PRÁCTICA 2:

Implementación y optimización de un algoritmo en ensamblador DLX

JUAN GIL SANCHO
PABLO CAÑO PASCUAL

1. Introducción:

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

$$M = V(a_1, a_2, a_3, a_4) \times \frac{(a_2/a_5) + (a_4/a_5)}{(a_1/a_5) + (a_3/a_5)}$$

$$a_1, a_2, a_3, a_4, a_5$$

$$checkA, \quad checkM$$

siendo:

- M matriz 4x4: $\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$
- $checkM = m_{11} + m_{12} + m_{13} + m_{14} + m_{21} + \cdots + m_{34} + m_{41} + m_{42} + m_{43} + m_{44}$ (suma de todos los elementos de M)
- a_x media aritmética de $listaX$
  - La media se podrá hacer sobre 0, 5, 10, 15 y 20 elementos de las listas en el orden dado. Esto se indicará en la variable $tamano$.
- $checkA = a_1 * a_2 * a_3 * a_4 * a_5$ (producto de las medias)
- $V(a_1, a_2, a_3, a_4)$ *la matriz de Vandermonde*:

$$V(a_1, a_2, a_3, a_4) = \begin{bmatrix} 1 & a_1 & {a_1}^2 & {a_1}^3 \\ 1 & a_2 & {a_2}^2 & {a_2}^3 \\ 1 & a_3 & {a_3}^2 & {a_3}^3 \\ 1 & a_4 & {a_4}^2 & {a_4}^3 \end{bmatrix}$$

En primer lugar desarrollaremos un código en el que no tendremos en cuenta las principales técnicas de optimización que hemos estudiado. Las estadísticas que nos muestra son las siguientes:

ESTADÍSTICAS (SIN OPTIMIZAR)	
Total	
Nº de ciclos:	2417
Nº de instrucciones ejecutadas (IDs):	1258
Stalls	
RAW Stalls.	784 (32,44 % de los ciclos)
LD Stalls:	141 (17,98% de los RAW stalls)
Branch/Jump stalls:	193 (24, 62% de los RAW stalls)
Floating point stalls:	450 (57,40% de los RAW stalls)
WAW stalls:	0 (0,00 % de los ciclos)
Structural stalls:	55 (2,28% de los ciclos)
Control stalls:	204 (8,44% de los ciclos)
Trap stalls:	2 (0,08% de los ciclos)
Total:	1045 stalls (43,24% de los ciclos)
Conditional Branches	
Total:	216 (17,17 % de las instrucciones)
Tomados:	180 (83,33% de las conditional branches)
No tomados:	36 (19,70% de los conditional branches)
Instrucciones Load/Store	
Total:	198 (15, 74% de las instrucciones)
Loads:	159 (80,30% de las instrucciones load/store)
Store:	39 (19,70% de las instrucciones load/store)
Instrucciones de punto flotante	
Total:	199 (15, 82% de las instrucciones)
Sumas:	120 (60,30% de las instrucciones en punto flotante)
Multiplicaciones:	57 (28,64% de las instrucciones en punto flotante)
Divisiones:	22(11,06% de las instrucciones en punto flotante)
Traps	
Traps:	Traps:1 (0,08% de las instrucciones)

WinDLX nos aporta gran cantidad de información acerca de la ejecución del programa, como vemos inicialmente hemos conseguido **2417 ciclos** que se verán reducidos drásticamente en sucesivas optimizaciones. También vemos el apartado “Stalls” que indica el número de ciclos que tienen paradas y su porcentaje con respecto al total de los ciclos. En este caso, el 32,44% de los ciclos tienen paradas. También destaca el apartado “Conditional Branches” que nos indica las instrucciones que están en un condicional.

2. Técnicas de optimización usadas:

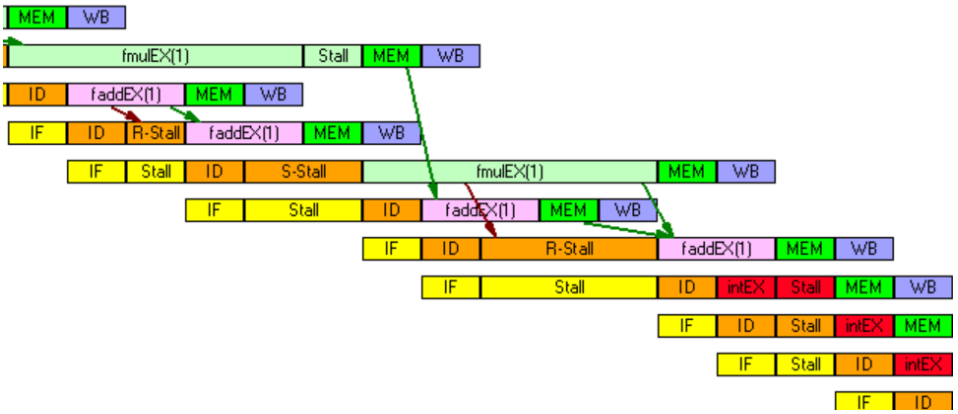
Las principales técnicas usadas para optimizar el código son las siguientes, el aprovechamiento de distintas unidades funcionales de forma paralela, la carga de números en coma flotante con registros double-float, el desenrollado de bucles y la redistribución de la carga con instrucciones de suma de coma flotante.

El aprovechamiento de distintas unidades funcionales ha sido usado al insertar el cálculo de las potencias entre divisiones de forma que aprovechamos mejor los recursos en vez de hacer estos cálculos en un bucle al final. Como vemos, inicialmente usábamos un bucle:

```
186
187             ;Este bucle en C seria algo así
188             ;for(x = 0; x < j; x+=4) valiendo j un multiplo de 4
189             ;y donde x = r13 y j = r8
190             bucle_potencias:
191                 ;Multiplicamos aux = aux (f8) * aX (f7)
192                 multf f8, f8, f7
193                 addi r13, r13, #4
194                 slt r4, r13, r8
195                 bnez r4, bucle_potencias
```

Como las divisiones requieren de muchos ciclos, mientras calculábamos las medias realizábamos los cálculos también de las potencias, estando a1 en f0, a2 en f1, a3 en f2, a4 en f3 y a5 en f4 como podemos ver:

```
312         medias:
313         divf f1, f1, f5
314         multf f22, f0, f0
315         multf f23, f22, f0
316         divf f2, f2, f5
317         multf f24, f1, f1
318         multf f25, f1, f24
319         multf f10, f0, f1
320         divf f3, f3, f5
321         multf f26, f2, f2
322         multf f27, f2, f26
323         multf f10, f10, f2
324         divf f4, f4, f5
325         multf f28, f3, f3
326         sd potencias, f22
327         sd potencias+8, f24
328         sd potencias+16, f26
329         multf f29, f3, f28
330         multf f10, f10, f3
```



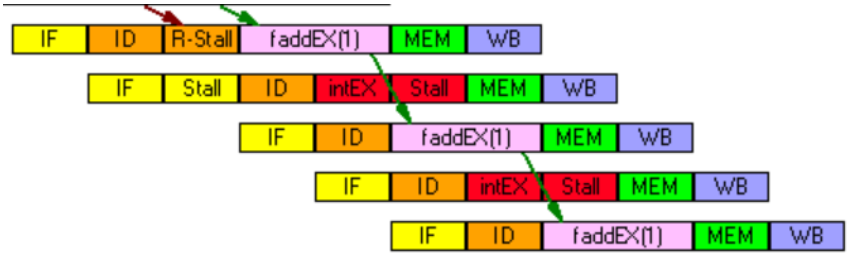
Ejemplo del uso de distintas unidades funcionales simultáneamente

El desenrollado de bucles se ha hecho de forma que esté explícita cada iteración ahorrándonos así muchos ciclos, tanto en la comprobación de condiciones de los bucles como en la redistribución de la carga. Inicialmente utilizábamos un bucle tal que:

```
116     ;Cargamos el elemento de la lista y lo añadimos a la suma
117     bucle_media:
118         lf f1, lista1(r7)
119         addf f2, f2, f1
120         addi r7, r7, #4
121         addi r3, r3, #1
122
123         ;Se acaba el bucle si r3 == r5 (numero de elementos / bytes de la lista)
124         seq r4, r3, r5
125         beqz r4, bucle_media
126     divf f3, f2, f4
127
```

La redistribución de la carga de las instrucciones la hemos hecho principalmente en la carga de las listas en los registros para calcular las medias, que es la etapa del programa que más ciclos consume. Esto es así porque las sumas con números en punto flotante duran seis ciclos en vez de cinco, nuestra solución fue intercalar estas sumas con la carga de los números y otras operaciones como comprobación de condiciones:

```
103     siguienteMedia1:
104         lf f27, lista2+16(r7)
105         addf f1, f10, f11
106         addf f1, f1, f12
107         ld f14, lista3(r7)
108         addf f1, f1, f13
109         ld f16, lista3+8(r7)
110         addf f1, f1, f27
111
112         lf f28, lista3+16(r7)
113         addf f2, f14, f15
114         addf f2, f2, f16
115         ld f18, lista4(r7)
116         addf f2, f2, f17
117         ld f20, lista4+8(r7)
118         addf f2, f2, f28
```



Ejemplo de la redistribución de carga en las instrucciones

Gracias a estos cambios, el número de ciclos ha disminuido cuantitativamente como podemos ver en la siguiente tabla:

ESTADÍSTICAS OPTIMIZADO	
Total	
Nº de ciclos:	466
Nº de instrucciones ejecutadas (IDs):	300
Stalls	
RAW Stalls.	73 (15, 66 % de los ciclos)
LD Stalls:	2 (2,74 % de los RAW stalls)
Branch/Jump stalls:	2 (2,74% de los RAW stalls)
Floating point stalls:	69 (94,52% de los RAW stalls)
WAW stalls:	0 (0,00 % de los ciclos)
Structural stalls:	74 (15,88% de los ciclos)
Control stalls:	4 (0,86 % de los ciclos)
Trap stalls:	3 (0,64% de los ciclos)
Total:	154 stalls (33,05% de los ciclos)
Conditional Branches	
Total:	10 (3,33 % de las instrucciones)
Tomados:	4 (40,00% de las conditional branches)
No tomados:	6 (60,00% de los conditional branches)
Instrucciones Load/Store	
Total:	91 (30,33% de las instrucciones)
Loads:	71 (78,02% de las instrucciones load/store)
Store:	20 (21,98% de las instrucciones load/store)
Instrucciones de punto flotante	
Total:	145 (48,33% de las instrucciones)
Sumas:	113 (77,93% de las instrucciones en punto flotante)
Multiplicaciones:	26 (17,93% de las instrucciones en punto flotante)
Divisiones:	6 (4,14% de las instrucciones en punto flotante)
Traps	
Traps:	Traps:1 (0,33% de las instrucciones)

3. Conclusión:

Utilizando las técnicas mencionadas hemos conseguido reducir de 2417 ciclos a 466 lo que supone un **518,67%** de mejora, también hemos visto cómo se han reducido el porcentaje de "Raw Stalls" y de "Conditional Branches" lo que confirma la importancia de la optimización en este tipo de procesadores.

Queda de manifiesto el hecho de que la optimización de programas en lenguaje ensamblador puede dar una mejora muy grande en el rendimiento de este tipo de programas. No obstante, la legibilidad, el tamaño del código y la facilidad de mantenimiento del código son directamente contrarias al código más óptimo posible. Al desenrollar bucles o intercalar distintos tipos de instrucciones para redistribuir la carga hace que sea más complicado entender el programa y modificarlo y puede resultar especialmente malo para proyectos grandes donde se pierda mucho tiempo.