15 DE JUNIO DE 2020

FASE INDIVIDUAL 3

ARQUITECTURA DE LOS COMPUTADORES

OSCAR CASADO LORENZO | 77580351V
UNIVERSIDAD DE ALICANTE
POLITÉCNICA SUPERIOR

PROGRAMA 1

```
.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

LD (Load Double):
Es una instrucción de almacenamiento que, en este caso accede a la posición de memoria Mem[num+r0] y copia su contenido (específicamente un byte por ser Load Double) en el registro R2.

DADD (Double-Precision Add):
Esta instrucción accede tanto al registro R8 como al R9 y los suma entre ellos (teniendo en cuenta el signo) para posteriormente guardar el resultado en R3.

DSUB (Double-Precision Substract):
De manera muy similar a la instrucción anterior, cambiando la operatoria de suma por la resta de los registros R5 y R6, finalmente escribe el registro R10 la solución.

DSLL (Doubleword Shift Left Logical):
Desplaza a la izquierda los bits de un espacio de memoria, el nº de veces indicado por el segundo argumento (en este caso uno). Lo aplica sobre el contenido del registro R4 y para no modificarlo lo almacena en R1.

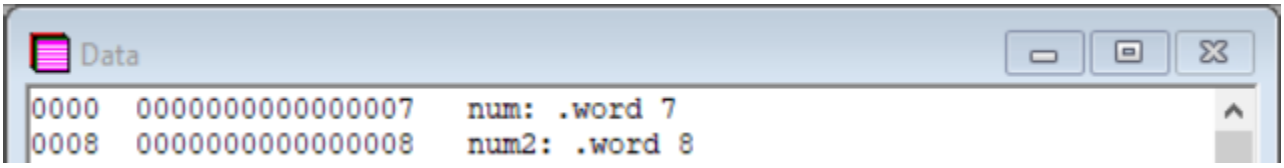
SD (Set Double):
Sobreescribe el registro R4 insertando el contenido que haya en la dirección de memoria Mem[num2+r0].

Halt:
Detiene a ejecución del simulador WinMips.

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

Como podemos observar en el código fuente de dicho programa únicamente reservamos memoria para 2 variables: **num** y **num2**, las cuales son inicializadas a 7 y 8 respectivamente. Además, cabe destacar que exactamente se les asignan 16 bits a cada una por el hecho de haber indicado la tipología **.word**.

Si deseamos localizarlas en el simulador, tendremos que acceder a la ventana **'Data'**, en la cual nos parecerá lo siguiente:

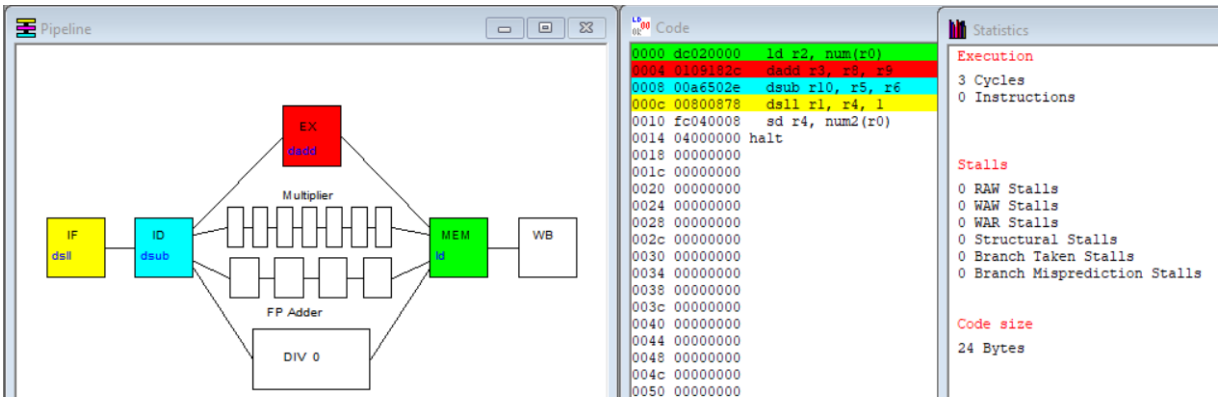


c) Ejecutad el programa en el simulador con la opción *Configure/Enable Forwarding* deshabilitada. Analizar ciclo a ciclo su funcionamiento con la opción "Single cycle" del menú Execute o bien presionando F7 sucesivamente. Examinad las distintas ventanas que se muestran en el simulador y responder:

- **¿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?**

  Contestando a la primera pregunta la variable **num** es consultada en el tercer ciclo del programa, esto se puede consultar fácilmente desde la pestaña **'Pipeline'** (en la cual se nos muestra en qué etapa se encuentra cada instrucción).

  Una vez ahí deberemos ir ejecutando paso a paso el programa hasta que lleguemos a la **etapa MEM** de la **instrucción LD** (acceso a num)**.** En ella accederemos directamente al banco de datos como se muestra en la siguiente imagen:

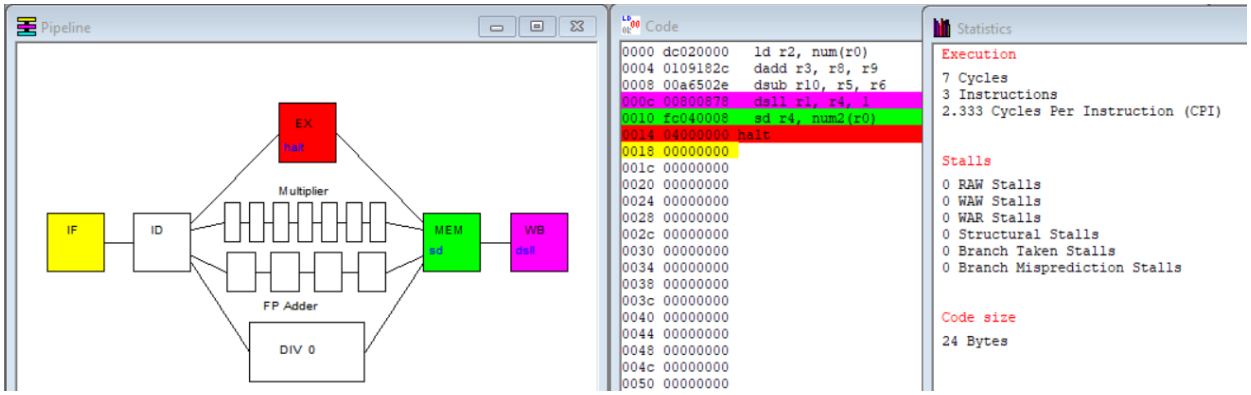


  Aunque pueda parecer confuso únicamente deberemos centrarnos en que nos encontramos en el **tercer ciclo** y la primera instrucción (LD) se encuentra resaltada en verde, justo el mismo color en el que se encuentra la casilla MEM del Pipeline, por lo que es justo en ese instante en el que leemos el dato.

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

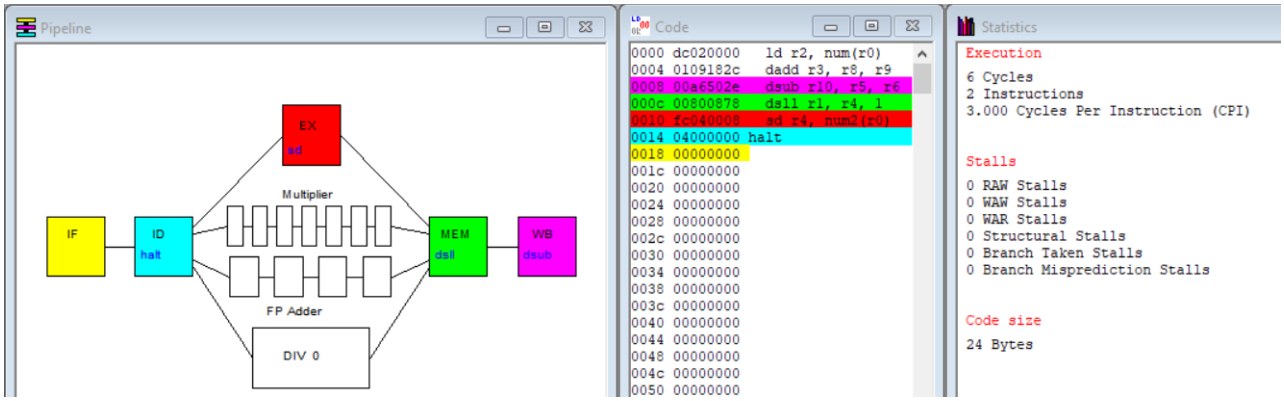
  Al igual que en el apartado anterior debemos esperar a que la instrucción que contiene a dicha variable (SD) se encuentre en la **etapa MEM,** lo que sucede al final del ciclo 7 como se muestra en la ventana **'Statistics'**.

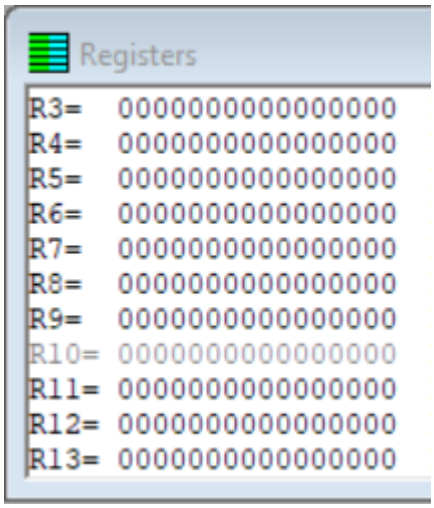
  Y quizás se produzcan dudas sobre ¿cómo es que dicha etapa posea varias funcionalidades? Esto se debe a que para evitar conflictos en la primera parte del ciclo lee y en la segunda escribe.



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

  Primero debemos localizar donde se puede producir esto para facilitar el trabajo. Explorando el código, después de hacer esto localizamos esto en la **instrucción DSUB** en la cual dicho registro es el destino del resultado. Una vez hecho este análisis previo únicamente debemos de reanudar la ejecución del programa hasta que dicha instrucción llegue a la **etapa WB** (encargada de la escritura en el banco de registros).





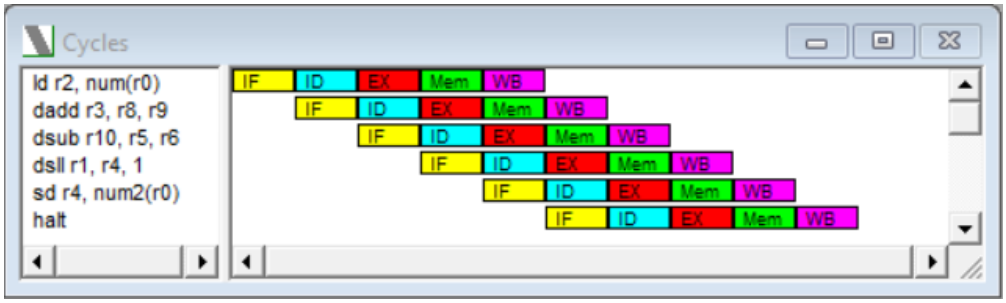
Como muestra el subrayado morado, la instrucción DSUB llega a esta etapa en el **ciclo 6**.

Además, como vemos en esta etapa, el registro afectado queda remarcado en la ventana **'Registers'**. Sin embargo, sigue valiendo cero debido a que al restar dos registros que se encuentran sin inicializar el resultado arrojado es ese.

- **Tras la ejecución, ¿se produce alguna detención en el cauce? Razona tu respuesta.**

  Antes de contestar a esta pregunta me gustaría resaltar que la forma más sencilla en la que apreciaremos la existencia o no de dichas detenciones/paradas es en la pestaña **'Cycles'**, en la que aparecerán las "burbujas" características de las ejecuciones sin forwarding.

  Dicho esto, he aquí el resultado de la ventana al ejecutar el programa:



Como se observa el cauce está completamente lleno, y no existen problemas de flujo ni burbujas, por lo que podemos afirmar que no existe ninguna detención a lo largo de la ejecución. Es cierto, que se podría haber llegado a la misma conclusión observando que en ese código no existe ninguna dependencia, indicado en la ventana '**Statistics'** como **0 _ STALLS** (RAW, WAR, Structural, etc).

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

  Mediante la siguiente formula junto con una serie de datos que arroja el propio simulador podemos hallar los CPI del programa:

```
Execution
10 Cycles
6 Instructions
1.667 Cycles Per Instruction (CPI)


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


Code size
24 Bytes
```

$$CPI = \frac{Ciclos\ de\ reloj\ CPU}{Recuento\ de\ Instrucciones}$$

Por lo que teniendo los siguientes datos es tan fácil como operar, obteniendo un resultado de 1,667 ciclos por instrucción, la misma cifra que arroja de manera automática **WinMips**.

Programa 2

```
.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 r5,r5,r1
        sd r7,D(r5)
        dadd r5,r5,r1
        sd r8,D(r5)
        daddi r9,r5,8
        ld r10,D(r5)
        sd r10,D(r9)
halt
```

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

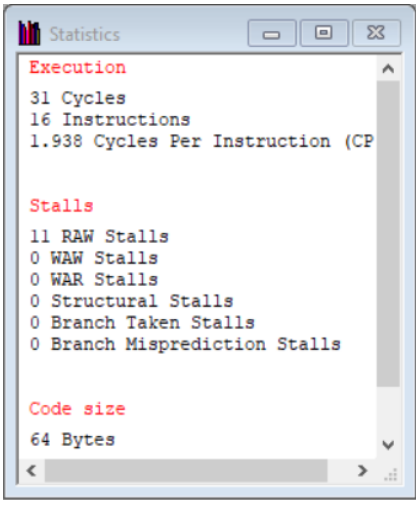
En la primera parte de este apartado me ceñiré a explicar brevemente aquellas líneas de código que difieran (en conceptos) de las existentes en el PROGRAMA 1.

XOR: Lleva acabo una **comparación** lógica **bit** a **bit** de dos registros, en este caso el mismo, utilizando como referencia la **puerta** compuesta **XOR**, almacenando el **resultado** nuevamente en dicho **registro**.
DADDI: Suma **R5 + 9** y lo **guarda** en el registro **R9**, es una operación que **no** tiene en cuenta el **signo** de los registros.

Cerrando este apartando, en cuanto al propósito del código es básicamente el de **recorrer** una serie de **valores** y **registros** para luego **cargarlos** en una nueva **dirección** de **memoria**.

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:

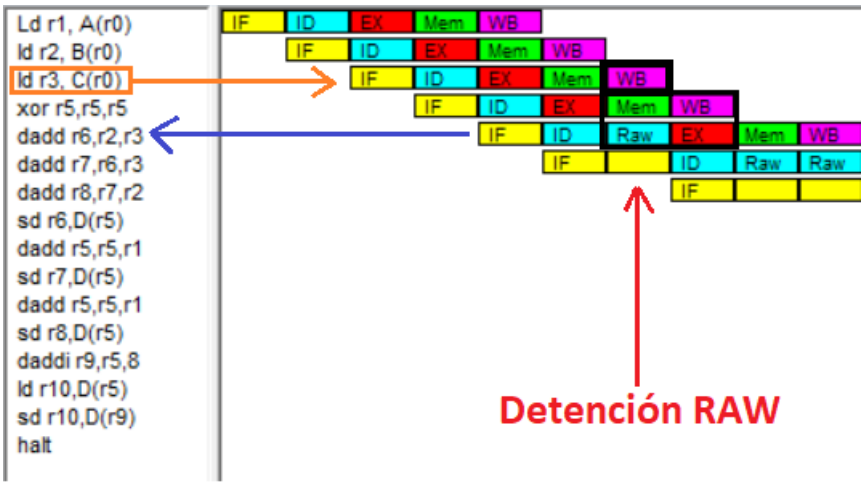
- **¿Cuántas detenciones RAW aparecen?**



Aparecen **11 RAW Stalls (Read after Write)** lo que significa que se intenta acceder a un registro antes de que este se haya actualizado correctamente tras su uso en una instrucción anterior.

- **¿Qué instrucciones están generando las mismas (stalls) en el cauce?**

  Antes de entrar en detalle sobre cuales son las instrucciones que producen los **stalls** me gustaría hacer un breve inciso sobre como se pueden deducir las mismas y el motivo de estas detenciones, para ello me serviré de una representación parcial del flujo de este mismo programa.



  Como podemos observar en la imagen las detenciones son una especie de 'burbujas' o de ciclos en los que una instrucción queda en espera de que algún posible problema sea solucionado con el propio paso del tiempo. Estos problemas, concretamente los que producen RAW Stalls se generan como mencioné en el apartado anterior por acceder a un dato que no ha terminado de actualizarse.

  Por ejemplo, a la casilla indicada por la **flecha roja** le correspondería un **EX**; sin embargo, esto no es posible ya que **no podemos** cargar un registro que aun no se encuentra disponible. Es decir, hasta que no haya finalizado la fase de escritura **(WB)** producida por la instrucción **'ld r3, C(r0)'** no podremos usar el registro R3; es esto, por tanto, lo que representa la detención.

- **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?

  Contestado en el apartado anterior.

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

  Como se ha mencionado anteriormente, todos los **Stalls** de este programa son **RAW** por lo que, si que se **consideran** el **mismo tipo** de riesgo, aunque lo produzcan instrucciones distintas, como lo pueden ser 'dadd' y 'sd'.
  Y el motivo de que varíe la cantidad de las mismas es simple, **lo cercanas que se encuentren en el tiempo 2 instrucciones con una dependencia**, este factor podría hacer desde que no se produzca ninguna hasta que dichas detenciones supusiesen un coste muy elevado.

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

  Contestado en el apartado anterior.

- **¿Cuál es el promedio de ciclos por instrucción (CPI) en la ejecución de este programa bajo esta configuración?**
  Como se muestra, de la misma forma que en el PROGRAMA 1 tras finalizar la ejecución de este, se arroja un CPI de 1,938. Podríamos volver a comprobarlo con el resto de datos intermedios, pero veo apropiado obviarlo.

c) 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?**
  Para empezar, hay que tener en cuenta que no siempre que habilitemos la opción de adelantamiento van a eliminarse todas las detenciones, ya que eso depende mucho del tipo de problema que se nos plantee y de las instrucciones empleadas para implementar el mismo.

  Dicho esto, me gustaría distinguir entre 2 tipos de adelantamiento: **Adelantamiento ALU** y **Adelantamiento de Memoria**, los cuales emplearemos dependiendo de la instrucción que produzca la detención de todas formas, quedará explicado a continuación.

- **¿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?.**

  Antes de aplicar **forwarding** teníamos en el primer riesgo el cauce de la siguiente forma. Como podemos observar esto nos producía como se enuncia, un riesgo en el ciclo número 6, producido por la instrucción resaltada en amarillo **LD**.

	3	4	5	6	7	8	9	10
LD	IF	ID	EX	MEM	WB			
XOR		IF	ID	EX	MEM	WB		
DADD			IF	-	ID	EX	MEM	WB

  Pero después de activar esta opción y reactivar el simulador obtenemos una variación con respecto al cauce original. Esto se debe a que como la instrucción que ha producido el riesgo es de carga, podemos **adelantar el resultado desde** la etapa de **MEM hasta** la **EX** de la instrucción afectada. Esto, si nos fijamos nos elimina la detención anterior ahorrándonos 1 ciclo.

	3	4	5	6	7	8	9
LD	IF	ID	EX	MEM	WB		
XOR		IF	ID	EX	MEM	WB	
DADD			IF	ID	EX	MEM	WB

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

  El caso que se nos presenta a continuación no se resuelve de igual forma, puesto que, como podemos observar, el riesgo no viene producido por una instrucción de carga, sino que se trata de una aritmética, entonces ¿podemos introducir un adelantamiento? y si es el caso ¿cómo lo hacemos?

DADD	IF	ID	EX	MEM	WB			
DADD		IF	-	-	ID	EX	MEM	WB

  Pues bien, esto se consigue de la siguiente forma, al tratarse de una expresión aritmética **(DADD)**, lo que podemos hacer es **obtener** el resultado de la **ALU** directamente y encauzarlo hasta la siguiente, esto en ámbito de etapas se basa en que se puede **adelantar** un dato desde la **fase EX** de la instrucción que produce el riesgo hasta la misma fase en el **siguiente ciclo** de **reloj**.

DADD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

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

  El último riesgo en comparación al primero es similar debido a que posee detenciones y, son producidas por una instrucción de carga por lo que se procederá al adelantamiento de la misma manera.

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

  Mientras que el anterior producía un CPI de **1,938** este logra mejorarlo considerablemente obteniendo una marca de **1,250**. Esto se debe a que, aun teniendo el **mismo** número de **instrucciones**, gracias a los adelantamientos **reducimos** (incluso eliminamos) todos aquellos **ciclos** excesivos que eran **consecuencia** de las **dependencias**.

PROGRAMA 3

```
.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?

Puesto que no hay **ninguna** instrucción **novedosa** con respecto a los problemas anteriores no haré **ningún comentario** sobre su funcionamiento, ya que lo encuentro **trivial**.

Sobre la implementación de dicho código en **2 instrucciones** de lenguaje en **alto nivel**, aunque he **probado** varias **combinaciones** que me resultaban lógicas a primera vista, **ninguna** se ha **ajustado estrictamente** al **funcionamiento** de este. Con **3** instrucciones si que he sido **capaz** de reproducirlo.

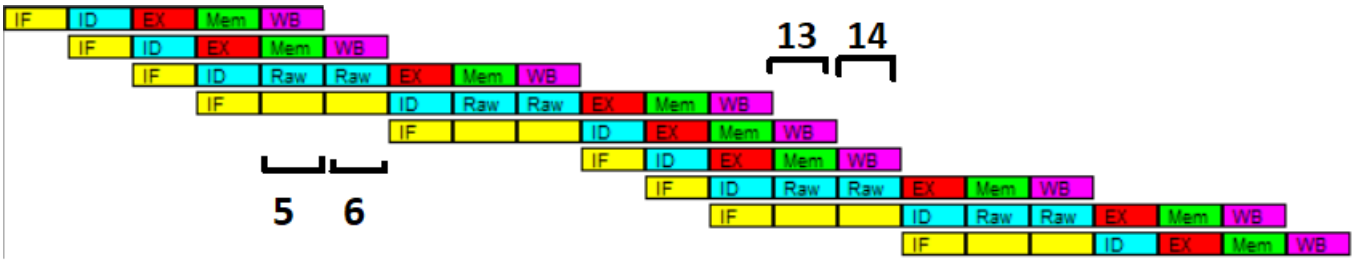
Para finalizar, como **conclusión** me gustaría destacar que, aunque no la haya encontrado lo mas **probable** es que la haya ya que conforme es **mayor** el **nivel** de **abstracción** mayor es la cantidad de **información** que se puede representar de forma mas **minimalista**.

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

Los datos utilizados en este programa son almacenados en posiciones de memoria en la cual se le reserva a cada uno 32 bits.

c) Ejecutad 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?**



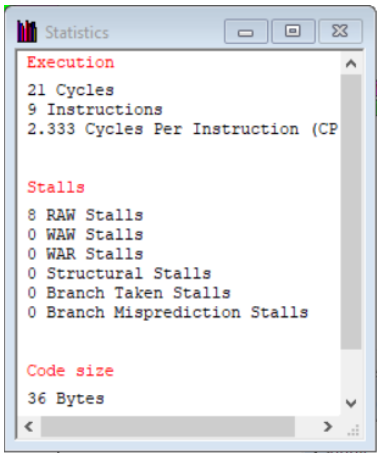
Como se muestra en la imagen en los **ciclos 5 y 6** se producen **detenciones** ya que la instrucción tiene que esperar a que el registro **solicitado**, en este caso **R3** estén **escritos**, puesto que hasta la última etapa no se cargan en el **banco** de **registros**, como hemos reiterado a lo largo de esta práctica.

Exactamente lo **mismo** sucede en los **ciclos 13 y 14** con la solicitud del registro **R6**.

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

Al igual que en los ciclos anteriores, ocurren **detenciones** del **cauce** debido a que la instrucción en el estado de decodificación **(ID) solicita** un **registro** que aún **no** están en **memoria**, por lo que deberá esperar a que estén **disponibles**, lo que como observamos en la imagen del apartado anterior (la cual abarca dichos ciclos) no sucede hasta el instante **nº17**.

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



Como se observa en la imagen sin adelantamiento los ciclos totales del programa ascienden hasta 21, siendo 8 de ellos detenciones.

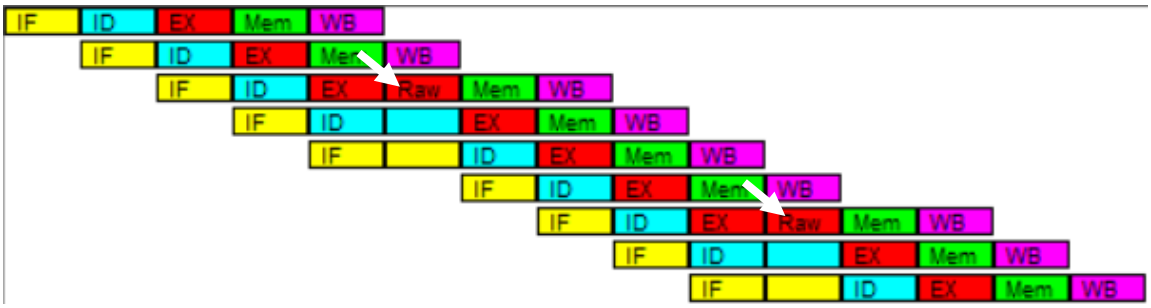
Esto en mi opinión es una ratio bastante elevada, lo que no supone grandes desperdicios de forma innecesaria que en problemas de mayor envergadura podrían suponer que el código fuente fuese directamente inviable para cálculos complejos como la sucesión de Fibonacci recursiva, etc.

- **Calcula el CPI**

El CPI resultante de este programa es de **2,33** resultante de **9 instrucciones** finalizadas en **21 ciclos**.

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:

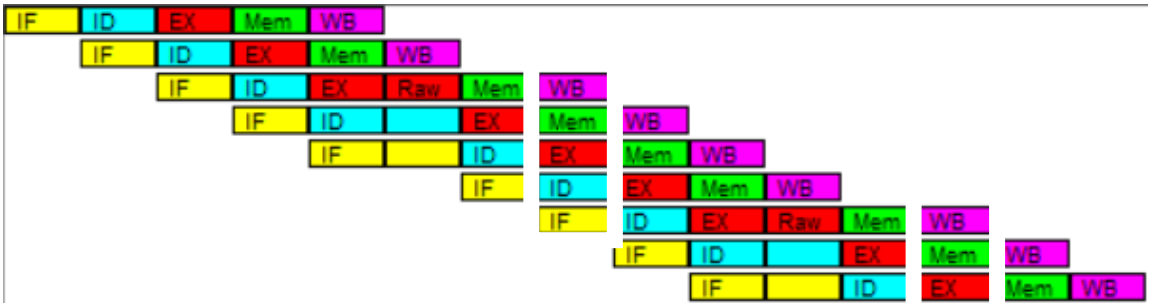
- **¿Qué ocurre en los ciclos 6 y 11 con las instrucciones dadd? ¿Se producen adelantamientos? En su caso indica cuales.**



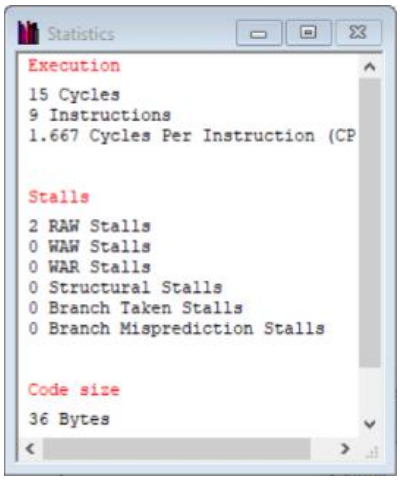
Como muestra el **Pipeline** del programa aplicando la estrategia de **forwarding** si que se presentan adelantamientos tanto en el **ciclo 6** como en el **11**, sin embargo, **no solucionan** por **completo** las **detenciones**. Esto se debe a que las **instrucciones** que las producen son de **carga**, permitiéndonos (como se ha visto en problemas anteriores) **adelantar** el dato, **R1** y **R4** respectivamente, desde la etapa **MEM hasta** la **EX** como muestran las flechas blancas.
Si dichas instrucciones fueran aritméticas podríamos asegurar que ambas detenciones iniciales serían suprimidas.

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

Efectivamente en **ambos** ciclos se **producen adelantamientos**, además podemos afirmar que dichos **adelantamientos** son del **mismo tipo**, generados por la **Unidad Aritmético Lógica**. Esto significa que una vez **finaliza la etapa EX** de una instrucción el dato puede ser usado en el **ciclo siguiente** por la siguiente instrucción mediante la etapa **EX** también.



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



Como breve resumen encontramos un CPI **'esperable'** con un total de **15 ciclos**, de los cuales únicamente **2** de ellos componen **detenciones**, específicamente **RAWs**.

- **¿Calcula el CPI?**
  En esta ocasión y, como era de esperar el **CPI** se ve **reducido** gracias a la **optimización** aportada por el **forwarding**, arrojando un valor de **1,667**.

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.**

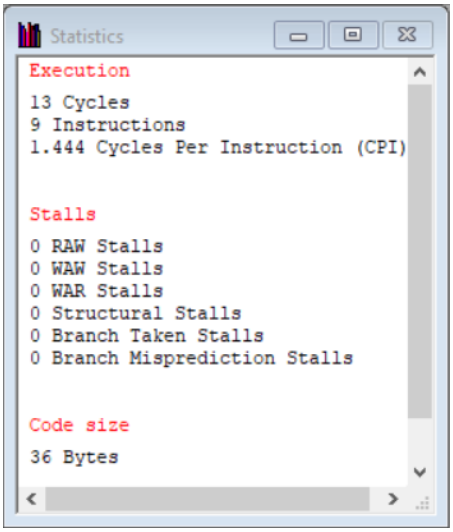
```
.data
        A: .word32 2
        B: .word32 3
        C: .word32 0
        D: .word32 4
        E: .word32 5
        F: .word32 0
.code
        lw r1, A(r0)
        lw r2, B(r0)
        lw r4, D(r0)
        lw r5, E(r0)
        dadd r3,r1,r2
        sw r3, C(r0)
        dadd r6,r4,r5
        sw r6, F(r0)
halt
```

Como ya se ha mencionado en esta documentación, la abundancia o escasez de detenciones recae en gran medida sobre lo distanciadas que se encuentren las dependencias en el tiempo.

Por ello, me parece adecuado crear esta nueva disposición en la que los accesos a memoria se produzcan al principio para reducir en su justa medida las detenciones colindantes.

Con el adelantamiento activado…

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



- **¿Calcula el CPI?**
  El CPI, como se muestra en la imagen resultante de las estadísticas producidas por el código **reordenado** es de **1,44** , un resultado **levemente inferior** al producido por el código **original**. Si entramos en detalles sobre por qué sucede esto, claramente se aprecia que la diferencia reside en la **ausencia** absoluta de **detenciones**, como se ha comentado en la pregunta superior.

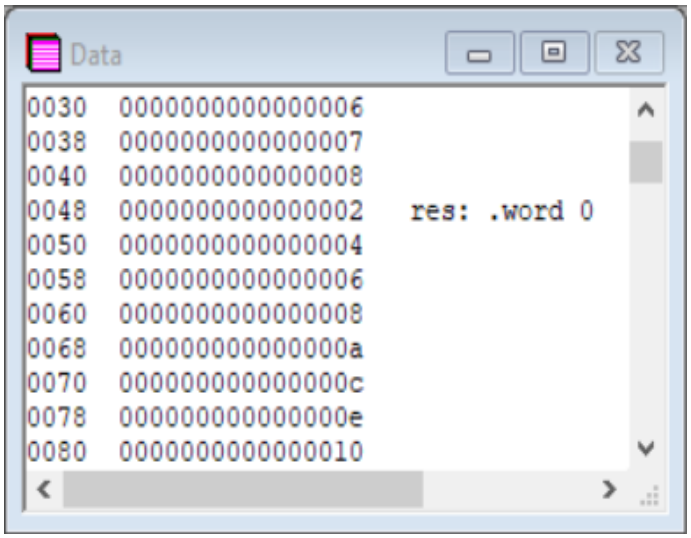
PROGRAMA 4

```
.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.



El objetivo se basa en **multiplicar** todos los elementos de un **vector 'datos'** por una **constante**, en este caso **2**. Esto lo haremos iterativamente hasta que indique la variable auxiliar denominada **'cant',** es decir, si su valor es 5 y tenemos un vector de 9 elementos únicamente **modificaremos** el valor de los **5 primeros**.

Dicho esto, la disposición de valores en memoria para este código en concreto debería ser: **2 ,4, 6, 8, c, e, 10**. Respecto a los caracteres, simplemente es que los valores se encuentran en hexadecimal, por tanto, c=12, e=14, etc. Para comprobarlo únicamente deberemos ejecutar el programa y, al finalizar consultar la pestaña **'Data'.**

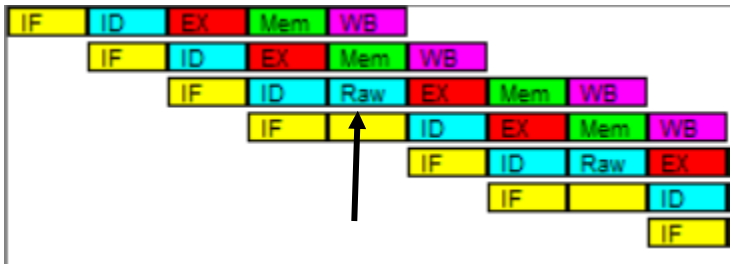
b) Identifica los riesgos por dependencia de datos que pueden aparecer.

Como dependencias potenciales podemos encontrar:

- **LD r3, datos(r1) :** En instrucción el registro R1 no se habrá escrito aun cuando intente ser leído, lo que producirá una parada.
- **DSLL r3, r3, 1 :** A esta le sucede lo mismo que a la anterior, con la diferencia que el registro al que buscaremos acceder es R3.
- **SD r3, res(r1) :** Exactamente la misma problemática que la segunda instrucción con dependencias.

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:

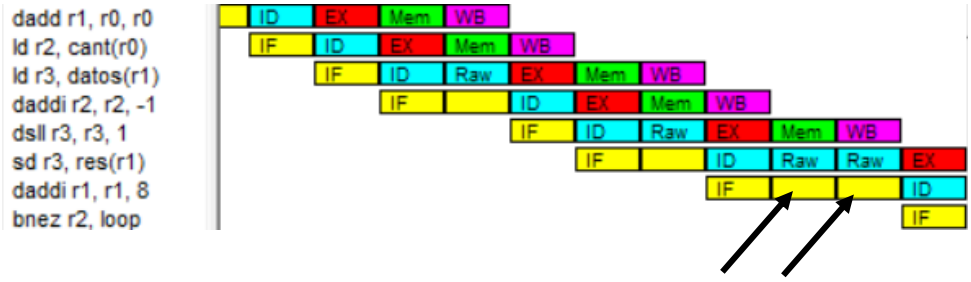
- **¿En qué ciclo ocurre la primera parada por dependencia de datos? ¿En qué instrucción?**



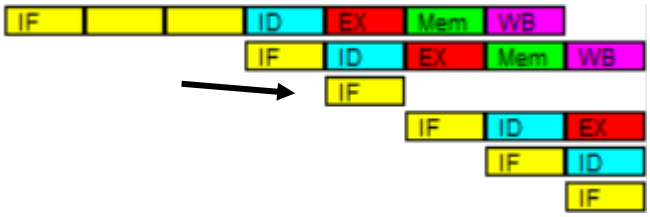
Como observamos en el flujo no se produce ninguna **detención** hasta el **ciclo nº5** por la **primera** instrucción que hemos citado en el **apartado anterior**.

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

Específicamente, el ciclo a partir del que se producen sendas detenciones corresponde con el **nº10 sucedido** por el **nº11**, producido por la **tercera** instrucción citada en el **análisis** previo de **riesgos** por **detención**. A esto se le suma que **temporalmente** se accede muy **pronto** a dicho registro, lo que creo no una sino **2 detenciones consecutivas**.

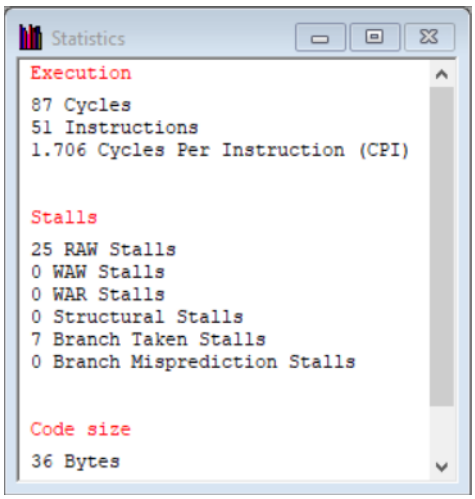


- **Observa que ocurre con la instrucción de salto. ¿En qué ciclo ocurre la primera parada por salto efectivo?**



Dicha parada se manifiesta en el **ciclo 13** de la **ejecución global** del programa.

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

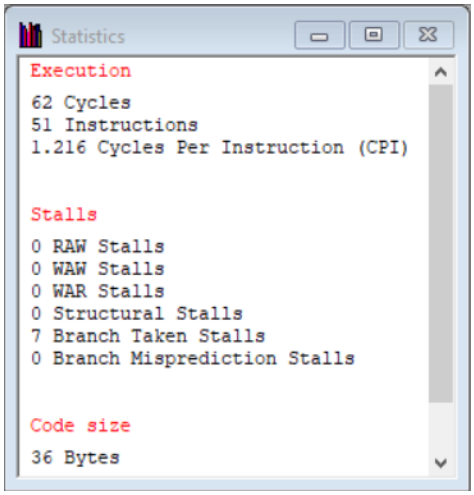


d) Ejecuta el programa en el simulador con la opción Configure/Enable Forwarding habilitada. Analizar ciclo a ciclo su funcionamiento con la opción "Single cycle" del menú Execute o bien presionando F7 sucesivamente. Examina las distintas ventanas que se muestran en el simulador y responder:

- **¿Cuántas dependencias de datos se han logrado solucionar?**

  ¿Cuántas paradas por riesgos de control se han producido?

  ¿Cuántos ciclos de reloj tarda el programa en ejecutarse? ¿Cuál es el CPI?



Como la solución a estas 3 preguntas se resuelven consultando nuevamente la pestaña **'Statistics'** he decidido **contestarlas** de forma **simultánea**.

En cuanto a las dependencias hemos conseguido **solucionarlas todas** al habilitar la opción **forwarding**, lo que podemos comprobar ya que todos los **Stalls** se encuentran a **0**, incluyendo los **riesgos** de **control**.

Finalmente, hemos obtenido un total de **62 ciclos** con un CPI de **1,2** ciclos por instrucción.

PROGRAMA 5

```
.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?**
  La mayor **diferencia** con respecto al **PROBLEMA 4**, es la **comparación** y sentencia de salto al **principio** del **bucle**, para que en caso de que el registro R2 llegue a valer 0, el flujo del programa se redirija a la etiqueta **'fin'.**

- **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?**

  Por cada instrucción **halt,** después de la instrucción de **salto incondicional**, producirá una **detención** por **riesgo** de **control** en todas las iteraciones **menos** en la **última** que sucederá en la instrucción **LD r3, datos(r1).**

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

	PROBLEMA 4	PROBLEMA 5
SIN FORWARDING	87 ciclos | 1,7 CPI | 25 RAW	98 ciclos | 1,6 CPI | 25 RAW
FORWARDING	62 ciclos | 1,2 CPI | 0 RAW	74 ciclos | 1,2 CPI | 1 RAW

- **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?**
  En mi caso para asegurarme que no afecta al resultado final, he decidido implementar dicha instrucción por debajo del salto incondicional **j loop.** Por ejemplo: **daddi r0, r0, 0.**
  Tras analizar ejecutar el programa, para mi sorpresa los resultados arrojados fueron los siguientes: **66 ciclos** y 6**1 instrucciones**, las cuales conforman un **CPI** de **1,09**. Por último, solo me arrojó **un Stall** de tipo **RAW** (Read after Write).

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

  Comparando los resultados expuestos en la tabla adjuntada en esta misma página, **no** podemos apreciar **diferencias notorias** entre ambos programas ya que la diferencia entre los **CPI** a mi parecer es prácticamente **nula**.

  Aunque, es cierto que **dependiendo** del tamaño del **problema** que nos propongan (por ejemplo, modificando el tamaño del vector) podría llegar a ser **crucial** dicha diferencia. De todas formas, aunque dicha diferencia sea pequeña, el **PROGRAMA 5** es el **óptimo** de los 2 ya que tiene **menos ciclos** acumulados en su **ejecución**.