addi x0, x0, 0																						F	D	X	M	W

Sin forwarding

EXECUTION TABLE

FULL LOOPS

Instruction	CPU Cycles																															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
addi a2, x0, 2	F	D	X	M	W																														
addi a0, gp, 8		F	D	X	M	W																													
beq a0, gp, 48			F	-	-	D	X	M	W																										
lw t0, 100(a0)						F	D	X	M	W																									
add t0, t0, a2							F	-	-	D	X	M	W																						
sw t0, 200(a0)										F	-	-	D	X	M	W																			
jal x0, -32													F	D	X	M	W																		
addi a0, a0, -4														F	D	X	M	W																	
beq a0, gp, 48															F	-	-	D	X	M	W														
lw t0, 100(a0)																		F	D	X	M	W													
add t0, t0, a2																			F	-	-	D	X	M	W										
sw t0, 200(a0)																						F	-	-	D	X	M	W							
jal x0, -32																									F	D	X	M	W						
addi a0, a0, -4																										F	D	X	M	W					
beq a0, gp, 48																											F	-	-	D	X	M	W		
lw t0, 100(a0)																														F	D	X	M	W	
addi x0, x0, 0																															F	D	X	M	W

Las principales diferencias entre utilizar forwarding y no utilizarlo son las siguientes:

- **Con forwarding:** se puede observar que la ejecución del programa se realiza con mayor rapidez, ya que se evita el retraso que se produce al tener que

esperar a que la información se traslade de un registro a otro.

- **Sin forwarding:** la ejecución del programa es más lenta, ya que se produce un retraso al tener que esperar a que la información se traslade de un registro a otro.

Calcula el CPI para cada uno de los casos y calcula la mejora mediante la ley de Amdahl.

CPI con forwarding: 26/8 = 3.25

CPI sin forwarding: 35/8 = 4.37

Ejercicio 2. Vuelve a configurar el simulador para que no soporte delay slot, es decir que flusheé las instrucciones cuando haya un salto. Carga el programa del ejemplo de nuevo e intenta ejecutarlo durante unas cuantas iteraciones del bucle y mira qué ocurre.

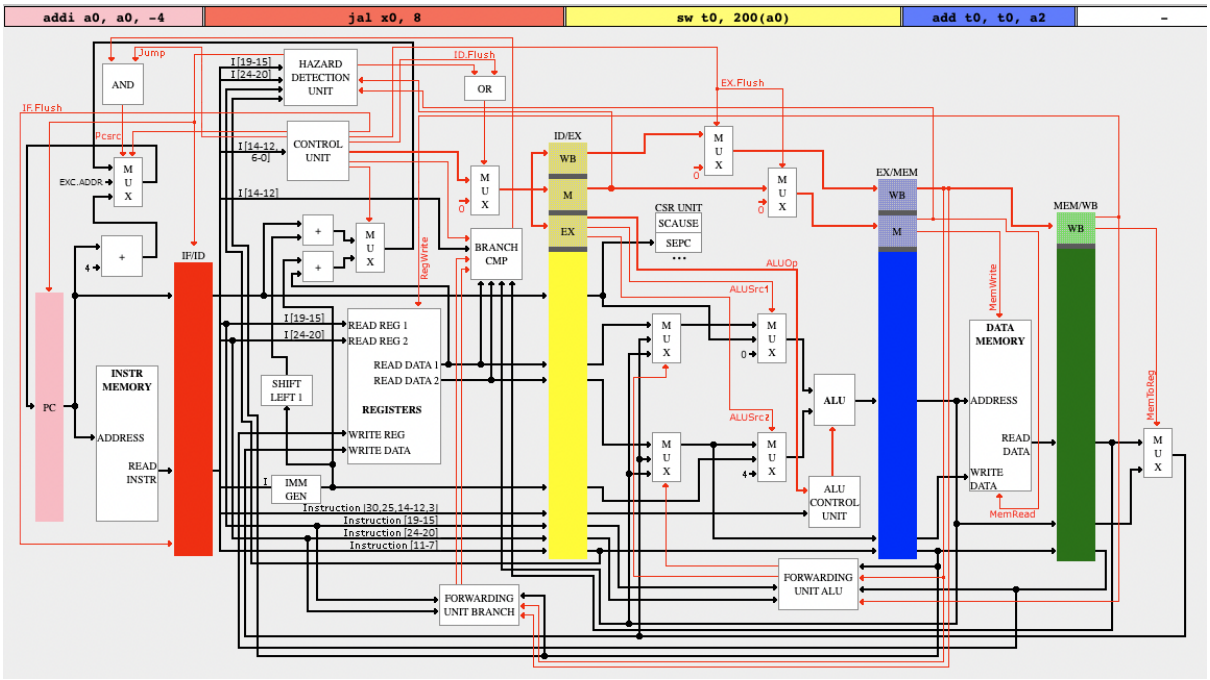
2.1. Explica conceptualmente qué está pasando para que el comportamiento del programa se haya modificado y sea “defectuoso”.

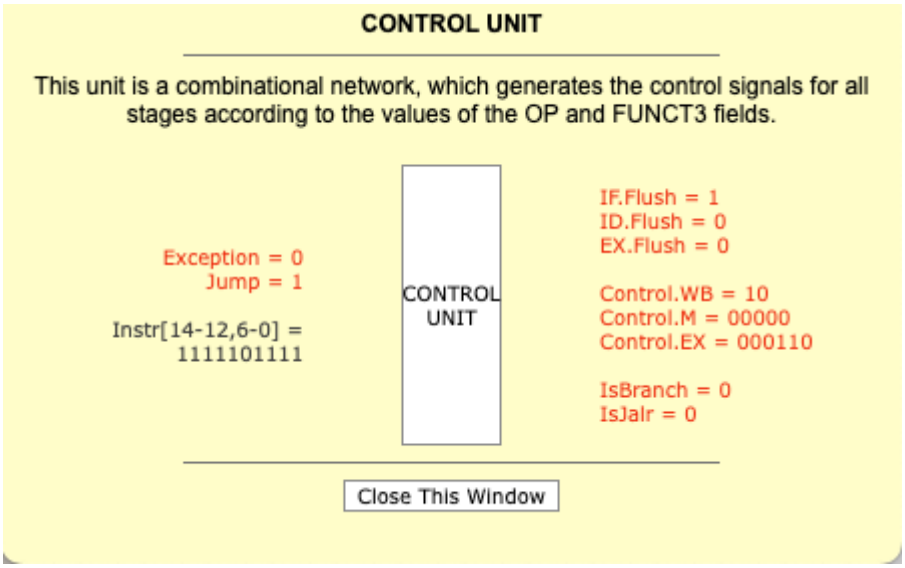
La estrategia de "Execute Delay Slot" asume que la instrucción que sigue al salto se ejecutará independientemente de si se toma el salto o no. Esto es beneficioso en escenarios donde la ubicación del salto es desconocida hasta el último momento. Sin embargo, si la ubicación del salto es conocida antes, este enfoque puede llevar a la ejecución de instrucciones innecesarias.

En contraste, la estrategia de "Flush Instruction" descarta todas las instrucciones en la tubería después de la instrucción de salto y las reemplaza por otras instrucciones una vez que se sabe si se toma el salto o no. Esto puede ser beneficioso si es probable que se tome el salto, pero puede llevar a un rendimiento más lento si el salto no se toma, ya que se desperdician ciclos de reloj limpiando y rellenando la tubería.

La estrategia de "Execute Delay Slot" puede estar permitiendo que el código se ejecute a pesar del bucle infinito debido a la forma en que maneja las instrucciones después de los saltos. Sin embargo, la estrategia de "Flush Instruction" puede estar causando un error debido a la forma en que intenta limpiar las instrucciones después del salto en el bucle infinito.

2.2 Explica el significado de aquellas salidas más adecuadas que está generando la unidad de control durante el ciclo 10. ¿Qué tipo de riesgo de cauce está ocurriendo en ese momento?





1. **IF.Flush = 1**: Esto indica que la instrucción que está en la etapa de Fetch (IF) debe ser descartada o "flushed". Es probable que esto se deba a un salto condicional o incondicional, es decir, un riesgo de control.
2. **ID.Flush = 0** y **EX.Flush = 0**: Esto indica que las instrucciones en la etapa de Decodificación (ID) y la etapa de Ejecución (EX) no se descartan.

3. **Control.WB = 10**: Esto es una señal de control para la etapa de Write Back (WB). La representación de estos bits puede variar dependiendo de la implementación de la CPU, pero típicamente indican operaciones como escribir en un registro o la memoria.
4. **Control.M = 00000**: Esta es una señal de control para la etapa de Memoria (M). De nuevo, la interpretación exacta de estos bits puede variar, pero típicamente pueden indicar operaciones como cargar desde la memoria, almacenar en la memoria, o ninguna operación de memoria.
5. **Control.EX = 000110**: Esta es una señal de control para la etapa de Ejecución (EX). Estos bits pueden indicar operaciones como suma, resta, operaciones lógicas, etc.
6. **IsBranch = 0**: Esto indica que la instrucción actual no es una instrucción de salto condicional.
7. **IsJalr = 0**: Esto indica que la instrucción actual no es una instrucción de salto y enlace (JALR).

Entonces, en el ciclo 10, parece que hay un riesgo de control. La instrucción en la etapa de Fetch (IF) está siendo descartada, probablemente debido a que se ha tomado un salto o una rama en una instrucción anterior. Sin embargo, las instrucciones en las etapas de Decodificación (ID) y Ejecución (EX) no se descartan, lo que indica que la CPU ha decidido que estas instrucciones pueden continuar ejecutándose a pesar del salto o la rama.

Ejercicio 3. Configura el simulador para que el forwarding esté desactivado y carga el programa de ejemplo denominado “Data Hazard Example”.

3.1. Realiza una tabla con un informe de ciclo a ciclo los hazards detectados por el simulador para cada una de las instrucciones. Menciona los componentes que has utilizado para comprobar esto.

EXECUTION TABLE

FULL LOOPS

Instruction	CPU Cycles												
	1	2	3	4	5	6	7	8	9	10	11	12	13
addi t0, x0, 1	F	D	X	M	W								
addi t1, x0, 2		F	D	X	M	W							
add t2, t0, t1			F	-	-	D	X	M	W				
add t3, t0, t2						F	-	-	D	X	M	W	
add t4, t0, t2									F	D	X	M	W

En este caso, parece que nos estamos refiriendo a hazards de datos. Hay dos tipos de hazards de datos: hazards de lectura después de escritura (RAW), también conocidos como dependencias verdaderas, y hazards de escritura después de escritura (WAW) o escritura después de lectura (WAR), también conocidos como dependencias de salida y antidependencias, respectivamente.

Asumiendo que cada instrucción toma un ciclo para completarse, podríamos tener un informe de hazard como este:

Ciclo	Instrucción	Hazard	Descripción
1	addi t0, x0, 1	Ninguno	Ninguno
2	addi t1, x0, 2	Ninguno	Ninguno
3	add t2, t0, t1	RAW	Hazard M=>X, t0 y t1 todavía no se han escrito cuando se leen
4	add t3, t0, t2	RAW	Hazard X=>X, t2 todavía no se ha escrito cuando se lee
5	add t4, t0, t2	RAW	Hazard M=>X, t2 todavía no se ha escrito cuando se lee

En una arquitectura de pipeline, estos hazards de datos se pueden mitigar a través de varias técnicas, como el reordenamiento de instrucciones, el estancamiento del pipeline, el reenvío de datos (también conocido como bypassing) y la predicción de ramificación.

3.2 Durante los ciclos 4 y 5, ¿qué componente está parando la ejecución del pipeline?

Durante los ciclos 4 y 5, la instrucción "add t2, t0, t1" está en las etapas de Fetch (IF) y Decode (ID), respectivamente. Sin embargo, los datos necesarios para esta instrucción (los valores de t0 y t1) todavía no están disponibles porque las

instrucciones que los generan ("addi t0, x0, 1" y "addi t1, x0, 2") todavía no han completado sus etapas de Write Back (WB).

Esto crea un hazard de lectura después de escritura (RAW), también conocido como hazard de dependencia de datos verdadera. En un pipeline sin mecanismos adicionales para manejar estos hazards, el pipeline tendría que estancarse hasta que los datos necesarios estén disponibles.

El componente que estaría deteniendo la ejecución del pipeline en este caso sería el controlador del pipeline. Detectaría el hazard y emitiría una señal de estancamiento, que detendría la ejecución de las siguientes instrucciones hasta que se resuelva el hazard. Dependiendo de la arquitectura exacta del procesador, este proceso puede ser más o menos sofisticado, pero en general, el controlador del pipeline es el componente que maneja estos casos.

3.3 En los ciclos 6 y 7 de dónde coge la instrucción “add t2, t0, t1” los datos para poder operar en la ALU.

Considerando la instrucción "add t2, t0, t1", en el ciclo 6 y 7, los datos para poder operar en la ALU (Unidad Aritmético Lógica) se obtendrían de la siguiente manera:

- Ciclo 6 (EX): En este ciclo, la instrucción "add t2, t0, t1" se encuentra en la etapa de ejecución. Aquí es donde la ALU realiza la suma de los valores de t0 y t1. Los datos de t0 y t1 se obtendrían de los registros correspondientes. Estos datos estarían disponibles en este punto porque las instrucciones "addi t0, x0, 1" y "addi t1, x0, 2" ya habrían pasado la etapa WB en los ciclos anteriores, lo que significa que los valores de t0 y t1 ya habrían sido actualizados en el banco de registros.
- Ciclo 7 (MEM): Aunque esta instrucción no necesita realizar ninguna operación de memoria (como la carga o almacenamiento de datos), las arquitecturas pipelined generalmente aún pasarán por esta etapa, aunque no se realiza ninguna operación. El resultado de la operación de la ALU en la etapa anterior (EX) se pasa a esta etapa.

Ejercicio 4. Ahora configura el simulador para que el forwarding esté activado y vuelve a cargar el programa “Data Hazard Example”.

4.1. ¿Qué diferencias hay en el esquemático del pipeline, en la etapa de ejecución, entre la microarquitectura con forwarding y la que no lo tiene?

EXECUTION TABLE

FULL LOOPS	CPU Cycles								
Instruction	1	2	3	4	5	6	7	8	9
addi t0, x0, 1	F	D	X	M	W				
addi t1, x0, 2		F	D	X	M	W			
add t2, t0, t1			F	D	X	M	W		
add t3, t0, t2				F	D	X	M	W	
add t4, t0, t2					F	D	X	M	W

Sin forwarding:

En la etapa de ejecución, la ALU toma dos operandos de la etapa de decodificación, realiza la operación requerida y luego pasa el resultado a la siguiente etapa del pipeline. No hay caminos de datos directos desde la etapa de ejecución a ninguna otra etapa del pipeline.

Con forwarding:

En la etapa de ejecución, además de la funcionalidad básica, también hay caminos de datos adicionales que permiten que los resultados de las operaciones se envíen directamente a las etapas anteriores del pipeline. Por ejemplo, podría haber un camino de datos desde la salida de la ALU a su entrada, lo que permitiría que los resultados de la ALU se reutilicen inmediatamente como operandos para la siguiente instrucción sin tener que pasar por el banco de registros. También podría haber un camino de datos desde la salida de la ALU a la entrada de la etapa de decodificación, lo que permitiría que los resultados de la ALU se utilicen como operandos para las instrucciones que están actualmente en la etapa de decodificación.

4.2 De la misma manera, realiza una tabla con un informe ciclo a ciclo de los hazards detectados por el simulador para cada una de las instrucciones. Menciona los componentes que has utilizado para comprobar esto. Explica por qué se generan las diferencias entre el ejercicio 3 y el 4.

Dado que el forwarding puede resolver los hazards de lectura después de escritura (RAW), no deberíamos ver esos hazards en esta secuencia de instrucciones. Aquí está la tabla de ciclo a ciclo con los hazards que podrían ser detectados por un simulador:

Ciclo	Instrucción	Hazard	Descripción
1	addi t0, x0, 1	Ninguno	Ninguno
2	addi t1, x0, 2	Ninguno	Ninguno
3	add t2, t0, t1	Ninguno	El forwarding puede resolver cualquier hazard RAW
4	add t3, t0, t2	Ninguno	El forwarding puede resolver cualquier hazard RAW
5	add t4, t0, t2	Ninguno	El forwarding puede resolver cualquier hazard RAW

El forwarding permite que los resultados de las instrucciones se envíen directamente a las etapas anteriores del pipeline. En este caso, los resultados de las instrucciones "addi t0, x0, 1" y "addi t1, x0, 2" pueden ser enviados directamente a la ALU para la instrucción "add t2, t0, t1" en el ciclo 3. De manera similar, el resultado de la instrucción "add t2, t0, t1" puede ser enviado directamente a la ALU para las instrucciones "add t3, t0, t2" y "add t4, t0, t2" en los ciclos 4 y 5, respectivamente.

4.3 En el ciclo 5, ¿de dónde coge los datos la instrucción “add t2, t0, t1” para poder operar en la ALU? ¿Qué componente indica de dónde son obtenidos dichos datos?

En el ciclo 5, la instrucción "add t2, t0, t1" ya ha pasado por las etapas de ejecución y memoria en los ciclos 3 y 4, respectivamente, en una arquitectura pipelined estándar. Por lo tanto, en el ciclo 5, esta instrucción estaría en la etapa de Write Back (WB), donde el resultado de la operación de suma se escribe de vuelta en el registro t2. Los datos que se escribirían en este punto serían el resultado de la suma de t0 y t1 que se calculó en la etapa de ejecución en el ciclo 3.

4.4. En los componentes del pipeline denominados “EXECUTE MULTIPLEXER 3” y “EXECUTE MULTIPLEXER 4”, ¿qué significan las cada una de sus posibles señales de control?

El "Execute Multiplexer 3" en la etapa de ejecución de un pipeline es un componente clave en la implementación del forwarding (o bypassing) en

arquitecturas de pipeline. Este multiplexor selecciona entre varias posibles fuentes de operandos para la ALU, basándose en las señales de control que recibe. El propósito de este multiplexor es permitir que los resultados de las instrucciones se reutilicen como operandos para las instrucciones subsiguientes sin tener que pasar por el banco de registros, lo que puede ayudar a minimizar los hazards de datos.

1. **ID/EX.ReadData1 = 0**: Esto parece ser el primer operando que ha sido leído del banco de registros durante la etapa de decodificación y que se ha pasado a la etapa de ejecución. Si la señal de "Control" selecciona esta entrada, entonces este sería el operando que se usaría para la operación de la ALU.
2. **WB.Data = 0**: Es el resultado de una instrucción que ha pasado por la etapa de Write Back. Si la señal de "Control" selecciona esta entrada, entonces este resultado se reutilizaría como un operando para la operación de la ALU sin tener que pasar por el banco de registros.
3. **EX/MEM.Data = 0**: Es el resultado de una instrucción que ha pasado por la etapa de ejecución y que se ha pasado a la etapa de memoria. Si la señal de "Control" selecciona esta entrada, entonces este resultado se reutilizaría como un operando para la operación de la ALU sin tener que pasar por el banco de registros.
4. **Temp ALU Operand 1 = 0**: Esta es una entrada temporal que almacena un operando para la ALU que aún no ha sido pasado a la siguiente etapa del pipeline.
5. **Control = 00**: Esta es la señal de control que determina cuál de las entradas anteriores se selecciona. El valor exacto de esta señal dependería de la implementación específica del pipeline. En general, diferentes valores de esta señal seleccionarían diferentes entradas. Por ejemplo, un valor de "00" podría seleccionar la entrada "ID/EX.ReadData1", mientras que un valor de "01" podría seleccionar la entrada "WB.Data", y así sucesivamente.

El "Execute Multiplexer 4" en la etapa de ejecución de un pipeline sirve un propósito similar al "Execute Multiplexer 3", pero para el segundo operando de la ALU en lugar del primero. Este multiplexor selecciona entre varias posibles fuentes de operandos para la ALU, basándose en las señales de control que recibe. Al igual que el "Execute Multiplexer 3", este multiplexor es un componente clave en la implementación del forwarding (o bypassing) en arquitecturas de pipeline.

Las señales que mencionas parecen ser las posibles entradas a este multiplexor, y la señal de "Control" determinaría cuál de estas entradas se selecciona. Aquí está lo que podrían significar cada una de estas señales:

1. **ID/EX.ReadData2 = 0**: Segundo operando que ha sido leído del banco de registros durante la etapa de decodificación y que se ha pasado a la etapa de ejecución. Si la señal de "Control" selecciona esta entrada, entonces este sería el operando que se usaría para la operación de la ALU.
2. **WB.Data = 0**: Instrucción que ha pasado por la etapa de Write Back. Si la señal de "Control" selecciona esta entrada, entonces este resultado se reutilizaría como un operando para la operación de la ALU sin tener que pasar por el banco de registros.
3. **EX/MEM.Data = 0**: Resultado de una instrucción que ha pasado por la etapa de ejecución y que se ha pasado a la etapa de memoria. Si la señal de "Control" selecciona esta entrada, entonces este resultado se reutilizaría como un operando para la operación de la ALU sin tener que pasar por el banco de registros.
4. **Temp ALU Operand 2 = 0**: Entrada temporal que almacena un operando para la ALU que aún no ha sido pasado a la siguiente etapa del pipeline.
5. **Control = 00**: Señal de control que determina cuál de las entradas anteriores se selecciona. El valor exacto de esta señal dependería de la implementación específica del pipeline. En general, diferentes valores de esta señal seleccionarían diferentes entradas. Por ejemplo, un valor de "00" podría seleccionar la entrada "ID/EX.ReadData2", mientras que un valor de "01" podría seleccionar la entrada "WB.Data", y así sucesivamente.

Ejercicio 5. Carga el programa de ejemplo denominado “Stall Example”. Con el forwarding activado realiza un seguimiento de este. ¿Por qué pese a tener dicha característica activada se produce un stall?

La razón es que la instrucción de carga (`lw`) tiene una latencia de al menos un ciclo extra en comparación con las operaciones ALU como `add`. Esto se debe a que `lw` necesita acceder a la memoria, lo cual es más lento que trabajar solo con los registros. En otras palabras, el resultado de `lw` no está inmediatamente disponible para la siguiente instrucción `add` que está tratando de usarlo, incluso con el forwarding activado.

El forwarding puede ayudar a resolver las dependencias de los datos entre las instrucciones que sólo implican operaciones de ALU, porque estas instrucciones tienen latencias de un ciclo. Sin embargo, no puede eliminar completamente los stalls causados por las instrucciones de carga debido a su latencia adicional.

Por lo tanto, en este caso, el procesador tendría que esperar ("stall") hasta que el resultado de `lw` esté disponible antes de que pueda ejecutar la siguiente instrucción `add`.

Ejercicio 6. Con el forwarding y el flush de instrucciones activados, carga los siguientes programas en el editor y responde a las preguntas.

```
add x20, x0, gp
addi x20, x20, 16
addi x21, x0, 99
add x22, x0, gp
Loop:
lw x31,0(x20) # x31=array element
add x31, x31, x21# add scalar in x21
sw x31,0(x20) # store result
addi x20,x20,-4 # decrement pointer
blt x22,x20,Loop # branch if x22 < x20
```

```
add x20, x0, gp
addi x20, x20, 16
addi x21, x0, 99
add x22, x0, gp
Loop:
lw x31,0(x20) # x31=array element
lw x30,-4(x20) # x30=array element
lw x29,-8(x20) # x29=array element
lw x28,-12(x20) # x28=array element
add x31, x31, x21# add scalar in x21
add x30, x30, x21
add x29, x29, x21
add x28, x28, x21
sw x31,0(x20) # store result
sw x30,-4(x20)
sw x29,-8(x20)
sw x28,-12(x20)
addi x20,x20,-16 # decrement pointer
blt x22,x20,Loop # branch if x22 < x20
```

6.1. ¿Qué realiza cada programa? Inserta una captura de pantalla que lo demuestre. Explica las diferencias entre cada uno de ellos

P1

1. `add x20, x0, gp`: Esta línea inicializa el registro x20 con el valor del puntero global `gp` (global pointer). Esto podría indicar que el array se encuentra en una posición relativa al puntero global.
2. `addi x20, x20, 16`: Añade 16 al registro x20. Esto podría indicar que el array comienza 16 bytes después de la posición apuntada por `gp`.
3. `addi x21, x0, 99`: Esto inicializa el registro x21 con el valor 99. Este es el escalar que se sumará a cada elemento del array.
4. `add x22, x0, gp`: Esto inicializa el registro x22 con el valor del puntero global `gp`. x22 podría servir como un límite inferior para el array.
5. `Loop:`: Este es el inicio de un bucle que se ejecutará hasta que se cumpla la condición del salto condicional `blt`.

Dentro del bucle:

- `lw x31,0(x20)`: Esto carga el valor del array en la posición apuntada por x20 en el registro x31.
- `add x31, x31, x21`: Esto añade el valor en x21 (99) al valor del elemento del array cargado en x31.
- `sw x31,0(x20)`: Esto almacena el resultado de la suma en la misma posición del array.
- `addi x20,x20,-4`: Esto disminuye el valor de x20 en 4 (dado que estamos trabajando con un array de enteros y en RISC-V un entero ocupa 4 bytes). Por lo tanto, esto mueve el puntero x20 al siguiente elemento del array (recuerda que estamos decrementando ya que el índice más alto del array está en la parte superior de la memoria).
- `blt x22,x20,Loop`: Este es un salto condicional que vuelve al inicio del bucle si x22 es menor que x20. En otras palabras, el bucle seguirá ejecutándose hasta que x20 (que apunta a los elementos del array) sea menor que x22 (que podría ser la base del array).

Este programa parece sumar 99 a cada elemento de un array en orden descendente en la memoria, comenzando desde `gp + 16` hasta `gp`.

P2

- `lw x31,0(x20)`: Esto carga el valor del array en la posición apuntada por x20 en el registro x31.
- `lw x30,-4(x20)`: Esto carga el valor del array en la posición apuntada por x20 menos 4 bytes en el registro x30.
- `lw x29,-8(x20)`: Esto carga el valor del array en la posición apuntada por x20 menos 8 bytes en el registro x29.
- `lw x28,-12(x20)`: Esto carga el valor del array en la posición apuntada por x20 menos 12 bytes en el registro x28.
- `add x31, x31, x21`: Esto añade el valor en x21 (99) al valor del elemento del array cargado en x31.
- `add x30, x30, x21`: Esto añade el valor en x21 (99) al valor del elemento del array cargado en x30.
- `add x29, x29, x21`: Esto añade el valor en x21 (99) al valor del elemento del array cargado en x29.
- `add x28, x28, x21`: Esto añade el valor en x21 (99) al valor del elemento del array cargado en x28.
- `sw x31,0(x20)`: Esto almacena el resultado de la suma en la misma posición del array.
- `sw x30,-4(x20)`: Esto almacena el resultado de la suma en la posición del array que está 4 bytes antes.
- `sw x29,-8(x20)`: Esto almacena el resultado de la suma en la posición del array que está 8 bytes antes.
- `sw x28,-12(x20)`: Esto almacena el resultado de la suma en la posición del array que está 12 bytes antes.
- `addi x20,x20,-16`: Esto disminuye el valor de x20 en 16 (dado que estamos trabajando con un array de enteros y en RISC-V un entero ocupa 4 bytes, y estamos procesando 4 elementos a la vez). Por lo tanto, esto mueve el puntero x20 a los siguientes 4 elementos del array.
- `blt x22,x20,Loop`: Este es un salto condicional que vuelve al inicio del bucle si x22 es menor que x20. En otras palabras, el bucle seguirá ejecutándose hasta que x20 (que apunta a los elementos del array) sea menor que x22 (que podría ser la base del array).

Este nuevo programa es similar al anterior, pero realiza la operación de suma escalar en cuatro elementos del array al mismo tiempo en cada iteración del bucle, en lugar de solo uno. Este tipo de optimización se llama "unrolling the loop" o desenrollado del bucle, y se utiliza a menudo para mejorar el rendimiento de los programas.

El desenrollado del bucle puede reducir la sobrecarga de la gestión del bucle (inicialización, prueba de finalización, etc.) y puede permitir una mayor paralelización y uso de la caché.

Este segundo programa hace lo mismo que el primero, pero procesa 4 elementos del array a la vez en cada iteración del bucle.

6.2. Mide el CPI de cada programa y calcula la mejora tras el desenrollado por renombramiento de registros. Explica por qué se produce dicha mejora.

EXECUTION TABLE

FULL LOOPS

Instruction	CPU Cycles																																				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
add s4, x0, gp	F	D	X	M	W																																		
addi s4, s4, 16		F	D	X	M	W																																	
addi s5, x0, 99			F	D	X	M	W																																
add s6, x0, gp				F	D	X	M	W																															
lw t6, 0(s4)					F	D	X	M	W																														
add t6, t6, s5						F	-	D	X	M	W																												
sw t6, 0(s4)								F	D	X	M	W																										
addi s4, s4, -4									F	D	X	M	W																									
blt s6, s4, -32										F	-	D	X	M	W																							
lw t6, 0(s4)													F	D	X	M	W																					
add t6, t6, s5														F	-	D	X	M	W																		
sw t6, 0(s4)																F	D	X	M	W																	
addi s4, s4, -4																	F	D	X	M	W																
blt s6, s4, -32																		F	-	D	X	M	W														
lw t6, 0(s4)																					F	D	X	M	W												
add t6, t6, s5																						F	-	D	X	M	W										
sw t6, 0(s4)																								F	D	X	M	W									
addi s4, s4, -4																									F	D	X	M	W								
blt s6, s4, -32																										F	-	D	X	M	W						
lw t6, 0(s4)																													F	D	X	M	W				
add t6, t6, s5																														F	-	D	X	M	W		
sw t6, 0(s4)																																F	D	X	M	W	
addi s4, s4, -4																																	F	D	X	M	W	
blt s6, s4, -32																																		F	-	D	X	M	W

P1

EXECUTION TABLE

FULL LOOPS

Instruction	CPU Cycles																					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
add s4, x0, gp	F	D	X	M	W																		
addi s4, s4, 16		F	D	X	M	W																	
addi s5, x0, 99			F	D	X	M	W																
add s6, x0, gp				F	D	X	M	W															
lw t6, 0(s4)					F	D	X	M	W														
lw t5, -4(s4)						F	D	X	M	W													
lw t4, -8(s4)							F	D	X	M	W												
lw t3, -12(s4)								F	D	X	M	W											
add t6, t6, s5									F	D	X	M	W										
add t5, t5, s5										F	D	X	M	W									
add t4, t4, s5											F	D	X	M	W								
add t3, t3, s5												F	D	X	M	W							
sw t6, 0(s4)													F	D	X	M	W						
sw t5, -4(s4)														F	D	X	M	W					
sw t4, -8(s4)															F	D	X	M	W				
sw t3, -12(s4)																F	D	X	M	W			
addi s4, s4, -16																	F	D	X	M	W		
blt s6, s4, -104																		F	-	D	X	M	W

P2

1. En el primer programa, cada iteración del bucle requiere 5 instrucciones (4 operaciones y una de control para el bucle). Si suponemos que no hay peligros de datos (data hazards) que puedan causar stalls (esperas), y que el branch prediction es perfecto (no hay penalización por el salto), podríamos decir que el CPI es 1, ya que cada instrucción puede ejecutarse en un solo ciclo.
2. En el segundo programa, cada iteración del bucle requiere 13 instrucciones (12 operaciones y una de control para el bucle). Sin embargo, estas instrucciones están procesando 4 elementos del array a la vez. Así que, en términos de instrucciones por elemento del array, tenemos 13/4 = 3.25 instrucciones por elemento. De nuevo, suponiendo que no hay data hazards y que el branch prediction es perfecto, podríamos decir que el CPI es también 1.

Por lo tanto, si bien el segundo programa tiene más instrucciones por iteración del bucle, también procesa más elementos del array por iteración. Así que, en términos de CPI, ambos programas podrían ser equivalentes, suponiendo las condiciones ideales mencionadas anteriormente.

El desenrollado del bucle a través del renombramiento de registros puede mejorar el rendimiento al reducir el número de veces que se ejecutan las instrucciones de

control del bucle y permitir un mayor paralelismo. Al procesar varios elementos del array a la vez, el procesador puede hacer un uso más eficiente de sus recursos, lo que puede llevar a un mejor rendimiento.