

# **Implementación y optimización de un algoritmo en ensamblador DLX**

JAIME GÓMEZ GARCÍA  
DIEGO BERMÚDEZ PÉREZ

# Versión sin optimizar

MF(a1,a2) y MF(a2,a4)

```
main:
; cargamos MF(a1,a2)

lf  f1,a1
lf  f2,a2
eqf f2,f0 ; Si dividimos entre 0 hace un salto al final del programa
bfpt fin
divf f2,f1,f2
lf  f3,a2
multf f4,f1,f3

; cargamos MF(a3,a4)

lf  f5,a3
lf  f6,a4
eqf f6,f0 ; Si dividimos entre 0 hace un salto al final del programa
bfpt fin
divf f6,f5,f6
lf  f7,a4
multf f8,f5,f7
```

Podemos ver las cargas y cálculos realizados para MF(a1,a2) y MF(a3,a4). El resultado de los registros sería:

## Producto de Kronecker

```
; producto de Kronecker de f9 a f24
; primera fila
multf f9,f1,f5
multf f10,f1,f6
multf f11,f2,f5
multf f12,f2,f6
; segunda fila
multf f13,f1,f7
multf f14,f1,f8
multf f15,f2,f7
multf f16,f2,f8
; tercera fila
multf f17,f3,f5
multf f18,f3,f6
multf f19,f4,f5
multf f20,f4,f6
; cuarta fila
multf f21,f3,f7
multf f22,f3,f8
multf f23,f4,f7
multf f24,f4,f8
```

A continuación calculamos el producto de Kronecker, almacenando en los registros de f9 a f23.

$$(a_1 + a_4) / |MF(a_2, a_3)|$$

```
;a1+a4
addf    f25,f1,f7

;calculamos MF(a2,a3) a2 esta en f3 y a3 esta en f5

eqf f5,f0    ;Si dividimos entre 0 hace un salto al final del programa
bfpt    fin
divf    f26,f3,f5
multf    f27,f3,f5

;calculamos el determinante de MF(a2,a3)

multf    f28,f3,f27
multf    f29,f5,f26
subf    f30,f28,f29

;calculamos (a1+a4)/|MF(a2,a3)|

eqf f30,f0    ;Si dividimos entre 0 hace un salto al final del programa
bfpt    fin
divf    f30,f25,f30
```

Primero sumamos  $a_1$  y  $a_4$  y lo guardamos en el registro  $f_{25}$ , después calculamos  $MF(a_2, a_3)$  y su determinante, finalmente hacemos la división.

## Calculamos y almacenamos M

```
;calculamos M de f9 a f24
multf    f9,f9,f30
multf    f10,f10,f30
multf    f11,f11,f30
multf    f12,f12,f30
multf    f13,f13,f30
multf    f14,f14,f30
multf    f15,f15,f30
multf    f16,f16,f30
multf    f17,f17,f30
multf    f18,f18,f30
multf    f19,f19,f30
multf    f20,f20,f30
multf    f21,f21,f30
multf    f22,f22,f30
multf    f23,f23,f30
multf    f24,f24,f30

;Almacenamos M

sf    M,f9
sf    M+4,f10
sf    M+8,f11
sf    M+12,f12
sf    M+16,f13
sf    M+20,f14
sf    M+24,f15
sf    M+28,f16
sf    M+32,f17
sf    M+36,f18
sf    M+40,f19
sf    M+44,f20
sf    M+48,f21
sf    M+52,f22
sf    M+56,f23
sf    M+60,f24
```

La matriz M es calculada multiplicando la matriz de Kronecker por el registro f30.

## Calculamos y almacenamos VH, HM

```
;Calculamos VM  
  
multf    f9,f9,f13  
multf    f10,f10,f14  
multf    f11,f11,f15  
multf    f12,f12,f16
```

```
;Calculamos HM  
  
multf    f17,f17,f21  
multf    f18,f18,f22  
multf    f19,f19,f23  
multf    f20,f20,f24
```

```
;Almacenamos VM
```

```
sf    VM,f9  
sf    VM+4,f10  
sf    VM+8,f11  
sf    VM+12,f12
```

```
;Almacenamos HM
```

```
sf    HM,f17  
sf    HM+4,f18  
sf    HM+8,f19  
sf    HM+12,f20
```

El cálculo de VH se realiza multiplicando los valores de la primera y segunda fila de M y el cálculo de VM multiplicando los de la tercera y cuarta fila.

## Calculamos y almacenamos el check

```
;Calculamos check lo guardamos en f21  
  
multf    f21,f21,f0  
addf     f21,f21,f9  
addf     f21,f21,f10  
addf     f21,f21,f11  
addf     f21,f21,f12  
addf     f21,f21,f17  
addf     f21,f21,f18  
addf     f21,f21,f19  
addf     f21,f21,f20
```

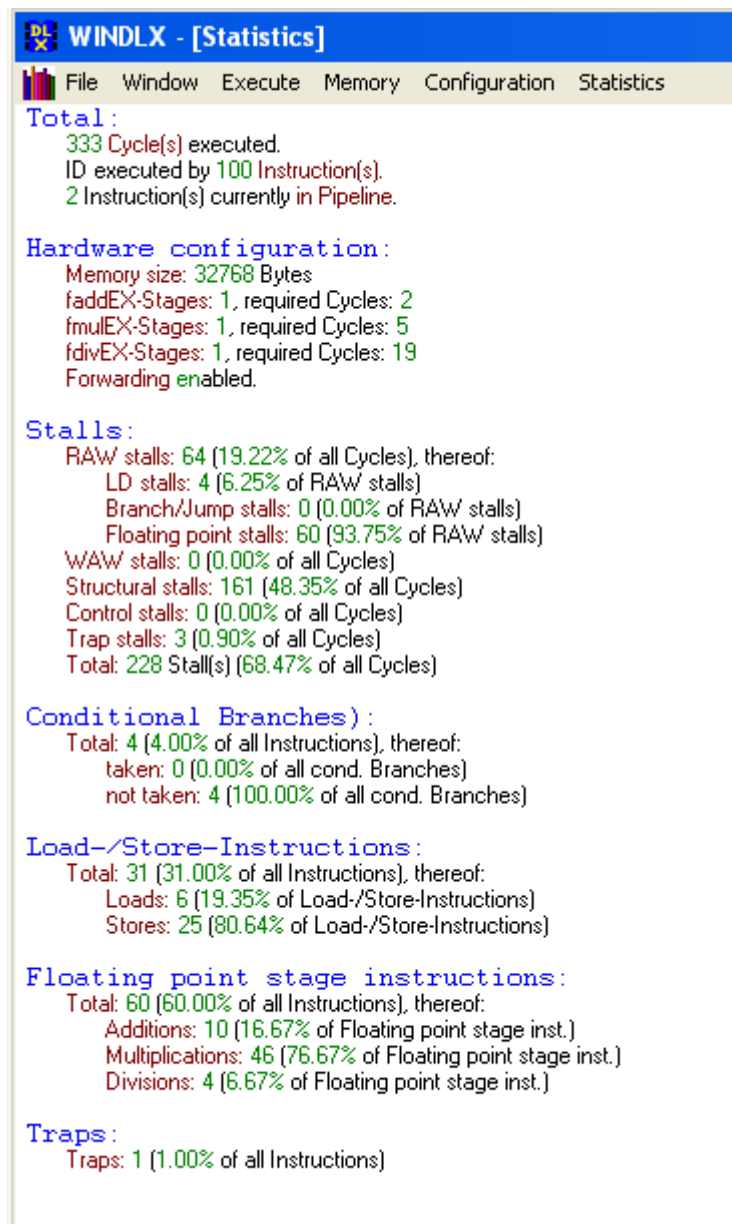
Por último cargamos el check en el registro 21.

## Resultados obtenidos de la versión sin optimizar

WINDLX - [Register]				
PC=	0x00000294	R31=	0x00000000	D28= 3.6
IMAR=	0x00000290	F0=	0	
IR=	0x00000000	F1=	1.1	
A=	0x00000000	F2=	0.5	
AHI=	0x00000000	F3=	2.2	
B=	0x00000000	F4=	2.42	
BHI=	0x00000000	F5=	3.3	
ETA=	0x00000000	F6=	0.75	
ALU=	0x00000000	F7=	4.4	
ALUHI=	0x00000000	F8=	14.52	
FFSR=	0x00000000	F9=	2.8021	
DMAR=	0x00001070	F10=	2.10157	
SDR=	0x42450fa6	F11=	0.578946	
SDRHI=	0x00000000	F12=	0.434209	
LDR=	0x00000000	F13=	1.93291	
LDRHI=	0x00000000	F14=	6.37859	
R0=	0x00000000	F15=	0.878594	
R1=	0x00000000	F16=	2.89936	
R2=	0x00000000	F17=	11.2084	
R3=	0x00000000	F18=	8.40629	
R4=	0x00000000	F19=	13.5622	
R5=	0x00000000	F20=	10.1716	
R6=	0x00000000	F21=	49.2653	
R7=	0x00000000	F22=	12.7572	
R8=	0x00000000	F23=	4.2524	
R9=	0x00000000	F24=	14.0329	
R10=	0x00000000	F25=	5.5	
R11=	0x00000000	F26=	0.666667	
R12=	0x00000000	F27=	7.26	
R13=	0x00000000	F28=	15.972	
R14=	0x00000000	F29=	2.2	
R15=	0x00000000	F30=	0.399361	
R16=	0x00000000	F31=	0	
R17=	0x00000000	D0=	0.0140625	
R18=	0x00000000	D2=	3.6	
R19=	0x00000000	D4=	76.8	
R20=	0x00000000	D6=	921.6	
R21=	0x00000000	D8=	19.3343	
R22=	0x00000000	D10=	7.7095e-05	
R23=	0x00000000	D12=	1.46326	
R24=	0x00000000	D14=	0.00206544	
R25=	0x00000000	D16=	1.2671e+06	
R26=	0x00000000	D18=	6.5521e+06	
R27=	0x00000000	D20=	1.8091e+11	
R28=	0x00000000	D22=	770.454	
R29=	0x00000000	D24=	4096	
R30=	0x00000000	D26=	49807.4	

Como podemos observar en la captura de pantalla el valor cargado en el registro f21 es 49.2653 siendo este el resultado correcto de la operación check.

## Rendimiento del programa



The screenshot shows the 'WINDLX - [Statistics]' window with a menu bar (File, Window, Execute, Memory, Configuration, Statistics). The statistics are as follows:

```
Total:
  333 Cycle(s) executed.
  ID executed by 100 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: 64 (19.22% of all Cycles), thereof:
    LD stalls: 4 (6.25% of RAW stalls)
    Branch/Jump stalls: 0 (0.00% of RAW stalls)
    Floating point stalls: 60 (93.75% of RAW stalls)
  WAW stalls: 0 (0.00% of all Cycles)
  Structural stalls: 161 (48.35% of all Cycles)
  Control stalls: 0 (0.00% of all Cycles)
  Trap stalls: 3 (0.90% of all Cycles)
  Total: 228 Stall(s) (68.47% of all Cycles)

Conditional Branches):
  Total: 4 (4.00% 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: 31 (31.00% of all Instructions), thereof:
    Loads: 6 (19.35% of Load-/Store-Instructions)
    Stores: 25 (80.64% of Load-/Store-Instructions)

Floating point stage instructions:
  Total: 60 (60.00% of all Instructions), thereof:
    Additions: 10 (16.67% of Floating point stage inst.)
    Multiplications: 46 (76.67% of Floating point stage inst.)
    Divisions: 4 (6.67% of Floating point stage inst.)

Traps:
  Traps: 1 (1.00% of all Instructions)
```

Los datos más relevantes de la imagen anterior son:

- El número total de ciclos es de 333.
- La cantidad de RAW stalls es de 64.
- El 68% de todos los ciclos son paradas, siendo un total de 228.
- La carga y almacenamiento son un total de 31 ciclos

# Versión optimizada

A partir de los resultados obtenidos en la versión anterior pudimos realizar una serie de cambios para poder mejorar el rendimiento del programa.

## Primera mejora:

Intercalamos el almacenamiento de los resultados de tal forma que aprovechamos al máximo los recursos, esto lo hacemos intentando no realizar dos operaciones iguales seguidas, para ello intercalamos las multiplicaciones con el almacenamiento en memoria, pero sin que haya dependencia entre las operaciones. Con esto conseguimos ahorrarnos 21 ciclos.

```
;calculamos M de f9 a f24

multf  f9,f9,f30
multf  f10,f10,f30
sf  M,f9
multf  f11,f11,f30
sf  M+4,f10
multf  f12,f12,f30
sf  M+8,f11
multf  f13,f13,f30
sf  M+12,f12
multf  f14,f14,f30
sf  M+16,f13
multf  f15,f15,f30
sf  M+20,f14
multf  f16,f16,f30
sf  M+24,f15
multf  f17,f17,f30
sf  M+28,f16
multf  f18,f18,f30
sf  M+32,f17
multf  f19,f19,f30
sf  M+36,f18
multf  f20,f20,f30
sf  M+40,f19
multf  f21,f21,f30
sf  M+44,f20
multf  f22,f22,f30
sf  M+48,f21
multf  f23,f23,f30
sf  M+52,f22
multf  f24,f24,f30
sf  M+56,f23
```

```
;Calculamos VM

multf  f9,f9,f13
sf  M+60,f24
multf  f10,f10,f14
sf  VM,f9
multf  f11,f11,f15
sf  VM+4,f10
multf  f12,f12,f16
sf  VM+8,f11

;Calculamos HM

multf  f17,f17,f21
sf  VM+12,f12
multf  f18,f18,f22
sf  HM,f17
multf  f19,f19,f23
sf  HM+4,f18
multf  f20,f20,f24
sf  HM+8,f19
```



## Segunda mejora:

Nos dimos cuenta que en vez de tener que esperar por la división de  $a_1/a_2$  podíamos ir calculando el primer cuadrante del producto de Kronecker, ya que para este primer cuadrante no necesitábamos dicha división, para ello también ahora calculamos antes  $MF(a_3, a_4)$ . Con esta mejora conseguimos reducir el cálculo total en 8 ciclos.

$\otimes$  el producto de *Kronecker*

$$\circ \quad A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}$$

## Tercera mejora:

Otra mejora que se nos ocurrió fue reordenar el código para que al finalizar cualquier división comience la siguiente, dado que la división es de las operaciones más costosas que se realizan. Con esta mejora conseguimos reducir el cálculo total en 7 ciclos.

## Cuarta mejora:

Para finalizar, la última mejora que realizamos fue acabar con la dependencia que existía para calcular la matriz  $M$ , esta dependencia era la necesidad de conocer el resultado de la división de  $a_1 + a_4 / |MF(a_2, a_3)|$ . Para ello reordenamos el código de tal forma que mientras se realiza esta división se calcula la cuarta fila del producto de Kronecker. Con esta mejora conseguimos reducir el cálculo total en 20 ciclos.

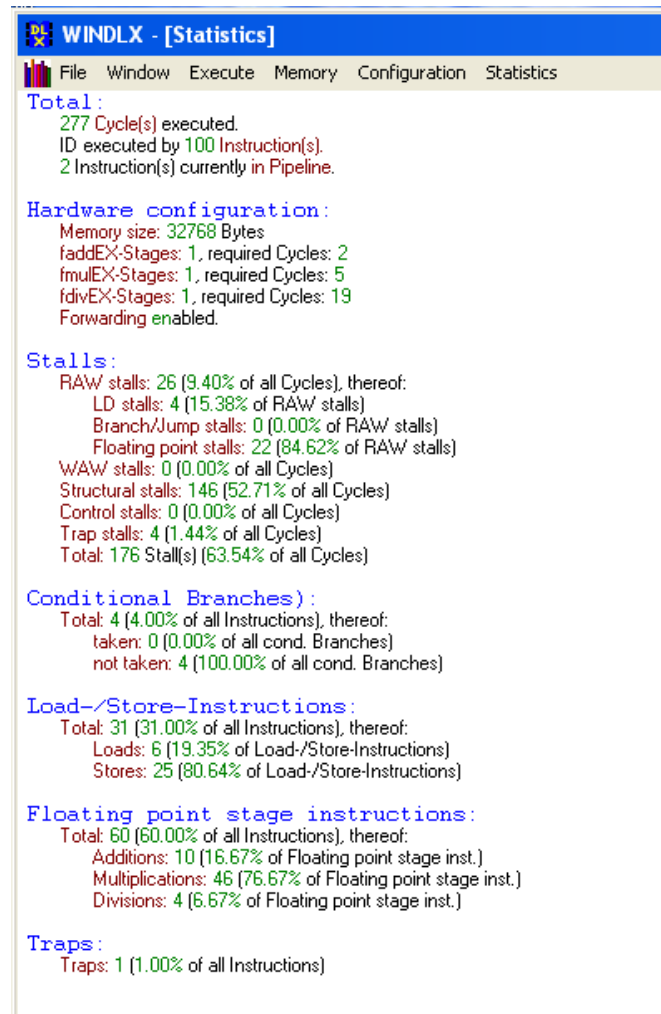
## Resultados obtenidos tras realizar las mejoras:

WINDLX - [Register]				
PC=	0x00000294	R31=	0x00000000	D28= 3.6
IMAR=	0x00000290	F0=	0	
IR=	0x00000000	F1=	1.1	
A=	0x00000000	F2=	0.5	
AHI=	0x00000000	F3=	2.2	
B=	0x00000000	F4=	2.42	
BHI=	0x00000000	F5=	3.3	
BTA=	0x00000000	F6=	0.75	
ALU=	0x00000000	F7=	4.4	
ALUHI=	0x00000000	F8=	14.52	
FPSR=	0x00000000	F9=	2.8021	
DMAR=	0x00001070	F10=	2.10157	
SDR=	0x42450fa6	F11=	0.578946	
SDRHI=	0x00000000	F12=	0.434209	
LDR=	0x00000000	F13=	1.93291	
LDRHI=	0x00000000	F14=	6.37859	
R0=	0x00000000	F15=	0.878594	
R1=	0x00000000	F16=	2.89936	
R2=	0x00000000	F17=	11.2084	
R3=	0x00000000	F18=	8.40629	
R4=	0x00000000	F19=	13.5622	
R5=	0x00000000	F20=	10.1716	
R6=	0x00000000	F21=	49.2653	
R7=	0x00000000	F22=	12.7572	
R8=	0x00000000	F23=	4.2524	
R9=	0x00000000	F24=	14.0329	
R10=	0x00000000	F25=	5.5	
R11=	0x00000000	F26=	0.666667	
R12=	0x00000000	F27=	7.26	
R13=	0x00000000	F28=	15.972	
R14=	0x00000000	F29=	2.2	
R15=	0x00000000	F30=	0.399361	
R16=	0x00000000	F31=	0	
R17=	0x00000000	D0=	0.0140625	
R18=	0x00000000	D2=	3.6	
R19=	0x00000000	D4=	76.8	
R20=	0x00000000	D6=	921.6	
R21=	0x00000000	D8=	19.3343	
R22=	0x00000000	D10=	7.7095e-05	
R23=	0x00000000	D12=	1.46326	
R24=	0x00000000	D14=	0.00206544	
R25=	0x00000000	D16=	1.2671e+06	
R26=	0x00000000	D18=	6.5521e+06	
R27=	0x00000000	D20=	1.8091e+11	
R28=	0x00000000	D22=	770.454	
R29=	0x00000000	D24=	4096	
R30=	0x00000000	D26=	49807.4	

Como podemos observar en la captura de pantalla el valor cargado en el registro f21 es 49.2653 siendo este el resultado correcto de la operación check.

Es el mismo valor que la versión sin optimizar, por tanto concluimos que al realizar la optimizaciones no ha ocurrido ningún error de cálculo.

## Rendimiento del programa



```
WINDLX - [Statistics]
File Window Execute Memory Configuration Statistics
Total:
277 Cycle(s) executed.
100 Instruction(s) executed.
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: 26 (9.40% of all Cycles), thereof:
  LD stalls: 4 (15.38% of RAW stalls)
  Branch/Jump stalls: 0 (0.00% of RAW stalls)
  Floating point stalls: 22 (84.62% of RAW stalls)
WAW stalls: 0 (0.00% of all Cycles)
Structural stalls: 146 (52.71% of all Cycles)
Control stalls: 0 (0.00% of all Cycles)
Trap stalls: 4 (1.44% of all Cycles)
Total: 176 Stall(s) (63.54% of all Cycles)

Conditional Branches:
Total: 4 (4.00% 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: 31 (31.00% of all Instructions), thereof:
  Loads: 6 (19.35% of Load-/Store-Instructions)
  Stores: 25 (80.64% of Load-/Store-Instructions)

Floating point stage instructions:
Total: 60 (60.00% of all Instructions), thereof:
  Additions: 10 (16.67% of Floating point stage inst.)
  Multiplications: 46 (76.67% of Floating point stage inst.)
  Divisions: 4 (6.67% of Floating point stage inst.)

Traps:
Traps: 1 (1.00% of all Instructions)
```

Como podemos ver en en las estadísticas:

- El número total de ciclos es de 277
- La cantidad de RAW stalls es de 26
- El 63% de todos los ciclos son paradas, siendo un total de 176.

Podemos ver que hemos reducido en 38 la cantidad de RAW stalls, el total de ciclos de parada también se ha reducido, en concreto 52 pero el porcentaje sigue siendo similar. Claramente el número de ciclos de carga y almacenamiento no ha variado en absoluto, ya que no hemos realizado ningún cambio.

En conclusión con esta serie de optimizaciones hemos conseguido pasar de 333 ciclos a 277 siendo 56 el total de ciclos que nos hemos ahorrado.