Ejercicio de seminario - el procesador segmentado, version 5

Estructura de computadores (Universitat Politecnica de Valencia)

ESTRUCTURA DE COMPUTADORES

El procesador segmentado

Curso 2012-2013
Versión 5

1. El tiempo de ejecución de los programas

PROBLEMA 1 El programa BROT tarda 34 segundos en ejecutarse en un computador que dispone de un procesador con un CPI medio de 2,5 y un reloj a 800 MHz.

1. Cuántas instrucciones máquina ejecuta este programa?
2. Si quisiéramos reducir el tiempo de ejecución por debajo de los 30 segundos, justifique cuál de las dos siguientes opciones será la más apropiada: aumentar la frecuencia de reloj en 50 MHz o cambiar el procesador por otro compatible (con el mismo juego de instrucciones) que trabaja a la misma frecuencia con un CPI medio de 2,1.

SOLUCIÓN

1. El número de instrucciones I se calcula por medio de la ecuación que expresa el tiempo de ejecución T de un programa como el producto del número de instrucciones N, el número medio de ciclos por instrucción CPI y el período del reloj t_c. Por lo tanto, resolvemos la ecuación:

$$34 = I \times 2{,}5 \times \frac{1}{800 \times 10^6}$$

   y obtenemos que el número de instrucciones I es de 10880×10^6.

2. Si gastamos la expresión anterior, la primera opción conseguirá un tiempo de ejecución de:

$$10880 \times 10^6 \times 2{,}5 \times \frac{1}{850 \times 10^6} = 32 \text{ s}$$

   La segunda opción, por su parte, conseguirá un tiempo de ejecución de:

$$10880 \times 10^6 \times 2{,}1 \times \frac{1}{800 \times 10^6} = 28{,}56 \text{ s}$$

   Por lo tanto, esta última opción rebaja el tiempo de ejecución por debajo de los 30 segundos.

■

2. Problemas sobre conflictos de datos

PROBLEMA 2 Al compilar el código de una función que calcula $\$v0 = 7 \times \$a0 + 1$ para el MIPS, un compilador genera esta serie de instrucciones:

```
1|        sll $v0,$a0,3
2|        sub $v0,$v0,$a0
3|        addi $v0,$v0,1
```

Suponga que el procesador incluye los cortocircuitos *MaEX*, *ERaEX* y *ERaM*. Considere que el valor inicial de $\$a0 = 4$ y que la lectura de la primera de las tres instrucciones se hace en el ciclo 1.

1. Identifique los conflictos de datos entre las instrucciones, expresándolo de la manera "Hay conflicto entre las instrucciones *x* y *y* por el uso del registro *r*".

2. ¿Qué cortocircuitos se aplican para resolver cada conflicto de los anteriores? ¿En qué ciclo? ¿A cuál de las dos entradas de la ALU (superior o inferior) se aplican en cada caso?

3. ¿En qué ciclos se actualiza el contenido del registro `$v0`?

4. ¿Que cambiaría en el apartado 2 si la función hubiera de calcular $\$v0 = -7 \times \$a0 + 1$, y el compilador generara las instrucciones siguientes?

```
1|        sll $v0,$a0,3
2|        sub $v0,$a0,$v0
3|        addi $v0,$t0,1
```

5. ¿Qué cambiaría en el apartado 2 si la función hubiera de calcular $\$v0 = 10 \times \$a0$, y el compilador generara las instrucciones siguientes?

```
1|        sll $t0,$a0,3
2|        sll $t1,$a0,1
3|        add $v0,$t0,$t1
```

SOLUCIÓN

1. Encontramos dos conflictos de datos:

Conflicto	entre las instrucciones	por el uso del registro
1	1 i 2	`$v0`
2	2 i 3	`$v0`

2. Los dos casos corresponden a dependencias de datos entre instrucciones aritméticas que van seguidas. El cortocircuito correspondiente es *MaEX*. En ambos casos, la instrucción consumidora del dato identifica el registro implicado como *rs*.

   El conflicto 1 se resuelve en el ciclo 4 (vea la Figura 1). En *DI/EX.S* se encuentra el valor antiguo del operando `$v0`, que se leyó del banco de registros en el ciclo anterior. El valor correcto está en *EX/M.ALUout* y el multiplexor superior de la ALU seleccionará el cortocircuito *MaEX*.

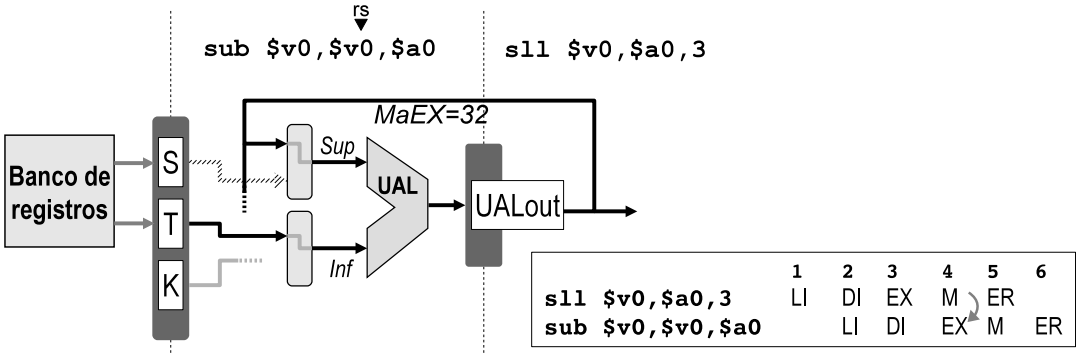

Figura 1: Apartado 2 — instántanea del ciclo 4

   El conflicto 2 se resuelve en el ciclo 4 (vea la Figura 2). De nuevo, el valor correcto está en *EX/M.ALUout* y el multiplexor superior de la ALU ha de seleccionar el cortocircuito *MaEX*

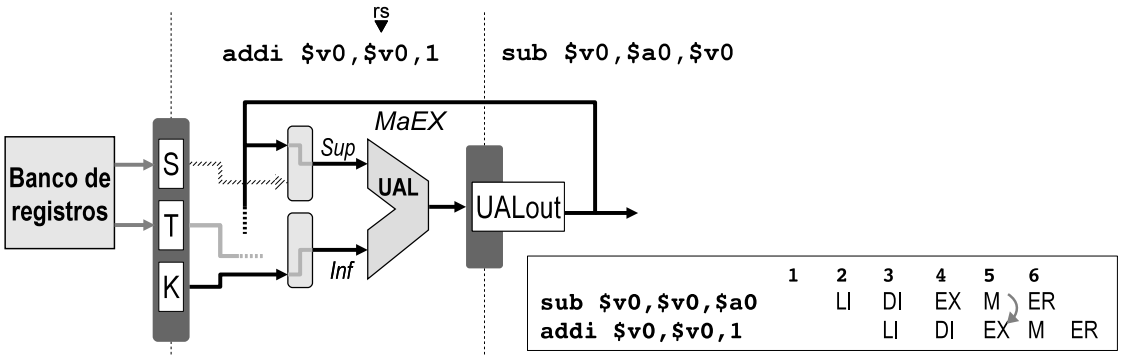

Figura 2: Apartado 2 — instantánea del ciclo 5

3. Las tres instrucciones escriben en `$v0`. Los valores son:

Ciclo	Valor
5	32
6	28
7	29

4. La diferencia está que en el conflicto 1, el registro `$v0` aparece en la posición *rt* de la instrucción consumidora. Por lo tanto, el multiplexor inferior de la ALU habrá de seleccionar el cortocircuito *MaEX* operar con el valor actualizado de `$v0`.

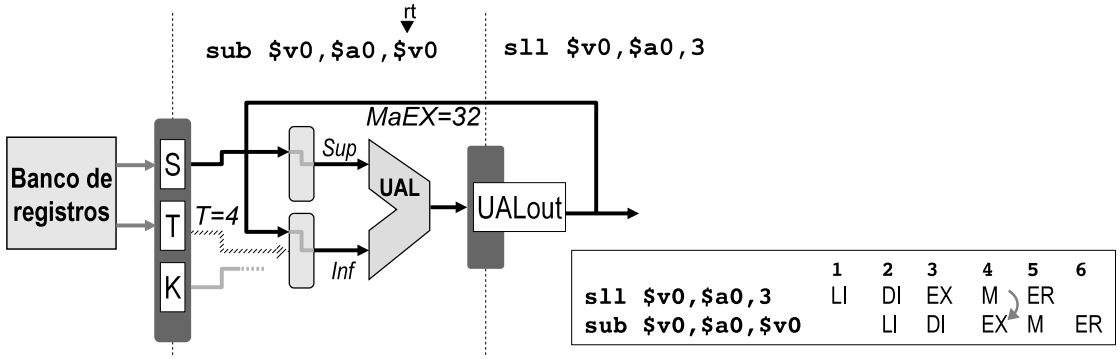

Figura 3: Apartado 4 — Instantánea del ciclo 4

5. Los dos conflictos de datos que podemos encontrar:

Conflicto	entre las instrucciones	por el uso del registro
1	1 i 3	`$t0`
2	2 i 3	`$t1`

se resuelven en el mismo ciclo (vea la Figura 4). El conflicto 1, que aparece entre dos instrucciones aritméticas dependientes separadas por otra instrucción, se puede resolver aquí con el curtcircuito *ERaEX*. El conflicto 2, por su parte, se resuelve con el cortocircuito *MaEX*.

■

PROBLEMA 3 El siguiente programa implementa la suma de los elementos de un vector $\vec{A}$ y deja el resultado en la dirección de memoria `suma`. La variable `N` indica el número de elementos que contiene el vector.

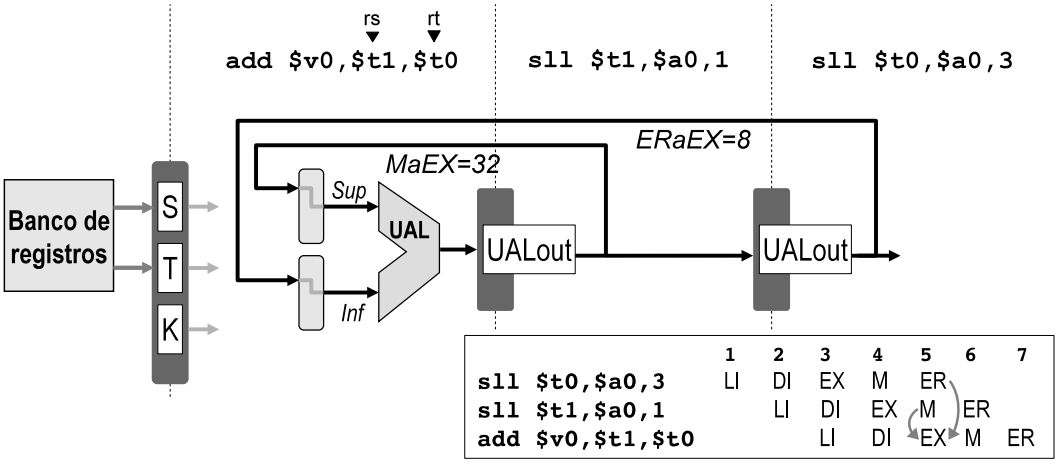

Figura 4: Apartado 5 — Instantánea del ciclo 5

```
            .data 0x10000000
      A:    .word 10, 20, 30, 40  # Vector A
      N:    .word 4               # Número de elementos del vector
   suma:    .word 0

            .text
1 | __start: lui $4, 0x1000      # $4 = dirección de A
2 |          addi $5, $4, 16     # $5 = dirección de N
3 |          lw $8, 0($5)        # $8 = N
4 |          ori $6, $0, 0       # $6 = suma = 0
5 |      do: lw $7, 0($4)        # $7 = A[i]
6 |          add $6, $6, $7      # suma = suma + A[i]
7 |          addi $4, $4, 4
8 |          addi $8, $8, -1     # N = N-1
9 |          bgtz $8, do
10 |         addi $5, $5, 4
11 |     fin: sw $6, 0 ($5)
```

El programa se ejecuta en un procesador segmentado semejante al MIPS R2000. Se pide:

1. Suponga que el procesador incluye los cortocircuitos *MaEX*, *ERaEX*, y *ERaM*. Cuando hay una instrucción de carga (instrucción `lw`) y la instrucción siguiente del programa consume el dato producido por `lw`, el hardware inserta un ciclo de parada. Por otra parte, los conflictos debidos a las instrucciones de salto se resuelven aplicando 3 ciclos de parada.

   Indique qué cortocircuitos se activarían durante la ejecución del programa y entre qué instrucciones. Si se aplica un cortocircuito *MaEX* o *ERaEX*, indique también qué entrada de la ALU lo seleccionaría ('S' para la superior e 'I' para la inferior). Rellene una tabla similar a la adjunta. Utilice el número situado a la izquierda de la instrucción para referirse a ella.

Entre instrucción	e instrucción	cortocircuito
1	2	*IDaER (S)*
...	...	...
...	...	...

2. Suponiendo las mismas hipótesis del apartado anterior, indique en qué ciclos se activarían los distintos cortocircuitos. Calcule el resultado solamente para los 20 primeros ciclos. Rellene una tabla similar a la adjunta. *Nota:* para rellenar la tabla conviene realizar el diagrama de instrucciones/ciclos.

Ciclo	Cortocircuito activado
4	*IDaER (S)*
...	...

3. Si el procesador soporta la carga retardada con un hueco de una instrucción, indique la o las instrucciones que se podrían insertar en el hueco de la instrucción de carga del bucle (`lw $7,0($4)`).

Este enunciado continúa en el problema 6

SOLUCIÓN

1. El cortocircuito *MaEX* es útil para resolver muchos conflictos de datos entre instrucciones dependientes que van seguidas, como es el caso de las dos primeras del programa:

```
lui $4, 0x1000
addi $5, $4, 16
```

   Como el registro que transmite el dato (**$4**) ocupa la posición de *rs* en el formato de instrucción, podemos deducir que el cortocircuito alimentará la entrada superior de la ALU.

   En el código aparecen otros conflictos por dependencia de datos entre instrucciones que van seguidas y que los resuelve el cortocircuito *MaEX* a través de la entrada superior de la ALU. La excepción es el conflicto de la instrucción de carga (5) y la que le sigue (6), cuya dependencia provoca el ciclo de parada que se expresa en este diagrama:

```
lw $7, 0($4)     LI  DI  EX  M   ER
add $6, $6, $7       LI  DI  EX  M   ER
```

   Por lo tanto, el cortocircuito aplicable es *ERaEX*. Además, el registro asociado al dato (**$7**) ocupa la posición *rt* en la instrucción consumidora (`add $6,$6,$7`) y se aplicará a la entrada inferior de la ALU.

   El cuadro final será este:

Entre instrucción	e instrucción	Cortocircuito
1	2	MaEX (S)
2	3	MaEX (S)
5	6	ERaEX (I)
8	9	MaEX (S)
10	11	MaEX (S)

2. Durante los 20 primeros ciclos se aplicarán los siguientes cortocircuitos:

Ciclo	Cortocircuito
4	MaEX
5	MaEX
9	ERaEX
12	MaEX
18	ERaEX

3. Se podría insertar la instrucción 7 o la 8, pero no la instrucción 6, ya que esta depende de la instrucción `lw`.

■

PROBLEMA 4 El siguiente programa suma a cada uno de los elementos de un vector $\vec{V}$ los de otro vector $\vec{W}$, tomando estos en sentido inverso. Los vectores son de cuatro elementos, numerados desde el 0 hasta el 3. Más formalmente, la operación que realiza es $v_i := v_i + w_{3-i}$ para valores de i desde 0 hasta 3. Los elementos de $\vec{V}$ y $\vec{W}$ son enteros de 4 bytes.

```
                .data 0              # Dirección origen de los datos = 0
           V:   .word 10,20,30,40    # Vector V
           W:   .word  5, 6, 7, 8    # Vector W
           N:   .word 4              # Número de iteraciones a realizar
           IV:  .word 0              # Desplazamiento del primer elemento
           IW:  .word 12             # Desplazamiento del último elemento

                .text
 1 |           lw    $8, N($0)      # $8 es el número de iteraciones
 2 |           lw   $10, IV($0)     # $10 = índice del primer elemento de V
 3 |           lw   $11, IW($0)     # $11 = índice del último elemento de W
 4 | bucle:    lw    $1, V($10)     # Cargar V(i)
 5 |           lw    $2, W($11)     # Cargar W(3-i)
 6 |           add   $3, $1, $2     # Sumarlos
 7 |           sw    $3, V($10)     # Almacenar resultado en V(i)
 8 |           addi $10, $10, 4     # Incrementar índice de V
 9 |           addi $11, $11,-4     # Decrementar índice de W
10 |           addi  $8, $8, -1     # Decrementar contador
11 |           bne   $0, $8, bucle  # Saltar si quedan elementos por sumar
```

Con los datos suministrados a este programa el resultado final es $\vec{V} = \{18, 27, 36, 45\}$. El programa se ejecuta sobre una versión del procesador MIPS segmentado que no trata ningún conflicto (no inserta ciclos de parada ni implementa cortocircuitos). La latencia de salto para las bifurcaciones es de tres ciclos. Se pide:

1. Identifique los conflictos de datos que existen en el programa. Utilice los números de línea para indicar los conflictos (por ejemplo, *conflicto entre la x y la y por uso del registro* $*reg*).

2. Inserte las instrucciones `nop` necesarias para resolver todos los conflictos de datos. Mantenga las instrucciones en el orden en que aparecen.

3. Calcule el número de ciclos que tardará en ejecutarse el programa, incluyendo las instrucciones `nop` del apartado anterior. Suponga que, para resolver los conflictos de control, el procesador inserta siempre tres ciclos de parada tras las instrucciones de salto condicional. No olvide que el bucle se repite 4 veces.

4. Suponga ahora que se incorporan al procesador todos los cortocircuitos necesarios para anticipación de datos, tanto de ALU como de memoria (MaEX, ERaEX y ERaM). De esta manera, el programa original sólo requeriría instrucciones `nop` para el caso de carga retardada, ya que no se insertan ciclos de parada para resolver este conflicto. Manteniendo la ordenación original de las instrucciones del programa, inserte las instrucciones `nop` que sean necesarias para la carga retardada e indique qué cortocircuito se aplica para resolver cada uno de los conflictos de datos identificados en el apartado 1.

5. Sobre el programa obtenido en el apartado anterior, y con las condiciones allí enunciadas, calcule el número de ciclos necesarios en este caso para la ejecución del programa. Siga suponiendo que el procesador inserta 3 ciclos de parada tras las instrucciones de bifurcación.

6. El número de ciclos se reduce cuando se aplican cortocircuitos de anticipación para resolver conflictos, en vez de insertar instrucciones `nop`. ¿Existe alguna desventaja en la eliminación de conflictos mediante cortocircuitos con respecto a la utilización de instrucciones `nop`?

Este enunciado continúa en el problema 7

SOLUCIÓN

1. Los conflictos de datos son los siguientes:
   - Entre la 4 y la 2 por uso de $10.
   - Entre la 5 y la 3 por uso de $11.
   - Entre la 6 y la 4 por uso de $1.
   - Entre la 6 y la 5 por uso de $2.
   - Entre la 7 y la 6 por uso de $3.
   - Entre la 11 y la 10 por uso de $8.
2. Para resolver los conflictos de datos, el código del programa quedaría modificado como sigue:

```
          .text
          lw    $8, N($0)       # $8 es el número de iteraciones
          lw   $10, IV($0)      # $10 = índice del primer elemento de V
          lw   $11, IW($0)      # $11 = índice del último elemento de W
          nop
bucle:    lw    $1, V($10)      # Cargar V(i)
          lw    $2, W($11)      # Cargar W(3-i)
          nop
          nop
          add   $3, $1, $2      # Sumarlos
          nop
          nop
          sw    $3, V($10)      # Almacenar resultado en V(i)
          addi $10, $10, 4      # Incrementar índice de V
          addi $11, $11,-4      # Decrementar índice de W
          addi  $8, $8, -1      # Decrementar contador
          nop
          nop
          bne   $0, $8, bucle   # Saltar si quedan elementos por sumar
```

3. Cálculo del número de ciclos. Hay 4 instrucciones antes del bucle y 14 en el bucle, contando las instrucciones `nop` insertadas en el apartado anterior, que se repetirán 4 veces. Por lo tanto,

$$I = 4 \text{ antes del bucle} + 4 \text{ iteraciones} \times 14 \text{ instrucciones/iteración} = 60 \text{ instrucciones}$$

   El procesador inserta 3 ciclos de parada tras la instrucción de bifurcación condicional

$$P = 4 \text{ iteraciones} \times 3 \text{ ciclos de parada/iteración} = 12 \text{ ciclos de parada}$$

   Para obtener el tiempo de ejecución, contabilizamos también los cuatro ciclos de llenado iniciales:

$$T = 4 + I + P = 76 \text{ ciclos}$$

4. Sólo hay un caso en que la instrucción `lw` vaya seguida inmediatamente por una instrucción que lea su registro destino: se trata de la instrucción 5, `lw $2, W($11)`, que va seguida por `add $3,$2,$1`, por lo que es necesario insertar una instrucción `nop` tras la instrucción 5 para carga retardada.

   En cuanto a los cortocircuitos aplicados, son los siguientes:

   - Entre la 4 y la 2 por uso de $10: se aplica ERaEX (a la entrada superior de la ALU).
   - Entre la 5 y la 3 por uso de $11: se aplica ERaEX (a la entrada superior de la ALU).
   - Entre la 6 y la 4 por uso de $1: la `nop` de carga retardada elimina este conflicto.
   - Entre la 6 y la 5 por uso de $2: dado que se inserta `nop` para carga retardada tras la instrucción 5, el cortocircuito aplicado es ERaEX (a la entrada inferior de la ALU).
   - Entre la 7 y la 6 por uso de $3: se aplica ERaM.
   - Entre la 11 y la 10 por uso de $8: se aplica MaEX (a la entrada inferior de la ALU).

5. Con la modificación del apartado anterior, el programa queda como el original, añadiendo únicamente una instrucción `nop` tras la instrucción 5.

   El número de instrucciones ejecutadas será pues:

   $$I = 3 \text{ (antes del bucle)} + 4 \text{ (iteraciones)} \times 9 \text{ (instrucciones/iteración)} = 39 \text{ instrucciones}$$

   Se siguen insertando los 3 ciclos de parada tras la `bne`, que se ejecuta 4 veces, dando un total de $P = 3 \times 4 = 12$ ciclos de parada. Así pues, el número total de ciclos será de $T = 39 + 12 + 4 = 55$.

6. La lógica de detección de conflictos y la circuitería que aplica el cortocircuito podrían alargar la duración necesaria del ciclo de reloj, lo que afecta directamente al tiempo de ejecución de todas las instrucciones.

■

---

3. Problemas sobre conflictos de control

PROBLEMA 5 Escriba en ensamblador del MIPS el código correspondiente a las expresiones de alto nivel de más abajo. En todos los casos, considere que el procesador tiene las características relevantes habituales en el R2000, que son:

- Todos los cortocircuitos que tengan sentido
- Carga retardada con hueco de una instrucción
- Salto retardado con hueco de una instrucción

Siga el convenio de programación habitual: las funciones reciben los argumentos en `$a0`, `$a1`, etc.; han de devolver el resultado en `$v0` y pueden gastar los registros temporales `$t0`, `$t1`, etc. No olvide que la instrucción que cierra el código de una función es `jr $ra`, y que esta instrucción también tiene hueco de salto. Igualmente, note que `jal` es también una instrucción con hueco de salto y que, en estas condiciones, la dirección de retorno es la de la instrucción siguiente al hueco. En todos los casos, intente optimizar el código llenando los huecos de salto.

1. Escriba en ensamblador del MIPS el código de las funciones que se indiquen más abajo.

   a) Multiplicación por 10: $\text{x10}(x) = 10 \times x$

   b) Valor absoluto: $\text{abs}(x) = |x|$

c) Número de bits a uno:

$$\text{ones}(x) = \sum_{i=0}^{31} b_i$$

2. Escriba el código que utiliza dos de las funciones anteriores para calcular la expresión $b = |a| \times 10$. Considere que una pseudoinstrucción como `lw $t0,a` se traduce en solo una instrucción máquina.

Solución

1. Código de las funciones:

   a) El código de la función `x10` para un procesador sin salto retardado es este:

   ```
   x10:    sll $v0,$a0,3    # 8x
           sll $t0,$a0,1    # 2x
           add $v0,$a0,$t0  # 8x + 2x = 10x
           jr $ra
   ```

   Si no aplicamos ninguna estrategia de optimización y nos limitamos a insertar instrucciones `nop` en el hueco de salto, el código apropiado es este:

   ```
   x10:    sll $v0,$a0,3
           sll $t0,$a0,1
           add $v0,$a0,$t0
           jr $ra
           nop
   ```

   Para optimizar, podemos insertar una instrucción previa independiente. Con instrucciones de salto incondicional nunca hay dependencias de datos y es fácil encontrar una buena candidata.

   ```
   x10:    sll $v0,$a0,3
           sll $t0,$a0,1
           jr $ra
           add $v0,$a0,$t0
   ```

   b) Un código no optimizado para el procesador con salto retardado sería:

   ```
   abs:
   if:     bgez $a0,else   # ¿Bit de signo de $a0 == 0?
           nop
   then:   sub $v0,$0,$a0  # Si bit de signo == 1, devuelvo -$a0
           jr $ra
           nop
   else:   add $v0,$0,$a0  # Si bit de signo == 0, devuelvo $a0
           jr $ra
           nop
   ```

   Insertando instrucciones previas, se puede optimizar el código:

   ```
   abs:
   if:     bgez $a0,else
           nop
   then:   jr $ra
           sub $v0,$0,$a0
   else:   jr $ra
           add $v0,$0,$a0
   ```

   c) Este es un posible código de *ones*, con las instrucciones `nop` colocadas señalando los huecos de salto

   ```
   # Cuenta bits a 1 de izquierda a derecha
   ones:   li $v0,0        # contador de unos = 0
           li $t0,32       # contador de iteraciones restantes
   ```

```
bucle:  bgez $a0,seguir # si bit de signo == 0, salto
        nop             #   (hueco
        addi $v0,$v0,1  # si bit de signo == 1, contador de unos++
seguir: sll $a0,$a0,1   # siguiente bit
        addi $t0,$t0,-1 # contador de bucle
        bgtz $t0,bucle
        nop
        jr $ra
        nop
```

Una optimización posible:

```
ones:   li $v0,0
        li $t0,32
bucle:  bgez $a0,seguir
        addi $t0,$t0,-1 # instrucción destino
        addi $v0,$v0,1
seguir: bgtz $t0,bucle
        sll $a0,$a0,1   # instrucción previa
        jr $ra
        nop
```

2. El código de $b = |a| \times 10$ para un procesador sin salto retardado es:

```
lw $a0,a
jal abs
move $a0,$v0
jal x10
sw $v0,b
```

Para optimizar el código para el caso de procesador con salto retardado, podemos insertar instrucciones previas:

```
jal abs
lw $a0,a
jal x10
move $a0,$v0
sw $v0,b
```

■

PROBLEMA 6 Considera el enunciado del problema 3.

1. Si el procesador soporta el salto retardado con un hueco de una instrucción, indique cómo se podría rellenar el hueco de salto utilizando instrucciones previas, objetivo o siguientes.

2. Si se aplican todos los cortocircuitos estudiados para resolver los conflictos de datos, la carga la resuelve el compilador reordenando el código e insertando la instrucción correspondiente en el hueco, y los conflictos de salto se resuelven aplicando la técnica *predict taken*, ¿cuántos ciclos tardaría en ejecutarse el programa? Suponga que la penalización debida a una predicción errónea es de 3 ciclos.

SOLUCIÓN

1. Las distintas alternativas son instrucciones previas, destino y siguientes.

   instrucciones previas: hay 2 alternativas, insertar en el hueco la instrucción 6 o bien la instrucción 7.

instrucciones objetivo: podemos poner la `lw` destino en el hueco. En este caso, se habría de añadir una nueva etiqueta (por ejemplo do+4) en la instrucción siguiente a la load. También habríamos de cambiar la etiqueta en la instrucción que la referencia (instrucción de salto). Es decir, la instrucción de salto quedaría `bgtz $8,do+4`.

instrucciones siguientes: llenar el salto con instrucciones siguientes sin reordenar el código no es posible, ya que si ponemos la instrucción `addi` cambiamos la semántica del programa. La solución pasa por reordenar, colocando la instrucción `addi` antes del bucle e insertando la instrucción `sw` en el hueco.

2. El tiempo que emplea el programa en ejecutarse son 33 ciclos.

   El desglose de los ciclos es el siguiente: 4 para llenar (o vaciar) la unidad segmentada, 4 para a las instrucciones que hay antes del bucle, 20 ciclos (5 instrucciones para 4 iteraciones) para el bucle, 3 de penalización ya que la última predicción falla y 2 ciclos para las dos últimas instrucciones.

■

PROBLEMA 7 Considere el enunciado del problema 4, en el que el procesador no trata los conflictos de datos; es decir, que la ruta de datos no implementa cortocircuitos y los conflictos de datos han de resolverse por software, bien con instrucciones `nop` o, mejor aún, con instrucciones útiles reordenadas. La latencia de salto es de tres ciclos.

1. Suponga que el procesador trata los conflictos de datos mediante salto retardado. Reordene las instrucciones del programa de manera que se eliminen al máximo las dependencias de datos (incluyendo las de carga) y los riesgos de control. Calcule el tiempo de ejecución en ciclos y el CPI del programa reordenado.

2. Suponga ahora que el procesador resuelve los conflictos de datos mediante cualquier cortocircuito que sea posible e inserta ciclos de parada si es necesario. Los conflictos de control los resuelve mediante *predict-not taken*. Calcule bajo este supuesto el número de ciclos que costará ejecutar el programa del enunciado del problema 4 así como el CPI resultante.

SOLUCIÓN

1. En los siguientes listados se mantienen los números de línea del original, para localizar los cambios más fácilmente.

   Si nos concentramos en los conflictos de datos y rellenamos (de momento) el hueco de salto con tres instrucciones `nop`, una solución podría ser esta:

```
 2 |         lw   $10, IV($0)
 3 |         lw   $11, IW($0)
 1 |         lw    $8, N($0)      # Elimina los dos primeros conflictos
 4 | bucle:  lw    $1, V($10)
 5 |         lw    $2, W($11)
10 |         addi  $8, $8, -1     # Resuelvo conflictos con add $3,$1,$2
 9 |         addi $11, $11,-4     #   id.
 6 |         add   $3, $1, $2
             nop                  # Resuelvo conflictos con sw $3,V($10)
             nop                  #   id.
 7 |         sw    $3, V($10)
 8 |         addi $10, $10, 4
11 |         bne   $0, $8, bucle
             nop
             nop
             nop
```

donde se han reordeando las instrucciones para alejar las que tienen relación de dependencia de datos entre sí, dejando siempre dos instrucciones independientes entre ellas.

Una mejora interesante puede ser esta:

```
 2 |         lw   $10, IV($0)
 3 |         lw   $11, IW($0)
 1 |         lw    $8, N($0)
 4 | bucle:  lw    $1, V($10)
 5 |         lw    $2, W($11)
10 |         addi  $8, $8, -1
 9 |         addi $11, $11,-4
 6 |         add   $3, $1, $2
 8 |         addi $10, $10, 4   # Escribe $10
             nop
 7 |         sw    $3, V($10)   # Lee valor $10 antiguo
11 |         bne   $0, $8, bucle
             nop
             nop
             nop
```

Notad que la instrucción `sw $3,V($10)` lee el registro `$10` no actualizado, porque la instrucción `addi $10,$10,4` no ha podido escribir el resultado todavía.

Finalmente, si reordenamos código para rellenar el hueco de salto, puede quedar así:

```
 2 |         lw   $10, IV($0)
 3 |         lw   $11, IW($0)
 1 |         lw    $8, N($0)      # Elimina los dos primeros conflictos
 4 | bucle:  lw    $1, V($10)
 5 |         lw    $2, W($11)
10 | bucle2: addi  $8, $8, -1     # Resuelvo conflictos con add $3,$1,$2
 9 |         addi $11, $11,-4     #   id.
 6 |         add   $3, $1, $2
 8 |         addi $10, $10, 4     # Resuelvo conflictos con sw $3,V($10)
11 |         bne   $0, $8, bucle2
 7 |         sw    $3, V($10)     # Instrucción previa
 4'|         lw    $1, V($10)     # Instrucción objetivo (duplicada)
 5'|         lw    $2, W($11)     #   id.
```

donde aparece una instrucción previa y dos objetivo duplicadas a continuación del salto `bne` final del bucle. Naturalmente, la dirección de salto es ahora `bucle2` (o sea, `bucle` $+ 8$), para omitir las dos instrucciones duplicadas. Note ahora que la instrucción `lw $1,V($10)` lee el valor actualizado de `$10`.

El programa queda libre de todo conflicto (tanto de datos como de control) y su CPI es igual a 1. El número de instrucciones ejecutadas es:

$$I = 5 \textit{ (antes de } \texttt{bucle2}) + 4 \textit{ (iteraciones)} \times 8 \textit{ (instrucciones/iteración)} = 37 \text{ instrucciones}$$

Como no hay ciclos de parada, el tiempo de ejecución será:

$$t = I + 4 = 41 \text{ ciclos}$$

2. Con el salto productivo se ejecutan las mismas instrucciones que ejecutaría una versión no segmentada del procesador, dado que el procesador cancela todas las instrucciones que se encuentran fuera del flujo de ejecución tan pronto como procesa las bifurcaciones. En este caso, el procesador leerá las tres instrucciones (desconocidas) que siguen a (11), pero las cancelará durante las 3 primeras iteraciones. Después de la última iteración, esas tres instrucciones sí que se ejecutarán, pero están fuera del enunciado.

Por lo tanto, el número de instrucciones que se ejecutan es:

$$I = 3 \text{ anteriores a } \texttt{bucle} + 4 \text{ iteraciones} \times 8 \text{ instrucciones/iteración} = 35 \text{ instrucciones}$$

Calculemos ahora el número de ciclos de parada.

- Las dependencias de datos fuerzan un ciclo de parada por iteración entre las instrucciones (5) y (6); el resto de conflictos de datos se resuelven mediante cortocircuitos.
- Las cuatro iteraciones se consiguen mediante la bifurcación (11), que es efectiva las tres primeras veces. En cada una de ellas, la técnica *predict-not taken* ocasiona tres ciclos de parada, correspondientes a las tres instrucciones canceladas cada vez que el salto es efectivo.

Por tanto,

$$\begin{aligned} P &= 1 \text{ parada por conflicto de datos} \times 4 \text{ iteraciones} \\ &+ 3 \text{ paradas por conflicto de control} \times 3 \text{ iteraciones} \\ &= 13 \text{ ciclos} \end{aligned}$$

Ahora podemos calcular el tiempo de ejecución

$$t = 4 + I + P = 52 \text{ ciclos}$$

y el CPI

$$CPI = 1 + \frac{P}{I} = 1{,}37 \text{ ciclos por instrucción}$$

■

---

4. Problemas completos

PROBLEMA 8 Considera un vector P formado por $N = 10^6$ punteros que apuntan a N palabras distribuidas por la memoria principal. Hay que hacer un programa que, a partir del vector P, escriba en la memoria un vector S de N elementos formado por todas las palabras a las que apuntaba P, como muestra la figura 5

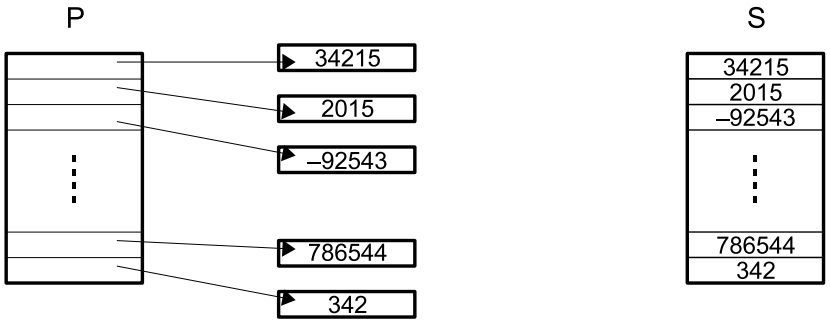

Figura 5: El vector P de punteros y el vector S resultante.

Escrito en un lenguaje de alto nivel, el programa debería declarar un vector P con elementos de tipo **long* y un vector S con elementos del tipo *long*. Así, un esquema del programa sería el siguiente:

```
long *P[N];
long S[N];
...
```

```
/* $r es un registro del procesador */
for(long $r = N; $r >= 0; $r --){
       S[i] = * P[i];
}
```

En código ensamblador del MIPS, el tipo *long* y cualquier puntero (y en particular **long*), se representan como variables de tipo *word*. Una versión no optimizada del programa anterior es esta:

```
        la $t0,P
        la $t1,S
        li $t2,N
bucle:  lw $t3,0($t0)
        lw $t4,0($t3)
        sw $t4,0($t1)
        addi $t0,$t0,4
        addi $t1,$t1,4
        addi $t2,$t2,-1
        bqtz $t2,bucle
        nop
        nop
        nop
```

Supón que este programa se ejecuta en una versión del MIPS con las características siguientes:

- El reloj oscila a 1 GHz.
- La ruta de datos incluye todos los curtcircuitos que tengan sentido.
- El control inserta un ciclo de parada en caso de una instrucción de carga de memoria seguida de una instrucción dependiente.
- Las instrucciones de salto modifican el CP en la etapa *EX*.
- Los conflictos de control se resuelven con salto retardado.

Se pide:

1. Analiza la ejecución de este programa sobre el camino de datos especificado. En particular:
   a) Calcula el número total de instrucciones que se ejecutan, considerando que las pseudoinstrucciones iniciales del programa se traducen en una sola instrucción de máquina. Razona brevemente la respuesta.
   b) Escribe el diagrama instrucciones/tiempo de una iteración, numerando los ciclos a partir de la fase de lectura de la instrucción `lw $t3,0($t0)`.
   c) Indica los cortocircuitos que se seleccionan (p.ej "ciclo 23: cortocircuito *ERaDI*").
   d) Calcula el CPI medio durante la ejecución del programa. Razona brevemente la respuesta.
   e) Calcula el tiempo de ejecución total del programa en milisegundos.
2. Reordena las instrucciones del programa para reducir el efecto de los conflictos de datos y de control que aparecen en la ruta de datos.
3. En qué cambiarían los apartados 1 y 2 si el procesador resolviera los conflictos de control mediante salto *predict-not taken*

Solución

1. Análisis del programa original

   a) Como el hueco de salto será de dos instrucciones, en cada iteración se ejecutarán 9 instrucciones. Por lo tanto, $I = 9 \cdot 10^6 + 3$.

   b) Las figuras 6 y 7 muestran dos diagramas posibles. La diferencia está en la etapa que genera el ciclo de parada. En la figura 6, la etapa *EX* produce la burbuja en el ciclo 4. En la figura 7, es la etapa *DI* quien la produce en el ciclo 3. Ambas soluciones son equivalentes: en los ciclos posteriores, se aprecia como la burbuja progresa y se manifiesta en el ciclo 6, cuando no acaba ninguna instrucción

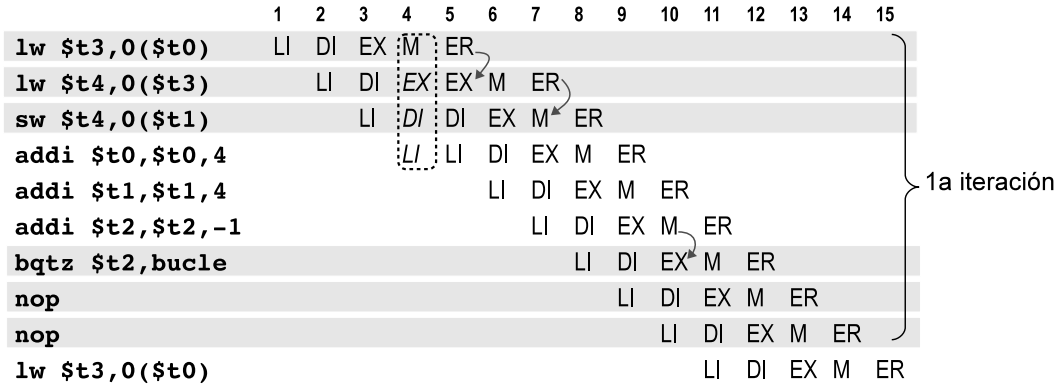

Figura 6: Cronograma de una iteración. La burbuja se crea en la etapa *EX*.

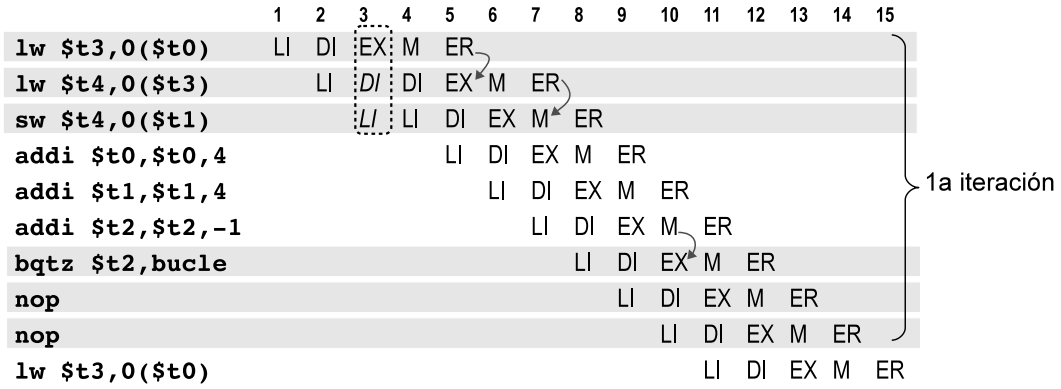

Figura 7: Cronograma alternativo a la Figura 6. La burbuja se crea en la etapa *DI*.

   c) Tabla de cortocircuitos:

ciclo	cortocircuito
5	ERaEX
7	ERaM
10	MaEX

   d) Cálculo del CPI: La mayor parte del tiempo se están ejecutando las instrucciones del bucle, y podemos despreciar las instrucciones iniciales. A cada iteración se produce un ciclo de parada. Por lo tanto:

$$CPI = 1 + \frac{1}{9} = 1{,}11$$

e) Cálculo del tiempo de ejecución:

$$\text{Tiempo de ejecución} = I \times CPI \times t_c = 9 \cdot 10^6 \times 1{,}11 \times 10^{-9} = 10 \text{ milisegundos}$$

2. Un posible código optimizado:

```
          la $t0,P
          la $t1,S
          li $t2,N
bucle:    lw $t3,0($t0)
          addi $t2,$t2,-1 # separa dependencias
          lw $t4,0($t3)
          sw $t4,0($t1)
          bqtz $t2,bucle
          addi $t0,$t0,4  # instrucción previa
          addi $t1,$t1,4  # instrucción previa
```

y otro:

```
          la $t0,P
          la $t1,S
          li $t2,N
bucle:    lw $t3,0($t0)   # esta instrucción se ha duplicado
bucle+4:  addi $t0,$t0,4
          lw $t4,0($t3)
          sw $t4,0($t1)
          addi $t2,$t2,-1
          bqtz $t2,bucle+4
          addi $t1,$t1,4  # instrucción previa
          lw $t3,0($t0)   # instrucción siguiente
```

3. Análisis del programa original con lógica *predict-not taken*

   - En todas las iteraciones (excepto la última) se ejecutarán 7 instrucciones. Por lo tanto, $I = 7 \cdot 10^6 + 6$ instrucciones. Las tres instrucciones `nop` posteriores sólo se ejecutarían una vez.
   - El diagrama de la primera iteración con *predict-not taken* aparece en la figura 8 (está basado en la figura 7, pero la burbuja asociada a la instrucción `lw $t4,0($t3)` también la podría haber generado la etapa *EX*). Excepto en la última iteración, en todas les demás el salto es efectivo y cancela las dos instrucciones *nop* siguientes, lo que se traduce en dos ciclos de parada.
   - La distribución de los cortocircuitos es la misma que en el apartado 1.
   - En cada iteración, excepto en la última, se producen tres ciclos de parada. Por lo tanto, tenemos que:
   $$CPI = 1 + \frac{3}{7} = 1{,}43$$
   - El tiempo de ejecución no cambia, porque el tiempo de ejecución de una iteración (vea la figura 8) es el mismo. Usando la fórmula,
   $$\text{Tiempo de ejecución} = I \times CPI \times t_c = 7 \cdot 10^6 \times 1{,}43 \times 10^{-9} = 10 \text{ milisegundos}$$
   - En cuanto a la optimización, sólo podemos eliminar los ciclos de parada producidos por los conflictos de datos. Por lo tanto, dos posibles optimizaciones son

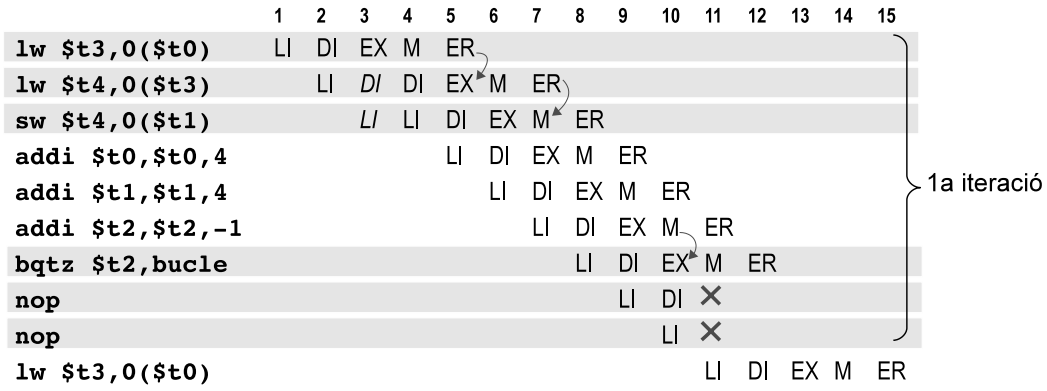

Figura 8: Cronograma de una iteración con *predict-not taken*.

```
                la $t0,P                        la $t0,P
                la $t1,S                        la $t1,S
                li $t2,N                        li $t2,N
        bucle:  lw $t3,0($t0)           bucle:  lw $t3,0($t0)
                addi $t2,$t2,-1                 addi $t0,$t0,4
                lw $t4,0($t3)                   lw $t4,0($t3)
                sw $t4,0($t1)                   sw $t4,0($t1)
                addi $t0,$t0,4                  addi $t2,$t2,-1
                addi $t1,$t1,4                  addi $t1,$t1,4
                bgtz $t2,bucle                  bgtz $t2,bucle
```

■

PROBLEMA 9 Dado un vector $\vec{V}$ de N elementos, el programa de más abajo obtiene el elemento máximo de dicho vector y su posición. Si el máximo se repite varias veces dentro de $\vec{V}$, el programa obtiene la posición del primero de ellos.

```
            .data 0x0
       V:   .word 1,2,3,4,5,6,7,8,9,10  # Vector V
       N:   .word 10          # Número de elementos de V
  maximo:   .word 0           # Elemento máximo
 pos_max:   .word 0           # Posición del máximo

            .text
 1|         add $t0,$0,$0           # $t0: i, índice para recorrer V (inicial a 0)
 2|         lw  $t1, N($0)          # $t1: núm de elementos de V (hace de contador)
 3|         lw  $t2, maximo($0)     # $t2: máximo
 4|         lw  $t5, pos_max($0)    # $t5: posición del máximo
 5| bucle:  lw  $t3,V($t0)          # Leer elemento v(i)
 6|         sub $t4,$t2,$t3        # $t4 = máximo - v(i)
 7|         bgez $t4, seguir       # Si maximo >= v(i) seguir
 8|         add $t2,$t3,$0         # $t2 = nuevo máximo = v(i)
 9|         srl $t5,$t0,2          # $t5 = pos_max = i
10| seguir: addi $t0,$t0,4         # $t0 = i++
11|         addi $t1,$t1,-1        # Decrementar contador
12|         bne $t1,$0,bucle
13|         sw $t2, maximo($0)     # Resultados
14|         sw $t5, pos_max($0)
```

El programa se ejecuta en el procesador MIPS segmentado en cinco etapas, pero con una ruta de datos con latencia de salto de 2 ciclos. Considere que no hay pseudo-instrucciones en el

programa. Observe también que, con el vector dado $\vec{V} = (1, 2, 3, 4, 5, 6, 7, 8, 10)$, la instrucción de comparación (`bgez`) siempre falla.

1. Indique qué conflictos de datos presenta el programa. Utilice los números asignados a cada instrucción para especificarlos como *conflicto entre la instrucción X y la Y por uso del registro Z*.

2. Calcule el número de ciclos que tardará el programa en ejecutarse, y el CPI con los datos que se incluyen en el mismo. Suponga que el procesador inserta los ciclos de parada apropiados para resolver los conflictos de datos y dos ciclos de parada tras cada instrucción de salto condicional.

3. Suponga ahora que el procesador incluye los cortocircuitos *MaEX* y *ERaEX*, pero no *ERaM*. En caso de que algún conflicto no quede resuelto por la anticipación, añada las instrucciones `nop` necesarias, pero no reordene el código. Los conflictos de salto se resuelven con ciclos de parada. Calcule el número de ciclos que tardará en ejecutarse el programa indicando, para cada conflicto, cuál es el cortocircuito que se aplica para resolverlo. Calcule el CPI en este caso.

4. Igual que en el apartado 3, pero suponga que se utiliza un técnica de predicción fija de salto no efectivo (*predict not-taken*). Calcule el CPI en este caso.

5. Manteniendo los cortocircuitos indicados en el apartado 3 para los conflictos de datos, suponga ahora que se emplea una técnica de salto retardado para resolver los conflictos de control. Para ello, reordene el código de forma que se aprovechen las instrucciones objetivo. Cuando esto no sea posible, inserte instrucciones `nop`.

SOLUCIÓN

1. Los conflictos de datos presentes en el código son los siguientes:

   - Conflicto 1 entre las instrucciones 5 y 6 por el uso del registro `$t3`.
   - Conflicto 2 entre las instrucciones 6 y la 7 por el uso de `$t4`.
   - Conflicto 3 entre las instrucciones 11 y la 12 por el uso de `$t1`.

2. Analicemos primero el comportamiento del programa y las incidencias durante su ejecución en esta ruta de datos.

   - El bucle incluye 8 instrucciones (de la 5 a la 12) que se ejecutan en todas las iteraciones, ya que el salto de la instrucción `bgez` (7) nunca es efectivo.
   - Fuera del bucle, hay seis instrucciones (de la 1 a la 4 más la 13 i la 14) que no provocan ningún conflicto.
   - El procesador trata cada uno de los tres conflictos de datos localizados en el apartado 1 con dos ciclos de parada, así que habrá que contar seis ciclos de parada en total.
   - Además, cada instrucción de salto provoca dos ciclos de parada para resolver el conflicto de control, y por lo tanto habrán cuatro por iteración.

   Calculemos el número de instrucciones que se ejecutan:

$$\begin{aligned} I &= 4 \text{ instrucciones antes del bucle} \\ &+ 10 \text{ iteraciones} \times 8 \text{ instrucciones por iteración} \\ &+ 2 \text{ instrucciones posteriores al bucle} \\ &= 86 \text{ instrucciones totales} \end{aligned}$$

Y el de ciclos de parada:

$$\begin{aligned} P &= 10 \text{ iteraciones} \times (6 \text{ ciclos de parada por conflictos de datos} + \\ &\quad + 4 \text{ ciclos de parada por conflictos de control}) \\ &= 100 \text{ ciclos de parada totales} \end{aligned}$$

El tiempo total de ejecución en ciclos será:

$$T \text{ (ciclos)} = 4 \text{ (llenado de las etapas)} + 86 \text{ (instrucciones)} + 100 \text{ (paradas)} = 190$$

Y el número medio de ciclos por instrucción:

$$CPI = 1 + \frac{100 \text{ ciclos de parada}}{86 \text{ instrucciones}} = 2{,}16 \text{ ciclos/instrucción}$$

3. Con este diseño, todos los conflictos de datos pueden resolverse por anticipación. Aunque el cortocircuito *ERaM* no se ha implementado. y podría necesitarlo cualquier instrucción `sw`). Sin embargo, no es así en nuestro caso ya que estas instrucciones (la 13 y 14) no presentan conflictos por dependencia de datos.

   - El conflicto 1 entre la 5 y la 6 se resuelve mediante una instrucción `nop` y el cortocircuito ERaEX.
   - Conflicto 2 entre la 6 y la 7 se resuelve mediante el cortocircuito MaEX.
   - Conflicto 3 entre la 11 y la 12 se resuelve mediante el cortocircuito MaEX.

   Así pues, el número de instrucciones se calcula como sigue:

$$\begin{aligned} I &= 4 \text{ instrucciones antes del bucle} \\ &+ 10 \text{ iteraciones} \times 9 \text{ instrucciones por iteración} \\ &+ 2 \text{ instrucciones posteriores al bucle} \\ &= 96 \text{ instrucciones totales} \end{aligned}$$

   Y el cálculo del número de ciclos de parada:

$$P = 10 \text{ iteraciones} \times 4 \text{ ciclos de parada por conflictos de control} = 40 \text{ ciclos de parada}$$

   El tiempo total de ejecución en ciclos será:

$$T \text{ (ciclos)} = 4 \text{ (llenado)} + 96 \text{ (instrucciones)} + 40 \text{ (paradas)} = 140$$

   Y el número medio de ciclos por instrucción:

$$CPI = 1 + \frac{40 \text{ ciclos de parada}}{96 \text{ instrucciones}} = 1{,}42 \text{ ciclos/instrucción}$$

4. Análisis de la ruta de datos con predicción de salto no efectivo (*predict-not taken*): Dado que hay que mantener la instrucción `nop` introducida en el apartado 3, el cálculo del número de instrucciones que se hizo allí todavía es válido. Por lo tanto,

$$I = 96 \text{ instrucciones}$$

   Para calcular el número de ciclos de parada, analizamos el comportamiento de las instrucciones de salto. Con *predict-not taken*, si la latencia de salto de la ruta de datos es de dos, para cada salto efectivo el procesador cancelará dos instrucciones. Si tenemos en cuenta los datos de $\vec{V}$,

veremos que la instrucción (7) `bgez $t4, seguir` nunca salta. La instrucción de salto (12) `bne $t1,$0,bucle` es efectiva 9 veces; es decir, todas las iteraciones excepto la última.

$$P = 9 \text{ cancelaciones} \times 2 \text{ ciclos de parada por cancelación} = 18 \text{ ciclos}$$

Por lo tanto,

$$T = 4 + 96 + 18 = 118 \text{ ciclos;} \qquad CPI = 1 + \frac{18}{96} = 1,19 \text{ ciclos/instrucción}$$

5. La reordenación de código, para insertar las instrucciones objetivo en el hueco del salto sería:

```
            .data
         V: .word 1,2,3,4,5,6,7,8,9,10  # Vector V
         N: .word 10        # Número de elementos de V
    maximo: .word 0         # Elemento máximo
  pos_max:  .word 0         # Posición del máximo

            .text
 1|         add $t0,$0,$0       # $t0: i, índice para recorrer V (inicial a 0)
 2|         lw  $t1, N          # $t1: Núm. de elementos de V (hace de contador)
 3|         lw  $t2, maximo     # $t2: Máximo
 4|         lw  $t5, pos_max    # $t5: posición del máximo
 5| bucle:  lw  $t3,V($t0)      # Leer elemento v(i)
 -|         nop
 6| bucle+8:sub $t4,$t2,$t3     # $t4 = máximo - v(i)
 7|         bgez $t4, seguir    # Si máximo >= v(i) seguir
 -|         nop
 -|         nop
 8|         add $t2,$t3,$0      # $t2 = nuevo maximo = v(i)
 9|         srl $t5,$t0,2       # $t5 = pos_max = i
10| seguir: addi $t0,$t0,4      # i++
11|         addi $t1,$t1,-1     # Decrementar contador
12|         bne $t1,$0,bucle+8
 5|         lw  $t3,V($t0)      # Leer elemento v(i)
 -|         nop
13|         sw $t2, maximo      # Resultados
14|         sw $t5, pos_max
```

No se pueden insertar las instrucciones objetivo del salto interior ya que las instrucciones siguientes dependen de estas. Puede haber una solución alternativa colocando la instrucción 10 en lugar de una de las instrucciones `nop` siguientes a (7).

■