# Riesgos de datos y saltos retardados en WinMIPS64

Salvador Carrillo Fuentes

Noviembre 2016

## 1 Resultados del programa *P2.s*

- Ciclos empleados en la ejecución del programa: 4586
- Instrucciones ejecutadas: 1819
- `CPI` (*Cycles Per Instruction*): 2.521
- Paradas del cauce debidas a riesgos:
  - `RAW`: 2652
  - `WAW`: 0
  - `WAR`: 0
- Paradas del cauce debidas a:
  - Saltos tomados: 111
  - Fallos en la predicción de salto: 0

## 2 Indicar en qué dirección de memoria está cada uno de los elementos, empezando por el elemento 0

El programa ha ordenador los elementos del array de menor a mayor. Ahora el array sigue el siguiente orden: `A`: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, donde la posición cero esta ocupada por el valor 0, la uno por el 1 y así sucesivamente. Las posiciones de memoria que ocupa cada elemento, empezando por la posición cero, es la siguiente:

Posición cero (valor 0) y posición uno (valor 1): ambos valores están en la posición 0018 de memoria, donde 0018 está expresado en hexadecimal y los 8 bytes (32 bits) más significativos corresponden al elemento uno (valor 1) y los 32 bits menos significativos corresponden al elemento cero (valor 0). Esto es así

porque la memoria está organizada en palabras de 64 bits. En consonancia con la ordenación *little-endian* donde el primer elemento ocupa los primeros 32 bits y el segundo los 32 restantes de los 64 que tiene la palabra completa.

Siguiendo esta lógica, van apareciendo el resto de los elementos del array tras la ordenación. A partir de ahora, me referiré a la posición en el array y no indicaré su valor por simplicidad, ya que siempre coincidirá.

Por tanto:

- Posición memoria 0020: elemento pos. dos (32 bits menos significativos) y elemento pos. tres (32 bits más significativos).
- Posición memoria 0028: elemento pos. cuatro (32 bits menos significativos) y elemento pos. cinco (32 bits más significativos).
- Posición memoria 0030: elemento pos. seis (32 bits menos significativos) y elemento pos. siete (32 bits más significativos).
- Posición memoria 0038: elemento pos. ocho (32 bits menos significativos) y elemento pos. nueve (32 bits más significativos).

# 3 Riesgos RAW

El primer riesgo `RAW` aparece entre las instrucciones:

```
daddi R2, R0, 10
  sw R2, n(R0)
```

Tenemos que insertar dos burbujas para solventar el riesgo y que el `sw` pueda leer el valor correcto del registro 2, alineando la etapa de `WB` de la instrucción `daddi` con la etapa `ID` de la instrucción `sw`.

De aquí deducimos que en esta implementación del `MIPS` hay adelantamiento en el banco de registros, puesto que se lee y se escribe en el mismo ciclo.

En conclusión, tenemos dos ciclos de pérdida debido a la inserción de dos burbujas en el cauce.

El segundo riesgo `RAW` aparece entre las instrucciones:

```
daddi R15, R0, 0
  sw R15, i(R0)
```

Al igual que en caso anterior, tenemos que insertar dos burbujas, perdiendo dos ciclos de reloj.

El tercer riesgo `RAW` aparece entre las instrucciones:

```
daddi R18, R0, 1
dsub R19, R17, R18
```

Tenemos que insertar dos burbujas para solucionar el riesgo de datos por el registro 18.

Nótese que también había dependencia entre `lw R17, n(R0)` y `dsub R19, R17, R18`, que habría sido un riesgo de datos que hubiera requerido la inserción de una sola burbuja por el registro 17.

Al solucionar la dependencia a distancia uno también hemos solucionado la dependencia a distancia dos.

Finalmente, el `HW` implementa anticipación en el banco de registros.

# 4 Comprobar el impacto del mecanismo de anticipación en el tiempo de ejecución del programa

- Ciclos empleados en la ejecución del programa: 3192
- Instrucciones ejecutadas: 1819
- `CPI` (*Cycles Per Instruction*): 1.755
- Paradas del cauce debidas a riesgos:
  - `RAW`: 1092
  - `WAW`: 0
  - `WAR`: 0
  - Estructurales: 166
- Paradas del cauce debidas a:
  - Saltos tomados: 111
  - Fallos en la predicción de salto: 0

Factor de mejora respecto a la implementación sin cortocircuitos (ratios):

- Ciclos empleados en la ejecución del programa: $\frac{4586}{3192} = 1.43$
- Instrucciones ejecutadas: $\frac{1819}{1819} = 1$; no hay mejora.
- `CPI` (*Cycles Per Instruction*): $\frac{2.521}{1.755} = 1.43$
- Paradas del cauce debidas a riesgos `RAW`: $\frac{2652}{1092} = 2.42$

# 5 Diferencias con la sección 3

En el ciclo 20 las estadísticas a destacar de la implementación sin cortocircuitos son:

- Nº instrucciones: 9
- `CPI`: 2.222
- Inserción de burbujas por riesgos `RAW`: 8

Por otro lado, la implementación con cortocircuitos presenta las siguientes estadísticas:

- Nº instrucciones: 14
- `CPI`: 1.429
- Inserción de burbujas por riesgos `RAW`: 1

Por tanto, cabe destacar que el número de instrucciones ejecutadas es mayor en la implementación con cortocircuitos, puesto que no se tienen que hacer tantas paradas como en la implementación sin cortocircuitos.

Esto, tiene su claro impacto en el `CPI`, ya que si para los mismos ciclos tenemos más instrucciones ejecutadas, significa que en la implementación con cortocircuitos el `CPI` será menor.

Por último, hay una diferencia notable en la inserción de burbujas por riesgos `RAW`, debido a que en la implementación con cortocircuitos se tiene el dato disponible nada más es producido y puede enviarse a la/s entrada/s de la unidad funcional que las consume.

Este hecho ahorra muchas inserciones de burbujas, pero no todas, como se verá a continuación.

Entre las instrucciones:

```
slt R20, R16, R19
beqz R20, ENDFOR1
```

hay un riesgo de datos por el registro 20. El `slt` produce el resultado al final de la etapa `EX`, y el `beqz` necesita ese resultado en la etapa `ID`.

El cortocircuito `EX/ID` (sin inserción de burbuja) no es factible en el tiempo y hay que insertar una burbuja para que, ahora sí, se pueda anticipar el resultado de la salida de la `ALU` a la entrada de `ID` de la instrucción `beqz`.

Esto implica que el salto se resuelve en `ID`, como se indicó en la práctica 1.

# 6 Optimizaciones

- Ciclos empleados en la ejecución del programa: 1127
- Instrucciones ejecutadas: 858
- `CPI` (*Cycles Per Instruction*): 1.314
- Paradas del cauce debidas a riesgos:
  - `RAW`: 154
  - `WAW`: 0
  - `WAR`: 0
  - Estructurales: 0
- Paradas del cauce debidas a:
  - Saltos tomados: 111
  - Fallos en la predicción de salto: 0

Factor de mejora respecto a la implementación sin cortocircuitos:

- Ciclos empleados en la ejecución del programa: $\frac{3192}{1127} = 2.83$
- Instrucciones ejecutadas: $\frac{1819}{858} = 2.12$
- `CPI` (*Cycles Per Instruction*): $\frac{1.755}{1.314} = 1.33$
- Paradas del cauce debidas a riesgos `RAW`: $\frac{1092}{154} = 7.09$

Ahora, comparo esta última implementación con la que además hace la optimización que sustituye la instrucción de resta por una de suma, quedando:

```
daddi R24, R0, 1 ; A [j-1]
dsub R25, R23, R24
```

Las estadísticas tras esta mejora son:

- Ciclos empleados en la ejecución del programa: 809
- Instrucciones ejecutadas: 585
- `CPI` (*Cycles Per Instruction*): 1.383
- Paradas del cauce debidas a riesgos:
  - `RAW`: 109
  - `WAW`: 0
  - `WAR`: 0
  - Estructurales: 0

- Paradas del cauce debidas a:
  - Saltos tomados: 111
  - Fallos en la predicción de salto: 0
- Ciclos empleados en la ejecución del programa: $\frac{1127}{809} = 1.39$
- Instrucciones ejecutadas: $\frac{858}{585} = 1.46$
- `CPI` (*Cycles Per Instruction*): $\frac{1.314}{1.383} = 0.95$ (peor)
- Paradas del cauce debidas a riesgos `RAW`: $\frac{154}{109} = 1.41$

# 7 Salto retardado y *data forwarding*

Al habilitar la opción *delay slot* (hueco de retardo), estamos configurando el procesador para que inserte instrucciones seguras tras los saltos. Como se usa la predicción de salto no tomado y el salto se resuelve en `ID`, cuando fallamos y sí se debería haber saltado hay que insertar una burbuja tras el *flush* correspondiente.

Al configurar la opción de hueco de retardo, metemos una instrucción que siempre se ejecutará, haciéndose efectivo el salto tras esa instrucción, así se eliminan las burbujas por salto en el `MIPS`, puesto que esa instrucción siempre será válida (o una instrucción `nop` si el compilador no encuentra una candidata válida).

Ejemplos de riesgos `RAW`:

```
slt R20, R8, R19
beqz R20, ENDFOR1;
```

Riesgos de datos a distancia uno. Se activa el cortocircuito `EX/ID` pero es necesario insertar una burbuja para ello.

Ese es el par de instrucciones tipo que generan todas las inserciones de burbujas por riesgos `RAW` en el programa.

Como el salto se resuelve en `ID`, necesita el valor del registro en el que escribe la instrucción anterior para esa etapa, no estando adelantada en el tiempo. Por tanto, es necesario una burbuja para poder activar el cortocircuito.

# 8 Rellenado de los huecos de retardo

Busco cambiar las instrucciones `nop` de la última versión del programa (*P2-1.s*) por instrucciones útiles. Intento primero en bloque básico principal, luego en bloque básico siguiente y, como última opción en el bloque básico destino.

- Hueco tras la llamada a la función `jal` Burbuja. Sustituyo el `nop` por:

  `daddi R10,R0,10` ; inic. variable $n$ en registro `R10` (disponible en el bloque destino)

- Primer hueco, tras `beqz R20, ENDFOR1 ;` Sustituyo el `nop` por:

  `daddi R9, R10, -1` ; `for` $j = n - 1$; (disponible en el bloque básico principal)

- Segundo hueco tras `beqz R27, ENDFOR2 ;` Sustituyo el `nop` por:

  `dsll R2, R9, 2 ; A[j]` (disponible en el bloque básico siguiente)

- Tercer hueco, tras `beqz R5, IFELSE1 ; if (A[j] < A[j-1])` Sustituyo el `nop` por:

  `dsll R16,R9,2 ; temp = A[j]` (disponible en el bloque básico principal)

- Cuarto hueco, tras `j IFEND1;` Sustituyo el `nop` por:

  `sw R11,A(R6); A[j-1] = temp` (disponible en el bloque básico principal)

- Quinto hueco, tras: `j FOR2;` Sustituyo el `nop` por:

  `daddi R9,R9,-1 ; j--` (disponible en el bloque básico principal)

- Sexto hueco, tras: `j FOR1` Sustituyo el `nop` por:

  `daddi R8,R8,1 ; i++` (disponible en el BBP)

- Séptimo hueco, tras: `jr R31 ; Return to MAIN`. Sustituyo el `nop` por:

  `halt` (disponible en el bloque de destino)

Por tanto, ha sido posible encontrar instrucciones útiles para sustituir todas las instrucciones `nop` que tenía el programa inicialmente.

Meter esas instrucciones tendrá un impacto positivo en el programa tal y como se ve a continuación.

Cabe recalcar que la sustitución de esas instrucciones útiles, no han cambiado la semántica del programa puesto que sigue actuando de la misma manera, ordenando los elementos de un array de menor a mayor.

El programa modificado se encuentra en el fichero *P2-2.s*

Las estadísticas obtenidas por esta nueva versión optimizada del programa son:

- Ciclos empleados en la ejecución del programa: 978
- Instrucciones ejecutadas: 820
- `CPI` (*Cycles Per Instruction*): 1.193
- Paradas del cauce debidas a riesgos:
  - `RAW`: 154
  - `WAW`: 0
  - `WAR`: 0
  - Estructurales: 0
- Paradas del cauce debidas a:
  - Saltos tomados: 0
  - Fallos en la predicción de salto: 0

Se puede observar una mejora ya que el programa *P2-1.s* ofrece las siguientes estadísticas:

- Ciclos empleados en la ejecución del programa: 1127
- Instrucciones ejecutadas: 969
- `CPI` (*Cycles Per Instruction*): 1.163
- Paradas del cauce debidas a riesgos:
  - `RAW`: 154
  - `WAW`: 0
  - `WAR`: 0
  - Estructurales: 0
- Paradas del cauce debidas a:
  - Saltos tomados: 0
  - Fallos en la predicción de salto: 0