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

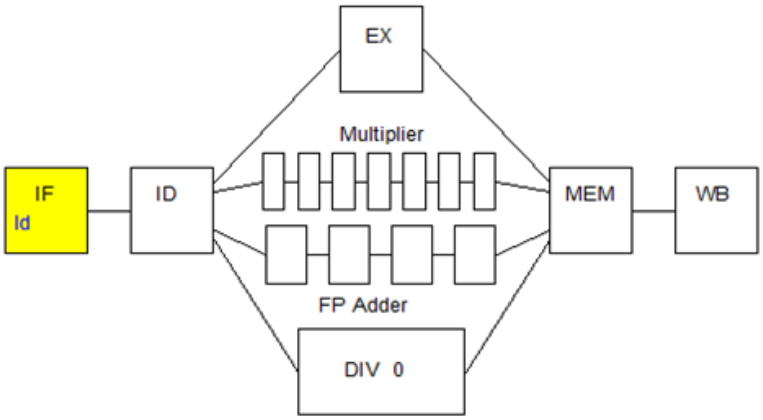
Los Ciclos Por Instrucción (CPI) son una medida de uso eficiente del procesador. Se definen como

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

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 B. Buscar la instrucción en memoria y llevarla a la CPU C. Calcular la dirección de un acceso a memoria D. Guardar el valor de un registro en memoria E. Traer de memoria un valor a un registro intermedio F. Almacenar el valor final de un registro G. Calcular la dirección de un salto H. Verificar si están disponibles los operandos necesarios para continuar con la ejecución de la instrucción	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`					
`dadd r2, r4, r3`					
`sd r3, tabla(r2)`					
`ld r3, tabla(r2)`					
`bneq r1, r2, loop`					
`j loop`					

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.

Responder:

1. La fórmula de CPI presentada anteriormente ¿se cumple para los dos programas?
2. En el segundo programa, se utilizan las instrucciones ddiv y dmul ¿cuántos se requieren para su ejecución?
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?
4. La instrucción HALT solo detiene la ejecución del programa ¿se cuenta en el cálculo 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?

`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`

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	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: dadd r3,r3,r2`	`dadd r2,r2,r3`
5	`and r4, r3, r1`	`j fin`	`daddi r1,r1,-1`	`daddi r1,r1,8`
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)`

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)

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).

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.

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 atascarán 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?
¿Cuál es el CPI?

b) Ejecutarlo en el simulador con la opción Configure/Enable Forwarding **habilitada**.
¿Por qué no se presenta ningún atasco en este caso? Explicar la mejora.
¿Cuál es el CPI?
¿Qué indica el color de los registros en la ventana Register durante la ejecución?

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?
- ¿Cuántos CPI tiene la ejecución del programa en este caso?
- Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de CPIs?

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.
- Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de CPIs?

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.
- Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de CPIs?

- En base a lo anterior ¿qué partes del programa conviene optimizar generalmente?

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?

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.

1. Cambiar el valor de B a 1000 y ejecutar. ¿Cómo cambió la cantidad de BTS? ¿y el CPI?
2. Completar la siguiente frase: Al ejecutar un loop simple con N iteraciones, se producen _____ atascos BTS.
3. ¿Por qué se producen los BTS? ¿Qué sucede con la instrucción siguiente al salto?
4. ¿Pueden evitarse los BTS reordenando instrucciones?
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?

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?
2. Notar que ahora aparece un nuevo tipo de atasco, llamado Branch Misprediction Stall (BMS). ¿Por qué sucede? ¿Cuántos ocurren?
3. Cambiar el valor de B a 1000, habilitar BTB y ejecutar. ¿Cómo cambió la cantidad de BTS y BMS? ¿y el CPI?
4. Completar la siguiente frase: Al ejecutar un loop simple con N iteraciones, si se habilita BTB, se producen _____ atascos de tipo BTS y _____ 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?

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.
b) Ejecutar el programa en el simulador con y sin BTB. ¿Qué programa es más eficiente?

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?
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					
BMS					
CPI					
#Instrucciones					

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?
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?

Parte 4: 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.
2. En el caso del WAR, ¿cuál es la instrucción lenta y cuál la rápida? ¿Qué registro se quiere leer y escribir?
3. Idem para el caso WAW.

```
.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?
b) Probar agregando un NOP ¿sigue el atasco? ¿y si se quita un NOP? ¿por qué?

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
```

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														
sd r3, MULT(r0)				IF	ID													
dadd r3,r1,r2					IF													
sd r3, SUMA(r0)							IF								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													
sd r3, prom(r0)				IF													
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)														ME	WB	
halt														EX	ME	WB

Parte 5: 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
```

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**.

A. Calcular manualmente el número de ciclos, CPI, RAWs y BTS/BMS, asumiendo forwarding activado y BTB desactivado.
B. Idem, asumiendo BTB activado.
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
          daddi r3, r3, 8
          bnez  r1, loop
          j     fin
 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**).

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

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