Arquitectura de Computadores 2018/2019

Práctica 2 – DLX

Alberto Hernández Pintor, 70958670J
Miguel Cabezas Puerto,70911497J

VNiVERSiDAD
D SALAMANCA

1. Descripción de la práctica

El objetivo de esta práctica es el desarrollo y optimización de un código que realice el siguiente cálculo. *(Las matrices A, B y C serán de dimensión 3x3 con valores en punto flotante.)*

$$M = (A \times B) \bullet \frac{1}{|A + B|} + C \bullet \alpha$$

siendo:

- A, B, C y M matrices 3x3
- α número real
- A determinante de la matriz
- "×", "+" producto y suma de matrices
- "•" producto de matriz por escalar

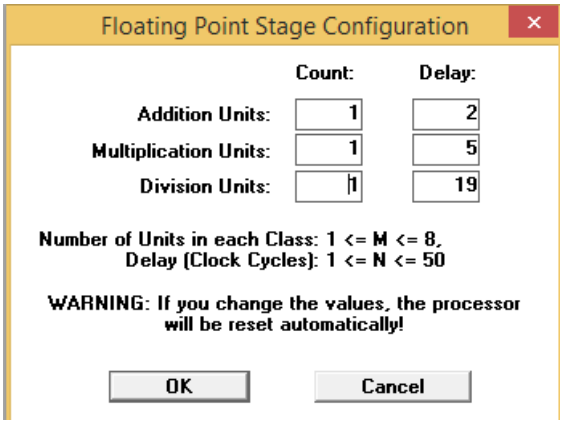
2. Planteamiento inicial

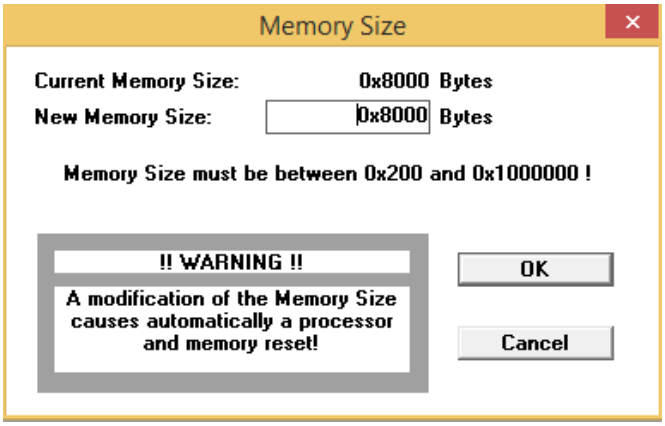
Para la realización de esta práctica hemos creado dos versiones que realizan el cálculo previamente expuesto, una no optimizada y otra optimizando el código mediante técnicas que explicaremos posteriormente. El objetivo es comparar los resultados obtenidos en ambas versiones para así ver que realmente la optimización de un determinado código puede ser muy útil para hacer programas más eficientes que tarden un menor número de ciclos con las menores detenciones posibles.

3. Configuración del entorno de ejecución

Como se indica en el enunciado las pruebas han de realizarse bajo una serie de condiciones de manera que hemos tenido que comprobar que la configuración de WINDLX era la misma que la que se especificaba en el enunciado.

1*Nota: en la versión no optimizada hemos decidido no tener bucles sino tener el código desenrollado, es por ello por lo que en la versión optimizada se mantiene. No obstante, se podría considerar este desenrollamiento una optimización.
2*El número de saltos tomados depende de si el determinante es cero o no lo es. En caso de no ser cero no se toma el salto, en caso de ser cero se toma el salto y finaliza el programa.





4. Versión no optimizada *PracticaFinal.s*

El objetivo principal ha sido que el programa realice el cálculo correctamente sin tener en cuenta los recursos utilizados ni el orden de las operaciones.

Para ello hemos codificado uno de los órdenes aritméticos posibles del cálculo de la siguiente forma:

1) Cargamos las matrices A y B.
2) Calculamos la matriz resultante del producto matricial de AxB
3) Calculamos la matriz resultante de la suma de A+B
4) Calculamos el determinante de dicha matriz. $|A+B|$
5) Comprobamos que el determinante calculado sea distinto de cero
6) En caso negativo (es igual a cero) finalizamos el programa
7) En caso afirmativo (es distinto de cero), calculamos el número resultante de dividir 1 entre el determinante calculado. $1/|A+B|$
8) Cargamos la matriz C
9) Calculamos la matriz resultante de la de haber multiplicado la de 2) por el número de 7). (AxB) •$(1/|A+B|)$
10) Cargamos la constante α
11) Calculamos la matriz resultante de multiplicar C • α
12) Calculamos la matriz resultante de sumar 9) y 11). ((AxB) •$(1/|A+B|)$ + (C • α)
13) Cargamos la matriz M con los valores de dicha matriz

Lanzamos una ejecución para los valores:

$$A = \begin{matrix} 1.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 2.0 \\ 2.0 & 0.0 & 1.0 \end{matrix} \quad B = \begin{matrix} 2.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 2.0 \\ 0.0 & 2.0 & 2.0 \end{matrix} \quad C = \begin{matrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{matrix} \quad \alpha=1.0$$

Obteniendo como resultado:

$$M = \begin{matrix} 0.667 & -0.333 & -0.5 \\ 0 & 0.1667 & -0.333 \\ -0.667 & -0.667 & 0.667 \end{matrix}$$

1*Nota: en la versión no optimizada hemos decidido no tener bucles sino tener el código desenrollado, es por ello por lo que en la versión optimizada se mantiene. No obstante, se podría considerar este desenrollamiento una optimización.
2*El número de saltos tomados depende de si el determinante es cero o no lo es. En caso de no ser cero no se toma el salto, en caso de ser cero se toma el salto y finaliza el programa.

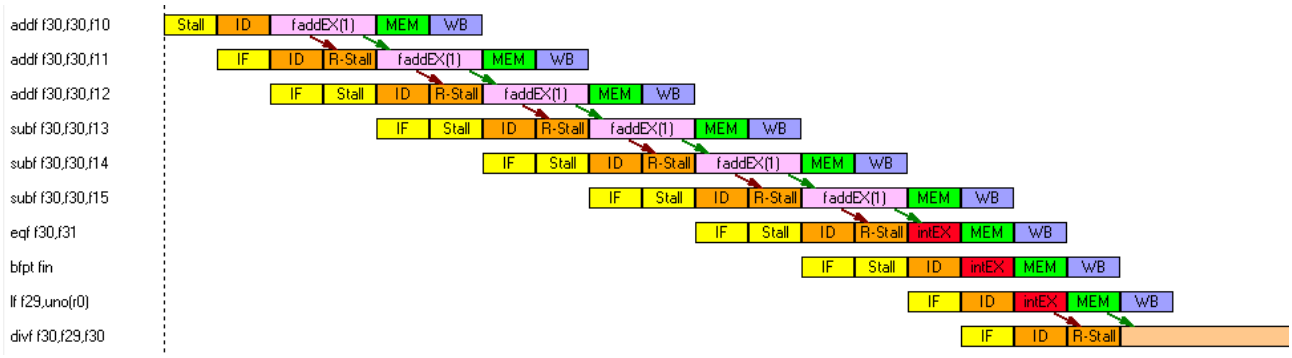
El resultado se podrá ver en los registros F0 a F8. Ha sido almacenado, como puede verse en el código, en la matriz M.

```
F0=      0.666667
F1=     -0.333333
F2=           -0.5
F3=              0
F4=      0.166667
F5=     -0.333333
F6=     -0.666667
F7=     -0.666667
F8=      0.666667
```

Podemos observar, en el código, que no hemos tenido en cuenta el orden de las instrucciones por lo que los resultados obtenidos en la ejecución no son los óptimos, aunque sí los correctos, como se muestra a continuación.

```
Total:
    390 Cycle(s) executed.
    ID executed by 142 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: 68 (17.44% of all Cycles), thereof:
        LD stalls: 2 (2.94% of RAW stalls)
        Branch/Jump stalls: 0 (0.00% of RAW stalls)
        Floating point stalls: 66 (97.06% of RAW stalls)
    WAW stalls: 0 (0.00% of all Cycles)
    Structural stalls: 175 (44.87% of all Cycles)
    Control stalls: 1 (0.26% of all Cycles)
    Trap stalls: 2 (0.51% of all Cycles)
    Total: 246 Stall(s) (63.08% of all Cycles)

Conditional Branches):
    Total: 1 (0.70% of all Instructions), thereof:
        taken: 0 (0.00% of all cond. Branches)
        not taken: 1 (100.00% of all cond. Branches)

Load-/Store-Instructions:
    Total: 38 (26.76% of all Instructions), thereof:
        Loads: 29 (76.32% of Load-/Store-Instructions)
        Stores: 9 (23.68% of Load-/Store-Instructions)

Floating point stage instructions:
    Total: 100 (70.42% of all Instructions), thereof:
        Additions: 42 (42.00% of Floating point stage inst.)
        Multiplications: 57 (57.00% of Floating point stage inst.)
        Divisions: 1 (1.00% of Floating point stage inst.)

Traps:
    Traps: 1 (0.70% of all Instructions)
```

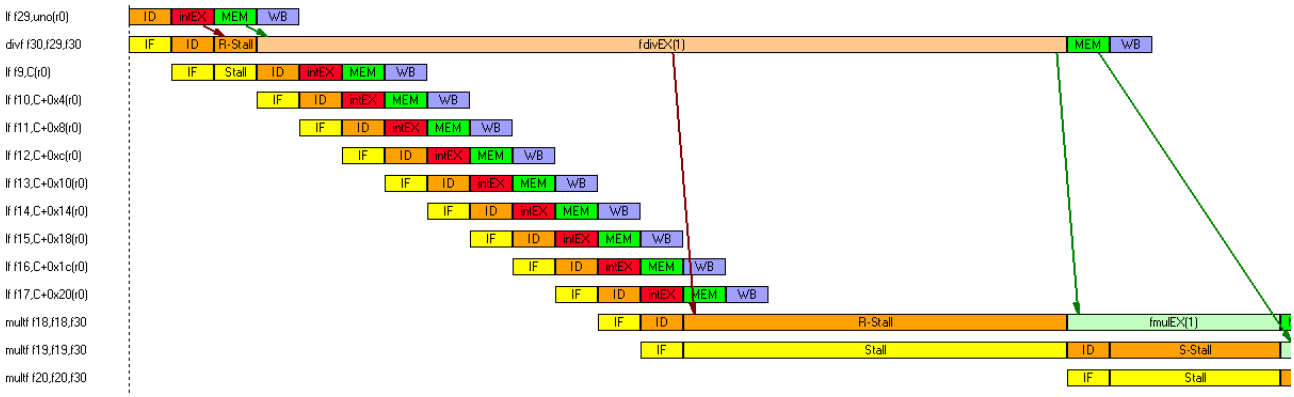
También podemos ver la no optimización en el diagrama de ciclos donde las flechas rojas indican las demoras. Ejemplo ilustrativo de ello se refleja en las siguientes imágenes.



1*Nota: en la versión no optimizada hemos decidido no tener bucles sino tener el código desenrollado, es por ello por lo que en la versión optimizada se mantiene. No obstante, se podría considerar este desenrollamiento una optimización.
2*El número de saltos tomados depende de si el determinante es cero o no lo es. En caso de no ser cero no se toma el salto, en caso de ser cero se toma el salto y finaliza el programa.

Todas las flechas rojas se deben a que las operaciones requieren de un dato resultado de una operación anterior que aún no ha finalizado (*RAW*)

La última flecha roja se debe a que la operación división usa un dato que se carga en la operación anterior, no dando tiempo suficiente para completar su carga.



La división tarda 15 ciclos por lo que, aunque haya muchas operaciones que separen a la división de la multiplicación, esta última, al necesitar el resultado de la división se queda esperando por él. Es por ello la presencia de la flecha roja larga.

Como estos dos ejemplos ilustrativos hay bastantes más, todo ello se soluciona, en la medida de lo posible, en la versión optimizada.

Hemos obtenido 390 ciclos de ejecución de los cuales 246 ciclos son de parada (*stalls*) lo que representa el 63,08% del total.
Para detalles más profundos acerca de las operaciones realizadas observar el código debidamente comentado.
Es para lograr una ejecución óptima con el menor número de demoras que desarrollamos la versión optimizada *optimizada.s*

5. Versión optimizada – *optimizada.s*

En el punto anterior hemos conseguido una versión funcional del programa la cual no está optimizada, así, a continuación, nuestro objetivo fue lograr una versión óptima donde las unidades funcionales estuviesen ocupadas el mayor tiempo posible evitando, en mayor media, los ciclos de parada o *stalls*.

Nuestra principal optimización ha sido disminuir el número de *RAW Stalls,* consiguiendo reducir el número de instrucciones que necesitan leer un dato que otra instrucción previa aún no ha generado, para así disminuir los ciclos de ejecución. Para ello hemos utilizado las siguientes estrategias:

1*Nota: en la versión no optimizada hemos decidido no tener bucles sino tener el código desenrollado, es por ello por lo que en la versión optimizada se mantiene. No obstante, se podría considerar este desenrollamiento una optimización.
2*El número de saltos tomados depende de si el determinante es cero o no lo es. En caso de no ser cero no se toma el salto, en caso de ser cero se toma el salto y finaliza el programa.

- Tras varias pruebas hemos llegado a la conclusión que si cargamos dos registros y después realizamos una operación que requiera de ambos registros se produce una espera, aumentando los *stalls,* sin embargo, si cargamos un tercer registro antes de la operación podemos observar que no se produce dicha espera y podemos aprovechar estos ciclos, que se desperdiciarían, para cargar datos que necesitásemos posteriormente si los hubiere.

```
main:
    lf f0,valor
    lf f1,valor2
    lf f2,valor3

    multf f5,f0,f1

    trap 0
```

Carga de dos valores en registros

```
main:
    lf f0,valor
    lf f1,valor2

    multf f5,f0,f1

    trap 0
```

Carga de tres valores en registros

```
Total:
    13 Cycle(s) executed.
    ID executed by 4 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: 1 (7.69% of all Cycles), thereof:
        LD stalls: 1 (100.00% 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: 0 (0.00% of all Cycles)
    Control stalls: 0 (0.00% of all Cycles)
    Trap stalls: 8 (61.54% of all Cycles)
    Total: 9 Stall(s) (69.23% of all Cycles)

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

Load-/Store-Instructions:
    Total: 2 (50.00% of all Instructions), thereof:
        Loads: 2 (100.00% of Load-/Store-Instructions)
        Stores: 0 (0.00% of Load-/Store-Instructions)

Floating point stage instructions:
    Total: 1 (25.00% of all Instructions), thereof:
        Additions: 0 (0.00% of Floating point stage inst.)
        Multiplications: 1 (100.00% of Floating point stage inst.)
        Divisions: 0 (0.00% of Floating point stage inst.)

Traps:
    Traps: 1 (25.00% of all Instructions)
```

Carga de dos valores en registros

```
Total:
    13 Cycle(s) executed.
    ID executed by 5 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: 0 (0.00% of all Cycles)
    Control stalls: 0 (0.00% of all Cycles)
    Trap stalls: 7 (53.85% of all Cycles)
    Total: 7 Stall(s) (53.85% of all Cycles)

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

Load-/Store-Instructions:
    Total: 3 (60.00% of all Instructions), thereof:
        Loads: 3 (100.00% of Load-/Store-Instructions)
        Stores: 0 (0.00% of Load-/Store-Instructions)

Floating point stage instructions:
    Total: 1 (20.00% of all Instructions), thereof:
        Additions: 0 (0.00% of Floating point stage inst.)
        Multiplications: 1 (100.00% of Floating point stage inst.)
        Divisions: 0 (0.00% of Floating point stage inst.)

Traps:
    Traps: 1 (20.00% of all Instructions)
```

Carga de tres valores en registros

1*Nota: en la versión no optimizada hemos decidido no tener bucles sino tener el código desenrollado, es por ello por lo que en la versión optimizada se mantiene. No obstante, se podría considerar este desenrollamiento una optimización.
2*El número de saltos tomados depende de si el determinante es cero o no lo es. En caso de no ser cero no se toma el salto, en caso de ser cero se toma el salto y finaliza el programa.

- Sucesión *multf-lf-lf (multiplicación-carga-carga)*
- Sucesión *multf-lf-addf (multiplicación-carga-suma)*
- Sucesión *multf-addf (multiplicación-suma)*

Dado que nuestra operación mayoritaria es la multiplicación hemos probado diversas sucesiones de operaciones para encontrar la forma ideal de cargar, sumar o ambas, al mismo tiempo que realizamos la multiplicación, con el fin de meter varias operaciones en un mismo ciclo y reducir las demoras.
Estas sucesiones las hemos obtenido teniendo en cuenta las dependencias de datos y recursos entre operaciones.

Otra de nuestras optimizaciones ha sido reducir el número de *Structural Stalls,* consiguiendo disminuir el número de instrucciones que tratan de hacer uso de un único recurso (unidad funcional). Lo hemos conseguido separando por al menos un mínimo de ciclos dos instrucciones que usen la misma unidad funcional, por ejemplo, separando con 4 ciclos dos instrucciones de multiplicación ya que estas utilizan la misma unidad funcional para realizar la operación. No obstante, no siempre es posible ya que según tenemos estructurado el código, si intentamos reducir las *S-Stalls,* aumenta bien el número de ciclos, bien las *R-Stalls,* o bien se pueden llegar a sobreescribir datos debido a que se pisen los registros. Por eso las sucesiones anteriores son las óptimas para nuestro código.

```
Total:
    321 Cycle(s) executed.
    ID executed by 142 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: 43 (13.40% of all Cycles), thereof:
        LD stalls: 1 (2.32% of RAW stalls)
        Branch/Jump stalls: 0 (0.00% of RAW stalls)
        Floating point stalls: 42 (97.67% of RAW stalls)
    WAW stalls: 0 (0.00% of all Cycles)
    Structural stalls: 131 (40.81% of all Cycles)
    Control stalls: 1 (0.31% of all Cycles)
    Trap stalls: 2 (0.62% of all Cycles)
    Total: 177 Stall(s) (55.14% of all Cycles)

Conditional Branches):
    Total: 1 (0.70% of all Instructions), thereof:
        taken: 0 (0.00% of all cond. Branches)
        not taken: 1 (100.00% of all cond. Branches)

Load-/Store-Instructions:
    Total: 38 (26.76% of all Instructions), thereof:
        Loads: 29 (76.32% of Load-/Store-Instructions)
        Stores: 9 (23.68% of Load-/Store-Instructions)

Floating point stage instructions:
    Total: 100 (70.42% of all Instructions), thereof:
        Additions: 42 (42.00% of Floating point stage inst.)
        Multiplications: 57 (57.00% of Floating point stage inst.)
        Divisions: 1 (1.00% of Floating point stage inst.)

Traps:
    Traps: 1 (0.70% of all Instructions)
```

1*Nota: en la versión no optimizada hemos decidido no tener bucles sino tener el código desenrollado, es por ello por lo que en la versión optimizada se mantiene. No obstante, se podría considerar este desenrollamiento una optimización.
2*El número de saltos tomados depende de si el determinante es cero o no lo es. En caso de no ser cero no se toma el salto, en caso de ser cero se toma el salto y finaliza el programa.

Conclusión

Comparando los resultados de ambas versiones, obtenemos lo siguiente:

ESTADÍSTICAS	No optimizada	Optimizada
TOTAL		
Nº Ciclos	390	321
Nº Instrucciones ejecutadas (ID)	142	142
STALLS		
RAW	68	43
LD	2	1
Branch/Jump	0	0
Floating Point	66	42
WAW	0	0
Structural	175	131
Control	1	1
Trap	2	2
Total	246	177
CONDITIONAL BRANCHES		
Total	1	1
Tomados	0*	0*
No Tomados	1*	1*
INSTRUCCIONES LOAD/STORE		
Total	38	38
Loads	29	29
Stores	9	9
INSTRUCCIONES EN PUNTO FLOTANTE		
Total	100	100
Sumas	42	42
Multiplicaciones	57	57
Divisiones	1	1
TRAPS		
Traps	1	1

1*Nota: en la versión no optimizada hemos decidido no tener bucles sino tener el código desenrollado, es por ello por lo que en la versión optimizada se mantiene. No obstante, se podría considerar este desenrollamiento una optimización.
2*El número de saltos tomados depende de si el determinante es cero o no lo es. En caso de no ser cero no se toma el salto, en caso de ser cero se toma el salto y finaliza el programa.