# ARQUITECTURA DE COMPUTADORES

# Práctica ensamblador DLX

# Ingeniería informática Universidad de Salamanca

# 8/05/2024


# Álvaro García Sánchez - 70924450V


## 1. Introducción

El objetivo de la práctica es el desarrollo y optimización de un código que realice el siguiente cálculo:

Calcular una secuencia de números “estilo 3n+1” partiendo de un valor inicial dado (valor_inicial word con rango de 1 a 100, incluidos), siendo que si su valor es, por ejemplo 3, la secuencia obtenida sería 3-10-5-16-8-4-2-1 (ES OBLIGATORIO USAR ESTE ALGORITMO DE RESOLUCIÓN).

$secuencia[0] = valor\_inicial$

$$secuencia[n] = \begin{cases} \frac{secuencia[n-1]}{2}, si\ secuencia[n-1] es\ par \\ secuencia[n-1]x3+1, \quad si\ secuencia[n-1] es\ impar \end{cases}$$

Además, se desea rellenar una lista de la siguiente manera (lista).

a. Siendo vT=secuencia_tamanho, vIni=valor_inicial, vMax=secuencia_maximo y vMed=secuencia_valor_medio

b. lista = [ vIni*vT, vMax*vT, vMed*vT, (vIni/vMax)*vT, (vIni/vMed)*vT, (vMax/vIni)*vT, (vMax/vMed)*vT, (vMed/vIni)*vT, vMed/vMax)*vT ]

En las variables secuencia, secuencia_tamanho, secuencia_maximo, secuencia_valor_medio, lista, lista_valor_medio al finalizar la ejecución del programa deberán aparecer los valores pedidos

## 2. Programa no optimizado

Para realizar este primer programa funcional hemos tomado las siguientes acciones basándonos en las definiciones de variables proporcionadas en el enunciado de la práctica:

- Cargamos el valor inicial en r1 y en r4, uno para almacenar el valor y otro para empezar a operar con él.

- Cargamos en r7 el valor inicial (r1) y el número 1 almacenado en r2 para determinar si el número inicial es par o impar.
- Añadimos el entero 2 al registro r10 y el entero 3 al registro r11, esto nos permitirá identificar si debemos dividir o multiplicar.
- Entramos en el bucle en el cual, se comienza guardando el valor inicial (r4) en la secuencia, posteriormente, se suma una cifra al tamaño de la secuencia y se adelantan 4 posiciones desde r6 para almacenar el siguiente número. Luego, se hace una comparación para determinar si el número que se está tratando es mayor que el anterior, de este modo, se hace un salto hasta la función “mayor” en la que se intercambia el valor del registro de r4 al de r8. Por último, se hace un sumatorio de todos los valores de la secuencia en r20, y se comprueba si el número es par o impar. En caso de que sea par, se realiza un salto a la etiqueta “par” donde se realiza la división y se retorna al bucle, lo mismo si el número resulta impar pero realizando una multiplicación y comprobando si la secuencia es igual a 1 (fin de la secuencia).
- Finalmente se almacenan el tamaño y el valor máximo de la secuencia en las variables correspondientes.
- La segunda parte del ejercicio es simplemente pasar los valores de los registros enteros sobre los que vamos a trabajar a registros float, en concreto, r20-f21, r5-f22, r1-f23 y r8-f24. Luego, realizamos las operaciones que se indican en el enunciado por orden y guardando los valores en “lista”
- En la función “sumalista”, recorremos la lista completamente y realizamos el sumatorio de todos los valores sobre f25
- Por último, en la etiqueta “fin”, añadimos al sumatorio el último elemento de la lista y cargamos en f9 el valor 9 (total de elementos de la lista) y calculamos la media de los valores que almacenamos en la variable lista_valor_medio.

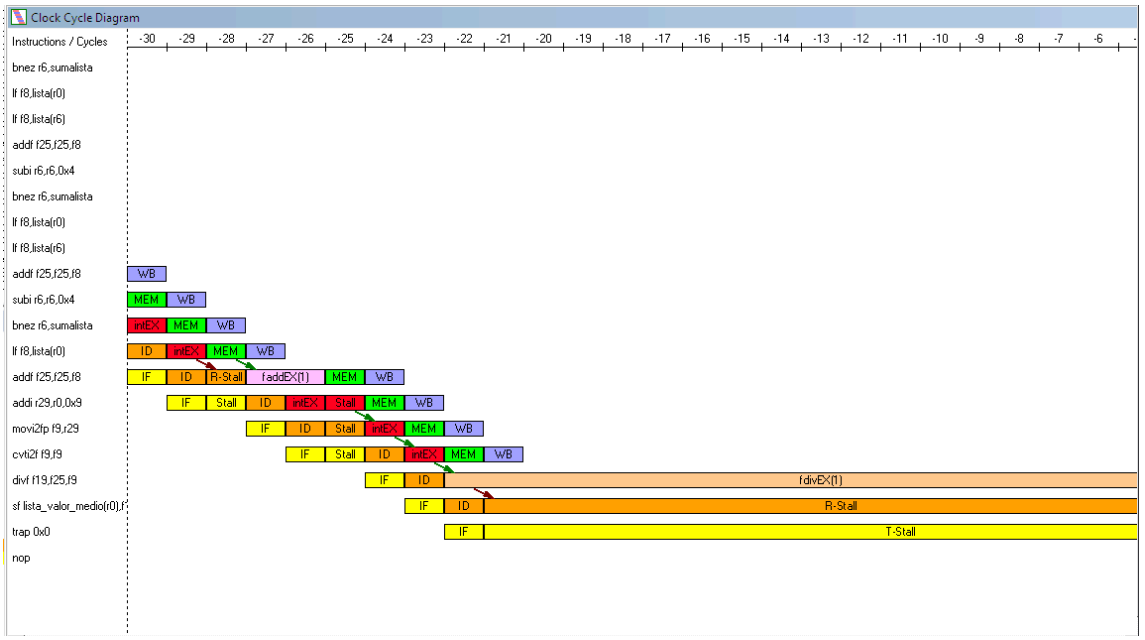
| ESTADÍSTICAS – No optimizado - Valor inicial = 3 | |
|---|---|
| Total | |
| Número de ciclos: | 528 |
| Número de instrucciones ejecutadas (IDs): | 190 |
| Stalls | |
| RAW stalls | 305 |
| LD stalls | 9 |
| Branch/jump stalls | 24 |
| Floating point stalls | 272 |
| WAW stalls | 0 |
| Structural stalls | 0 |
| Control stalls | 28 |
| Trap stalls | 21 |
| Total | 354 |
| Conditional branches | |
| Total | 27 |
| taken | 17 |
| Not taken | 10 |
| Load / store - instructions | |
| Total | 32 |
| Load | 11 |
| Store | 21 |
| Floating point stage instructions | |
| Total | 34 |
| Additions | 9 |
| Multiplications | 12 |
| Divisions | 13 |
| Traps | |
| Traps | 1 |

## 3. Algoritmo optimizado

Usando la herramienta winDLX para ejecutar el programa, obtenemos una gráfica de las funciones que se van ejecutando y podemos ver en qué zonas se “atasca” el programa y que por tanto podemos trabajar para mejorar su rendimiento

Las opereciones de división implementadas producían Stalls muy amplios que reducían considerablemente la eficiencia y rápidez del programa. La solución que aportamos fue simplemente reordenar las operaciones para añadir más espacio entre la división y la operación de guardar que espera por el resultado de dicha división.

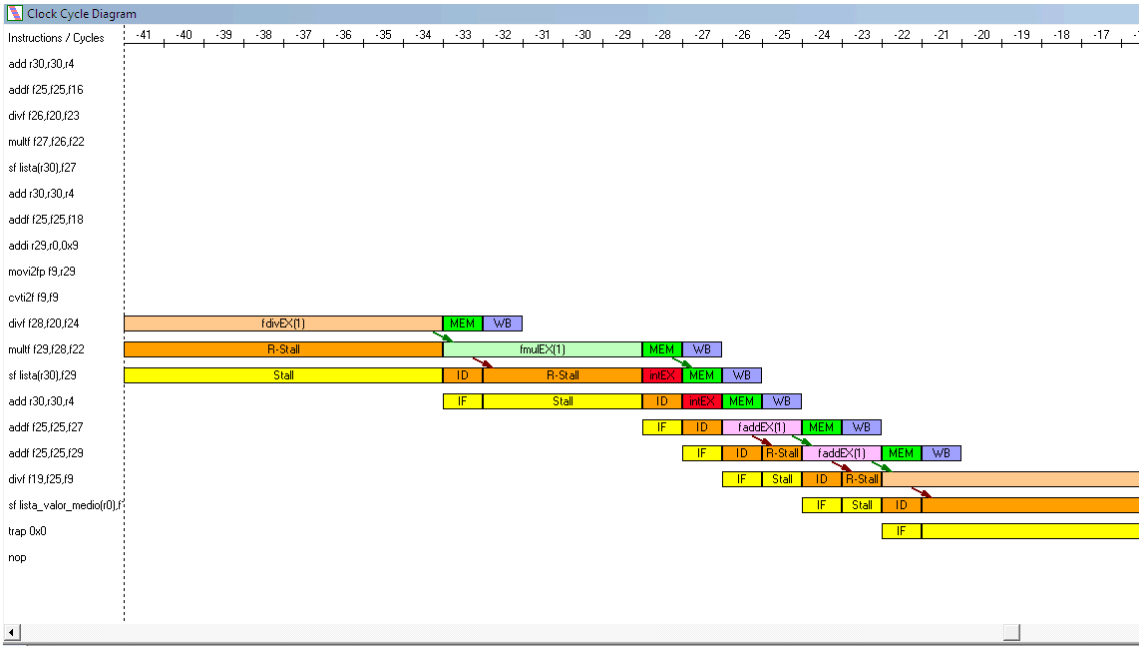
Antes:




Como podemos ver, el programa está parado un tiempo considerable en la función de “sumalista” ya que tiene que recorrer la lista completa para calcular el sumatorio y almacenarlo, esta parte se puede corregir integrando el sumatorio mientras se realizan las operaciones de los componentes de la lista.

Una vez corregido, obtenemos el siguiente resultado.

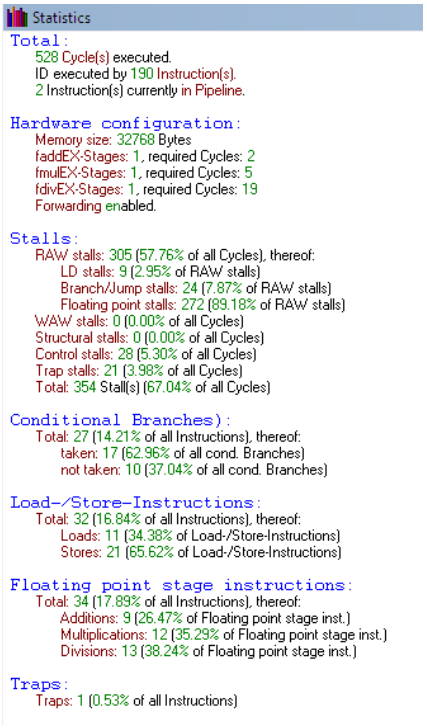
Después:



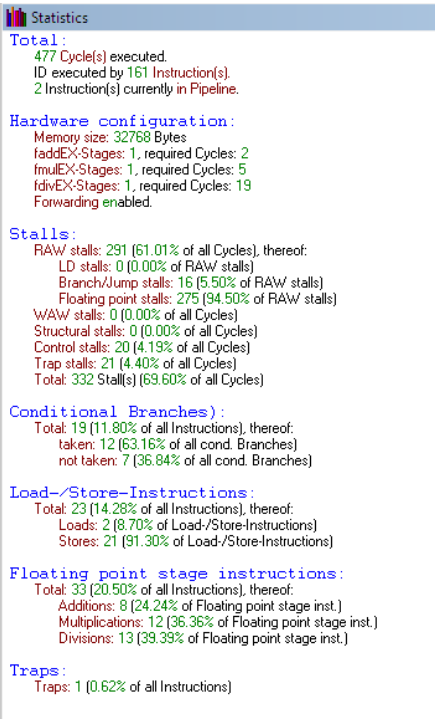
| ESTADÍSTICAS – Optimizado | |
|---|---|
| Total | |
| Número de ciclos: | 477 |
| Número de instrucciones ejecutadas (IDs): | 161 |
| Stalls | |
| RAW stalls | 291 |
| LD stalls | 0 |
| Branch/jump stalls | 16 |
| Floating point stalls | 275 |
| WAW stalls | 0 |
| Structural stalls | 0 |
| Control stalls | 20 |
| Trap stalls | 21 |
| Total | 332 |
| Conditional branches | |
| Total | 19 |
| taken | 12 |
| Not taken | 7 |
| Load / store - instructions | |
| Total | 23 |
| Load | 2 |
| Store | 21 |
| Floating point stage instructions | |
| Total | 33 |
| Additions | 8 |
| Multiplications | 12 |
| Divisions | 13 |
| Traps | |
| Traps | 1 |

## 4. Comparación de resultados

Optimizado

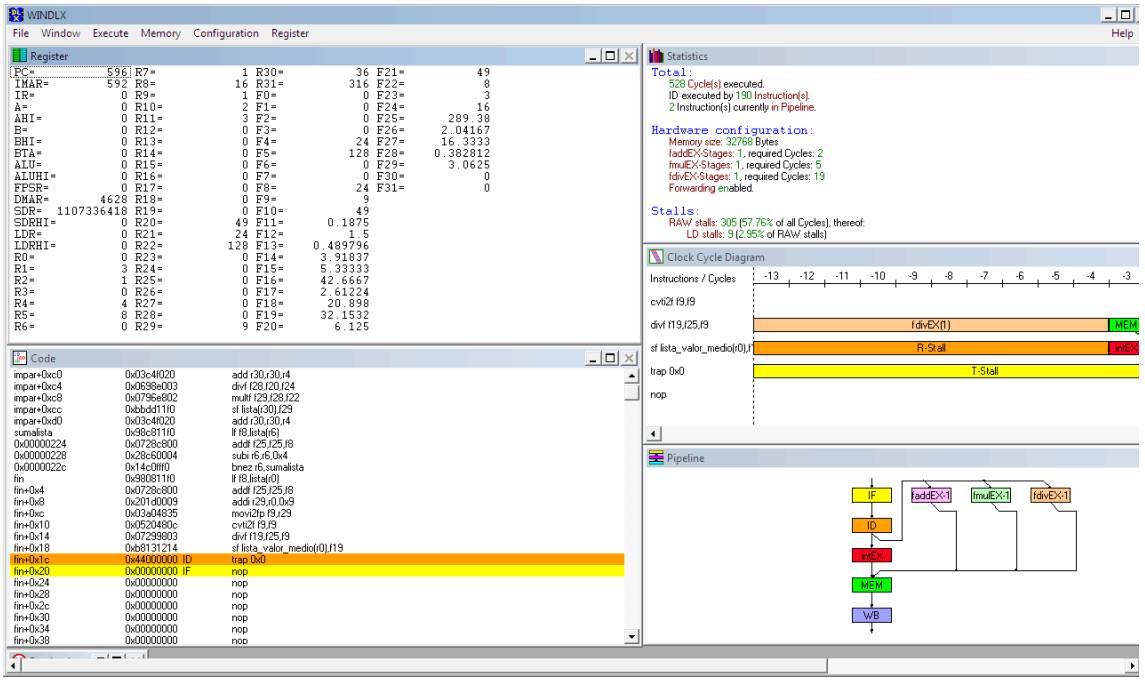



Sin optimizar



## 5. Prueba de ejecución

No optimizado




Optimizado