

Memoria DLX

Optimización cálculo en DLX

David Cruz García 70957692R
Sergio Fernández Marcos 70912721H

Contenido

1. Análisis del problema y código sin optimizar.....	2
2. Primera optimización.....	4
3. Segunda optimización	5
3. Pruebas.....	11
4. Conclusiones.....	13

1. Análisis del problema y código sin optimizar

En este problema se nos proponía realizar un cálculo relativamente sencillo en el que se implicaban matrices y vectores, utilizando la siguiente configuración:

- Forwarding activado
- Hardware disponible y número de ciclos para cada tipo de operación:

	Count:	Delay:
Addition Units:	1	2
Multiplication Units:	1	5
Division Units:	1	19

Realizando un pequeño análisis previo, pudimos ver que la mayor limitación en el número total de ciclos que se van a emplear para este cálculo viene dado por el gran número de multiplicaciones que hay que hacer en total, ya que el resto de operaciones se utilizan mínimamente comparado con las multiplicaciones.

Para ver el número total de multiplicaciones que se iban a emplear en este cálculo, decidimos codificar todas ellas sin ningún tipo de optimización, consiguiendo el siguiente resultado:

```
Statistics
Total:
320 Cycle(s) executed.
ID executed by 96 Instruction(s).
2 Instruction(s) currently in Pipeline.

Hardware configuration:
Memory size: 32768 Bytes
faddEX-Stages: 1, required Cycles: 2
fmulEX-Stages: 1, required Cycles: 5
fddivEX-Stages: 1, required Cycles: 19
Forwarding enabled.

Stalls:
RAW stalls: 33 (10.31% of all Cycles), thereof:
LD stalls: 0 (0.00% of RAW stalls)
Branch/Jump stalls: 0 (0.00% of RAW stalls)
Floating point stalls: 33 (100.00% of RAW stalls)
WAW stalls: 0 (0.00% of all Cycles)
Structural stalls: 186 (58.12% of all Cycles)
Control stalls: 0 (0.00% of all Cycles)
Trap stalls: 4 (1.25% of all Cycles)
Total: 223 Stall(s) (69.70% of all Cycles)

Conditional Branches):
Total: 4 (4.17% of all Instructions), thereof:
taken: 0 (0.00% of all cond. Branches)
not taken: 4 (100.00% of all cond. Branches)

Load-/Store-Instructions:
Total: 29 (30.21% of all Instructions), thereof:
Loads: 4 (13.79% of Load-/Store-Instructions)
Stores: 25 (86.21% of Load-/Store-Instructions)

Floating point stage instructions:
Total: 58 (60.42% of all Instructions), thereof:
Additions: 9 (15.52% of Floating point stage inst.)
Multiplications: 45 (77.60% of Floating point stage inst.)
Divisions: 4 (7.00% of Floating point stage inst.)

Traps:
Traps: 1 (1.04% of all Instructions)
```

Estadísticas de ejecución del código sin optimizar
(fichero **DLX_sinOptimizar.S**)

Como acabamos de decir, en un primer intento sin ningún tipo de optimización y codificando todas las operaciones, obtenemos un resultado bastante esperado.

Hay un total de 58 operaciones con números reales (o de punto flotante) entre las cuales **45 son multiplicaciones**, 9 sumas y restas, y 4 divisiones.

El análisis del número de operaciones es muy importante, ya que tener tantas multiplicaciones y disponer de un único hardware para realizar esta operación implica muchos retrasos ya que solo podrá realizar las multiplicaciones de 1 en 1.

Tras analizar esta limitación, llegamos a la siguiente conclusión:

45 multiplicaciones * 5 ciclos/multiplicación = 225 ciclos

Suponiendo que el resto de operaciones, incluyendo sumas, restas y divisiones, instrucciones de carga y descarga de datos, y los saltos condicionales necesarios para comprobar que no se divide entre 0, se podría optimizar el código hasta los 225 ciclos, pero si no disponemos de otra unidad para realizar operaciones o reducimos los ciclos que tarda cada multiplicaciones en realizarse, no podremos bajar de esos 225 ciclos mínimos.

Además, todo esto suponiendo un caso ideal en el que la mayor limitación venga dada por las multiplicaciones y que el resto de operaciones se puedan realizar mientras se realizan estas multiplicaciones, pero esto no es así:

- Antes de poder multiplicar por primera vez, se necesita al menos cargar los dos operandos con los que se va a realizar la multiplicación (por ejemplo, si la primera multiplicación es $a1*a2$, necesitamos cargar en los registros de punto flotante los valores de $a1$ y $a2$).

Por lo tanto, además de esos 225 ciclos mínimos, tenemos que sumarles mínimo **otros 2 ciclos** más correspondientes a las 2 cargas.


- Una vez se ha multiplicado por última vez, el último resultado hay que sumarlo para calcular el check, y posteriormente guardar en memoria el resultado del check, por lo tanto tenemos que **sumar otros 3 ciclos mínimo** (porque la suma tarda 2 ciclos en realizarse y además hay dependencia de datos).
- Todo esto lo hemos hecho sin tener en cuenta dependencias verdaderas entre instrucciones que calculan un resultado y guardan ese resultado (o lo usan para calcular otro), es decir, no hemos

tenido en cuenta conflictos RAW (que como podemos ver, hay 33). Por lo tanto, puede que haya todavía más esperas en estos casos. Además, hay que sumarle el ciclo necesario para el trap final para que acabe el programa.

Es decir, en total tenemos que, **aproximadamente**, podremos optimizar este código hasta los **230 ciclos en el mejor caso**, que son 90 menos que los que tenemos sin optimizar, así que es un gran avance.

2. Primera optimización

Como ya hemos comentado, como primera optimización tratamos de conseguir estos 230 ciclos mínimos, consiguiendo el siguiente resultado:



```
Statistics
Total:
  241 Cycle(s) executed.
  ID executed by 96 Instruction(s).
  2 Instruction(s) currently in Pipeline.

Hardware configuration:
  Memory size: 32768 Bytes
  faddEX-Stages: 1, required Cycles: 2
  fmulEX-Stages: 1, required Cycles: 5
  fdivEX-Stages: 1, required Cycles: 19
  Forwarding enabled.

Stalls:
  RAW stalls: 0 (0.00% of all Cycles), thereof:
    LD stalls: 0 (0.00% of RAW stalls)
    Branch/Jump stalls: 0 (0.00% of RAW stalls)
    Floating point stalls: 0 (0.00% of RAW stalls)
  WAW stalls: 0 (0.00% of all Cycles)
  Structural stalls: 139 (57.68% of all Cycles)
  Control stalls: 0 (0.00% of all Cycles)
  Trap stalls: 4 (1.66% of all Cycles)
  Total: 143 Stall(s) (59.34% of all Cycles)

Conditional Branches):
  Total: 4 (4.17% of all Instructions), thereof:
    taken: 0 (0.00% of all cond. Branches)
    not taken: 4 (100.00% of all cond. Branches)

Load-/Store-Instructions:
  Total: 29 (30.21% of all Instructions), thereof:
    Loads: 4 (13.79% of Load-/Store-Instructions)
    Stores: 25 (86.21% of Load-/Store-Instructions)

Floating point stage instructions:
  Total: 58 (60.42% of all Instructions), thereof:
    Additions: 9 (15.52% of Floating point stage inst.)
    Multiplications: 45 (77.60% of Floating point stage inst.)
    Divisions: 4 (7.00% of Floating point stage inst.)

Traps:
  Traps: 1 (1.04% of all Instructions)
```

Estadísticas de ejecución del código
tras 1ª optimización

Como podemos ver, hemos conseguido bajar de 320 a 241 ciclos, ahorrando casi 80 ciclos para realizar el cálculo, consiguiendo una mejora de un 25% aproximadamente.

Además, no solo eso, sino que conseguimos quitar todos los conflictos RAW, pero realmente, como habíamos comentado, el problema no solo estaba en las dependencias, si no que estábamos limitados principalmente por las multiplicaciones, asique decidimos trabajar para reducir este número de multiplicaciones.

Este resultado lo conseguimos, principalmente, reordenando las instrucciones, aprovechando en los 5 ciclos que se realiza cada multiplicación para hacer otro tipo de operaciones, además de conseguir tener las divisiones lo mejor separadas posible ya que tardan 19 ciclos y necesitábamos sus resultados sin ningún retraso (ya que si alguna tuviera que esperar, tendríamos que esperar más ciclos, y 19 son muchos comparados con otras operaciones). Además de esto, solo utilizamos variables en memoria para cargar los valores iniciales (a1...a4) y para sacar los resultados (M, VM, HM y check), utilizando los registros de punto flotante para realizar todas las operaciones y evitar accesos a memoria para la carga/descarga de variables.

3. Segunda optimización

Como hemos comentado, tras obtener los 241 ciclos estábamos muy limitados por el número tan alto de multiplicaciones ya que como ya comentamos, el número mínimo aproximado de ciclos era de 230 (sin tener en cuenta posibles dependencias), asique habíamos obtenido un buen resultado dentro de lo esperado, pero se podía optimizar más.

Para poder seguir optimizando este cálculo, teníamos que reducir el número de multiplicaciones. Para ello, decidimos desarrollar los cálculos sin sustituir las variables por valores numéricos, buscando principalmente multiplicaciones que pudieran ser redundantes y que pudiéramos hacer una sola vez, sirviéndonos ese resultado para otras operaciones sin tener que calcularlo varias veces.

Primero, desarrollamos el producto de Kronecker entre $MF(a1,a2)$ y $MF(a3, a4)$ que es la principal operación donde tenemos una gran parte de multiplicaciones. Con esto conseguimos el siguiente resultado:

$$MF(a1,a2) = \begin{pmatrix} a1 & a1/a2 \\ a2 & a1 * a2 \end{pmatrix} \otimes MF(a3,a4) = \begin{pmatrix} a3 & a3/a4 \\ a4 & a3 * a4 \end{pmatrix}$$

$$\otimes = \begin{pmatrix} a1 * a3 & a1 * \frac{a3}{a4} & \frac{a1}{a2} * a3 & \frac{a1}{a2} * \frac{a3}{a4} \\ a1 * a4 & a1 * a3 * a4 & \frac{a1}{a2} * a4 & \frac{a1}{a2} * a3 * a4 \\ a2 * a3 & a2 * \frac{a3}{a4} & a1 * a2 * a3 & a1 * a2 * \frac{a3}{a4} \\ a2 * a4 & a2 * a3 * a4 & a1 * a2 * a4 & a1 * a2 * a3 * a4 \end{pmatrix}$$

Tras desarrollar este producto vimos que hay muchas operaciones que se pueden organizar de tal forma que pudiéramos reutilizar el valor de una misma multiplicación para varias posiciones de la matriz.

Ahora partiendo de esta matriz desarrollada, tomamos diferentes grupos de operaciones:

- Como las divisiones tardan 19 ciclos en ejecutarse y no nos interesa tener más divisiones, decidimos dejar las dos divisiones que ya teníamos, tomando : $x2 = \frac{a1}{a2}$, $x3 = \frac{a3}{a4}$.
- Decidimos tomar los siguientes grupos para las multiplicaciones para un primer "agrupamiento": $x1 = a1 * a3$, $x4 = a2 * a4$.

Ahora sustituyendo, obtuvimos la siguiente matriz:

$$\begin{pmatrix} x1 & a1 * x3 & x2 * a3 & x2 * x3 \\ a1 * a4 & x1 * a4 & x2 * a4 & x2 * a3 * a4 \\ a2 * a3 & a2 * x3 & x1 * a2 & a1 * a2 * x3 \\ x4 & a3 * x4 & a1 * x4 & x1 * x4 \end{pmatrix}$$

Con este resultado, realmente teníamos el mismo número de operaciones, pero ya daba la forma que nosotros buscábamos. Ahora, siguiendo el mismo procedimiento decidimos crear 2 nuevas sustituciones: $x5 = x2 * a3$ y $x6 = a2 * a3$.

Volviendo a sustituir en la matriz ahora tenemos:

$$\begin{pmatrix} x1 & a1 * x3 & x5 & x2 * x3 \\ a1 * a4 & x1 * a4 & x2 * a4 & x5 * a4 \\ x6 & a2 * x3 & x1 * a2 & a1 * a2 * x3 \\ x4 & a3 * x4 & a1 * x4 & x1 * x4 \end{pmatrix}$$

Ahora sí, hemos ahorrado 1 multiplicación para calcular el producto de Kronecker.

Ahora, para poder continuar, necesitábamos desarrollar el determinante de MF(a2,a3), además de tener en cuenta que está multiplicado por (a1+a4). Por tanto, llamamos a todo esto en su conjunto '**y**' y comenzamos a desarrollarlo:

$$y = \frac{a1 + a4}{\begin{vmatrix} a2 & a2/a3 \\ a3 & a2 * a3 \end{vmatrix}} = \frac{a1 + a4}{\begin{vmatrix} a2 & a2/a3 \\ a3 & x6 \end{vmatrix}} = \frac{a1 + a4}{a2 * x6 + a3 * \left(\frac{a2}{a3}\right)} = \frac{a1 + a4}{a2 * x6 + a2}$$

Como podemos ver, no solo nos hemos ahorrado 1 multiplicación, sino que además nos hemos ahorrado una división que también es muy importante, ya que no dependemos de ese cálculo de 19 ciclos, y además nos ahorramos una comprobación de división entre 0 (por tanto, ahorramos también instrucciones condicionales).

Ahora, esta '**y**' calculada la tenemos que multiplicar por cada elemento de la matriz, obteniendo lo siguiente:

$$M = \begin{pmatrix} x1 * y & a1 * x3 * y & x5 * y & x2 * x3 * y \\ a1 * a4 * y & x1 * a4 * y & x2 * a4 * y & x5 * a4 * y \\ x6 * y & a2 * x3 * y & x1 * a2 * y & a1 * a2 * x3 * y \\ x4 * y & a3 * x4 * y & a1 * x4 * y & x1 * x4 * y \end{pmatrix}$$

Siguiendo la técnica para aplicar sustituciones que seguimos antes, decidimos realizar las siguientes sustituciones (acordes a las operaciones que se repetían más veces):

$$z1 = x1 * y, z2 = x4 * y, z3 = a1 * y, z4 = x3 * y, z5 = a4 * y$$

Sustituyendo en la matriz M obtenemos:

$$M = \begin{pmatrix} z1 & z4 * a1 & x5 * y & x2 * z4 \\ a4 * z3 & a4 * z1 & x2 * z5 & x5 * z5 \\ x6 * y & a2 * z4 & z1 * a2 & a1 * a2 * z4 \\ z2 & a3 * z2 & a1 * z2 & z1 * x4 \end{pmatrix}$$

En este caso, hemos conseguido ahorrar casi otras 10 multiplicaciones.

Ahora ya solo nos quedaba calcular VM, HM y check, y tras meditarlo, decidimos calcularlo como ya lo teníamos. La razón es que solo se trata de multiplicaciones, además de multiplicaciones sobre cálculos y valores que acabábamos de calcular para calcular M, por lo que para poder realizar agrupaciones implicaría volver atrás y realizar otros cálculos que al final seguramente provocarían más multiplicaciones que hacerlo directamente, así que decidimos hacerlas directamente con los valores ya obtenidos.

Con todo esto, desarrollamos este nuevo cálculo optimizado de la matriz M en un nuevo programa y sin ninguna optimización sobre el código (no recolocación del código, ni desenrollar bucles ni ningún otro tipo de optimización), obteniendo el siguiente resultado:

```

Statistics
Total:
245 Cycle(s) executed.
ID executed by 81 Instruction(s).
2 Instruction(s) currently in Pipeline.

Hardware configuration:
Memory size: 32768 Bytes
faddEX-Stages: 1, required Cycles: 2
fmulEX-Stages: 1, required Cycles: 5
fdivEX-Stages: 1, required Cycles: 19
Forwarding enabled.

Stalls:
RAW stalls: 39 (15.92% of all Cycles), thereof:
LD stalls: 0 (0.00% of RAW stalls)
Branch/Jump stalls: 0 (0.00% of RAW stalls)
Floating point stalls: 39 (100.00% of RAW stalls)
WAW stalls: 0 (0.00% of all Cycles)
Structural stalls: 119 (48.57% of all Cycles)
Control stalls: 0 (0.00% of all Cycles)
Trap stalls: 4 (1.63% of all Cycles)
Total: 162 Stall(s) (66.12% of all Cycles)

Conditional Branches):
Total: 3 (3.70% of all Instructions), thereof:
taken: 0 (0.00% of all cond. Branches)
not taken: 3 (100.00% of all cond. Branches)

Load-/Store-Instructions:
Total: 29 (35.80% of all Instructions), thereof:
Loads: 4 (13.79% of Load-/Store-Instructions)
Stores: 25 (86.21% of Load-/Store-Instructions)

Floating point stage instructions:
Total: 45 (55.56% of all Instructions), thereof:
Additions: 9 (20.00% of Floating point stage inst.)
Multiplications: 33 (73.33% of Floating point stage inst.)
Divisions: 3 (6.67% of Floating point stage inst.)

Traps:
Traps: 1 (1.23% of all Instructions)
  
```

Estadísticas de ejecución del código tras optimización matemática y sin optimización de código

Con esto, habíamos conseguido nuestro objetivo, e incluso mejor del esperado, pasando de:

- 58 operaciones en punto flotante → 45
- **45 multiplicaciones → 33 multiplicaciones**
- 4 divisiones → 3 divisiones (con el ahorro de las instrucciones para la comprobación de posible división entre 0).

Los resultados eran muy buenos, habíamos conseguido, sin optimizar código, casi los mismos ciclos que en la primera versión con el código optimizado. Además, reduciendo el número de multiplicaciones de 45 a 33 suponía que el límite mínimo de operaciones en el caso ideal también había bajado, teniendo ahora:

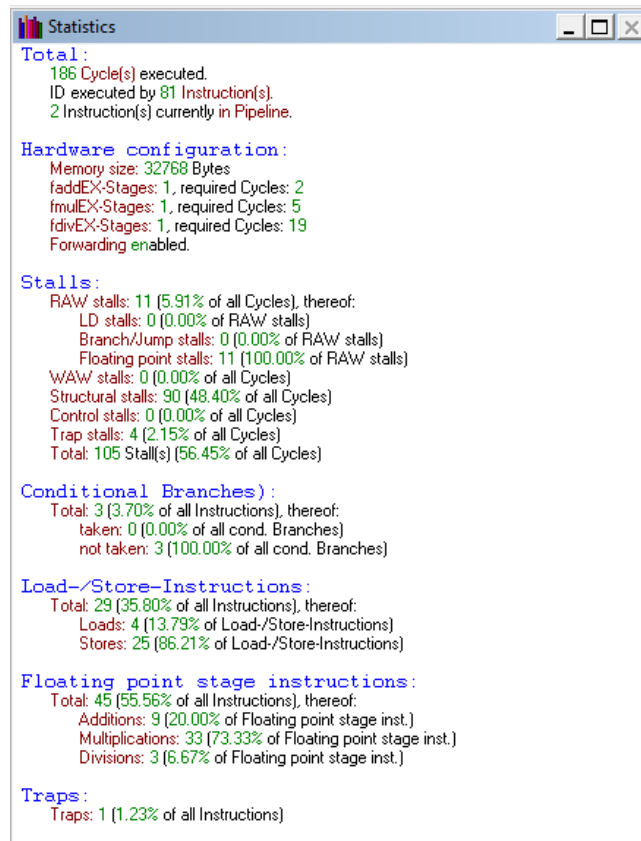
33 multiplicaciones * 5 ciclos/multiplicación = 165 ciclos

Esto además hay que sumarle como dijimos antes algunas operaciones en el antes y el después de estas multiplicaciones, rondando por lo tanto los **170 ciclos mínimos**. Esto nos daba mucho más margen para trabajar ya

que ahora podíamos bajar **más de 50 ciclos** con respecto al primer diseño, cosa que antes nos era imposible porque el mínimo era de más o menos 230 ciclos.

Ahora, ya solo faltaba optimizar este código, principalmente reordenando instrucciones como hicimos antes, para conseguir aprovechar que, mientras que ejecutaban las multiplicaciones y divisiones (las operaciones más costosas en cuanto a ciclos) se ejecutaran otro tipo de instrucciones.

Así conseguimos el siguiente resultado:



```
Statistics
Total:
186 Cycle(s) executed.
ID executed by 81 Instruction(s).
2 Instruction(s) currently in Pipeline.

Hardware configuration:
Memory size: 32768 Bytes
faddEX-Stages: 1, required Cycles: 2
fmulEX-Stages: 1, required Cycles: 5
fdivEX-Stages: 1, required Cycles: 19
Forwarding enabled.

Stalls:
RAW stalls: 11 (5.91% of all Cycles), thereof:
  LD stalls: 0 (0.00% of RAW stalls)
  Branch/Jump stalls: 0 (0.00% of RAW stalls)
  Floating point stalls: 11 (100.00% of RAW stalls)
WAW stalls: 0 (0.00% of all Cycles)
Structural stalls: 90 (48.40% of all Cycles)
Control stalls: 0 (0.00% of all Cycles)
Trap stalls: 4 (2.15% of all Cycles)
Total: 105 Stall(s) (56.45% of all Cycles)

Conditional Branches):
Total: 3 (3.70% of all Instructions), thereof:
  taken: 0 (0.00% of all cond. Branches)
  not taken: 3 (100.00% of all cond. Branches)

Load-/Store-Instructions:
Total: 29 (35.80% of all Instructions), thereof:
  Loads: 4 (13.79% of Load-/Store-Instructions)
  Stores: 25 (86.21% of Load-/Store-Instructions)

Floating point stage instructions:
Total: 45 (55.56% of all Instructions), thereof:
  Additions: 9 (20.00% of Floating point stage inst.)
  Multiplications: 33 (73.33% of Floating point stage inst.)
  Divisions: 3 (6.67% of Floating point stage inst.)

Traps:
Traps: 1 (1.23% of all Instructions)
```

Estadísticas de ejecución del código tras optimización matemática y optimización de código (fichero **DLX_optimizado.S**)

Como podemos ver, hemos pasado de un código sin optimizar de 320 ciclos, a un código optimizado de 241 ciclos, pero que tras un trabajo matemático se ha conseguido una nueva versión mucho más optimizada de **186 ciclos**.

Para conseguir esta optimización hemos utilizado, principalmente, la reordenación de instrucciones, intentando aprovechar al máximo el realizar todo tipo de instrucciones en las esperas entre multiplicaciones y divisiones.

Además, para evitar problemas entre divisiones se ha tratado de separarlas todo lo posible para que los cálculos sean lo más tempranos posibles, ya

que gracias a la división que da como resultado 'y' podemos continuar calculando las 'z's, y por tanto, acabando de calcular M.

De hecho, la principal dependencia de datos y que creaba varios conflictos RAW era esta división para calcular 'y', por lo tanto tratamos de comenzar su cálculo lo antes posible pero como había que calcular varios valores antes para poder realizarla, comprobar la posible división entre 0 y necesitar de forma temprana el resultado de las otras dos divisiones, estuvimos obligados a mantener la división como la segunda en realizarse.

En mejora a esta solución y aunque hayamos ahorrado una división, hemos aprovechado ciclos que desperdiciábamos por conflictos RAW para comprobar si a3 es distinto de 0 o no (aunque la división a2/a3 ya no la tenemos). Esto es porque al no tener ya esa división, aunque a3 fuera 0 el programa iba a continuar con todos los cálculos ejecutando los 186 ciclos igual y el resultado no sería el mismo al código sin optimizar, asique decidimos que en caso de que sea 0 y aprovechando paradas innecesarias saltara al final del programa ya que aunque no saliera error el resultado debía ser considerado como inválido (de hecho, guardaba valores en memoria como -0 y cosas así que no nos aportaban nada).

Por tanto, el resultado final tras optimizar todo y con los valores de entrada dados fue el siguiente:

```
Statistics
Total:
186 Cycle(s) executed.
ID executed by 83 Instruction(s).
2 Instruction(s) currently in Pipeline.

Hardware configuration:
Memory size: 32768 Bytes
faddEX-Stages: 1, required Cycles: 2
fmulEX-Stages: 1, required Cycles: 5
fdvEX-Stages: 1, required Cycles: 19
Forwarding enabled.

Stalls:
RAW stalls: 9 (4.84% of all Cycles), thereof:
LD stalls: 0 (0.00% of RAW stalls)
Branch/Jump stalls: 0 (0.00% of RAW stalls)
Floating point stalls: 9 (100.00% of RAW stalls)
WAW stalls: 0 (0.00% of all Cycles)
Structural stalls: 90 (48.40% of all Cycles)
Control stalls: 0 (0.00% of all Cycles)
Trap stalls: 4 (2.15% of all Cycles)
Total: 103 Stall(s) (55.38% of all Cycles)

Conditional Branches):
Total: 4 (4.82% of all Instructions), thereof:
taken: 0 (0.00% of all cond. Branches)
not taken: 4 (100.00% of all cond. Branches)

Load-/Store-Instructions:
Total: 29 (34.94% of all Instructions), thereof:
Loads: 4 (13.79% of Load-/Store-Instructions)
Stores: 25 (86.21% of Load-/Store-Instructions)

Floating point stage instructions:
Total: 45 (54.22% of all Instructions), thereof:
Additions: 9 (20.00% of Floating point stage inst.)
Multiplications: 33 (73.33% of Floating point stage inst.)
Divisions: 3 (6.67% of Floating point stage inst.)

Traps:
Traps: 1 (1.20% of all Instructions)
```

Estadísticas de ejecución del código tras optimización matemática y optimización de código, y añadida la pequeña mejora.
(fichero **DLX_optimizado.S**)

3. Pruebas

Para comprobar que nuestra solución, tanto optimizada como no, era correcta, decidimos hacer varias pruebas, entre las que utilizábamos distintos valores en la entrada para ver si todo funcionaba correctamente:

1ª prueba: **a1=1.1 a2=2.2 a3=3.3 a4=4.4**

\$DATA	1.1	2.2	3.3	4.4
M	1.44968	0.329473	0.658946	0.14976
M+0x10	1.93291	6.37859	0.878594	2.89936
M+0x20	2.89936	0.658946	3.1893	0.72484
M+0x30	3.86581	12.7572	4.2524	14.0329
HM	11.2084	8.40629	13.5622	10.1716
VM	2.8021	2.10157	0.578946	0.434209
check	49.2653	0	0	0

Statistics
Total: 320 Cycle(s) executed.

Código sin optimizar

\$DATA	1.1	2.2	3.3	4.4
M	1.44968	0.329473	0.658946	0.14976
M+0x10	1.93291	6.37859	0.878594	2.89936
M+0x20	2.89936	0.658946	3.1893	0.72484
M+0x30	3.86581	12.7572	4.2524	14.0329
HM	11.2084	8.40629	13.5622	10.1716
VM	2.8021	2.10157	0.578946	0.434209
check	49.2653	0	0	0

Statistics
Total: 186 Cycle(s) executed.

Código optimizado

2ª prueba: **a1=1.0 a2=2.2 a3=0.0 a4=4.4**
(Provocamos una división entre 0)

\$DATA	1	2.2	0	4.4
M	0	0	0	0
M+0x10	0	0	0	0
M+0x20	0	0	0	0
M+0x30	0	0	0	0
HM	0	0	0	0
VM	0	0	0	0
check	0	0	0	0

Statistics
Total: 49 Cycle(s) executed.
ID executed by 15 Instruction(s).
2 Instruction(s) currently in Pipeline.

endProgram: trap 0x0
Trap #0 occurred.

Código sin optimizar

\$DATA	1	2.2	0	4.4
M	0	0	0	0
M+0x10	0	0	0	0
M+0x20	0	0	0	0
M+0x30	0	0	0	0
HM	0	0	0	0
VM	0	0	0	0
check	0	0	0	0

Statistics

Total:
 47 Cycle(s) executed.
 ID executed by 23 Instruction(s).
 2 Instruction(s) currently in Pipeline.

Hardware configuration:
 Memory size: 32768 Bytes
 faddEX-Stages: 1, required Cycles: 2
 fmulEX-Stages: 1, required Cycles: 5

endProgram: trap 0x0

 Trap #0 occurred.

Aceptar

Código optimizado

3ª prueba: **a1=1.0 a2=1.2 a3=3.0 a4=2.5**

\$DATA	1	1.2	3	2.5
M	3.36538	1.34615	2.8045	1.12179
M+0x10	2.8045	8.41346	2.33707	7.01122
M+0x20	4.03846	1.61538	4.03846	1.61538
M+0x30	3.36538	10.0962	3.36538	10.0962
HM	13.591	16.3092	13.591	16.3092
VM	9.43818	11.3258	6.5543	7.86515
check	94.9837	0	0	0

Statistics

Total:
 320 Cycle(s) executed.

Código sin optimizar

\$DATA	1	1.2	3	2.5
M	3.36538	1.34615	2.8045	1.12179
M+0x10	2.8045	8.41346	2.33707	7.01122
M+0x20	4.03846	1.61538	4.03846	1.61538
M+0x30	3.36538	10.0962	3.36538	10.0962
HM	13.591	16.3092	13.591	16.3092
VM	9.43818	11.3258	6.5543	7.86515
VM+0x10	94.9837	0	0	0

Statistics

Total:
 186 Cycle(s) executed.

Código optimizado

4. Conclusiones

Como hemos dicho antes, la principal limitación en el cálculo propuesto era el alto número de multiplicaciones disponiendo solo de 1 unidad para realizar estas operaciones.

Con todo el desarrollo que hemos hecho nos hemos dado cuenta que no solo es importante optimizar el código sino que también es importante un análisis “matemático” (si se puede llamar así) previo al código que nos puede ayudar, y mucho, en nuestro problema.

Cabe destacar que en la versión final optimizada, la de 186 ciclos, seguimos teniendo conflictos RAW. Algunos de estos conflictos RAW realmente no suponen un problema, ya que muchos de ellos son entre multiplicaciones con dependencias de datos pero que gracias al Forwarding no supone ningún retraso, ya que la 2ª multiplicación tiene que esperar a la 1ª por el hecho de tener 1 solo unidad de cálculo, y el resultado se le avanza gracias al Forwarding. Por tanto, algunos conflictos RAW los podríamos contar como “Structural Stalls” donde la 2ª multiplicación espera a que acabe la primera por solo tener 1 unidad de cálculo.

Finalmente comentar que el problema de crear agrupaciones siguiendo nuestra idea tiene como limitación la creación de dependencias de datos, y por tanto, posibles conflictos RAW. De hecho, en otra versión que consideramos conseguimos evitar otra multiplicación más, pero las agrupaciones tenían tantas dependencias de datos que creaban conflictos RAW que no pudimos conseguir bajar por debajo de los 195 ciclos. Esto mismo crea un pequeño retraso de 3 ciclos por conflicto RAW en nuestra solución, ya que como hemos comentado para calcular las 'z's todas ellas son un valor multiplicado por 'y', y esta 'y' es calculada por una división por lo tanto hasta que esa división no acabara no podía continuar el programa, y como no disponíamos de más operaciones para insertar en ese tiempo, tuvimos que quedarnos con esos 3 ciclos perdidos (conflicto RAW entre la división para calcular y, y las multiplicaciones siguientes para calcular z).