

Práctica 6: Segmentación de cauce y atascos

Parte 1: Estructura del Cauce

Ejercicio 1

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

Ejercicio 2

Instrucción	IF	ID	EX	MEM	WB
daddi r1, r3, 4	Busca la instrucción en memoria y la lleva a la CPU	Decodifica la instrucción. Verifica si está disponible el registro r3. Recupera r3 del banco de registros.	Realiza el cálculo de $r3 + 4$		Guarda en r1 el resultado de la suma de r3 y 4
dadd r2, r4, r3	Busca la instrucción en memoria y la lleva 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 cálculo de $r4 + r3$		Guarda en r2 el resultado de la suma de r4 y r3
sd r3, tabla(r2)	Busca la instrucción en memoria y la lleva a la CPU	Decodifica la instrucción. Verifica si están disponibles los operandos r3 y r2. Recupera r3 y r2 del banco de registros.	Realiza la suma de $\text{tabla} + r2$, que es la dirección a acceder en memoria	Accede a la dirección de memoria calculada y guarda en ella el valor de r3	

ld r3, tabla(r2)	Busca la instrucción en memoria y la lleva 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 acceder en memoria	Accede a la dirección de memoria calculada y recupera el valor allí almacenado	Se guarda en r3 el valor leído de memoria
bne r1, r2, loop	Busca la instrucción en memoria y la lleva a la CPU	Decodifica la instrucción. Verifica si están disponibles los operandos r1 y r2. Recupera r1 y r2 del banco de registros. Si r1 y r2 son distintos, calcula el nuevo valor del registro IP.			
j loop	Busca la instrucción en memoria y la lleva a la CPU	Decodifica la instrucción. Calcula el nuevo valor del registro IP.			

Observación: la quinta instrucción era originalmente “bneq r1, r2, loop”. Al no tratarse de una instrucción válida, se decidió reemplazarla por “bne r1, r2, loop”, tal como se exhibe en la tabla.

Ejercicio 3

a)

Programa 1:

Instrucción	Cauce del procesador								
daddi r2, r3, 5	IF	ID	EX	MEM	WB				
dsub r4, r3, r5		IF	ID	EX	MEM	WB			
xor r6, r3, r5			IF	ID	EX	MEM	WB		
nop				IF	ID	EX	MEM	WB	
halt					IF	ID	EX	MEM	WB

La fórmula para el cálculo del CPI expuesta en la introducción del presente trabajo práctico funciona se cumple para este programa porque todas sus instrucciones presentan las mismas 5 unidades de ejecución para sus respectivos cauces.

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

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

$$CPI = \frac{\text{Instrucciones} + 4}{\text{Instrucciones}} = \frac{5 + 4}{5} = \frac{9}{5} = 1,8$$

Programa 2:

Instrucción	Cauce del procesador																
	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	12 ciclos DIV	MEM	WB
ddiv r6, r3, r5																	
dsub r4, r3, r5		IF	ID	M0	M1	M2	M3	M4	M5	M6	MEM	WB					
xor r6, r3, r5			IF	ID	EX	MEM	WB										
halt				IF	ID	EX	MEM	WB									

La fórmula para el cálculo del CPI expuesta en la introducción del presente trabajo práctico funciona no se cumple para este programa porque no todas sus instrucciones presentan las mismas 5 unidades de ejecución para sus respectivos cauces. En particular, el número de ciclos de ejecución de la etapa Execute (EX) difiere drásticamente entre las distintas instrucciones ejecutadas.

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

$$CPI = \frac{\text{Ciclos}}{\text{Instrucciones}} = \frac{1 + 1 + 24 + 1 + 1}{4} = \frac{28}{4} = 7$$

$$CPI = \frac{\text{Instrucciones} + 4}{\text{Instrucciones}} = \frac{4 + 4}{4} = \frac{8}{4} = 2$$

b)

Si se mantiene la configuración original del simulador WinMIPS64, la etapa de ejecución de las instrucciones “dmul” y “ddiv” requiere 7 y 24 ciclos de ejecución respectivamente para completarse:

Configurar Arquitectura

Bus de Direcciones de Código

10

Bus de Direcciones de Datos

10

Latencia del Sumador PF

4

Latencia del Multiplicador

7

Latencia del Divisor

24

OK

Cancelar

Advertencia: Esto causará un Reset!

Por lo tanto, los números de ciclos de ejecución de ambas instrucciones serán los siguientes:

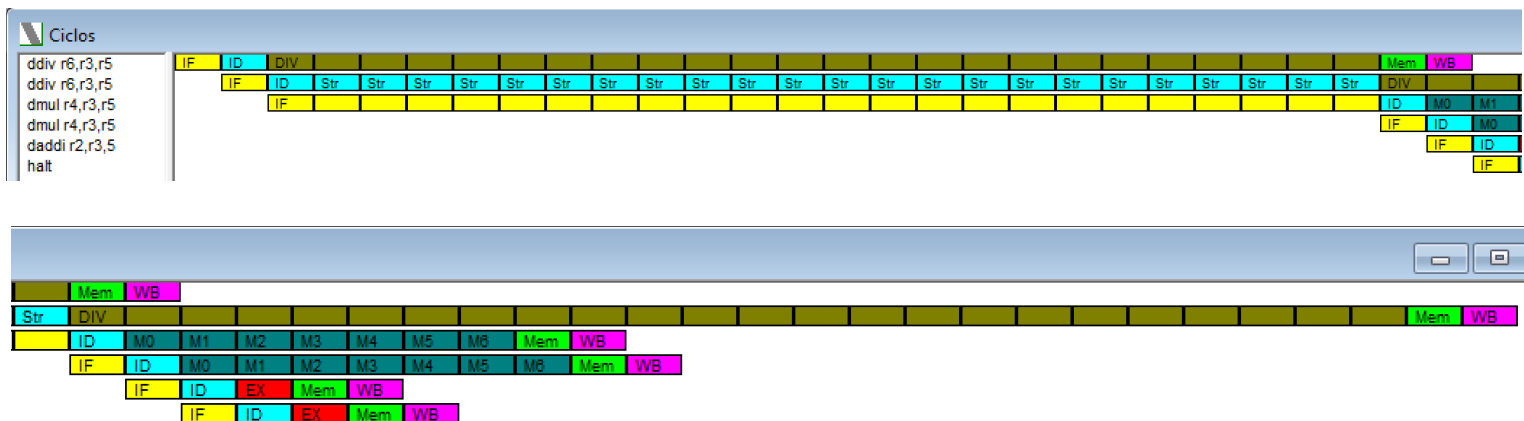
$$C_{dmul} = C_{IF} + C_{ID} + C_M + C_{MEM} + C_{WB} = 1 + 1 + 7 + 1 + 1 = 11 \text{ ciclos de ejecución}$$

$$C_{ddiv} = C_{IF} + C_{ID} + C_{DIV} + C_{MEM} + C_{WB} = 1 + 1 + 24 + 1 + 1 = 28 \text{ ciclos de ejecución}$$

c)

El cauce de la etapa de ejecución de la instrucción de división “dmul” no se encuentra segmentado. En consecuencia, dicha etapa solo se podrá ejecutar en forma secuencial: cualquier nueva división a efectuarse en la ALU deberá esperar obligatoriamente a que el cálculo actual, si lo hubiera, se haya completado, independientemente de la etapa particular de su cauce que se encuentre en ejecución en este momento.

```
.code
ddiv r6, r3, r5
ddiv r6, r3, r5      # instrucción duplicada
dmul r4, r3, r5
dmul r4, r3, r5      # instrucción duplicada
daddi r2, r3, 5
halt
```



d)

La instrucción “halt” posee la misma estructura para su cauce de ejecución que cualquier otra instrucción tradicional del programa que no calcule una multiplicación, división u operación aritmética en punto flotante. Por lo tanto, corresponde considerarla en el cálculo del CPI de cualquier programa ejecutado.

e)

```
.code
daddi r2, r3, 5
dsub r4, r3, r5
xor r6, r3, r5
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
```

```
nop  
nop  
nop  
nop  
nop  
nop  
nop  
nop  
nop  
nop  
nop  
nop  
nop  
halt
```

Las 20 nuevas instrucciones “nop” presentan las mismas 5 unidades de ejecución para sus respectivos cauces que las instrucciones originales de este programa. En consecuencia, es válida para el cálculo del CPI la reutilización de la fórmula expuesta en la introducción del presente trabajo práctico:

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

$$\text{CPI} = \frac{\text{Ciclos}}{\text{Instrucciones}} = \frac{5 + 24}{25} = \frac{29}{25} = 1,16$$

$$\text{CPI} = \frac{\text{Instrucciones} + 4}{\text{Instrucciones}} = \frac{25 + 4}{25} = \frac{29}{25} = 1,16$$

Aunque el CPI aquí calculado sea inferior a aquél obtenido en el inciso a) para el programa original, cuyo CPI había sido igual a “1,8”, esto no necesariamente implica una mayor eficiencia de este nuevo programa. Esta afirmación se argumenta en que la eficiencia no solo depende del promedio de ciclos ejecutados por cada instrucción completa del programa (CPI), sino de la tarea útil efectivamente realizada durante dichos ciclos. En este caso, como el trabajo llevado a cabo es exactamente el mismo en ambos programas, podría plantearse de hecho que el primer programa es en realidad más eficiente porque carece de todas las instrucciones adicionales “nop” que no efectúan ninguna labor provechosa, logrando así realizar las mismas actividades del segundo programa en una cantidad menor de ciclos de ejecución (aunque su CPI resulte de hecho mayor): 9 ciclos en lugar de 29. Esta diferencia de 20 ciclos, como habrá podido adivinarse, yace estrictamente en las 20 instrucciones “nop”, una por cada ciclo, incorporadas innecesariamente en el segundo programa, excluyendo el obvio objetivo conceptual con el cual esta consigna fue concebida: establecer que el CPI no representa por sí solo un estimador absoluto e infalible del nivel de eficiencia de un programa.

Parte 2: Atascos RAW

Ejercicio 1

a)

Programa 1:

Número	Instrucción	Dependencia/s de datos de lectura
1	daddi r1, r0, 5	
2	daddi r2, r0, 7	
3	slt r3, r1, r2	Instrucciones 1 y 2: registros r1 y r2 respectivamente
4	daddi r1, r0, 1	
5	and r4, r3, r1	Instrucciones 3 y 4: registros r3 y r1 respectivamente
6	daddi r1, r0, 8	
7	sd r4, A(r1)	Instrucciones 5 y 6: registros r4 y r1 respectivamente

Programa 2:

Número	Instrucción	Dependencia/s de datos de lectura
1	ld r1, A(r0)	
2	ld r2, B(r0)	
3	bne r1, r2, no	Instrucciones 1 y 2: registros r1 y r2 respectivamente
4	daddi r3, r0, 1	
5	j fin	
6	no: daddi r3, r0, 0	
7	fin: sd r3, C(r0)	Instrucción 4 o 6: registro r3

Programa 4:

Número	Instrucción	Dependencia/s de datos de lectura
1	daddi r1, r0, 0	
2	daddi r2, r0, 0	
3	loop: ld r3, A(r1)	Instrucción 1 o 5: registro r1
4	dadd r2, r2, r3	Instrucciones 2 o 4 y 3: registros r2 y r3 respectivamente
5	daddi r1, r1, 8	Instrucción 1 o 5: registro r1
6	bnez r3, loop	Instrucción 3: registro r3
7	sd r2, RES(r0)	Instrucción 4: registro r2

b)

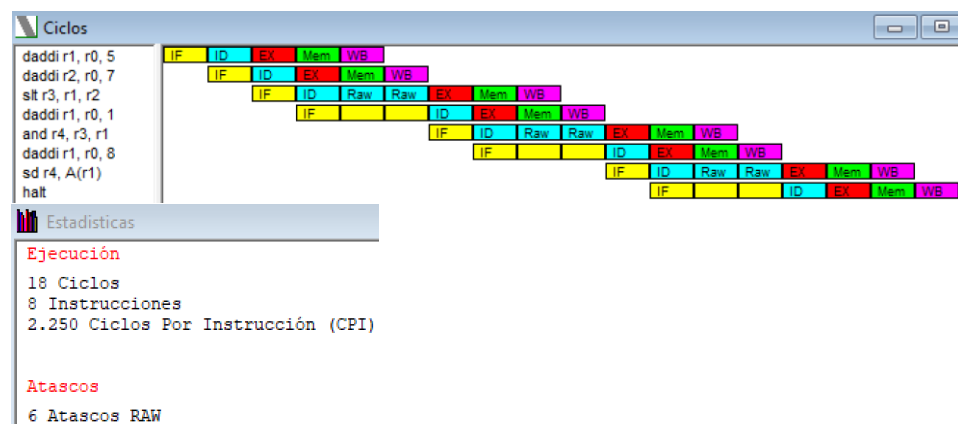
En cualquier programa solo presentarán atascos por dependencias de datos de tipo RAW aquellas instrucciones que se ejecuten con una proximidad con respecto a las instrucciones de las que dependen tal que estas dependencias no hayan alcanzado a resolverse una vez que debería comenzar la etapa de la instrucción que requiere efectivamente el dato a leer para completarse.

Todos los programas fueron ejecutados en el simulador con la técnica Forwarding desactivada. Asimismo se les realizaron algunas modificaciones menores en su implementación como, por ejemplo, declarar una sección de datos o una instrucción “halt” al final, para lograr su correcta simulación.

Programa 1:

```
.data
A: .word 0

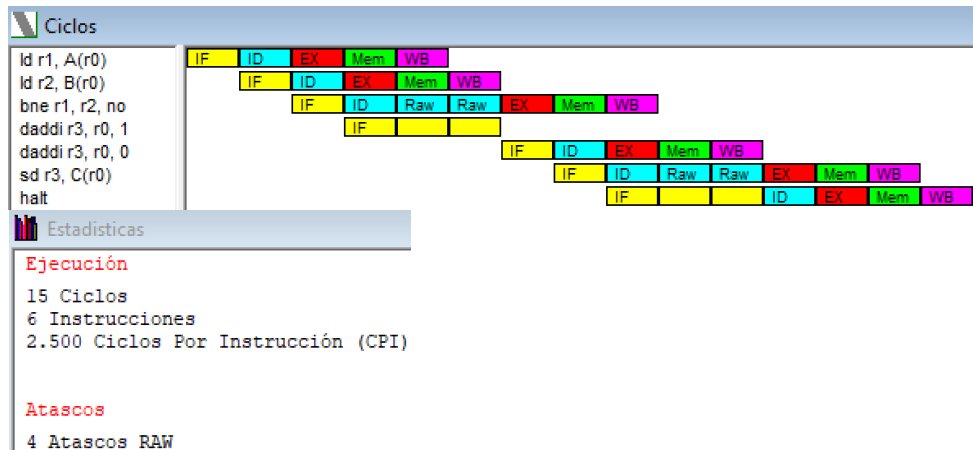
.code
daddi r1, r0, 5
daddi r2, r0, 7
slt r3, r1, r2
daddi r1, r0, 1
and r4, r3, r1
daddi r1, r0, 8
sd r4, A(r1)
halt
```



Programa 2:

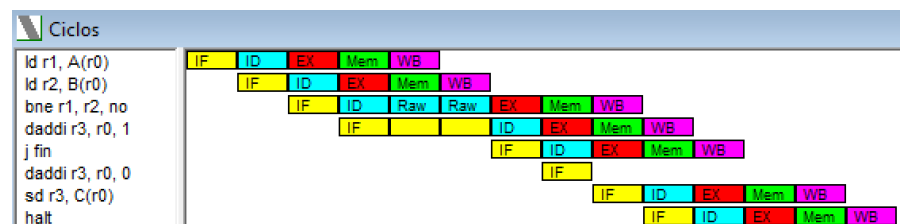
```
.data
A: .word 1 # A≠B
B: .word 2
C: .word 0

.code
ld r1, A(r0)
ld r2, B(r0)
bne r1, r2, no
daddi r3, r0, 1
j fin
no: daddi r3, r0, 0
fin: sd r3, C(r0)
halt
```



```
.data
A: .word 2 # A=B
B: .word 2
C: .word 0

.code
ld r1, A(r0)
ld r2, B(r0)
bne r1, r2, no
daddi r3, r0, 1
```



```
j fin
no:  daddi r3, r0, 0
fin: sd r3, C(r0)
halt
```

Estadísticas

Ejecución

14 Ciclos
7 Instrucciones
2.000 Ciclos Por Instrucción (CPI)

Atascos

2 Atascos RAW

Programa 4:

```
.data
A:  .word 5, 3, -4, 0
RES: .word 0

.code
daddi r1, r0, 0
daddi r2, r0, 0
loop: ld r3, A(r1)
      dadd r2, r2, r3
      daddi r1, r1, 8
      bnez r3, loop
      sd r2, RES(r0)
halt
```

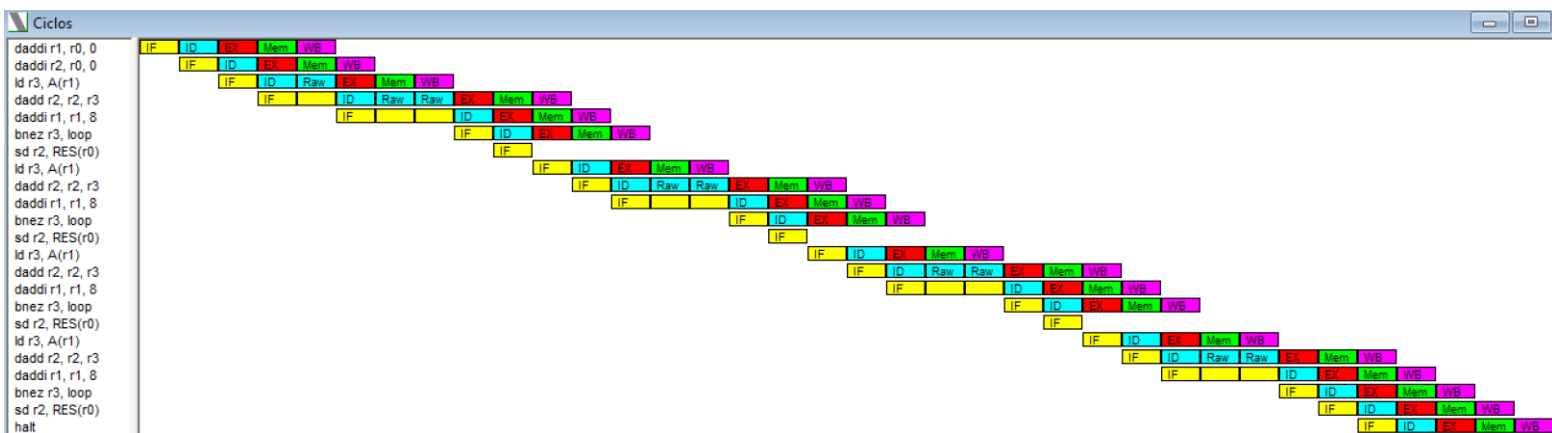
Estadísticas

Ejecución

36 Ciclos
20 Instrucciones
1.800 Ciclos Por Instrucción (CPI)

Atascos

9 Atascos RAW



c)

Las instrucciones correspondientes a los programas 1 y 2 no pueden ser reordenadas con el objetivo de reducir la cantidad de atascos generados por las dependencias de datos de tipo RAW presentes entre ellas sin modificar el resultado final de los programas originales.

Sin embargo, en los programas 3 y 4 la existencia de bucles internos facilita la posibilidad de un reordenamiento de sus instrucciones para aumentar la separación o distancia entre aquellas que presenten dependencias de datos de tipo RAW. De este modo, al comenzar la ejecución de la instrucción potencialmente afectada por la dependencia y que pueda generar un atasco, dicha dependencia ya se encontraría resuelta en el cauce del procesador y el atasco se evitaría por completo (o, al menos, tendría una duración menor).

Programa 4:

```
.data
A:  .word 5, 3, -4, 0
RES: .word 0
```

Estadísticas

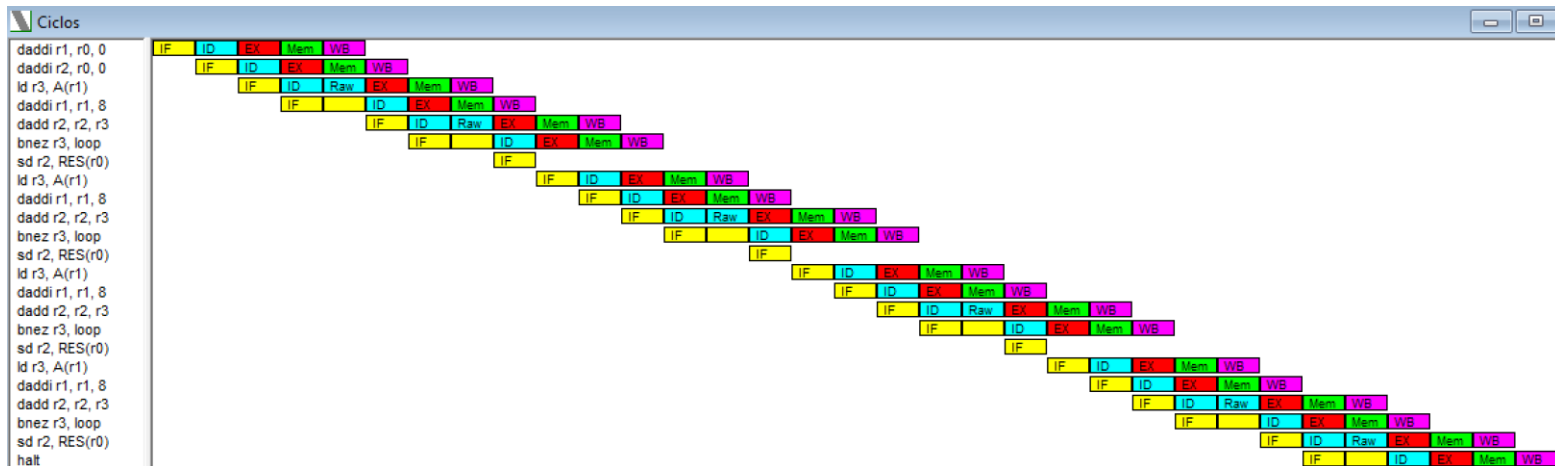
Ejecución

33 Ciclos
20 Instrucciones
1.650 Ciclos Por Instrucción (CPI)

Atascos

6 Atascos RAW


```
.code
daddi r1, r0, 0
daddi r2, r0, 0
loop: ld r3, A(r1)
      daddi r1, r1, 8      # Estas dos instrucciones fueron
      dadd r2, r2, r3      # intercambiadas entre sí
      bnez r3, loop
      sd r2, RES(r0)
      halt
```



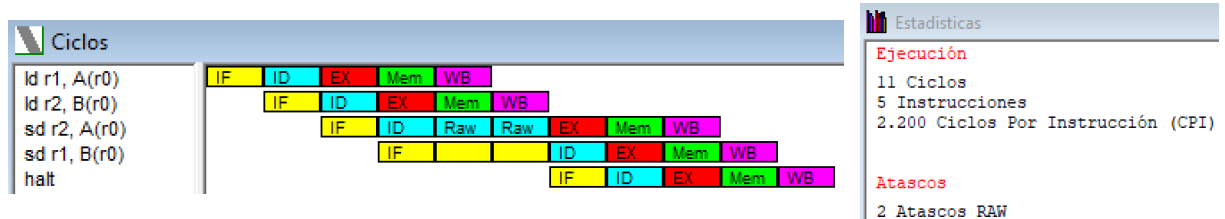
Ejercicio 2

```
.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
```

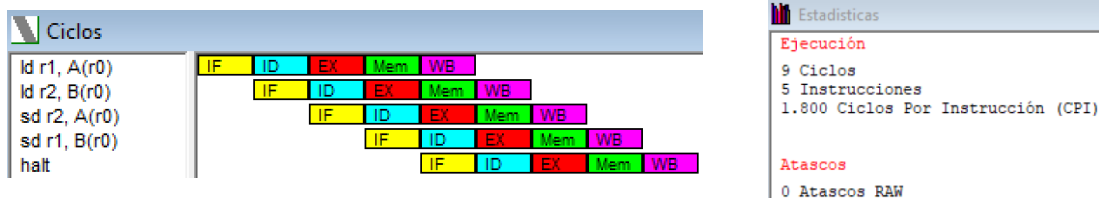
a)

Si el programa propuesto en la presente consigna se ejecuta en el simulador WinMIPS64 con la técnica Forwarding desactivada, se obtendrán los resultados expuestos a continuación. Básicamente, existen dos atascos RAW en el cauce del procesador, ambos correspondientes a la tercera instrucción, “sd r2, A(r0)”, ya que la misma debe esperar a que la segunda instrucción, “ld r2, B(r0)”, cargue en su respectiva etapa WB el valor del registro r2 para poder proceder a leerlo. Por su parte, tal como se puede observar en la captura, el CPI del programa es igual a 2,2.



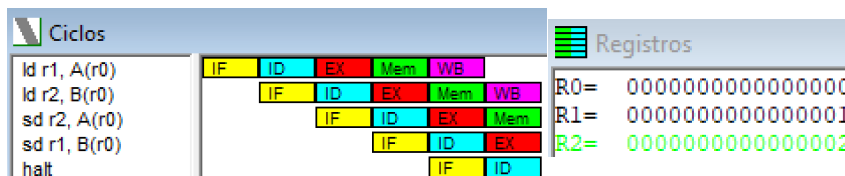
b)

En este nuevo escenario, al activarse la técnica de Forwarding, el valor del registro r2 es “adelantado” por la segunda instrucción, “ld r2, B(r0)”, a la tercera, “sd r2, A(r0)”, tan pronto como lo lee de memoria en la etapa MEM (antes “sd” debía esperar a que “ld” lo escribiera en r2 en la etapa WB). Por su parte, la tercera instrucción “atrassa”, “demora” o “pospone” la verificación de la disponibilidad del valor de r2 (la cual antes siempre se hacía en la etapa ID) hasta su etapa MEM, ya que recién aquí requería realmente dicho valor para almacenarlo (escribirlo) en memoria. Como se observa en el cauce del programa representado en la siguiente captura, al resultar la ejecución de la etapa “MEM” de la tercera instrucción posterior a la etapa “MEM” de la segunda instrucción, la dependencia RAW entre ambas instrucciones logra resolverse en forma completa antes de que genere un atasco en el cauce de la CPU. Una clara y predecible ventaja consiste en una disminución del CPI del programa: 1,8 en lugar de 2,2. Al mantenerse inalterable el número de instrucciones, puede afirmarse que la ejecución del programa es más eficiente porque se lleva a cabo la misma tarea útil en una cantidad más reducida de ciclos de reloj (9 en vez de 11).



Si uno de los registros en la ventana Register del simulador se colorea durante la ejecución del programa, se está indicando que ese registro tiene un valor actualizado en el último ciclo ejecutado, y que dicho valor se encontró disponible para su uso (acceso) en forma adelantada por cualquier instrucción que lo requiriera. El color depende de la etapa en la cual fue generado, determinado u obtenido el valor del registro: rojo y verde para las etapas EX y MEM respectivamente.

Por ejemplo, como se comprobará en la próxima captura, para este programa (ejecutado “Paso a Paso” en el simulador para apreciar el progreso paulatino de su cauce), el registro r2 fue actualizado con el valor 2 en la etapa WB de la segunda instrucción. Sin embargo, al encontrarse el Forwarding activo, dicho valor ya se encontró en realidad disponible desde la finalización de la etapa MEM, en la cual fue obtenido por esta instrucción al leerlo de memoria, para ser accedido en forma adelantada por cualquier instrucción que lo necesitara. En este caso, dicha instrucción sería la tercera del programa, la cual, como se explicó previamente, mediante el Forwarding se encontró en condiciones de demorar la validación de la disponibilidad del valor del registro r2 hasta su propia etapa MEM y así evitar generar un atasco en el cauce de la CPU.



Ejercicio 3

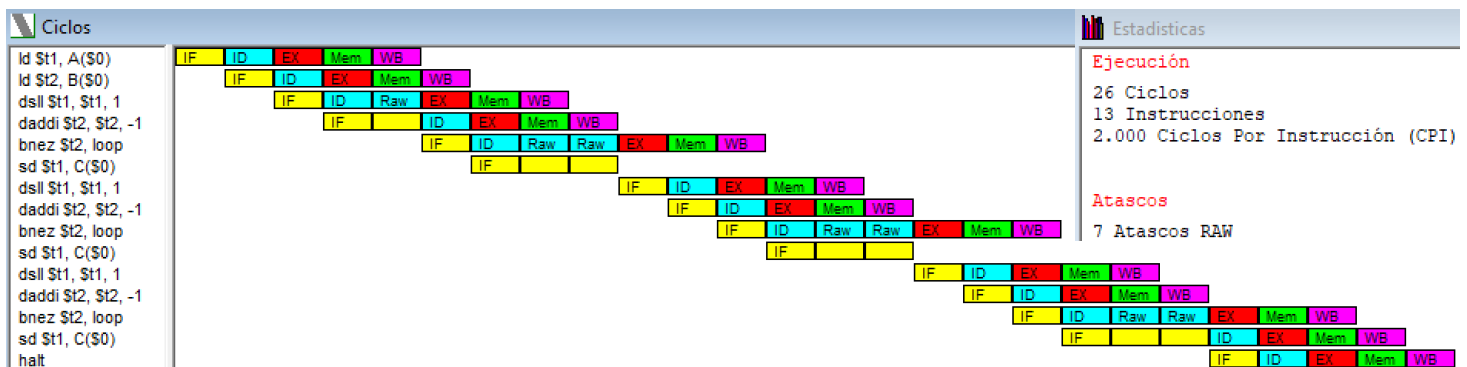
```
.data
A:   .word 1
B:   .word 3
C:   .word 0
```

```
.code
ld $t1, A($0)
ld $t2, B($0)
loop: dsll $t1, $t1, 1
      daddi $t2, $t2, -1
      bnez $t2, loop
      sd $t1, C($0)
      halt
```

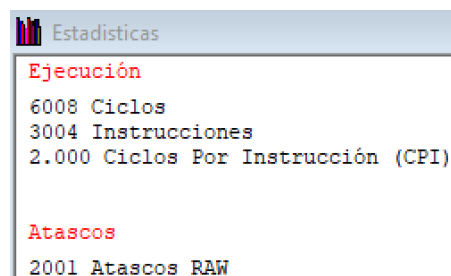
a)

Si se ejecuta el presente programa desactivando el Forwarding, se generará en la primera iteración del bucle un atasco RAW en la tercera instrucción, “dsll”, por una dependencia de datos de lectura existente entre ella y la primera instrucción, “ld”, con respecto al valor del registro \$t1.

Asimismo, también se provocarán en cada repetición del bucle dos atascos RAW en la quinta instrucción, “bnez”, por una dependencia de datos de lectura presente entre ella y la cuarta instrucción, “daddi”, con respecto al valor del registro \$t2.



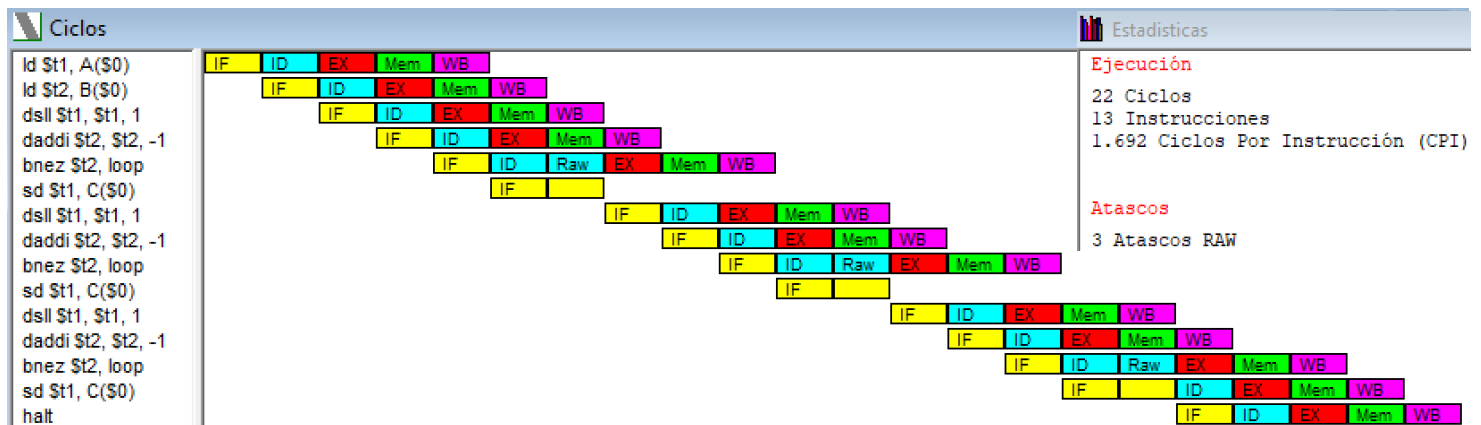
El valor de B determina directamente la cantidad de iteraciones del bucle. Por lo tanto, si se cambia el valor de B a 1000, aumentará la cantidad de instrucciones ejecutadas, por lo cual uno podría asumir inicialmente, observando la fórmula para el cálculo del CPI, que el valor de éste será menor ahora. No obstante, analizando nuevamente la ejecución del programa, se observa que cada iteración adicional del bucle involucra la presencia de otros dos atascos RAW (y un atasco Branch Taken, excepto en la última repetición del lazo) en el cauce del procesador, por lo cual el CPI en realidad no disminuye en absoluto, sino que permanece constante.



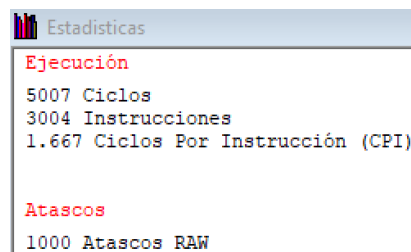
b)

Si se ejecuta el presente programa activando el Forwarding, se generará en cada repetición del bucle un atasco RAW en la quinta instrucción, “bnez”, por una dependencia de datos de lectura presente entre ella y la cuarta instrucción, “daddi”, con respecto al valor del registro \$t2.

Al comparar este escenario con aquél planteado al comienzo del inciso a), se puede adivinar que la cantidad total de atascos aquí provocados será muy inferior a la del caso anterior. En consecuencia, el CPI también será menor, obteniéndose así una mayor eficiencia para la ejecución del programa a partir de la implementación del Forwarding.



Por su parte, si se lo ejecuta modificando el valor de B a 1000, de una manera muy similar al inciso anterior no se observa una notoria disminución en el CPI, por los mismos motivos allí expuestos.



c)

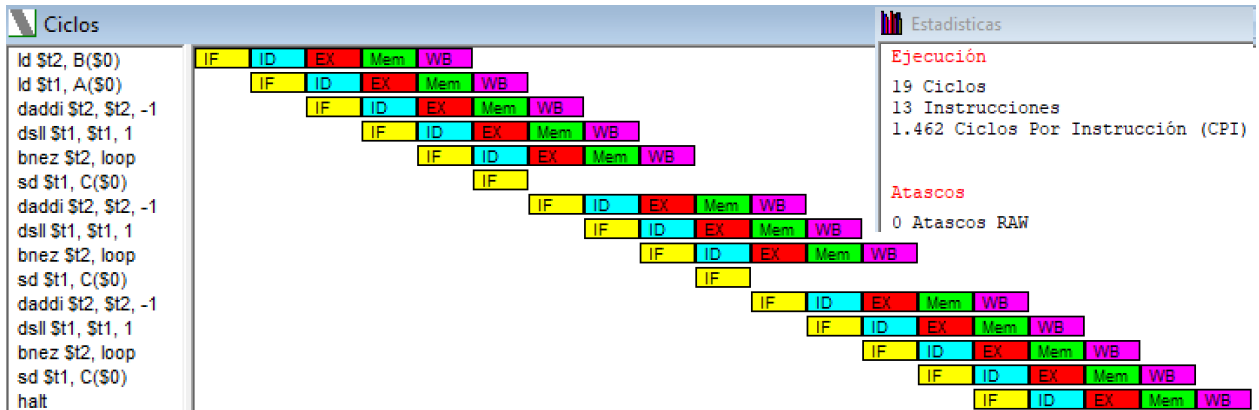
```

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

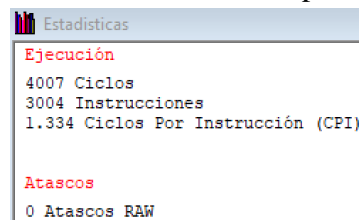
.code
ld $t2, B($0)      # Estas dos instrucciones fueron
ld $t1, A($0)      # intercambiadas entre sí
loop: daddi $t2, $t2, -1 # Estas dos instrucciones fueron
    dsll $t1, $t1, 1   # intercambiadas entre sí
    bnez $t2, loop
    sd $t1, C($0)
    halt
  
```

Si se realiza este reordenamiento en las instrucciones del programa y se ejecuta este último en el simulador habilitando el Forwarding, se observará que no se genera ningún atasco en el cauce del procesador. Este fenómeno se justifica en que ahora las instrucciones que poseen

dependencias de datos de lectura presentan la separación necesaria entre sí para que dichas dependencias alcancen a resolverse antes de que cada una de las instrucciones afectadas por los potenciales atascos requieran efectivamente su respectivo dato crítico.



Por su parte, al ejecutar el nuevo programa modificando el valor de B a 1000, se observa una disminución más considerable en el CPI si se la compara con los incisos anteriores, cuyos correspondientes programas presentaban atascos RAW en sus bucles. Sin embargo, la constante presencia de un atasco Branch Taken en cada iteración del lazo (excepto la última) impide que la reducción en el CPI pueda ser aún más notoria.



En general, para cualquier programa, en caso de pretenderse mejorar su eficiencia, siempre habrá de procurar centrarse en optimizar en primer lugar aquellas secciones del mismo que requieran los números más elevados de ciclos de ejecución para completarse, ya sea por presentar una alta cantidad de atascos (independientemente de su naturaleza) o bien por tratarse de un bucle.

En particular, la sección crítica de este programa cuya optimización resultaría especialmente prioritaria estaría conformada por las tres instrucciones que se ejecutan internamente en cada iteración del bucle “loop”: “dsll”, “daddi” y “bnez”, tratándose éstas de la tercera, cuarta y quinta instrucción del programa respectivamente.

d)

La diferencia entre la cantidad de instrucciones que constituyen el programa (siete) y aquéllas efectivamente ejecutadas en el simulador (trece) yace en las múltiples iteraciones realizadas del bucle “loop”. De esta manera, el conjunto de las tres instrucciones que conforman dicho bucle se ejecutará en más de una oportunidad. Ergo, el número de total de instrucciones ejecutadas del programa será indudablemente mayor a la cantidad de instrucciones de su código estático (previo a su ejecución).

En un programa que presente un bucle en su ejecución, siempre y cuando no exista ninguna instrucción de transferencia de control condicional o incondicional capaz de alterar su flujo de ejecución normal, podría determinarse la cantidad de instrucciones ejecutadas a través de una

suma: por un lado, el número de instrucciones ajenas y externas al lazo y, por otro, el producto entre el valor total de iteraciones del bucle (este valor tendría que averiguarse previamente) y la cantidad de instrucciones internas de dicho bucle (y, por lo tanto, se ejecutarán en cada una de sus iteraciones).

Por ejemplo, para el caso de este programa, asumiendo $A = 1$ y $B = 3$:

Instrucciones del programa $\rightarrow I_P = 7$

Instrucciones ajenas al lazo $\rightarrow I_A = 4$

Total de iteraciones del lazo $\rightarrow N_I = B = 3$

Instrucciones internas del lazo $\rightarrow I_I = 3$

Instrucciones ejecutadas $\rightarrow I_E = I_A + N_I * I_I = 4 + 3 * 3 = 4 + 9 = 13$

Se observa que la cantidad total de instrucciones ejecutadas aquí calculada coincide efectivamente con aquella informada en la ventana “Estadísticas” del simulador una vez ejecutado el programa en él.

Parte 3: Atascos por dependencias de control

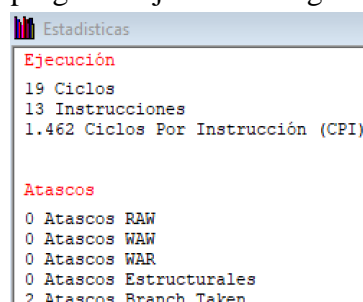
Ejercicio 1

a)

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

.code
ld $t2, B($0)      # Estas dos instrucciones fueron
ld $t1, A($0)      # intercambiadas entre sí
loop: daddi $t2, $t2, -1 # Estas dos instrucciones fueron
    dsll $t1, $t1, 1   # intercambiadas entre sí
    bnez $t2, loop
    sd $t1, C($0)
    halt
```

Al tratarse este programa del mismo que aquél ejecutado en el inciso c del ejercicio 3 de la parte 2 de este mismo trabajo práctico, se consideró innecesaria la representación del cauce de la ejecución de dicho programa debido a que ya puede observarse en ese inciso. Por lo tanto, aquí solamente se presentarán las estadísticas informadas por el simulador una vez finalizada la ejecución, pero en este caso incluyendo también la cantidad de atascos BTS (Branch Taken Stall) además de aquéllos de tipo RAW. La primera y segunda capturas corresponden al programa ejecutado asignándole los valores 3 y 1000 respectivamente a la variable B.



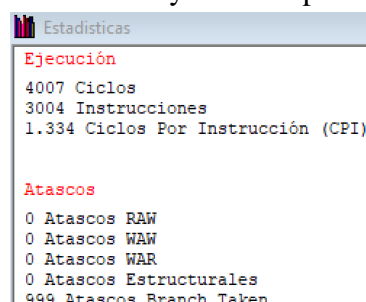
Estadísticas

Ejecución

19 Ciclos
13 Instrucciones
1.462 Ciclos Por Instrucción (CPI)

Atascos

0 Atascos RAW
0 Atascos WAW
0 Atascos WAR
0 Atascos Estructurales
2 Atascos Branch Taken



Estadísticas

Ejecución

4007 Ciclos
3004 Instrucciones
1.334 Ciclos Por Instrucción (CPI)

Atascos

0 Atascos RAW
0 Atascos WAW
0 Atascos WAR
0 Atascos Estructurales
999 Atascos Branch Taken

La cantidad de atascos Branch Taken está directamente asociada al número de saltos “tomados” o efectuados durante la ejecución del programa. En este caso, dicha cantidad será igual al número total de iteraciones del bucle decrementado en una unidad, ya que durante la última repetición inevitablemente la condición de salto no se cumplirá y se procederá a abandonar el lazo sin que se lleve a cabo salto alguno (en caso contrario, no sería la última iteración del bucle).

Al elevarse el valor de B, se incrementa la cantidad de repeticiones del bucle, aumentando el número de instrucciones ejecutadas del programa, pero también los atascos generados en el cauce del procesador, por lo cual la mejora en el CPI del segundo programa no es demasiado significativa con respecto al primero.

b)

Al ejecutar un loop simple con N iteraciones, se producen $N - 1$ atascos BTS (por los motivos ya expresados en el inciso previo).

c)

Los atascos BTS se producen cuando se provoca un salto en la ejecución del programa. La CPU sabe que una instrucción es de salto cuando se decodifica en la etapa ID. Además, por defecto, implementa una técnica de predicción de saltos estática a través de la cual siempre asume que el salto no se va a efectuar. Por lo tanto, inevitable e invariablemente procederá a captar la instrucción inmediatamente posterior al salto para su próxima ejecución. En consecuencia, si el salto se llegara a tomar, dicha instrucción no se ejecutará y deberá descartarse por completo ya que en su lugar se va a ejecutar la instrucción correspondiente a la dirección destino del salto.

d)

Los BTS no pueden evitarse reordenando instrucciones porque, a diferencia de los atascos RAW, no se producen por una dependencia de datos sino por haber empezado a ejecutar erróneamente la etapa IF de una instrucción que no pertenecerá al flujo de ejecución del programa.

e)

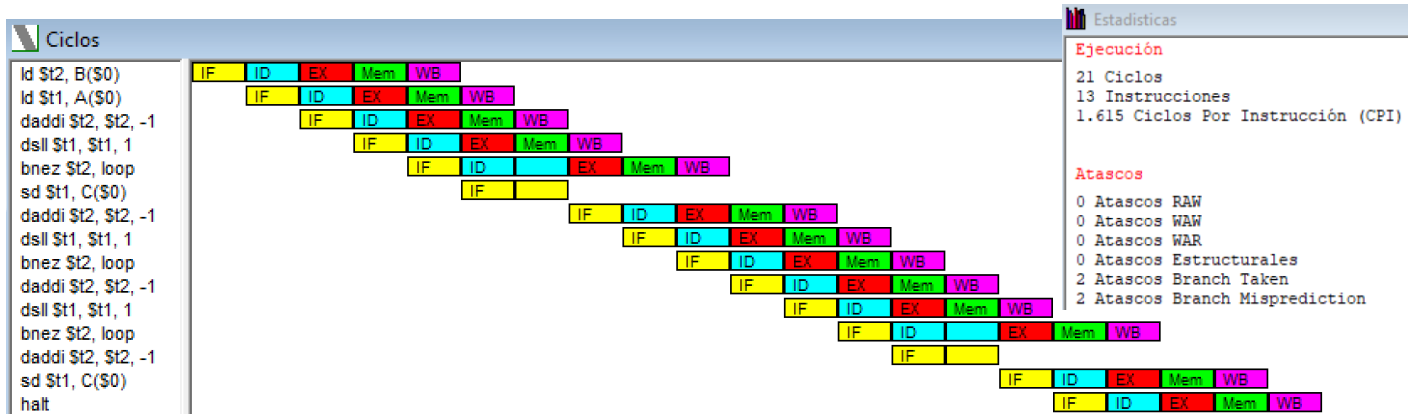
Las instrucciones descartadas no cuentan para las estadísticas: solo se consideran aquellas instrucciones cuyo respectivo ciclo de ejecución se haya completado desde el principio (etapa IF) hasta el final (etapa WB), independientemente de la estructura y latencia de su etapa de ejecución Execute (EX, M o DIV según corresponda).

Ejercicio 2

a) y b)

Si se activa la técnica Branch Target Buffer (BTB), ahora en el cauce del procesador se generarán cuatro atascos en total. Dos de ellos serán BTS (Branch Taken) por el salto provocado en la instrucción “bnez” en la primera iteración del bucle. Por su parte, los otros dos se tratarán de BMS (Branch Misprediction Stalls) por el salto sin efectuarse en la misma instrucción “bnez”, pero en este caso en la última repetición del lazo, ya que en la tabla BTB almacenada

en caché se había predicho para esta instrucción que el salto se llevaría a cabo.



En este caso, en el cual B es igual a 3, al ser muy reducido el número de iteraciones del programa, la cantidad de atascos generados en el cauce del procesador es mayor aquí que en el ejercicio anterior, en el cual no se había utilizado la técnica BTB, por lo que el CPI en este nuevo escenario es mayor y la ejecución del programa, más ineficiente.

c)

Para B = 1000 (solo se incluyen capturas de ejecuciones parciales del programa centradas exclusivamente en los atascos generados en el cauce):



Al activarse la técnica Branch Target Buffer (BTB), en el cauce del procesador se han generado cuatro atascos en total, igual que en el inciso a. Dos de ellos serán BTS (Branch Taken) por el salto provocado en la instrucción “bnez” en la primera iteración del bucle. Por su parte, los otros dos se tratarán de BMS (Branch Misprediction Stalls) por el salto sin efectuarse en la misma instrucción “bnez”, pero en este caso en la última repetición del lazo, ya que en la tabla BTB almacenada en caché se había predicho para esta instrucción que el salto se llevaría a cabo. En las demás iteraciones no existirá atasco alguno porque en todas ellas la predicción de la tabla BTB para la instrucción será correcta: el salto asociado a “bnez” se tomará.

Si se compara este caso con el del ejercicio 1 para B igual a 1000, al ser en consecuencia más elevado el número de iteraciones del programa, se comprueba que la cantidad de atascos es muy inferior aquí, por lo que el CPI resulta considerablemente menor (de hecho, es prácticamente igual a 1) y la ejecución del programa, más eficiente.

d)

Al ejecutar un loop simple con N iteraciones, si se habilita BTB, se producen 2 (dos) atascos de tipo BTS y 2 (dos) atascos de tipo BMS.

e)

Analizando las observaciones indicadas en los incisos anteriores, se puede concluir fácilmente que resulta especialmente recomendable utilizar la técnica BTB en programas que presenten bucles simples con un elevado número de iteraciones.

Ejercicio 3

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

        .code
        ld $t1, MAX($0)
        daddi $t2, $0, 0
        daddi $t3, $0, 6
loop:   ld $t4, A($t2)
        slt $t5, $t1, $t4
        beqz $t5, chico
        daddi $t1, $t4, 0
chico:  daddi $t2, $t2, 8
        daddi $t3, $t3, -1
        bnez $t3, loop
        sd $t1, MAX($0)
        halt
```

a)

- Instrucción “beqz”:
 - ❖ 1° iteración → el salto no se toma. La instrucción no está en la tabla BTB: no hay atascos Branch Taken (BTS).
 - ❖ 2° iteración → el salto se toma. La instrucción no está en la tabla BTB: 2 atascos Branch Taken (BTS) y la instrucción se agrega a la tabla (solo se agregan las instrucciones que provoquen saltos).
 - ❖ 3° iteración → el salto no se toma. La instrucción está en la tabla BTB: 2 atascos Branch Misprediction (BMS) y la instrucción se borra de la tabla (solo se preservan las instrucciones que hayan provocado saltos).
 - ❖ 4° iteración → el salto se toma. La instrucción no está en la tabla BTB: 2 atascos Branch Taken (BTS) y la instrucción se agrega a la tabla.
 - ❖ 5° iteración → el salto no se toma. La instrucción está en la tabla BTB: 2 atascos Branch Misprediction (BMS) y la instrucción se borra de la tabla.

- ❖ 6° iteración → el salto se toma. La instrucción no está en la tabla BTB: 2 atascos Branch Taken (BTS) y la instrucción se agrega a la tabla.
- ❖ Total de atascos por dependencias de control → 10 atascos: 6 BTS y 4 BMS
- Instrucción “bnez”:
 - ❖ 1° iteración → el salto se toma. La instrucción no está en la tabla BTB: 2 atascos Branch Taken (BTS) y la instrucción se agrega a la tabla.
 - ❖ 2° a 5° iteración → el salto se toma. La instrucción está en la tabla BTB: no hay atascos Branch Misprediction (BMS).
 - ❖ 6° iteración → el salto no se toma. La instrucción está en la tabla BTB: 2 atascos Branch Misprediction (BMS) y la instrucción se borra de la tabla.
 - ❖ Total de atascos por dependencias de control → 4 atascos: 2 BTS y 2 BMS
- Total de atascos del programa por dependencias de control → 14 atascos: 8 BTS y 6 BMS

b)

En las siguientes capturas se reflejan las estadísticas de la ejecución del programa activando y desactivando la técnica Branch Target Buffer (BTB) en la primera y segunda figura respectivamente:

Estadísticas
Ejecución
80 Ciclos
44 Instrucciones
1.818 Ciclos Por Instrucción (CPI)
Atascos
18 Atascos RAW
0 Atascos WAW
0 Atascos WAR
0 Atascos Estructurales
8 Atascos Branch Taken
6 Atascos Branch Misprediction

Estadísticas
Ejecución
74 Ciclos
41 Instrucciones
1.805 Ciclos Por Instrucción (CPI)
Atascos
18 Atascos RAW
0 Atascos WAW
0 Atascos WAR
0 Atascos Estructurales
11 Atascos Branch Taken
0 Atascos Branch Misprediction

Como se observará, a pesar de que el bucle posee múltiples iteraciones, su flujo de ejecución no puede ser considerado “simple” porque cuenta internamente con una instrucción de transferencia de control condicional que se ejecuta una vez en cada iteración. Dicha instrucción, además, al variar en cada oportunidad el resultado de la evaluación de su condición de salto con respecto a la repetición anterior, genera demasiados atascos BTS y BMS en el cauce del CPU utilizando BTS hasta un extremo tal que se concluye, observando los CPI para ambos escenarios de ejecución del programa, que resulta más eficiente desactivar el BTB al menos para este programa en particular.

Ejercicio 4

a)

A continuación, se presentan nuevamente el programa, el cauce de la CPU y las estadísticas de la ejecución ofrecidas por el simulador, todos ellos previamente exhibidos en el inciso a del ejercicio 3 de la parte 2 (“Atascos RAW”), desactivando el Forwarding, BTB y Delay Slot. Cabe mencionar que, al ya haberse introducido formalmente en esta instancia de la presente práctica los atascos por dependencia de control, también se han incluido en las estadísticas aquí mostradas los atascos Branch Taken (BTS) y Branch Misprediction (BMS).

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

.code
ld $t1, A($0)
ld $t2, B($0)
loop: dsll $t1, $t1, 1
      daddi $t2, $t2, -1
      bnez $t2, loop
      sd $t1, C($0)
      halt
```

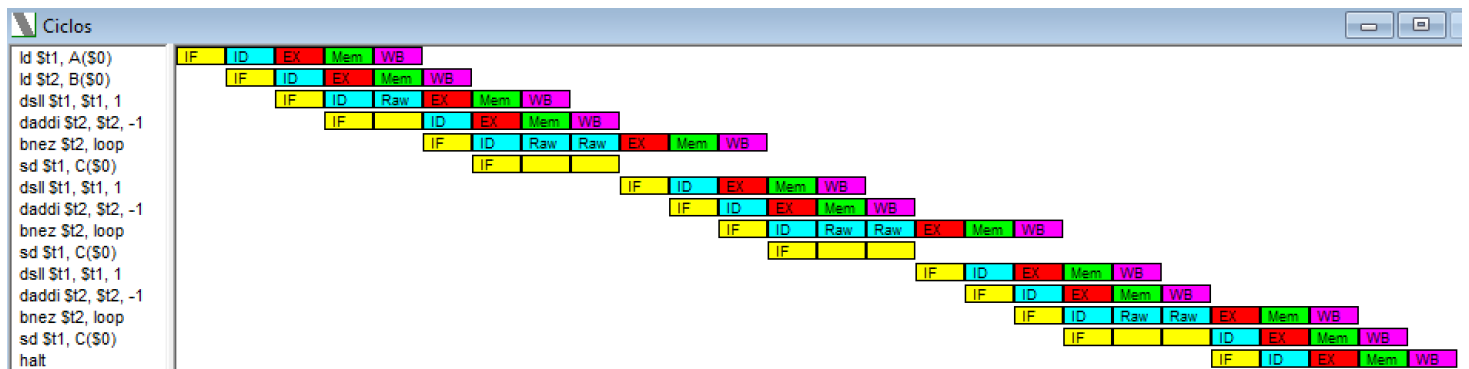
Estadísticas

Ejecución

26 Ciclos
13 Instrucciones
2.000 Ciclos Por Instrucción (CPI)

Atascos

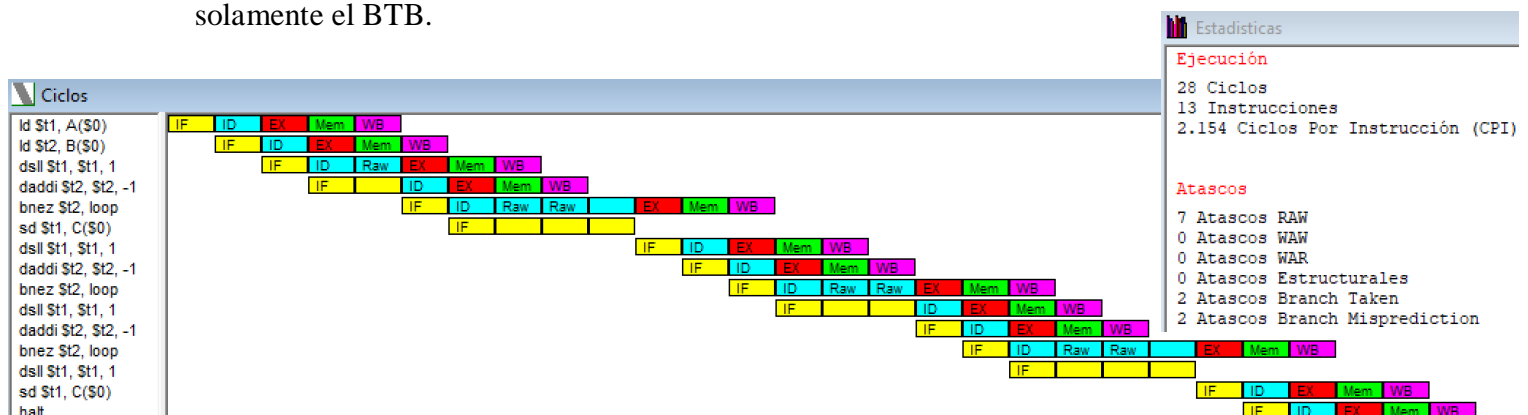
7 Atascos RAW
0 Atascos WAW
0 Atascos WAR
0 Atascos Estructurales
2 Atascos Branch Taken
0 Atascos Branch Misprediction



b)

Si solamente se habilita la técnica BTB, pero se mantiene desactivado el Forwarding (BTB y Delay Slot son mutuamente excluyentes: no pueden activarse simultáneamente), centrando el análisis exclusivamente en las dependencias de control, se generan, para empezar, dos atascos Branch Taken (BTS) en la primera iteración del bucle porque la instrucción “bnez” no se encuentra en la tabla BTB, pero el salto se toma. Por su parte, se provocan otros dos atascos Branch Misprediction (BMS) en la última iteración porque “bnez” está en la tabla BTB, pero el salto no se efectúa.

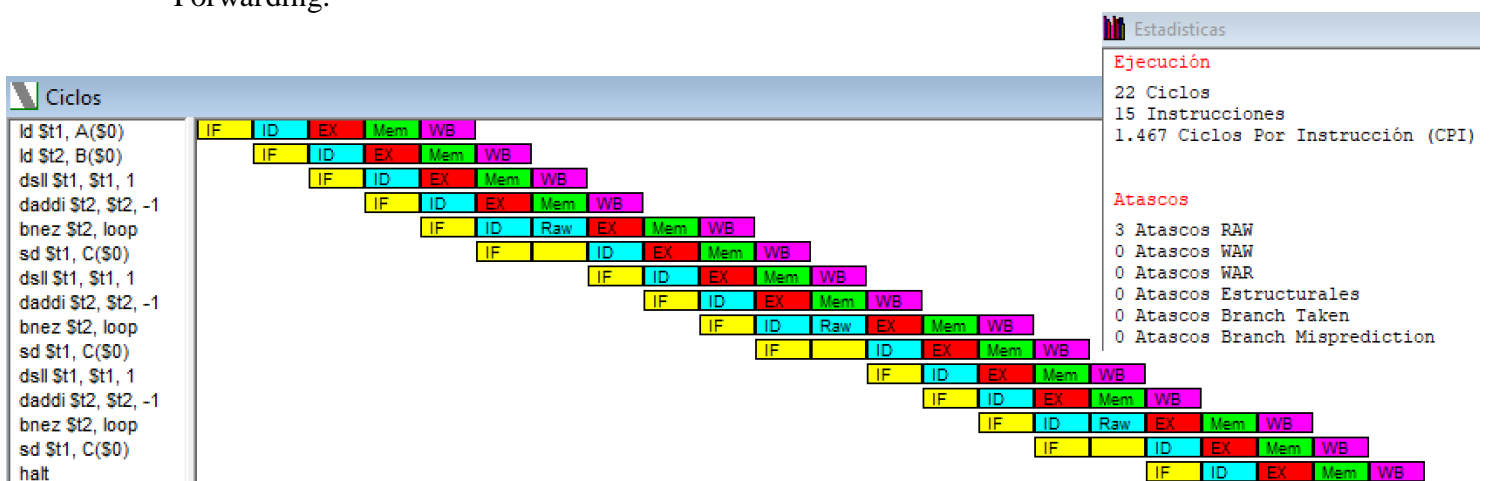
Por lo tanto, en comparación con el inciso anterior, existe el mismo número de atascos RAW ya que nuevamente se ejecutó el programa sin activar el Forwarding, pero se presentan dos atascos adicionales por dependencias de control de tipo BMS. Ergo, la cantidad total de atascos generados en el cauce del procesador será mayor aquí, por lo que el CPI será mayor en este nuevo escenario alternativo: la eficiencia del programa es menor en caso de habilitarse solamente el BTB.



c)

Si se activan tanto el Forwarding como el Delay Slot (en consecuencia, el BTB permanece desactivado), se reduce drásticamente la cantidad de atascos RAW por el uso del Forwarding tal como fue explicado en el inciso b del ejercicio 3 de la práctica 2. Con respecto a los atascos por dependencias de control, ahora no existe ninguno en el cauce debido al Delay Slot. No obstante, cada uno de los huecos de retardo de salto generados al tomarse el salto de la instrucción “bnez” en todas las iteraciones del bucle (excepto la última, por obvias razones) es ocupado por la instrucción “sd” dado que se trata de aquella situada inmediatamente a continuación de la instrucción “halt”. Esta acción no sería adecuada dado que, a pesar de no afectar el comportamiento original del programa ni el resultado finalmente almacenado en la variable “C” en memoria, sí se preservan en ella en forma temporal e innecesaria todos los valores parciales del registro \$t1 obtenidos en cada iteración del bucle. De esta manera, no alcanzaría únicamente con verificar el CPI del programa en el nuevo escenario para evaluar la eficiencia de su ejecución: al existir varias instrucciones ejecutadas que no realizan ninguna tarea útil, debe comprobarse que el número total de ciclos del programa ejecutados sea menor al del caso original. Solo en caso afirmativo podrá entonces afirmarse que la nueva ejecución es más eficiente debido a que logra llevarse a cabo en ella la misma magnitud de trabajo útil en un tiempo menor. Se observará que existen ciertas similitudes entre este ejemplo y el inciso e del ejercicio 3 de la parte 1, en el que se habían incorporado al programa 20 instrucciones NOP, que en realidad no realizan operación alguna, para evaluar la relación entre el CPI y la eficiencia de la ejecución del programa.

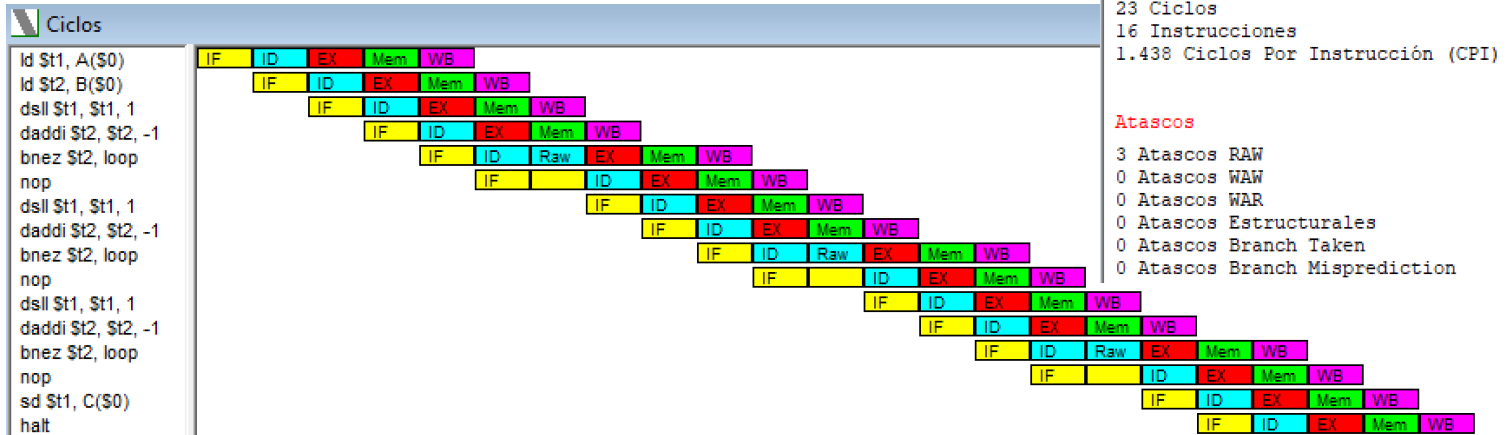
Observando las estadísticas de la ejecución, se concluye que el número de ciclos ejecutados para finalizar el programa es inferior al del escenario elegido en el inciso a (el más eficiente hasta el momento), por lo que el programa ahora se ejecuta aun más eficientemente. Sin embargo, es altamente probable que esta mejora no se deba a un uso adecuado de la técnica Delay Slot, sino a la disminución de los atascos RAW como consecuencia de la activación del Forwarding.



d)

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

```
.code
ld $t1, A($0)
ld $t2, B($0)
loop: dsl1 $t1, $t1, 1
      daddi $t2, $t2, -1
      bnez $t2, loop
      nop          # Nueva instrucción incorporada
      sd $t1, C($0)
halt
```



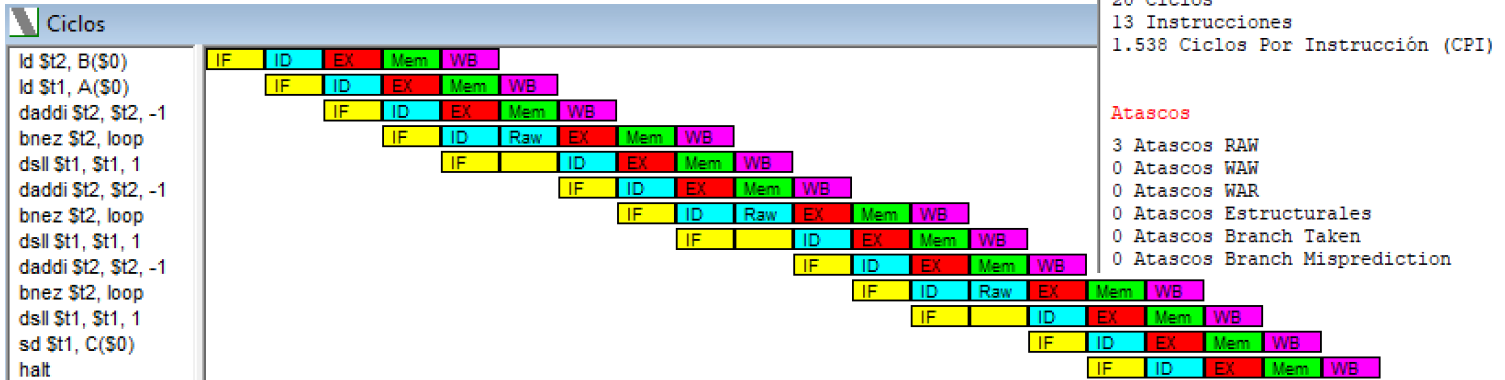
La incorporación de una instrucción “nop” inmediatamente a continuación de la instrucción de transferencia de control “bnez” implica que, a diferencia del inciso anterior, ahora será esta nueva instrucción en lugar de “sd” aquella a ser ineludiblemente ejecutada en cada hueco de retardo de salto generado en el cauce del CPU en caso de efectuarse el salto asociado a “bnez”. Este evento, como ya es conocido, ocurrirá en todas las iteraciones del bucle del programa, excepto la última, luego de la cual se ejecutarán tanto “nop” como “sd” antes de completarse la instrucción “halt” y finalizar el programa. Por lo tanto, la cantidad total de instrucciones ejecutadas se incrementará en comparación con el inciso previo y, en consecuencia, el CPI disminuirá. Este fenómeno podría inducir a presumir que este escenario de ejecución es más eficiente. No obstante, la presencia de las instrucciones “nop”, al no aportar realmente ningún trabajo útil, indica que esta presunción podría ser errónea: debe compararse el número de ciclos de ejecución de ambos programas y solo aquél que requiera la menor cantidad de ciclos para completar su tarea presentará la mayor eficiencia. Una vez llevada a cabo la comparación, se concluye que la ejecución aquí planteada es de hecho más ineficiente que la anterior dado que necesita un ciclo más de ejecución para finalizarse (el trabajo útil efectuado es idéntico al del inciso anterior: “nop” solo hace operaciones nulas).

e)

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

```
.code
ld $t2, B($0)          # Estas dos instrucciones fueron
ld $t1, A($0)          # intercambiadas entre sí
```

```
loop: daddi $t2, $t2, -1
      bnez $t2, loop
      dsll $t1, $t1, 1      # Esta instrucción ha sido reubicada
      sd $t1, C($0)
      halt
```



Aunque esta acción no se solicitaba en la consigna, las dos primeras instrucciones fueron intercambiadas entre sí para evitar la generación de nuevos atascos RAW en el cauce de la CPU como consecuencia de la reubicación de la instrucción “dsll”. De esta manera, se demuestra fácilmente que, al ocuparse los huecos de retardo de salto del programa con una instrucción útil como “dsll”, junto con el uso activo del Forwarding, se obtiene la modalidad de ejecución más eficiente posible del programa para este ejercicio. Asimismo, al realizarse siempre y en todo momento una tarea útil, alcanza con obtener y estudiar el CPI para evaluar la eficiencia de esta ejecución: no es necesario analizar la cantidad de ciclos de cada programa. También, por otra parte, puede observarse que la cantidad de instrucciones ejecutadas coincide con aquella correspondiente al comienzo del presente ejercicio, tal como se requería en la consigna.

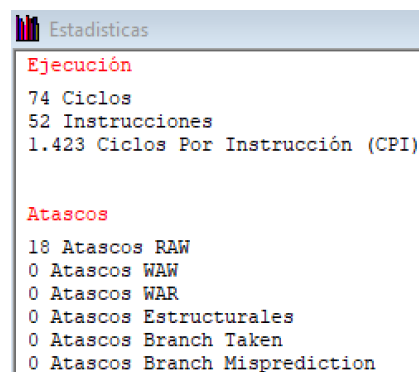
Escenario Estadística	Sin mejoras (inciso a)	BTB (inciso b)	DS (inciso c)	DS y NOP (inciso d)	DS y reordenamiento (inciso e)
BTS	2	2	0	0	0
BMS	0	2	0	0	0
CPI	2	2,154	1,467	1,438	1,538
N° de instrucciones ejecutadas	13	13	15	16	13

Ejercicio 5

a)

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

.code
ld $t1, MAX($0)
daddi $t2, $0, 0
daddi $t3, $0, 6
loop:   ld $t4, A($t2)
        slt $t5, $t1, $t4
        beqz $t5, chico
```




```

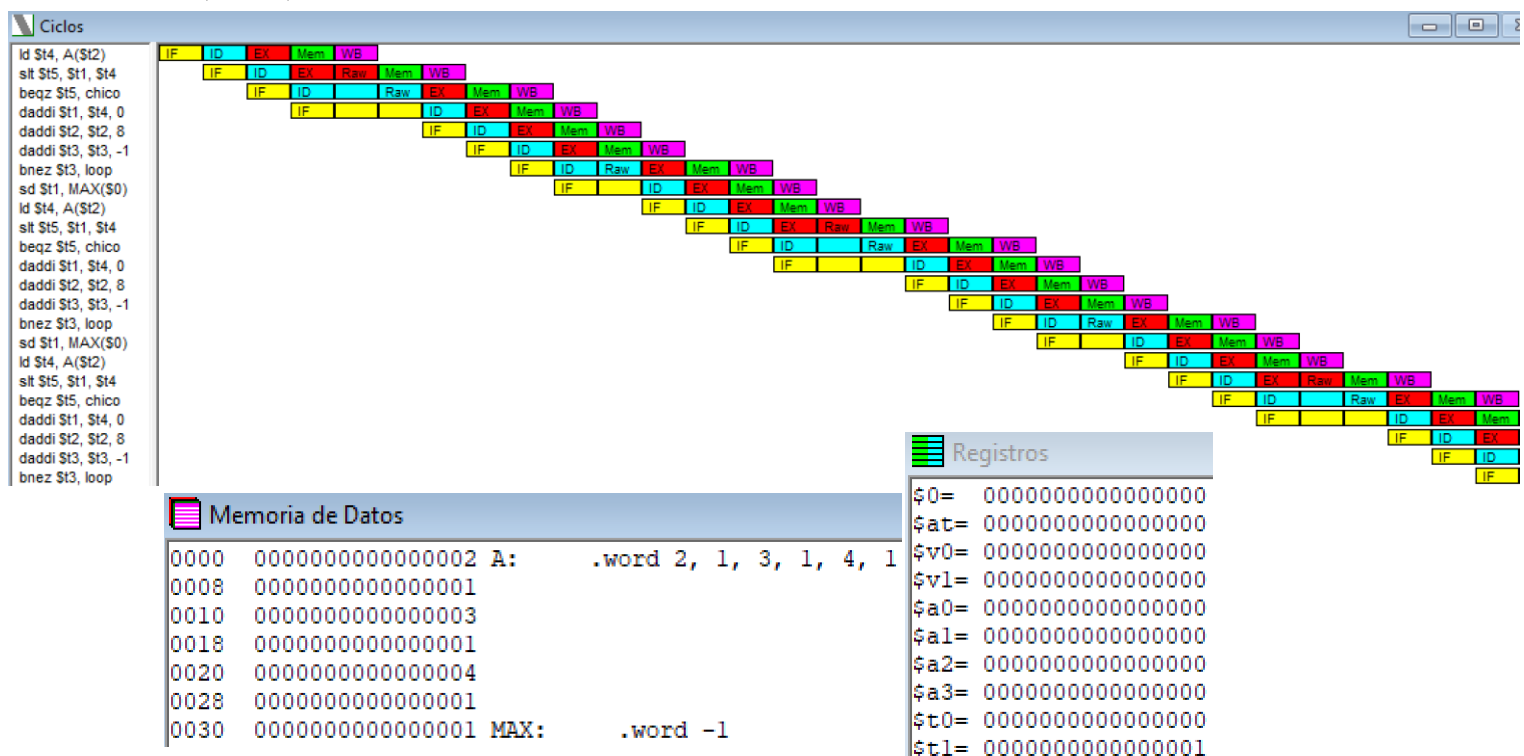
chico:  daddi $t1, $t4, 0
        daddi $t2, $t2, 8
        daddi $t3, $t3, -1
        bnez $t3, loop
        sd $t1, MAX($0)
        halt

```

Al activarse la técnica Delay Slot, la instrucción “daddi” por un lado y la instrucción “sd” por otro, ubicadas a continuación de las instrucciones de transferencia de control “beqz” y “bnez” respectivamente, se ejecutarán ineludiblemente según corresponda ya que cada una ocupará el hueco de retardo generado en el cauce de la CPU al provocarse un salto en la ejecución de su instrucción de salto asociada.

La ejecución reiterada de la instrucción “sd”, aunque innecesaria ya que solo debería almacenarse en la variable MAX en memoria el valor máximo obtenido en el registro \$t1 al final del programa y no los máximos parciales determinados durante el transcurso de la ejecución del mismo, no afectaría en términos generales su comportamiento general ya que eventualmente terminaría guardándose en MAX el último máximo calculado en \$t1.

Sin embargo, la ejecución incondicional de la instrucción “daddi” sí altera el funcionamiento del programa, impidiendo que cumpla correctamente su tarea asignada, debido a que actualizará permanentemente, es decir, en todas las iteraciones del bucle, el valor del registro \$t1 con el número recientemente leído del vector A independientemente de si efectivamente se trata o no de un nuevo máximo, por lo que, al finalizar el programa, \$t1 no necesariamente contiene el máximo valor del vector sino su último número, y éste será el que luego se almacene en MAX antes de finalizar el programa. De hecho, podrá observarse en las siguientes capturas que el valor almacenado tanto en \$t1 como en MAX una vez concluido el programa es 1 (uno), el último del vector, el cual es incorrecto dado que no se trata del máximo, es decir, el número 4 (cuatro).

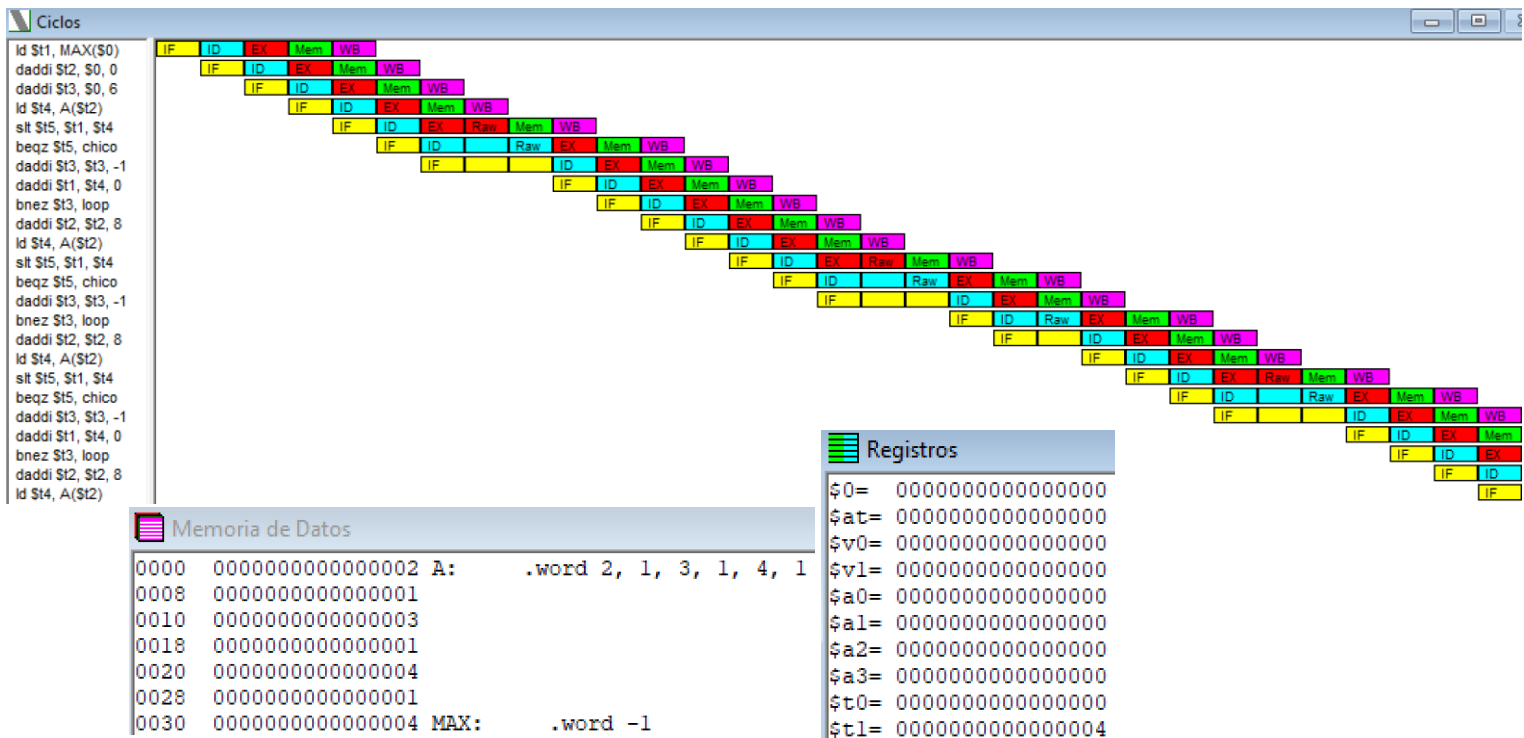


b)

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

.code
ld $t1, MAX($0)
daddi $t2, $0, 0
daddi $t3, $0, 6
loop:   ld $t4, A($t2)
        slt $t5, $t1, $t4
        beqz $t5, chico
        daddi $t3, $t3, -1 # Esta instrucción ha sido reubicada
        daddi $t1, $t4, 0
chico:  bnez $t3, loop
        daddi $t2, $t2, 8 # Esta instrucción ha sido reubicada
        sd $t1, MAX($0)
        halt
```

Estadísticas	
Ejecución	
63 Ciclos	
44 Instrucciones	
1.432 Ciclos Por Instrucción (CPI)	
Atascos	
15 Atascos RAW	
0 Atascos WAW	
0 Atascos WAR	
0 Atascos Estructurales	
0 Atascos Branch Taken	
0 Atascos Branch Misprediction	

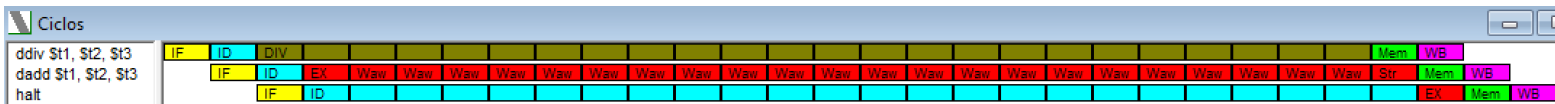


Se puede observar que, al reordenarse el programa para ubicar en los huecos de retardo generados en el cauce instrucciones apropiadas de manera tal que siempre se realicen tareas útiles sin modificar el funcionamiento original, se logran todos los objetivos propuestos: la cantidad de ciclos disminuye (mejorando así la eficiencia de la ejecución ya que en este escenario no corresponde comparar los CPI), la última instrucción “sd” se ejecuta en una única ocasión inmediatamente antes de finalizar el programa y tanto en el registro \$t1 como en la variable MAX termina almacenándose efectivamente el valor correcto: 4 (cuatro), el número máximo del vector A.

Parte 4: Atascos por WAR, WAW y estructurales

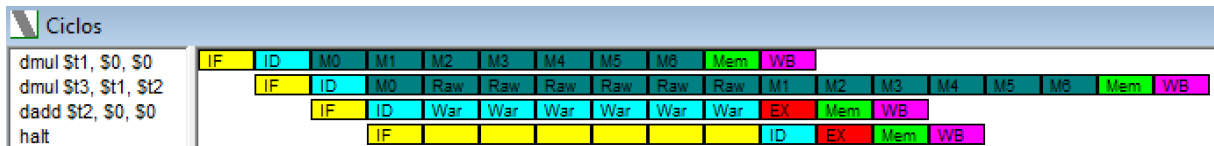
Ejercicio 1

	Ejecución	Atascos
.code		0 Atascos RAW
ddiv \$t1, \$t2, \$t3	30 Ciclos	22 Atascos WAW
dadd \$t1, \$t2, \$t3	3 Instrucciones	0 Atascos WAR
halt	10.000 Ciclos Por Instrucción (CPI)	1 Atasco Estructural
		0 Atascos Branch Taken
		0 Atascos Branch Misprediction



En el programa previo, ejecutado con el Forwarding activo, se generan 22 atascos WAW por una dependencia de datos de escritura entre las instrucciones “ddiv” y “dadd” con respecto al valor del registro \$t1. Al tener un ciclo de ejecución más largo y lento, conformado por una etapa Execute (DIV) con una latencia de 24 ciclos de reloj, la instrucción “ddiv”, anterior a “dadd” en el cauce de la CPU, actualizaría incorrectamente \$t1 después de la escritura realizada en él por “dadd”. Para evitar este evento, debe demorarse la ejecución de “dadd” hasta que “ddiv” escriba efectivamente su valor en \$t1 y se resuelva la dependencia para que “dadd” pueda proceder a efectuar su propia actualización de \$t1 sin inconveniente alguno.

	Ejecución	Atascos
.code		6 Atascos RAW
dmul \$t1, \$0, \$0	18 Ciclos	0 Atascos WAW
dmul \$t3, \$t1, \$t2	4 Instrucciones	6 Atascos WAR
dadd \$t2, \$0, \$0	4.500 Ciclos Por Instrucción (CPI)	0 Atascos Estructurales
halt		0 Atascos Branch Taken
		0 Atascos Branch Misprediction

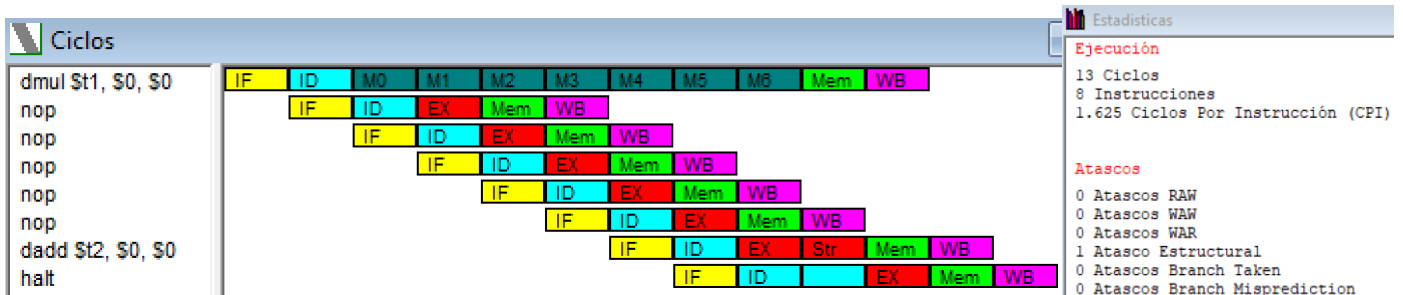


En el programa previo, ejecutado con el Forwarding activo, se generan 6 atascos WAR por una dependencia de datos de escritura entre las instrucciones “dmul” (segunda instrucción) y “dadd” con respecto al valor del registro \$t2. Al tener un ciclo de ejecución más largo y lento, conformado por una etapa Execute (M) con una latencia de 7 ciclos de reloj, la instrucción “dmul”, anterior a “dadd” en el cauce de la CPU, leería incorrectamente \$t2 después de que “dadd” lo haya actualizado. Para evitar este evento, debe demorarse la ejecución de “dadd” hasta que “dmul” lea efectivamente el valor de \$t2 y se resuelva la dependencia para que “dadd” pueda proceder a efectuar su actualización de \$t2 sin inconveniente alguno.

Ejercicio 2

a)

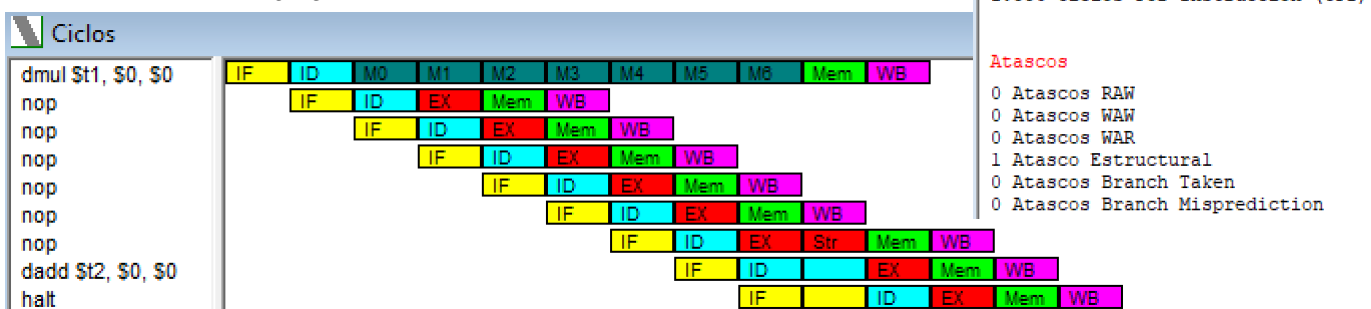
```
.code
dmul $t1, $0, $0
nop
nop
nop
nop
nop
dadd $t2, $0, $0
halt
```



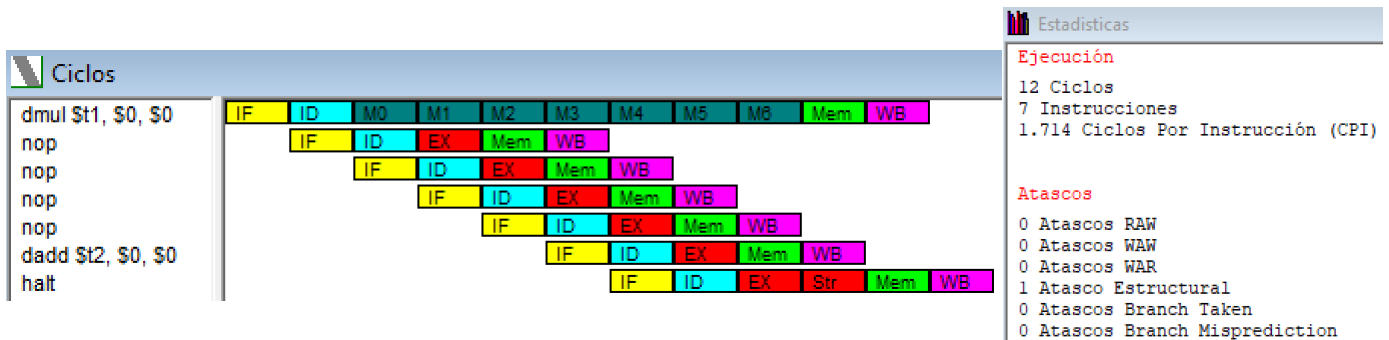
En el programa previo, ejecutado con el Forwarding activo, se genera un atasco estructural en la instrucción “dadd” a causa de que tanto ésta como “dmul” pretenden ejecutar simultáneamente la misma etapa de sus respectivos ciclos de ejecución, MEM en este caso. Al no encontrarse este paralelismo permitido ni soportado por la segmentación implementada en el cauce del procesador, debe demorarse “dadd” hasta que “dmul” complete su etapa MEM y se libere así esta última para que “dadd” pueda proceder a efectuarla. Se observará que este conflicto existe incluso si en la etapa crítica no se lleva a cabo realmente ninguna tarea útil: tanto “dmul” como “dadd” permanecen ociosas durante la ejecución de la etapa MEM. Finalmente, el atasco se provoca en “dadd” en lugar de “dmul” porque “dadd” comenzó a ejecutarse en el programa después de “dmul”. De esta manera, se preserva el orden de ejecución original de las instrucciones, garantizándose que bajo ninguna circunstancia una instrucción del programa se complete luego de que finalice otra que comenzó a ejecutarse después que ella.

b)

```
.code
dmul $t1, $0, $0
nop
nop
nop
nop
nop
nop
nop # Se ha añadido una instrucción NOP
dadd $t2, $0, $0
halt
```



```
.code
dmul $t1, $0, $0
nop
nop
nop
nop
nop # Se ha eliminado una instrucción NOP
dadd $t2, $0, $0
halt
```

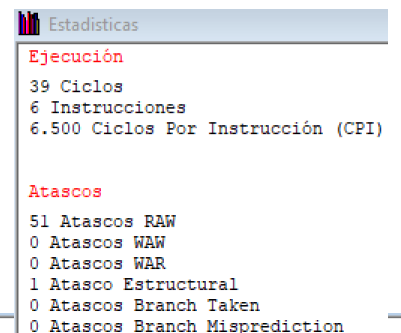


En los dos programas previos, ejecutado con el Forwarding activo, continúa generándose un atasco estructural. La única diferencia consiste en que, mientras que en el primero se demora la nueva instrucción “nop” añadida, en el segundo se retrasa la instrucción “halt”. La causa del atasco permanece inalterable: tanto la instrucción demorada, ya sea “nop” o “halt” según el caso, como “dmul” pretenden ejecutar simultáneamente la misma etapa de sus respectivos ciclos de ejecución, MEM en este escenario. Al no encontrarse este paralelismo permitido ni soportado por la segmentación implementada en el cauce del procesador, debe demorarse “nop” o “halt” hasta que “dmul” complete su etapa MEM y se libere así esta última para que “nop” o “halt” pueda proceder a efectuarla. Se confirma asimismo la observación indicada en el inciso a, según la cual este conflicto existía incluso si en la etapa crítica no se llevaba a cabo realmente ninguna tarea útil: tanto “nop” o “halt” como “dadd” permanecen ociosas durante la ejecución de la etapa MEM. Finalmente, el atasco se provoca en “nop” o “halt” en lugar de “dmul” porque “nop” o “halt” comenzó a ejecutarse en el programa después de “dmul”. De esta manera, se preserva el orden de ejecución original de las instrucciones, garantizándose que bajo ninguna circunstancia una instrucción del programa se complete luego de que finalice otra que comenzó a ejecutarse después que ella.

Ejercicio 3

```

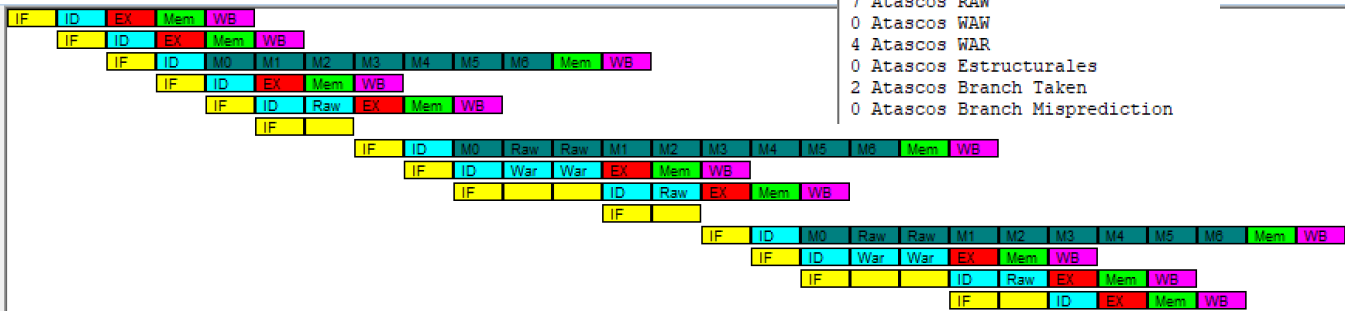
# resto: calcula en $t4 el resto de $t1 div $t2
.code
daddi $t1, $0, 30      # a = 30
daddi $t2, $0, 4        # b = 4
ddiv $t3, $t1, $t2      # c = a div b = 7
dmul $t3, $t3, $t2      # c * b = 7 * 4 = 28
dsub $t4, $t1, $t3      # resto = a - c * b = 2
halt
  
```



En el programa previo, ejecutado con el Forwarding activo, se genera un atasco estructural en la instrucción “dsub” a causa de que tanto ésta como “dmul” pretenden ejecutar simultáneamente la misma etapa de sus respectivos ciclos de ejecución, MEM en este caso. Al no encontrarse este paralelismo permitido ni soportado por la segmentación implementada en el cauce del procesador, debe demorarse “dsub” hasta que “dmul” complete su etapa MEM y se libere así esta última para que “dsub” pueda proceder a efectuarla.

```
# factorial: calcula en $t2 el factorial de $t1
.code
daddi $t1, $0, 3      # n = 3
daddi $t2, $0, 1      # f = 1
loop: dmul $t2, $t2, $t1 # f = f * n
daddi $t1, $t1, -1    # n = n - 1
bnez $t1, loop
halt
```

```
daddi $t1, $0, 3
daddi $t2, $0, 1
dmul $t2, $t2, $t1
daddi $t1, $t1, -1
bnez $t1, loop
halt
dmul $t2, $t2, $t1
daddi $t1, $t1, -1
bnez $t1, loop
halt
dmul $t2, $t2, $t1
daddi $t1, $t1, -1
bnez $t1, loop
halt
```



Estadísticas

Ejecución

27 Ciclos
12 Instrucciones
2.250 Ciclos Por Instrucción (CPI)

Atascos

7 Atascos RAW
0 Atascos WAW
4 Atascos WAR
0 Atascos Estructurales
2 Atascos Branch Taken
0 Atascos Branch Misprediction

En el programa previo, ejecutado con el Forwarding activo, se generan 2 atascos WAR en cada iteración del bucle por una dependencia de datos de escritura entre las instrucciones “dmul” y “daddi” (cuarta instrucción del programa) con respecto al valor del registro \$t1. Al tener un ciclo de ejecución más largo y lento, conformado por una etapa Execute (M) con una latencia de 7 ciclos de reloj, la instrucción “dmul”, anterior a “daddi” en el cauce de la CPU, leería incorrectamente \$t1 después de que “daddi” lo haya actualizado. Para evitar este evento, debe demorarse la ejecución de “daddi” hasta que “dmul” lea efectivamente el valor de \$t1 y se resuelva la dependencia para que “daddi” pueda proceder a efectuar su actualización de \$t1 sin inconveniente alguno.

Ejercicio 4

Los dos programas propuestos en el presente ejercicio fueron ejecutados con el Forwarding activado.

a)

Producto y suma:

```
.data
A: .word 7
B: .word 4
MULT: .word 0
SUMA: .word 0

.code
ld $t1, A($0)
ld $t2, B($0)
dmul $t3, $t1, $t2
sd $t3, MULT($0)
dadd $t3, $t1, $t2
sd $t3, SUMA($0)
halt
```

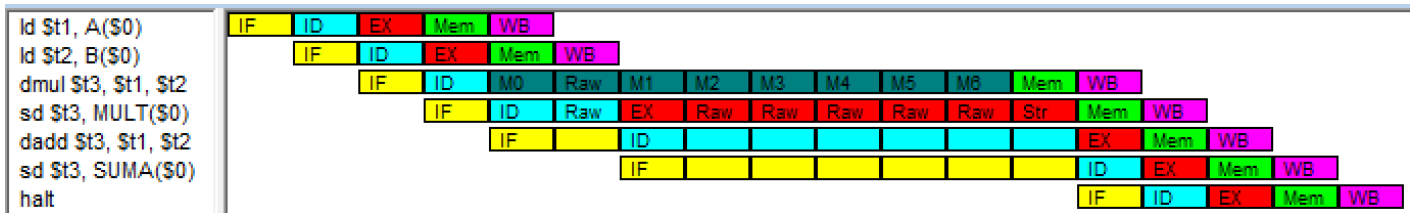
Estadísticas

Ejecución

18 Ciclos
7 Instrucciones
2.571 Ciclos Por Instrucción (CPI)

Atascos

7 Atascos RAW
0 Atascos WAW
0 Atascos WAR
1 Atasco Estructural
0 Atascos Branch Taken
0 Atascos Branch Misprediction



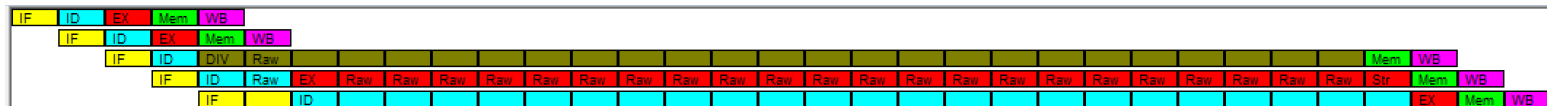
b)

Promedio:

```
.data
SUMA: .word 10
CANT: .word 4
PROM: .word 0
```

```
.code
ld $t1, SUMA($0)
ld $t2, CANT($0)
ddiv $t3, $t1, $t2
sd $t3, PROM($0)
halt
```

Estadísticas
Ejecución
33 Ciclos
5 Instrucciones
6.600 Ciclos Por Instrucción (CPI)
Atascos
24 Atascos RAW
0 Atascos WAW
0 Atascos WAR
1 Atasco Estructural
0 Atascos Branch Taken
0 Atascos Branch Misprediction



Parte 5: Ejercicios de repaso o tipo parcial

Ejercicio 1

```
.data
A: .word 2, 1, 3
SUM: .word 0
```

```
.code
daddi $t2, $0, 0
ld $t1, SUM($0)
daddi $t3, $0, 3
loop: ld $t4, A($t2)
      dadd $t1, $t1, $t4
      daddi $t2, $t2, 8      # Instrucción agregada para que la
                          # suma calculada sea correcta
      daddi $t3, $t3, -1
      bnez $t3, loop
      sd $t1, SUM($0)
      halt
```

a)

A continuación, se representa el cauce segmentado del procesador a través del tiempo en caso de ejecutarse el presente programa desactivando el Forwarding, el Branch Target Buffer y el Delay Slot.

Cabe indicar que resulta posible exhibir sin inconveniente alguno en forma separada e independiente el cauce del procesador previo, durante y posterior a la ejecución del bucle. Sin embargo, puede comprobarse asimismo la existencia de dependencias de datos de lectura entre, por un lado, las instrucciones “daddi \$t2, \$0, 0”, anterior al lazo, y “ld \$t4, A(\$t2)”, perteneciente al bucle (con respecto al valor del registro \$t2), y, por otro, las instrucciones “dadd \$t1, \$t1, \$t4”, correspondiente al lazo, y “sd \$t1, SUM(\$0)”, ubicada después del bucle (con respecto al valor del registro \$t1). Aun así, al presentar cada par de instrucciones interdependientes una separación entre sí en el programa de al menos dos instrucciones, puede garantizarse, incluso con el Forwarding desactivado, que las respectivas dependencias se alcanzarán a resolver antes de que sea inevitablemente necesario generar efectivamente un atasco RAW en el cauce del procesador.

Finalmente, la cantidad de ciclos de ejecución del programa fue obtenida utilizando la fórmula definida en la introducción del presente trabajo práctico para el cálculo del CPI. Dicha fórmula es apta y válida para este escenario ya que todas las instrucciones del programa presentan el mismo ciclo de ejecución tradicional: una estructura conformada por cinco etapas con una latencia de un ciclo cada una.

Ejecución del programa previa al lazo							
Instrucción	Cauce del procesador						
daddi \$t2, \$0, 0	IF	ID	EX	MEM	WB		
ld \$t1, SUM(\$0)		IF	ID	EX	MEM	WB	
daddi \$t3, \$0, 3			IF	ID	EX	MEM	WB

Ejecución del programa dentro del lazo (no hay BTS en la última iteración)													
Instrucción	Cauce del procesador												
ld \$t4, A(\$t2)	IF	ID	EX	MEM	WB								
dadd \$t1, \$t1, \$t4		IF	RAW	RAW	ID	EX	MEM	WB					
daddi \$t2, \$t2, 8			IF	-----	-----	ID	EX	MEM	WB				
daddi \$t3, \$t3, -1						IF	ID	EX	MEM	WB			
bnez \$t3, loop							IF	RAW	RAW	ID	EX	MEM	WB
sd \$t1, SUM(\$0)								IF	-----	-----	BTS		

Ejecución del programa posterior al lazo								
Instrucción	Cauce del procesador							
sd \$t1, SUM(\$0)	IF	-----	-----	ID	EX	MEM	WB	
halt				IF	ID	EX	MEM	WB

- Iteraciones = $N = 3$ (longitud del vector A)
- Atascos RAW = $A_{RAW} = (2 + 2) * N = (2 + 2) * 3 = 4 * 3 = 12$
- Atascos WAR = $A_{WAR} = 0$
- Atascos WAW = $A_{WAW} = 0$
- Atascos STR (estructurales) = $A_{STR} = 0$
- Atascos Branch Taken (BTS) = $N - 1 = 3 - 1 = 2$
- Atascos Branch Misprediction (BMS) = 0
- Atascos = $A_{RAW} + A_{WAR} + A_{WAW} + A_{STR} + BTS + BMS = 12 + 0 + 0 + 0 + 2 + 0 = 14$

- Instrucciones ejecutadas antes del lazo = $I_1 = 3$
- Instrucciones ejecutadas en el lazo = $I_2 = 5 * N = 5 * 3 = 15$
- Instrucciones ejecutadas luego del lazo = $I_3 = 2$
- Instrucciones = $I_1 + I_2 + I_3 = 3 + 15 + 2 = 20$
- Ciclos = Instrucciones + Atascos + 4 = $20 + 14 + 4 = 38$
- CPI = Ciclos / Instrucciones = $38 / 20 = 1,9$

b)

A continuación, se representa el cauce segmentado del procesador a través del tiempo en caso de ejecutarse el presente programa activando el Forwarding, pero manteniendo desactivados tanto el Branch Target Buffer como el Delay Slot.

Cabe indicar que resulta posible exhibir sin inconveniente alguno en forma separada e independiente el cauce del procesador previo, durante y posterior a la ejecución del bucle. Sin embargo, puede comprobarse asimismo la existencia de dependencias de datos de lectura entre, por un lado, las instrucciones “daddi \$t2, \$0, 0”, anterior al lazo, y “ld \$t4, A(\$t2)”, perteneciente al bucle (con respecto al valor del registro \$t2), y, por otro, las instrucciones “dadd \$t1, \$t1, \$t4”, correspondiente al lazo, y “sd \$t1, SUM(\$0)”, ubicada después del bucle (con respecto al valor del registro \$t1). Aun así, si se garantizó en el inciso anterior, con el Forwarding desactivado, que al presentar cada par de instrucciones interdependientes una separación entre sí en el programa de al menos dos instrucciones, las respectivas dependencias se alcanzarían a resolver antes de que fuera inevitablemente necesario generar efectivamente un atasco RAW en el cauce del procesador, la misma certeza puede ofrecerse aquí, con el Forwarding activado.

Finalmente, la cantidad de ciclos de ejecución del programa fue obtenida utilizando la fórmula definida en la introducción del presente trabajo práctico para el cálculo del CPI. Dicha fórmula es apta y válida para este escenario ya que todas las instrucciones del programa presentan el mismo ciclo de ejecución tradicional: una estructura conformada por cinco etapas con una latencia de un ciclo cada una.

Ejecución del programa previa al lazo							
Instrucción	Cauce del procesador						
daddi \$t2, \$0, 0	IF	ID	EX	MEM	WB		
ld \$t1, SUM(\$0)		IF	ID	EX	MEM	WB	
daddi \$t3, \$0, 3			IF	ID	EX	MEM	WB

Ejecución del programa dentro del lazo (no hay BTS en la última iteración)											
Instrucción	Cauce del procesador										
ld \$t4, A(\$t2)	IF	ID	EX	MEM	WB						
dadd \$t1, \$t1, \$t4		IF	ID	RAW	EX	MEM	WB				
daddi \$t2, \$t2, 8			IF	ID	-----	EX	MEM	WB			
daddi \$t3, \$t3, -1				IF	-----	ID	EX	MEM	WB		
bnez \$t3, loop						IF	RAW	ID	EX	MEM	WB
sd \$t1, SUM(\$0)							IF	-----	BTS		

Ejecución del programa posterior al lazo							
Instrucción	Cauce del procesador						
sd \$t1, SUM(\$0)	IF	-----	ID	EX	MEM	WB	
halt			IF	ID	EX	MEM	WB

- Iteraciones = $N = 3$ (longitud del vector A)
- Atascos RAW = $A_{RAW} = (1 + 1) * N = (1 + 1) * 3 = 2 * 3 = 6$
- Atascos WAR = $A_{WAR} = 0$
- Atascos WAW = $A_{WAW} = 0$
- Atascos STR (estructurales) = $A_{STR} = 0$
- Atascos Branch Taken (BTS) = $N - 1 = 3 - 1 = 2$
- Atascos Branch Misprediction (BMS) = 0
- Atascos = $A_{RAW} + A_{WAR} + A_{WAW} + A_{STR} + BTS + BMS = 6 + 0 + 0 + 0 + 2 + 0 = 8$
- Instrucciones ejecutadas antes del lazo = $I_1 = 3$
- Instrucciones ejecutadas en el lazo = $I_2 = 5 * N = 5 * 3 = 15$
- Instrucciones ejecutadas luego del lazo = $I_3 = 2$
- Instrucciones = $I_1 + I_2 + I_3 = 3 + 15 + 2 = 20$
- Ciclos = Instrucciones + Atascos + 4 = $20 + 8 + 4 = 32$
- CPI = Ciclos / Instrucciones = $32 / 20 = 1,6$

c)

```

.data
A:    .word 2, 1, 3
SUM:  .word 0

.code
daddi $t3, $0, 3      # Instrucción reubicada
ld $t1, SUM($0)
daddi $t2, $0, 0      # Instrucción reubicada
loop: ld $t4, A($t2)
      dadd $t1, $t1, $t4
      daddi $t2, $t2, 8
      daddi $t3, $t3, -1
      bnez $t3, loop
      sd $t1, SUM($0)
      halt

```

Al situarse ahora las instrucciones “daddi \$t2, \$0, 0” y “ld \$t4, A(\$t2)” en forma inmediatamente consecutiva, es decir, sin encontrarse separadas por ninguna otra instrucción, la dependencia de datos de lectura existente entre ambas con respecto al valor del registro \$t2 no alcanza a resolverse antes de que “ld” requiera ineludiblemente leer dicho valor para realizar su tarea. Por lo tanto, con el Forwarding desactivado, deberán generarse en el cauce del procesador dos atascos RAW adicionales solamente en la primera iteración del bucle, hasta que la dependencia logre resolverse.

A continuación, se representa el cauce segmentado del procesador a través del tiempo en caso de ejecutarse este nuevo programa desactivando el Forwarding, el Branch Target Buffer y el

Delay Slot. Únicamente se exhiben las primeras seis instrucciones ya que su análisis resulta especialmente relevante para el escenario planteado en este inciso.

Ejecución inicial del programa														
Instrucción	Cauce del procesador													
daddi \$t3, \$0, 3	IF	ID	EX	MEM	WB									
ld \$t1, SUM(\$0)		IF	ID	EX	MEM	WB								
daddi \$t2, \$0, 0			IF	ID	EX	MEM	WB							
ld \$t4, A(\$t2)				IF	RAW	RAW	ID	EX	MEM	WB				
dadd \$t1, \$t1, \$t4					IF	----	----	RAW	RAW	ID	EX	MEM	WB	
daddi \$t2, \$t2, 8								IF	----	----	ID	EX	MEM	WB

- Iteraciones = $N = 3$ (longitud del vector A)
- Atascos RAW = $A_{RAW} = 2 + (2 + 2) * N = 2 + (2 + 2) * 3 = 2 + 4 * 3 = 2 + 12 = 14$
- Atascos WAR = $A_{WAR} = 0$
- Atascos WAW = $A_{WAW} = 0$
- Atascos STR (estructurales) = $A_{STR} = 0$
- Atascos Branch Taken (BTS) = $N - 1 = 3 - 1 = 2$
- Atascos Branch Misprediction (BMS) = 0
- Atascos = $A_{RAW} + A_{WAR} + A_{WAW} + A_{STR} + BTS + BMS = 14 + 0 + 0 + 0 + 2 + 0 = 16$
- Instrucciones ejecutadas antes del lazo = $I_1 = 3$
- Instrucciones ejecutadas en el lazo = $I_2 = 5 * N = 5 * 3 = 15$
- Instrucciones ejecutadas luego del lazo = $I_3 = 2$
- Instrucciones = $I_1 + I_2 + I_3 = 3 + 15 + 2 = 20$
- Ciclos = Instrucciones + Atascos + 4 = $20 + 16 + 4 = 40$
- CPI = Ciclos / Instrucciones = $40 / 20 = 1,9$

Ejercicio 3

a)

```
.data
CANT:    .word 8
DATOS:   .word 1, 2, 3, 4, 5, 6, 7, 8
RES:     .word 0

.code
dadd $t1, $0, $0
ld $t2, CANT($0)
loop:   ld $t3, DATOS($t1)
        daddi $t2, $t2, -1
        dsll $t3, $t3, 1
        sd $t3, RES($t1)
        daddi $t1, $t1, 8
        bnez $t2, loop
halt
```

A continuación, se representa el cauce segmentado del procesador a través del tiempo en caso de ejecutarse el presente programa desactivando el Forwarding, el Branch Target Buffer y el Delay Slot.

Ejecución inicial del programa									
Instrucción	Cauce del procesador								
dadd \$t1, \$0, \$0	IF	ID	EX	MEM	WB				
ld \$t2, CANT(\$0)		IF	ID	EX	MEM	WB			
ld \$t3, DATOS(\$t1)			IF	RAW	ID	EX	MEM	WB	
daddi \$t2, \$t2, -1				IF	-----	ID	EX	MEM	WB

Ejecución del programa dentro del lazo (no hay BTS en la última iteración)													
Instrucción	Cauce del procesador												
ld \$t3, DATOS(\$t1)	IF	ID	EX	MEM	WB								
daddi \$t2, \$t2, -1		IF	ID	EX	MEM	WB							
dsll \$t3, \$t3, 1			IF	RAW	ID	EX	MEM	WB					
sd \$t3, RES(\$t1)				IF	-----	RAW	RAW	ID	EX	MEM	WB		
daddi \$t1, \$t1, 8						IF	-----	-----	ID	EX	MEM	WB	
bnez \$t2, loop									IF	ID	EX	MEM	WB
halt										IF	BTS		

Ejecución del programa posterior al lazo						
Instrucción	Cauce del procesador					
halt	IF	ID	EX	MEM	WB	

- Iteraciones = $N = 8$ (valor de CANT)
- Atascos RAW = $A_{RAW} = 1 + (1 + 2) * N = 1 + (1 + 2) * 8 = 1 + 3 * 8 = 1 + 24 = 25$
- Atascos WAR = $A_{WAR} = 0$
- Atascos WAW = $A_{WAW} = 0$
- Atascos STR (estructurales) = $A_{STR} = 0$
- Atascos Branch Taken (BTS) = $N - 1 = 8 - 1 = 7$
- Atascos Branch Misprediction (BMS) = 0
- Atascos = $A_{RAW} + A_{WAR} + A_{WAW} + A_{STR} + BTS + BMS = 25 + 0 + 0 + 0 + 7 + 0 = 32$
- Instrucciones ejecutadas antes del lazo = $I_1 = 2$
- Instrucciones ejecutadas en el lazo = $I_2 = 6 * N = 6 * 8 = 48$
- Instrucciones ejecutadas luego del lazo = $I_3 = 1$
- Instrucciones = $I_1 + I_2 + I_3 = 2 + 48 + 1 = 51$
- Ciclos = Instrucciones + Atascos + 4 = $51 + 32 + 4 = 87$
- CPI = Ciclos / Instrucciones = $87 / 51 \approx 1,706$

b)

```
.data
CANT:    .word 8
DATOS:   .word 1, 2, 3, 4, 5, 6, 7, 8
RES:     .word 0
```

```

.code
dadd $t1, $0, $0
ld $t2, CANT($0)
loop: ld $t3, DATOS($t1)
      daddi $t2, $t2, -1
      dsll $t3, $t3, 1
      sd $t3, RES($t1)
      bnez $t2, loop      # Instrucciones reordenadas para que
      daddi $t1, $t1, 8   # el programa funcione correctamente
                          # con el Delay Slot activado

halt

```

A continuación, se representa el cauce segmentado del procesador a través del tiempo en caso de ejecutarse este nuevo programa activando el Delay Slot, pero manteniendo desactivados tanto el Forwarding como el Branch Target Buffer. Solo se exhibe la ejecución de la sección del programa cuyo análisis resulta especialmente relevante para el escenario planteado en este inciso.

Ejecución parcial del programa dentro del lazo (se excluye la primera instrucción de la iteración actual y se incluye la primera de la próxima repetición)															
Instrucción	Cauce del procesador														
daddi \$t2, \$t2, -1	IF	ID	EX	MEM	WB										
dsll \$t3, \$t3, 1		IF	RAW	ID	EX	MEM	WB								
sd \$t3, RES(\$t1)			IF	----	RAW	RAW	ID	EX	MEM	WB					
bnez \$t2, loop					IF	----	----	ID	EX	MEM	WB				
daddi \$t1, \$t1, 8								IF	ID	EX	MEM	WB			
ld \$t3, DATOS(\$t1)									IF	RAW	RAW	ID	EX	MEM	WB

- Iteraciones = $N = 8$ (valor de CANT)
- Atascos RAW = $1 + (1 + 2 + 2) * (N - 1) + (1 + 2) = 1 + (1 + 2 + 2) * (8 - 1) + (1 + 2)$
- Atascos RAW = $A_{RAW} = 1 + 5 * 7 + 3 = 1 + 35 + 3 = 39$
- Atascos WAR = $A_{WAR} = 0$
- Atascos WAW = $A_{WAW} = 0$
- Atascos STR (estructurales) = $A_{STR} = 0$
- Atascos Branch Taken (BTS) = 0
- Atascos Branch Misprediction (BMS) = 0
- Atascos = $A_{RAW} + A_{WAR} + A_{WAW} + A_{STR} + BTS + BMS = 39 + 0 + 0 + 0 + 0 + 0 = 39$
- Instrucciones ejecutadas antes del lazo = $I_1 = 2$
- Instrucciones ejecutadas en el lazo = $6 * (N - 1) + 5 = 6 * (8 - 1) + 5 = 6 * 7 + 5 = 42 + 5$
- Instrucciones ejecutadas en el lazo = $I_2 = 47$
- Instrucciones ejecutadas luego del lazo = $I_3 = 2$
- Instrucciones = $I_1 + I_2 + I_3 = 2 + 47 + 2 = 51$
- Ciclos = Instrucciones + Atascos + 4 = $51 + 39 + 4 = 94$
- CPI = Ciclos / Instrucciones = $94 / 51 \approx 1,843$