Práctica 2: Análisis de segmentación

Arquitectura de Sistemas Integrados

Alejandro Solá
asola01@ucm.es

Pablo Suárez
pasuar03@ucm.es

Febrero 2023

Índice

1. Introducción

En esta segunda práctica nos hemos centrado en el análisis del número de ciclos, del número de instrucciones y hemos realizado pequeños cálculos de **CPIs**, con el fin de analizar y comparar el impacto generado en el **SoC** por parte de una implementación, frente a otra distinta. Para ello, se han construido tablas y gráficas que nos han permitido entender mejor las métricas obtenidas en cada uno de los lanzamientos del código.

Como ya estudiamos en teoría, la aplicabilidad de técnicas de optimización a nivel de software dependen directamente del código implementado, por ello, estas no resultan efectivas en el 100 % de los casos, y esto es lo que hemos estudiado en la segunda sesión de laboratorio.

Por otra parte, hemos analizado el impacto del uso de variables de tipo `float` en las operaciones realizadas por el procesador, relacionando el hardware disponible en el **SoC** y analizando el código ensamblador producido.

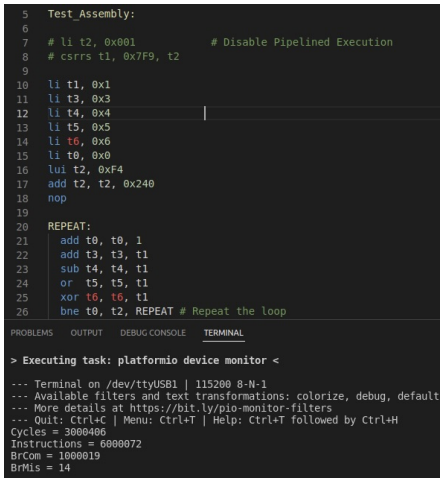
2. Sesión 1

Al comenzar esta primera sesión, hemos cargado las capas que estudiamos en la práctica anterior, para ello hemos incluido y cargado el `bitstream` correspondiente, luego hemos activado el puerto serie, para permitir una correcta conexión entre el sistema embebido y el ordenador desde el cual trabajamos. Tras configurar correctamente el entorno de trabajo para esta primera sesión, hemos abierto en `Visual Studio Code` el proyecto: `HWCounters_Example`, una vez cargado el proyecto, hemos comprobado que el resultado al correr el código es igual al mostrado en el enunciado de esta práctica.

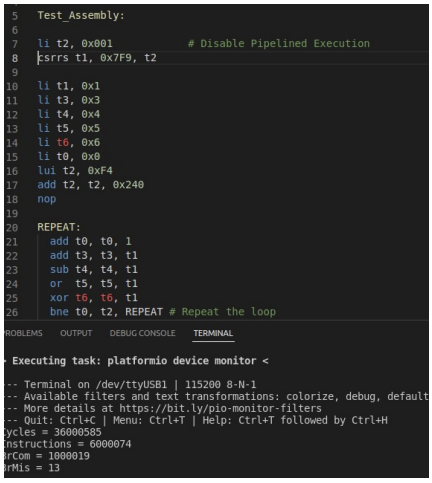
2.1. Entregable 1

¿Qué le hace pensar el CPI obtenido? Modifique las líneas 7 y 8 del `test_Assembly.S` [1], ejecutando las líneas comentadas y vuelva a medir el CPI, ¿qué está ocurriendo?

Los resultados obtenidos en este apartado son los siguientes:



(a) **Métricas** con lineas 7 y 8 comentadas



(b) **Métricas** con lineas 7 y 8 sin comentar

Figura 1: **Métricas** obtenidas tras la ejecución del `test_Assembly.S`

Los **CPIs** resultantes son los siguientes:

- **CPI_con_pipe** ≈ 0.5
- **CPI_sin_pipe** ≈ 6

Vemos como la diferencia entre los dos resultados obtenidos es abismal, esto se debe a la naturaleza del código, ya que el código del archivo `test_Assembly.S` no está diseñado para correrse en arquitecturas como

[1] Des-comentando las lineas 7 y 8 del `test_Assembly.S` conseguimos eliminar la ejecución del código con pipeline.

la del sistema empotrado que estamos utilizando en el laboratorio.

Cabe destacar que a nivel de ciclos si que vemos un aumento notable cuando descomentamos las siguientes lineas:

```
li t2, 0x001
csrrs t1, 0x7F9, t2
```

En realidad, al descomentar las lineas anteriores estamos desactivando la ejecución que permite el *issue* de dos instrucciones, la **ALU** secundaria y la ejecución por *pipeline*, este aumento tiene sentido, porque al no tener cortocircuito ni una segunda **ALU**, la lectura de registros en nuevas instrucciones se retrasa, algo que aplica a todas las instrucciones del código produciendo un aumento de ciclos.

Una vez analizado el rendimiento del código ensamblador procedemos a estudiar el rendimiento del filtro FIR.

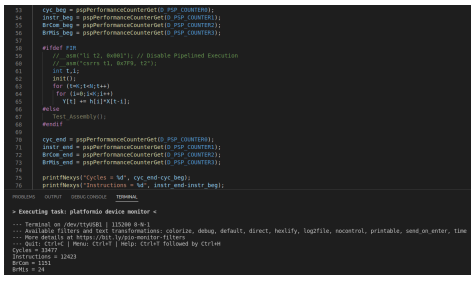
2.2. Entregable 2

Muestre el número de instrucciones y ciclos del filtro, con y sin incluir la inicialización. A la vista de la arquitectura del SweRV EH1, realice una estimación siguiendo la metodología vista en clase. ¿Existe algún tipo de desviación con respecto al modelo teórico del RISCV? ¿Y con respecto al procesador MIPS? Realice una gráfica con los diferentes CPIs.

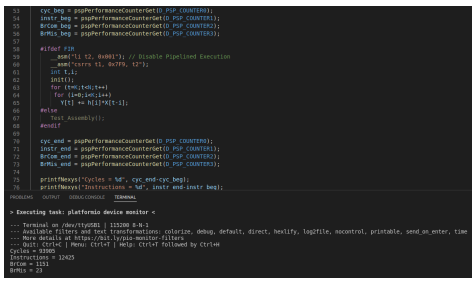
Para este apartado, hemos tenido en cuenta las cuatro combinaciones posibles:

- Ejecución del filtro **CON** pipeline y **CON** `init()`.
- Ejecución del filtro **CON** pipeline y **SIN** `init()`.
- Ejecución del filtro **SIN** pipeline y **CON** `init()`.
- Ejecución del filtro **SIN** pipeline y **SIN** `init()`.

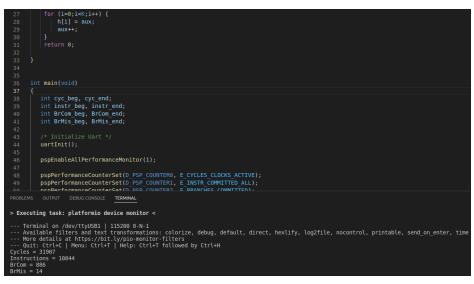
Véanse a continuación las métricas obtenidas tras la ejecución del filtro:



(a) `CPI` **CON** pipeline y **CON** `init()`



(b) `CPI` **SIN** pipeline y **CON** `init()`



(c) `CPI` **CON** pipeline y **SIN** `init()`

(d) `CPI` **SIN** pipeline y **SIN** `init()`

Figura 2: Métricas obtenidas tras la ejecución del código

Los códigos en ensamblador obtenidos durante la depuración son los siguientes:

```
0x00000138: 13 01 01 fd        addi    sp,sp,-48
0x0000013c: 23 26 11 02        sw  ra,44(sp)
0x00000140: 23 24 81 02        sw  s0,40(sp)
0x00000144: 23 22 91 02        sw  s1,36(sp)
0x00000148: 23 20 21 03        sw  s2,32(sp)
0x0000014c: 23 2e 31 01        sw  s3,28(sp)
0x00000150: 23 2c 41 01        sw  s4,24(sp)
0x00000154: 23 2a 51 01        sw  s5,20(sp)
0x00000158: 23 28 61 01        sw  s6,16(sp)
0x0000015c: 23 26 71 01        sw  s7,12(sp)
0x00000160: ef 00 40 49        jal ra,0x5f4 <uartInit>
0x00000164: 13 05 10 00        li  a0,1
0x00000168: ef 00 10 05        jal ra,0x9b8 <pspEnableAllPerformanceMonitor>
0x0000016c: 93 05 10 00        li  a1,1
0x00000170: 13 05 80 00        li  a0,8
0x00000174: ef 00 10 05        jal ra,0x9c4 <pspPerformanceCounterSet>
0x00000178: 93 05 40 00        li  a1,4
0x0000017c: 13 05 00 01        li  a0,16
0x00000180: ef 00 50 04        jal ra,0x9c4 <pspPerformanceCounterSet>
0x00000184: 93 05 80 01        li  a1,24
0x00000188: 13 05 00 02        li  a0,32
0x0000018c: ef 00 90 03        jal ra,0x9c4 <pspPerformanceCounterSet>
0x00000190: 93 05 90 01        li  a1,25
0x00000194: 13 05 00 04        li  a0,64
0x00000198: ef 00 d0 02        jal ra,0x9c4 <pspPerformanceCounterSet>
0x0000019c: 13 05 80 00        li  a0,8
0x000001a0: ef 00 10 07        jal ra,0xa10 <pspPerformanceCounterGet>
```

(a) `CPI` **CON** pipeline y **CON** `init()`

```
0x00000138: 13 01 01 fd        addi    sp,sp,-48
0x0000013c: 23 26 11 02        sw  ra,44(sp)
0x00000140: 23 24 81 02        sw  s0,40(sp)
0x00000144: 23 22 91 02        sw  s1,36(sp)
0x00000148: 23 20 21 03        sw  s2,32(sp)
0x0000014c: 23 2e 31 01        sw  s3,28(sp)
0x00000150: 23 2c 41 01        sw  s4,24(sp)
0x00000154: 23 2a 51 01        sw  s5,20(sp)
0x00000158: 23 28 61 01        sw  s6,16(sp)
0x0000015c: 23 26 71 01        sw  s7,12(sp)
0x00000160: ef 00 c0 49        jal ra,0x5fc <uartInit>
0x00000164: 13 05 10 00        li  a0,1
0x00000168: ef 00 90 05        jal ra,0x9c0 <pspEnableAllPerformanceMonitor>
0x0000016c: 93 05 10 00        li  a1,1
0x00000170: 13 05 80 00        li  a0,8
0x00000174: ef 00 90 05        jal ra,0x9cc <pspPerformanceCounterSet>
0x00000178: 93 05 40 00        li  a1,4
0x0000017c: 13 05 00 01        li  a0,16
0x00000180: ef 00 d0 04        jal ra,0x9cc <pspPerformanceCounterSet>
0x00000184: 93 05 80 01        li  a1,24
0x00000188: 13 05 00 02        li  a0,32
0x0000018c: ef 00 10 04        jal ra,0x9cc <pspPerformanceCounterSet>
0x00000190: 93 05 90 01        li  a1,25
0x00000194: 13 05 00 04        li  a0,64
0x00000198: ef 00 50 03        jal ra,0x9cc <pspPerformanceCounterSet>
0x0000019c: 13 05 80 00        li  a0,8
0x000001a0: ef 00 90 07        jal ra,0xa18 <pspPerformanceCounterGet>
```

(b) `CPI` **SIN** pipeline y **CON** `init()`

```
0x000000d0: 13 01 01 fd        addi    sp,sp,-48
0x000000d4: 23 26 11 02        sw  ra,44(sp)
0x000000d8: 23 24 81 02        sw  s0,40(sp)
0x000000dc: 23 22 91 02        sw  s1,36(sp)
0x000000e0: 23 20 21 03        sw  s2,32(sp)
0x000000e4: 23 2e 31 01        sw  s3,28(sp)
0x000000e8: 23 2c 41 01        sw  s4,24(sp)
0x000000ec: 23 2a 51 01        sw  s5,20(sp)
0x000000f0: 23 28 61 01        sw  s6,16(sp)
0x000000f4: 23 26 71 01        sw  s7,12(sp)
0x000000f8: ef 00 00 49        jal ra,0x588 <uartInit>
0x000000fc: 13 05 10 00        li  a0,1
0x00000100: ef 00 d0 04        jal ra,0x94c <pspEnableAllPerformanceMonitor>
0x00000104: 93 05 10 00        li  a1,1
0x00000108: 13 05 80 00        li  a0,8
0x0000010c: ef 00 d0 04        jal ra,0x958 <pspPerformanceCounterSet>
0x00000110: 93 05 40 00        li  a1,4
0x00000114: 13 05 00 01        li  a0,16
0x00000118: ef 00 10 04        jal ra,0x958 <pspPerformanceCounterSet>
0x0000011c: 93 05 80 01        li  a1,24
0x00000120: 13 05 00 02        li  a0,32
0x00000124: ef 00 50 03        jal ra,0x958 <pspPerformanceCounterSet>
0x00000128: 93 05 90 01        li  a1,25
0x0000012c: 13 05 00 04        li  a0,64
0x00000130: ef 00 90 02        jal ra,0x958 <pspPerformanceCounterSet>
0x00000134: 13 05 80 00        li  a0,8
0x00000138: ef 00 d0 06        jal ra,0x9a4 <pspPerformanceCounterGet>
```

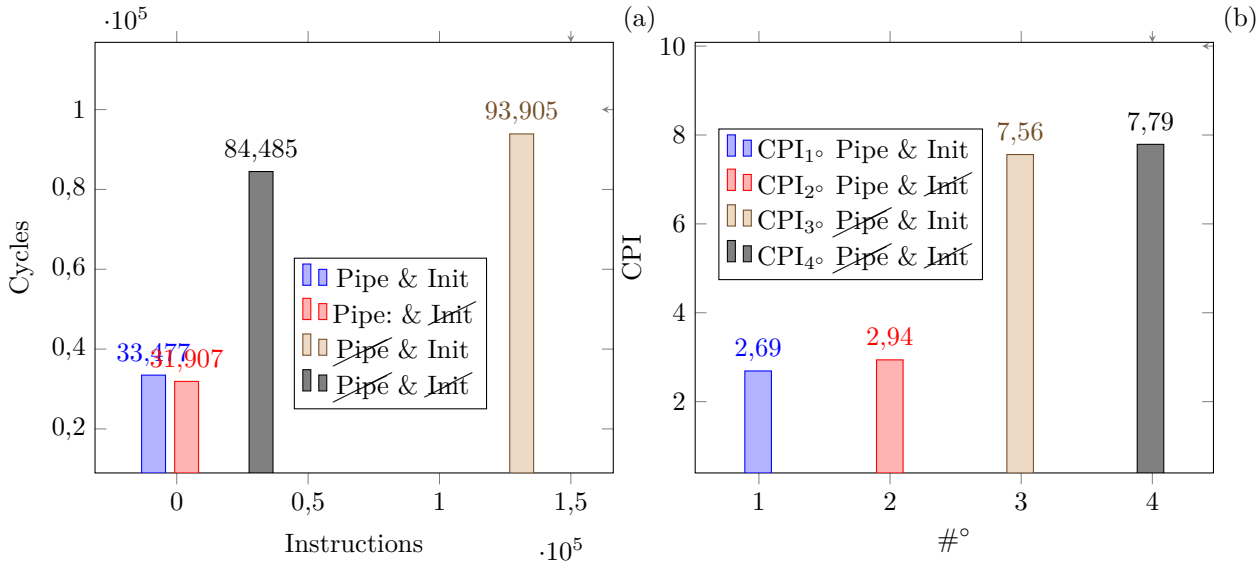
(c) `CPI` **CON** pipeline y **SIN** `init()`

```
0x000000d0: 13 01 01 fd        addi    sp,sp,-48
0x000000d4: 23 26 11 02        sw  ra,44(sp)
0x000000d8: 23 24 81 02        sw  s0,40(sp)
0x000000dc: 23 22 91 02        sw  s1,36(sp)
0x000000e0: 23 20 21 03        sw  s2,32(sp)
0x000000e4: 23 2e 31 01        sw  s3,28(sp)
0x000000e8: 23 2c 41 01        sw  s4,24(sp)
0x000000ec: 23 2a 51 01        sw  s5,20(sp)
0x000000f0: 23 28 61 01        sw  s6,16(sp)
0x000000f4: 23 26 71 01        sw  s7,12(sp)
0x000000f8: ef 00 80 49        jal ra,0x590 <uartInit>
0x000000fc: 13 05 10 00        li  a0,1
0x00000100: ef 00 50 05        jal ra,0x954 <pspEnableAllPerformanceMonitor>
0x00000104: 93 05 10 00        li  a1,1
0x00000108: 13 05 80 00        li  a0,8
0x0000010c: ef 00 50 05        jal ra,0x960 <pspPerformanceCounterSet>
0x00000110: 93 05 40 00        li  a1,4
0x00000114: 13 05 00 01        li  a0,16
0x00000118: ef 00 90 04        jal ra,0x960 <pspPerformanceCounterSet>
0x0000011c: 93 05 80 01        li  a1,24
0x00000120: 13 05 00 02        li  a0,32
0x00000124: ef 00 d0 03        jal ra,0x960 <pspPerformanceCounterSet>
0x00000128: 93 05 90 01        li  a1,25
0x0000012c: 13 05 00 04        li  a0,64
0x00000130: ef 00 10 03        jal ra,0x960 <pspPerformanceCounterSet>
0x00000134: 13 05 80 00        li  a0,8
0x00000138: ef 00 50 07        jal ra,0x9ac <pspPerformanceCounterGet>
```

(d) `CPI` **SIN** pipeline y **SIN** `init()`

Figura 3: Código en ensamblador obtenido tras la ejecución del código

Una vez tenemos las métricas y el desensamblado del código hemos construido las siguientes gráficas que nos permiten entender mejor que está pasando en cada una de las 4 ejecuciones mostradas antes;



Como las instrucciones no están entrelazadas entre sí, es prácticamente indiferente si tenemos pipe o no. En el bucle del entregable anterior, las instrucciones estaban entrelazadas, por ello aumentaba tanto el CPI, ya que existía una dependencia de registros.

Analizando los resultados que se muestran en los gráficos de arriba, vemos como en el gráfico de la derecha el **CPI** aumenta considerablemente para los dos casos en los que no está implementado el cortocircuito ($\mathbf{CPI_{3°}} \approx 2{,}8 \cdot \mathbf{CPI_{1°}}$ y $\mathbf{CPI_{4°}} \approx 2{,}8 \cdot \mathbf{CPI_{2°}}$), además centrando la atención en esos dos casos, vemos como la diferencia en el **CPI** es de **0.23**. Haciendo lo mismo para el $\mathbf{CPI_{1°}}$ y el $\mathbf{CPI_{2°}}$, la diferencia es de **0,25**. Teniendo en cuenta la similitud entre estas dos diferencias obtenidas, podemos concluir que inicializar o NO

las variables del filtro **FIR** resulta casi indiferente, debido al impacto ínfimo que ello conlleva. Es importante destacar que esto no aplica a todos los casos y ni siquiera aplicaría de manera consistente a este caso, ya que hay más aspectos que se deben analizar. Para determinar su efectividad deberíamos saber el coste de implementar dicha inicialización y el *tradeoff* entre el tiempo empleado en ello y los resultados. Si se realiza un proyecto en el que esa diferencia puede despreciarse, debido al uso de grandes parámetros entonces no es práctico inicializar el filtro.

Una vez analizado el impacto del uso del cortocircuito y la inicialización del filtro, hemos estimado el **CPI** que obtendríamos en una arquitectura **RISC V** y en una arquitectura **MIPS** [2].

En un procesador **MIPS** multiciclo, i.e:

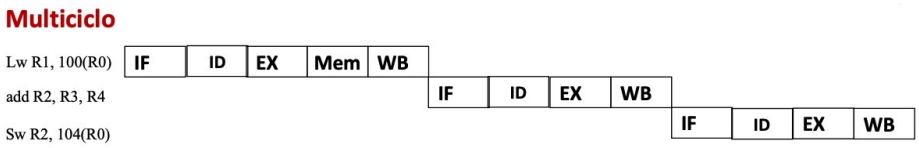


Figura 4: **MIPS** multiciclo

Sabiendo que no todas las instrucciones pasan por la etapa de memoria, analizamos el código ensamblador para la ejecución del filtro **FIR** con inicialización. Para saber cual es el código ensamblador correcto, hemos puesto dos *breakpoints*, el primero lo hemos puesto después de la inicialización, y el segundo lo hemos puesto debajo del bucle y fuera del mismo. Ejecutando el código, vemos que las lineas de ensamblador que nos interesan para este caos son aquellas comprendidas entre la linea 38 y 66 aproximadamente, sabiendo esto hemos analizado el código y para un **MIPS** multiciclo , se obtiene el siguiente **CPI**:

- 29 instrucciones en total.
  - 3 instrucciones de las 29 totales son de tipo `lw`, por lo que pasan por la etapa de memoria, y tendrán una duración de 5 ciclos.
  - Las 26 instrucciones restantes, no pasan por memoria, por lo que tardan solo 4 ciclos.

$$\textbf{CPI} = \frac{3 \cdot 5 + 26 \cdot 4}{29} = 4{,}10$$

Como podemos comprobar el CPI es enorme, esto es lógico porque es un procesador multiciclo.

Ahora hacemos lo mismo para la arquitectura del **RISC V**, el procedimiento para el cáclulo del **CPI** es el siguiente:

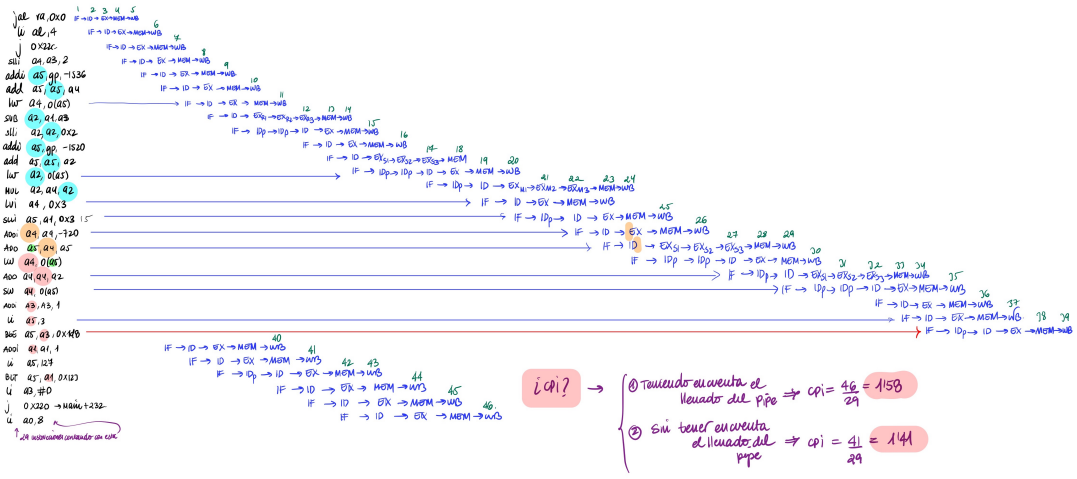


Figura 5: **RISC V** procesador segmentado

[2]Por teoría, sabemos que una arquitectura MIPS esta caracterizada por realizar operaciones entre registros, solo dos operaciones interactúan con memoria (`lw` y `str`), cuenta con un banco de 32 registros de 32 bytes cada uno y el tamaño de cada palabra es de 4 bytes

Vemos que el **CPI** obtenido sin tener en cuenta el llenado del pipe es de 1.41, por lo que no está muy alejado del **CPI** ideal estipulado para un procesador segmentado, ya que el ideal es de 1 ciclo por instrucción.

Estas dos últimas estimaciones, son aplicables para el caso en el que no se implementa la función de inicialización del filtro (i.e : `init()`), ya que a la hora de depurar el código ensamblador nos hemos dado cuenta de que este era idéntico al caso en el que se implementaba `init()`) a excepción de una linea de más en el caso de este último. Esto podemos verlo en los gráficos de abajo, donde comprobamos que se obtiene el mismo **CPI** *aproximadamente*. Para evitar cualquier duda, al final de este informe se han adjuntado los códigos ensamblador producidos.

Para estimar el **CPI** teórico del código implementado en el sistema embebido del laboratorio, hemos tenido en cuenta su arquitectura [3] (véase la siguiente figura)

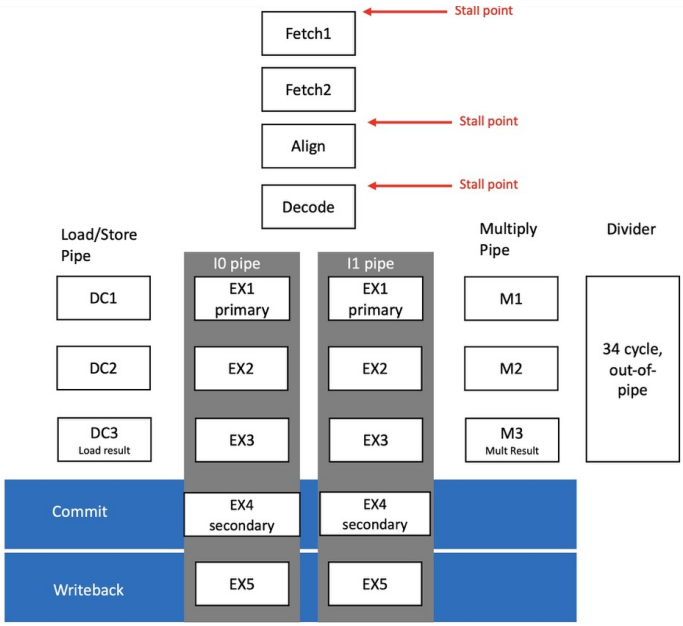


Figura 6: **SWERV** procesador del lab

A continuación se muestra el proceso del cálculo del CPI para dicha arquitectura:

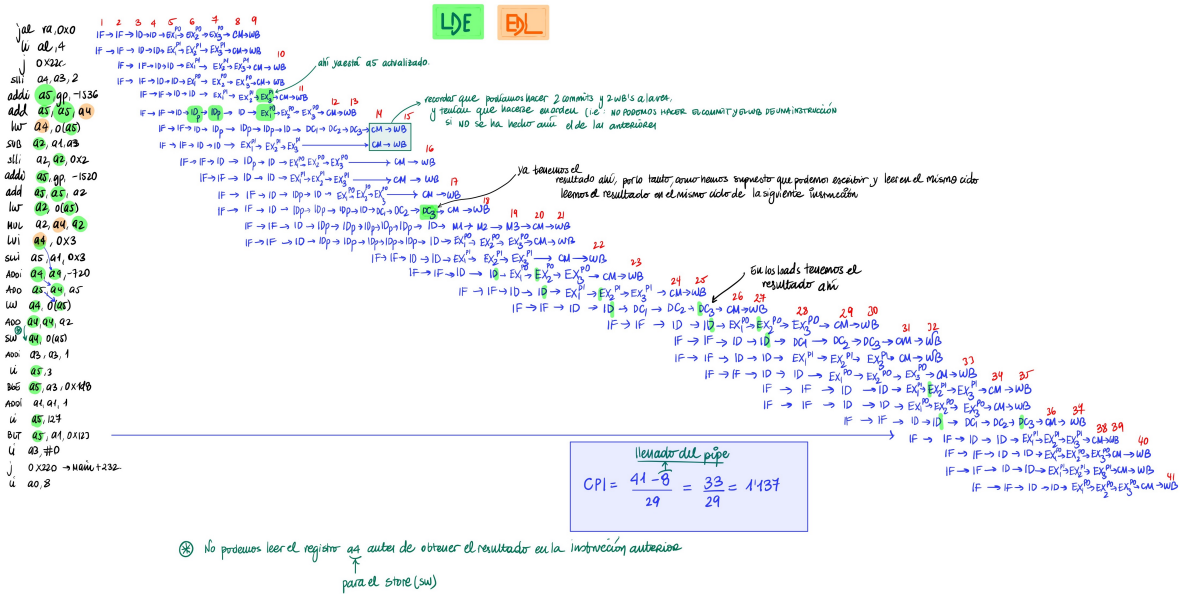
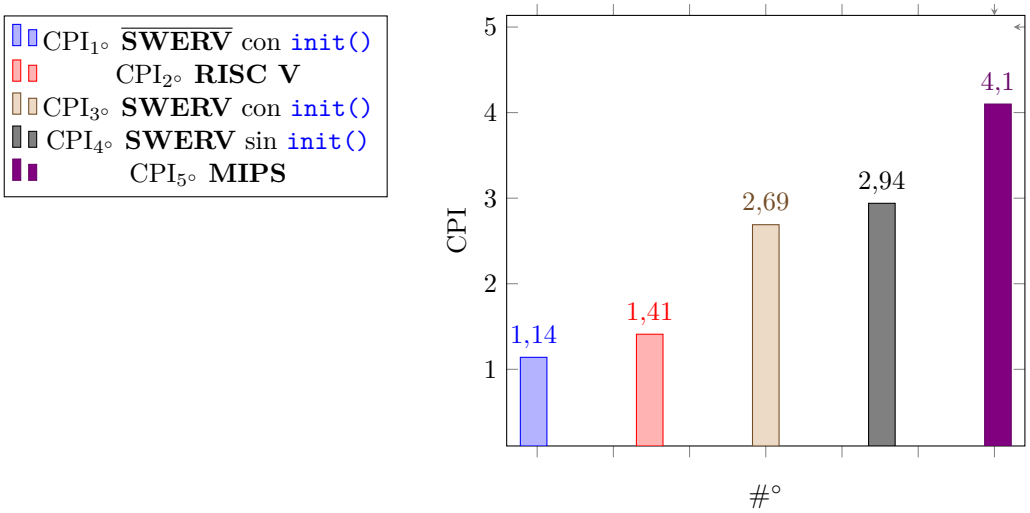


Figura 7: **SWERV** Cálculo del **CPI**

[3] Como no podía ser de otra manera

Vemos una mejora respecto al **CPI** del caso anterior, ya que sin tener en cuenta el llenado del pipe, el **CPI** que se obtenía era de: 1.41. Las dos estimaciones realizadas del **CPI** para una arquitectura segmentada del **RISC V** (i.e: **CPI** = 1.41) y esta última (i.e: **CPI** = 1.14) han sido revisadas numerosas veces para estar seguros del resultado.

A continuación el gráfico que recoge los **CPIs** obtenidos:



Vemos como la estimación del **CPI** para la arquitectura **SWERV** mejora respecto a la estimada para una arquitectura **MIPS** segmentada, de hecho, el **CPI** estimado es menos de la mitad de aquel obtenido en el laboratorio (para el caso en el que inicializamos el filtro y en el contrario). Esto último se debe al resto de lineas de código ensamblador que no hemos tenido en cuenta en la estimación, ya que no formaban parte del filtro FIR. Si revisamos el código del final del informe, vemos como esta es la causa del **CPI** obtenido, ya que nosotros hemos estimado el **CPI** contemplando las instrucciones comprendidas entre las lineas 39-66. Vemos como después de la linea 66 hasta la 94, hay más instrucciones de carga, suma y resta, algunas de ellas incluso con dependencias, por lo que si hubiésemos tenido en cuenta esas instrucciones, el **CPI** sería más alto respecto al que hemos estimado, no por que hubiese mas instrucciones, sino porque hay muchas más dependencias, y eso habría aumentado el número de ciclos.

Vemos como el **CPI** más alto es el del procesador multiciclo. Respecto a si hay o no desviación dentro del conjunto de las estimaciones realizadas para cada arquitectura, la respuesta es si. Es importante comentar que ha sido poco acertado realizar una estimación sin tener en cuenta el mismo numero de instrucciones que aquellas contempladas por el procesador, pero dentro del marco teórico, pueden explicarse las desviaciones en el **CPI**, ya que son arquitecturas diferentes, pero la diferencia obtenida, es mucho menor a aquella esperada. Esta desviación está estrechamente relacionada con el código implementado, ya que el tipo de código implementado no está pensado para ser ejecutado en una arquitectura como la del laboratorio, por ello tampoco vemos una diferencia tan abismal entre una arquitectura segmentada y la del laboratorio que se supone ser mejor (en términos muy generales). Esto último también tiene que ver con el alto **CPI** obtenido para la arquitectura **SWERV**, ya que el **CPI** ideal en este tipo de arquitecturas (superescalares) es de 0.5, lejos del 1.14 estimado y aún más de aquellos obtenidos en la práctica.

3. Segunda sesión

3.1. Entregable 1

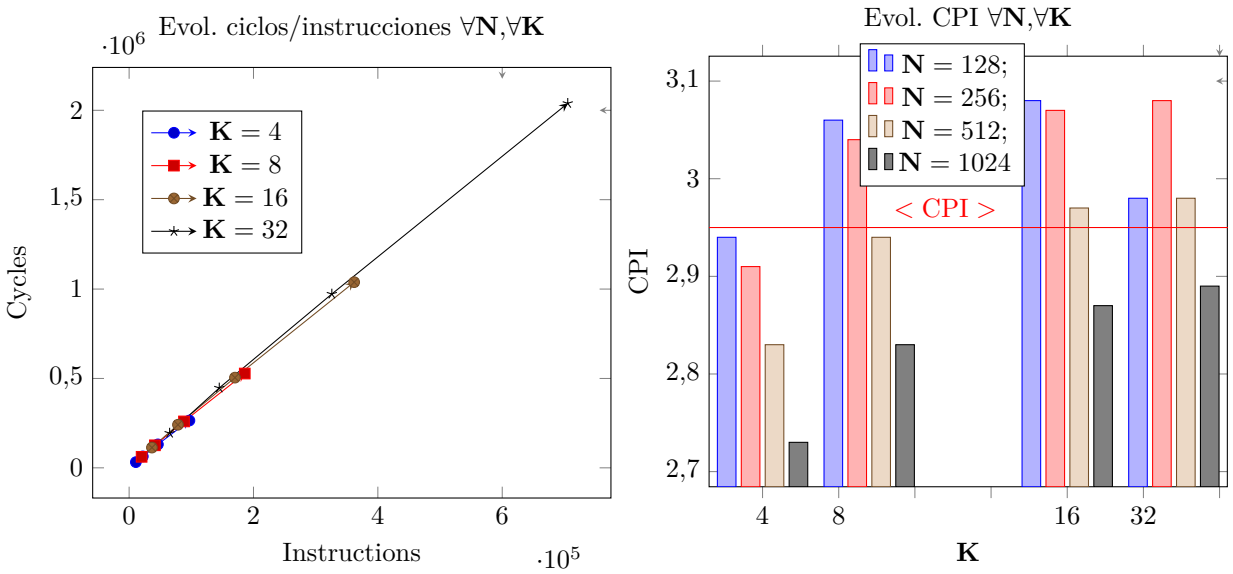
Elabore gráficas en las que muestre la evolución del número de instrucciones y del número de ciclos con las diferentes combinaciones de parámetros. De igual forma muestre la evolución de CPI. Analice los resultados obtenidos.

En este apartado hemos tenido que realizar 64 mediciones en total, ya que se ha medido el **CPI** para todas las combinaciones de **K** y **N**, sabiendo que: $\mathbf{K} \in \{4, 8, 16, 32\}$ y $\mathbf{N} \in \{128, 256, 512, 1024\}$. A continuación se adjuntan las tablas que recogen los resultados fruto de todas las combinaciones posibles:

Cuadro 1: Resultados obtenidos con el código base.

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI
4	31885	10844	2,94	63957	21980	2,91	131155	46284	2,83	264694	96956	2,73
8	61686	20096	3,06	126088	41472	3,04	259594	88256	2,94	527477	185984	2,83
16	113294	36680	3,08	240975	78536	3,07	504562	170184	2,97	1038536	361928	2,87
32	194236	65144	2,98	446873	144984	3,08	972349	325976	2,98	2039033	705368	2,89

A continuación se adjunta un gráfico que muestra la tendencia del **SoC** en lo que a instrucciones se refiere según varían los parámetros **K** y **N**.



En la figura de arriba vemos una tendencia claramente lineal para todos los casos. El CPI medio (i.e: < CPI >) es de 2,95 esto puede verse en la recta de la figura de arriba, a la derecha en rojo[4] Cabe destacar la cercanía que hay entre los resultados obtenidos para $\mathbf{K} = 8$ y $\mathbf{K} = 16$, aunque vemos como se repite la tendencia observada en resultados anteriores (i.e: $\uparrow$ **N** $\rightarrow$ **CPI** $\downarrow$). Para $\mathbf{K} = 32$ no vemos un aumento del **CPI**, al contrario, vemos como el valor del **CPI** para $\mathbf{N} = 128$ disminuye respecto al **CPI** obtenido para ese mismo valor de **N** pero en $\mathbf{K} = 16$, esto es algo un tanto sorprendente, ya que el valor de **K** ha aumentado el doble, por lo que esperábamos que este cambio también se viese reflejado en el **CPI**.

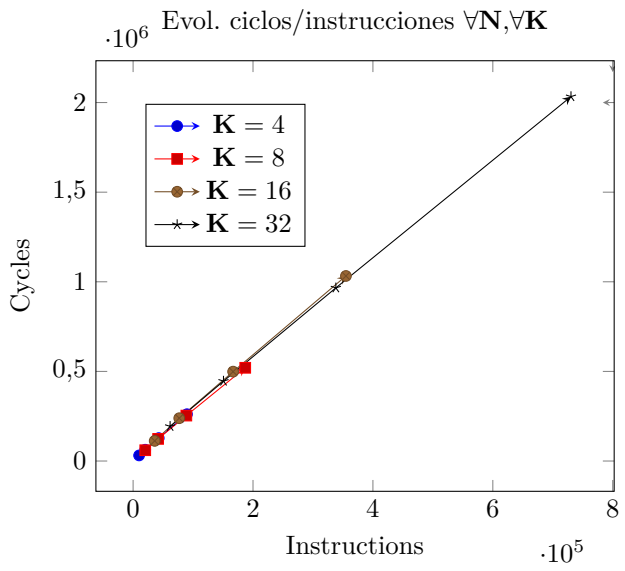
3.2. Entregable 2

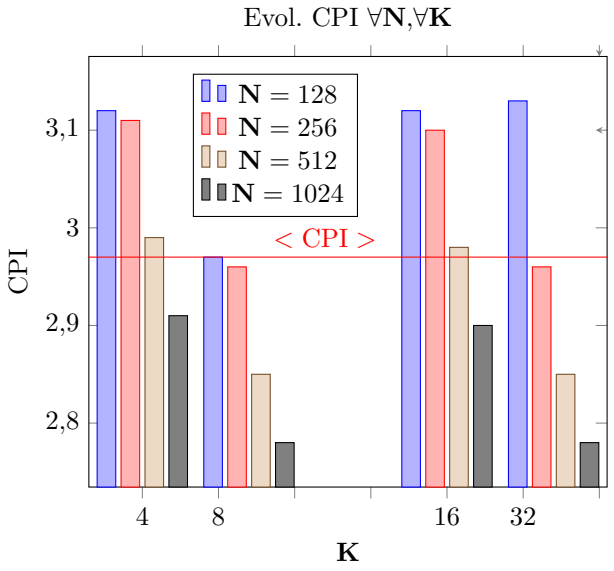
Elabore gráficas en las que muestre el impacto en el número de instrucciones y en el número de ciclos con las diferentes optimizaciones. Analice los resultados obtenidos.

[4]En esta gráfica y todas las que la suceden, en el eje **X** estamos representando los valores de **K**, por lo que la separación que hay entre los valores en **X** son indiferentes, simplemente sirve para indicar a que valor de **K** pertenecen cada uno de los 4 posibles valores de **N**.

Cuadro 2: Resultados obtenidos tras aplicar el *loop exchange*.

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI
4	31219	10004	3,12	62993	20244	3,11	127972	42756	2,99	261207	89844	2,91
8	60220	20264	2,97	123534	41768	2,96	253478	88808	2,85	520198	187048	2,78
16	112317	36008	3,12	238855	76968	3,10	498734	166824	2,98	1032465	354984	2,90
32	193270	61720	3,13	445678	150776	2,96	965931	338168	2,85	2033156	730360	2,78

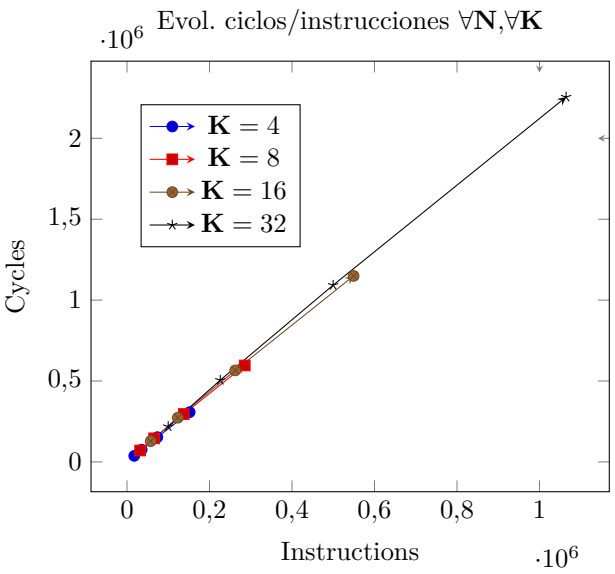


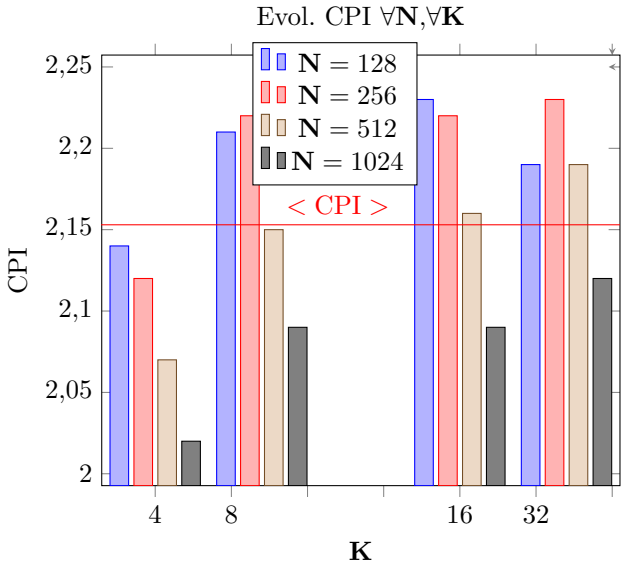


En la figura de arriba vemos una tendencia claramente lineal para todos los casos, aunque en **K** = 32 vemos una ligera inclinación en la tendencia. El CPI medio (i.e: < CPI >) es de 2,97, esto puede verse en la recta de la figura de arriba a la derecha. Además, en la gráfica del CPI a la derecha, vemos una cambio en la tendencia en **K**=8 ∀**N** ya que los valores de CPI obtenidos ∀**N** ∈ (**K** = 8) son menores que ∀**N** ∈ (**K** = 16) y que ∀**N** ∈ (**K** = 32). Comparando los resultados obtenidos con aquellos del caso anterior, vemos como el CPI empeora ascendiendo 0.02 puntos respecto al caso anterior. Algo negativo, pues se supone que el *loop exchange* es una técnica de optimización, y en nuestro caso empeora los resultados frente al código base. Debido al tipo del código (filtro FIR) concluimos en que no es una técnica apta para esta tarea.

Cuadro 3: Resultados obtenidos tras aplicar el *loop unrolling*.

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI
4	37447	17540	2,14	75563	35588	2,12	152887	73716	2,07	307522	152036	2,02
8	70477	31856	2,21	145706	65776	2,22	295933	137648	2,15	596176	285552	2,09
16	128006	57512	2,23	273308	123176	2,22	565603	262440	2,16	1149554	549416	2,09
32	218733	99896	2,19	505226	226072	2,23	1092437	499736	2,19	2255253	1064472	2,12

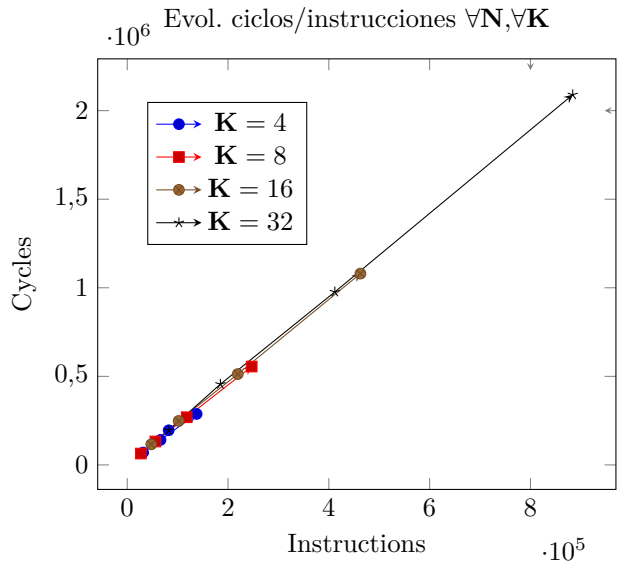


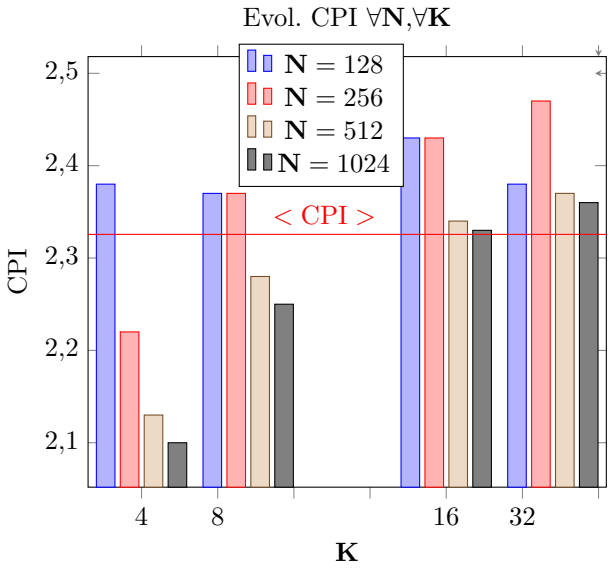


En este caso, el < CPI >= 2,15 vemos una clara disminución del mismo con respecto de los otros, pero ¿a costa de que? a costa de un mayor número de instrucciones. Respecto al CPI medio, vemos como este mejora considerablemente respecto a los anteriores. De hecho esta es la mejor técnica para este código, ya que a medida que aumentan las **Ks** y las **Ns** el efecto de esta técnica aumenta, debido a la estructura que toma el bucle al desenrollarlo.

Cuadro 4: Resultados obtenidos tras aplicar el *software pipeline*.

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI
4	195505	82232	2,38	69888	31556	2,22	141075	66096	2,13	287851	137756	2,10
8	64181	27056	2,37	132372	55856	2,37	269077	117992	2,28	555359	246944	2,25
16	116052	47656	2,43	248167	102056	2,43	512892	219288	2,34	1080106	462728	2,33
32	195498	82232	2,38	455760	184856	2,47	976477	411896	2,37	2089777	883928	2,36





En este caso, el < CPI >= 2,326, vemos una clara disminución del mismo con respecto de los otros, pero ¿a costa de que? a costa de un mayor número de instrucciones. Observamos un comportamiento similar al caso anterior, ya que en ambos casos, el CPI mejora según aumentan los valores de las **Ks** y las **Ns**. Estudiando el gráfico de la izquierda observamos una tendencia lineal y semejante al de las anteriores gráficas. Es importante destacar, el desvío que se observa en el CPI para N = 256 en K = 32, ya que el

CPI es mayor que aquel obtenido para la misma K pero para N = 128, esto no es nuevo ya que lo hemos observado en los resultados de las técnicas anteriores, no sabemos porque se observa este desvío en el CPI obtenido, pero nos ha resultado curioso.

3.3. Entregable 3

Investigue cómo se realizan las operaciones en tiempo flotante, utilizando la depuración y la conversión a ensamblador. Puede recurrir también a la documentación de los documentos Lab 11.pdf y Lab 12.pdf para saber el hardware disponible en el SoC para las operaciones aritméticas.

Para este apartado hemos tomado los siguientes valores para el código base pero con instrucciones en punto flotante.

Cuadro 5: Resultados obtenidos con instrucciones de tipo `float` en el código base.

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI	Cycles	Instr	CPI
4	150961	70364	2,14	300964	142940	2,10	629891	290124	2,17	1290056	586556	2,20
8	288986	135296	2,14	589398	279552	2,11	1240071	572096	2,17	2561807	1161344	a2,2
16	537335	251720	2,14	1139190	539336	2,11	2433635	1122504	2,17	5062537	2297288	2,20
32	939296	433784	2,17	2113610	1005144	2,10	4694417	2169176	2,16	9956161	4514648	2,20

Comparando los resultados obtenidos en cada columna **N** para los distintos valores de **K**, obtenemos un CPI similar en la mayoría de los casos.

Es importante tener en cuenta, que el número de ciclos obtenidos para este caso es aproximadamente 5 veces más respecto a los casos anteriores, pero el **CPI** ha bajado, por lo que la única respuesta a esto, es que el número de instrucciones también se ha visto incrementado. Por el mero hecho de haber obtenido un **CPI** menor en este último caso, no significa que sea mejor realizar las operaciones en punto flotante, ya que el número de instrucciones ha aumentado considerablemente, lo que se traduce en una mayor latencia, término muy famoso dentro del mundo de los *benchmarks*.

Las operaciones en tiempo flotante se realizan utilizando la aritmética de punto flotante, que es una forma de representar números reales en una computadora. La mayoría de las computadoras modernas utilizan el estándar IEEE 754 para este tipo de aritmética.

Para convertir un programa en lenguaje de alto nivel que involucra operaciones en punto flotante a ensamblador, se utiliza un compilador, el cual traduce las instrucciones en código de máquina. El código de máquina generado se puede examinar para entender cómo se realizan las operaciones aritméticas en la CPU. También se pueden utilizar herramientas de depuración como GDB o Visual Studio para examinar el estado de los registros y la memoria durante la ejecución del programa.

Es importante recordar que la aritmética de punto flotante puede ser propensa a errores debido a la representación limitada de ciertos números reales en la computadora. Por lo tanto, es importante tener en cuenta los errores de precisión y redondeo al trabajar estos números.

En RISC-V, las operaciones en tiempo flotante se realizan utilizando extensiones específicas para la aritmética de punto flotante. El estándar RISC-V incluye extensiones opcionales para la aritmética de punto flotante que se pueden agregar a la arquitectura base.

Las extensiones de aritmética de punto flotante de RISC-V incluyen la Extensión de Aritmética de Punto Flotante de Precisión Simple (F) y la Extensión de Aritmética de Punto Flotante de Doble Precisión (D), que permiten realizar operaciones en punto flotante con números de precisión simple y doble, respectivamente.

RISC-V ha desarrollado una extensión de instrucciones comprimidas (RVC) que reduce el tamaño de las instrucciones de enteros y punto flotante comunes a sólo 16 bits. Esto se logra reduciendo los campos de control, inmediato y registro, y haciendo uso de los registros redundantes o implícitos. Como resultado, se disminuyen los costos, se reduce el consumo de energía y se necesita menos memoria. La etapa de alineación

(Align) se encarga de descomprimir las instrucciones de 16 bits cuando es necesario, antes de pasarlas a la etapa de decodificación (Decode), la cual sólo es capaz de decodificar instrucciones de 32 bits. Para llevar a cabo esta tarea, se utiliza el módulo `if_compress_ctl`, el cual se encuentra incorporado dentro del alineador (módulo `ifu_aln_ctl`).

4. Códigos implementados

- Loop exchange:

```
#ifdef FIR
//__asm("li t2, 0x001"); // Disable Pipelined Execution
//__asm("csrrs t1, 0x7F9, t2");
int t,i,sum0,sum1;
//init();

//Aqui declaramos las variables locales que hemos creado para reducir el numero de accesos a memoria
short x0,h0,x1,h1;

for (t=K;t<N;t++){
    sum0 = 0;
    sum1 = 0;
    x0 = X[t];
    h0 = h[i];

    for (i=0;i<K;i++){

            x1 = X[t+i+1];
            sum0 += x0 * h0;
            sum1 += x1 * h0;
            x0 = X[t+i+2];
            h1 = h[i+1];
            sum0 += x1 * h1;
            sum1 += x0 * h1;
    }
     Y[t] = sum0; // tenemos que cambiar la magnitud del desplazamiento
     Y[t+1] = sum1; // tenemos que cambiar le 15 por el numero del desplazamiento que queramos efectuar

```

- Loop unrolling:

```
#ifdef FIR
//__asm("li t2, 0x001"); // Disable Pipelined Execution
//__asm("csrrs t1, 0x7F9, t2");
int t,i,sum0,sum1;
//init();

//Aqui declaramos las variables locales que hemos creado para reducir el numero de accesos a memoria
short x0,h0,x1,h1;

for (t=K;t<N;t++){
    sum0 = 0;
    sum1 = 0;

    for (i=0;i<K;i++){
        x0 = X[t];
        h0 = h[i];
        x1 = X[t+i+1];
        sum0 += x0 * h0;
        sum1 += x1 * h0;
        x0 = X[t+i+2];
        h1 = h[i+1];
        sum0 += x1 * h1;
        sum1 += x0 * h1;
        }
     Y[t] = sum0; // tenemos que cambiar la magnitud del desplazamiento
     Y[t+1] = sum1; // tenemos que cambiar le 15 por el numero del desplazamiento que queramos efectuar

```

- Software pipeline:

```
#ifdef FIR
//__asm("li t2, 0x001"); // Disable Pipelined Execution
//__asm("csrrs t1, 0x7F9, t2");
int t,i,sum0,sum1;
//init();

//Aqui declaramos las variables locales que hemos creado para reducir el numero de accesos a memoria
short x0,h0,x1,h1;

for (t=K;t<N;t++){
    sum0 = 0;
    sum1 = 0;
    x0 = X[t];
    h0 = h[i];
    for (i=0;i<K;i++){
        x1 = X[t+i+1];
        sum0 += x0 * h0;
        sum1 += x1 * h0;
        x0 = X[t+i+2];
        h1 = h[i+1];
        sum0 += x1 * h1;
        sum1 += x0 * h1;
        }
     Y[t] = sum0; // tenemos que cambiar la magnitud del desplazamiento
     Y[t+1] = sum1; // tenemos que cambiar le 15 por el numero del desplazamiento que queramos efectuar

```

5. Códigos ensamblador (primera sesión)

Con `init()`

```
0x00000138: 13 01 01 fd        addi sp,sp,-48
0x0000013c: 23 26 11 02        sw   ra,44(sp)
0x00000140: 23 24 81 02        sw   s0,40(sp)
0x00000144: 23 22 91 02        sw   s1,36(sp)
0x00000148: 23 20 21 03        sw   s2,32(sp)
0x0000014c: 23 2e 31 01        sw   s3,28(sp)
0x00000150: 23 2c 41 01        sw   s4,24(sp)
0x00000154: 23 2a 51 01        sw   s5,20(sp)
0x00000158: 23 28 61 01        sw   s6,16(sp)
0x0000015c: 23 26 71 01        sw   s7,12(sp)
0x00000160: ef 00 40 49        jal ra,0x5f4 <uartInit>
0x00000164: 13 05 10 00        li   a0,1
0x00000168: ef 00 10 05        jal ra,0x9b8 <pspEnableAllPerformanceMonitor>
0x0000016c: 93 05 10 00        li   a1,1
0x00000170: 13 05 80 00        li   a0,8
0x00000174: ef 00 10 05        jal ra,0x9c4 <pspPerformanceCounterSet>
0x00000178: 93 05 40 00        li   a1,4
0x0000017c: 13 05 00 01        li   a0,16
0x00000180: ef 00 50 04        jal ra,0x9c4 <pspPerformanceCounterSet>
0x00000184: 93 05 80 01        li   a1,24
0x00000188: 13 05 00 02        li   a0,32
0x0000018c: ef 00 90 03        jal ra,0x9c4 <pspPerformanceCounterSet>
0x00000190: 93 05 90 01        li   a1,25
0x00000194: 13 05 00 04        li   a0,64
0x00000198: ef 00 d0 02        jal ra,0x9c4 <pspPerformanceCounterSet>
0x0000019c: 13 05 80 00        li   a0,8
0x000001a0: ef 00 10 07        jal ra,0xa10 <pspPerformanceCounterGet>
0x000001a4: 13 0b 05 00        mv   s6,a0
0x000001a8: 13 05 00 01        li   a0,16
0x000001ac: ef 00 50 06        jal ra,0xa10 <pspPerformanceCounterGet>
0x000001b0: 13 0a 05 00        mv   s4,a0
0x000001b4: 13 05 00 02        li   a0,32
0x000001b8: ef 00 90 05        jal ra,0xa10 <pspPerformanceCounterGet>
0x000001bc: 13 09 05 00        mv   s2,a0
0x000001c0: 13 05 00 04        li   a0,64
0x000001c4: ef 00 d0 04        jal ra,0xa10 <pspPerformanceCounterGet>
0x000001c8: 13 04 05 00        mv   s0,a0
0x000001cc: ef f0 5f e3        jal ra,0x0 <init>
0x000001d0: 93 05 40 00        li   a1,4
0x000001d4: 6f 00 80 05        j 0x22c <main+244>
0x000001d8: 13 97 26 00        slli   a4,a3,0x2
0x000001dc: 93 87 01 a0        addi   a5,gp,-1536
0x000001e0: b3 87 e7 00        add a5,a5,a4
0x000001e4: 03 a7 07 00        lw   a4,0(a5)
0x000001e8: 33 86 d5 40        sub a2,a1,a3
0x000001ec: 13 16 26 00        slli   a2,a2,0x2
0x000001f0: 93 87 01 a1        addi   a5,gp,-1520
0x000001f4: b3 87 c7 00        add a5,a5,a2
0x000001f8: 03 a6 07 00        lw   a2,0(a5)
0x000001fc: 33 06 c7 02        mul a2,a4,a2
0x00000200: 37 37 00 00        lui a4,0x3
0x00000204: 93 97 25 00        slli   a5,a1,0x2
0x00000208: 13 07 07 d3        addi   a4,a4,-720 # 0x2d30 <Y>
0x0000020c: b3 07 f7 00        add a5,a4,a5
0x00000210: 03 a7 07 00        lw   a4,0(a5)
0x00000214: 33 07 c7 00        add a4,a4,a2
0x00000218: 23 a0 e7 00        sw   a4,0(a5)
0x0000021c: 93 86 16 00        addi   a3,a3,1
0x00000220: 93 07 30 00        li   a5,3
0x00000224: e3 da d7 fa        bge a5,a3,0x1d8 <main+160>
0x00000228: 93 85 15 00        addi   a1,a1,1
0x0000022c: 93 07 f0 07        li   a5,127
0x00000230: 63 c6 b7 00        blt a5,a1,0x23c <main+260>
0x00000234: 93 06 00 00        li   a3,0
0x00000238: 6f f0 9f fe        j 0x220 <main+232>
0x0000023c: 13 05 80 00        li   a0,8
0x00000240: ef 00 00 7d        jal ra,0xa10 <pspPerformanceCounterGet>
0x00000244: 93 0b 05 00        mv   s7,a0
0x00000248: 13 05 00 01        li   a0,16
0x0000024c: ef 00 40 7c        jal ra,0xa10 <pspPerformanceCounterGet>
0x00000250: 93 0a 05 00        mv   s5,a0
```

```
0x00000254: 13 05 00 02        li  a0,32
0x00000258: ef 00 80 7b        jal ra,0xa10 <pspPerformanceCounterGet>
0x0000025c: 93 09 05 00        mv  s3,a0
0x00000260: 13 05 00 04        li  a0,64
0x00000264: ef 00 c0 7a        jal ra,0xa10 <pspPerformanceCounterGet>
0x00000268: 93 04 05 00        mv  s1,a0
0x0000026c: b3 85 6b 41        sub a1,s7,s6
0x00000270: 37 15 00 00        lui a0,0x1
0x00000274: 13 05 05 a7        addi  a0,a0,-1424 # 0xa70
0x00000278: ef 00 c0 64        jal ra,0x8c4 <printfNexys>
0x0000027c: b3 85 4a 41        sub a1,s5,s4
0x00000280: 37 15 00 00        lui a0,0x1
0x00000284: 13 05 c5 a7        addi  a0,a0,-1412 # 0xa7c
0x00000288: ef 00 c0 63        jal ra,0x8c4 <printfNexys>
0x0000028c: b3 85 29 41        sub a1,s3,s2
0x00000290: 37 15 00 00        lui a0,0x1
0x00000294: 13 05 05 a9        addi  a0,a0,-1392 # 0xa90
0x00000298: ef 00 c0 62        jal ra,0x8c4 <printfNexys>
0x0000029c: b3 85 84 40        sub a1,s1,s0
0x000002a0: 37 15 00 00        lui a0,0x1
0x000002a4: 13 05 c5 a9        addi  a0,a0,-1380 # 0xa9c
0x000002a8: ef 00 c0 61        jal ra,0x8c4 <printfNexys>
0x000002ac: 6f 00 00 00        j 0x2ac <main+372>
```

Sin `init()`

```
0x000000d0: 13 01 01 fd        addi  sp,sp,-48
0x000000d4: 23 26 11 02        sw  ra,44(sp)
0x000000d8: 23 24 81 02        sw  s0,40(sp)
0x000000dc: 23 22 91 02        sw  s1,36(sp)
0x000000e0: 23 20 21 03        sw  s2,32(sp)
0x000000e4: 23 2e 31 01        sw  s3,28(sp)
0x000000e8: 23 2c 41 01        sw  s4,24(sp)
0x000000ec: 23 2a 51 01        sw  s5,20(sp)
0x000000f0: 23 28 61 01        sw  s6,16(sp)
0x000000f4: 23 26 71 01        sw  s7,12(sp)
0x000000f8: ef 00 00 49        jal ra,0x588 <uartInit>
0x000000fc: 13 05 10 00        li  a0,1
0x00000100: ef 00 d0 04        jal ra,0x94c <pspEnableAllPerformanceMonitor>
0x00000104: 93 05 10 00        li  a1,1
0x00000108: 13 05 80 00        li  a0,8
0x0000010c: ef 00 d0 04        jal ra,0x958 <pspPerformanceCounterSet>
0x00000110: 93 05 40 00        li  a1,4
0x00000114: 13 05 00 01        li  a0,16
0x00000118: ef 00 10 04        jal ra,0x958 <pspPerformanceCounterSet>
0x0000011c: 93 05 80 01        li  a1,24
0x00000120: 13 05 00 02        li  a0,32
0x00000124: ef 00 50 03        jal ra,0x958 <pspPerformanceCounterSet>
0x00000128: 93 05 90 01        li  a1,25
0x0000012c: 13 05 00 04        li  a0,64
0x00000130: ef 00 90 02        jal ra,0x958 <pspPerformanceCounterSet>
0x00000134: 13 05 80 00        li  a0,8
0x00000138: ef 00 d0 06        jal ra,0x9a4 <pspPerformanceCounterGet>
0x0000013c: 13 0b 05 00        mv  s6,a0
0x00000140: 13 05 00 01        li  a0,16
0x00000144: ef 00 10 06        jal ra,0x9a4 <pspPerformanceCounterGet>
0x00000148: 13 0a 05 00        mv  s4,a0
0x0000014c: 13 05 00 02        li  a0,32
0x00000150: ef 00 50 05        jal ra,0x9a4 <pspPerformanceCounterGet>
0x00000154: 13 09 05 00        mv  s2,a0
0x00000158: 13 05 00 04        li  a0,64
0x0000015c: ef 00 90 04        jal ra,0x9a4 <pspPerformanceCounterGet>
0x00000160: 13 04 05 00        mv  s0,a0
0x00000164: 93 05 40 00        li  a1,4
0x00000168: 6f 00 80 05        j 0x1c0 <main+240>
0x0000016c: 13 97 26 00        slli  a4,a3,0x2
0x00000170: 93 87 01 a0        addi  a5,gp,-1536
0x00000174: b3 87 e7 00        add a5,a5,a4
0x00000178: 03 a7 07 00        lw  a4,0(a5)
0x0000017c: 33 86 d5 40        sub a2,a1,a3
0x00000180: 13 16 26 00        slli  a2,a2,0x2
0x00000184: 93 87 01 a1        addi  a5,gp,-1520
0x00000188: b3 87 c7 00        add a5,a5,a2
0x0000018c: 03 a6 07 00        lw  a2,0(a5)
0x00000190: 33 06 c7 02        mul a2,a4,a2
0x00000194: 37 37 00 00        lui a4,0x3
```

```
0x00000198: 93 97 25 00        slli  a5,a1,0x2
0x0000019c: 13 07 87 cc        addi  a4,a4,-824 # 0x2cc8 <Y>
0x000001a0: b3 07 f7 00        add a5,a4,a5
0x000001a4: 03 a7 07 00        lw  a4,0(a5)
0x000001a8: 33 07 c7 00        add a4,a4,a2
0x000001ac: 23 a0 e7 00        sw  a4,0(a5)
0x000001b0: 93 86 16 00        addi  a3,a3,1
0x000001b4: 93 07 30 00        li  a5,3
0x000001b8: e3 da d7 fa        bge a5,a3,0x16c <main+156>
0x000001bc: 93 85 15 00        addi  a1,a1,1
0x000001c0: 93 07 f0 07        li  a5,127
0x000001c4: 63 c6 b7 00        blt a5,a1,0x1d0 <main+256>
0x000001c8: 93 06 00 00        li  a3,0
0x000001cc: 6f f0 9f fe        j 0x1b4 <main+228>
0x000001d0: 13 05 80 00        li  a0,8
0x000001d4: ef 00 00 7d        jal ra,0x9a4 <pspPerformanceCounterGet>
0x000001d8: 93 0b 05 00        mv  s7,a0
0x000001dc: 13 05 00 01        li  a0,16
0x000001e0: ef 00 40 7c        jal ra,0x9a4 <pspPerformanceCounterGet>
0x000001e4: 93 0a 05 00        mv  s5,a0
0x000001e8: 13 05 00 02        li  a0,32
0x000001ec: ef 00 80 7b        jal ra,0x9a4 <pspPerformanceCounterGet>
0x000001f0: 93 09 05 00        mv  s3,a0
0x000001f4: 13 05 00 04        li  a0,64
0x000001f8: ef 00 c0 7a        jal ra,0x9a4 <pspPerformanceCounterGet>
0x000001fc: 93 04 05 00        mv  s1,a0
0x00000200: b3 85 6b 41        sub a1,s7,s6
0x00000204: 37 15 00 00        lui a0,0x1
0x00000208: 13 05 45 a0        addi  a0,a0,-1532 # 0xa04
0x0000020c: ef 00 c0 64        jal ra,0x858 <printfNexys>
0x00000210: b3 85 4a 41        sub a1,s5,s4
0x00000214: 37 15 00 00        lui a0,0x1
0x00000218: 13 05 05 a1        addi  a0,a0,-1520 # 0xa10
0x0000021c: ef 00 c0 63        jal ra,0x858 <printfNexys>
0x00000220: b3 85 29 41        sub a1,s3,s2
0x00000224: 37 15 00 00        lui a0,0x1
0x00000228: 13 05 45 a2        addi  a0,a0,-1500 # 0xa24
0x0000022c: ef 00 c0 62        jal ra,0x858 <printfNexys>
0x00000230: b3 85 84 40        sub a1,s1,s0
0x00000234: 37 15 00 00        lui a0,0x1
0x00000238: 13 05 05 a3        addi  a0,a0,-1488 # 0xa30
0x0000023c: ef 00 c0 61        jal ra,0x858 <printfNexys>
0x00000240: 6f 00 00 00        j 0x240 <main+368>
```