PRÁCTICA 6

Segmentación de cauce y atascos en procesadores RISC

Objetivos *Comprender el funcionamiento de la segmentación de cauce del procesador MIPS de 64 bits. Analizar las ventajas e inconvenientes de este tipo de arquitectura.*

NOTA: Para simplificar los programas que veremos, en esta práctica en muchos ejercicios NO se utilizará la convención, y se utilizan directamente los registros r0…r31. No obstante, para resolver ejercicios de programación como los de las prácticas 4 y 5 siempre se requerirá usarla.

CPI

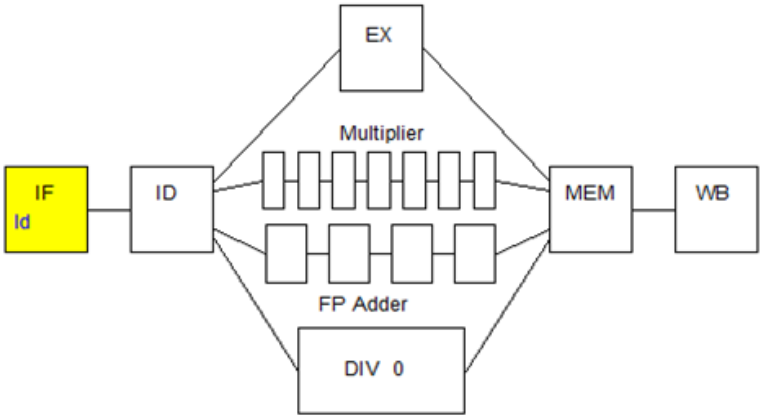
Los Ciclos Por Instrucción (CPI) son una medida de uso eficiente del procesador. Se definen como

$$\text{CPI} = \frac{Ciclos}{Instrucciones}$$

De esta forma, un procesador es más eficiente si para las mismas instrucciones, requiere menos ciclos. En ese caso, su CPI es más *bajo.*

Valor mínimo del CPI En arquitecturas tradicionales, el CPI no puede ser menor que 1, ya que en el mejor caso se termina una instrucción distinta en cada ciclo. En arquitecturas con múltiples unidades de ejecución si puede ser menor a 1.

La arquitectura del simulador WinMIPS tiene un pipeline de 5 etapas para las instrucciones más simples que usan la etapa EX, donde cada etapa requiere 1 ciclo para completarse si no hay atascos..



Por ese motivo, generalmente para un programa sin atascos el CPI puede calcularse como:

$$\text{CPI} = \frac{Ciclos}{Instrucciones} = \frac{Instrucciones + 4}{Instrucciones}$$

En este caso, el valor 4 proviene de que en los primeros 4 ciclos no se termina ninguna instrucción

Valor máximo del CPI En teoría, el CPI no tiene un límite superior, y podría ser tan grande como la ineficiencia del procesador. Idealmente, para un programa optimizado el CPI debería ser cercano a 1. Por ejemplo, un valor común para un programa optimizado sería 1.1 o 1.2. No obstante, esos valores de CPI generalmente solo se obtienen si un programa requiere ejecutar una gran cantidad de instrucciones que no se **atasquen** demasiado.

Atascos Las instrucciones pueden atascarse por diversos motivos, principalmente por falta de datos requeridos para ejecutarse. Cuando una instrucción se **atasca**, la misma no puede avanzar y por ende requiere más ciclos para ejecutarse, aumentando el CPI. En una arquitectura donde las etapas de las instrucciones duran 1 ciclo, entonces el número de ciclos requeridos para ejecutar la instrucción aumenta en 1 por cada atasco. En ese caso, para calcular el CPI podemos asumir:

$$\text{CPI} = \frac{Ciclos}{Instrucciones} = \frac{Instrucciones + 4}{Instrucciones} = \frac{Instrucciones + Atascos + 4}{Instrucciones}$$

Esta fórmula solo funciona cuando las instrucciones pasan todas por las mismas 5 unidades de ejecución básicas del WinMIPS64. A continuación, veremos los distintos tipos de atascos que pueden darse en la ejecución de los programas y cómo minimizarlos.

Parte 1: Estructura del Cauce

Para comprender cómo se puede atascar una instrucción, primero se debe comprender qué parte de la instrucción se ejecuta en cada etapa del cauce.

1. Funciones de las etapas del cauce⭐

Indicar a qué etapa corresponde cada función que realiza el procesador al ejecutar una instrucción típica:

Función	Etapa
A. Decidir si se toma un salto o no ID B. Buscar la instrucción en memoria y llevarla a la CPU IF C. Calcular la dirección de un acceso a memoria EX D. Guardar el valor de un registro en memoria MEM E. Traer de memoria un valor a un registro intermedio MEM F. Almacenar el valor final de un registro WB G. Calcular la dirección de un salto ID H. Verificar si están disponibles los operandos necesarios para continuar con la ejecución de la instrucción ID	1. IF: Obtención de Instrucción 2. ID: Decodificación de Instrucción 3. EX: Ejecución principal de instrucción 4. MEM: Acceso a Memoria 5. WB: Escritura de resultado en registro destino

2. Etapas de una instrucción ⭐

Indicar, para las siguientes instrucciones, qué realizan en cada una de las etapas IF/ID/EX/MEM/WB.

Instrucción	IF	ID	EX	MEM	WB
`daddi r1, r3, 4`	Buscar la instrucción en memoria y llevarla a la CPU	Decodifica la instrucción. Verifica si está disponible el registro **r3**. Recupera **r3** del banco de registros..	Realiza el calculo de **r3** + 4	-	Guarda en **r1** el resultado de la suma de **r3** y 4
`dadd r2, r4, r3`	Buscar la instrucción en memoria y llevarla a la CPU	Decodifica la instrucción. Verifica si están disponibles los operandos **r4** y **r3**. Recupera **r4** y **r3** del banco de registros.	Realiza el calculo de **r4** + **r3**	-	Guarda en **r2** el resultado de la suma de **r4** y **r3**
`sd r3, tabla(r2)`	Buscar la instrucción en memoria y llevarla a la CPU	Decodifica la instrucción. Verifica si están disponibles el operando **r3**	Realiza la suma de tabla + r2, que es la dirección a la que se va a acceder a	Accede a la posición de memoria calculada y guarda el valor de r3	-

		y **r2**. Recupera **r3** y **r2** del banco de registros.	memoria	ahí.	
`ld r3, tabla(r2)`	Buscar la instrucción en memoria y llevarla a la CPU	Decodifica la instrucción. Verifica si está disponible el operando **r2**. Recupera **r2** del banco de registros.	Realiza la suma de tabla + r2, que es la dirección a la que se va a acceder a memoria	Accede a la posición de memoria calculada y recupera el valor.	Se guarda el valor de memoria leido en **r3**.
`bneq r1, r2, loop`	Buscar la instrucción en memoria y llevarla a la CPU	Decodifica la instrucción. Verifica si están disponibles el registro **r1** y **r2**. Recupera **r1** y **r2** del banco de registros. Si **r1** y **r2** son distintos se calcula el nuevo valor de PC.	-	-	-
`j loop`	Buscar la instrucción en memoria y llevarla a la CPU	Decodifica la instrucción. Calcula el nuevo valor de PC.	-	-	-

3. Cálculo de CPI ⭐

Simular la ejecución de los siguientes programas **manualmente** (sin usar el simulador), dibujando el cauce como lo hace el simulador, y calculando la cantidad de instrucciones, ciclos, y CPIs.

Programa A	**Programa B**

`daddi r2,r3,5`
`dsub r4,r3,r5`
`xor r6,r3,r5`
`nop`
`halt`
	`ddiv r6,r3,r5`
	`dmul r4,r3,r5`
	`daddi r2,r3,5`
	`halt`

Simulación Programa A

Ciclo	1	2	3	4	5	6	7	8	9
daddi r2,r3,5	IF	ID	EX	ME	WB				
dsub r4,r3,r5		IF	ID	EX	ME	WB			
xor r6,r3,r5			IF	ID	EX	ME	WB		
nop				IF	ID	EX	ME	WB	
halt					IF	ID	EX	ME	WB

$$\text{CPI} = \frac{9\ Ciclos}{5\ Instrucciones} = 1,8 \frac{Ciclos}{Instrucción}$$

Simulación Programa B

Ciclo	1	2	3	4	5	6	7	8	9	11	11	12	13	14	15	16	17
ddiv r6,r3,r5	IF	ID	DIV														
dmul r4,r3,r5		IF	ID	M0	M1	M2	M3	M4	M5	M6	ME	WB					
daddi r2,r3,5			IF	ID	EX	ME	WB										
halt				IF	ID	EX	ME	WB									

Ciclo	18	19	20	21	22	23	24	25	26	27	28
ddiv r6,r3,r5										ME	WB
dmul r4,r3,r5											
daddi r2,r3,5											
halt											

$$\text{CPI} = \frac{28\ Ciclos}{4\ Instrucciones} = 7 \frac{Ciclos}{Instrucción}$$

Responder:

1. La fórmula de CPI presentada anteriormente ¿se cumple para los dos programas?

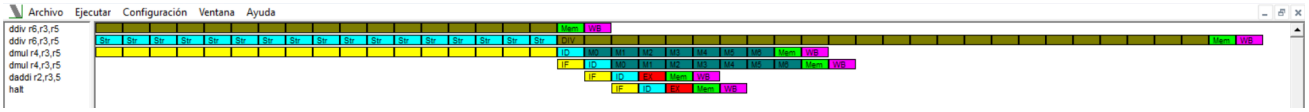
Para el programa A si se cumple. En el caso del programa B no se cumple ya que usa instrucciones que usan unidades de ejecución diferentes que tardan más ciclos de instrucción que la etapa EX.

2. En el segundo programa, se utilizan las instrucciones ddiv y dmul ¿cuántos se requieren para su ejecución?

La instrucción *ddiv* tarda 28 ciclos en realizar la ejecución completa de la instrucción (desde la etapa IF hasta WB). La etapa de ejecución propia es una sola etapa que tarda 24 ciclos.
La instrucción *dmul* tarda 11 ciclos en realizar la ejecución completa de la instrucción. (desde la etapa IF hasta WB). La multiplicación se realiza en 7 etapas de 1 ciclo cada una.

3. ¿Por qué la instrucción dmul tiene varias etapas llamadas M1, M2, etc, y ddiv no? Modificar el segundo programa, duplicando las instrucciones ddiv y dmul, de manera que cada una aparezca dos veces, y volver a simular. ¿Qué sucede en el caso de la división?

Si duplicamos las instrucciones *ddiv* y *dmul* la simulación nos da:



Lo que sucede con la segunda instrucción *ddiv* es que se "atasca" a la espera de que la etapa de división termine para poder ingresar. Cuando un atasco ocurre porque una etapa que una instrucción necesita acceder no esta disponible porque otra instrucción esta pasando por esa etapa, se llama **atasco estructural**.

En el caso de *dmul* no ocurre lo mismo porque la división ocurre en etapas diferentes, por lo tanto mientras la primer instrucción va para una etapa de la multiplicación, la segunda instrucción *dmul* puede utilizar cualquier etapa previa de la multiplicación.

4. La instrucción HALT solo detiene la ejecución del programa ¿se cuenta en el cálculo del CPI?

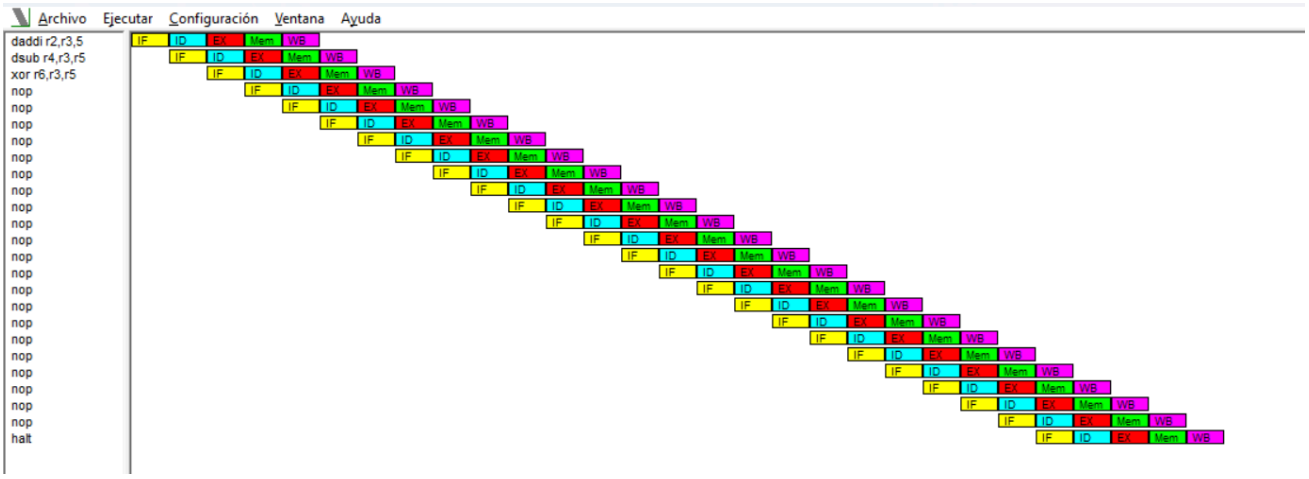
Si, la instrucción se cuenta para el calculo del CPI.

5. El simulador considera a la instrucción NOP para calcular el CPI. No obstante, dicha instrucción no realiza ninguna tarea. Agregar 20 instrucciones NOP más en el primer programa, y calcular nuevamente el CPI. ¿Qué valor toma? En el caso de que sea menor, ¿eso quiere decir que el programa es más eficiente?

$$\text{CPI} = \frac{29\ Ciclos}{25\ Instrucciones} = 1,16 \frac{Ciclos}{Instrucción}$$

Esto puede resultar engañoso, desde el punto de vista de ejecución de la CPU el valor es menor lo que implica que la CPU tardó en promedio menos ciclos para ejecutar cada instrucción, lo que haría pensar que mejoro la eficiencia. El tema es que esta versión del programa realiza las mismas tareas que el anterior pero tarda 20 ciclos más.

Es importante notar que a medida que agreguemos instrucciones *NOP* al programa, los CPI van a ir tendiendo al valor 1, ya que como el calculo sería $\frac{Instrucciones + 4}{Instrucciones}$ a medida que la cantidad de instrucciones crece el 4 se hace cada vez menos significativo.



Parte 2: Atascos RAW

1. Dependencia de datos ⭐

Los **atascos RAW** son los más comunes en un programa. Para determinar si hay un atasco RAW entre dos instrucciones, primero hay que determinar si hay una **dependencia de datos de lectura** entre las mismas.

Las dependencia de datos de lectura ocurren cuando una instrucción B requiere el valor de un registro, pero debe esperar a que otra instrucción A escriba el valor de ese registro. En este caso, B es una instrucción que comienza a ejecutarse luego de A.

La dependencia de datos no siempre implica un atasco, porque si las instrucciones están muy alejadas en el tiempo la primera termina antes de que la segunda necesite el dato.

a) Los siguientes programas cortos contienen instrucciones que utilizan registros similares. Indicar en cada caso qué instrucciones tienen dependencias de datos de lectura entre ellas.

	1
1	`daddi r1,r0,5`
2	`daddi r2,r0, 7`
3	`slt r3, r1, r2`
4	`daddi r1,r0,1`
5	`and r4, r3, r1`
6	`daddi r1, r0, 8`
7	`sd r4, A(r1)`

Ejemplo con el programa 1:
3 depende de 1 (por r1) y 2 (por r2)
5 depende de 3 (por r3) y 4 (por r1)
7 depende de 5 (por r4) y 6 (por r1)

	2
1	`ld r1, A(r0)`
2	`ld r2, B(r0)`
3	`bne r1, r2, no`
4	`daddi r3,r0,1`
5	`j fin`
6	`no: daddi r3,r0, 0`
7	`fin: sd r3, C(r0)`

Dependencias de datos el programa 2:
3 depende de 1 (por r1) y 2 (por r2)
7 depende de 5 (si r1 y r2 son iguales) o 6 (si r1 y r2 son iguales)

	3
1	`daddi r1,r0,4`
2	`daddi r2,r0,3`
3	`daddi r3,r0,0`
4	`loop: dadd r3,r3,r2`
5	`daddi r1,r1,-1`
6	`benz r1, loop`
7	`sd r3, res(r0)`

Dependencias de datos el programa 3:

4 depende de 2 (por r2) y 3 (por r3) y 4 (por r3) en cada vuelta del loop
5 depende de 1 (por r1) y de 5 en cada vuelta del loop (por r1)
6 depende de 5 (por r1)
7 depende de 4 (por r3)

	4
1	`daddi r1,r0,0`
2	`daddi r2,r0,0`
3	`loop: ld r3,A(r1)`
4	`dadd r2,r2,r3`
5	`daddi r1,r1,8`
6	`bnez r3, loop`
7	`sd r2, RES(r0)`

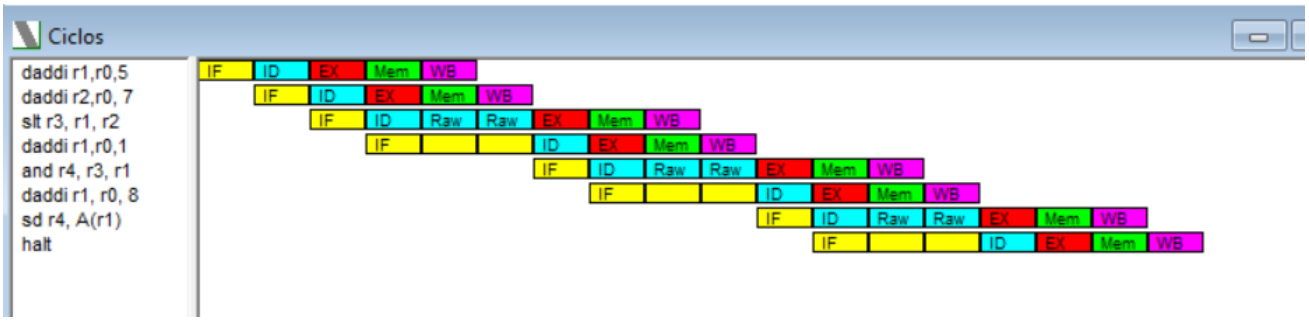
Dependencias de datos el programa 4:

3 depende de 1 (por r1) y 5 (por r1) en cada vuelta del loop
4 depende de 3 (por r3) y 2 (por r2) y 4 en cada vuelta del loop
5 depende de 1 (por r1) y de 5 (por r1) en cada vuelta del loop
6 depende de 3 (por r3)
7 depende de 4 (por r2)

b) ¿Cuáles de las dependencias de datos por lectura te parece que causarán atascos RAW? Probar los programas en el simulador y anotar la cantidad de atascos y CPI de cada uno. **Nota**: Ignorar los atascos por "Branch Taken Stall" (los veremos más adelante).

No todas las dependencias de datos van a generar necesariamente atascos. Las instrucciones cercanas generan atascos. Si las instrucciones están distanciadas en ejecución no los van a generar.

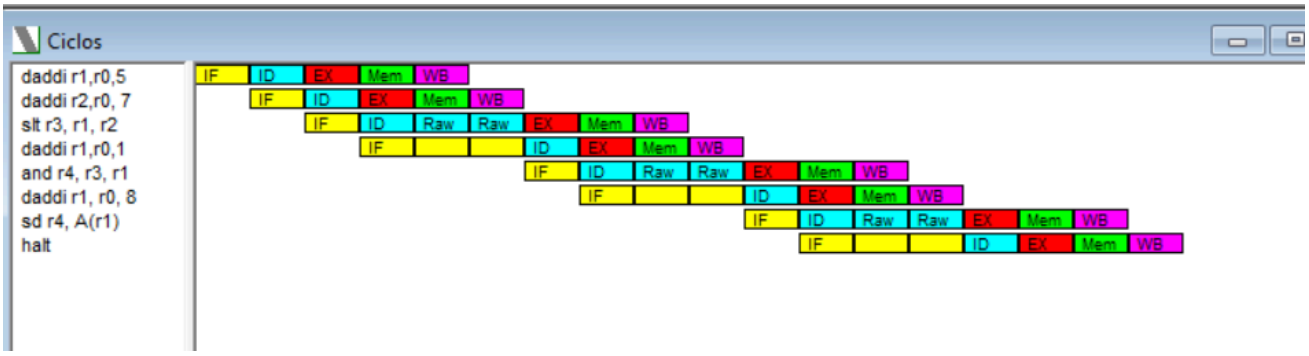
Programa 1



$CPI = \frac{18}{8} = 2,25$

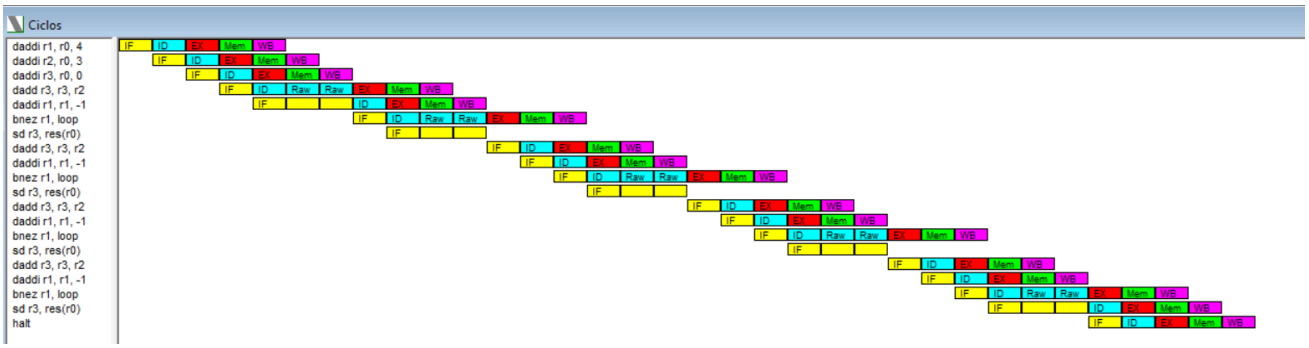
Programa 2

Con A = 2 y B = 3



$$\text{CPI} = \frac{18}{8} = 2,25$$

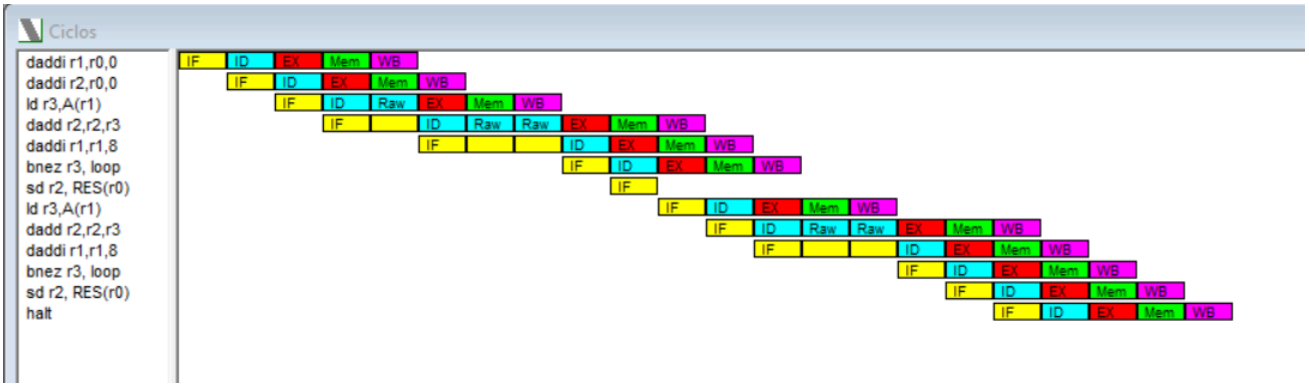
Programa 3



$$\text{CPI} = \frac{34}{17} = 2$$

Programa 4

Con A = 2



$$\text{CPI} = \frac{22}{12} = 1,833$$

c) Modificar los programas para que se reduzca la cantidad de atascos RAW, reordenando las instrucciones de forma que el resultado final del programa sea el mismo. Comparar la cantidad de atascos y CPI de cada uno con el caso anterior.

Los programas 1 y 2 no se pueden reordenar las instrucciones para evitar atascos.

En los programas 3 y 4 se puede reordenar para no tener tantos atascos RAW

	1	2	3	4
1	`daddi r1,r0,5`	`ld r1, A(r0)`	`daddi r1,r0,4`	`daddi r1,r0,0`
2	`daddi r2,r0, 7`	`ld r2, B(r0)`	`daddi r2,r0,3`	`daddi r2,r0,0`
3	`slt r3, r1, r2`	`bne r1, r2, no`	`daddi r3,r0,0`	`loop: ld r3,A(r1)`
4	`daddi r1,r0,1`	`daddi r3,r0,1`	`loop: daddi r1,r1,-1`	`daddi r1,r1,8`
5	`and r4, r3, r1`	`j fin`	`dadd r3,r3,r2`	`dadd r2,r2,r3`
6	`daddi r1, r0,8`	`no: daddi r3,r0, 0`	`bnez r1, loop`	`bnez r3, loop`
7	`sd r4, A(r1)`	`fin: sd r3, C(r0)`	`sd r3, res(r0)`	`sd r2, RES(r0)`

2) Atascos RAW y forwarding ⭐

En el ejercicio previo, vimos que reordenando las instrucciones se puede obtener una mejora en el CPI del programa. Otra forma de solucionar los atascos por dependencia de datos es utilizando el Adelantamiento de Operandos o Forwarding. Un procesador con Forwarding tiene un hardware modificado que permite que menos dependencias de datos se conviertan en atascos RAW. Para lograrlo, utiliza dos estrategias complementarias:

1) El valor que se calcula en las etapas EX o MEM está disponible para que otras instrucciones lo accedan ni bien se calcula, y no se requiere esperar a la etapa WB. Por ejemplo, la instrucción **daddi r1,r2,5** tendrá disponible el resultado de la suma al finalizar la etapa **EX**.

2) Las instrucciones no requieren todos sus operandos en la etapa **ID**. En lugar de eso, si necesitan un operando y no está disponible, se van a atascar en la etapa en que lo necesiten realmente. Por ejemplo, si la instrucción **sd r1, A(r0)** llega a la etapa **ID** pero el valor de r1 todavía está siendo calculado por otra instrucción, avanzará igual y se atascará recién en la etapa **MEM.**

Para analizar el efecto del forwarding, veamos el siguiente programa que intercambia el contenido de dos palabras de la memoria de datos, etiquetadas A y B.

```
                .data
A:              .word  1
B:              .word  2
                .code
                ld      r1, A(r0)
                ld      r2, B(r0)
                sd      r2, A(r0)
                sd      r1, B(r0)
                halt
```

a) Ejecutarlo en el simulador con la opción Configure/Enable Forwarding **deshabilitada**.
¿Cuántos atascos RAW hay? Hay 2 atascos RAW en el sd de r2, esperando la lectura de r2.
¿Cuál es el CPI? 11/5 = 2,2

b) Ejecutarlo en el simulador con la opción Configure/Enable Forwarding **habilitada**.
¿Por qué no se presenta ningún atasco en este caso? El atasco ya no ocurre porque el valor de r2 es leido de memoria luego de la etapa MEM de la lectura de r2 y disponible para la instrucción sd en la etapa de MEM que es donde lo necesita.

¿Cuál es el CPI? 9/5 = 1,8

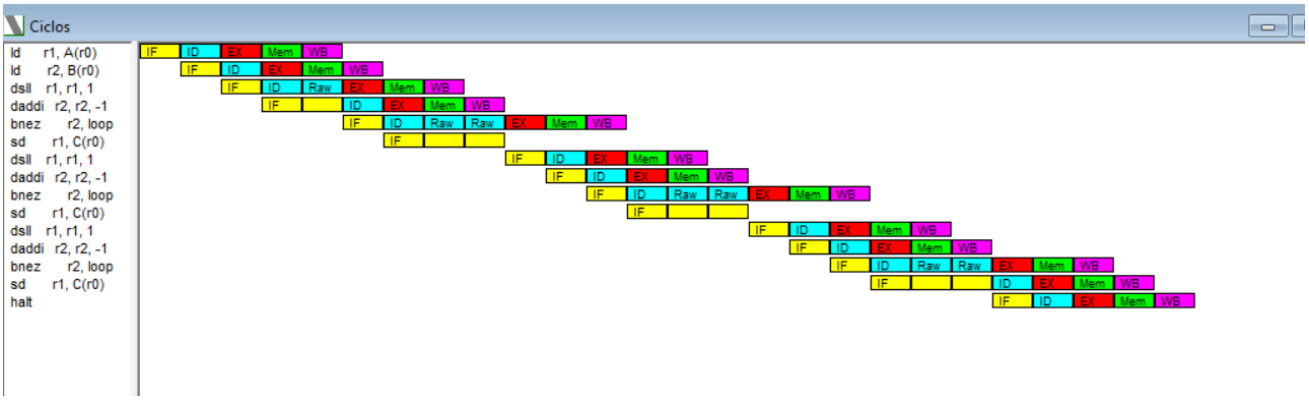
¿Qué indica el color de los registros en la ventana Register durante la ejecución? Que ese registro tiene un valor actualizado y puede ser utilizado de forma adelantada. El color depende de la etapa donde se genero el valor (MEM o EX)

3) Atascos RAW con lazos ⭐⭐

Ignorando por ahora los atascos por salto (Branch Taken Stall), analizar el siguiente programa:

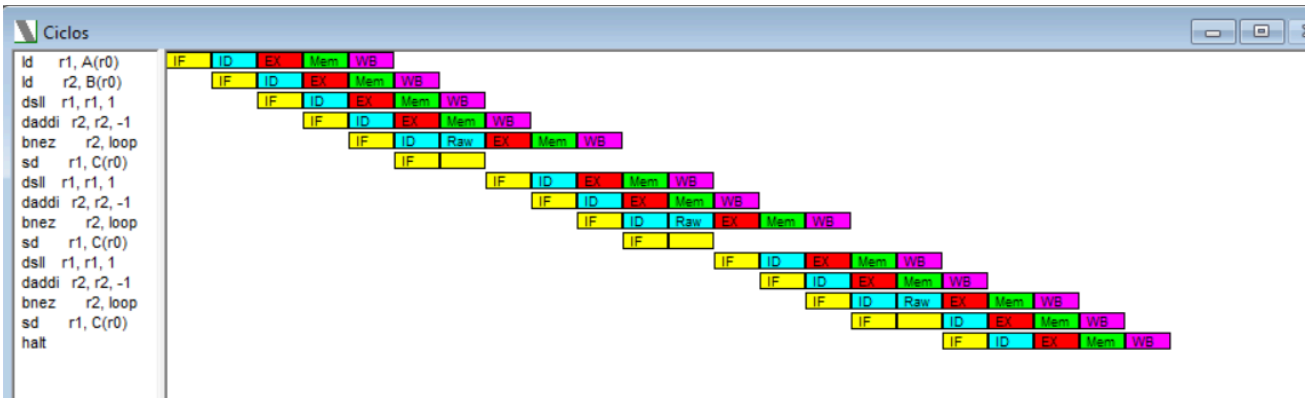
```
.data
A:              .word 1
B:              .word 3
C:              .word 0
.code
        ld      r1, A(r0)
        ld      r2, B(r0)
loop:   dsll    r1, r1, 1
        daddi   r2, r2, -1
        bnez    r2, loop
        sd      r1, C(r0)
        halt
```

a) **Loop sin forwarding** Ejecutar el programa deshabilitando el Forwarding y responder:



- ¿Qué instrucciones generan los atascos tipo RAW y por qué? ¿En qué etapa del cauce se produce el atasco en cada caso y durante cuántos ciclos?
    - El atasco lo generan las instrucciones del salto condicional y son 2 ciclos por cada ejecución de la instrucción..
- ¿Cuántos CPI tiene la ejecución del programa en este caso? 26/13 = 2
- Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de CPIs? No, se mantiene igual.

b) **Loop con forwarding** Ejecutar el programa con Forwarding habilitado y responder:



- ¿Cuántos CPI tiene la ejecución de este programa? Tomar nota del número de ciclos, cantidad de instrucciones y CPI. Comparar con el caso anterior .
    - El CPI ahora es 22/13 = 1,692. Bajo. Ahora el atasco es uno solo por cada ejecución de la instrucción de salto condicional.
- Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de CPIs?
    - CPI ahora es 1.667. Si, si bien la cantidad de atascos por dependencia de datos son menos, ahora aparecen atascos por los saltos tomados

c) **Reordenamiento de instrucciones para optimización de CPI:**

- Reordenar las instrucciones para que la cantidad de RAW sea 0 en la ejecución del programa, ejecutando con Forwarding **habilitado** y B=3.

CPI ahora es: 1.462

```
.data
A:              .word 1
B:              .word 3
C:              .word 0

.code
        ld      r2, B(r0)
        ld      r1, A(r0)
loop:   daddi   r2, r2, -1
        dsll    r1, r1, 1
        bnez    r2, loop
        sd      r1, C(r0)
        halt
```

- Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de CPIs?
    - Si. Bajo 1.334. Si ponemos un valor más grande en B va a ir bajando
- En base a lo anterior ¿qué partes del programa conviene optimizar generalmente?
    - En general hay que poner más atención en optimizar las partes que más tardan en ejecutarse del programa. Cuando hay ciclos y/o repeticiones es importante ver que sean óptimos.

d) Cantidad de instrucciones de un lazo

- Viendo el código del programa de forma estática (es decir, sin ejecutarlo), la cantidad de instrucciones que tiene sería 7. No obstante, las estadísticas del simulador indican que hay 13 instrucciones ¿de dónde sale esta diferencia? ¿Cómo calcularías la cantidad de instrucciones de un programa con un lazo?

- La cantidad de instrucciones que muestra en las estadísticas el simulador son las instrucciones que se ejecutaron, no la cantidad de instrucciones que tiene el programa. Salvo en programas muy simples la cantidad de instrucciones del programa coincide con la cantidad de instrucciones que el simulador ejecuta.

Parte 3: Atascos por dependencias de control

1) Atascos por salto (BTS) ⭐

Ejecutar el programa del ejercicio 3 de la Segunda Parte, reordenado y con Forwarding, de modo que no haya atascos RAW. El programa sigue teniendo algunos atascos, llamados Branch Taken Stall (BTS), también conocidos como atascos por salto.

- Con B = 3. Ejecutando el programa tenemos 19 ciclos, 13 instrucciones ejecutadas, CPI 1.46 y 2 atascos por BTS.

1. Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de BTS? ¿y el CPI?
   - Con B=1000. 4007 ciclos, 3004 instrucciones, el CPI 1.334. 999 atascos por BTS.

2. Completar la siguiente frase: Al ejecutar un loop simple con N iteraciones, se producen **(N-1)** atascos BTS.
3. ¿Por qué se producen los BTS? ¿Qué sucede con la instrucción siguiente al salto?
   - Los BTS se producen cuando tenemos un salto en la ejecución del programa. La CPU sabe que una instrucción es de salto cuando se decodifica, por lo tanto hay que descartar la instrucción que se capto en la etapa IF que no se va a ejecutar (porque se va a ejecutar la instrucción correspondiente al salto).

4. ¿Pueden evitarse los BTS reordenando instrucciones?
   - Los BTS no se producen por una dependencia de datos sino por haber empezado a procesar (en etapa IF) una instrucción que no se va a ejecutar.

5. Al ocurrir un BTS, una instrucción se empieza a ejecutar y luego se descarta, pero ¿debe contarse como una instrucción para el cálculo del CPI?
   - Las instrucciones descartadas no cuentan para las estadísticas. Si la instrucción se ejecuta en otro momento si se la contara, pero solo en el momento que se ejecute efectivamente.

2) Reducción de BTS con Branch Target Buffer ⭐

Habilitar la opción Branch Target Buffer (BTB) y volver a ejecutar el programa anterior con B=3.

1. ¿Cómo cambió la cantidad de BTS? ¿y el CPI?
   - 21 ciclos, 13 instrucciones. CPI = 1,615. Hay 2 atascos BTS, pero ahora tenemos 2 atascos BMS.

2. Notar que ahora aparece un nuevo tipo de atasco, llamado Branch Misprediction Stall (BMS). ¿Por qué sucede? ¿Cuántos ocurren?
   - Ahora tenemos 2 atascos BMS que antes no estaban. Estos atascos ocurren cuando hay una mala predicción del salto. Si se predijo que el salto iba a ocurrir y no ocurre, o se predice que no va a ocurrir y ocurre, tenemos ese atasco.

3. Cambiar el valor de B a 1000, habilitar BTB y ejecutar. ¿Cómo cambió la cantidad de BTS y BMS? ¿y el CPI?
   - La cantidad de BTS y BMS no cambio, el CPI por consiguiente se redujo (tenemos mas instrucciones ejecutadas para la misma cantidad de atascos). El CPI ahora es 1.003.

4. Completar la siguiente frase: Al ejecutar un loop simple con N iteraciones, si se habilita BTB, se producen **2** atascos de tipo BTS y **2** atascos de tipo BMS.

5. En base a los resultados anteriores ¿es mejor utilizar BTB cuando se realizan pocas o cuando se realizan muchas iteraciones?
   - BTB nos beneficia más cuando se realizan más iteraciones.

3) Utilidad del BTB en distintos casos ⭐

El BTB puede aumentar el desempeño significativamente en la mayoría de los lazos. No obstante, en algunos puede tener un comportamiento patológico y de hecho reducir la eficiencia del programa. El siguiente programa calcula el máximo de un vector.

```
.data
A:   .word 2,1,3,1,4,1
MAX: .word -1

.code
      ld r1, MAX(r0)
      daddi r2,r0,0
      daddi r3,r0,6
loop: ld r4, A(r2)
      slt r5,r1,r4
      beqz r5, chico
      daddi r1,r4,0
chico:daddi r2,r2,8
      daddi r3, r3, -1
      bnez r3, loop
      sd r1, MAX(r0)
      halt
```

a) Antes de ejecutar en el simulador, encontrar las instrucciones de salto, y pensar cómo se comportará el BTB en cada caso. Recordar que la predicción del BTB guarda un bit de historia distinto por cada instrucción de salto.
   - Tenemos 2 saltos condicionales, el primero (beqz r5, chico) depende. El segundo salto condicional (bnez r3, loop) depende dela cantidad de elementos del arreglo. El ultimo salto condicional va a funcionar como vimos en el punto anterior con BTB activo. El segundo va a depender de los elementos.

b) Ejecutar el programa en el simulador con y sin BTB. ¿Qué programa es más eficiente?
   - Sin BTB activado: 74 ciclos, 44 instrucciones, CPI: 1,682 (8 BTS, 18 RAWS, 6 STR)
   - Con BTB desactivado: 80 ciclos, 44 instrucciones, CPI: 1,818 (8 BTS, 6 BTM, 18 RAWS, 6 STR)
- La eficiencia mejora cuando desactivamos BTB en este caso particular.

4) Reducción de BTS con Delay Slot (DS) ⭐⭐

El delay slot (DS) cambia el funcionamiento de los saltos. Cuando está activado, el salto se realiza con un retardo de un ciclo. Esto significa que la instrucción **siguiente** al salto también se ejecuta. De esta forma, los BTS desaparecen completamente, aunque esto no significa necesariamente que la ejecución sea más eficiente.

a) Anotar en la tabla de abajo las estadísticas del programa del ejercicio 3 de la Segunda Parte al ejecutarse sin forwarding, y con **BTB** y **delay slot** desactivados
b) Ejecutar el programa del ejercicio 3 de la Segunda Parte, pero ahora con BTB activado y forwarding desactivado.
c) Ejecutar el programa del ejercicio 3 de la Segunda Parte, pero ahora con **delay slot** activado, BTB desactivado y forwarding activado. Anotar los CPI y la cantidad de instrucciones en la tabla de abajo. ¿Qué sucede con la instrucción **sd r1, C(r0)**? ¿Cuántas veces se ejecuta? ¿Son necesarias todas las ejecuciones?

- Con delay slot activo, se capta en cada vuelta de los ciclos la instrucción **sd r1, C(r0)** por lo que se va a ejecutar 3 veces. Como el programa va calculando el desplazamiento, no tiene sentido ir guardando en memoria los valores parciales de cálculo.

d) Modificar el programa del ejercicio 3 de la Segunda Parte agregando un NOP antes de la instrucción **sd r1, C(r0)**. Ejecutar el programa en el simulador y anotar los CPI. Los CPI serán menores que en el caso anterior
e) Modificar el programa del ejercicio 3 de la Segunda Parte, pero ahora reordenando las instrucciones de modo de ejecutar una instrucción **útil** debajo del salto, sin aumentar el número total de instrucciones a ejecutar.

Ejercicio 3	Sin mejoras	BTB	DS	DS + NOP	DS + Reordenamiento
BTS	2	2	0	0	0
BMS	0	2	0	0	0
CPI	2,000	2,154	1,467	1,438	1,615
#Instrucciones	13	13	15	16	13

5) Corrección de programas con Delay Slot ⭐⭐⭐

En el programa del ejercicio 3 de la Segunda Parte, habilitar DS hace que una instrucción se ejecute varias veces. Si bien el programa es más ineficiente, el resultado final es el mismo. No obstante, en otros casos habilitar DS puede hacer que el programa no funcione correctamente.

a) Correr el programa del ejercicio 3 de esta parte pero ahora con DS habilitado y anotar el CPI. Notar el resultado almacenado en la variable MAX ¿Por qué es incorrecto?

CPI = 1,423 - Ciclos 74 - Instrucciones 52

```
(01)       ld         r1,MAX(r0)
(02)       daddi      r2,r0,0
(03)       daddi      r3,r0,6
(04)loop:  ld         r4, A(r2)
(05)       slt        r5,r1,r4
(06)       beqz       r5, chico
(07)       daddi      r1,r4,0
(08)chico: daddi      r2,r2,8
(09)       daddi      r3, r3, -1
(10)       bnez       r3, loop
(11)       sd         r1, MAX(r0)
(12)       halt
```

Al ejecutarse la instrucción de salto condicional (06) la siguiente instrucción se ejecuta siempre y esa instrucción es la asigna a r1 el valor actual, que es el valor que se calcula como el máximo.

b) Reordenar las instrucciones del programa para que funcione correctamente, y además la instrucción **sd r1, MAX(r0)** solo se ejecute una vez, sin agregar instrucciones a ejecutar. Anotar el CPI, y comparar con el caso anterior ¿qué mejora se obtuvo?

CPI = 1,432 - Ciclos 63 - 44 Instrucciones

Se redujo la cantidad de ciclos de 74 a 63. Y la CPU ejecuta 8 instrucciones menos.

```
.data
A:   .word 2,1,3,1,4,1
MAX: .word -1

.code
       ld r1, MAX(r0)
       daddi r2,r0,0
       daddi r3,r0,6
loop:  ld r4, A(r2)
       slt r5,r1,r4
       beqz r5, chico
       daddi  r3, r3, -1
       daddi r1,r4,0
chico: bnez   r3, loop
       daddi r2,r2,8
       sd r1, MAX(r0)
       halt
```

Parte 3: Atascos por WAR, WAW y estructurales.

La etapa EX usa la ALU de sumas y restas de punto fijo. En un programa que solo utiliza instrucciones que pasan por la etapa EX, las instrucciones no se pueden *sobrepasar*, ya que todas tardan el mismo tiempo y pasan por las mismas etapas. En los procesadores modernos, no obstante, existen distintas ALUs para las operaciones de suma/resta (EX), multiplicación (MUL) y división (DIV). Además, las instrucciones de multiplicación y división tardan más ciclos que las de suma. Por este motivo, las instrucciones de suma pueden **comenzar después** que las de multiplicación o división, y **terminar antes,** generando efectivamente una **ejecución fuera de orden** (out of order execution).
Este tipo de ejecución abre la posibilidad a tres nuevos tipos de atascos: WAR (Write After Read, o **atascar la escritura para que termine la lectura**), WAW (Write After Write, o **atascar la escritura para que termine la otra escritura**) y STR (estructurales, o **atascar una instrucción porque la siguiente etapa o estructura del procesador está siendo utilizada**).

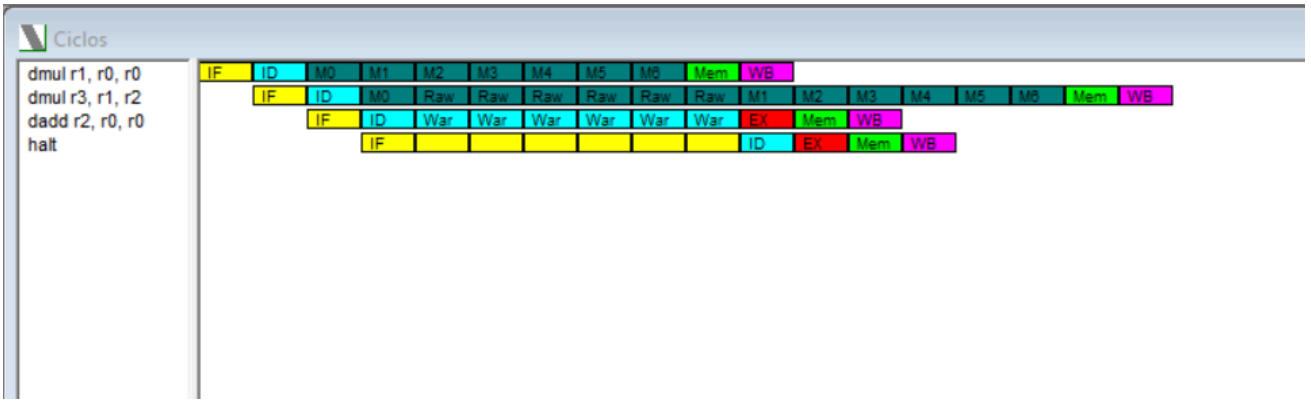
1) Atascos WAR y WAW ⭐

Los atascos WAR y WAW son la contracara de los RAW. Si bien es mucho más difícil que se produzcan en un programa común, son posibilidades que debe tener en cuenta el procesador. Estos atascos suceden cuando una **instrucción más rápida** se adelante a una **instrucción más lenta**. Los siguientes programas presentan ejemplos minimalistas de tipo de atascos. Responder:

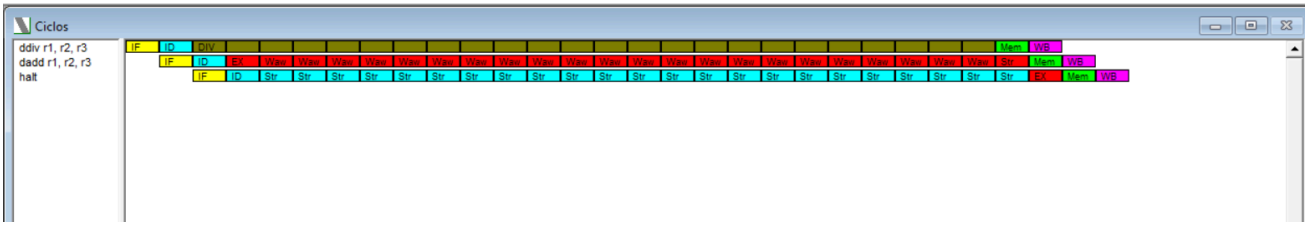
1. Estudiar el código sin simularlo y responder ¿cuál es el programa que tiene WAR y cual WAW? Simular los programas para comprobar. Usar la opción forwarding activa.
   - El programa B tiene atascos WAR, la dependencia está en el registro r2, la instrucción dadd debe esperar a que la segunda instrucción dmul lea el valor.
   - El programa A tiene atascos WAW, las 2 instrucciones escriben el valor de r1, y ddiv es más lenta, por lo que si no se atasca el registro r1 terminaría con el valor de la división y no la suma.

2. En el caso del WAR, ¿cuál es la instrucción lenta y cuál la rápida? ¿Qué registro se quiere leer y escribir?

- La instrucción más lenta es *dmul* (segunda) y la rápida es *dadd*.
- El registro que genera el atasco es *r2*, que *dadd* escribe y *dmul* leer.



3. Idem para el caso WAW.

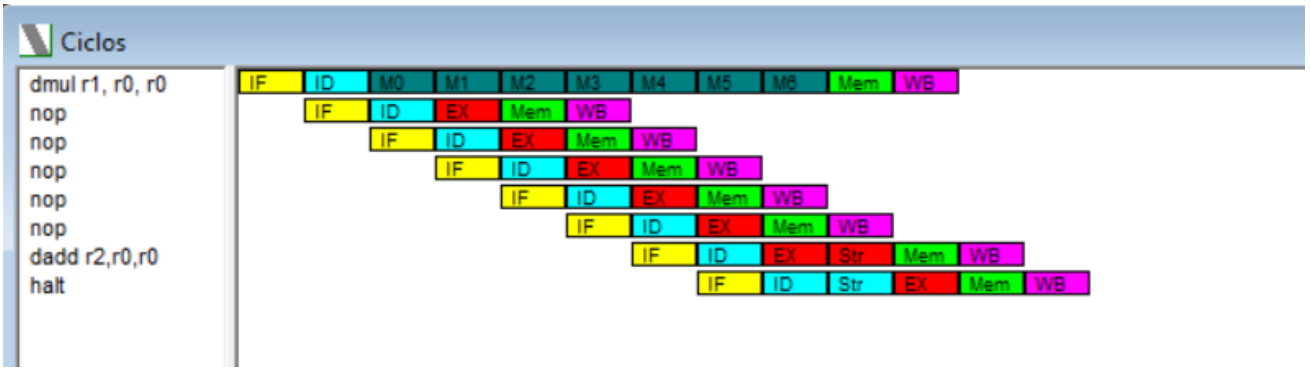


Programa A	**Programa B**
.code ddiv r1, r2, r3 dadd r1, r2, r3 halt	**.code** dmul r1, r0, r0 dmul r3, r1, r2 dadd r2, r0, r0 halt

2) Atascos estructurales (STR) ⭐

La ejecución fuera de orden también permite otro tipo de atasco. Los atascos estructurales suceden cuando dos instrucciones quieren acceder al mismo tiempo a la misma etapa del cauce. En el simulador, esto sucede en la etapa MEM, ya que sin importar la ALU que usen las instrucciones, luego del cálculo siempre deben pasar por la etapa MEM. Como las instrucciones rápidas pueden sobrepasar a las lentas, puede suceder que dos o más instrucciones terminen su etapa de ejecución al mismo tiempo, y por ende también quieran pasar a MEM al mismo tiempo.

.code dmul r1, r0, r0 nop nop nop	nop nop dadd r2,r0,r0 halt



a) Ejecutar el código anterior y verificar que ocurre un atasco STR. ¿Entre qué instrucciones sucede el atasco? ¿cuál es la instrucción que se atasca? ¿Por qué esa y no la otra?
   - El atasco se da entre el *dmul r1,r0, r0* y *dadd r2, r0, r0*. La instrucción que se atasca es la más nueva (*dadd*), esto se hace para preservar el orden de ejecución.

b) Probar agregando un NOP ¿sigue el atasco? ¿y si se quita un NOP? ¿por qué?
   - Si quito o agrego un NOP el atasco estructural va a seguir estando, pero en el NOP o en el halt respectivamente. Esto ocurre porque la instrucción de dmul que se ejecuta en mas ciclos necesita acceder a la etapa de MEM y la instrucción con la que se atasca también. Si eliminamos un nop más, el atasco no sucede.

3) Análisis de atascos ⭐⭐

Los siguientes programas presentan ejemplos naturales de atascos estructurales y WAR. Analizar e identificar dónde pueden ocurrir estos atascos. Ejecutar en el simulador y comprobar el resultado.

```
; Resto: Calcula en r4 el resto de
r1 div r2
.code
daddi r1,r0,30 ; a = 30
daddi r2,r0,4 ; b = 4
ddiv  r3,r1,r2 ; c = a div b = 7
dmul r3, r3, r3 ; c*b = 7*4 = 28
dsub r4, r1,r3 ; resto = a-c*b = 2
halt
```

```
; factorial: Calcula en r2 el
factorial de r1
   .code
        daddi r1,r0,5 ; n=5
        daddi r2,r0,1 ; f=1
   loop: dmul r2,r2,r1 ; f=f*n
        daddi r1,r1,-1  ; n=n-1
        bnez r1, loop
        halt
```

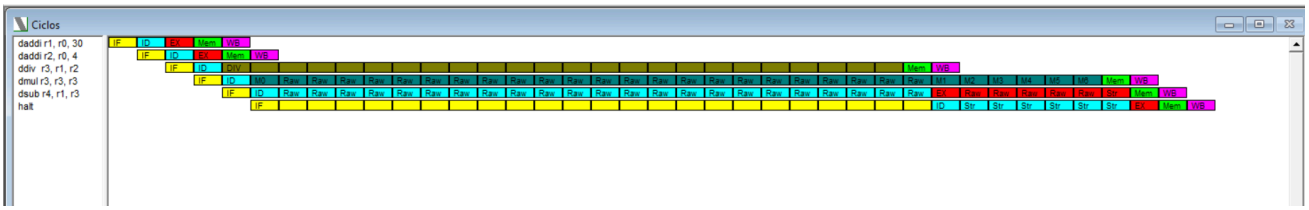
Programa Resto

En este programa tenemos 2 dependencias de datos:

- El primero es entre la instrucción *dmul* y *ddiv*, por el registro r3, *dmul* tiene que esperar que se termine de calcular la división para tener acceso al registro.
- La segunda dependencia es entre *dsub* y *dmul* también por el registro r3, pero ahora para esperar el cálculo el valor de la multiplicación.

Al ser instrucciones que son de ejecución más larga que el resto de las instrucciones esto puede ocasionar múltiples atascos. Por el mismo motivo, distintas instrucciones en el mismo ciclo pueden verse detenidas por atascos diferentes.

Ocurren algunos atascos estructurales al tener instrucciones con distintas etapas de ejecución y las instrucciones intentan acceder a la misma etapa en el ciclo.

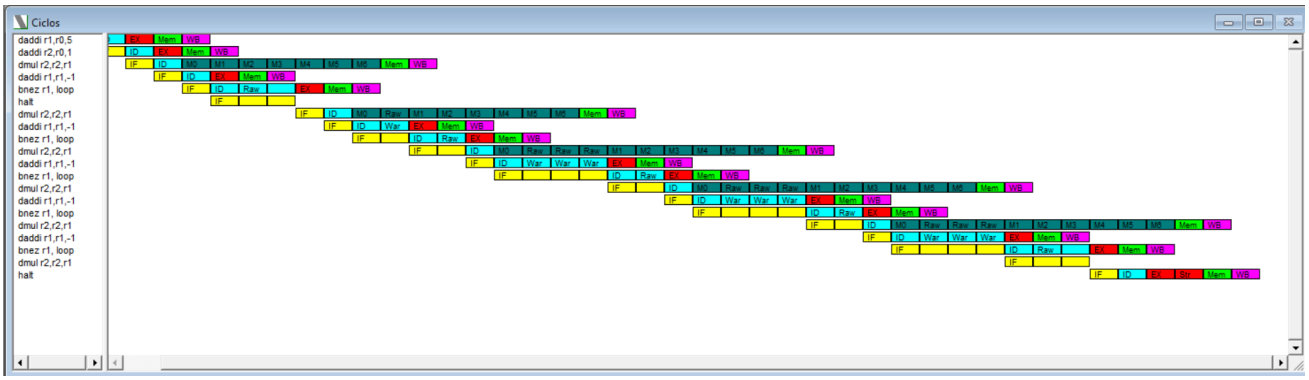


Programa Factorial

En el programa factorial tenemos un salto condicional, como tenemos BTB activo, vamos a tener 2 BTS y 2 BMS.

Como dependencia de datos tenemos:

- La instrucción de salto condicional depende de r1, que es decrementado en la instrucción anterior, esto produce un atasco RAW.
- La instrucción dmul depende tanto de r1 como de *r2*, con *r1* no hay conflictos de dependencia porque es modificada después (ver siguiente punto) pero el registro *r2* que es multiplicado por esta misma instrucción tarda la cantidad suficientes de ciclos como para generar atascos RAW sobre si misma.
- La instrucción *dmul* usa el registro r1 como argumento de la multiplicación, que es decrementado en la instrucción posterior. Esto puede generar atascos WAR cuando la instrucción *dmul* se atasca.
- Difícil de predecir, pero probables atascos estructurales por la ejecución fuera de orden.



	Programa Resto	Programa Factorial
#Instrucciones	6	18
Ciclos	39	42
CPI	6,5	2,333
RAW	51	15
WAR	0	10
WAW	0	0
BTS	0	2
BMS	0	2
Estructurales	7	1

4) Etapas y atascos ⭐⭐

Completar las etapas en la ejecución de los siguientes programas, asumiendo forwarding activado.

a) Suma y producto

Ciclo	1	2	3	4	5	6	7	8	9	11	11	12	13	14	15	16	17	18
ld r1, A(r0)	IF	ID	EX	ME	WB													
ld r2, B(r0)		IF	ID	EX	ME	WB												
dmul r3,r1,r2			IF	ID	RAW	M0	M1	M2	M3	M4	M5	M6	ME	WB				
sd r3, MULT(r0)				IF	RAW	ID	EX	EX	EX	EX	EX	EX	EX	ME	WB			
dadd r3,r1,r2					IF		STR	STR	STR	STR	STR	STR	ID	EX	ME	WB		
sd r3, SUMA(r0)							IF							ID	EX	ME	WB	
halt														IF	ID	EX	ME	WB

b) Promedio

Ciclo	1	2	3	4	5	6	7	8	9	11	11	12	13	14	15	16	17
ld r1, suma(r0)	IF	ID	EX	ME	WB												
ld r2, cant(r0)		IF	ID	EX	ME	WB											
ddiv r3,r1,r2			IF	ID	DIV												
sd r3, prom(r0)				IF	RAW	ID	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW
halt					F		ID										

Ciclo	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
ld r1, suma(r0)																
ld r2, cant(r0)																
ddiv r3,r1,r2													ME	WB		
sd r3, prom(r0)	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	RAW	EX	ME	WB	
halt														EX	ME	WB

Parte 4: Ejercicios de repaso o tipo parcial

1) Análisis de atascos con lazos ⭐⭐⭐

El siguiente programa calcula la suma del arreglo A y almacena el resultado en la variable SUM. Calcular la cantidad de instrucciones, atascos y ciclos totales que toma el programa, y los CPIs, sin utilizar el simulador, asumiendo BTB/DS desactivados, y:

a) Asumiendo **forwarding desactivado**
b) Asumiendo **forwarding activado**.
c) En el caso a), ¿qué sucedería si la instrucción daddi r2,r0,0 se intercambia por daddi r3,r0,3?

Verificar los resultados con el simulador en cada caso.

```
.data
A:   .word 2,1,3
SUM: .word 0
.code
        daddi   r2,r0,0
        ld      r1,SUM(r0)
        daddi   r3,r0,3
loop:   ld      r4, A(r2)
        dadd    r1,r1,r4
        daddi   r3, r3, -1
        bnez    r3, loop
        sd       r1, SUM(r0)
        halt
```

En el inciso c) al reordenar las instrucciones se produce un atasco por dependencia de datos adicional por r2, entre la instrucción **daddi r2, r0, r0** y **ld r4, A(r2)**.

	Sin forwarding	Con forwarding	Sin forwarding + cambio
#Instrucciones	17	17	17
Ciclos	35	29	37
CPI	2,059	1,706	2,176
RAW	12	6	14
BTS	2	2	2
BMS	0	0	0
Estructurales	0	3	0

Pista: Es posible simular manualmente la ejecución de todo el programa ciclo por ciclo, pero esto resulta engorroso, sobre todo para los lazos con varias iteraciones. En lugar de eso, analizar el programa en 3 partes separadas: el código antes del loop, el código del loop (que se repite 3 veces) y el código después del loop. Por cada parte, determinar la cantidad de instrucciones ejecutadas. Para calcular los ciclos, primero determinar las dependencias de datos, en base a eso los atascos, y en base a eso y utilizar la fórmula vista anteriormente para calcular la cantidad de ciclos totales.

2) Cálculo de CPI con lazo tipo while y con forwarding ⭐⭐⭐

El siguiente programa busca determinar si el número **num** se encuentra dentro del vector **tabla**.

```
.data
tabla: .word 20, 1, 14, 7, 12, 11
num:   .word 7
long:  .word 6
 res:   .word 0

.code
  (01)          ld      r1, long(r0)
  (02)          ld      r2, num(r0)
  (03)          dadd    r3, r0, r0
  (04)          dadd    r10, r0, r0
  (05)loop:     ld      r4, tabla(r3)
  (06)          beq     r4, r2, listo
  (07)          daddi   r1, r1, -1
  (08)          daddi   r3, r3, 8
  (09)          bnez    r1, loop
  (10)          j       fin
  (11)listo:    daddi   r10, r0, 1
  (12)fin:      sd r10, res(r0)
  (13)          halt
```

A. Calcular manualmente el número de ciclos, CPI, RAWs y BTS/BMS, asumiendo forwarding activado y BTB desactivado.

Primero calculamos la cantidad de instrucciones que se ejecutan:

- Las 4 primeras instrucciones se ejecutan 1 vez **(01)** a **(04)**
- 5 instrucciones se ejecutan 3 veces en el bucle**(05)** a **(09**), para los valores de tabla 20, 1 y 14
- En la 4ta pasada del bucle, se ejecutan 2 instrucciones **(05)** a **(06)** por ser el valor igual a 7
- Las 3 últimas instrucciones se ejecutan una vez y el programa termina.

  #Instrucciones = 4 + 3 x 5 + 2 + 3 = 4 + 15 + 2 + 3 = 24

Atascos BTS, son 4 en total:

- Hay 3 atascos BTS por los 3 primeros saltos condicionales de la instrucción **(09)** la instrucción que se descarta es **(10)**.
- En la 4ta vuelta del ciclo, cuando el valor 7 se encuentra, en el salto condicional de la instrucción **(06)**, la instrucción descartada es **(07)**.

Atascos RAW, hay 8 en total:

- Cada vez que se inicia el bucle, hay dos atascos RAW por la dependencia de datos de r4 entre las instrucciones (05) y (06). Aunque tengamos forwarding la instrucción *ld* puede adelantar r4 luego de leerlo de memoria *(MEM)* y por ser una instrucción de salto condicional *beq* necesita el valor de *r4* en la etapa de decodificación *(ID)*. Como esas instrucciones se ejecutan para los 4 primeros valores de la tabla, la cantidad de atascos RAW será 8.

La cantidad de ciclos en los que se ejecuta el programa será:

Ciclos sin atascos = #Instrucciones + 4 = 28

Ciclos atascos BTS = 8

Ciclos atascos BTS = 4

Ciclos = ciclos sin atascos + ciclos por atascos RAW + ciclos por atascos BTS = 28 + 8 + 4 = 40

El CPI será : 40 / 24 = 1,666.

B. Idem, asumiendo BTB activado.

El análisis es similar al punto anterior, nos cambia en el análisis de los saltos.

De los 4 saltos que se producen, en los saltos que se producen por volver a iniciar el bucle, que se produce 3 veces, la primera vez no va a ser predecida correctamente (produce un BTS y BMS) y 2 aciertos. El salto que se produce por encontrar el elemento en la tabla no será predecido por lo que también producirá un atasco BTS y otro BMS.

La cantidad de ciclos en los que se ejecuta el programa será:

Ciclos sin atascos = #Instrucciones + 4 = 28

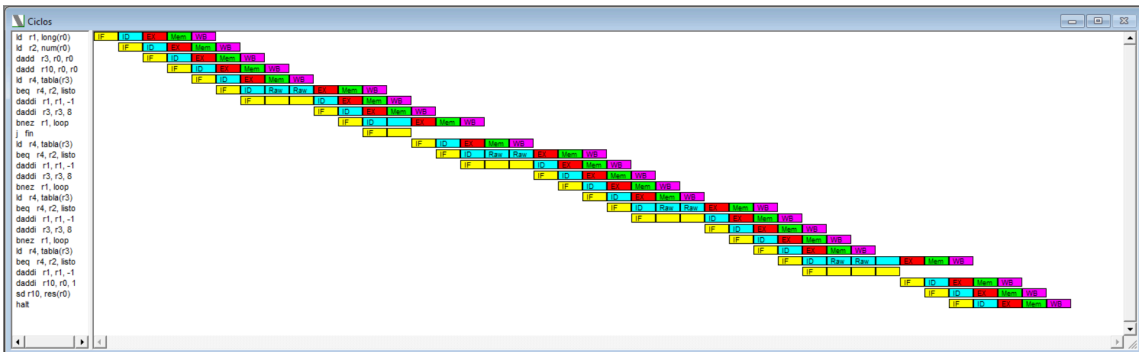
Ciclos atascos BTS = 8

Ciclos atascos BTS = 2

Ciclos atascos BMS = 2

Ciclos = ciclos sin atascos + ciclos por atascos RAW + ciclos por atascos BTS y BMS = 28 + 8 + 4 = 40

El CPI será : 40 / 24 = 1,666.



C. Modificar el programa para que con DS activado funcione correctamente y no ejecute instrucciones de más.

```
.data
tabla: .word 20, 1, 14, 7, 12, 11
num:   .word 7
long:  .word 6
res:   .word 0

.code
               ld      r1, long(r0)
               ld      r2, num(r0)
               dadd    r3, r0, r0
               dadd    r10, r0, r0
loop:          ld      r4, tabla(r3)
               beq     r4, r2, listo
               daddi   r1, r1, -1

               bnez    r1, loop
               daddi   r3, r3, 8

               j       fin
               nop

listo:         daddi   r10, r0, 1
fin:           sd r10, res(r0)
               halt
```

3) Cálculo de CPI con lazo tipo for y sin forwarding ⭐⭐⭐

El siguiente programa multiplica por 2 los valores del vector datos mediante un desplazamiento a la izquierda (**dsll**).

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

.code
(01)           dadd    r1, r0, r0
(02)           ld      r2, cant(r0)
(03)loop:      ld      r3, datos(r1)
(04)           daddi   r2, r2, -1
(05)           dsll    r3, r3, 1
(06)           sd      r3, res(r1)
(07)           daddi   r1, r1, 8
(08)           bnez    r2, loop
(09)           halt
```

A. Calcular manualmente el número de ciclos, CPI, RAWs y BTS/BMS, asumiendo **forwarding desactivado**, **BTB/DS desactivados.**

Cantidad total de instrucciones, 51 instrucciones ejecutadas:

- 2 instrucciones al inicio del programa **(01)** a **(02)**

- Se repite 8 veces el bucle de 6 instrucciones **(03)** a **(08)**. 48 en total
- Se ejecuta 1 vez la instrucción *halt* **(09)**.

Como el ciclo se repite 8 veces, ocurrirán 7 atascos BTS.

Atascos por dependencia de datos:

- Como no hay forwarding hay un atasco de un ciclo por la disponibilidad de r1 entre **(01)** y **(03)** al inicio del programa.
- Si bien hay una dependencia de datos entre **(02)** y **(04)**, no se genera atasco por el atasco que se genera en el punto anterior.
- En cada vuelta del bucle ocurrirán 3 atascos, como se ejecuta 8 veces serán 24 RAWs en total
    - Un atasco RAW entre **(03)** y **(05)**
    - Dos atascos RAW entre **(05)** y **(06)**

La cantidad de ciclos en los que se ejecuta el programa será:

Ciclos sin atascos = #Instrucciones + 4 = 55

Ciclos atascos BTS = 25

Ciclos atascos BTS = 7

Ciclos = ciclos sin atascos + ciclos por atascos RAW + ciclos por atascos BTS = 55 +25 + 7 = 87

El CPI será : 87 / 51 = 1,706

B. Modificar el programa para funcionar correctamente con DS. Calcular manualmente el número de ciclos, CPI, RAWs y BTS/BMS, asumiendo **forwarding desactivado** y **DS activado.**

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

.code
(01)          dadd   r1, r0, r0
(02)          ld     r2, cant(r0)
(03)loop:     ld     r3, datos(r1)
(04)          daddi  r2, r2, -1
(05)          dsll   r3, r3, 1
(06)          sd     r3, res(r1)
(07)          bnez   r2, loop
(08)          daddi  r1, r1, 8
(09)          halt
```

Reordenando las instrucciones la cantidad de instrucciones totales que se ejecutan no cambia.

Como cambiamos el orden de las instrucciones, ahora generamos otros atascos, principalmente porque ahora las instrucciones (09) y (03) estan consecutivas y genera dos atascos RAW.

Ya no vamos a tener atascos por saltos, por usar **delay slot**.

De los atascos RAWs que ocurrieron originalmente sumamos 2 atascos RAW entre (09) y (03) por cada vuelta al inicio del bucle que se realice. Como el bucle se ejecuta 8 veces, se vuelve 7, es decir sumamos 7 veces 2 atascos RAW, es decir 14. La cantidad de RAWs asciende de 25 a 39.

La cantidad de ciclos en los que se ejecuta el programa será:

Ciclos sin atascos = #Instrucciones + 4 = 55

Ciclos atascos BTS = 39

Ciclos = ciclos sin atascos + ciclos por atascos RAW = 55 +39 = 94

El CPI será : 94/ 51 = 1,843