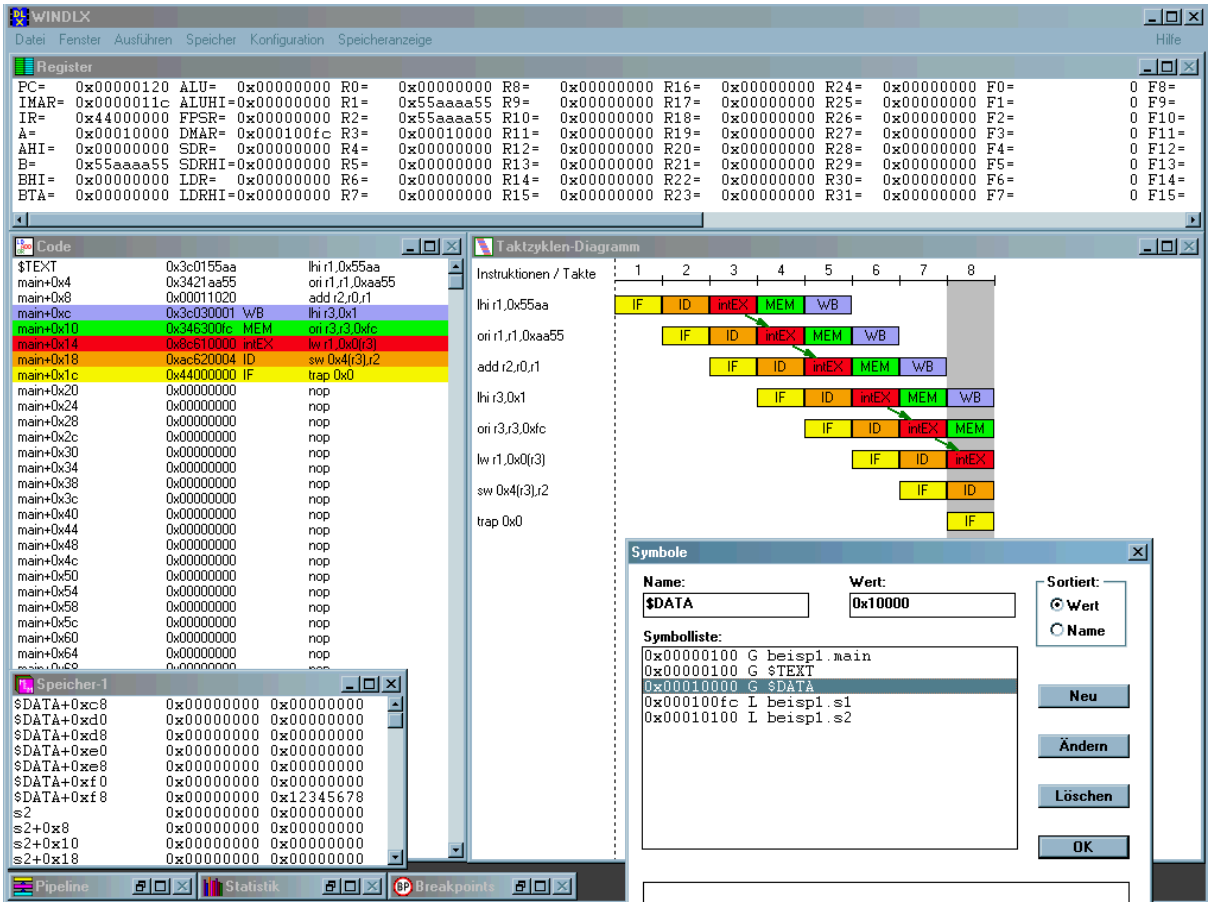
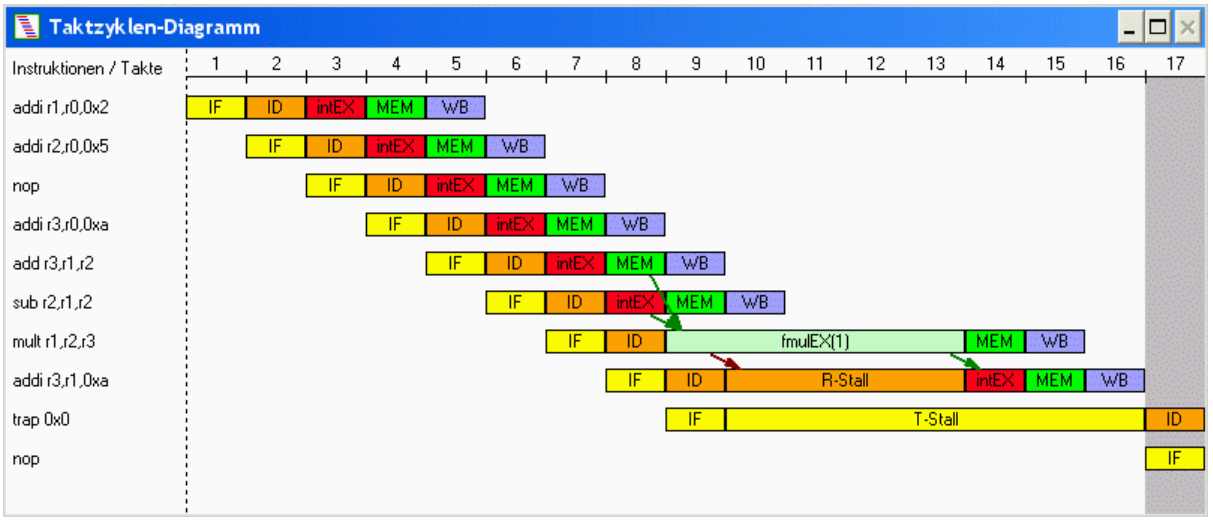
Práctica 2: ENSAMBLADOR DLX

CARLOS CONDE VICENTE **70919412Q**

DIEGO DE CASTRO MERILLAS **71043687E**

Asignatura: Arquitectura de Computadores
Curso: 3º Ingeniería Informática





ÍNDICE

Introducción

¿Qué es DLX?
DLX es un simulador de procesadores RISC, en el cual podemos ver el funcionamiento de la segmentación encauzada (pipeline).
La arquitectura de DLX es: procesador, registros, memoria.
son sencillas.

¿Objetivos del proyecto?
El proyecto consiste en hacer un programa usando el simulador DLX que no esté optimizado, es decir, usar las órdenes pues como entienda el programador que sea correcto, normalmente cuando se programa en lenguajes de alto nivel se usa, que queda bonito el código, uniendo instrucciones, junto que tambien sea optimizado, no usar un if incorrectamente, o usar instrucciones que sean redundantes, es decir un codigo de buen programador.
Luego se pide un programa optimizado, es decir, al final hay optimizaciones que hace el compilador mejorando así la rapidez del programa, lo que se nos pide es que con el simulador DLX, hagamos un programa que genere menos ciclos que él no optimizado, haciendo así que sea mejor energéticamente y en tiempo

Resultados obtenidos del código

A continuación se mostrarán dos tablas con la información de la ejecución del código no optimizado y el optimizado. Se usará como número iniciales el 97, 66 y 10.

ESTADÍSTICAS para valor inicial 97 NO optimizado	
Total	
Nº de ciclos:	2680
Nº de instrucciones ejecutadas (IDs):	1669
Stalls	
RAW stalls:	704
LD stalls:	1
Branch/Jump stalls:	356
Floating point stalls:	347
WAW stalls:	0
Structural stalls:	0
Control stalls:	303
Trap stalls:	3
Total	1010
Conditional Branches	
Total:	356
Tomados:	183
No tomados:	173
Instrucciones Load/Store	
Total:	135
Loads:	2
Stores:	133
Instrucciones de punto flotante	
Total:	69
Sumas:	9
Multiplicaciones:	52
Divisiones:	8
Traps	
Traps:	1

ESTADÍSTICAS para valor inicial 97 optimizado	
Total	
Nº de ciclos:	977
Nº de instrucciones ejecutadas (IDs):	847
Stalls	
RAW stalls:	15
LD stalls:	0
Branch/Jump stalls:	0
Floating point stalls:	15
WAW stalls:	0
Structural stalls:	41
Control stalls:	69
Trap stalls:	7
Total	132
Conditional Branches	
Total:	121
Tomados:	51
No tomados:	70
Instrucciones Load/Store	
Total:	134
Loads:	2
Stores:	132
Instrucciones de punto flotante	
Total:	23
Sumas:	9
Multiplicaciones:	9
Divisiones:	5
Traps	
Traps:	1

ESTADÍSTICAS para valor inicial 66 NO optimizado	
Total	
Nº de ciclos:	811
Nº de instrucciones ejecutadas (IDs):	440
Stalls	
RAW stalls:	291
LD stalls:	1
Branch/Jump stalls:	83
Floating point stalls:	207
WAW stalls:	0
Structural stalls:	0
Control stalls:	76
Trap stalls:	3
Total	370
Conditional Branches	
Total:	83
Tomados:	47
No tomados:	36
Instrucciones Load/Store	
Total:	44
Loads:	2
Stores:	42
Instrucciones de punto flotante	
Total:	34
Sumas:	2
Multiplicaciones:	17
Divisiones:	8
Traps	
Traps:	1

ESTADÍSTICAS para valor inicial 66 optimizado	
Total	
Nº de ciclos:	318
Nº de instrucciones ejecutadas (IDs):	241
Stalls	
RAW stalls:	15
LD stalls:	0
Branch/Jump stalls:	0
Floating point stalls:	15
WAW stalls:	0
Structural stalls:	41
Control stalls:	16
Trap stalls:	7
Total	79
Conditional Branches	
Total:	30
Tomados:	11
No tomados:	19
Instrucciones Load/Store	
Total:	43
Loads:	2
Stores:	41
Instrucciones de punto flotante	
Total:	23
Sumas:	9
Multiplicaciones:	9
Divisiones:	5
Traps	
Traps:	1

ESTADÍSTICAS para valor inicial 10 NO optimizado	
Total	
Nº de ciclos:	384
Nº de instrucciones ejecutadas (IDs):	160
Stalls	
RAW stalls:	200
LD stalls:	1
Branch/Jump stalls:	20
Floating point stalls:	179
WAW stalls:	0
Structural stalls:	0
Control stalls:	20
Trap stalls:	3
Total	223
Conditional Branches	
Total:	29
Tomados:	12
No tomados:	8
Instrucciones Load/Store	
Total:	23
Loads:	2
Stores:	21
Instrucciones de punto flotante	
Total:	27
Sumas:	9
Multiplicaciones:	10
Divisiones:	8
Traps	
Traps:	1

ESTADISTICAS para valor inicial 10 optimizado	
Total	
Nº de ciclos:	169
Nº de instrucciones ejecutadas (IDs):	103
Stalls	
RAW stalls:	15
LD stalls:	0
Branch/Jump stalls:	0
Floating point stalls:	15
WAW stalls:	0
Structural stalls:	0
Control stalls:	5
Trap stalls:	7
Total	68
Conditional Branches	
Total:	9
Tomados:	5
No tomados:	4
Instrucciones Load/Store	
Total:	22
Loads:	2
Stores:	20
Instrucciones de punto flotante	
Total:	22
Sumas:	9
Multiplicaciones:	9
Divisiones:	5
Traps	
Traps:	1

Conclusiones

1. Reducción en el número total de ciclos: El código optimizado reduce significativamente el número total de ciclos requeridos para ejecutar el programa, lo que indica una mejora en la eficiencia del código.

2. Reducción en el número de instrucciones ejecutadas: El código optimizado ejecuta menos instrucciones en comparación con el código no optimizado, lo que demuestra una optimización efectiva del flujo del programa.

3. Reducción en los stalls: El código optimizado tiene menos stalls en general, lo que indica una mejor gestión de los riesgos de dependencia de datos y de control.

4. Reducción en instrucciones de punto flotante: El código optimizado ejecuta menos instrucciones de punto flotante, lo que refleja una optimización en el uso de recursos de hardware, especialmente en operaciones de multiplicación y división.

5. Mejora en el rendimiento general: En general, el código optimizado muestra un mejor rendimiento en términos de ciclos de reloj, número de instrucciones ejecutadas y uso eficiente de recursos, lo que lo hace más adecuado para su implementación en hardware con las características especificadas.

Optimización del algoritmo 3x+1

El algoritmo "3x+1", también conocido como la conjetura de Collatz (nos referiremos con ese nombre al algoritmo para no confundirlo con la operación de cálculo 3x+1), es un algoritmo simple que opera sobre números enteros positivos. Se define de la siguiente manera:

1. Si el número x es par, divídelo por 2 (es decir, (x = x / 2)).
2. Si el número x es impar, multiplícalo por 3 y suma 1 (es decir, (x = 3x + 1)).

El algoritmo se repite hasta que el valor de x es igual a 1 (por lo que el algoritmo se realiza dentro de un bucle).

Para la realización de esta práctica, además de las instrucciones necesarias para la realización del algoritmo, se pide recopilar datos para realizar cálculos estadísticos, esto afecta al número de ciclos ya que se tienen que añadir las instrucciones para la recopilación de datos. Los datos que se recopilan durante la ejecución del algoritmo son:

- Sumatorio de todos los números de la secuencia
- Sumatorio con la cantidad de números que hay en la secuencia
- El número máximo de la secuencia.

Nuestro algoritmo de resolución ejecuta 4 tipos de sentencias distintas dependiendo del número de secuencia que se esté ejecutando en el algoritmo, nosotros hemos considerado tres tipos de números que coinciden con las etiquetas de nuestro programa: par, impar, par_mayor e impar_menor.

Para optimizar el algoritmo, no podemos hacer ninguna predicción, pero podemos afirmar las siguientes consideraciones sobre la conjetura de Collazt, que harán que nuestro código reduzca considerablemente el número de ciclos y de instrucciones:

- Se puede afirmar que cada vez se ejecuta la operación para números impares, es decir un número impar se multiplica por 3 y se suma 1, dará siempre un número par. Esta afirmación nos permite evitar tener que comprobar si el resultado de 3x+1 es un número par o impar para la siguiente interacción del algoritmo y pasar directamente a la ejecución de sentencias de pares.

- Para llegar al final se la secuencia, es decir el número actual de la secuencia es igual a 1, el número anterior previamente tiene que haber sido dividido entre 2, por lo que solo se puede llegar al resultado 1 si el número previo era par. Esto nos permite deducir que el algoritmo llegará al valor de 1 cuando se ejecuten las instrucciones para los pares (dividir entre 2). Esta afirmación nos permite evitar comprobar si se ha

llegado al final de la secuencia después de haber ejecutado las sentencias para un número impar (3x+1).

- Cuando un número se divide entre 2, el 50% de las veces da como resultado un número impar, y el otro 50% da como resultado un número par. Esta afirmación nos permite tener que evitar comprobar si se ha llegado al final de la secuencia cuando el número que se ha dividido entre 2 es par, ya que el número 1 es impar.

- Se puede afirmar que, el número máximo de la secuencia solo se podrá encontrar después de ejecutar las sentencias para un número impar (3x+1) ya que los pares se dividen entre dos por lo que el valor más alto se encontrará después de ejecutar 3x+1. Esta afirmación nos permite evitar tener que comprobar si el número calculado en las sentencias de los pares es el mayor de la secuencia.

- El número más pequeño después de ejecutar la secuencia para números impares (3x+1) es 4, por lo que los números que se calculen al ejecutar 3x+1 los llamaremos par_mayor. Para estos números podemos evitar tener que comprobar si se ha llegado al final de la secuencia o no.

- Puede ocurrir que después de ejecutar la secuencia par (x/2) salga otro número par, y como previamente a esta secuencia se habrá ejecutado antes par_mayor el número que teníamos después de ejecutar la secuencia de impares ha sido dividido entre 4, o incluso puede ser dividido más veces entre 2. A estos números los llamaremos impar_menor, y en esa secuencia para números impares, no comprobaremos si el número calculado es el mayor de la secuencia ya que en ningún caso podrá serlo, ya que (¾)

Estas afirmaciones permiten que el código ejecute menos instrucciones y menos instrucciones de tipo condicional y salto lo que nos reduce considerablemente el número de instrucciones que son abortadas por las instrucciones condicionales.

Optimización estadísticas finales

Programa NO Optimizado

Se va a explicar el programa NO optimizado en su parte de estadísticas y lo que se ha hecho para poder bajar los ciclos en esa zona de código.

Lo primero que vemos es que hay muchas multiplicaciones y divisiones en el código, que hace que se aumenten muchos ciclos, ya que estas operaciones son más complejas que la suma y la resta en todos los aspectos. La implementación en el hardware suele requerir circuitos más complejos, puede haber dependencias de datos, ya que estas instrucciones pueden hacer cuentas que no vemos en el uso de la instrucción, por lo que puede que haya que esperar por una cuenta mientras se hace la división y multiplicación.

Podemos ver el código de la parte no optimizada:

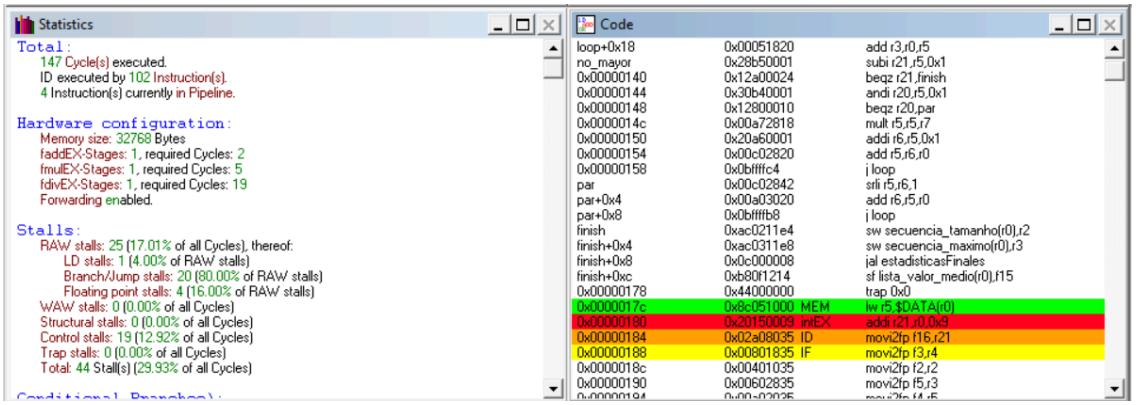
```
105     estadisticasFinales:
106
107             lw              r5, valor_inicial
108             addi    r21, r0, 9
109
110             movi2fp f16, r21
111             movi2fp f3, r4  ;       suma_secuencia
112             movi2fp f2, r2  ;       secuencia_tamanho vT
113             movi2fp f5, r3  ;       secuencia_maximo
114             movi2fp f4, r5  ;       valor inicial
115
116             ;movi2fp        f24, r9
117
118
119
120             ;cvti2f f3, f3
121             ;Es la suma sola
122             ;Tenemos secuencia_valor_medio
123             divf    f1, f3, f2              ;vmed
124
125
126
127             cvti2f  f3, f3
128             cvti2f  f2, f2
129             cvti2f  f5, f5
130             cvti2f  f4, f4
131             cvti2f  f16,f16
132
133             ; vamos sumandole 4 a lista y añadiendole elementos
134             add             r20, r0, lista
135
136
137             sf              secuencia_valor_medio, f1
138
```

```
140         ; vIni*vT       0
141         multf   f6, f4, f2
142         sf              0(r20), f6
143         addf    f15, f15, f6
144         addi    r20, r20, 4
145         ; vMax*vT       4
146         multf   f7, f5, f2
147         sf              0(r20), f7
148         addf    f15, f15, f7
149         addi    r20, r20, 4
150         ; vMed*vT       8
151         multf   f8, f1, f2
152         sf              0(r20), f8
153         addf    f15, f15, f8
154         addi    r20, r20, 4
155         ; (vIni/vMax)*vT        12
156         divf    f9, f4, f5
157         multf   f9, f9, f2
158         sf              0(r20), f9
159         addf    f15, f15, f9
160         addi    r20, r20, 4
161         ; (vIni/vMed)*vT        16
162         divf    f10, f4, f1
163         multf   f10, f10, f2
164         sf              0(r20), f10
165         addf    f15, f15, f10
166         addi    r20, r20, 4
```

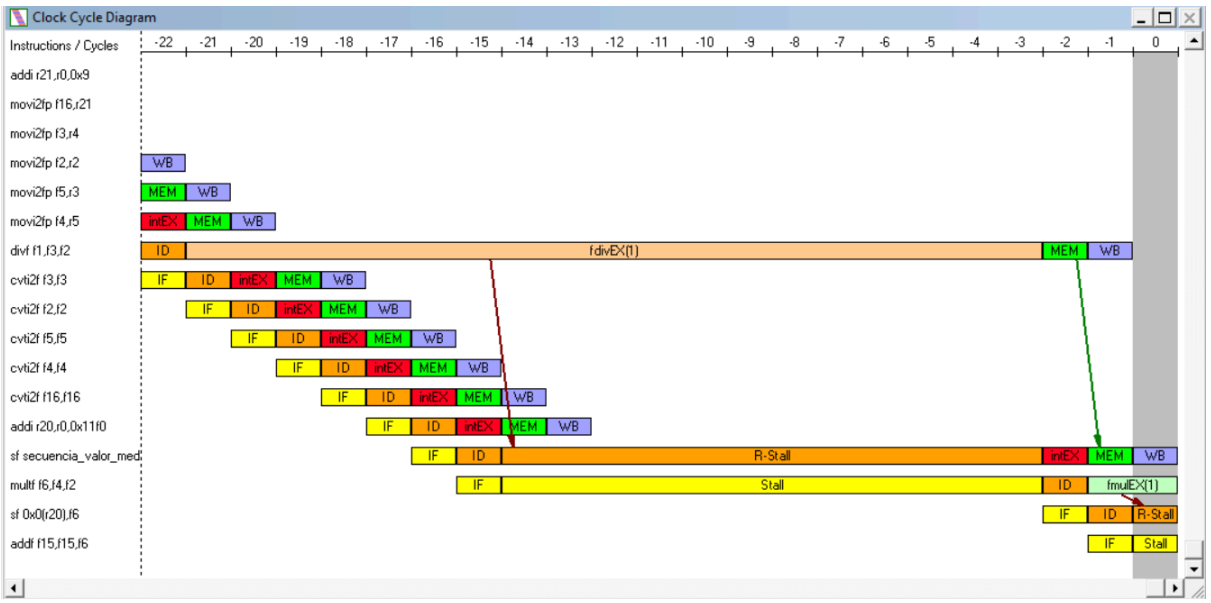
```
; (vMax/vIni)*vT        20
divf    f11, f5, f4
multf   f11, f11, f2
sf              0(r20), f11
addf    f15, f15, f11
addi    r20, r20, 4
; (vMax/vMed)*vT        24
divf    f12, f5, f1
multf   f12, f12, f2
sf              0(r20), f12
addf    f15, f15, f12
addi    r20, r20, 4
; (vMed/vIni)*vT        28
divf    f13, f1, f4
multf   f13, f13, f2
sf              0(r20), f13
addf    f15, f15, f13
addi    r20, r20, 4
; (vMed/vMax)*vT        32
divf    f14, f1, f5
multf   f14, f14, f2
sf              0(r20), f14
addf    f15, f15, f14


divf    f15, f15, f16
sf              lista_valor_medio, f15
```

Vemos lo que comentábamos antes, el código tiene muchas divisiones y multiplicaciones, junto con guardados en memoria y sumas para aumentar el puntero a la memoria.

Con este código hemos tenido los siguientes ciclos y raw-stalls:
En el conjunto global hemos tenido 387 ciclos con 10 números, ya que no es un bucle esa parte del código solo lo ejecutará una vez siendo siempre los mismos ciclos.
223 stalls

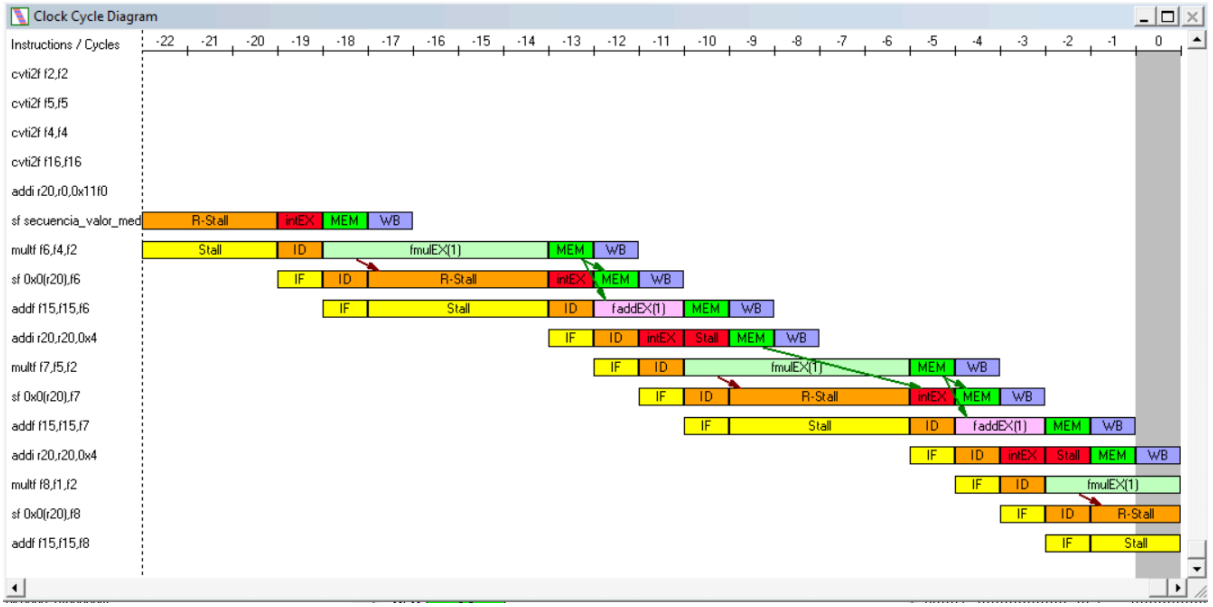


Estos ciclos son los que tenemos nada más llegar a la parte de las estadísticas aproximadamente, entonces, con el total de ciclos que son 387 - 147 = 240.
240 son los ciclos que consumen la parte de las estadísticas, con unos stalls de 223 - 44= 179.
Vemos el gran número de ciclos que consume la parte de las estadísticas, todo por el número de stalls que hace las multiplicaciones y divisiones.
Aquí podemos verlo:



La división ha consumido 22 ciclos, y teniendo en cuenta que hay en el código de las estadísticas 9 divisiones hace que haya grandes dependencias de datos (RAW STALLS, significa que tiene que esperar por un dato que todavía no se ha generado).

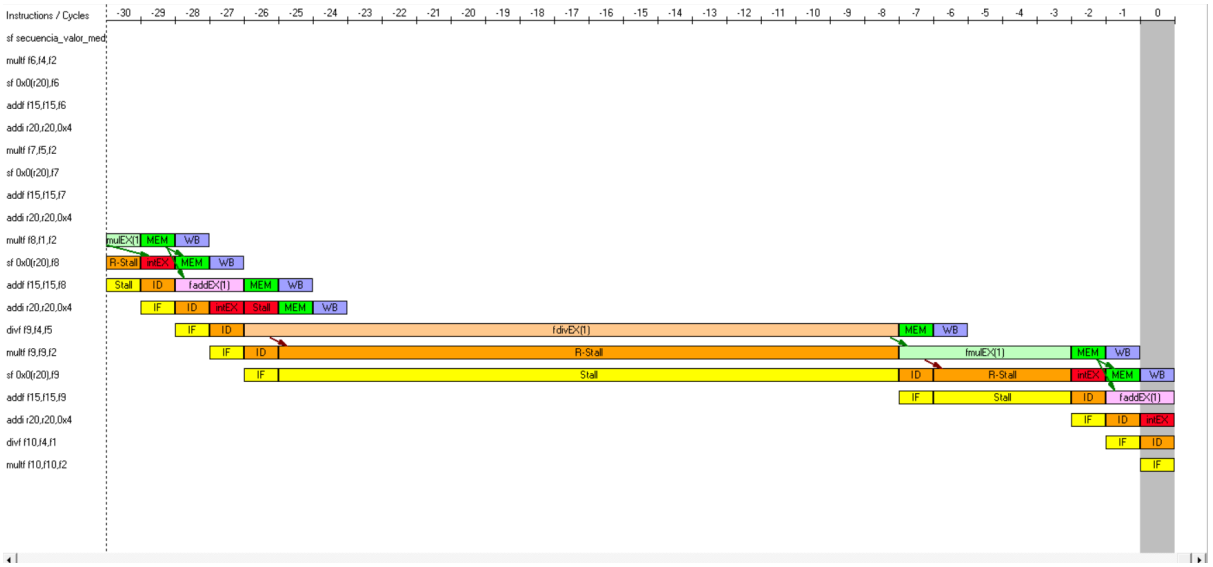
Como vemos en la imagen, la instrucción de escritura en memoria(sf) tiene que esperar 13 ciclos hasta que puede iniciar su búsqueda de registro (ID) porque ese registro está siendo



La multiplicación no son tantos ciclos de espera como ocurre en la división, son únicamente 4 ciclos, pero el que haya otras 9 operaciones de multiplicación hace que el código se vuelva a ralentizar.
Vuelve a pasar los Raw-Stalls, ya que como son tantos ciclos de espera, las operaciones siguientes necesitan los datos para poder introducir en memoria, lo que hace que se quede esperando por el dato

Por último comentar las operaciones que están generando mas stalls, como necesitamos hacer una operación de división y multiplicación conjuntas, ya que es un mismo dato, hace

que la multiplicación espere por la división, y la escritura en memoria espere por la multiplicación, generando lo siguiente:



Generamos en dos operaciones 26 stalls solo para poder escribir en un array una operación.

Programa Optimizado

En el programa optimizado se han conseguido bajar bastantes ciclos.
Este proceso ha sido complicado ya que al no haber instrucciones distintas, siendo en casi todo el código instrucciones de multiplicación, división, suma y guardar en memoria repetidas periódicamente, no se podía jugar mucho moviendo instrucciones, por lo que se optó por intentar quitar instrucciones. Si conseguimos quitar una división, son 22 stalls menos, por lo que menos ciclos, por lo que más rápido el programa.

Programa Optimizado:

```
estadisticasFinales:
        ; Cargamos los valores de entrada desde la memoria

        lf              f4, valor_inicial               ; Cargamos el valor inicial de la secuencia en f2

        movi2fp f2, r2                  ; Convertimos el tamaño de la secuencia a punto flotante
        movi2fp f3, r4                  ; Convertimos el máximo de la secuencia a punto flotante

        cvti2f  f2, f2
        cvti2f  f3, f3
        cvti2f  f4, f4

        divf    f1, f3, f2               ; Dividimos el valor medio de la secuencia por el tamaño de la secuencia y lo almacenamos en f1
        multf   f6, f4, f2 ; vIni*vT

        movi2fp f5, r3                  ; Convertimos el máximo de la secuencia a punto flotante
        cvti2f  f5, f5
        sw              secuencia_tamanho, r2     ; Cargamos el tamaño de la secuencia desde la memoria a r6
        sw              secuencia_maximo, r3      ; Cargamos el máximo de la secuencia desde la memoria a r3

        multf   f7, f5, f2 ; vMax*vT


        ; f3 = vmed *v t
        cvti2f  f15, f15
        sf              lista, f6 ; Guardamos vIni*vT en la lista
        addf    f15, f15, f6 ; Sumamos el valor a la suma total

        sf              lista+0x4, f7 ; Guardamos vMax*vT en la lista
        addf    f15, f15, f7 ; Sumamos el valor a la suma total
```

```
        divf    f20, f17, f1 ; 1/vMed



        multf   f11, f19, f7 ; (1/vIni)*vMax*vT
        sf              lista+0x20, f14 ; Guardamos (vMed/vMax)*vT en la lista
        addf    f15, f15, f14 ; Sumamos el valor a la suma total

        multf   f13, f19, f3 ; (1/vIni)*vMed*vT
        sf              lista+0x14, f11 ; Guardamos (vMax/vIni)*vT en la lista
        addf    f15, f15, f11 ; Sumamos el valor a la suma total

        sf              lista+0x1c, f13 ; Guardamos (vMed/vIni)*vT en la lista
        addf    f15, f15, f13 ; Sumamos el valor a la suma total

        multf   f10, f20, f6 ; (1/vMed)*vIni*vT


        multf   f12, f20, f7 ; (1/vMed)*vMax*vT
        addf    f15, f15, f10 ; Sumamos el valor a la suma total
        sf              lista+0x10, f10 ; Guardamos (vi/vMed)*vT en la lista
        addf    f15, f15, f12 ; Sumamos el valor a la suma total
        sf              lista+0x18, f12 ; Guardamos (vMax/vMed)*vT en la lista


        multf   f22, f15, f16 ; Dividimos el valor medio de la secuencia por el tamaño de la secuencia y lo almacenamos en f1


        sf              lista_valor_medio, f22 ; Guardamos el valor medio de la secuencia en la lista


        trap 0


        ; Finalizamos el programa
        ;jalr   r31
```

En el código nos podemos fijar que se han movido instrucciones para poder así conseguir menos ciclos, ya que si no tiene que esperar por stalls una instrucción, esto hará que no aumente en ciclos.
Luego se puede observar cómo se han guardado en registros instrucciones que eran usadas varias veces y que se podían hacer únicamente una vez, quitando así operaciones y usando el registro en vez de hacer la operación varias veces.

calculado en la división (divf f1, f3,f2), lo que hace que tenga que esperar, implicando una dependencia de datos.
Luego explicaremos cómo se ha solucionado.

```
165             divf    f18, f17, f5 ; 1/vMax
166
167             divf    f19, f17, f4 ; 1/vIni
168
169             multf   f9, f18, f6 ; (1/vMax)*vIni*vT
170             sf              lista+0x8, f3 ; Guardamos vMed*vT en la lista
171             addf    f15, f15, f3 ; Sumamos el valor a la suma total
172             sf              secuencia_valor_medio, f1       ; Cargamos el valor medio de la secuencia desde la memoria a f1
173
174             multf   f14, f18, f3 ; (1/vMax)*vMed*vT
175             sf              lista+0xc, f9 ; Guardamos (vi/vMax)*vT en la lista
176             addf    f15, f15, f9 ; Sumamos el valor a la suma total
177
178
179             divf    f20, f17, f1 ; 1/vMed
```

Vemos lo comentado anteriormente, se han guardado en registros las divisiones, haciendo así que solo se ejecuten 1 vez cada una, y no varias veces, en el código podemos ver 4 instrucciones de división, hemos bajado 5 instrucciones, con la bajada de ciclos por instrucción que eso conlleva (22/inst).
Hecho eso, aunque haya menos instrucciones de división, las de la multiplicación tendrán que aumentar, pero es mejor usar una multiplicación que una división ya que son 4 ciclos la multiplicación y 22 la división.

```
multf   f11, f19, f7 ; (1/vIni)*vMax*vT
sf              lista+0x20, f14 ; Guardamos (vMed/vMax)*vT en la lista
addf    f15, f15, f14 ; Sumamos el valor a la suma total
```

Vemos lo dicho anteriormente, en f19 tenemos guardado (1/vIni) y en f7 (vMAX*Vt), una multiplicación en vez de repetir varias veces las mismas operaciones.

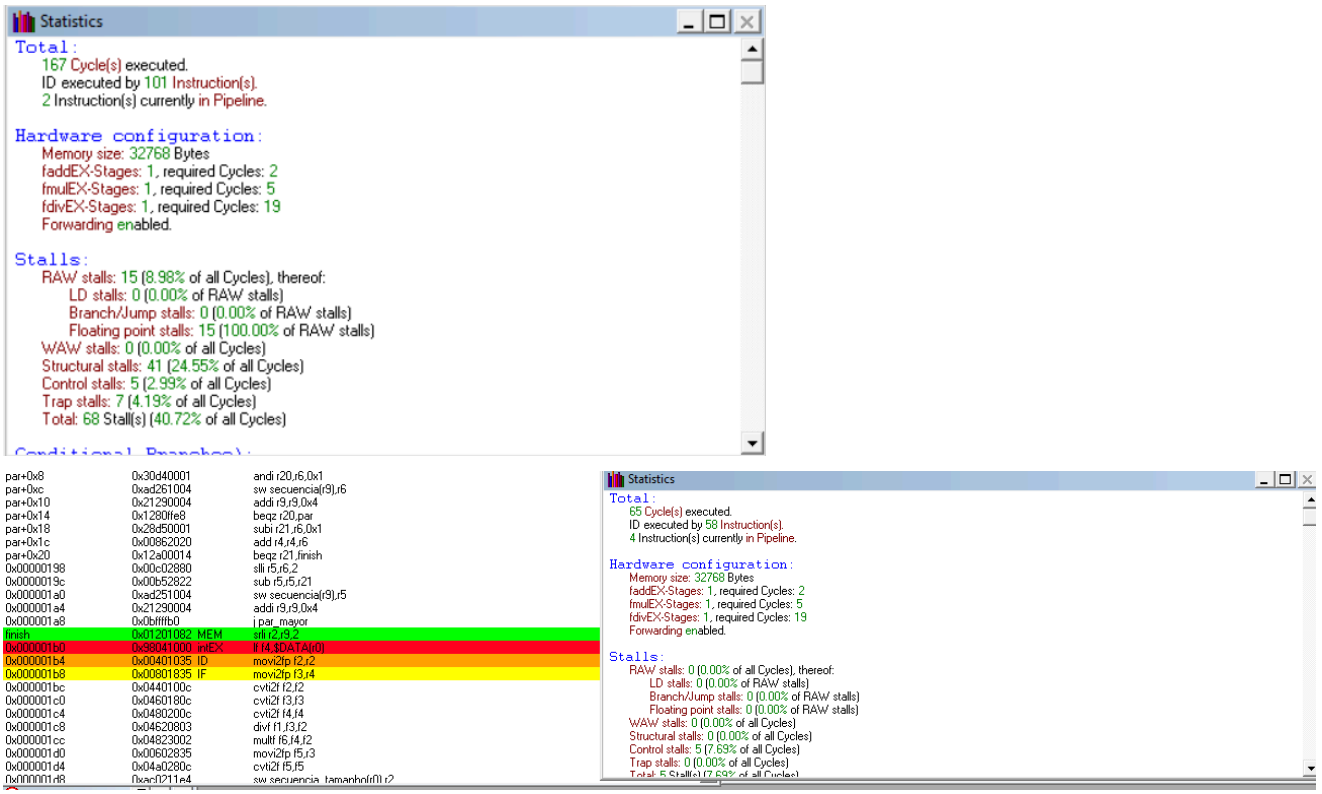
También en ese fragmento de código vemos una organización de código menor, aunque sea menos intuitivo que el otro código, vemos como el número de ciclos bajan, ya que la colocación que se ha hecho es para que no haya tantos stalls. Se podrían colocar las divisiones juntas, pero haría que “EX” en el pipeline, quedase esperando por la segunda división, ya que no habría terminado de ejecutar la primera división y haría varios ciclos de stalls.

Otro elemento a resaltar es la variación de como escribimos en memoria, antes usábamos sf y además un registro donde guardábamos el puntero donde había que guardar en la memoria. Hemos suprimido la suma poniendo directamente en que parte de la memoria se tiene que escribir.

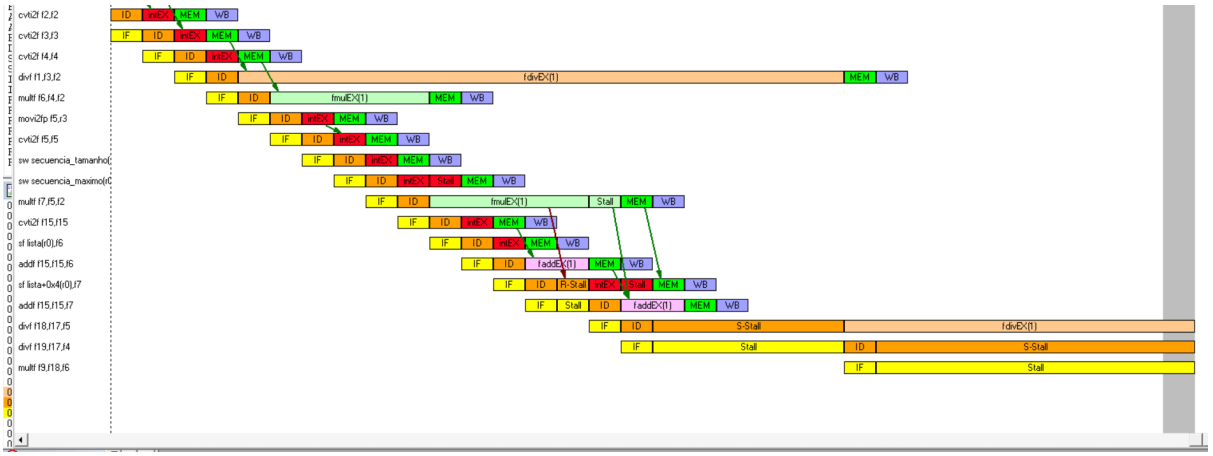
```
multf   f13, f19, f3 ; (1/vIni)*vMed*vT
sf              lista+0x14, f11 ; Guardamos (vMax/vIni)*vT en la lista
addf    f15, f15, f11 ; Sumamos el valor a la suma total

sf              lista+0x1c, f13 ; Guardamos (vMed/vIni)*vT en la lista
addf    f15, f15, f13 ; Sumamos el valor a la suma total
```

Vemos que se usa lista+0x14, es la memoria donde se va a guardar

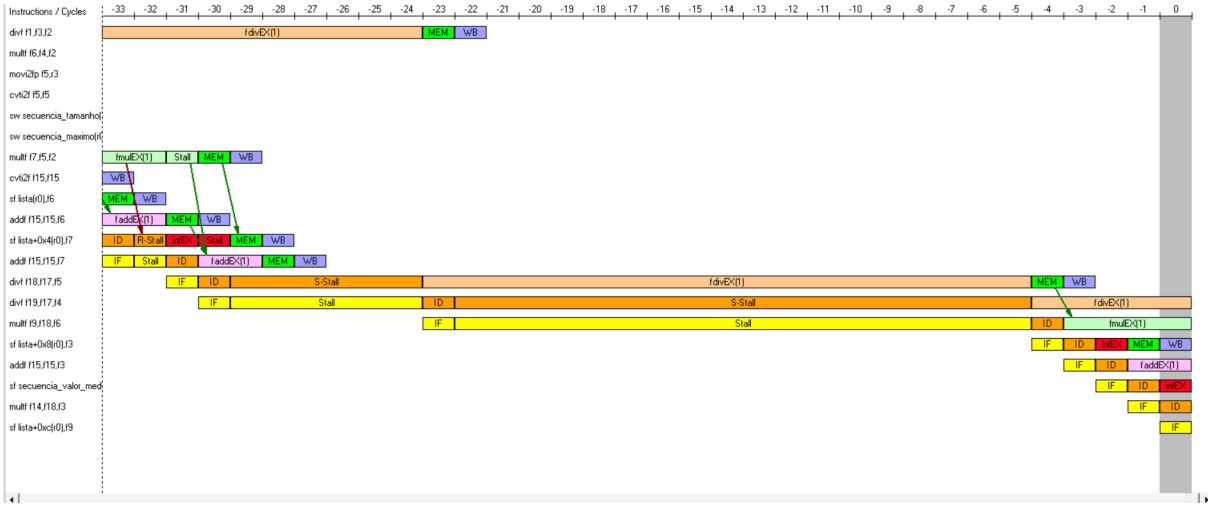
Vemos que lo explicado tiene fundamento, viendo el número de ciclos que ha disminuido:5



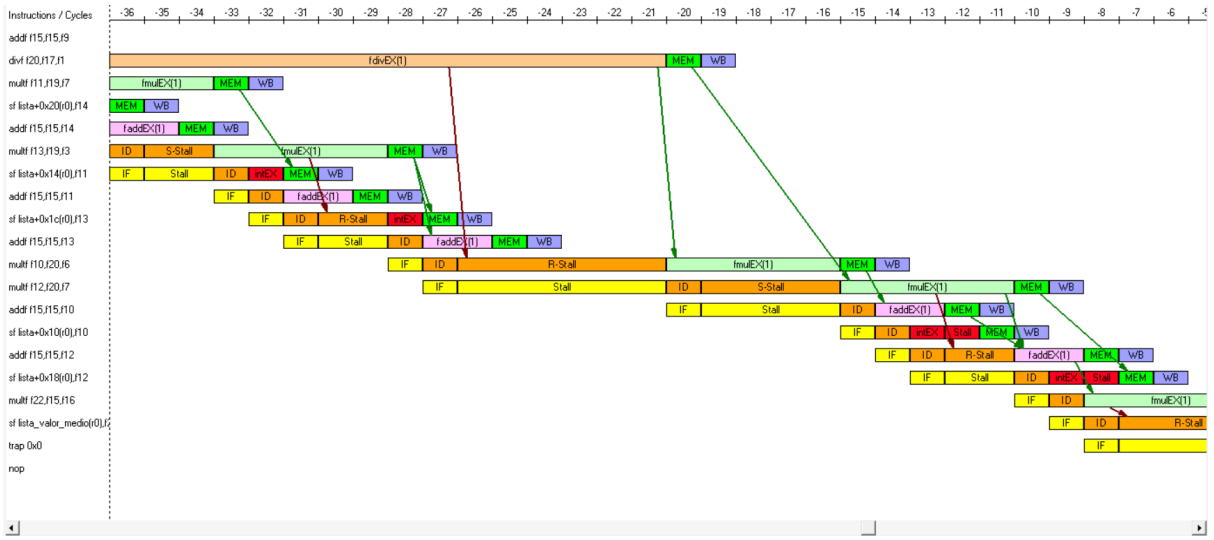
El número de ciclos antes de entrar en las estadísticas, por lo que 167-65=102. Se ha conseguido bajar de 240 ciclos a 102, 138 ciclos de diferencia. Al ser en tan pocas instrucciones, se ha aumentado bastante el rendimiento.



Aunque hayamos conseguido bajar tantos ciclos, hay stalls que no son posibles de quitar, como dijimos anteriormente, al haber tantas operaciones juntas, por mucho que se quiera poder espaciar entre ellas para que no haya stalls es imposible, y es lo que ocurre aquí. Tenemos varias instrucciones entre la primera y segunda división, pero al tener que hacer las operaciones es imposible dejar más espacio.



Entre la segunda y tercera división ocurre lo mismo, no tenemos otro sitio para poder ejecutarla, y la ocurrencia de stalls es inevitable. Vemos que ya solo con esas tres divisiones hay 22 structured stalls, y en todo el programa hay 41. Esto se denomina así ya que existen cuando se forman por las limitaciones de la estructura física del procesador.



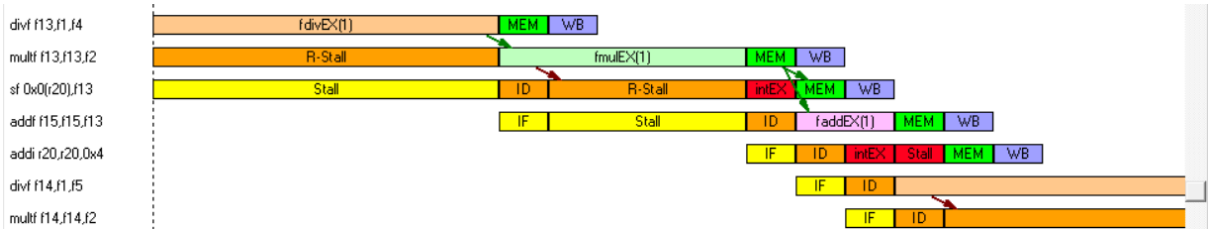
Vemos que ocurre lo mismo que en el programa no optimizado, el que haya tantas operaciones juntas hace que obligatoriamente haya stalls, por eso se denomina S-stalls, no hay forma de evitar el conflicto porque el procesador es limitado y no puede evitar esas esperas. Luego podemos ver algún R-stall, que existen entre multiplicaciones, guardados en memoria y la división (divf f20, f17,f1).

Optimización raw stalls y más consideraciones

¿Qué son los RAW Stalls?

Los "RAW stalls" son un tipo de obstáculo en la ejecución de instrucciones en un procesador. RAW es un acrónimo de "Read After Write", que significa "leer después de escribir". Este tipo de stall ocurre cuando una instrucción necesita leer un operando que aún no ha sido escrito por una instrucción previa en el pipeline.

Para entenderlo mejor, consideremos el siguiente ejemplo de código del programa no optimizado:



En un procesador con pipeline, las instrucciones se ejecutan en varias etapas secuenciales, lo que permite que múltiples instrucciones estén en diferentes etapas de ejecución simultáneamente. Sin embargo, incluso en un pipeline, algunas instrucciones pueden tomar más tiempo que otras en completarse debido a la naturaleza de la operación que realizan.

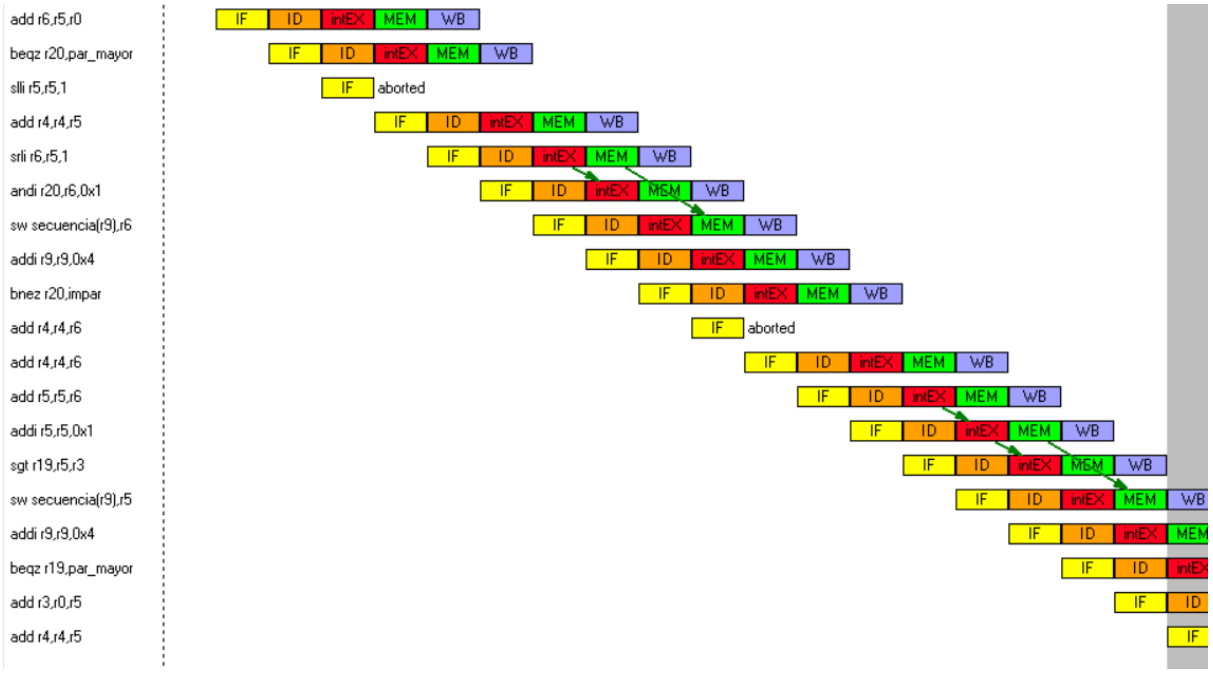
Por ejemplo, consideremos la multiplicación que tenemos en la imagen y que tarda 4 ciclos de reloj en completarse. Mientras esta multiplicación está en curso, otras instrucciones pueden avanzar a través del pipeline y pasar por etapas como decodificación, ejecución y escritura de resultados. Sin embargo, en la etapa de ejecución del pipeline, donde ocurre la multiplicación, el procesador se detiene momentáneamente porque solo hay una unidad de ejecución disponible y está ocupada por la multiplicación. Esto significa que no se pueden ejecutar otras instrucciones que requieran la misma unidad de ejecución hasta que la multiplicación se complete y la unidad esté disponible nuevamente.

Por lo tanto, aunque otras instrucciones pueden continuar avanzando a través del pipeline y pasar por etapas posteriores, la instrucción de multiplicación puede causar un "stall" en la etapa de ejecución, lo que significa que otras instrucciones que necesiten la misma unidad de ejecución deben esperar. Este fenómeno se conoce como "stall" o "parada" y puede afectar el rendimiento del procesador al reducir la eficiencia y el grado de paralelismo que se puede lograr.

Mejoras Números RAW-STALLS

Para mejorar los stalls, se hicieron varios cambios en el algoritmo, nos dimos cuenta que las multiplicaciones y divisiones tenían gran peso, haciendo que las instrucciones siguientes tuvieran que esperar varios ciclos, esto se cambió intentando disminuir el número de divisiones y multiplicaciones, y estas operaciones hacerlas a nivel de bit. Esto se puedo hacer en el algoritmo, en la parte de estadísticas no, ya que eran otro tipo de operaciones.

Como el algoritmo era, si es impar, multiplicar por 3 + 1, se vió que podíamos multiplicar por 2 y sumar el elemento + 1, pero la multiplicación por 2 hacerla a nivel de Bit, es decir, desplazando un bit a la derecha. Esto hizo que en vez de pasar de 4 ciclos y uno la suma, que se hiciese en 2 ciclos.

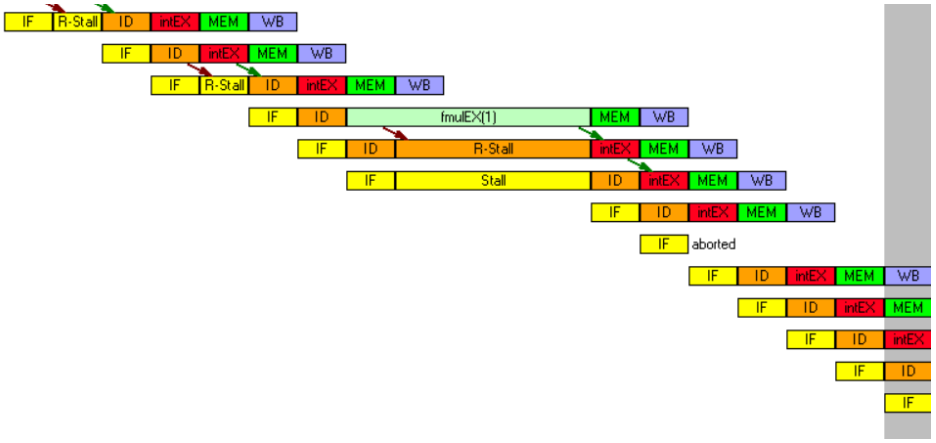


Vemos como en la parte del loop se ha conseguido que no haya ningún stall, ya que al hacer las operaciones a nivel de bit, es solo un ciclo lo que tarda en mover el bit, mientras que la multiplicación tarda más.

```
55      beqz    r20, par_mayor        ; Salta a 'par' si A[n-1] es par
56
57      sll     r5, r5, 1
58
59  impar:
```

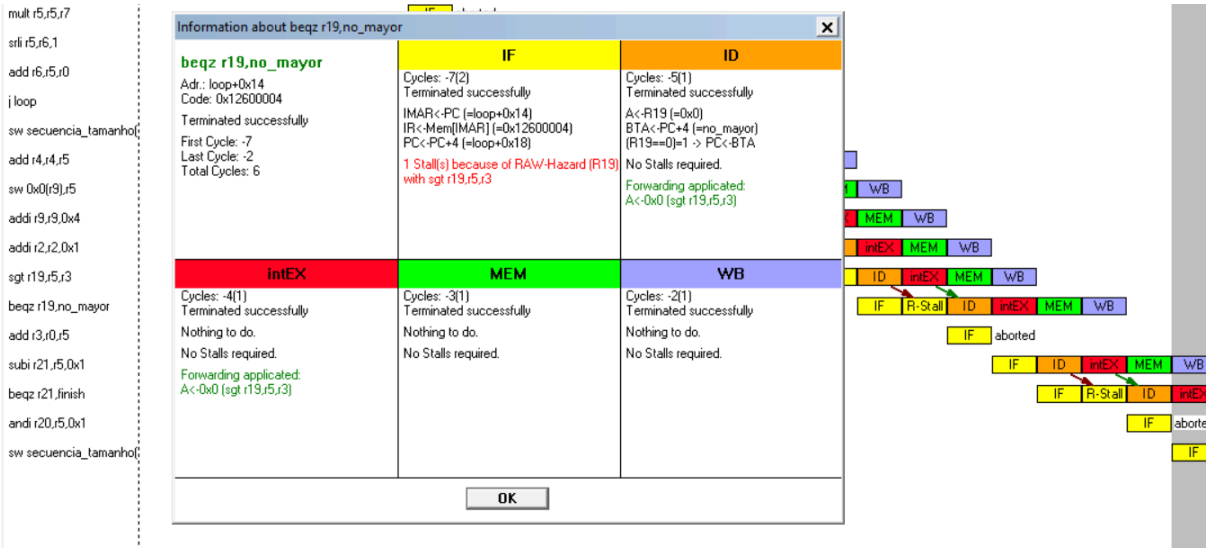
En el código se puede ver que se introduce en r5 el mover un bit a la derecha. También lo que se ha hecho es guardar el valor anterior, y hay algunos casos, que como se divide entre dos cuando es par, si el siguiente es impar, no es necesario usar la instrucción srl, ya que el valor anterior es el mismo, y lo tenemos guardado en un registro.

En el código sin optimizar podemos ver lo perjudiciales que son las ordenes multiplicar y dividir. Con una sola operación, hemos parado el sistema 4 ciclos.



Stall en:

```
;Añadimos el valor maximo   r12 es 1 si r3 > secuencia_maximo
;Comprobamos valor
sgt     r19, r5, r3
beqz    r19, no_mayor
```



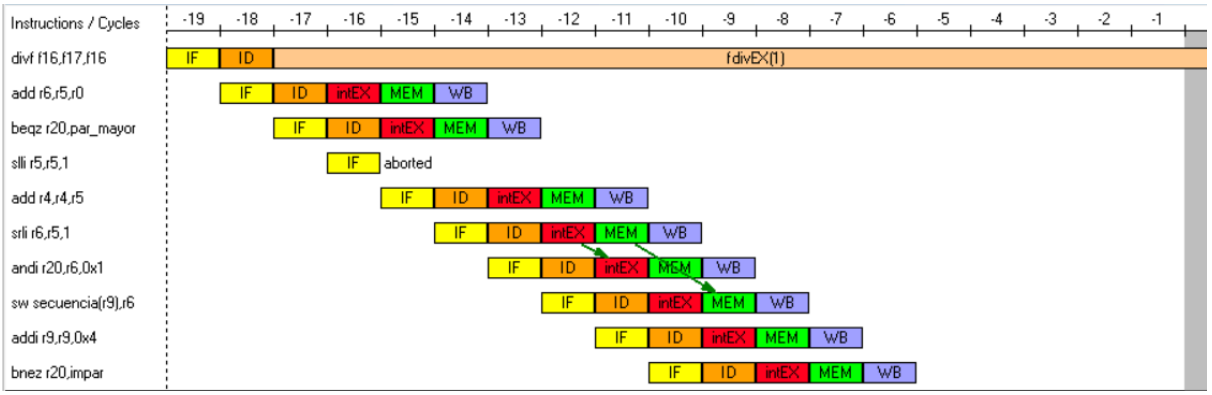
Esta operación ha creado en el operador IF un stall, significa que no estaba disponible el dato, entonces ha tenido que hacer una parada.

En el optimizado hemos solucionado este problema aplicado lo siguiente:

```
sgt  r19, r5, r3                ; Comparar r3 con secuencia_maximo

sw   secuencia(r9), r5                  ; Almacenar el nuevo elemento en la secuencia
addi  r9, r9, 4                ; Calcular la dirección del nuevo elemento

beqz r19, par_mayor               ; Saltar si r3 no es mayor que secuencia_maximo
add  r3, r0, r5                ; Actualizar secuencia_maximo con r3
```

Vemos como únicamente con una reordenación de código hemos conseguido no hacer un stall, por lo que el sistema no se para. Solo con esta pequeña acción, se consigue que el programa vaya más rápido.

Podemos ver esta división, en el loop no se usa en ningún momento, pero como estamos manipulando enteros, y eso es punto flotante conseguimos que se haga la operación sin parar el pipeline, ya que usa el EX float, no el de entero. Este registro donde metemos el valor (1/9) lo usamos en las estadísticas, para dividir el valor medio, si lo hiciésemos en esa parte de código nos crearía un stall grande ya que como estaríamos ejecutando todo con floats ya que al se lista en gran parte divisiones y multiplicaciones haría que se parase el programa. Como en esta parte son enteros, podemos ejecutarlo y que no haya parada.



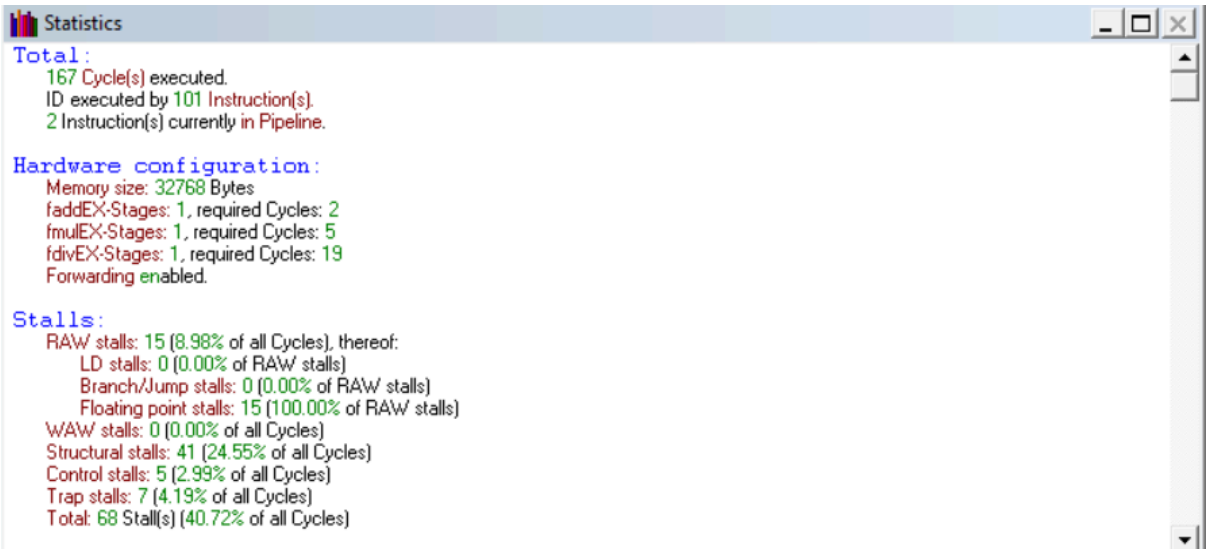
El poder decir que no haya stalls en un bucle, permite que a cuantos más números haya, más se note la diferencia entre ejecuciones que tienen stalls y ejecuciones que no, ya que con el número 10, al ser un número bajo, no se nota mucho, por si se usase el 100, un programa sin paradas permite ir mucho más rápido, que uno que tiene que estar esperando por ejecuciones de instrucciones por encima suyo.

Ejecución de programa con 10

- **Sin Optimizar**

```
Total:
    384 Cycle(s) executed.
    ID executed by 160 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: 200 (52.08% of all Cycles), thereof:
        LD stalls: 1 (0.50% of RAW stalls)
        Branch/Jump stalls: 20 (10.00% of RAW stalls)
        Floating point stalls: 179 (89.50% of RAW stalls)
    WAW stalls: 0 (0.00% of all Cycles)
    Structural stalls: 0 (0.00% of all Cycles)
    Control stalls: 20 (5.21% of all Cycles)
    Trap stalls: 3 (0.78% of all Cycles)
    Total: 223 Stall(s) (58.07% of all Cycles)
```

- **Optimizado**

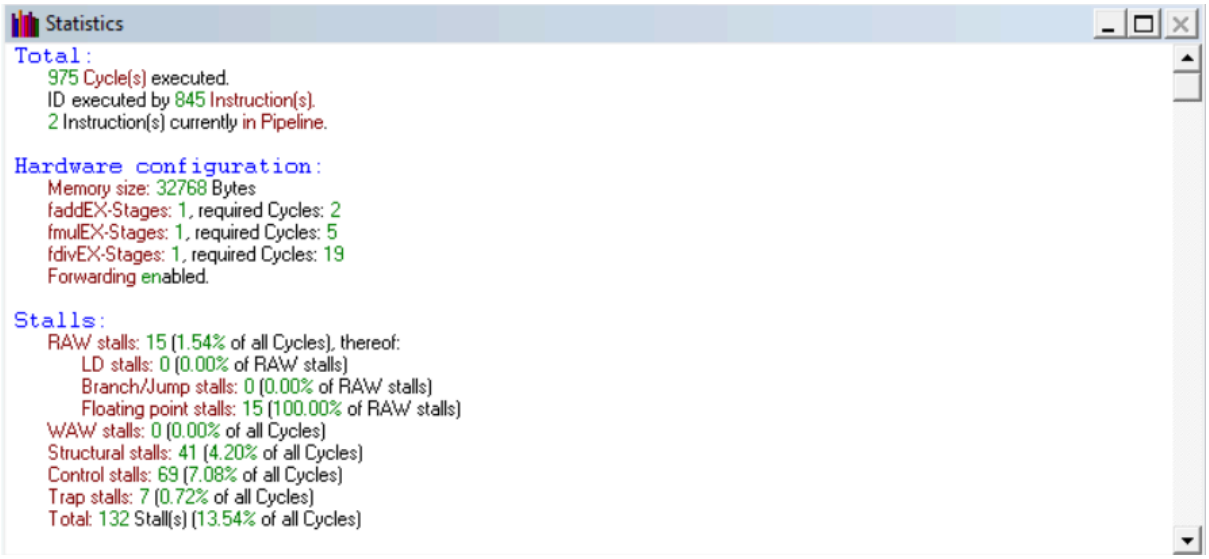


Con valor 10 podemos observar que tampoco hay mucha diferencia entre valores, al final es un valor pequeño y no hay mucho que hacer en el bucle. Al final lo que más hace en un valor tan pequeño son las estadísticas ya que el bucle no permite hacer una optimización tan clara. Si nos acordamos de los valores de ciclos de las estadísticas, el programa sin optimizar eran 240 ciclos y el optimizado 102, como vemos en la parte del loop no hace mucha diferencia entre el optimizado y el no, ya que no hace muchas vueltas y no hay grandes stalls en el no optimizado para hacer grandes paradas.

Ejecución de programa con 97

- **Sin Optimizar**

- **Optimizado**



En esta operación, al haber un número más alto, hará que tome gran importancia el bucle, ya que se ha conseguido bajar de 1000 ciclos, mientras que en el otro programa esta en 2680. Esto hace que tome gran importancia el bucle, cuanto más se logre optimizar y se ponga un número más grande, más diferencia habrá.