# Práctica 1:

# Procesadores Segmentados con WinDLX

**5º curso Ing. Informática**
**Arquitectura de Computadores I**
**José Antonio Guerrero Avilés**
**75485683M**
**cany@correo.ugr.es**

**1. Ejecute el programa incluido en pr1_1.s con el simulador WinDLX y determine el tiempo de procesamiento, el número de ciclos en los que el cauce ha estado detenido (stall) y las causas de las detenciones. Determine el número de instrucciones por ciclo que ejecuta el procesador y calcule la ganancia de velocidad con respecto al procesador sin segmentar de referencia ssrDLX. (Nota: el número de ciclos en las unidades funcionales ha de ser el mismo que en el procesador ssrDLX).**

Cargamos el programa en WINDLX pulsando F3 o seleccionando File → Load Code or Data… y luego pulsamos F5 para ejecutarlo o seleccionamos Execute → Run.

Una vez ejecutado podemos responder a la pregunta viendo la ventana Statistics.

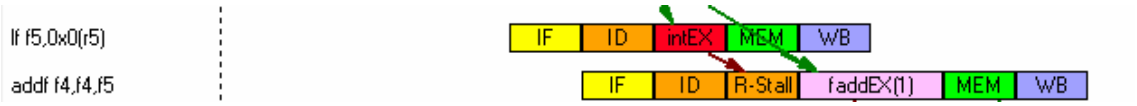
- Tiempo de procesamiento: 168 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 39 ciclos (23.21%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

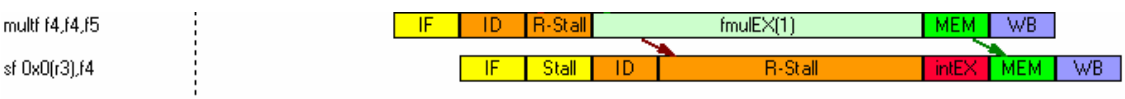



Como vemos en este ejemplo, en la primera instrucción (*lf f5, 0x0(r5)*) se carga un valor en el registro f5 pero no se escribe hasta la etapa WB, por lo que la segunda instrucción (*addf f4,f4,f5*) debe de esperar a ejecutar su etapa de ejecución (*faddEX(1)*) hasta que la primera instrucción ejecute la etapa WB. Como estoy utilizando la opción Enable Forwarding, la etapa de escritura de la primera instrucción (*WB*) se puede realizar en el mismo ciclo que la etapa de ejecución (*faddEX(1)*) .De lo contrario, la segunda instrucción debería esperar un ciclo más (hasta que terminara la etapa WB de la primera instrucción) para poder ejecutar su etapa de ejecución (*faddEX(1)*).

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 25

Un ejemplo de este tipo de detección es el de esta imagen:




Se produce debido a que una instrucción intenta leer un registro antes de que una instrucción anterior termine de realizar una operación en coma flotante con él en la etapa de ejecución (EX), por lo que se espera tantos ciclos vacíos (stall) como diferencia haya entre los ciclos que ocupa su etapa ID y los que ocupa la etapa de ejecución de la

anterior instrucción para que, al terminar la primera instrucción su etapa EX, la siguiente instrucción pueda utilizar el registro actualizado.
En este caso, la instrucción *sf 0x0(r3), f4* depende de la ejecución de la operación que se realiza en la instrucción *multf f4, f4, f5*, es decir, *sf 0x0(r3), f4* debe esperar 3 ciclos ya que necesita que el registro *f4* de la instrucción *multf f4, f4, f5* esté actualizado.

> *Interrupciones:* 4

Podemos apreciarlo en la siguiente imagen:


Lo que ocurre es que al captar la instrucción de interrupción (etapa IF), espera a que terminen todas las instrucciones con etapas en curso para realizar una llamada al sistema en su etapa ID.

## Cálculo del Número de Instrucciones por Ciclo.

**IPC** = NumInstruccionesEjecutadas / NumCiclos = 125 / 168 = **0,744 instr/ciclo**

## Ganancia de velocidad con respecto al procesador no segmentado ssrDLX.

Hay que tener en cuenta varias consideraciones:

- IF, intEX, y MEM duran un ciclo de reloj cada una.
- ID y WB duran un 80% del tiempo de ciclo de reloj.
- faddEX, fmulEX, y fdivEX duran el mismo número de ciclos que en el caso del procesador segmentado.

Para calcular el tiempo de procesamiento del procesador sin segmentar (ssrDLX) hay que estimar el tiempo que tarda cada una de las instrucciones.
Al tratarse de un procesador no segmentado, cada instrucción tardaría únicamente el tiempo asociado al procesamiento de cada fase, y únicamente se tienen en cuenta las fases que necesita esa instrucción.
Hay que empezar calculando el tiempo de ciclo para cada uno de los tipos de instrucciones que aparecen en el programa:

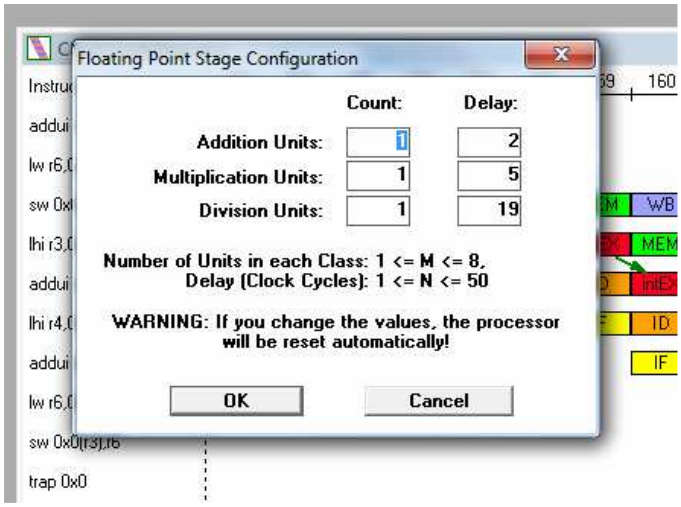
- Instrucciones de carga: lhi, lf, lw.

Las instrucciones lf y lw pasan por todas las etapas (IF-ID-intEX-MEM-WB), que duran 1 ciclo cada una, excepto las etapas ID y WB que duran el 80% de las etapas IF, MEM o intEX. Por tanto, **el tiempo de las instrucciones lf y lw en el procesador ssrDLX es** 1 + 0.8 + 1 + 1 + 0.8 = **4.6 ciclos.**

La instrucción lhi no pasa por la etapa ID (no accede al banco de registros porque es un dato inmediato) ni por la etapa MEM (porque la dirección es 0x00). Por tanto, **el tiempo de la instrucción lhi en el procesador ssrDLX es** 1 + 1 + 0.8 **= 2.8 ciclos.**

- Instrucciones aritméticas: addui, addf, multf.

Estas instrucciones no parían por todas las etapas porque es un modelo registro-registro y por tanto, no se ejecutarían las operaciones correspondientes a la etapa MEM.
También existe diferencia en la etapa EX, ya que la instrucción addui duraría 1 ciclo porque realiza una suma de enteros y las instrucciones addf y multf tendrían una duración de 2 y 5 ciclos respectivamente debido a la configuración del simulador:




Con todo esto y teniendo en cuenta que etapas ID y WB duran el 80% de las etapas IF, MEM o intEX tenemos que el tiempo de procesamientos es de:

**addui = 3.6 ciclos**
**addf = 4.6 ciclos**
**multf = 7.6 ciclos**

- Instrucciones de almacenamiento: sf, sw.

Estas instrucciones pasan por todas las etapas excepto por WB, ya que al ser una instrucción de acceso a memoria no se escribe en ningún registro (que es lo que se hace en la etapa WB).

Teniendo en cuenta que las etapas ID y WB que el 80% de las etapas IF, MEM o intEX y que la etapa WB no se lleva a cabo, tenemos que el **tiempo de procesamiento de las instrucciones sf y sw en el procesador ssrDLX es** 1+ 0.8 + 1 + 1 = **3.8 ciclos.**

- Instrucciones de interrupción: trap#0.

En esta instrucción se ejecutan solo las etapas IF-ID, por tanto, como ID dura el 80% de IF y esta dura 1 ciclo, tenemos que **el tiempo de procesamiento de la instrucción trap#0 es** 1 + 0.8 = **1.8 ciclos.**

Una vez que sabemos el tiempo que tarda en procesarse cada instrucción calculamos el tiempo total de ejecución del programa haciendo entre el número de instrucciones de cada tipo por el tiempo que tardan. Así que tenemos:

T= LHI + LF +LW + ADDUI + ADDF + MULTF + SF + SW + TRAP

**T** = (38*2.8) + (20*4.6) + (4*4.6) + (38*3.6) + (5*4.6) + (5*7.6) + (10*3.8) + (4*3.8) + (1*1.8) = 106.4 + 92 + 18.4 + 136.8 + 23 + 38 +38 + 15.2 +1.8 = **469.6 ciclos**

Con todos los datos, la ganancia será:

**Ganancia** = 469.6 / 168 = **2.795**

Y la eficiencia (ganacia/num etapas):

**Eficiencia =** 2.795 / 5 **= 0.559**

Recordar que todos estos datos son así, debido a la configuración que estamos usando en el simulador, en este caso, el camino by pass (está marcada la opción Enable forwarding en el menú Configuration)

**2. Teniendo en cuenta las causas de las detenciones en la ejecución del programa incluido en pr1_1.s, y las posibles ineficiencias en el uso de los recursos del procesador, optimice las prestaciones obtenidas:**

**a) Modificando manualmente pr1_1.s mediante transformaciones del tipo de movimiento de instrucciones, renombrado de registros, etc**

En la tabla siguiente podemos ver en la columna de la izquierda el código original y en la columna de la derecha (marcado en rojo) las modificaciones realizadas usando movimiento de instrucciones, renombrado de registros, etc. (archivo 2a.s)

```
        .data

.global _a
        .align 4
_a:
        .float 0.000000000000
.global _b
        .align 4
_b:
        .float 1.000000000000
.global _c
        .align 4
_c:
        .float 1.000000000000
.global _d
        .align 4
_d:
        .float -1.000000000000


.global _n
_n:     .space 8
.global _m
_m:     .space 8
.global _l
_l:     .space 8
.global _k
_k:     .space 8
.global _j
_j:     .space 8
.global _i
_i:     .space 8
.global _h
_h:     .space 8
.global _g
_g:     .space 8
.global _f
_f:     .space 8
.global _e
_e:     .space 8

        .align 4

        .text

.global _main

_main:
        lhi r3,(_e>>16)&0xffff
        addui r3,r3,(_e&0xffff)
```

```
        .data

.global _a
        .align 4
_a:
        .float 0.000000000000
.global _b
        .align 4
_b:
        .float 1.000000000000
.global _c
        .align 4
_c:
        .float 1.000000000000
.global _d
        .align 4
_d:
        .float -1.000000000000


.global _n
_n:     .space 8
.global _m
_m:     .space 8
.global _l
_l:     .space 8
.global _k
_k:     .space 8
.global _j
_j:     .space 8
.global _i
_i:     .space 8
.global _h
_h:     .space 8
.global _g
_g:     .space 8
.global _f
_f:     .space 8
.global _e
_e:     .space 8

        .align 4

        .text

        .global _main

_main:
        lhi r3,(_e>>16)&0xffff
        addui r3,r3,(_e&0xffff)
```

```
lhi r4,(_a>>16)&0xffff
addui r4,r4,(_a&0xffff)
lhi r5,(_b>>16)&0xffff
addui r5,r5,(_b&0xffff)
lf f4,0(r4)
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4
lhi r3,(_f>>16)&0xffff
addui r3,r3,(_f&0xffff)
lhi r4,(_a>>16)&0xffff
addui r4,r4,(_a&0xffff)
lhi r5,(_b>>16)&0xffff
addui r5,r5,(_b&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4
lhi r3,(_g>>16)&0xffff
addui r3,r3,(_g&0xffff)
lhi r4,(_c>>16)&0xffff
addui r4,r4,(_c&0xffff)
lhi r5,(_d>>16)&0xffff
addui r5,r5,(_d&0xffff)
lf f4,0(r4)
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4
lhi r3,(_h>>16)&0xffff
addui r3,r3,(_h&0xffff)
lhi r4,(_c>>16)&0xffff
addui r4,r4,(_c&0xffff)
lhi r5,(_d>>16)&0xffff
addui r5,r5,(_d&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4
lhi r3,(_i>>16)&0xffff
addui r3,r3,(_i&0xffff)
lhi r4,(_e>>16)&0xffff
addui r4,r4,(_e&0xffff)
lhi r5,(_g>>16)&0xffff
addui r5,r5,(_g&0xffff)
lf f4,0(r4)
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4
lhi r3,(_j>>16)&0xffff
addui r3,r3,(_j&0xffff)
```

```
lhi r4,(_a>>16)&0xffff
addui r4,r4,(_a&0xffff)
lhi r5,(_b>>16)&0xffff
addui r5,r5,(_b&0xffff)
lf f5,0(r5)
lf f4,0(r4)
lhi r7,(_f>>16)&0xffff
addui r7,r7,(_f&0xffff)
addf f4,f4,f5
lhi r4,(_a>>16)&
sf 0(r3),f4
addui r4,r4,(_a&0xffff)
lhi r5,(_b>>16)&0xffff
addui r5,r5,(_b&0xffff)
lf f4,0(r4)
lf f5,0(r5)
lhi r3,(_g>>16)&0xffff
addui r3,r3,(_g&0xffff)
multf f4,f4,f5
lhi r4,(_c>>16)&0xffff
addui r4,r4,(_c&0xffff)
lhi r5,(_d>>16)&0xffff
addui r5,r5,(_d&0xffff)
sf 0(r7),f4
lf f4,0(r4)
lf f5,0(r5)
lhi r7,(_h>>16)&0xffff
addui r7,r7,(_h&0xffff)
addf f4,f4,f5
lhi r4,(_c>>16)&0xffff
sf 0(r3),f4
addui r4,r4,(_c&0xffff)
lhi r5,(_d>>16)&0xffff
addui r5,r5,(_d&0xffff)
lf f4,0(r4)
lf f5,0(r5)
lhi r3,(_i>>16)&0xffff
addui r3,r3,(_i&0xffff)
multf f4,f4,f5
lhi r4,(_e>>16)&0xffff
addui r4,r4,(_e&0xffff)
lhi r5,(_g>>16)&0xffff
addui r5,r5,(_g&0xffff)
sf 0(r7),f4
lf f4,0(r4)
lf f5,0(r5)
lhi r7,(_j>>16)&0xffff
addui r7,r7,(_j&0xffff)
addf f4,f4,f5
lhi r4,(_e>>16)&0xffff
```

```
lhi r4,(_e>>16)&0xffff
addui r4,r4,(_e&0xffff)
lhi r5,(_f>>16)&0xffff
addui r5,r5,(_f&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4
lhi r3,(_k>>16)&0xffff
addui r3,r3,(_k&0xffff)
lhi r4,(_g>>16)&0xffff
addui r4,r4,(_g&0xffff)
lhi r5,(_h>>16)&0xffff
addui r5,r5,(_h&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4
lhi r3,(_l>>16)&0xffff
addui r3,r3,(_l&0xffff)
lhi r4,(_f>>16)&0xffff
addui r4,r4,(_f&0xffff)
lhi r5,(_h>>16)&0xffff
addui r5,r5,(_h&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4
lhi r3,(_m>>16)&0xffff
addui r3,r3,(_m&0xffff)
lhi r4,(_j>>16)&0xffff
addui r4,r4,(_j&0xffff)
lhi r5,(_k>>16)&0xffff
addui r5,r5,(_k&0xffff)
lf f4,0(r4)
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4
lhi r3,(_n>>16)&0xffff
addui r3,r3,(_n&0xffff)
lhi r4,(_i>>16)&0xffff
addui r4,r4,(_i&0xffff)
lhi r5,(_l>>16)&0xffff
addui r5,r5,(_l&0xffff)
lf f4,0(r4)
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4
lhi r3,(_a>>16)&0xffff
addui r3,r3,(_a&0xffff)
```

```
sf 0(r3),f4
addui r4,r4,(_e&0xffff)
lhi r5,(_f>>16)&0xffff
addui r5,r5,(_f&0xffff)
lf f4,0(r4)
lf f5,0(r5)
lhi r3,(_k>>16)&0xffff
addui r3,r3,(_k&0xffff)
multf f4,f4,f5
lhi r4,(_g>>16)&0xffff
addui r4,r4,(_g&0xffff)
lhi r5,(_h>>16)&0xffff
addui r5,r5,(_h&0xffff)
sf 0(r7),f4
lf f4,0(r4)
lf f5,0(r5)
lhi r7,(_l>>16)&0xffff
addui r7,r7,(_l&0xffff)
multf f4,f4,f5
lhi r4,(_f>>16)&0xffff
addui r4,r4,(_f&0xffff)
lhi r5,(_h>>16)&0xffff
addui r5,r5,(_h&0xffff)
sf 0(r3),f4
lf f4,0(r4)
lf f5,0(r5)
lhi r3,(_m>>16)&0xffff
addui r3,r3,(_m&0xffff)
multf f4,f4,f5
lhi r4,(_j>>16)&0xffff
addui r4,r4,(_j&0xffff)
lhi r5,(_k>>16)&0xffff
addui r5,r5,(_k&0xffff)
sf 0(r7),f4
lf f4,0(r4)
lf f5,0(r5)
lhi r7,(_n>>16)&0xffff
addui r7,r7,(_n&0xffff)
addf f4,f4,f5
lhi r4,(_i>>16)&0xffff
sf 0(r3),f4
addui r4,r4,(_i&0xffff)
lhi r5,(_l>>16)&0xffff
addui r5,r5,(_l&0xffff)
lf f4,0(r4)
lf f5,0(r5)
lhi r8,(_a>>16)&0xffff
addui r8,r8,(_a&0xffff)
addf f4,f4,f5
lhi r4,(_i>>16)&0xffff
```

```
        lhi r4,(_i>>16)&0xffff
        addui r4,r4,(_i&0xffff)
        lw r6,0(r4)
        sw 0(r3),r6
        lhi r3,(_b>>16)&0xffff
        addui r3,r3,(_b&0xffff)
        lhi r4,(_m>>16)&0xffff
        addui r4,r4,(_m&0xffff)
        lw r6,0(r4)
        sw 0(r3),r6
        lhi r3,(_c>>16)&0xffff
        addui r3,r3,(_c&0xffff)
        lhi r4,(_k>>16)&0xffff
        addui r4,r4,(_k&0xffff)
        lw r6,0(r4)
        sw 0(r3),r6
        lhi r3,(_d>>16)&0xffff
        addui r3,r3,(_d&0xffff)
        lhi r4,(_n>>16)&0xffff
        addui r4,r4,(_n&0xffff)
        lw r6,0(r4)
        sw 0(r3),r6

L1:
        trap #0
        nop
```

```
        sf 0(r7),f4
        addui r4,r4,(_i&0xffff)
        lw r6,0(r4)
        lhi r3,(_b>>16)&0xffff
        sw 0(r8),r6
        addui r3,r3,(_b&0xffff)
        lhi r4,(_m>>16)&0xffff
        addui r4,r4,(_m&0xffff)
        lw r6,0(r4)
        lhi r8,(_c>>16)&0xffff
        sw 0(r3),r6
        addui r8,r8,(_c&0xffff)
        lhi r4,(_k>>16)&0xffff
        addui r4,r4,(_k&0xffff)
        lw r1,0(r4)
        lhi r3,(_d>>16)&0xffff
        addui r3,r3,(_d&0xffff)
        lhi r4,(_n>>16)&0xffff
        addui r4,r4,(_n&0xffff)
        lw r6,0(r4)
        sw 0(r8),r1
        sw 0(r3),r6

L1:
        trap #0
        nop
```

Con este nuevo código, los datos de ejecución son los siguientes:

- Tiempo de procesamiento: 139 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 3 ciclos (2.19%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 0

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 0

> *Interrupciones:* 3

Con estas mejoras, y respecto al programa sin modificar, tenemos una ganancia de:

**Ganancia1**$_{\textbf{MejoraEj2a}}$ = 168 / 139 = **1,208**

Y una eficiencia de:

**Eficiencia1**$_{\textbf{MejoraEj2a}}$ = 1,208 / 5 = **0.2416**

En comparación con los datos obtenidos del procesador no segmentado, tenemos una ganacia de :

**Ganancia2**$_{\textbf{MejoraEj2a}}$ = 469.6 / 139 = **3.378**

Y una eficiencia de:

**Eficiencia2**$_{\textbf{MejoraEj2a}}$ = 3.378 / 5 = **0.6756**

**b) Realizando un programa en ensamblador que implemente directamente el cálculo descrito en la Figura 1.1.**

El código de este programa está en el archivo 2b.s y el contenido es el siguiente:

```
        .data

.global _a

        .align 4
_a:
        .float 0.000000000000
.global _b
        .align 4
_b:
        .float 1.000000000000
.global _c
        .align 4
_c:
        .float 1.000000000000
.global _d
        .align 4
_d:
        .float -1.000000000000

        .text

.global _main
_main:

        lhi r4,(_a>>16)&0xffff
        lhi r5,(_b>>16)&0xffff
        addui r4,r4,(_a&0xffff)
        addui r5,r5,(_b&0xffff)
        lf f4,0(r4)
        lf f5,0(r5)
        lhi r6,(_c>>16)&0xffff
        lhi r7,(_d>>16)&0xffff
        multf f9,f4,f5
        addf f8,f4,f5
        addui r6,r6,(_c&0xffff)
        addui r7,r7,(_d&0xffff)
        lf f6,0(r6)
        lf f7,0(r7)
        multf f15,f8,f9
        addf f10,f6,f7
        multf f11,f6,f7
        addf f12, f8, f10
        multf f13, f9,f11
```

```
        sf 0(r4),f12
        multf f14, f10,f11
        addf f4,f13,f12
        addf f16,f15,f14
        sf 0(r7),f4
        sf 0(r6),f14
        sf 0(r5),f16

L1:
        trap #0
        nop
```

Con este nuevo código, los datos de ejecución son los siguientes:

- Tiempo de procesamiento: 40 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 8 ciclos (20%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 7

> *Estructurales:* 2

> *Interrupciones:* 4

Las detenciones estructurales son debido a que no podemos utilizar la unidad funcional, tanto de suma como de multiplicación, entre varias instrucciones, así que hay que esperar a que esté libre para poder utilizarla.

Con estas mejoras, y respecto al programa sin modificar, tenemos una ganancia de:

$$\textbf{Ganancia1}_{\textbf{MejoraEj2b}} = 168 / 45 = \textbf{3.733}$$

Y una eficiencia de:

$$\textbf{Eficiencia1}_{\textbf{MejoraEj2b}} = 3.733 / 5 = \textbf{0.7466}$$

En comparación con los datos obtenidos del procesador no segmentado, tenemos una ganacia de :

$$\textbf{Ganancia2}_{\textbf{MejoraEj2b}} = 469.6 / 45 = \textbf{10.435}$$

Y una eficiencia de:

$$\textbf{Eficiencia2}_{\textbf{MejoraEj2b}} = 10.435 \ / 5 = \textbf{2.087}$$

Si calculamos la ganancia con respecto a los datos obtenidos en el ejercicio 2a, tenemos una ganacia de:

$$\textbf{Ganancia3}_{\textbf{MejoraEj2b}} = 139 / 45 = \textbf{3.088}$$

Y podemos ver así, como esta implementación tarda menos tiempo en ejecutarse que la implementación reordenando código (ej 2a).

**4. Utilizando la mejor y la peor versión del programa del fichero *pr1_1.s* determine la influencia que tiene en las prestaciones el cambio en los ciclos del multiplicador y del sumador del cauce. Para ello, aumente 10 veces en paralelo la latencia de estas dos unidades, partiendo de los valores por defecto (2,5) hasta (20,50).**

La mejor versión es la implementada en el ejercicio 2b, y la peor, la que nos da.
Con esto, y modificando la latencia de las unidades (delay) en el menú Configiration→Floating PointStages…, tenemos los siguientes datos:

**- Con la peor versión:**

**Delay (2,5)**

- Tiempo de procesamiento: 168 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 39 ciclos (23.21%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 25

> *Interrupciones:* 4

**Delay (4,10)**

- Tiempo de procesamiento: 203 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 74 ciclos (36.45%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 60

> *Interrupciones:* 4

**Delay (6,15)**

- Tiempo de procesamiento: 238 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 109 ciclos (45.80%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 95

> *Interrupciones:* 4

**Delay (8,20)**

- Tiempo de procesamiento: 273 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 144 ciclos (52.75%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 130

> *Interrupciones:* 4

**Delay (10,25)**

- Tiempo de procesamiento: 308 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 179 ciclos (58.12%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 165

> *Interrupciones:* 4

**Delay (12,30)**

- Tiempo de procesamiento: 343 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 214 ciclos (62.39%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 200

> *Interrupciones:* 4

**Delay (14,35)**

- Tiempo de procesamiento: 378 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 249 ciclos (65.87%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 235

> *Interrupciones:* 4

**Delay (16,40)**

- Tiempo de procesamiento: 413 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 284 ciclos (68.76%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 270

> *Interrupciones:* 4

**Delay (18,45)**

- Tiempo de procesamiento: 448 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 319 ciclos (71.20%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 305

> *Interrupciones:* 4

**Delay (20,50)**

- Tiempo de procesamiento: 483 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 354 ciclos (73.29%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 14

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 340

> *Interrupciones:* 4

**- Con la mejor versión (2b.s):**

**Delay (2,5)**

- Tiempo de procesamiento: 40 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 8 ciclos (20%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 7

> *Estructurales:* 2

> *Interrupciones:* 4

**Delay (4,10)**

- Tiempo de procesamiento: 57 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 26 ciclos (43.33%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 25

> *Estructurales:* 7

> *Interrupciones:* 4

**Delay (6,15)**

- Tiempo de procesamiento: 84 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 45 ciclos (53.57%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 44

> *Estructurales:* 12

> *Interrupciones:* 4

**Delay (8,20)**

- Tiempo de procesamiento: 108 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 64 ciclos (59.26%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 63

> *Estructurales:* 17

> *Interrupciones:* 9

**Delay (10,25)**

- Tiempo de procesamiento: 132 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 83 ciclos (59.26%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 82

> *Estructurales:* 22

> *Interrupciones:* 11

**Delay (12,30)**

- Tiempo de procesamiento: 156 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 102 ciclos (65.38%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 101

> *Estructurales:* 27

> *Interrupciones:* 13

**Delay (14,35)**

- Tiempo de procesamiento: 180 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 121 ciclos (67.22%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 120

> *Estructurales:* 32

> *Interrupciones:* 15

**Delay (16,40)**

- Tiempo de procesamiento: 204 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 140 ciclos (68.33%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 139

> *Estructurales:* 37

> *Interrupciones:* 17

**Delay (18,45)**

- Tiempo de procesamiento: 228 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 159 ciclos (69.74%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

* Branch/Jump (de salto): 0

* Floating point (de coma flotante): 158

> *Estructurales:* 42

> *Interrupciones:* 19

**Delay (20,50)**

- Tiempo de procesamiento: 252 ciclos.

- Número de ciclos en los que el cauce ha estado detenido (stall): 178 ciclos (73.29%).

- Causas de las detecciones:

> *RAW (Read Alter Write):*

* LD (de carga): 1

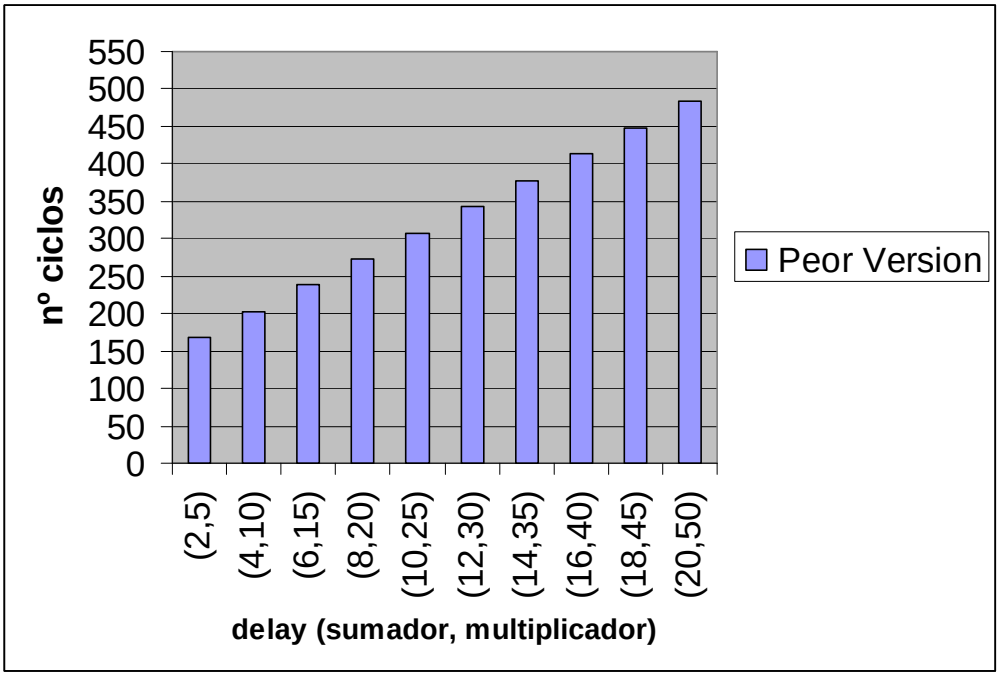
* Branch/Jump (de salto): 0

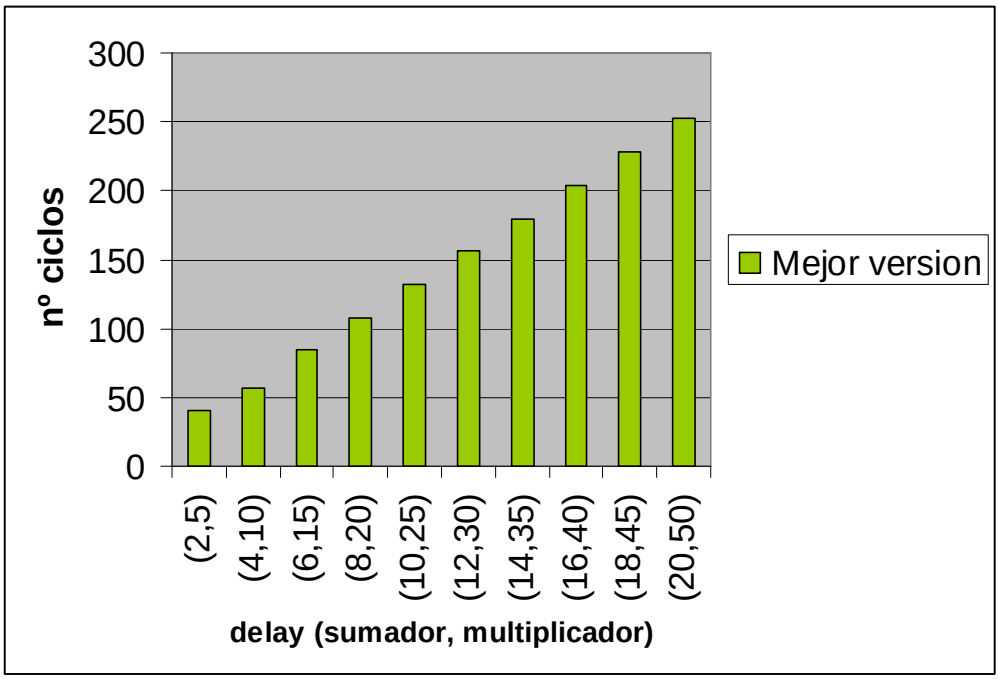
* Floating point (de coma flotante): 177

> *Estructurales:* 47

> *Interrupciones:* 21

Podemos ver mejor estos datos en los siguientes gráficos:

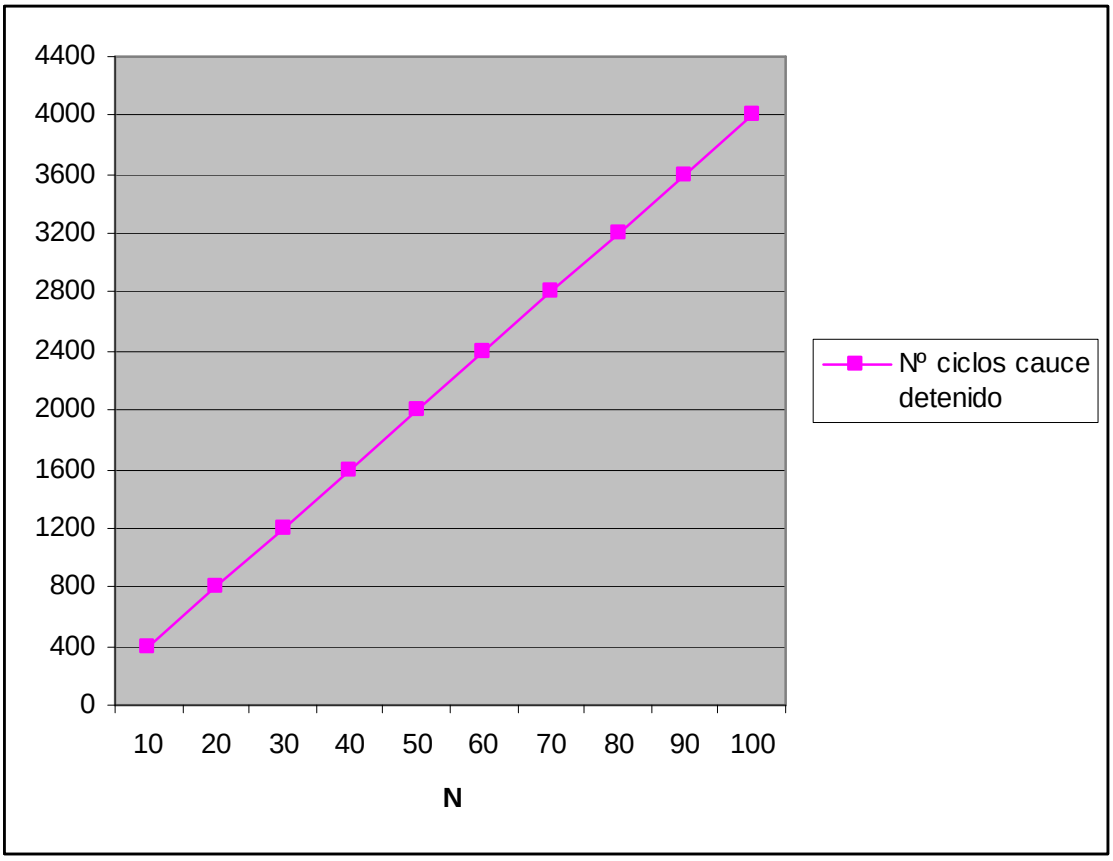



Como podemos comprobar en cada uno de los gráficos, el aumento del delay para el multiplicador y el sumador, hace que los datos obtenidos sean cada vez peores.
Esto es debido a tener que esperar más ciclos a que se ejecute una suma o una multiplicación.

**5. Suponga que la secuencia de operaciones de la Figura 1.1 se encuentra dentro de un bucle que repite un número de veces N. Evalúe las prestaciones que se obtienen en función de N para el código del programa *pr1_1.s*. Para ello, varíe el valor de N desde 1 hasta 100 de 10 en 10.**

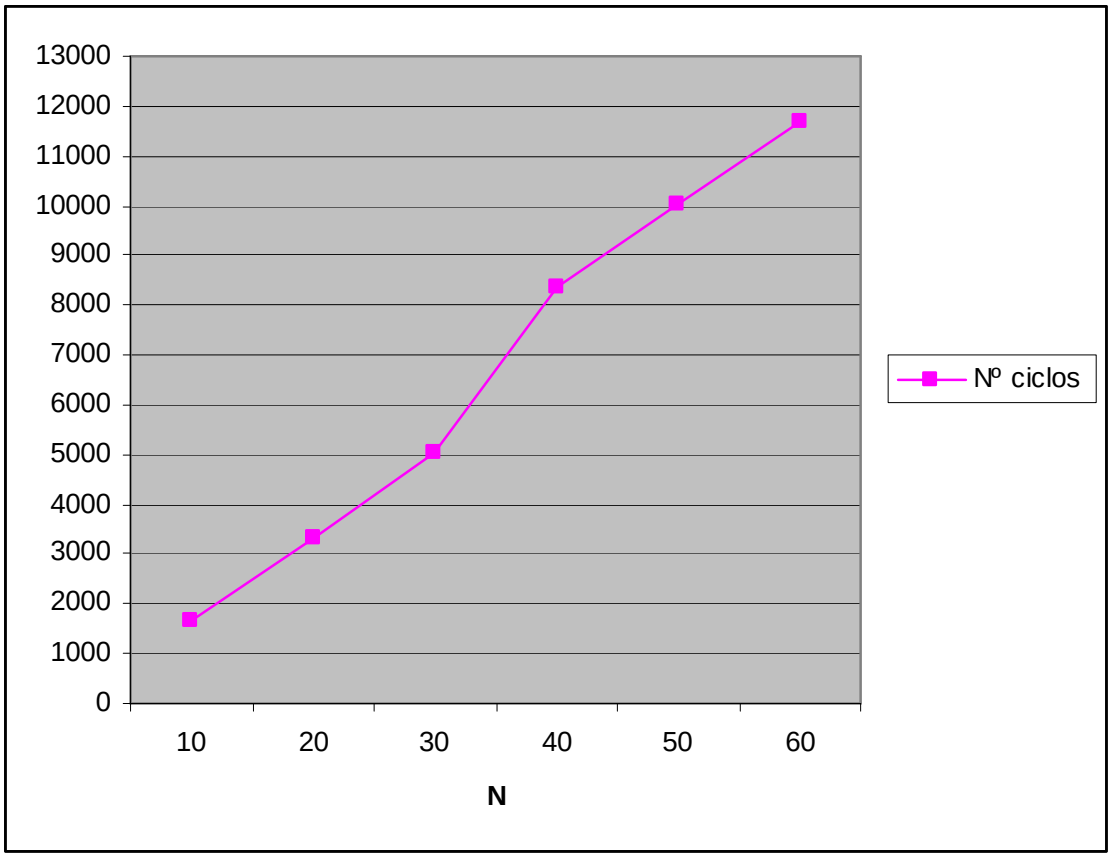
Primero modifico el programa para incluir el bucle (archivo ej5.s), y luego voy variando la N para ir obteniendo los diferentes resultados.
La siguiente gráfica muestra el nº de ciclos en los que el cauce ha estado detenido (stalls) en cada una de las pruebas realizadas con cada N.




Como era de esperar, el resultado es lineal con cada N. Si en N=10 tarda 400, es lógico que en N=20 sea el doble puesto que N es el doble, y así, sucesivamente.

Del mismo modo, el tiempo que tarda en ejecutarse el programa, también será lineal.

Esto puede verse en las primeras ejecuciones del programa en la siguiente gráfica:



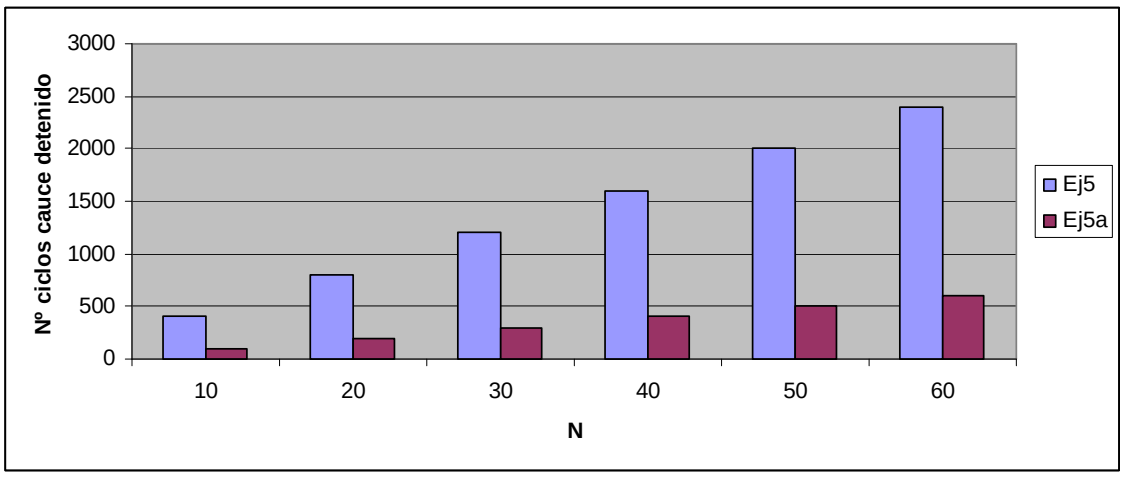

El programa tarda cada vez más y más cuantas más iteraciones tiene el bucle (cuanto mayor es N). No es necesario seguir hasta N=100 puesto que, a parte de que la ejecución comienza a tardar bastante tiempo, se ve claramente como empeora enormemente cada vez que aumentamos nuestra N.

En conclusión, las prestaciones de este programa empeoran linealmente cada vez que aumentamos el número de iteraciones.

**Optimice el programa obtenido:**
**(a) mediante técnicas como desenrollado de bucles, salto retardado, reordenación de instrucciones, etc.**

Una vez comprobado que cuanto mayor sea el número de iteraciones, el rendimiento empeora linealmente, voy a realizar varias modificaciones en el programa para mejorarlo.
Como en uno de los ejercicios anteriores (2a) tuve que realizar una de las modificaciones de código que se nos piden en este apartado (reordenación de instrucciones) y otras como adición de registros; voy a optimizar el programa a partir del creado en el ejercicio 2a.
Aplicaré también la mejora de desenrollado de bucles, pero hay que tener en cuenta que, aunque esta técnica mejora las prestaciones, hay que tener un compromiso entre el uso de memoria y la mejora, por tanto, para no ocupar mucha memoria, voy a desenrollar el bucle solamente 2 veces.

En el siguiente gráfico podemos ver la comparación entre el programa inicial y este con el desenrollado de bucle, reordenamiento de código,etc.



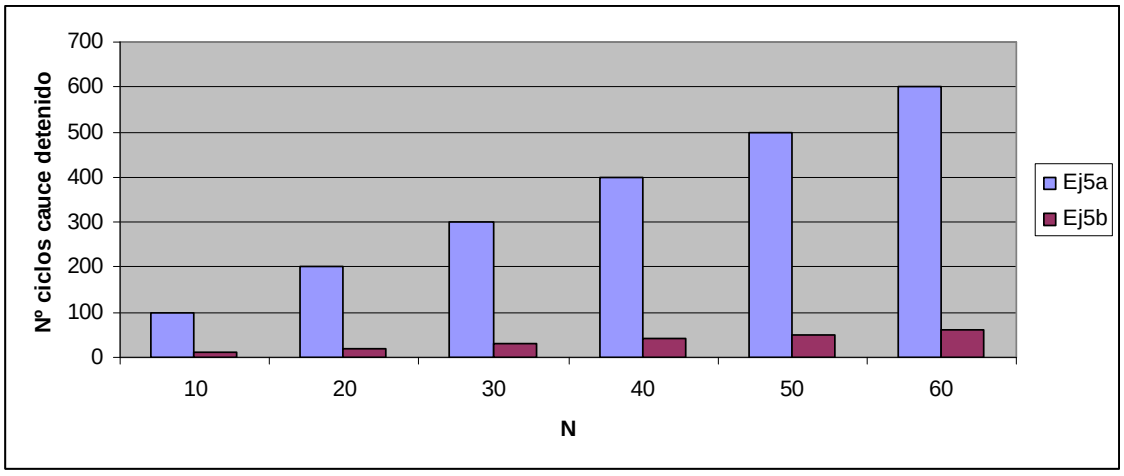

El código del programa está en el archivo ej5a.s

**(b) realizando directamente el programa en ensamblador que realice el mismo cálculo.**

Para ello voy a usar el programa del ejercicio 2a, que ya hacía esto y lo voy a modificar para añadirle el bucle.

Como vimos en los resultados del ejercicio 2, obviamente, va a mejorar al programa anterior, ¿pero cuánto?

Pues en el siguiente gráfico se puede observa la gran mejora:




Y el número de ciclos disminuye de igual manera.
Como podemos comprobar, mediante un programa equivalente, el rendimiento aumenta considerablemente.

El código del programa está en el archivo ej5b.s