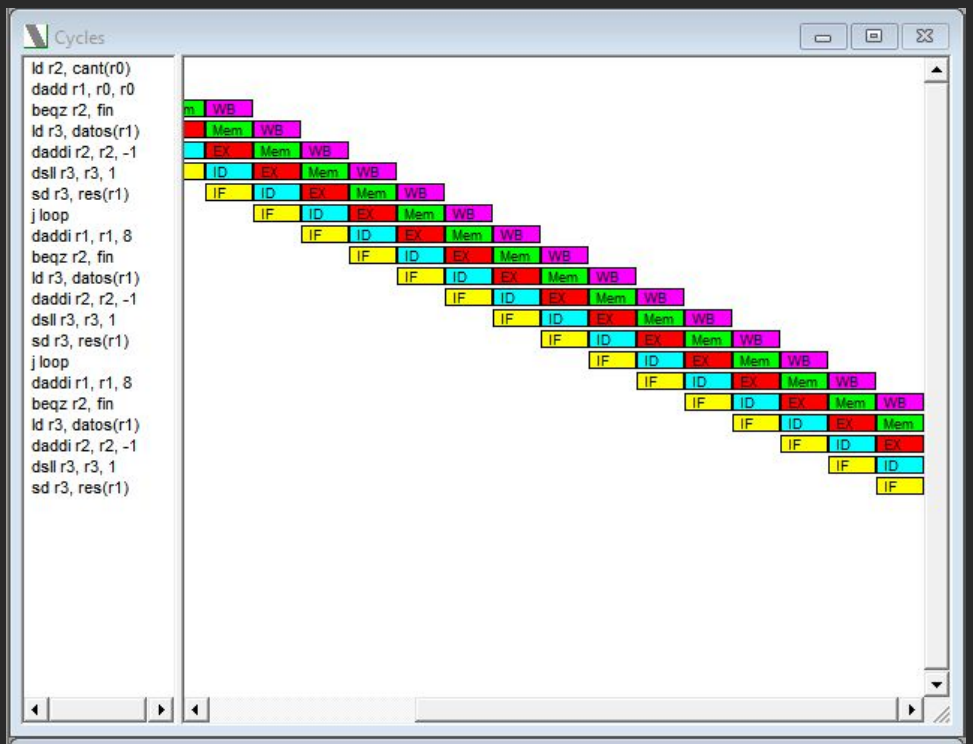
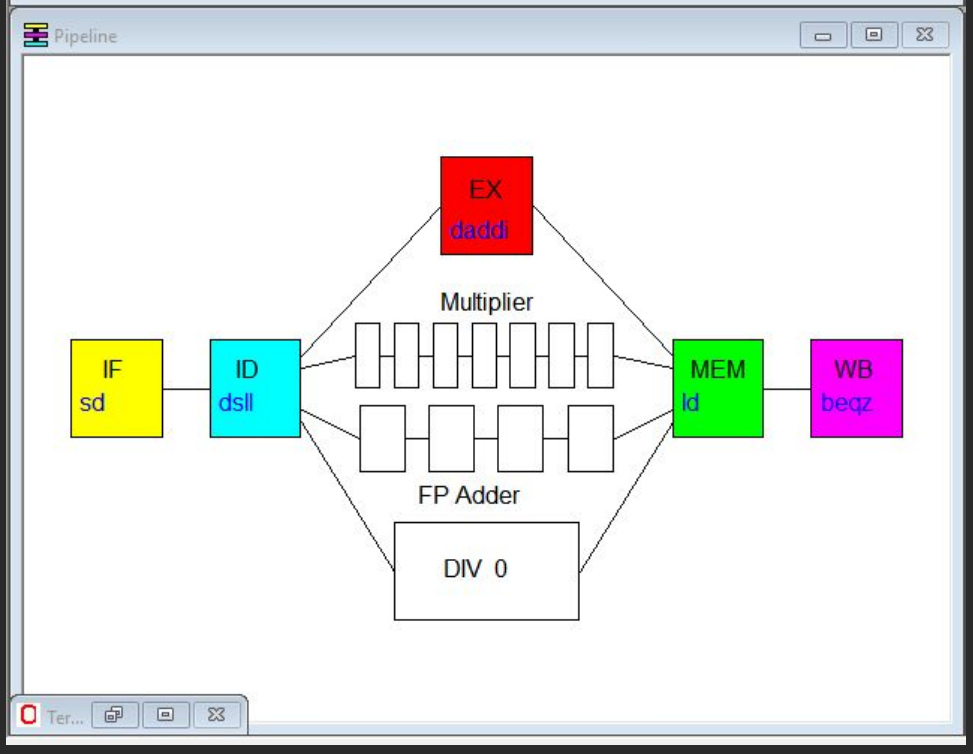
Fase 3

Rendimiento en arquitecturas RISC
Estudio de los riesgos de segmentación en MIPS64





- **Programa 1.**

Introduce el siguiente código en el simulador. Para ello genera un archivo de texto con este código y grábalo con la extensión .s, luego introdúcelo con el simulador a través del comando "Load" del menú "File":

```
.data

  num: .word 7
  num2: .word 8

.code

  ld r2, num(r0)
  dadd r3, r8, r9
  dsub r10, r5, r6
  dsll r1, r4, 1
  sd r4, num2(r0)

halt
```

a) Indica qué función realiza cada una de las instrucciones del código

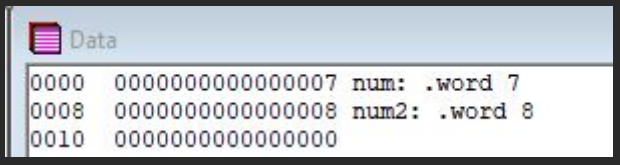
```
.code -> Comienzo del segmento de código.

 ld r2, num(r0) -> Copia en "r2" un double word (64bits) desde la dir "num + r0".

 dadd r3, r8, r9 -> Suma "r8" con "r9" y deja el valor en "r3" con signo.

 dsub r10, r5, r6 -> Resta "r6" a "r5" y deja el valor en "r10" con signo.

 dsll r1, r4, 1 -> Desplaza a la izq "1" veces los valores de los bits de "r4" y
deja el resultado en "r1".

 sd r4, num2(r0) -> Guarda el valor de "r4" en la posición de memoria "num + r0".

halt -> Detiene el simulador.
```

b) Indica qué variables de datos usa este programa y dónde se muestran en el simulador.

Las variables que se usan son las siguientes:

```
.data -> Comienzo del segmento de datos.

 num: .word 7 -> Reserva un espacio en memoria del tamaño de una palabra (64 bits)
y le asigna el valor "7".

 num2: .word 8 -> Reserva un espacio en memoria del tamaño de una palabra (64 bits)
y le asigna el valor "8".
```

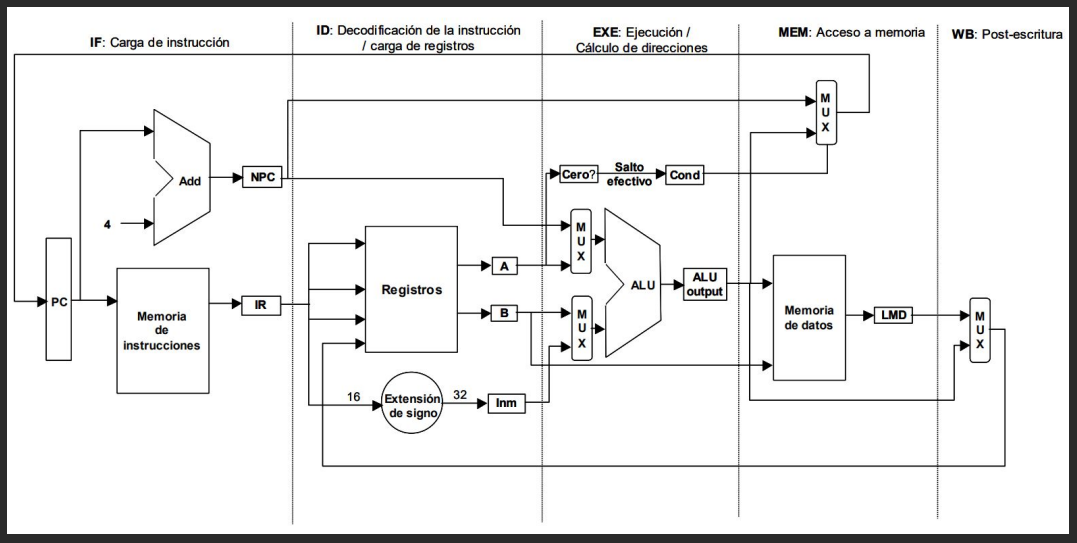
Las variables se muestran en el simulador en la ventana "Data":



c) Ejecutar el programa en el simulador con la opción **Configure / Enable Forwarding deshabilitada**. Analizar ciclo a ciclo su funcionamiento con la opción "**Single cycle**" del menú Execute o bien presionando F7 sucesivamente. Examinad las distintas ventanas que se muestran en el simulador y responder:

¿En qué ciclo del total del programa se lee el dato que hay en la variable num? ¿a qué fase corresponde de la instrucción load?

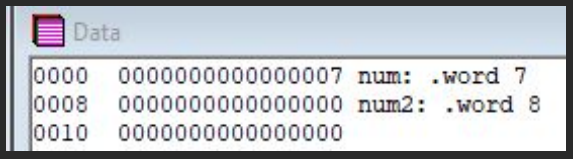
Ya que es la primera instrucción, será en el ciclo 4, en la fase MEM que es cuando se accede a la memoria.

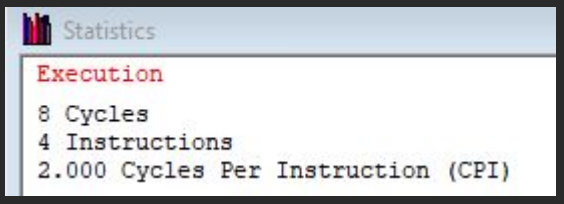


(Hay que mencionar que el simulador empieza desde el ciclo 0, por ello lo que se muestra en la ventana "Pipeline" no es lo que se está procesando en el momento, sino, lo que se va a procesar en el siguiente ciclo, y esto lo hace algo confuso (y más en el ejercicio 4 donde lidiamos con saltos) ya que la ventana pipeline muestra una fase "por delante")

¿En qué ciclo del total del programa se escribe el dato en la variable num2? ¿a qué fase corresponde de la instrucción store?

En el ciclo 8, en la fase MEM.
(num2 ahora es “0”)





¿En qué ciclo del total del programa se escribe un dato en el registro r10? ¿a qué fase corresponde de la instrucción de resta?

En el ciclo 7, en la fase WB se escribe en el registro R10.
La instrucción de la resta corresponde a la fase EX.

Tras la ejecución, ¿se produce alguna detención en el cauce? Razona tu respuesta.
¿Cuál es el promedio de ciclos por instrucción (CPI) en la ejecución de este programa?

No se realiza ninguna detención debido a que no hay ningún riesgo.
El promedio de ciclos es Ciclos / Instrucciones = 10 / 6 = 1.667.

- **Programa 2.**

Introduce el siguiente código en el simulador.

```
.data

  A: .word 8
  B: .word 6
  C: .word 3
  D: .word 0,0,0,0

.code

  ld r1, A(r0)
  ld r2, B(r0)
  ld r3, C(r0)

  xor r5, r5, r5

  dadd r6, r2, r3
  dadd r7, r6, r3
  dadd r8, r7, r2

  sd r6, D(r5)
  dadd dd r5, r5, r1
  sd r7, D(r5)
  dadd add r5, r5, r1
  sd r8, D(r5)
  dadd r9, r5, 8

  ld r10, D(r5)
  sd r10, D(r9)

halt
```

Elvi Mihai Sabau Sabau AC: Fase 3 51254875L

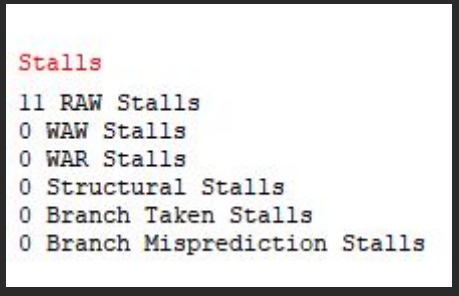
a) Comenta cada una de las líneas del código y explica brevemente qué realiza el código

```
.data ; Inicio de segmento de datos.

  A: .word 8 ; Reserva en memoria una variable (64 bits) y le asigna el valor 8.
  B: .word 6 ; Reserva en memoria una variable (64 bits) y le asigna el valor 6.
  C: .word 3 ; Reserva en memoria una variable (64 bits) y le asigna el valor 3.
  D: .word 0, 0, 0, 0 ; Reserva en memoria 4 variables (64 bits), y les asigna el valor 0 a cada una.

.code ; Inicio del segmento de código.

  ld r1, A(r0) ; Carga desde la dirección en memoria A + R0 = 8 + 0 (R0 = 0, no está seteado), y lo guarda en R1.
  ld r2, B(r0) ; Carga desde la dirección en memoria B + R0 = 6 + 0 (R0 = 0, no está seteado), y lo guarda en R2.
  ld r3, C(r0) ; Carga desde la dirección en memoria C + R0 = 8 + 0 (R0 = 0, no está seteado), y lo guarda en R3.

  xor r5, r5, r5 ; Hace xor de R5 con R5 y lo guarda en R5, R5 no estar seteado (por defecto valor 0), el xor devolverá
                 0.

  dadd r6, r2, r3 ; Suma R3 y R2 y lo guarda en R6.
  dadd r7, r6, r3 ; Suma R3 y R6 y lo guarda en R7.
  dadd r8, r7, r2 ; Suma R2 y R7 y lo guarda en R8.

  sd r6, D(r5) ; Guarda en la dir de memoria D + R5 el valor de R6.
  dadd r5, r5, r1 ; Suma R1 y R5 y lo guarda en  R5.
  sd r7, D(r5) ; Guarda en la dir de memoria D + R5 el valor de R7.
  dadd r5, r5, r1 ; Suma R1 y R5 y lo guarda en  R5.
  sd r8, D(r5) ; Guarda en la dir de memoria D + R5 el valor de R8.
  daddi r9, r5, 8 ; Suma 8 y R5 y lo guarda en R9.

  ld r10, D(r5) ; Carga el valor de la dir de memoria D + R5 y lo guarda en R10.
  sd r10, D(r9) ; Guarda el valor de la dir de memoria D + R9 y lo guarda en R10.

halt ; Detiene el programa.
```

Realiza varias operaciones con riesgo de dependencia entre sí. Seguramente para observar cómo se detiene el cauce y de qué manera prosigue las instrucciones en cada fase.

b) Ejecuta el programa en el simulador con la opción **Configure / Enable Forwarding deshabilitada**. Analizar paso a paso su funcionamiento, examinad las distintas ventanas que se muestran en el simulador y responde:

¿Cuántas detenciones RAW aparecen?



11, hay tantas no solo debido a la cantidad de instrucciones dependientes, sino, a que muchas de ellas están una detrás de la otra, lo que causa una doble detención del cauce por cada dependencia.
¿Qué instrucciones están generando las mismas (stalls) en el cauce?

Registro	Instrucción Origen	Instrucción Depend.	Tipo
R6	LD	DADD	RAW
R6	DADD	DADD	RAW
R5	DADD	DADD	RAW
R5	DADD	DSDD	RAW
R5	DADD	DADDI	RAW
R9	DADDI	SD	RAW

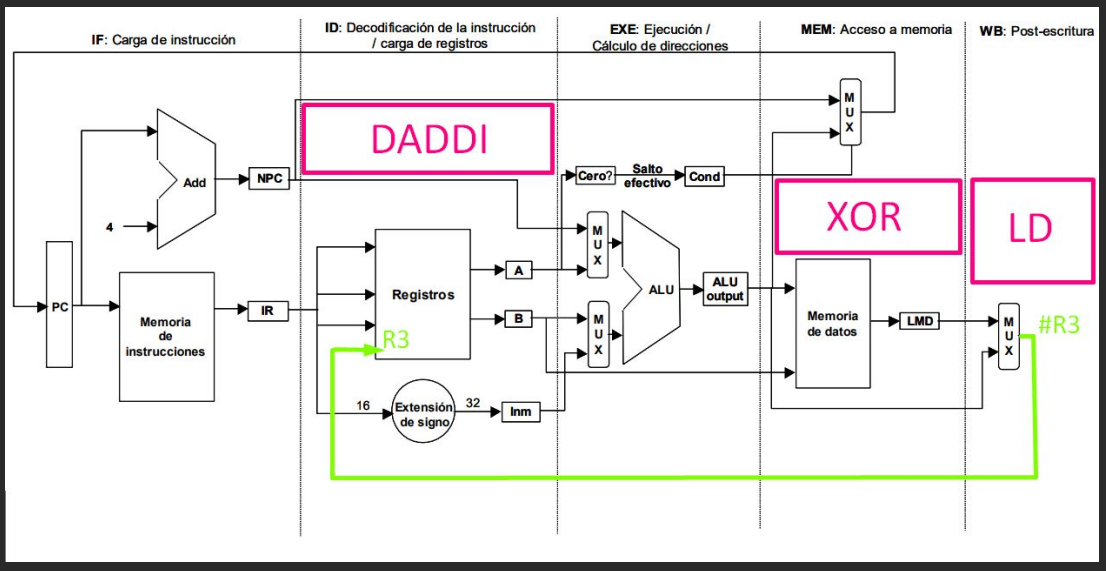
Explica por qué se producen las detecciones teniendo en cuenta las instrucciones y los registros implicados. Para ello, piensa en:

- ¿Por qué se produce la primera detención en el ciclo 6?. ¿Con qué registro e instrucción?

  Registro R3, con las instrucciones LD y DADD.

- ¿Es el mismo riesgo que se produce en la segunda detención?. ¿Por qué tenemos 2 detenciones y sólo 1 en la primera detención?

  Es el mismo tipo de riesgo, pero en el primero se detiene una sola vez porque hay una instruccion entre ambas (XOR), cosa que permite que la instrucción LD esté 2 fases por delante de la instrucción DADD, por ello, para que la instruccion LD acabe, solo se necesita 1 NOP, ya que cuando la instruccion DADD está en la fase IF, la instruccion LD estará en la fase EX, por ello, con un solo NOP, la instruccion LD estará en la fase MEM, y para cuando la instruccion DADD llegue a la fase ID, la instruccion LD estará en la fase WB, y DADD tendrá disponible el dato.



Siguiendo la lógica anterior, la segunda detención es de dos NOPs porque no hay ninguna instrucción por enmedio que se pueda ejecutar mientras la instrucción dependiente espera a la siguiente en acabar.

En resumen, en ambos casos, se necesitan 2 detenciones para que las instrucciones no sufran riesgo de dependencia. Es solo que en la primera dependencia, se “recicla” una etapa para que los recursos de esta etapa no estén sin hacer nada.

- ¿Son todos los mismos tipos de riesgos? Compara el primer riesgo producido con la última instrucción que se detiene (sd)

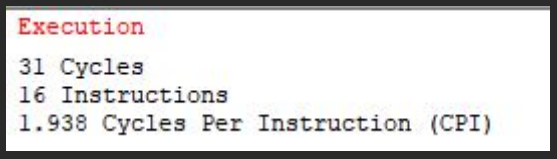
  Si, todos los riesgos son por dependencia de datos, aunque sean por diferentes instrucciones (sumas, guardado, carga, xor) sigue siendo debido a instrucciones no finalizadas y por ende a datos no escritos.

Registro	Instrucción Origen	Instrucción Depend.	Tipo
R3	LD	DADD	RAW
R9	DADDI	SD	RAW

Lo dicho, aunque sean diferentes instrucciones, ambos riesgos son por dependencia porque dependen de la escritura de una instrucción todavía no finalizada.

- ¿Cuál es el promedio de ciclos por instrucción (CPI) en la ejecución de este programa bajo esta configuración?

  CPI = Ciclos / Instrucciones = 31 / 16 = 1.938



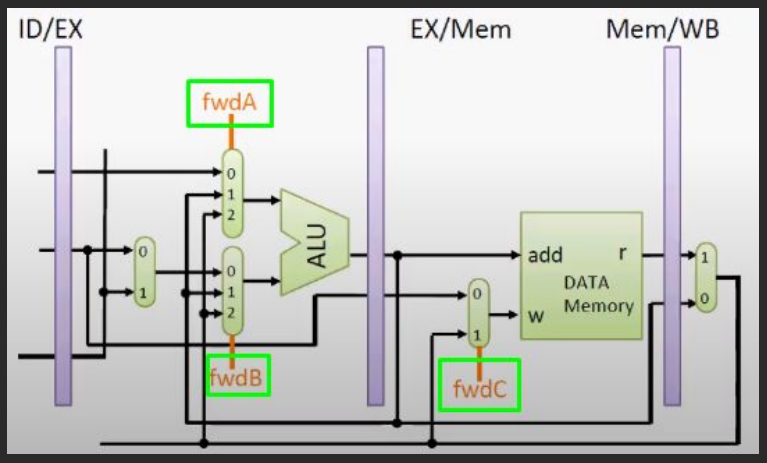
b) Una forma de solucionar las paradas por dependencia de datos es utilizar el adelantamiento de operandos o Forwarding. Ejecuta nuevamente el programa anterior con la opción **Enable Forwarding** habilitada y responde:

¿Por qué no se presenta ninguna parada en este caso? Explicar la mejora teniendo en cuenta:

- ¿Cómo se ha solucionado el primer riesgo? ¿Desde qué unidad funcional se ha adelantado el dato para resolver el riesgo que se producía en el ciclo 6?.

  Se ha solucionado gracias a que se ha adelantado el resultado de la fase MEM / WB a la entrada de EX.

  Esto se puede lograr añadiendo a la ruta de datos varios multiplexores que permiten conectar las salidas de los registros de segmentación (EX / MEM y MEM / WB) con las entradas de las fases EX y MEM.



- ¿Se resuelve de la misma forma la segunda detención anterior?. ¿Desde qué unidad se adelanta?

  Sí, se adelanta el resultado de la fase EX / MEM a la entrada de EX.

- Compara la solución del primer riesgo producido con la del último (sd)

  La solución es la misma, el adelantamiento de los datos requeridos.

¿Cuál es el promedio de ciclos por instrucción (CPI) en este caso? Comparar con el anterior.

Con forwarding:

```
Execution

20 Cycles
16 Instructions
1.250 Cycles Per Instruction (CPI)
```

Sin forwarding:

```
Execution

31 Cycles
16 Instructions
1.938 Cycles Per Instruction (CPI)
```

Forwarding es 1.550 veces más rápido que sin forwarding, lo que viene ser 33% mas rápido.

- **Programa 3.**

Introduce el siguiente código en el simulador. Para ello genera un archivo de texto con este código y grábalo con la extensión .s, luego introdúcelo con el simulador a través del comando "Load" del menú "File":

```
.data

  A: .word32 2
  B: .word32 3
  C: .word32 0
  D: .word32 4
  E: .word32 5
  F: .word32 0

.code

  lw r1, A(r0)
  lw r2, B(r0)
  dadd r3,r1,r2
  sw r3, C(r0)

  lw r4, D(r0)
  lw r5, E(r0)
  dadd r6,r4,r5
  sw r6, F(r0)

halt
```

a) Indica qué función realiza cada una de las instrucciones del código. ¿Podrías expresar el código mediante dos instrucciones de un lenguaje de alto nivel?

```
.data -> Inicio del segmento de datos.

  A: .word32 2 -> Reserva una var. en memoria de una palabra de 32bits con valor 2.
  B: .word32 3 -> Reserva una var. en memoria de una palabra de 32bits con valor 3.
  C: .word32 0 -> Reserva una var. en memoria de una palabra de 32bits con valor 0.
  D: .word32 4 -> Reserva una var. en memoria de una palabra de 32bits con valor 4.
  E: .word32 5 -> Reserva una var. en memoria de una palabra de 32bits con valor 5.
  F: .word32 0 -> Reserva una var. en memoria de una palabra de 32bits con valor 0.

.code -> Inicio del segmento de instrucciones.

  lw r1, A(r0) -> Carga en R1 el valor de la dir de memoria de "A + R0".
  lw r2, B(r0) -> Carga en R2 el valor de la dir de memoria de "B + R0".
  dadd r3,r1,r2 -> Suma R2 y R1 y guarda el resultado en R3.
  sw r3, C(r0) -> Guarda el valor de R3 en la dir de memoria "C + R0".

  lw r4, D(r0) -> Carga en R4 el valor de la dir de memoria de "D + R0".
  lw r5, E(r0) -> Carga en R5 el valor de la dir de memoria de "E + R0".
  dadd r6,r4,r5 -> Suma R5 y R4 y guarda el resultado en R6.
  sw r6, F(r0) -> Guarda el valor de R6 en la dir de memoria "F + R0".

halt -> Para el programa.
```

Un código que realice una operación similar en C++ sería algo semejante a esto:

```
int main()
{
  int A = 2, B = 3, C = 0, D = 4, E = 5, F = 0;

  // Las dos instrucciones:
  int C = A + B;
  int F = D + E;

  return 0;
}
```

b) Indica qué tipos de datos se están utilizando y relaciónalo con las instrucciones que se utilizan.

El tipo que se usa son palabras de 32 bits. Se carga su valor con LW en los registros, se suma su valor, y se vuelve a guardar con SW. Se realizan las siguientes operaciones: C = A + B; F = D + E;

c) Ejecutar el programa en el simulador con la opción **Configure / Enable Forwarding deshabilitada**. Analizar ciclo a ciclo su funcionamiento con la opción “Single cycle” del menú Execute o bien presionando F7 sucesivamente. Examinad las distintas ventanas que se muestran en el simulador y responder:

- ¿Qué ocurre en los ciclos 5, 6, 13 y 14 con las instrucciones dadd?

  En el ciclo 5 - 6, la instrucción LW entra y acaba la fase WB, cosa que permite que la instrucción que es dependiente DADD pueda proceder a la fase ID, teniendo finalmente el valor en R2 disponible, y reanudando el cauce.

  En los ciclo 13 - 14, pasa exactamente lo mismo que en los ciclos 5 - 6, pero con el valor de R5.

- ¿Qué ocurre en los ciclos 8, 9, 16, 17 con las instrucciones sw?

  En los ciclo 8 - 9, la instrucción DADD entra y acaba la fase WB, cosa que permite que la instrucción que es dependiente SW pueda proceder a la fase ID, teniendo finalmente el valor en R3 disponible, y reanudando el cauce.

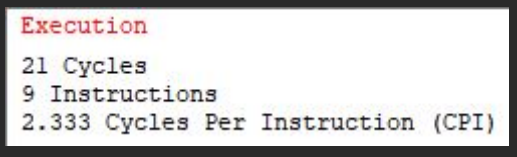
  En el ciclo 16 - 17, pasa exactamente lo mismo que en los ciclos 8 - 9, pero con el valor de R6.

- ¿Cuántos ciclos se consumen en total y cuántos de ellos son detenciones?

  Hay un total de 21 ciclos, y de esos 21, 8 son detenciones del cauce por riesgo de dependencia.

- ¿Calcula el CPI?

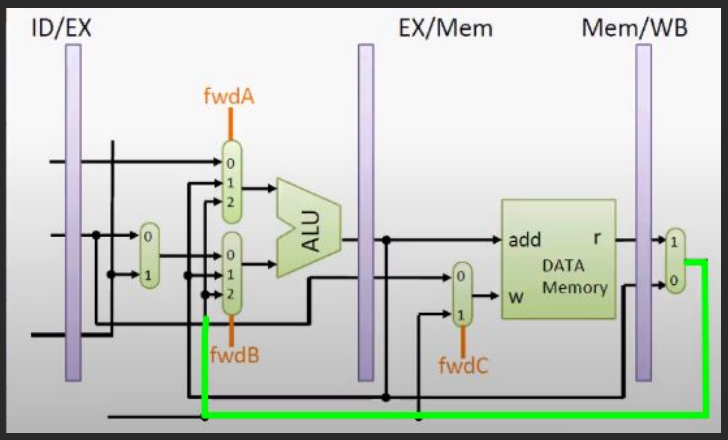
  CPI = Ciclos / Instrucciones = 21 / 9 = 2.333



d) Ejecutad el programa en el simulador con la opción **Configure / Enable Forwarding habilitada**. Analizar ciclo a ciclo su funcionamiento con la opción “Single cycle” del menú Execute o bien presionando F7 sucesivamente. Examinad las distintas ventanas que se muestran en el simulador y responder:

- ¿Qué ocurre en los ciclos 6 y 11 con las instrucciones dadd? ¿Se producen adelantamientos? En su caso indica cuales.

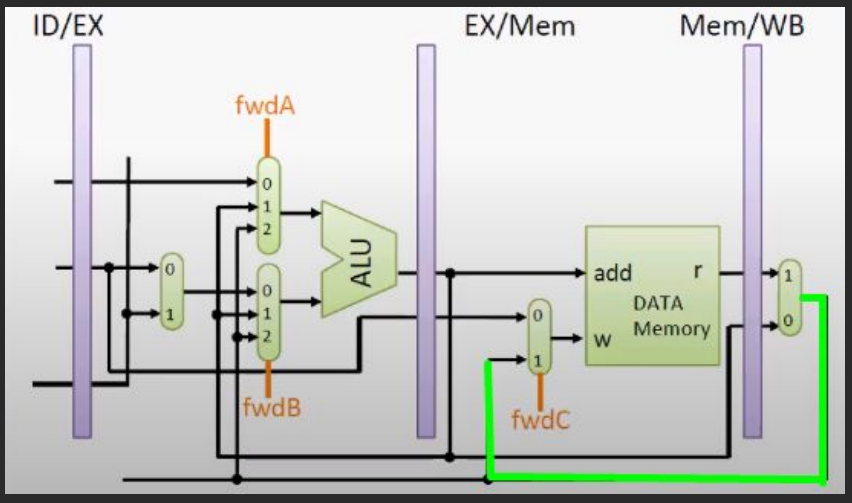
  En los ciclos 6 y 11, se detecta un adelantamiento, por lo tanto, se le permite a la instrucción DADD avanzar a la fase ID, se detiene el cauce 1 etapa para que, vía forwarding, el dato de LW esté disponible en la etapa EX para la instrucción DADD.



  Los adelantamientos se producen en los ciclos 5 - 6. y 10 - 11.

- ¿Se producen adelantamientos en los ciclos 8 y 13 vinculados a las instrucciones sw? En su caso indica cuales.

  En los ciclos 8 y 13 se detecta un adelantamiento, por lo tanto, se le permite a la instrucción SW avanzar a la fase EX, y en la siguiente etapa, el dato de DADD que estará en la etapa WB será mandado a través de los multiplexores a la etapa MEM para su uso en la instrucción SW. En este caso no hace falta detener el cauce, ya que la instrucción SW solo necesita dicho dato en la etapa MEM a diferencia de la instrucción DADD que necesita el dato en la etapa EX.



- ¿Cuántos ciclos se consumen en total y cuántos de ellos son detenciones?

  Hay un total de 15 ciclos, de los cuales 2 son detenciones del cauce por riesgo de dependencia.

- ¿Calcula el CPI?

  CPI = Ciclos / Instrucciones = 15 / 9 = 1.667

```
Execution
15 Cycles
9 Instructions
1.667 Cycles Per Instruction (CPI)
```

e) Propón una reorganización del código para reducir el número de detenciones manteniendo el resultado final del programa sobre los registros y memoria.
Escribe el código reorganizado:

```
.data

  A: .word32 2
  B: .word32 3
  C: .word32 0
  D: .word32 4
  E: .word32 5
  F: .word32 0

.code

  lw r1, A(r0)
  lw r2, B(r0)
  lw r4, D(r0)
  lw r5, E(r0)
```

```
  dadd r3, r1, r2
  sw r3, C(r0)
  dadd r6, r4, r5
  sw r6, F(r0)

halt
```

De esta manera, las DADD no tienen que esperar tanto (con forwarding ni esperan) a que los datos estén listos, ya que para cuando las instrucciones DADD se ejecutan, dichos datos están ya escritos.

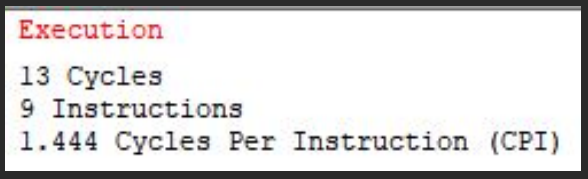
Con el **adelantamiento activado**:

- ¿Cuántos ciclos se consumen en total y cuántos de ellos son detenciones?

  Hay un total de 13 ciclos, con 0 detenciones.

- ¿Calcula el CPI?

  CPI = Ciclos / Instrucciones = 13 / 9 = 1.444



- **Programa 4.**

Introduce el siguiente código en el simulador:

```
.data

  cant: .word 8
  datos: .word 1, 2, 3, 4, 5, 6, 7, 8
  res: .word 0

.code

  dadd r1, r0, r0
  ld r2, cant(r0)
  loop: ld r3, datos(r1)
    daddi r2, r2, -1
    dsll r3, r3, 1
    sd r3, res(r1)
    daddi r1, r1, 8
  bnez r2, loop

halt
```

a) Indica cuál es el objetivo del código y cuál será el resultado.

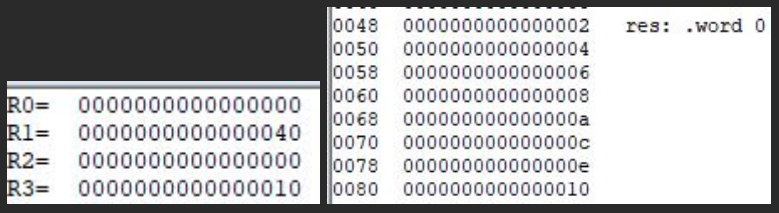
R0 = 0, R1 = 0, R2 = “cant + R0”
loop: LD R3 “datos + R1”
DADDI R2 -= 1
DSLL R3 1
SD R3 “res + R1”
DADDI R1 += 8
IF R2 != 0; loop

R2 se usa como contador, se define una vez con el valor 8, y por cada iteración se le resta 1, cuando es 0, se sale del bucle.

R3 se usa para cargar los datos del vector datos, usando R1 para seleccionar la posición, después, se desplaza los bits a la izq 1 vez. También para guardar el valor del desplazamiento en la dirección de memoria “res + R1”.

R1 se usa para apuntar a diferentes posiciones de memoria, para seleccionar una posición del vector, y para guardar en una posición de memoria que apunta res y superiores. tras cada iteración se le suma 8 para acceder a la siguiente posición de memoria, y 8 específicamente ya que los registros son de 64 bits.

El resultado es el siguiente:



b) Identifica los riesgos por dependencia de datos que pueden aparecer.

Registro	Instrucción Origen	Instrucción Depend.	Tipo
R1	DADD	LD	RAW
R3	LD	DSLL	RAW
R3	DSLL	SD	RAW

c) Ejecuta el programa en el simulador con la opción **Configure / Enable Forwarding deshabilitada**. Analizar ciclo a ciclo su funcionamiento con la opción “Single cycle” del menú Execute o bien presionando F7 sucesivamente. Examina las distintas ventanas que se muestran en el simulador y responde a las siguientes cuestiones:

- ¿En qué ciclo ocurre la primera parada por dependencia de datos? ¿En qué instrucción?

  En el Ciclo 4, en la instrucción LD.

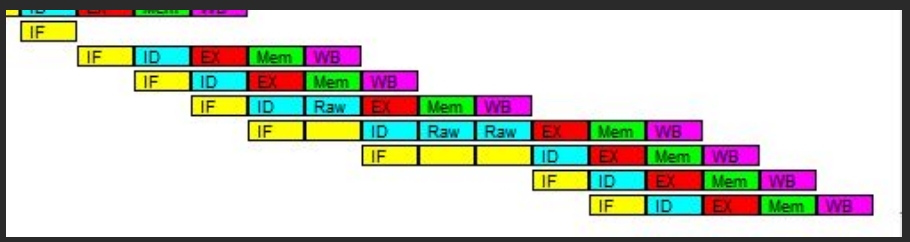
- ¿En qué ciclo se producen dos detenciones en la misma instrucción? ¿A qué se debe?

  En los Ciclos 9 - 10, se debe a que no hay ninguna instrucción entre esas instrucciones (DSLL y SD) que pueda acaparar el cauce mientras la instrucción SDLL finaliza.

- Observa que ocurre con la instrucción de salto. ¿En qué ciclo ocurre la primera parada por salto efectivo?

  En el ciclo 12, la parada es necesaria para asegurar el salto correctamente.

- Ejecuta el programa completo y observa que la última vez que se ejecuta el salto no hay parada debido a que el salto ha sido no efectivo.



- ¿Cuántos ciclos tarda el programa en ejecutarse? ¿Cuántos de ellos son detenciones? ¿Cuál es el CPI?

  Hay un total de 87 ciclos, con 32 detenciones, el CPI es 87 / 51 = 1.706

```
Execution

87 Cycles
51 Instructions
1.706 Cycles Per Instruction (CPI)
```

```
Stalls

25 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
7 Branch Taken Stalls
0 Branch Misprediction Stalls
```

e) Ejecuta el programa en el simulador con la opción **Configure / Enable Delay Slot habilitada**. Analizar ciclo a ciclo su funcionamiento con la opción “Single cycle” del menú Execute o bien presionando F7 sucesivamente. Examinad las distintas ventanas que se muestran en el simulador y responde a las siguientes cuestiones:

- ¿Cuántos ciclos se han consumido? ¿Por qué ha seguido ejecutándose la instrucción halt a partir del ciclo 9?

  Hay un total de 13. Porque en este caso se ha tomado la instrucción halt como instrucción del delay slot.

- Modifica el programa cambiando la instrucción daddi r1, r1, 8 de lugar y colocándola después del salto (en el delay slot). Ejecuta de nuevo el programa y observa qué ocurre. ¿Es correcto el resultado?

  Sí, funciona correctamente debido a que con delay slot, la instrucción después del salto se ejecuta antes del salto, y no permitiendo ejecutar la siguiente instrucción (halt).

- ¿Cuántas paradas por riesgo de control se han eliminado? ¿Se ha incrementado el número de instrucciones a ejecutar? ¿Cuántos ciclos de reloj se han consumido?

  Antes

```
Stalls

25 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
7 Branch Taken Stalls
0 Branch Misprediction Stalls
```

  Después

```
Stalls

39 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
0 Branch Taken Stalls
0 Branch Misprediction Stalls
```

  Se han eliminado 7 paradas por riesgo de control, pero han aumentado las de datos.

  Antes

```
Execution

87 Cycles
51 Instructions
1.706 Cycles Per Instruction (CPI)
```

  Después.

```
Execution

94 Cycles
51 Instructions
1.843 Cycles Per Instruction (CPI)
```

  El número de instrucciones son las mismas, aunque ahora hay un total de 94 Ciclos.

- ¿Cuál es el CPI?

  CPI = Ciclos / Instrucciones = 94 / 51 = 1.843

- ¿Qué otra instrucción se hubiera podido colocar en el delay slot? ¿Habría sido necesario realizar algún otro cambio en el código?

```
  daddi r2, r2, -1
```

  La instrucción DADDI del contador podría haberse usado igualmente en la delay slot.
  Para obtener los mismos valores exactamente, tendríamos que ejecutar dicha instrucción justo después de cargar R2 en la instrucción LD. El código sería algo similar al siguiente:

```
.data

  cant: .word 8
  datos: .word 1, 2, 3, 4, 5, 6, 7, 8
  res: .word 0

.code

  dadd r1, r0, r0
  ld r2, cant(r0)
  daddi r2, r2, -1

  loop: ld r3, datos(r1)
    dsll r3, r3, 1
    sd r3, res(r1)
    daddi r1, r1, 8
  bnez r2, loop
  daddi r2, r2, -1

halt
```

f) El código del programa 5 realiza la misma función que el programa 4 que acabas de estudiar:

- **Programa 5.**

Introduce el siguiente código en el simulador:

```
.data

  cant: .word 8
  datos: .word 1, 2, 3, 4, 5, 6, 7, 8
  res: .word 0

.code

  ld r2, cant(r0)
  dadd r1, r0, r0
  loop:
    beqz r2, fin
    ld r3, datos(r1)
    daddi r2, r2, -1
    dsll r3, r3, 1
    sd r3, res(r1)
    daddi r1, r1, 8
  j loop
  fin:

halt
```

- ¿Qué diferencias observas?

  El programa 4 es un do_while, y el programa 5 es un while.

- Ejecuta el programa en el simulador con la opción **Configure / Enable Delay Slot deshabilitada**. ¿En qué instrucciones y en qué casos se produce detención por riesgo de control?

  En los Ciclos 14, 25, 36, 47, 58, 69, 80, 91, 93.
  En las instrucciones j loop, y en beqz al acabar el programa.

- Compara las estadísticas que se obtienen en el número de ciclos y CPI con las obtenidas en el programa 4 con las opción **Configure / Enable Forwarding habilitada** y **deshabilitada**.

Sin forwarding.

```
Execution
98 Cycles
60 Instructions
1.633 Cycles Per Instruction (CPI)

Stalls
25 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
9 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
40 Bytes
```

Con forwarding.

```
Execution
74 Cycles
60 Instructions
1.233 Cycles Per Instruction (CPI)

Stalls
1 RAW Stall
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
9 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
40 Bytes
```

- Coloca una instrucción válida en el delay slot y ejecuta con la opción **Configure / Enable Delay Slot habilitada**. ¿Cuántos ciclos se consumen? ¿Cuál es el CPI?.

Con **Forwarding** y **Delay slot habilitada**, consumimos 66 ciclos.
CPI = Ciclos / Instrucciones = 66 / 61 = 1.082

```
Execution
66 Cycles
61 Instructions
1.082 Cycles Per Instruction (CPI)
```

- ¿Qué conclusiones puedes extraer al comparar el número de ciclos y CPI de los dos programas estudiados?

Que la segunda manera es mucho más óptima que la primera, ya que su CPI y detenciones del cauce es mucho menor en comparación.