Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Avances en el uso eficiente de sistemas multi-softcores en FPGAs

Tesis de Licenciatura en Ciencias de la Computación

Daniel Claverino

Director: David Alejandro González Marquez

Codirector: Esteban Mocskos

Buenos Aires, 2023

Resumen

Desde hace años, el avance tecnológico en la fabricación de circuitos integrados ha permitido implementar múltiples procesadores en una misma pastilla con una creciente cantidad de núcleos, memoria e, inclusive, contener unidades funcionales con distinto grado de especialización. Sin embargo, a la hora de resolver problemas específicos, los procesadores de propósito general pueden ser superados por **aceleradores**, que son sistemas especialmente diseñados. Los más comunes son los aceleradores de vídeo que permiten, por ejemplo, procesar imágenes o descomprimir un flujo (*stream*) de vídeo de forma muy eficiente.

Las **FPGAs** (*Field-Programmable Gate Array*) surgen como plataformas que permiten implementar soluciones de hardware programable, es decir, hardware que puede ser modificado o rediseñado por el usuario sin necesidad de reemplazar el circuito integrado. Consisten en una grilla de celdas de distinta especialización que pueden conectarse de diversas formas, resultando en una plataforma con alta flexibilidad.

Las FPGAs resultan un camino válido para el diseño, prototipado y construcción de hardware y, en particular, aceleradores. Siempre dentro de sus limitaciones de recursos, permiten implementar hardware y se puede lograr un alto grado de paralelismo.

Dado este nivel de flexibilidad, un componente que se implementa en FPGA recibe el nombre de **softcore**. Estos pueden ir desde una máquina de estados o un sumador, hasta procesadores multi-core con varios niveles de cache. El límite dependerá de la cantidad de celdas y las formas de conexión disponibles, que es fija para una FPGA dada.

El presente trabajo busca avanzar en técnicas que permitan el uso eficiente de múltiples *softcores* dentro de una FPGA, proponiendo un *framework* que permitirá estudiar distintos sistemas de procesamiento. Haremos especial uso del `MicroBlaze`, uno de los *softcores* más utilizados, desarrollado por Xilinx y creado específicamente para su uso en FPGAs.

Una de las principales limitaciones de los sistemas basados en *softcores* es el acceso a memoria. Partiendo de este punto y utilizando bloques RAM de FPGA (BRAM), se propone estudiar un conjunto de estrategias de uso de la memoria: usar el *stack* en BRAM, correr código del programa en BRAM, y correr código de funciones de sincronización en BRAM. Esto implica ejecutar programas cuyos accesos a memoria utilizan distintas memorias, cada una con características distintas (BRAM y Cache+DDR).

Respecto a las tres formas de uso de BRAM propuestas, notamos que mover y correr código de funciones de sincronización en BRAM no genera mejoras significativas salvo en casos donde hay muchos llamados o varios procesadores en espera. Dependiendo del algoritmo y su implementación, usar el stack en BRAM puede ir desde tener poco efecto a tener un impacto significativo en el desempeño, por encima de mover las funciones de sincronización. Finalmente, mover y correr código del algoritmo en BRAM genera el mayor impacto positivo en el rendimiento. Este impacto puede verse reducido si hay partes del código de uso frecuente que no se han movido a BRAM y continúan en memoria principal.

Tener control del hardware nos permite implementar soluciones que se ajusten al problema a resolver. Trabajar desde una capa de abstracción baja implica que los desarrolladores deben manejar detalles de muy bajo nivel, como los procesadores mismos, su interconexión, y la distribución de memoria utilizando distintos tipos de memoria.

Palabras claves: FPGA, SMP, BRAM, *softcore*, MicroBlaze, memoria, cache, *stack*, *spinlock*, performance.

Abstract

For years, technological advances in the manufacture of integrated circuits have allowed the implementation of multiple processors on the same chip with an increasing number of cores, memory, and even functional units with different degrees of specialization. However, when it comes to solving specific problems, general-purpose processors can be outperformed by accelerators, which are specially designed-systems.

The most common are video accelerators that allow, for example, to process images or decompress a video stream very efficiently.

FPGAs (*Field-Programmable Gate Array*) are platforms allowing programmable hardware implementation. The user can modify or redesign this hardware while using the same chip. FPGAs are composed of a variety of specialized cells distributed on a grid. These cells can be connected in different ways, resulting in a highly flexible platform.

FPGAs are a valid path for hardware design, prototyping, and implementation. It is possible to implement several hardware components within their fixed and limited resources. Their nature favors highly parallel components and efficient access to I/O external events. Accelerators are one such particular piece of hardware.

Given this flexibility, a component implemented on an FPGA is called **softcore**. These can go from a state machine or an adder to a multi-core processor with multiple cache levels, but some restrictions still apply. The number of cells and available connections are fixed on an FPGA limiting what can be implemented.

In this thesis, we aim to study efficient ways of running multiple softcores on an FPGA. We propose a framework that allows studying a wide range of processing systems, and we use it to implement one particular multi-core system. We use Xilinx's MicroBlaze, one of the most widely used softcore and specifically created to be deployed on FPGAs.

One limitation of softcore systems is memory access. Having this in mind, we use RAM Blocks (BRAM) to study different memory usage strategies: point the stack to BRAM, execute program code on BRAM, and execute synchronization code on BRAM. This implies running programs where memory accesses are done to different memory types, each with its characteristics (BRAM and Cache+DDR).

Regarding the three proposed BRAM uses, we found moving and executing synchronization code on BRAM to yield no significant improvements except in heavy synchronization cases or where processors were competing too strongly with memory access. Depending on the algorithm and its implementation, stack on BRAM results ranged from no impact to a significant performance improvement, even over the prior case. Lastly, moving and executing program code on BRAM showed the most positive impact on the performance of the three strategies. The latter's results were reduced when the program was not completely moved to BRAM and frequently executed code was still executed on other memory.

Controlling the hardware lets us implement a specifically designed solution to a problem. Working on a low-abstraction layer implies developers need to handle low-level details, such as processors, the interconnection between them, and memory distribution using different memory types.

Keywords: FPGA, SMP, BRAM, softcore, MicroBlaze, memory, cache, stack, spin-lock, performance.

Agradecimientos

A mi familia, que conozco desde chico y siempre estuvo presente.

A David, mi director de tesis, por la propuesta de investigación, el seguimiento y particularmente el trato humano.

A los profesores de la facultad que me acompañaron en este camino y me incentivaron a aprender más y más.

A los números primos, que no voy a enumerar porque seguramente me olvide alguno.

A mi perro, que no sabe leer pero sí sabe hacerme reír.

A mi mamá.

Índice general

1. Introducción **9**
- 1.1. Motivación . . . 12
- 1.2. Objetivos del trabajo . . . 13
- 1.3. Trabajo relacionado . . . 13
- 1.4. FPGA . . . 15
  - 1.4.1. Tipos de celdas . . . 15
  - 1.4.2. Interconectividad . . . 17
  - 1.4.3. PL y PS . . . 18
  - 1.4.4. Limitaciones . . . 19
  - 1.4.5. Tipos y usos . . . 19
  - 1.4.6. Diseño . . . 20
- 1.5. Plataforma . . . 23
  - 1.5.1. Hardware . . . 23
  - 1.5.2. Vivado . . . 23
  - 1.5.3. MicroBlaze . . . 24
  - 1.5.4. Otras herramientas . . . 26

2. Infraestructura desarrollada **28**
- 2.1. Arquitectura . . . 28
- 2.2. Arquitectura (Hardware) . . . 30
  - 2.2.1. Uso de Memoria . . . 31
  - 2.2.2. MicroBlaze . . . 32
  - 2.2.3. Cache L2 . . . 34
  - 2.2.4. Softcore AXI Controller . . . 35
  - 2.2.5. Softcore Controller . . . 38
  - 2.2.6. Consideraciones . . . 41
- 2.3. Arquitectura (Software) . . . 43
  - 2.3.1. Programas . . . 45
  - 2.3.2. Compilación . . . 48
  - 2.3.3. Generación de Comandos . . . 49
  - 2.3.4. Preparación para ejecución . . . 53
  - 2.3.5. Ejecución . . . 54
  - 2.3.6. Chequeo y parseo de resultados . . . 57

3. Algoritmos y experimentos **58**
- 3.1. Técnica . . . 58
- 3.2. Algoritmos trivialmente paralelizables . . . 64
  - 3.2.1. Experimentos . . . 65
  - 3.2.2. Resultados y Análisis . . . 66

3.2.3. Conclusión . . . 73
3.3. Multiplicación de matrices . . . 74
3.3.1. Técnica de paralelizado . . . 75
3.3.2. Experimentos . . . 76
3.3.3. Resultados . . . 76
3.3.4. Conclusión . . . 79
3.4. Heapsort . . . 81
3.4.1. Técnica de paralelización . . . 83
3.4.2. Experimentos . . . 84
3.4.3. Resultados . . . 84
3.4.4. Conclusión . . . 87
3.5. FFT - Fast Fourier Transform . . . 88
3.5.1. Introducción . . . 88
3.5.2. Técnica de paralelización . . . 89
3.5.3. Funciones intrínsecas . . . 94
3.5.4. Experimentos . . . 94
3.5.5. Resultados y análisis . . . 96
3.5.6. Conclusión . . . 101
3.6. Resumen . . . 102

4. Conclusiones **103**
4.1. Futuro trabajo . . . 104

Capítulo 1

Introducción

El avance tecnológico en la fabricación de circuitos integrados permite alcanzar niveles de integración cada vez mayores, permitiendo implementar múltiples procesadores en una misma pastilla con mayor cantidad de núcleos, memoria, o incluso mejores interfaces. Estos procesadores pueden ser de propósito general o incluso tener diferente grado de especialización dentro de un sistema de cómputo heterogéneo, optimizando su uso para reducir el gasto energético. Algunos de estos cuentan con múltiples unidades funcionales que permiten la ejecución de operaciones en paralelo. Un ejemplo paradigmático son las instrucciones vectoriales, clasificadas como `SIMD` (*Single Instruction Multiple Data*), que permiten aplicar la misma operación a varios elementos de un vector simultáneamente. Otros cuentan, incluso, con instrucciones especiales para cálculos no triviales, como por ejemplo encriptación, descompresión, o incluso para el procesamiento de audio y video. También, pueden llegar a ejecutar las operaciones fuera de orden, cambiando la secuencia original del programa para poder procesarlas más eficientemente pero de forma tal que se preserve la coherencia entre ellas.

Si bien esto tiende a mejorar los tiempos de cómputo, un procesador de propósito general está diseñado para lidiar con cualquier tipo de carga de trabajo (*workload*) sin considerar sus potenciales características distintivas que podrían permitir alguna clase de optimización. Por eso, a la hora de resolver problemas específicos, los procesadores de propósito general pueden ser superados por sistemas especialmente diseñados para propósitos específicos.

En este lugar entran en juego los **aceleradores**. Los más comunes son los aceleradores de vídeo que permiten, por ejemplo, procesar imágenes o descomprimir un flujo (*stream*) de vídeo de forma muy eficiente sin consumir recursos del procesador principal.

Un acelerador de vídeo contiene una gran cantidad de procesadores que pueden operar simultáneamente, realizando las mismas operaciones sobre una gran cantidad de datos. Esta masiva capacidad de cómputo paralelo resulta muy útil a la hora de trabajar con imágenes o señales, resultando incluso atractiva para otros problemas como muchos de los que aparecen en diferentes áreas del cómputo científico.

No obstante, esta ventaja no es aprovechable en todos los contextos. Por ejemplo, cuando el problema requiere que el código tome distintos caminos para distintas partes de su entrada, es decir, el flujo del programa diverge y para algunas partes de su entrada toma diferentes caminos. Otro caso en los que estos aceleradores no pueden ser aprovechados es cuando la cantidad de datos de entrada a procesar es limitada y los recursos de cómputo que posee son sub-utilizados.

A la hora de diseñar hardware que solucione eficientemente un problema, tenemos que tener en cuenta que el resultado será poco flexible o estará limitado a una familia de problemas. Por ejemplo, un procesador que implementa instrucciones para evaluar funciones de

hash, estará limitado a las funciones de *hash* implementadas, algunas de las cuales pueden compartir parte de la implementación. Si se desea cambiar o agregar otras funciones de *hash*, es posible que ya no se pueda hacer uso de la operatoria común a las instrucciones de *hash* ya existentes.

A pesar de todo, el hardware no necesariamente está libre de errores y es posible que se descubran, por ejemplo, problemas de seguridad. Tal es el caso de algunas vulnerabilidades presentes en ciertos procesadores modernos que pueden ser explotadas por Spectre [1] y Meltdown [2]. El hardware, habitualmente, no puede cambiarse una vez que ha sido producido, pero si éste fuera maleable entonces sería posible modificarlo para ajustarse ante estas eventualidades, potencialmente con poco impacto. Dicho esto, vulnerabilidades como las mencionadas, se podrían mitigar con una simple actualización de software, pero con una caída drástica en la desempeño del procesador.

Esta y otras situaciones se vuelven más críticas cuando la tarea que se realiza es muy específica, los límites de desempeño son aún mayores, y el costo del equipo es muy alto, como podría ser una antena de telefonía, un sistema de control de una planta industrial o un sistema de navegación aéreo.

Los `ASIC`s (*Application-Specific Integrated Circuits*) suelen ser una forma eficiente y compacta de implementar hardware de propósito específico. Sin embargo, resultan muy rígidos una vez fabricados y sus costos de producción son altos, ya que habitualmente se trata de una relativamente baja cantidad de unidades producidas. Típicamente, utilizan tecnología `MOS` (*metal-oxide-semiconductor*) y pueden manejar circuitos complejos de millones de compuertas lógicas. Un `ASIC` puede llegar a incluir microprocesadores enteros, bloques de memoria, y otras unidades funcionales.

En contraste, las FPGAs (*Field-Programmable Gate Array*) consisten en una grilla de celdas de distinta especialización que pueden conectarse de diversas formas, obteniendo un sistema mucho más flexible. Las celdas implementan desde tablas lógicas y bloques de memoria hasta operaciones complejas como multiplicación y acumulación.

Si bien la FPGA puede integrarse en un `ASIC`, el *routing* que indica cómo se conectan sus celdas, así como la configuración de cada celda no están fijos. Esta información se almacena y se lee desde una memoria que puede ser modificada, permitiendo alterar la lógica del circuito y su funcionalidad.

Al implementar hardware sobre estas celdas mediante una configuración modificable estamos sacrificando eficiencia a cambio de un enorme grado de flexibilidad, especialmente si se lo compara con la implementación fija de un `ASIC`. A su vez, las FPGAs aparecen como una opción extremadamente útil para explorar nuevos diseños y soluciones, ayudando a reducir los costos y tiempos de desarrollo. En contraposición, implementar hardware en una FPGA es más complejo que hacerlo directamente en hardware, pero así como existen herramientas para generar una implementación en un `ASIC` a partir de un diseño de alto nivel, también existen herramientas similares que generan una configuración apropiada para que un hardware se implemente en FPGA.

Cuando se trata de hardware que está implementado en una FPGA, realizar una actualización sería, entonces, una cuestión de cambiar el diseño y permitir que las herramientas calculen el nuevo *routing* y la configuración de celdas adecuada. Es decir, una actualización de hardware se podría lograr mediante una *sencilla* actualización de software.

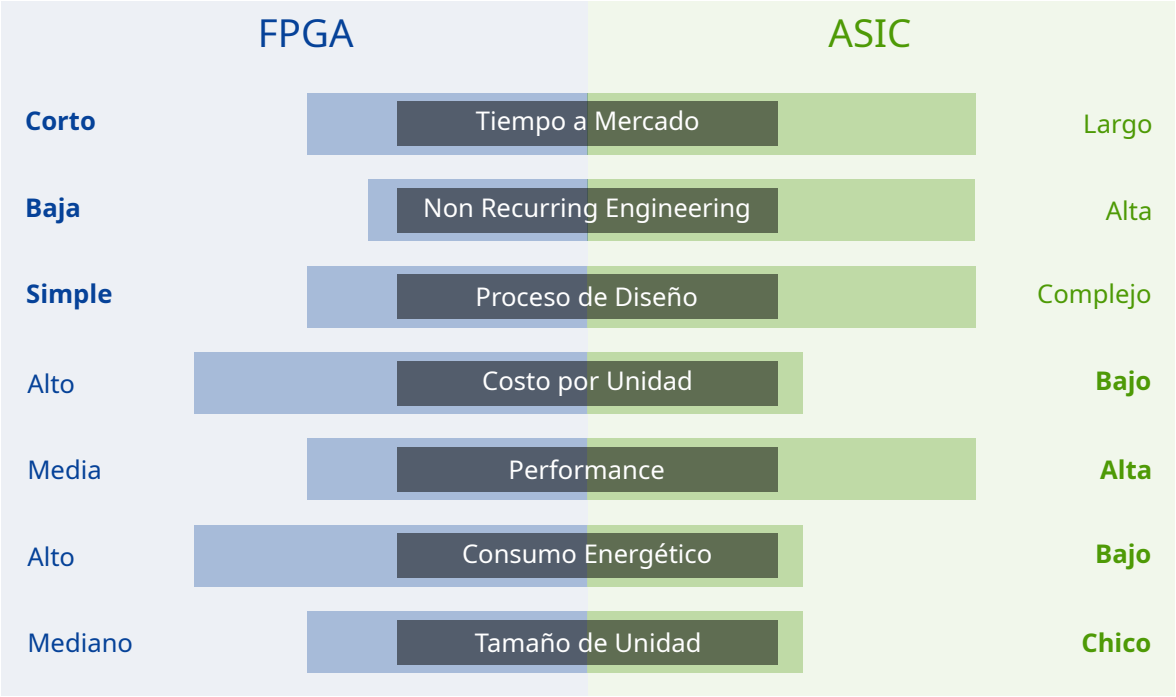


Figura 1.1: Comparación entre FPGA y `ASIC`. Se destacan los puntos favorables a cada tecnología en negrita.

En la figura 1.1 podemos ver un diagrama que compara distintas características entre FPGA y `ASIC`. Factores como el tiempo a mercado, *non recurring engineering*[1] y el proceso de diseño son puntos favorables de las FPGAs. En cambio, factores como el costo por unidad, la *performance*, el consumo energético y el tamaño de unidad son favorecidos por los `ASIC`.

Dado este grado de flexibilidad, un componente que se implementa en FPGA recibe el nombre de **softcore**. Pueden ir desde hardware relativamente simple como puede ser una máquina de estados o un sumador, hasta procesadores *multi-core* con varios niveles de *cache*. El límite dependerá de la cantidad de celdas disponibles, que es fija para una FPGA dada.

Supongamos que tenemos un diseño de un componente que puede realizar una tarea de cómputo de manera muy eficiente, de forma tal que un sistema podría hacer la delegación (*offloading*) de la tarea, y quisiéramos explorar distintas variantes de este acelerador usando una FPGA. Podríamos utilizar la misma FPGA como punto de contacto con el sistema principal y diseñar el hardware necesario para la comunicación.

En este escenario, sería posible que necesitemos de un conjunto de *softcores* para resolver nuestro problema y uno de ellos podría ser un procesador. Al ser un *softcore*, existe la posibilidad de realizarle modificaciones hasta ajustarlo a nuestras necesidades, e incluso actualizarlo si éstas cambian con el tiempo.

Las FPGAs resultan un camino válido para el diseño, prototipado y construcción de aceleradores: permiten implementar hardware para realizar el procedimiento que se desee, siempre dentro de sus limitaciones de recursos. Más aún, esto se puede lograr con un alto grado de paralelismo y con la posibilidad de acceder eficientemente a eventos externos de entrada-salida.

[1] El costo total de las etapas de creación de un producto: investigación, diseño, desarrollo y testeo.

1.1. Motivación

En la actualidad, la mayor parte de los procesadores de uso general son multi-núcleo (*multi-core*). Una manera de aprovecharlos es utilizar algoritmos paralelos, especialmente útiles a la hora de procesar grandes volúmenes de datos.

Existen FPGAs disponibles comercialmente que permiten implementar procesadores *multi-core*, teniendo capacidad suficiente para añadir distintos tipos de unidades funcionales, memoria *cache*, o incluso un *pipeline* de varias etapas.

Algunos de estos procesadores implementables en FPGA están restringidos a una plataforma específica (chips de Xilinx o de Altera, por ejemplo), otros son de código abierto, y algunos están basados en procesadores existentes.

Hajduk [3] da un ejemplo de esto último con el objetivo de correr código aplicativo real y que puede aprovecharse de la flexibilidad de una FPGA. Allí diseñan y analizan tres variantes de un microcontrolador de 8 bits para FPGA basado en la familia PIC16 de Microchip que resulta ser entre 4 y 8 veces más rápido.

Hay diversos trabajos que exploran los usos de una FPGA aplicados a distintos campos.

En el área médica podemos nombrar una implementación basada en RISC-V para aplicaciones biomédicas de bajo consumo por Garcia-Ramirez et al. [4].

Un interesante producto de consumo se propone en el trabajo de Echanobe et al. [5], donde diseñan y evalúan un multiprocesador embebido para controlar parámetros ambientales en una casa inteligente y que aprende de las preferencias del usuario.

Para el área de investigación y desarrollo destacamos el trabajo de Nolting et al. [6] en donde presentan una técnica novedosa de hardware dinámicamente configurable durante la ejecución de un programa, y Liu et al. [7] donde evalúan aceleradores híbridos de hardware para inteligencia artificial con FPGAs.

Estos pocos ejemplos ponen de manifiesto el alcance de soluciones provistas por FPGAs.

Con los recursos existentes hoy en día, las FPGAs suelen estar embebidas en un híbrido que posee además una parte "dura". Esta parte no maleable puede incluir un procesador implementado en *chip* (como un `ARM`), bloques de conversión Analógico/Digital, puertos de entrada/salida y hardware adicional.

Este tipo de híbridos resulta ser el caso ideal para utilizar una FPGA como un acelerador para el procesador principal (que correspondería a la parte mencionada como dura).

En estas configuraciones existen restricciones como el ancho de banda de acceso a memoria, las unidades funcionales pre-implementadas o la cantidad de puertos de entrada/salida que limitan el funcionamiento de este tipo de dispositivos. Si bien los procesadores tradicionales (`ASIC`) actuales también tienen estas limitaciones, suelen estar órdenes de magnitud por encima de lo disponible en una FPGA (ver figura 1.1).

Dadas estas limitaciones, si se quiere ejecutar código paralelizado dentro de una FPGA, una de las opciones es implementar múltiples procesadores completos (*softcores*) con un sistema de interconexión.

Uno de los tipos de celdas más comunes en FPGA son los que implementan bloques de memoria (BRAM). Esta memoria cuenta con una muy reducida latencia, lo que permite su uso en la implementación de *caches*.

Dada la posibilidad de implementar memorias en FPGA, algunos *softcores* prediseñados implementan internamente o contienen puertos especiales que permiten acceder de manera eficiente a una BRAM configurable por el diseñador.

Tomando como punto de partida las prestaciones que presenta este tipo de hardware y considerando la necesidad de ejecutar código dentro de la FPGA, se propone estudiar formas de hacer esto mismo, poniendo foco en el uso eficiente de la memoria.

1.2. Objetivos del trabajo

Este trabajo tiene como objetivo general,

> *Avanzar en técnicas que permitan el uso eficiente de múltiples* softcores *dentro de una FPGA.*

Para alcanzar este objetivo se proponen las siguientes metas:

1. Desarrollar un *framework* de evaluación de múltiples *softcores* escalable, que permita considerar distintos sistemas y configuraciones, minimizando los cambios requeridos.

2. Identificar fuentes de las limitaciones en eficiencia y en la capacidad de escalar en sistemas multicores basados en *softcores*.

3. Proponer formas de mejorar la eficiencia de programas de propósito general sobre múltiples *softcores*.

El estudio busca hacer foco en el uso eficiente de la memoria. En particular, vamos a añadir una BRAM aislada a cada *softcore*. Analizaremos varias formas de aprovechar esta memoria estática para mejorar la performance de distintos programas.

1.3. Trabajo relacionado

Uno de los *softcores* más utilizados es el MicroBlaze, desarrollado por Xilinx y creado específicamente para su uso en FPGAs. En Matthews et al. [8] se extienden varios MicroBlaze para correr Linux SMP (*Symmetrical MultiProcessing*), un sistema operativo utilizado en procesadores comunes. Allí detallan los inconvenientes a la hora de interconectar varios MicroBlaze para soportar dicho sistema operativo. Entre ellos se encuentran las interrupciones, MMU (*Memory Management Unit*), *timers* y operaciones atómicas.

En esta tesis se considerarán algunos de los problemas mencionados en ese trabajo, haciendo foco en el uso de operaciones atómicas (herramientas fundamentales para la implementación de mecanismos de sincronización). En nuestro caso particular, la coherencia entre procesadores está garantizada por el uso del protocolo `ACE` [9]. Esto nos permite implementar funciones de sincronización por software, a diferencia de Matthews et al. [8], donde garantizan la atomicidad mediante un bloque de hardware específicamente diseñado para implementar un árbitro.

Por otro lado, la configuración para inicializar los *softcores* es diferente. En Matthews et al. [8] los MicroBlaze ejecutan código de *firmware* precargado en BRAM y reciben una señal de interrupción de otro procesador cuando son necesarios. En nuestro trabajo, en cambio, todos los MicroBlaze comienzan ejecutando el mismo código directamente en memoria principal, logrando un comportamiento determinístico en el encendido del sistema.

Adicionalmente, en este trabajo utilizamos `LMB` (*Local Memory Bus*) para comunicar cada MicroBlaze con una memoria BRAM aislada y `OPB` (*On-chip Peripheral Bus*) para conectar cada procesador a una *cache* L2. Bečvář et al. [10] evaluaron todas estas interfaces, junto con *Fast Simplex Link* o `FSL` (canales unidireccionales diseñados para conectar un MicroBlaze con aceleradores de hardware), donde conectaron un coprocesador con el MicroBlaze. Se observó que `LMB` y `FSL` son similares en desempeño, ambas presentando mejoras respecto de `OPB`, y con un menor uso del área de FPGA en el caso de `FSL`.

Otro ejemplo que ilustra el uso de Microblaze es el caso de Cvikl et al. [11], donde se implementa un detector de pulsos para un ECG. Esta implementación se compara entre un procesador PowerPC integrado contra un MicroBlaze, y se llega a la conclusión que para la aplicación realizada no es necesario un microprocesador de alto desempeño como el PowerPC ya que el MicroBlaze resulta suficiente.

El entorno utilizado es similar a los experimentos realizados en este trabajo, la señal de entrada que el MicroBlaze debe procesar está en memoria DDR junto con el código del programa. Este código es lo suficientemente grande como para no entrar completo en una memoria BRAM, por lo cual se hace uso del `OPB`. Además utilizan el LMB para acceder a una BRAM dedicada para el MicroBlaze. Esta memoria se utiliza para mantener un *stack* y un *heap*.

Es posible que, de poder particionar el código y siguiendo el espíritu de nuestro trabajo, se pueda hacer que el procesador copie las partes de uso más frecuente a BRAM o las vaya intercambiando a medida que necesitan ejecutar. Hacer esto en conjunto con usar una *cache* de instrucciones podría llegar a reducir los accesos a memoria y mejorar la performance.

Una comparación general entre *softcores*, así como un análisis comparativo de performance entre MicroBlaze de Xilinx y LEON3 (procesador open-source que implementa el set de instrucciones SPARC V8 desarrollado por Sun Microsystems) puede verse en Makni et al. [12].

Como observación interesante, se destaca que el uso de recursos de FPGA resulto similar entre ambos procesadores. Esta diferencia es levemente menor para el LEON3 en general, con la excepción de un uso importante de LUTs (*Look-Up tables*) debido a una FPU más completa.

Por otro lado, el trabajo analiza los tiempos de ejecución para un JPEG encoder y decoder utilizando una arquitectura de 1, 2 y 4 MicroBlaze contra una similar con LEON3, resultando el *single-core* MicroBlaze 2 veces más rápido que su contraparte y 3 veces más rápido para el caso *quad-core*. En dicha arquitectura los *softcores* tienen una relación *master-slave* similar a la de nuestro trabajo, donde uno de ellos se encarga de informar a los demás cuando deben comenzar a ejecutar sus tareas.

En el libro *The Zynq Book* [13], se describe con detalle la plataforma Zynq-7000 de Xilinx (utilizada en este trabajo) y se ofrece una comparación entre un MicroBlaze implementado en la FPGA y un ARM *dual-core* incluído en la plataforma, teniendo el *dual-core* aproximadamente 20 veces más capacidad de procesamiento que el *softcore* (5000 DMIPS vs 260 DMIPS)[2].

Advierten en el texto que las diferencias en la performance pueden deberse a un set de instrucciones más extenso para el ARM, una FPU de precisión doble (la del MicroBlaze es de

[2]DMIPS o *Dhrystone Millions of Instructions Per Second* es la cantidad de operaciones por segundo que alcanza un procesador corriendo el test standard Dhrystone. Este test está especialmente diseñado para que el procesador corra un conjunto representativo de operaciones sin ser representativo de una aplicación real.

precisión simple), una configuración de *cache* de dos niveles (el *softcore* permite configurar sólo un nivel), y a que se trata de una comparación entre un *dual-core* (ARM) y un *single-core* (el MicroBlaze). Es posible que esta brecha se reduzca para una arquitectura *multi-core* de MicroBlaze y para aplicaciones restringidas a punto flotante de precisión simple.

Si al lector le interesa conocer más sobre la plafatorma, el libro también detalla el proceso de diseño de hardware para FPGA, y ofrece varias aplicaciones que hacen uso de una FPGA, entre otros puntos relevantes.

Por último, se hace mención al *benchmark* Dhrystone [14] como un ejemplo de *benchmark* sintético. Este tipo de *benchmark* tiene la intención de evaluar una o más características del sistema, procesador o compilador mediante programas simples que ayudan a cubrir estadísticamente los usos que sean relevantes.

La performance obtenida, sin embargo, no es necesariamente indicativa de la que habría para una aplicación real.

Otro tipo de *benchmark* es el de aplicación completa, donde utilizan código de algoritmos reales (comunes y de uso frecuente). Un punto intermedio es un *benchmark* de sólo algoritmos, en el cual se comparan algoritmos en áreas de aplicación especiales.

En nuestro trabajo haremos uso de estos dos últimos tipos de *benchmark* para analizar la diferencia de performance entre las distintas variantes. Principalmente porque es necesario entender el código que estamos ejecutando y cómo aplicar en éste las técnicas de optimización.

1.4. FPGA

Como mencionamos anteriormente, una FPGA (*Field-Programmable Gate Array*) consiste en una grilla de celdas de distinta especialización que pueden conectarse de diversas formas (Figura 1.2).

En esta sección entraremos en más detalle. Hablaremos sobre los tipos de celdas más comunes, así como las conexiones disponibles en la grilla, hasta las aplicaciones y limitaciones de las FPGAs. Además, introduciremos detalles del proceso de diseño, terminologías y protocolos útiles.

1.4.1. Tipos de celdas

Existen diferentes tipos de celdas, cada una diseñada para resolver un problema específico. A continuación se describen los tipos básicos de celdas que se pueden encontrar dentro de una FPGA.

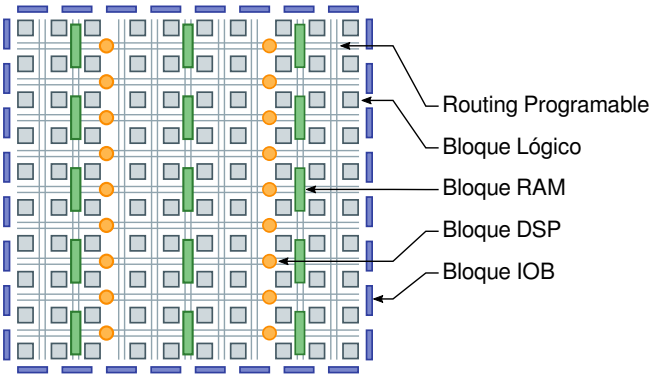


Figura 1.2: Estructura básica de una FPGA

CLB

Cada celda lógica configurable (CLB, *Configurable Logic Block*) puede tener LUTs (*Look-Up Table*) de N bits, así como multiplexores, *flip-flops* y otros componentes lógicos (Figura 1.3). Las CLB pueden ser tan simples o tan complejas como sean diseñadas por el fabricante. Son la unidad lógica más básica de la FPGA y suelen ser los bloques de mayor presencia dentro del circuito integrado.

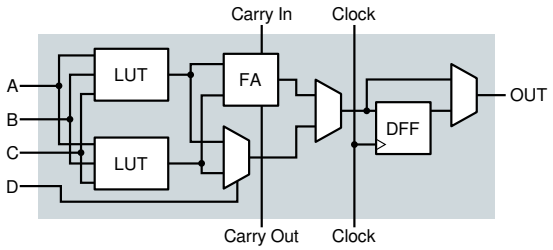


Figura 1.3: Una celda lógica con una LUT de 4 bits, un sumador *full-adder* (FA), un *flip-flop* D (DFF) y varios multiplexores.

BRAM, DSP, IOB

La mayoría de las FPGAs modernas contienen otros tipos de bloques además de los CLB. Estos están especializados para implementar ciertas operaciones de manera rápida y eficiente, o bien cubrir otros usos. Entre ellos destacamos los bloques de RAM (*Block RAM* o BRAM), bloques para procesamiento digital (DSP, *Digital Signal Processing*) y bloques de entrada/salida (IOB, *Input-Output Block*)[15].

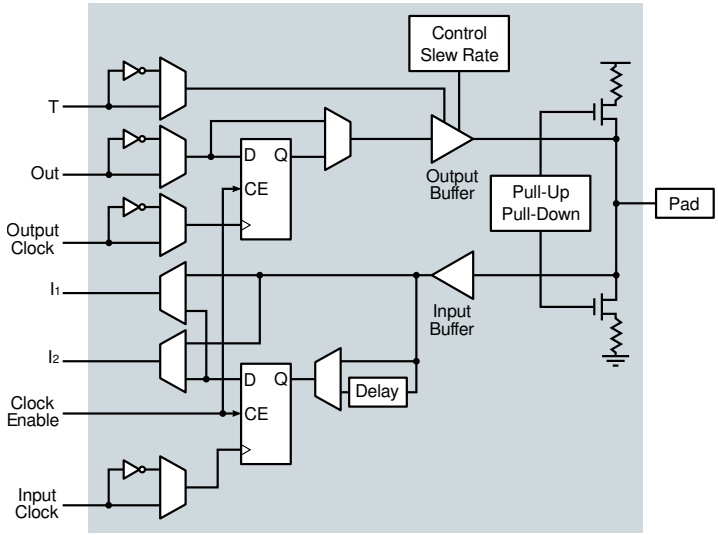


Figura 1.4: Diagrama de un IOB.

A la hora de implementar la *cache* L1 de un procesador, si bien teóricamente se pueden usar los *flip-flops* de las CLB, su cantidad limitada haría inviable alcanzar un tamaño útil. Más aún, se estarían desaprovechando los otros componentes de la celda. Si en cambio se utilizan bloques BRAM (figura 1.6), estaremos aprovechando un componente especializado para esta tarea, y con mucha mayor capacidad.

Por otra parte, una de las operaciones más utilizadas del procesamiento digital es la multiplicación y acumulación (MAC, *Multiplier-Accumulator*). Los bloques DSP permiten

resolver estas operaciones sin sacrificar grandes cantidades de celdas lógicas (figura 1.5). Un componente que puede hacer uso de bloques DSP es una FPU (*Floating Point Unit*).

Finalmente, hay ocasiones en las que se busca interactuar con algún componente externo a la FPGA. Por ejemplo, un botón o interruptor, un LED, o alguna otra señal digital. Los IOB permiten implementar esta interfaz como puertos (Figura 1.4).

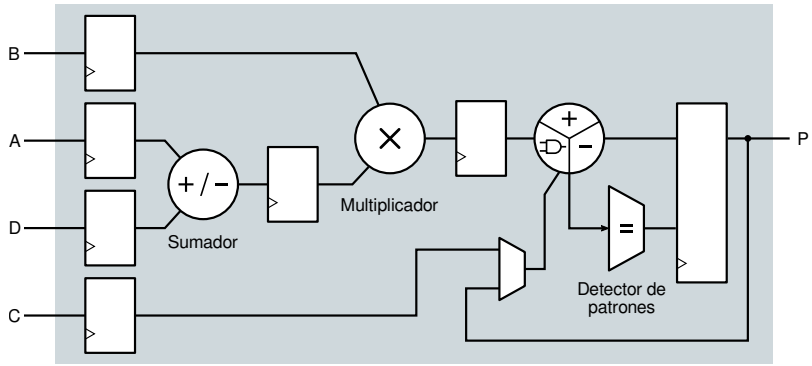


Figura 1.5: Diagrama de un bloque DSP implementando un multiplicador y acumulador de 48 bits.

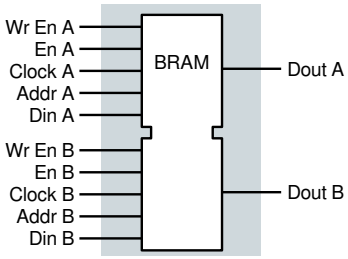


Figura 1.6: Diagrama de puertos de un BRAM que soporta escritura / lectura simultánea.

1.4.2. Interconectividad

Como ya mencionamos, los bloques o celdas están dispuestos en forma de grilla. La salida de una celda puede conectarse con la entrada de otra, o continuar su camino hacia la próxima celda. Los *switches* son los elementos que definen estas conexiones, encontrándose en las esquinas de cada celda en la grilla (figura 1.7).

Hay distintos tipos de líneas (o cables) que llegan a los *switches*. Se distinguen principalmente por su longitud en términos de cuántas celdas atraviesan. Existen además líneas que están dedicadas a señales específicas. Un ejemplo de esto son las señales de sincronización o *clock* (`CLK`).

Es posible que un mismo *clock* se utilice en varias celdas, tal vez distribuidas en distintas áreas de la FPGA, requiriéndose que la señal sea lo más "simultánea" posible. Hay tipos de líneas "globales" o de *routing* global que distribuyen una señal de *clock* con bajo *skew*.

Mucho antes de alcanzar el 100 % del uso de bloques[3], la arquitectura de grilla suele ser el mayor limitante para implementar un diseño en FPGA [16, 17], ya sea por la falta de conexiones o por los *delays* que se introducen al aumentar la complejidad de las rutas que deben tomar las señales.

[3]En este caso nos referimos a todos los tipos de bloque disponibles. Es común que no pueda implementarse un diseño particular porque la FPGA utilizada no posee suficientes bloques especializados como DSP o BRAM.

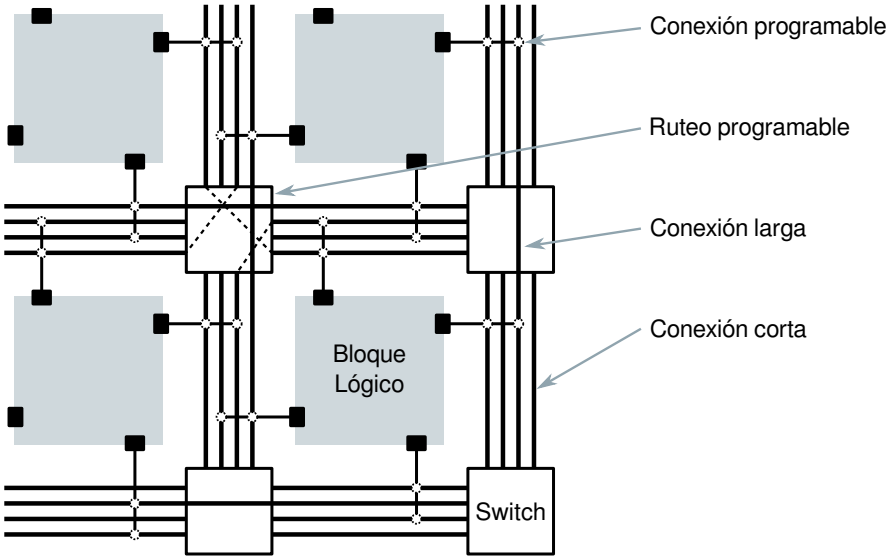


Figura 1.7: Diagrama de una parte de la grilla.

1.4.3. PL y PS

Dentro del mismo chip que contiene la FPGA pueden existir otros circuitos integrados como procesadores o controladores, a los que se accede utilizando una interfaz dedicada. Este tipo de arquitecturas divide el circuito integrado en dos partes: la "flexible" que permite programar el hardware, también llamada PL (*Programmable Logic*), y su contraparte "dura" con componentes embebidos, denominada PS (*Processing System*).

En un sistema de procesamiento, se podría dedicar la parte PS a hacer operaciones de carácter serializado, típica de los procesadores, y utilizar la gran capacidad de procesamiento paralelo de la PL para problemas específicos.

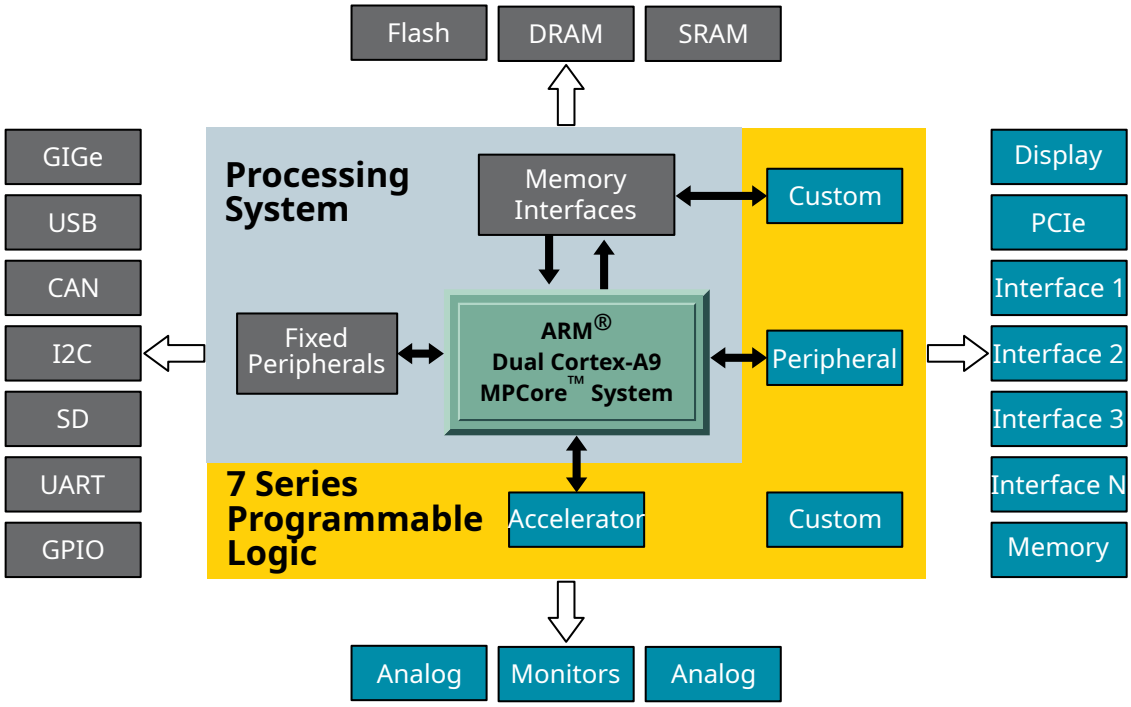


Figura 1.8: Un chip con una FPGA y un ARM integrado. La zona amarilla corresponde a la parte PL.

Las interfaces disponibles podrían permitir, por ejemplo, conectar la lógica programable al controlador de memoria, a señales externas e incluso a señales del mismo procesador (figura 1.8).

Un ejemplo que utiliza ambas partes (tanto la lógica programable como el sistema de procesamiento) podría ser un sistema de post-procesador de video en tiempo real. Por un lado, se utilizaría la PL para implementar filtros por hardware y, por el otro, el procesador en la PS se ocuparía de ejecutar un sistema operativo donde se correría software ya existente para la transmisión del vídeo.

1.4.4. Limitaciones

La FPGA utiliza celdas para implementar hardware. Las celdas están compuestas por varios componentes (LUT, FF, multiplexores, etc) lo que les aporta un grado de configurabilidad. Comparando con una implementación directa, esto implica una mayor cantidad de componentes. El *routing* además introduce retardos que suman mayores limitaciones a las implementaciones en FPGAs. Todo esto se traduce en una mayor área de chip, mayor consumo energético, y peor *performance* que una implementación nativa en hardware utilizando un ASIC (figura 1.1).

El problema de la temporalidad entre eventos (o *timing*) se debe tener en cuenta durante la implementación de cualquier diseño. A la hora de elegir las celdas y definir el *routing* de las señales, el proceso debe garantizar que el *timing* no supere ciertos límites. Esto puede devenir en un proceso lento y es posible, incluso, que no encuentre una solución. Es decir, que no haya una implementación válida en la FPGA disponible.

Una problemática asociada al *timing* es la frecuencia final de trabajo que tendrá un diseño una vez implementado. Esta frecuencia máxima de trabajo no puede ser asignada a *priori*, sino que será resultado de cómo fue implementado el diseño.

En ASIC, en cambio, al no depender de una arquitectura de grilla, hay mayor libertad a la hora de definir los componentes, sus ubicaciones e interconexiones. Esta libertad desemboca en una gama de implementaciones prohibitivas para FPGA donde el *timing* y la frecuencia de trabajo, por ejemplo, pueden adecuarse a las necesidades.

Implementaciones sobre distintos dispositivos pueden utilizar más celdas de algún tipo particular, dependiendo de sus características, incluso, tipos de celdas distintos, llevando a una diferencia de desempeño que impacta directamente sobre el resultado implementado y que no depende directamente del diseño del mismo.

Por último, se considera la limitante del área disponible. No se podrá implementar un diseño si el proceso requiere el uso de componentes cuyo número supera los disponibles en la FPGA.

1.4.5. Tipos y usos

Considerando que el área del chip es limitada, existen en el mercado múltiples tipos de FPGAs. Cada una se caracteriza por las prestaciones con las que cuenta, su frecuencia de trabajo, grado de especialización y tamaño.

Si contáramos con una FPGA con una gran cantidad de BRAM, podríamos aprovecharla en casos que hagan uso intensivo de la memoria. En cambio, dado el menor espacio en el chip para otros bloques (lógicos, DSP, IOB, etc), las tareas que requieran más cómputo o mayor complejidad podrían no ser implementables.

Por otro lado, si es necesaria una FPGA para monitorear eventos externos y controlar dispositivos, se buscaría un diseño con gran cantidad de bloques IOB.

Su facilidad de reconfigurabilidad la hacen una plataforma indispensable para prototipar nuevos microprocesadores o microcontroladores antes de su implementación final en ASIC, más rápida y eficiente pero donde es muy costoso hacer cambios posteriores. También sirven como una herramienta de estudio, siendo posible implementar en una FPGA procesadores o piezas de hardware ya existentes.

En ocasiones, se necesita una solución de hardware embebido que excede a un microcontrolador, pero que es demasiado compleja de implementar con otros componentes (por ejemplo, una interfaz para comunicar dos circuitos integrados). Por otra parte, hay procesos que son muy lentos al ser ejecutados en un procesador estándar, o quizás no se dispone de una GPU o un procesador con suficientes núcleos. Es incluso posible que la tarea que se busca implementar no sea tan compleja como para requerir su uso.

Las FPGAs proveen una opción configurable, rápida (implementada en hardware) y altamente paralelizable.

1.4.6. Diseño

Para implementar una solución en FPGA, se puede recurrir tanto a la codificación en un lenguaje de descripción de hardware, como al uso de una herramienta de diseño en bloques de alto nivel. Generalmente en diseños complejos se requiere hacer uso de ambos enfoques.

Ejemplos de lenguajes de descripción de hardware son `Verilog` y `VHDL`. El primero es más reciente y está basado en el lenguaje de programación C. `VHDL` es más explícito en cuanto a su sintaxis y está basado en los lenguajes Ada y Pascal. Ambos HDL (*Hardware Description Language*) son utilizados ampliamente en la industria.

Software como Vivado (de Xilinx) o Quartus (de Altera) cuentan con una interfaz gráfica que permite armar diseños con bloques e incluso implementar bloques mediante HDL. Este tipo de herramientas permiten establecer las conexiones entre los distintos bloques (*IP Cores*) disponibles, ya sea hacia puertos externos (entrada/salida), o hacia otros *IP Cores*. Si bien el diseño en bloques es luego traducido a HDL, facilita la etapa de diseño al permitirle al diseñador abstraer módulos y tratarlos como una caja negra.

Se le dice **síntesis** al proceso que toma una descripción en HDL y genera un circuito lógico (*netlist*) compuesto por compuertas, *flip-flops* y otros elementos interconectados. Además, verifica la sintaxis y analiza que el diseño sea optimizado para la arquitectura elegida (FPGA o ASIC). En este punto se puede obtener un estimado de los recursos necesarios para implementarlo en ésta.

El proceso de **implementación** se divide en tres partes:

- **Translate**: En primer lugar, se combina la *netlist* con restricciones físicas como los puertos (pines, *switches*, botones, etc) y restricciones de *timing*.
- **Map**: Luego, se mapea el circuito dividiéndolo en sub bloques que pueden asignarse a los bloques que dispone la FPGA elegida.
- **Place and Route**: Finalmente, se ubican estos sub bloques en la FPGA y se hace el *routing* respetando las restricciones.

Puede hacerse una verificación durante varias de estas etapas. Por ejemplo, se puede hacer una simulación del comportamiento del diseño a partir del HDL previo a la síntesis. También se puede simular el circuito generado en la etapa *Translate*, o luego de la etapa de *Place and Route*.

El último paso que resta es programar la FPGA. Para ello se traduce a *bitstream* el diseño generado en la etapa de implementación (un formato que la FPGA comprende). Dependiendo del tipo de FPGA, se puede requerir de una memoria no volátil externa, o incluir una en el mismo chip. En cualquier caso, este *bitstream* deberá ser escrito en memoria y la FPGA podrá finalmente ser configurada con nuestro diseño.

IP Core

Un *IP Core* (*Intellectual Property Core*) consiste básicamente en una entidad con puertos de entrada (*slave*) y de salida (*master*) sobre la que pueden especificarse interacciones entre las señales, ya sea mediante otros *IP Core* interconectados, o utilizando HDL.

Para que *IP cores* con cierto grado de complejidad puedan comunicarse entre sí, generalmente se hace uso de un protocolo llamado `AXI` (o su variante `ACE`, que garantiza coherencia). También pueden implementarse interfaces con protocolos personalizados.

AXI

`AXI` proviene de sus siglas en inglés (*Advanced eXtensible Interface*), siendo su última versión `AXI4`. Es parte del estándar `ARM AMBA 3.0` que fue originalmente desarrollado por `ARM` para usarse en microcontroladores. Tras revisiones y extensiones se ha convertido en un estándar de comunicación entre componentes de un chip [13].

Cuenta con tres variantes:

- `AXI4`: Comunicación entre varios dispositivos de alta *performance* utilizando mapeos en memoria. Es decir, se reserva un rango de direcciones al cual se puede leer o escribir para comunicarse con otro dispositivo. Con una dirección base permite hasta 256 palabras (words) enviadas en *burst*.
- `AXI4 Lite`: Comunicación simplificada, también utilizando mapeos en memoria. No permite *burst*.
- `AXI4 Stream`: Comunicación rápida entre dos dispositivos y sin utilizar mapeo en memoria. Soporta *burst* sin restricción.

ACE

La variante `ACE` es una extensión de `AXI4` con soporte para la intercomunicación entre *caches* con coherencia por hardware [9]. Esto permite, por ejemplo, que varios dispositivos puedan leer y escribir información compartida cuando el valor más reciente no se encuentra necesariamente en la memoria principal.

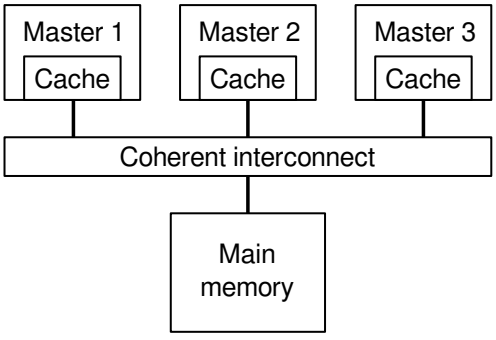


Figura 1.9: Diagrama de conexión ACE.

El protocolo `ACE` asegura que todos los *masters* accederán a la última modificación realizada en cualquier dirección mapeada. Este comportamiento se logra forzando la existencia de una sola copia del valor luego de una modificación, y permitiendo que cada *master* obtenga una copia para su *cache* local.

El protocolo permite la implementación de una política *write-back* o *write-through*.

Para comprobar la multiplicidad de copias de un valor, el protocolo `ACE` agrega un bus auxiliar (*snoop bus*). Este bus permite la comunicación entre los *masters* para verificar quiénes tienen una copia del valor de una dirección, si es que la tienen. Esto permite identificar si existe una copia del dato y, en consecuencia, invalidarla para que otra de las *cache* pueda utilizarla. La operación requiere un mecanismo de transacciones distribuidas sobre las *caches*, que también es implementado sobre el *snoop bus* para garantizar la invalidación y escritura exclusiva.

Softcore processor

Un *Softcore processor* es un caso particular de *IP Core*. Se trata de un microprocesador que puede ser implementado en FPGA, es decir que puede ser sintetizado.

Existen múltiples ejemplos de este tipo de procesadores, entre los que podemos destacar el Picoblaze (8 bits, de código abierto) y el MicroBlaze (32 bits y 64 bits) de Xilinx, así como también el Rocket Chip[4] que implementa el set de instrucciones de RISC-V.

Si bien todos estos ejemplos son catalogados de *softcores*, encontramos diferencias entre ellos: de la misma forma que existen microcontroladores y procesadores distintos, hay *softcores* para uso específico y otros de uso general. Algunos permiten configurarse para cumplir requerimientos específicos, como la reducción de espacio utilizado en la FPGA, mejor desempeño, o incluso una ISA (*Instruction Set Architecture*) ampliable. También podemos encontrar *Softcore processors* con versiones de código abierto o cerrado, siendo ambas implementables en FPGA.

En este trabajo vamos a utilizar MicroBlaze, del cual hablaremos más en detalle en las secciones subsiguientes.

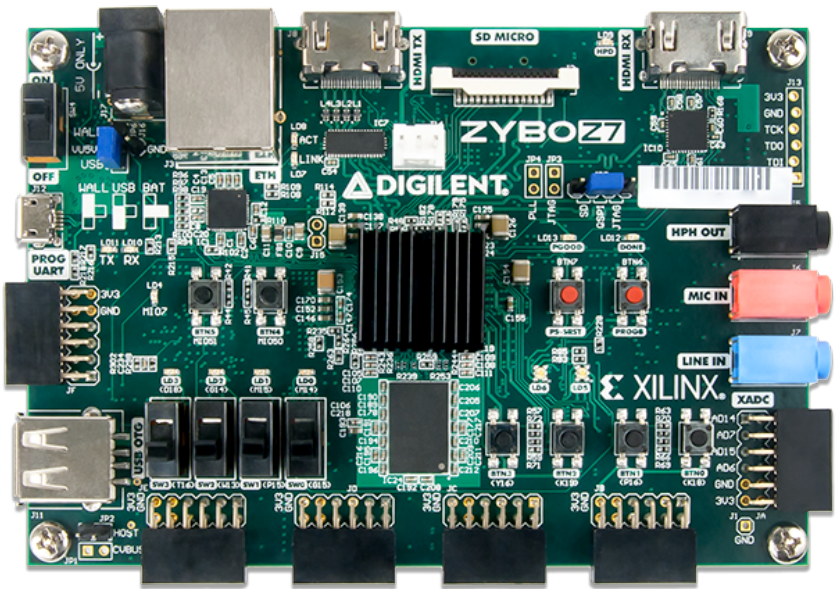


Figura 1.10: Zybo Z7 de Digilent

[4] `https://github.com/chipsalliance/rocket-chip`

1.5. Plataforma

Para este trabajo se utilizó el *Development Board Zybo Z7* de Digilent[5] (figura 1.10).

Si bien existen plataformas de desarrollo mś potentes, se eligió la presente porque contiene una FPGA con recursos suficientes para poder implementar múltiples MicroBlaze. Contiene, además, un ARM *dual-core* capaz de correr un sistema operativo, y desde el cual es posible programar la FPGA. Este tipo de FPGAs híbridas resultan ideales para realizar los experimentos planteados en este trabajo.

1.5.1. Hardware

Las características principales de la plataforma de desarrollo elegida son[6]:

- Alimentación por conector de 5 V, USB o batería.
- 1 GB de RAM DDR3 @ 1066 MHz.
- microSD.
- USB y Ethernet.
- HDMI de entrada y salida.
- Salida de audio en forma de headphone jack, line in, mic in.
- Chip XC7Z020-1CLG400C que integra:
  - ARM Cortex-A9 dual core de 667 MHz.
  - FPGA equivalente a una Artix-7
    - 106,400 flip flops
    - 630 kB de Block RAM.
    - 56,200 LUTs de 6 bits.
    - 220 DSPs.
- Varios I/O conectados al ARM o FPGA.
- Botones, switches, LEDs, algunos conectados al ARM.

1.5.2. Vivado

El software utilizado en este trabajo para construir los diagramas de bloques que permiten conectar los *IP Cores*, sintetizar y generar el *bitstream* necesario para configurar la FPGA fue Vivado 2018.3.

Vivado provee una galería de diversos *IP Cores* con distintos grados de configuración que pueden ser utilizados en cualquier desarrollo. Por ejemplo: multiplexores, procesadores, caches, y controladores de memoria. Ejemplos que además de estar pre-diseñados pueden ser implementados de forma eficiente en el hardware específico que se tenga disponible.

También permite crear *IP Cores* básicos con una cantidad definible puertos AXI de entrada/salida. Si bien la implementación de estos puertos es compleja por la cantidad de señales y la complejidad del protocolo, Vivado genera código en Verilog o VHDL de manera automática durante la creación del *IP Core*. El diseñador tiene libertad de hacer las modificaciones que crea adecuadas sobre el código.

[5] `https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/start`

[6] Zybo Z7 Board Reference Manual

El software ofrece al usuario la posibilidad de ajustar los parámetros de cada etapa (síntesis, implementación, generación de *bitstream*). Por ejemplo, es posible definir una prioridad que optimice el uso de área a expensas del *timing*. Incluso se le puede incrementar el tiempo dedicado a los procesos de síntesis e implementación a fin de encontrar una mejor solución.

Vivado permite simular la implementación antes de que sea programada en el dispositivo. Posee además un IDE específico para implementar y debuggear código sobre MicroBlaze. El programa queda compilado como un archivo .elf y se carga en la memoria BRAM asociada al MicroBlaze, dentro del *bitstream*.

Si bien esto puede ser suficiente para otros desarrollos (como sistemas embebidos), en este trabajo estamos haciendo uso de la FPGA como acelerador y requerimos tanto del sistema operativo en el ARM como del acceso a memoria DDR por parte de la FPGA, los cuales escapan el uso del IDE y exceden el alcance del simulador.

1.5.3. MicroBlaze

El MicroBlaze [18] es un *softcore processor* RISC de Xilinx[7]. No es *open source* pero está disponible para su implementación en FPGAs de Xilinx de manera gratuita.

Posee su versión de un GNU *toolchain*, permitiendo compilar, linkear y debuggear programas en C y C++. Tener una *toolchain* es fundamental para desarrollos complejos, ya que estandariza la mayor parte del código a escribir y simplifica la generación de programas. Esto permite, además, poder cambiar de *softcore processor* por otro que también posea una GNU *toolchain* y correr los mismos programas, con modificaciones menores.

Es un *IP Core* configurable en varios sentidos, lo que permite su uso en diversos proyectos. En la figura 1.11 podemos ver un ejemplo de su representación en Vivado.

Utilizamos la configuración por defecto con algunos cambios: sin interfaz de *debug* y con el agregado de dos buses ACE independientes para datos (M_ACE_DC) e instrucciones (M_ACE_IC). La configuración por defecto incluye puertos independientes para acceder a una memoria local (DLMB, ILMB), señales de *Clock* (Clk) y *Reset*. El puerto INTERRUPT permite atender interrupciones externas que no utilizaremos en este trabajo y no puede ser ocultado.

En la Sección 2.2.2 veremos la configuración elegida para este trabajo. En [19] se presenta el *IP Core* con mayor detalle.

Figura 1.11: Ejemplo de *IP Core* MicroBlaze en Vivado configurado para su uso en este trabajo. A la izquierda están los puertos de control (*slave*) y a la derecha se encuentran los de acceso a memoria (*master*).

[7] `https://www.xilinx.com/products/design-tools/microblaze.html`

A fines ilustrativos, destacamos el siguiente listado de opciones disponibles:

- 32 bits o 64 bits de tamaño de palabra.
- Implementación priorizando menor uso de área en la FPGA, mejor *performance* o mayor frecuencia.
- Pipeline 3-stage, 5-stage o 8-stage.
- ISA básica o con instrucciones adicionales como multiplicación y división entera (entre otras).
- FPU opcional: básica (operaciones aritméticas y de comparación), extendida (conversión *int-float* e instrucción raíz cuadrada).
- *Cache* opcional L1 para instrucciones y/o data.
- *Branch prediction* y *branch cache* opcionales.
- MMU opcional (Memoria Virtual).
- Bus de memoria local (BRAM) opcional.
- Bus AXI o ACE para data e instrucciones externas
- Interfaz de *debug* por hardware.

En la figura 1.12 podemos ver un ejemplo de diagrama de bloques autogenerado por Vivado a la hora de agregar un MicroBlaze sin *IP Core debugger*, junto con las conexiones entre *IP Cores* que Vivado puede realizar automáticamente. Entre ellas, se destaca la BRAM (microblaze_0_local_memory) que queda conectada con los puertos DLMB y ILMB.

Los puertos M_AXI_DP y M_AXI_IP son puertos AXI4 que en la figura 1.11 reemplazamos por variantes ACE.

A esto le agregamos una conexión con la memoria DDR utilizando interconectores AXI (axi_mem_intercon).

El *IP Core ZYNQ* representa la interfaz con el ARM y el resto del hardware integrado, denominada PS. *Processor System Reset* permite manejar el *reset* de todo el sistema (el AXI Interconnect, el MicroBlaze, la BRAM)[8].

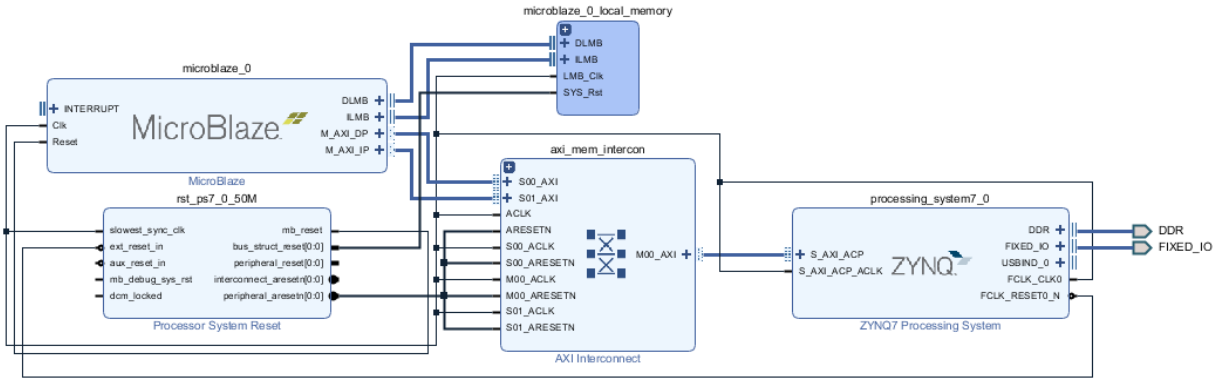


Figura 1.12: Diagrama de bloques con un MicroBlaze. Incluye una BRAM (microblaze_0_local_memory) y acceso a memoria DDR mediante un interconector AXI (axi_mem_intercon) conectado al puerto ACP del *IP Core* ZYNQ.

[8] `https://docs.xilinx.com/v/u/en-US/pg164-proc-sys-reset`

La figura 1.13 muestra la implementación del diseño marcando las celdas que van a utilizarse y mostrando el área total disponible de la FPGA.

En la mayoría de los casos, no tendremos un 100 % del uso del área, aún en diseños más complejos. Esto se debe, como mencionamos previamente, a restricciones de *timing*, o incluso a que ciertos tipos de celdas con menor disponibilidad (por ejemplo, las BRAM o las DSP) quedan totalmente utilizadas, lo cual impide implementar el diseño.

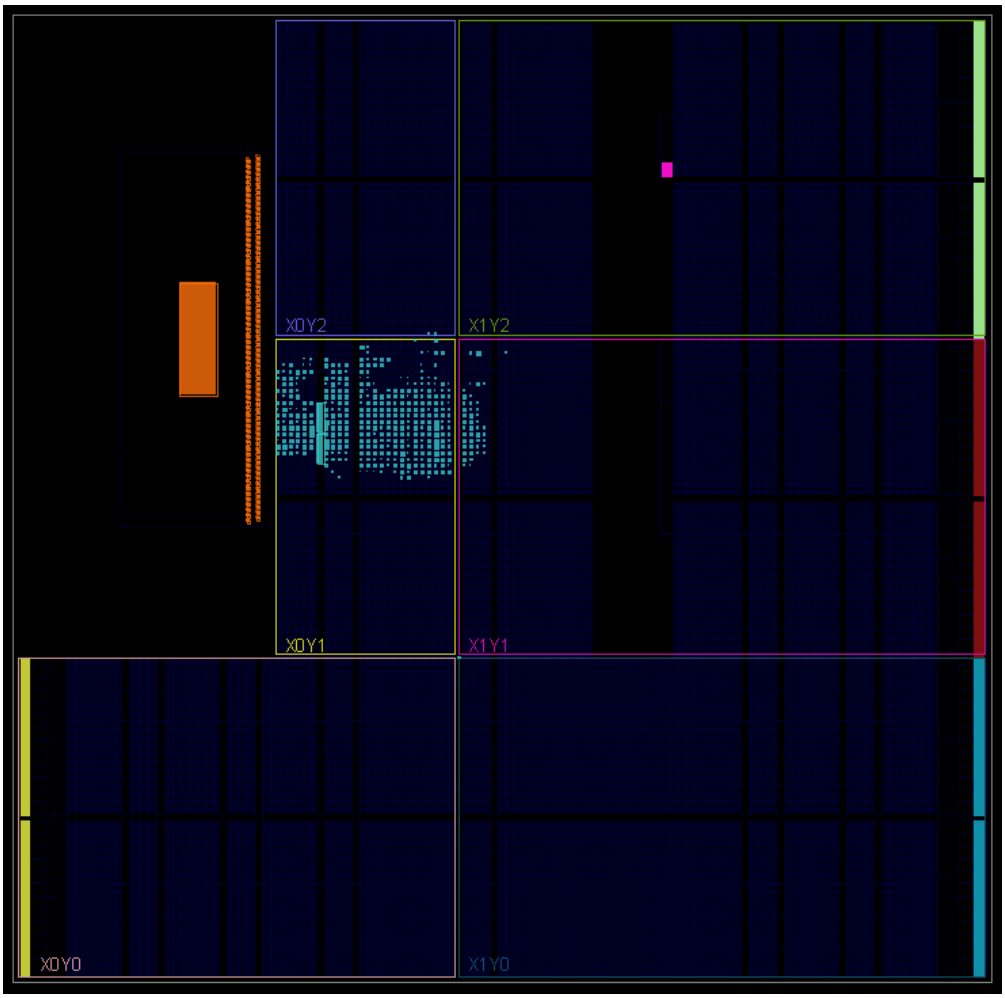


Figura 1.13: Representación del resultado de sintetizar e implementar el diagrama visto en la figura 1.12. Se muestran las celdas que quedaron configuradas (CLB, DSP, etc), las más largas correspondiendo a BRAM. Otras celdas pueden estar siendo parcial o totalmente utilizadas.

1.5.4. Otras herramientas

Partimos del trabajo realizado por Ichiro Kawazome, que se encuentra disponible en su repositorio de github[9]. El repositorio contiene una serie de ejemplos de interacción entre el ARM y la FPGA para diferentes tipos de hardware. En particular rescatamos los siguientes elementos que nos resultaron útiles para la realización de este trabajo:

- Una imagen completa del file system con un kernel de Linux compilado para el hardware utilizado.

[9] `https://github.com/ikwzm/FPGA-SoC-Linux`

- Drivers y servicios que permiten reprogramar la FPGA desde *userspace* o modificar el *device tree* del *kernel* sin tener que reiniciar.
- Un proyecto en Vivado implementando un DMA (*Direct Memory Access*) en un *IP Core*, incluyendo drivers de Linux que lo utilizan.

El contenido del repositorio resultó ser un buen punto de partida durante el desarrollo de este trabajo. Los servicios y diseños existentes para la FPGA fueron estudiados para comprender cómo funciona la comunicación entre un programa desde *userspace* en el ARM y un *IP Core* en la capa PL.

Con esta base, se procedió a desarrollar las herramientas de software y hardware necesarias para el trabajo.

Capítulo 2

Infraestructura desarrollada

Al implementar el hardware, tenemos un increíble grado de control sobre las capacidades y limitaciones de nuestro sistema final. Esto incluye múltiple escenarios que no estaríamos considerando de tener que utilizar un hardware fijo.

Esta situación es similar a escribir código en un lenguaje de alto nivel donde, por ejemplo, nos vemos obligados a interactuar con un *garbage collector*; en comparación con desarrollar en un lenguaje de bajo nivel donde el manejo de memoria es explícito.

Siendo que estamos tratando la capa más baja en un sistema de cómputo, queda en nosotros, tomando el papel de diseñadores, implementar la arquitectura del sistema.

En este capítulo presentamos la arquitectura propuesta. Comenzaremos con una visión general en la sección 2.1 para luego separar la explicación desde el punto de vista del hardware (sección 2.2) y del software (sección 2.3).

2.1. Arquitectura

Tomaremos las siguientes máximas de diseño:

- **Utilizar SMP** (*Symmetric Multi-Processor*). Nos interesa utilizar un sistema *multi-core* bajo un único tipo de procesador. En este caso, será el MicroBlaze de Xilinx.
- **Mantener flags de concurrencia en memoria externa**. Si bien hay módulos de hardware que implementan mutex y semáforos sin depender en memoria externa, no los utilizaremos en este trabajo. Nos interesa entender los cuellos de botella que existen con este método y si podemos mejorar el desempeño para programas con distintos grados de sincronización.
- **Utilizar BRAM aislada e individual**. Cada CPU (o *core*) tendrá un módulo de BRAM disponible para su uso individual e inaccesible para el resto. El programa en ejecución será quien defina cómo se utiliza.
- **Manejar datos de entrada/salida en memoria externa**. El programa y sus datos de entrada estarán disponibles en memoria externa al momento de iniciar la ejecución. La salida deberá escribirse a memoria externa.

Llamaremos `Sistema` a un conjunto de *softcores* simétricos (SMP) donde cada uno tendrá acceso a un módulo de BRAM de manera independiente, y donde todos los *softcores* estarán conectados a una *cache* L2.

Llamaremos `Framework` al resto de los componentes en la FPGA que sirven de puente entre el `Sistema` y el `ARM`.

La APU (Application Processor Unit) comprende al par de procesadores ARM y provee una interfaz para la parte PL. Esta interfaz puede usarse para comunicar la parte PL con el ARM. Además permite acceder a memoria principal de manera coherente.

La figura 2.1 muestra estos componentes, junto con la memoria principal (DDR, externa a la FPGA), y da una visión general de su interconectividad.

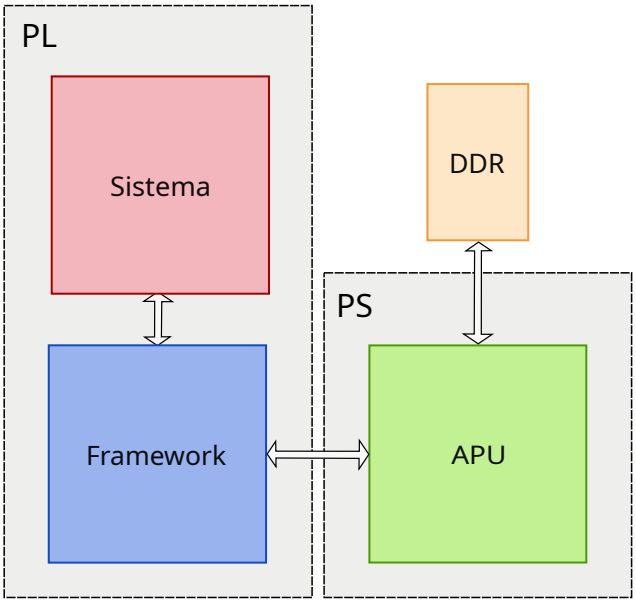


Figura 2.1: Diagrama de alto nivel de la arquitectura a implementar. Las flechas indican la existencia de una o más señales (o buses) entre los componentes. Se indica si estos se encuentran en la capa PL, en la capa PS, o son externos.

El Sistema estará limitado en su interacción a señales de entrada mínimos (CLK, RST), así como a un único puerto de salida que utiliza una interfaz AXI para acceder a memoria principal. Esto nos permite abstraernos del tipo de *softcore* y nivel de *cache* utilizado.

Hacer uso de las señales de CLK y RST nos permitirá iniciar, pausar y resetear el Sistema a elección. Tener control sobre ellas nos permite, a su vez, medir el tiempo de ejecución con una precisión de ciclos de *clock*.

Si bien el *softcore* utilizado provee una interfaz de *debug*, no haremos uso de la misma porque no resulta lo suficientemente genérica y otros *softcores* podrían implementarla de forma diferente. Por otro lado, tampoco es necesario acceder a interrupciones ya que vamos a estar midiendo desempeño y no requerimos modificar el flujo de ejecución del programa del *softcore*.

Desde el punto de vista del ARM, utilizaremos el sistema operativo dentro del ARM para carga, puesta en marcha y monitoreo del programa a correr, así como la extracción de resultados y su chequeo. El programa a correr será cargado en memoria principal (DDR) con el Sistema desactivado.

El Framework estará monitoreando accesos a memoria principal por parte del Sistema. Esto permite que los programas puedan notificar su progreso y finalización mediante un mecanismo genérico. Tras activar el Sistema, el programa en ejecución hará accesos a posiciones de memoria distinguidas que permitirán notificar al ARM la transición entre distintas etapas de su ejecución.

Para esto desarrollamos e implementamos dos *IP Cores*: un Softcore AXI Controller que nos permite monitorear los accesos a memoria principal, y un Softcore Controller con el cual el ARM interactúa para configurar el Framework, así como para activar, pausar y resetear el Sistema según sea necesario.

A continuación, y con el fin de ordenar los puntos a tratar en el presente capítulo, se plantea una división en términos de hardware y software.

La parte hardware trata sobre la implementación en la FPGA, es decir, el Sistema junto con los *IP cores* mencionados anteriormente. Además, detalla la forma en que se administrará la memoria principal entre el ARM y el Sistema.

La parte de software trata sobre las herramientas que nos permitirán la implementación, carga en memoria, ejecución y análisis de resultados de los experimentos.

2.2. Arquitectura (Hardware)

En la figura 2.2 podemos observar el detalle de tres de los componentes vistos en la figura 2.1.

El APU (Application Processor Unit) contiene el ARM y provee una interfaz para la capa PL de la cual vamos a utilizar tres puertos:

- El puerto ACP (Acceleration Coherency Port) provee un acceso AXI a la memoria principal que garantiza coherencia con el ARM.
- El puerto INT permite generar interrupciones hacia el ARM. Estas interrupciones informan al ARM de eventos como la transición entre etapas descrita en la introducción de este Capítulo.
- El puerto GP0 (General Purpose 0) sirve de canal de comunicación desde el ARM. Esta comunicación usa el protocolo AXI y le permite leer y escribir registros dentro del Softcore Controller.

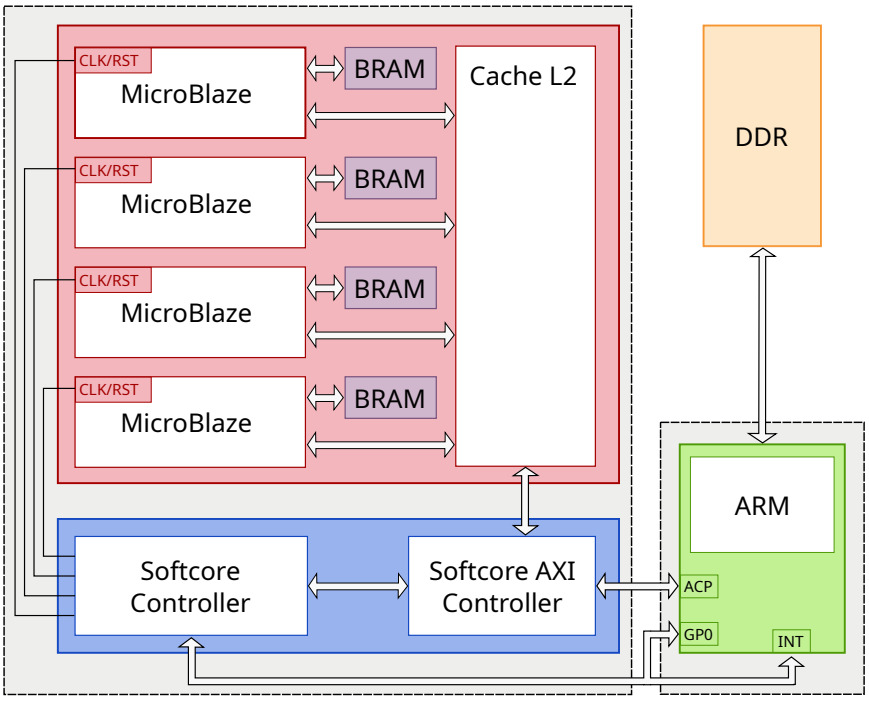


Figura 2.2: Diagrama visto en la figura 2.1 con el detalle de los componentes utilizados. CLK/RST, ACP, GP0 e INT son puertos de sus componentes respectivos. Las flechas indican la existencia de una vía de comunicación entre los componentes. Las líneas simples indican una comunicación en forma de señales.

El Framework está compuesto por los ya mencionados *IP Cores* Softcore Controller y Softcore AXI Controller. En cambio, el denominado Sistema está compuesto por *IP Cores* provistos por Xilinx: cuatro MicroBlaze, cada uno con su módulo de BRAM, y una *cache* L2.

Las señales de CLK (*Clock*) y RST (*Reset*) de los MicroBlaze están controladas por el Softcore Controller. La L2 se comunica con la DDR mediante el Softcore AXI Controller, quien redirecciona todas las transacciones AXI al puerto ACP, agregando un offset a la dirección de memoria (el cual explicaremos más adelante).

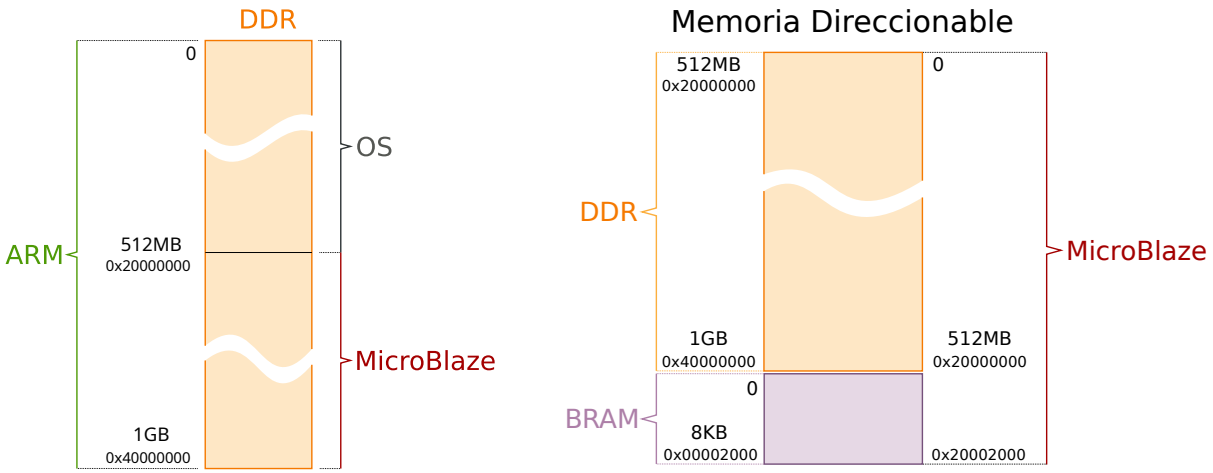
El Softcore Controller y el Softcore AXI Controller se comunican entre sí para configurar qué direcciones de memoria y/o tipos de acceso generarán eventos. Estos eventos nos permitirán identificar las transiciones de etapa del programa, las cuales iremos detallando durante el resto del Capítulo.

Un punto a destacar es que si bien utilizamos el puerto ACP, este sólo garantiza coherencia de lectura/escritura con el ARM[1]. Es decir, si se hace un acceso de lectura, el resultado será el mismo desde el punto de vista del ARM. Si se hace una escritura, el dato en las *cache* del ARM (de estar ahí) será invalidado y tendrá que buscarlo en la DDR.

Es importante mencionar que esto **no implica** que la parte PL utilizando este puerto tiene coherencia con el ARM. Por ejemplo, una cache conectada al puerto ACP debe hacer un *flush* para que el ARM y la DDR estén al tanto de los cambios que hace. De la misma forma, escrituras del ARM a DDR no son notificadas por el puerto ACP (el protocolo AXI no maneja coherencia).

Ambos puntos fueron tomados en cuenta y el *workaround* elegido está explicado más adelante.

2.2.1. Uso de Memoria



(a) Partición de memoria principal (DDR) entre el sistema operativo corriendo sobre el ARM y los MicroBlaze.

(b) Asignación de memoria para los MicroBlaze. Para mayor detalle, ver Figura 2.14.

Figura 2.3: Uso de memoria.

[1]El APU provee otros puertos de acceso directo a memoria que no tienen coherencia con el ARM pero que requieren implementar sincronización y coherencia por software.

DDR

La memoria DDR total disponible en la placa es de 1 GB. Si bien el ARM tiene acceso total, para evitar conflictos entre los *softcores* y el sistema operativo se reconfiguró el kernel de manera que utilice únicamente los primeros 512 MB. La fracción de memoria sobrante será accedida solo para cargar programas con sus *inputs* y para leer la salida de la ejecución. Los *softcores* solo tienen acceso a la parte superior de la memoria, es decir a los 512 MB más altos, indicados en la figura 2.3a.

El Softcore AXI Controller se ocupa, entre otras tareas, de agregar un *offset* de 0x20000000 a la memoria que direccionan los *softcores* (los 512 MB altos del 1 GB de la memoria). Esta diferencia puede observarse en la figura 2.3b.

Gracias a este *offset*, el acceso a la parte alta de la memoria resultará transparente para los MicroBlaze, los cuales estarán configurados para iniciar la ejecución desde la dirección 0x00000000. Este offset, sumado a que el protocolo AXI permite limitar las direcciones accesibles por puerto, permite restringir a los *softcores* de forma que no puedan acceder a los primeros 512 MB de memoria DDR.

BRAM

Como vimos en la figura 2.2 y hemos mencionamos previamente, cada MicroBlaze tiene una BRAM propia de 8 kB (sugerida por Vivado para el *softcore*), aunque podría configurarse una memoria de 4 kB, 16 kB, 32 kB, 64 kB y 128 kB. En la figura 2.3b se ilustra cómo se encuentra mapeada esta memoria a partir de la dirección 0x20000000 (inmediatamente después de los 512 MB del espacio de direccionamiento de los MicroBlaze).

A pesar de operar en el mismo espacio de direccionamiento, las BRAM no estarán conectadas al MicroBlaze por medio de un bus AXI. Utilizan, en cambio, un bus rápido y especializado llamado LMB (*Local Memory Bus*), el cual posee un protocolo más simple y tiene latencia mínima (1 ciclo de *clock*).

2.2.2. MicroBlaze

Buscamos una configuración que permita experimentar sobre un *softcore* que posea buen desempeño, capaz de ejecutar una amplia variedad de programas mono-tarea pero en múltiples cores.

Para ello, optamos por un MicroBlaze de 32 bit, big-endian, con una *cache* L1 de 8 kB para instrucciones y otra L1 de 8 kB para datos, ambas con líneas de cuatro palabras.

En la configuración del procesador se priorizó el rendimiento, utilizando una *cache* adicional para la predicción de saltos (*branches*). Por otro lado, se utilizó un ISA con todas las extensiones por hardware permitidas, contando con una FPU de 32 bit que permite realizar operaciones de punto flotante de precisión simple por hardware. Para el caso de las operaciones con doble precisión, éstas deberán ser simuladas por software.

Los programas serán ejecutados como monotarea, con el nivel de privilegio máximo desde la primera instrucción a ejecutar (posición 0 de memoria) y sin intervención de un kernel o sistema operativo. Tampoco nos interesa proteger ni limitar el acceso a memoria. Por ello no dispondremos de una MMU ni de ningún tipo de protección de memoria.

En la figura 2.4 se ilustra una de las pantallas de configuración que provee Vivado para el MicroBlaze. En [19] se exponen estas y otras configuraciones con más detalle.

Se seleccionó esta pantalla en particular porque provee las configuraciones generales para el procesador.

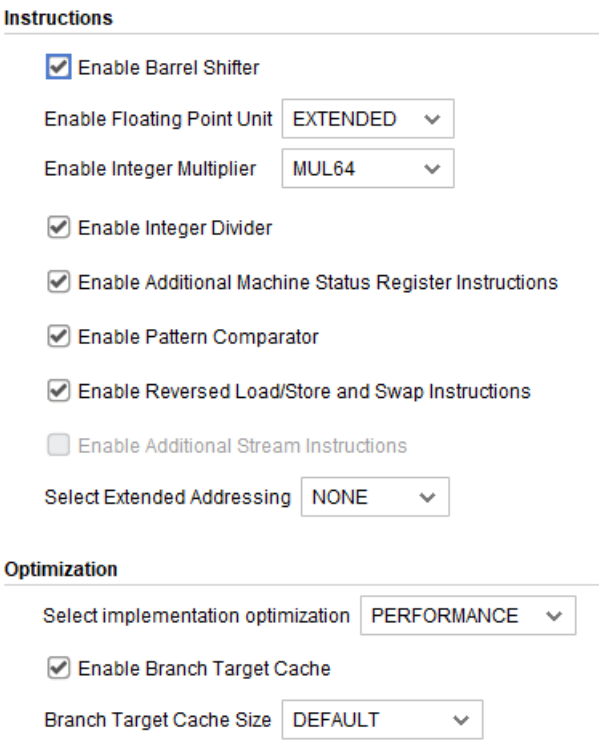


Figura 2.4: Configuración elegida de ISA y optimizaciones de MicroBlaze.

A fin de exponer con mayor detalle las opciones de la figura 2.4, se describe a continuación cada una de ellas:

- **Barrel Shifter**: Habilita instrucciones de bit shifting. Puede mejorar la *performance* pero aumenta el tamaño del procesador.
- **Enable Floating Point Unit**: Habilita instrucciones de punto flotante. Aumenta drásticamente el tamaño del procesador.
  - **NONE**: Sin instrucciones de punto flotante por hardware.
  - **BASIC**: Instrucciones para operaciones básicas de punto flotante (suma, resta, multiplicación, división, comparación).
  - **EXTENDED**: Incluye las operaciones básicas, agrega la instrucción para calcular raíz cuadrada e instrucciones de conversión entre enteros y punto flotante.
- **Enable Integer Multiplier**: Habilita instrucciones de multiplicación. Aumenta el uso de bloques DSP.
  - **NONE**: Sin instrucciones de multiplicación por hardware.
  - **MUL32**: Instrucciones de multiplicación de 32 bit.
  - **MUL64**: Instrucciones de multiplicación de 32 bit y 64 bit.
- **Integer Divider**: Habilita instrucciones de división entera.
- **Enable Additional Machine Status Register Instructions**: Habilita instrucciones que permiten setear el valor de bits específicos en los registros de máquina (MSR).
- **Pattern Comparator**: Habilita instrucciones de comparación. Por ejemplo, PCMPBF devuelve la posición del primer byte que comparten dos palabras. Otro ejemplo es la instrucción CLZ que calcula la cantidad de ceros a la izquierda.

- **Reversed Load/Store and Swap Instructions**: Habilita instrucciones load/store/swap utilizadas en datos con diferente *endianness* (en nuestro caso serían para datos *little-endian*).
- **Extended Addressing**: Habilita extensiones de las instrucciones `load/store` para que puedan exceder el direccionamiento por defecto (en nuestro caso, siendo un procesador de 32 bit, serían 4 GB).

- **Implementation Optimization**: Factores a optimizar.
  - **PERFORMANCE**: Optimizar *performance* computacional. Utiliza un five-stage pipeline. Es el setting recomendado por Vivado, sugiriendo que las otras alternativas deberían considerarse sólo para casos muy específicos.
  - **AREA**: Reducir el área utilizada en la FPGA. Utiliza un three-stage pipeline con menor throughput de instrucciones. No está disponible en sistemas con MMU, Branch *cache* o ACE, entre otros.
  - **FREQUENCY**: Aumenta la frecuencia final del MicroBlaze. Utiliza un eight-stage pipeline. Se recomienda para alcanzar una frecuencia objetivo, particularmente cuando se utiliza memoria externa basada en *cache*, MMU o mucha memoria LMB (BRAM).
- **Branch Target Cache**: Habilita la predicción y el *caching* de branches de ejecución.
- **Branch Target Cache Size**: Cantidad de entradas. 1024 por defecto. 8 como mínimo. 2048 como máximo.

- **Fault Tolerance**: Se agrega paridad y soporte de ECC (Error Correcting Codes) a la BRAM en el LMB. Protege ante un bit flip en la BRAM.

Junto con el resto de los componentes, logramos insertar cuatro `MicroBlaze` con estas configuraciones manteniendo *delays* y *timings* en un rango seguro, según el reporte obtenido en Vivado. Buscamos equipar a los `MicroBlaze` con el soporte necesario para resolver cualquier programa general o numérico de asegurar su desempeño.

Para poder identificar cada *softcore* en tiempo de ejecución, se les configuró un identificador de 0 a 3 que puede leerse por código via el registro RPVR1. Esto nos permite diferenciarlos fácilmente y, de ser necesario, asignarles tareas de manera directa. En ocasiones nos referiremos a los cuatro *softcores* como CPU0, CPU1, CPU2 y CPU3, siendo el "Main CPU" CPU0 (generalmente).

2.2.3. Cache L2

La *cache* L2 es compartida por todos los *softcores*. Fue configurada como una memoria de 32 kB con dos líneas de 64 B. La comunicación desde los *softcores* hacia esta *cache* se hace usando el protocolo ACE que garantiza coherencia entre todos los dispositivos que la usan. Se conecta a memoria principal mediante el `Softcore AXI Controller` utilizando el protocolo AXI.

Esta *cache* L2 es un *IP Core* provisto por Vivado y está optimizado para MicroBlaze.

2.2.4. Softcore AXI Controller

El `Softcore AXI Controller` actúa principalmente como un *sniffer*. Se encarga de observar los accesos que realiza el `Sistema` a la DDR (en este caso, provenientes de la L2), así como de proveer información sobre estos al `Softcore Controller`.

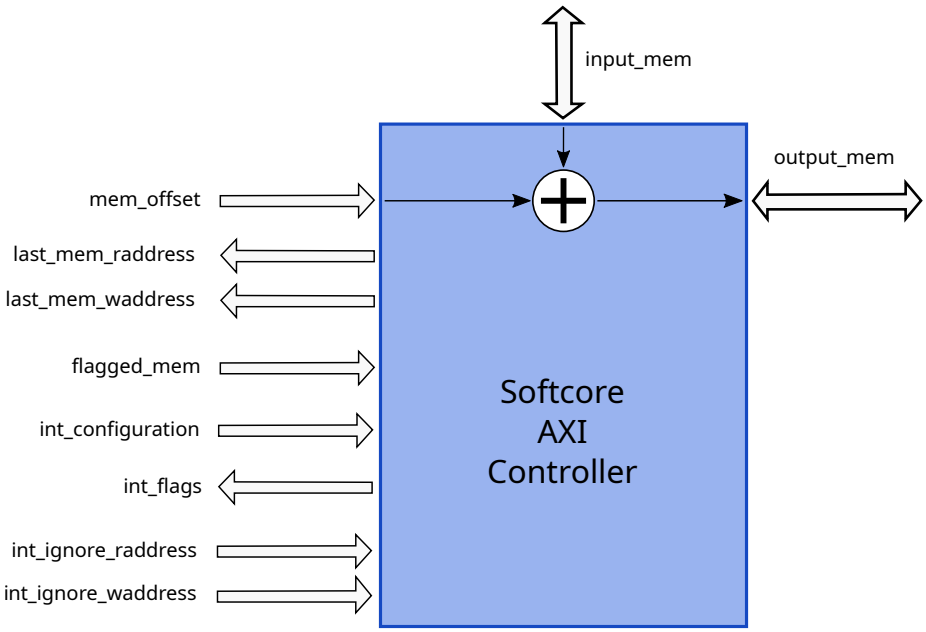


Figura 2.5: Interfaz del componente `Softcore AXI Controller`.

Permite redireccionar los accesos a memoria sumándoles un *offset* (puerto `mem_offset`). En nuestro caso, este offset fue implementado como una constante no configurable por el `Softcore Controller` y equivale a 512 MB.

Además del *offset*, el *IP Core* guarda la última posición de memoria que fue leída/escrita en registros internos (expuestos a través de los puertos `last_mem_raddress` y `last_mem_waddress`). Ambos incluyen un bit que indica si la dirección es válida o no. Los registros son escritos de manera sincrónica con la modificación de `input_mem`.

En caso de un `reset`, ninguna posición será válida hasta la primera lectura/escritura.

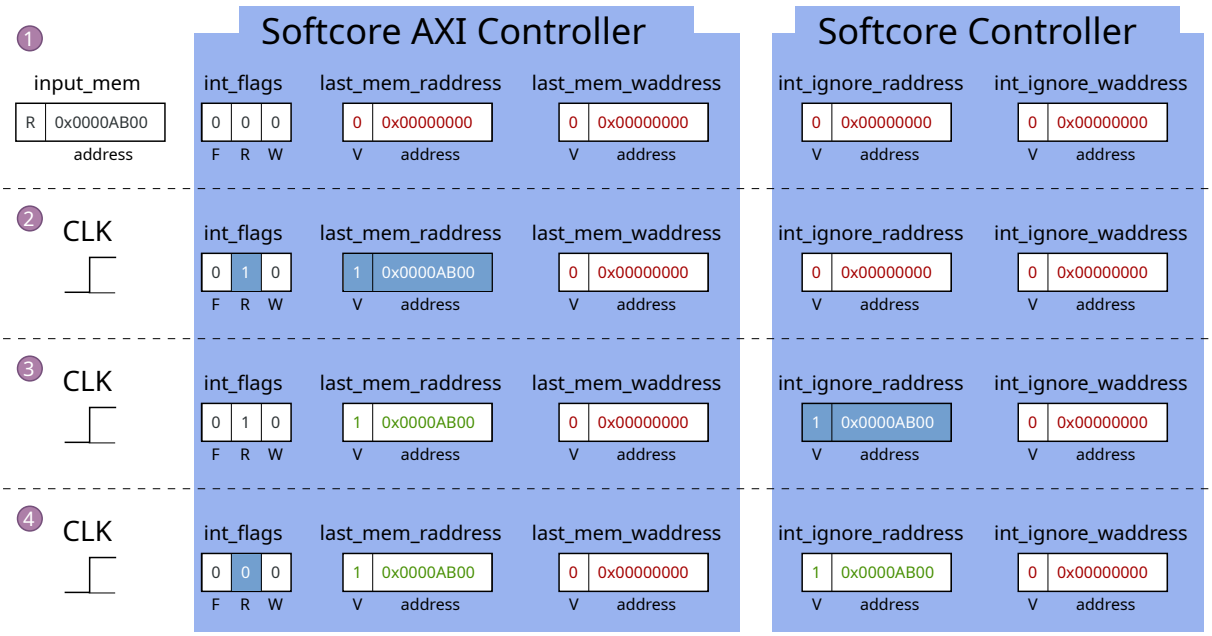
Este *softcore* ofrece una forma de monitorear todos los accesos de memoria, ya sean lecturas o escrituras. En base a estos accesos, se modifican los *flags* R y W expuestos en `int_flags`, que señalizan si se trata de una escritura o lectura respectivamente. Se pondrán en `1` cada una de estas señales, cuando se detecte que la señal de dirección de memoria sea válida y distinta a la anterior.

El `Softcore Controller` estará atento a los cambios de las señales `last_mem_raddress` y `last_mem_waddress`, y seteará las señales `int_ignore_raddress` e `int_ignore_waddress` copiando los valores de los arriba mencionados.

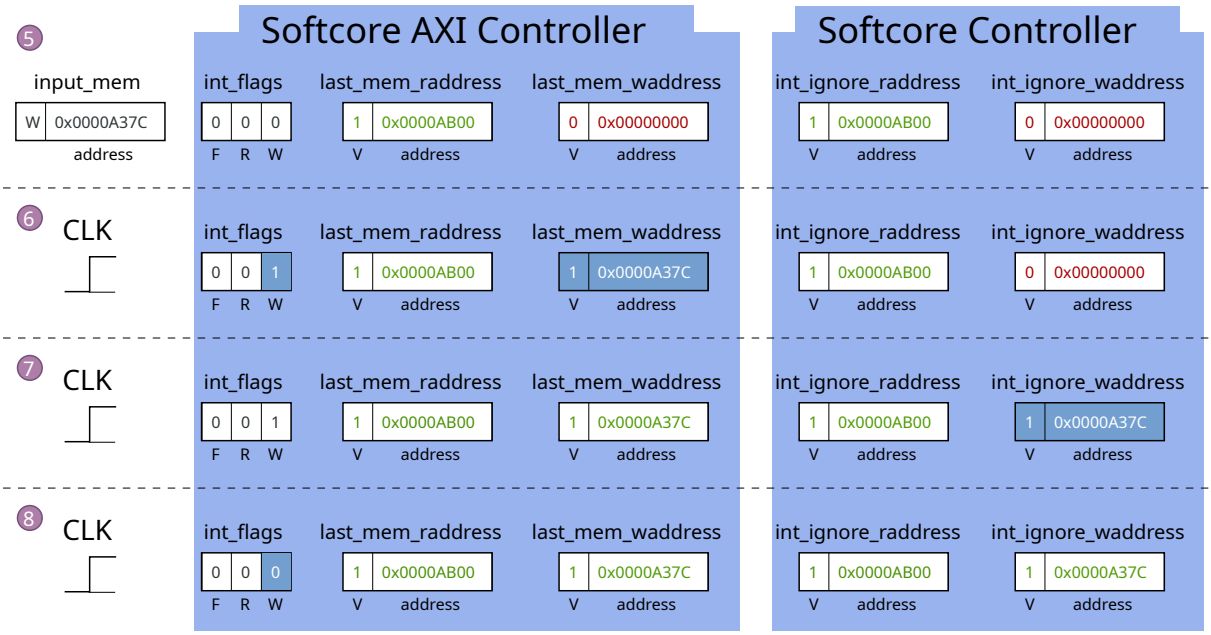
Luego, el `Softcore AXI Controller`, al identificar las modificaciones del registro de control `int_ignore_Xaddress` (donde `X` es `r` o `w`), seteará el *flag* indicado (R o W) en `0`. Todo este proceso será explicado en detalle en la sección 2.2.5.

En términos generales, el *flag* será `1` cuando el último acceso a memoria sea válido y diferente de `int_ignore_Xaddress`, y será `0` en otro caso.

En la figura 2.6 podemos ver un ejemplo de una primera lectura a memoria, seguida por una primera escritura:



(a) Se detecta un pedido de lectura de memoria desde el `Sistema`.



(b) Se detecta un pedido de escritura de memoria desde el `Sistema`.

Figura 2.6: Un posible escenario post inicialización. Se indican los cambios que ocurren en las señales ante cada evento (lectura, escritura, `CLK`).

1. Post inicialización, todos los valores están en cero. Esto incluye los bits de validez de las direcciones. En algún momento, el `Sistema` hace un pedido de lectura a la posición 0x0000AB00, seteando la dirección en `input_mem`.

2. Al recibir un pulso de *clock*, el `Softcore AXI Controller` se detecta el pedido en `input_mem` y procede a copiar la dirección a `last_mem_raddress`, seteando su bit de validez en `1`. Además, setea en `1` el flag R de `int_flags`.

3. Durante el siguiente ciclo de *clock*, el `Softcore Controller` detecta el cambio en `last_mem_raddress` y setea `int_ignore_raddress`, copiando el valor del primero.

4. Un ciclo de *clock* más tarde, el Softcore AXI Controller verifica que `int_ignore_raddress` es la dirección actual de lectura según `input_mem` y procede a setear el flag R en `0`.
5. En algún momento, el Sistema hace un pedido de escritura a la posición 0x0000A37C, seteando la dirección en `input_mem`.
6. Al recibir un pulso de *clock*, el Softcore AXI Controller se detecta el pedido en `input_mem` y procede a copiar la dirección a `last_mem_waddress`, seteando su bit de validez en `1`. Además, setea en `1` el *flag* W de `int_flags`.
7. Durante el siguiente ciclo de *clock*, el Softcore Controller detecta el cambio en `last_mem_waddress` y setea `int_ignore_waddress`, copiando el valor del primero.
8. Un ciclo de *clock* más tarde, el Softcore AXI Controller verifica que `int_ignore_waddress` es la dirección actual de escritura según `input_mem` y procede a setear el flag W en `0`.

El `CLK` del Softcore Controller es el mismo que el CLK del puerto `input_mem`, por lo cual estas señales se modifican de manera sincrónica.

El Sistema implementado en este trabajo posee una *cache* L2. Esto significa que los valores que llegan por `input_mem` serán únicamente los *cache misses* y los *flushes* de la L2. En un Sistema sin *cache* sería posible utilizar esta información para *debuggear* los accesos a memoria hechos por la ejecución del programa.

Resta mencionar el tercer y último *flag* en `int_flags`: el *flag* F se pondrá en `1` ante un acceso de lectura o escritura a la posición de memoria indicada en `flagged_mem`. El *flag* volverá a `0` cuando la dirección de memoria indicada por `input_mem` (lectura y/o escritura) sea distinta de `flagged_mem` (o se invalide).

Este *IP Core* puede usarse como una herramienta de *debug* y permite seleccionar qué *flags* nos interesa utilizar dependiendo del experimento que se busque correr. Para ello utilizamos `int_configuration` (Tabla 2.1).

`int_configuration`	**Flag R**	**Flag W**
0 ($\overline{\text{RW}}$)	Ignora accesos R (es siempre 0)	Ignora accesos W (es siempre 0)
1 (R$\overline{\text{W}}$)	Cambia según accesos R	Ignora accesos W (es siempre 0)
2 ($\overline{\text{R}}$W)	Ignora accesos R (es siempre 0)	Cambia según accesos W
3 (RW)	Cambia según accesos R	Cambia según accesos W

Tabla 2.1: Posibles valores de `int_configuration` y sus significados.

Por otro lado, el evento de acceso a memoria específica (*flag* F) será habilitado apenas se configure `flagged_mem`. Dicha señal es una dirección de 32 bit sin bit de validez, por lo que deberá ser distinta de cero.

En las condiciones de uso de los experimentos se considera que el Softcore Controller eventualmente modificará `flagged_mem` si es necesario continuar. Por ello no existe una variante `int_ignore_flagged_mem_address`.

Los *flags* R y W fueron muy útiles en las primeras etapas de este trabajo y ayudaron a detectar problemas de hardware y software. Al permitirnos pausar ante cada lectura y/o escritura, nos fue posible *debuggear* el MicroBlaze y analizar problemas relacionados con carga del programa e input, así como su ejecución.

Como no hace uso de una interfaz específica, es posible que otros *softcore processors* puedan también hacer uso de estos *flags*. Se debe tener en cuenta que, de existir, la *cache* de estos *softcores* puede estar capturando accesos de lectura/escritura y/o delegándolos a memoria principal a destiempo.

Dicho esto, recordamos que nos interesa medir el rendimiento de los experimentos. A fines prácticos, no utilizaremos los *flags* R y W pues resultarían en una interrupción constante del programa. Por lo tanto, el valor de configuración `int_configuration` será `0` para todos los experimentos.

Sí queremos detectar cuándo el programa finaliza y cuándo hace *flush* de su *cache*. Para ello, configuraremos *flagged_mem* usando direcciones específicas que siempre generarán un *cache miss*, a las que el programa hará acceso según el estado de su ejecución.

2.2.5. Softcore Controller

Los objetivos del Softcore Controller son controlar al Sistema, proveer tiempos de ejecución de manera precisa, e informar al ARM ante eventos de interés (como el acceso a una posición de memoria específica).

Respecto al "control", nos interesa tener las herramientas para:

i) Iniciar la ejecución en determinado momento (desde el ARM),

ii) Pausarlo ante algún evento (las interrupciones ya descritas),

iii) Reanudar la ejecución desde la pausa (desde el ARM), y

iv) Poder resetear al Sistema (desde el ARM).

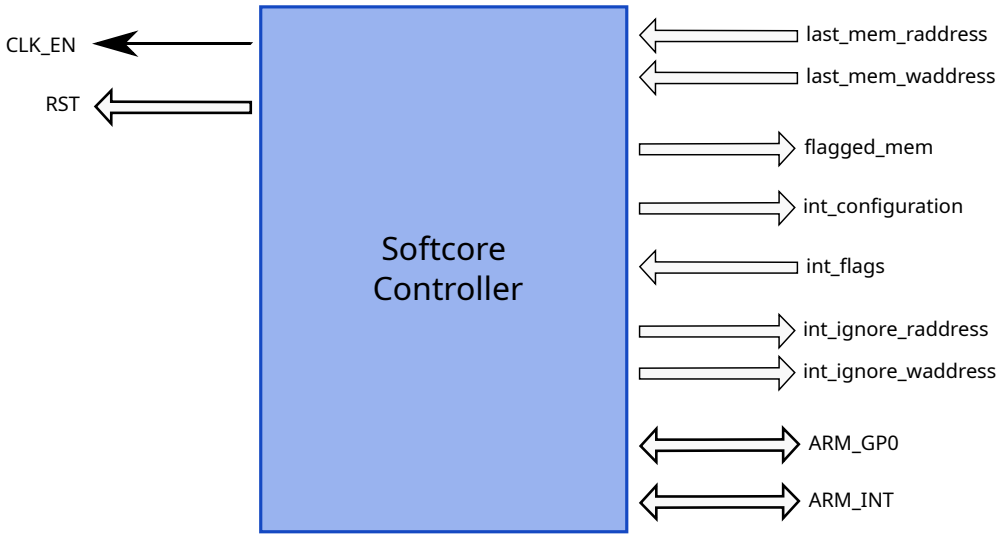


Figura 2.7: Interfaz del componente Softcore Controller.

El Softcore Controller (Figura 2.7) es entonces, el único punto de control que tiene el ARM sobre el Sistema. Permite pausar el *clock* del Sistema de manera general (*CLK_EN*) y resetear *softcores* individualmente (*RST*). Esto último hace posible mantener la señal de RST activa para un subconjunto de *softcores* y, de esta manera, realizar una ejecución utilizando un número reducido de CPUs.

El protocolo ACE requiere que todos los dispositivos conectados sean capaces de atender las solicitudes del bus. Si bien la especificación contempla un modo de uso *low-power* donde es posible ignorar un puerto, requiere que el dispositivo haya iniciado una transacción para entrar en dicho modo.

A fines prácticos, esta transacción no es posible de hacer por hardware apenas se inicia el Sistema pues depende del MicroBlaze iniciar la acción para desactivarse. De activar un RST individual, el protocolo ACE quedaría esperando la respuesta del *softcore* desactivado, lo cual dejaría al Sistema colgado a nivel hardware. Por este motivo, no es posible utilizar los RST de manera individual para el Sistema implementado, quedando limitados a un uso total y simultáneo.

En conclusión, si bien el *IP Core* permite una activación selectiva, desde el ARM estaremos habilitando todos los CPUs.

Esta restricción de hardware no impide que se puedan realizar ejecuciones utilizando un subconjunto de los *softcores*. La solución a esta problemática fue resuelta por software. Si bien todos los *softcores* serán activados al inicio, aquellos que querramos "desactivar" estarán ejecutando código específico para ello. En la Sección 2.3.1 explicaremos esto en mayor detalle.

Como podemos ver en la Figura 2.7, la mayoría de las señales del Softcore AXI Controller (Figura 2.5) comparten nombre con las del Softcore Controller. Estas señales son conexiones entre ambos *IP Core*. Entre ellas, la única directamente configurable por el ARM es *flagged_mem*. Los demás dependerán del Softcore Controller o Softcore AXI Controller.

Para comunicarse con este IP, el ARM puede acceder a la posición de memoria 0x43C00000, habiendo un registro cada 4 bytes. Estos registros están mapeados a memoria y pueden verse en la Tabla 2.2.

Índice	Registro	R/W	Descripción
0	Status	R+W	Resetea o activa los *softcores*.
1	CLK_W	R	Ciclos de CLK transcurridos en ejecución.
2	CLK_I	R	Ciclos de CLK transcurridos en pausa.
3	CLK_T	R	Ciclos de CLK transcurridos desde el inicio.
4	flagged_mem	R+W	La dirección de memoria cuyo intento de acceso pausa la ejecución.
5	CPU_EN + Info	R+W	Ver Figura 2.8
6	last_mem_raddress	R	Dirección de último acceso de lectura.
7	last_mem_waddress	R	Dirección de último acceso de escritura.

Tabla 2.2: Registros de 32 bits del Softcore Controller accesibles por el ARM. Todos son legibles por el ARM (R) y se indica si pueden ser modificados (W), ya sea parcial o totalmente.

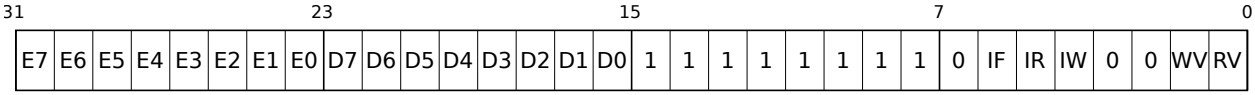


Figura 2.8: Registro CPU_EN + Info. E_i es configurable y define los CPUs que pueden correr. D_i es sólo legible y será 1 si el CPU no está corriendo (depende de *RST*). IF, IR e IW son los *flags* de interrupción en int_flags. WV y RV indican si los valores de last_mem_Xaddress son válidos.

El bus AXI utiliza el mismo clock que el Softcore Controller, el cual no se detiene con el Sistema. Esto habilita la posibilidad de que un acceso a memoria adicional sobreescriba int_flags y perdamos la información sobre cuál interrupción se generó. Por ello, en caso de detectar cualquier tipo de interrupción, los *flags* detectados en ese momento serán guardados en el registro CPU_EN + Info (*flags* IF, IR, IW) y el Sistema será pausado (CLK_EN = 0).

El ARM deberá leer este registro para obtener información del tipo de interrupción en curso.

A través del registro Status, el ARM podrá resetear, ejecutar y reanudar la ejecución de los *softcores*. Además, Status será modificado por el *IP Core* cuando se active uno de los *flags* de int_flags. Podemos ver la máquina de estados correspondiente a este comportamiento en la figura 2.9.

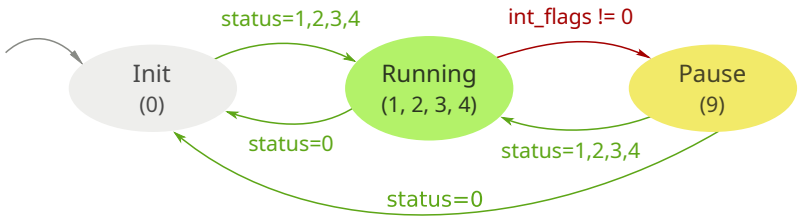


Figura 2.9: Máquina de estados representativa del registro Status del Softcore Controller. En verde están las transiciones que se generan desde el ARM. En rojo estarán las que ocurren debido al Sistema.

Según el registro Status, varias de las señales del Softcore Controller estarán tomando distintos valores. Estos están resumidos en la tabla 2.3 y los explicaremos a continuación:

Registros y Señales	**Status**					
	0	**1**	**2**	**3**	**4**	**9**
CLK_EN	0	1	1	1	1	0
RST_i	1	E_i (según haya sido configurado CPU_EN)				
int_configuration[2]	*N/A*	$\overline{RW}$	$R\overline{W}$	$\overline{RW}$	RW	*N/A*
CLK_W_t	0	$CLK\_W_{t-1}+1$				CLK_W_{t-1}
CLK_I_t	0	CLK_I_{t-1}				$CLK\_I_{t-1}+1$
ARM_INT	0	0	0	0	0	1

Tabla 2.3: Configuración según el valor del registro Status.

- **Status = 0**: Estado por defecto. Los *softcores* están todos reseteados.
  - CLK_EN = 0.
  - $RST_i = 1$.
  - int_configuration = 0.
  - CLK_W = 0.
  - CLK_I = 0.
  - ARM_INT = 0.

[2]Interrupciones de R/W activadas según Tabla 2.1

- `Status` = **1, 2, 3, 4**: Los *softcores* no desactivados pueden correr.
  - `CLK_EN` = 1.
  - RST_i = `E`$_i$ (según haya sido configurado `CPU_EN`).
  - `int_configuration` = `Status` - 1 (interrupciones de R/W activadas según tabla 2.1).
  - `CLK_W` se incrementa con cada pulso de CLK.
  - `CLK_I` = `CLK_I`.
  - `ARM_INT` = 0.
- `Status` = **9**: Uno de los *flags* de interrupción de `int_flags` se activó.
  - `CLK_EN` = 0.
  - RST_i = `E`$_i$ (según haya sido configurado `CPU_EN`).
  - `int_configuration` = 0.
  - `CLK_W` = `CLK_W`.
  - `CLK_I` se incrementa con cada pulso de CLK.
  - `ARM_INT` = 1.

El `ARM` configurará los registros según sea necesario antes de cambiar `Status` a 1, 2, 3 o 4. Puede hacer *polling* o esperar a la interrupción `ARM_INT` para verificar el valor de `Status`.

Si `Status` es `9`, el `ARM` puede escribir `flagged_mem` y reanudar la ejecución seteando `Status` como 1, 2, 3 o 4. En caso contrario, seteará `Status` en `0`, finalizando la ejecución y reseteándose los *softcores*, la L2 y el `Softcore AXI Controller` mediante la señal `RST`.

Si bien `Status` corresponde a una posición de memoria que puede escribirse desde el `ARM`, solo los valores 0, 1, 2, 3 y 4 son válidos. Escribir otro valor (incluyendo `9`) no modificará `Status`.

Como recordatorio, las señales `int_ignore_Xaddress` se setearán como su contraparte `last_mem_Xaddress` apenas ésta sea válida y de manera sincrónica (es decir, con cada pulso de *clock*).

Finalmente, es fundamental para este trabajo contar con una medida precisa del tiempo de ejecución. Para esto implementaremos dos registros de 32 bit: `CLK_W`, que cuenta ciclos de *clock* de ejecución, y `CLK_I`, para los ciclos de *clock* pasados durante las interrupciones (tabla 2.2). Estos serán reseteados cuando los *softcores* lo sean.

2.2.6. Consideraciones

Al diseñar hardware, entran en juego elementos que no se tienen en cuenta al desarrollar software. Los componentes introducen *delays* que se van acumulando a medida que se encadenan, y sus interacciones a nivel circuito pueden provocar efectos secundarios muy sutiles pero con consecuencias graves.

Un ejemplo de esto es la diferencia entre un circuito combinatorio y uno sincrónico. El primero es tan veloz como el tiempo de conmutación de los transistores involucrados. El segundo se activa y ejecuta cuando se detecta que una señal (CLK) pasa de nivel bajo (`0`) a nivel alto (`1`), o viceversa (*falling edge*, *rising edge*).

Especificando en HDL, es común tener que definir la salida en función de operaciones lógicas con la entrada. Por ejemplo, si queremos habilitar o deshabilitar una señal de CLK a gusto, una solución simple es utilizar una compuerta AND y conectarla con nuestra entrada de CLK.

Si la señal que habilita el CLK pasa a 0 mientras el CLK está en 1, se estará produciendo un *falling edge* prematuro, lo cual puede producir problemas de sincronización en algunos circuitos. Otro problema yace en si el escenario anterior ocurre cuando el CLK recién estaba activándose. En este caso es difícil saber si los componentes sincrónicos habrán detectado un *falling edge*. Estos problemas y otros similares están ilustrados en la figura 2.10.

Una solución es sincronizar la operación AND (por ejemplo utilizando CLK). Esto se puede hacer colocando un *flip-flop* D entre el *input* activador y la compuerta AND. De esta forma, el *input* activador solo sería tomado en cuenta durante un *rising edge* de CLK.

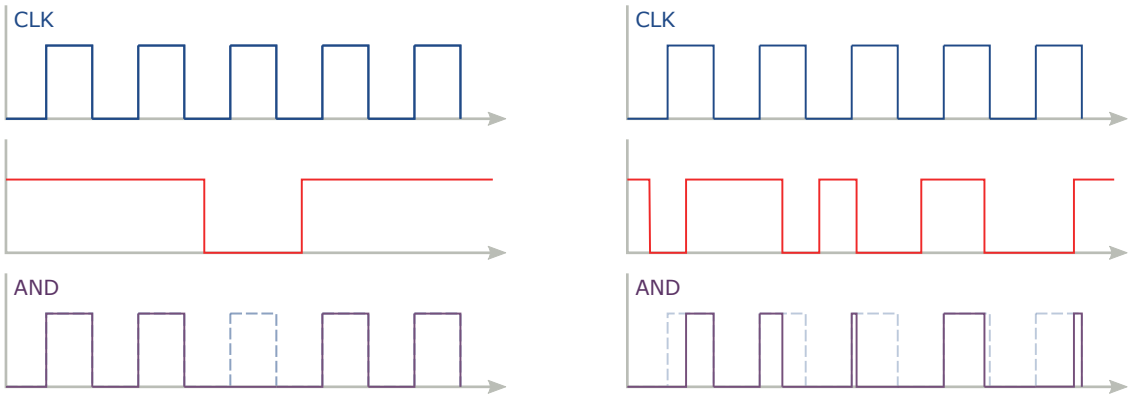


Figura 2.10: Resultado de conectar una compuerta AND entre una señal de CLK y una señal de activación no sincrónica. A la izquierda se muestra un caso ideal. A la derecha se muestran distintos casos que podrían tener efectos no deseados.

Este escenario se conoce como *clock-gating* y no es una buena práctica en FPGAs.

Cuando una señal de *clock* proviene de una operación con compuertas, dicha señal naturalmente sufre un cierto *delay* (el tiempo de conmutación de los transistores, por ejemplo), lo cual puede generar resultados no deseados en casos donde el *timing* es muy importante.

En ASIC esto se puede llegar a corregir pues todo el diseño es modificable pero las FPGAs ya tienen una distribución de sus componentes fija (y, por lo tanto, las distancias entre estos también es fija).

Para garantizar restricciones de *timing* en los componentes que las utilicen, las señales de *clock* en FPGA viajan por una circuitería especialmente ubicada. El *delay* introducido por el *clock-gating* puede atentar contra estas restricciones.

Esto no significa que no se puede generar una señal de *clock* a partir de otra. Por ejemplo, Xilinx provee el componente `BUFGCE` justamente para poder generar un *clock* a partir de otro y de una condición. Este nuevo *clock* viajará por esta circuitería especial de *clocks*.

Otro ejemplo es el concepto de dominios. Se dice que los componentes sincrónicos atados a la misma señal de CLK están en el mismo dominio. De la misma forma, las señales de *reset* (RST) en componentes sincrónicos están atadas a un dominio de CLK.

Un CLK puede estar desfasado de otro, tener una frecuencia distinta, o puede ser activado por una señal externa. Cada uno de estos casos implica un dominio distinto. Los componentes que utilicen más de un dominio de CLK deben tener en cuenta estas diferencias.

En nuestro caso, la L2 requiere tener el mismo dominio que todos sus `MicroBlaze`. Este dominio es distinto del `Softcore Controller` (quien controla la señal que activa y desactiva el `CLK` de los `MicroBlaze`) y del `Softcore AXI Controller` (ambos *IP Cores* están bajo el mismo dominio). La figura 2.11 muestra ambos dominios de CLK y sus señales de RST conjuntas.

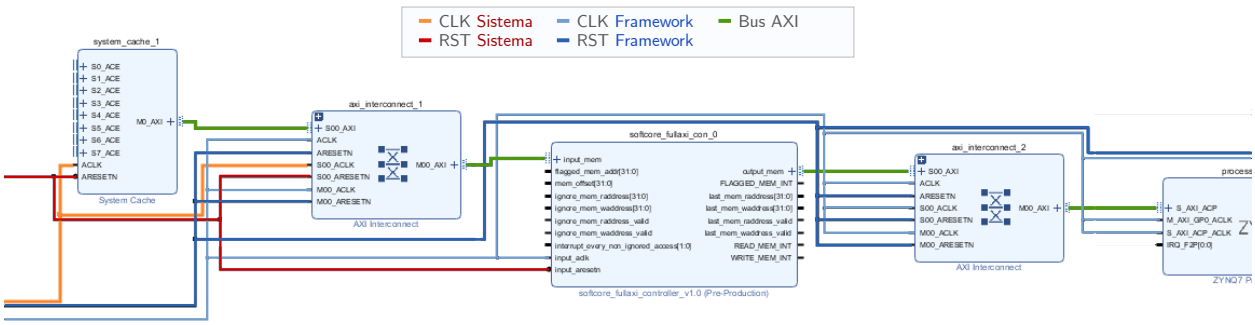


Figura 2.11: Conectividad entre L2 (*system cache*), `Softcore AXI Controller` y puerto ACP de Zynq. A modo ilustrativo se eliminaron varias conexiones que obstaculizarían la visibilidad de los dominios de CLK.
Se muestra el dominio de CLK de la L2 y el RST asignado en naranja y rojo oscuro, respectivamente. De la misma forma, el dominio de CLK y RST de los componentes `Softcore Controller` y `Softcore AXI Controller` está en azul claro y oscuro. En verde denotamos la cadena de conexiones AXI entre los *IP cores*.

Por lo visto arriba, dado que la L2 y el `Softcore AXI Controller` utilizan dominios distintos de *clock*, la interconexión entre ambos requiere una conversión de dominio. El componente *AXI Interconnect* de Xilinx implementa esto de manera transparente al diseñador (figura 2.12).

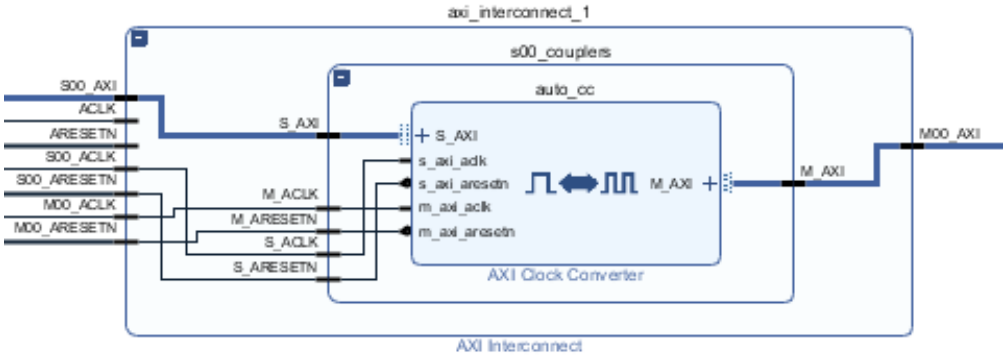


Figura 2.12: Detalle de implementación del *AXI Interconnect* entre L2 y `Softcore AXI Controller`. En este caso, el componente *AXI Interconnect* termina utilizando de fondo un *AXI Clock Converter* para la conversión de dominio.

2.3. Arquitectura (Software)

En la figura 2.13 se detalla el proceso utilizado para realizar los experimentos. Dividimos un experimento en un conjunto de variantes de programa a correr en el `Sistema` (las distintas optimizaciones y soluciones), y un conjunto de instrucciones generales para preparar, poner a ejecutar los programas, y registrar su *output* (el archivo de *template*).

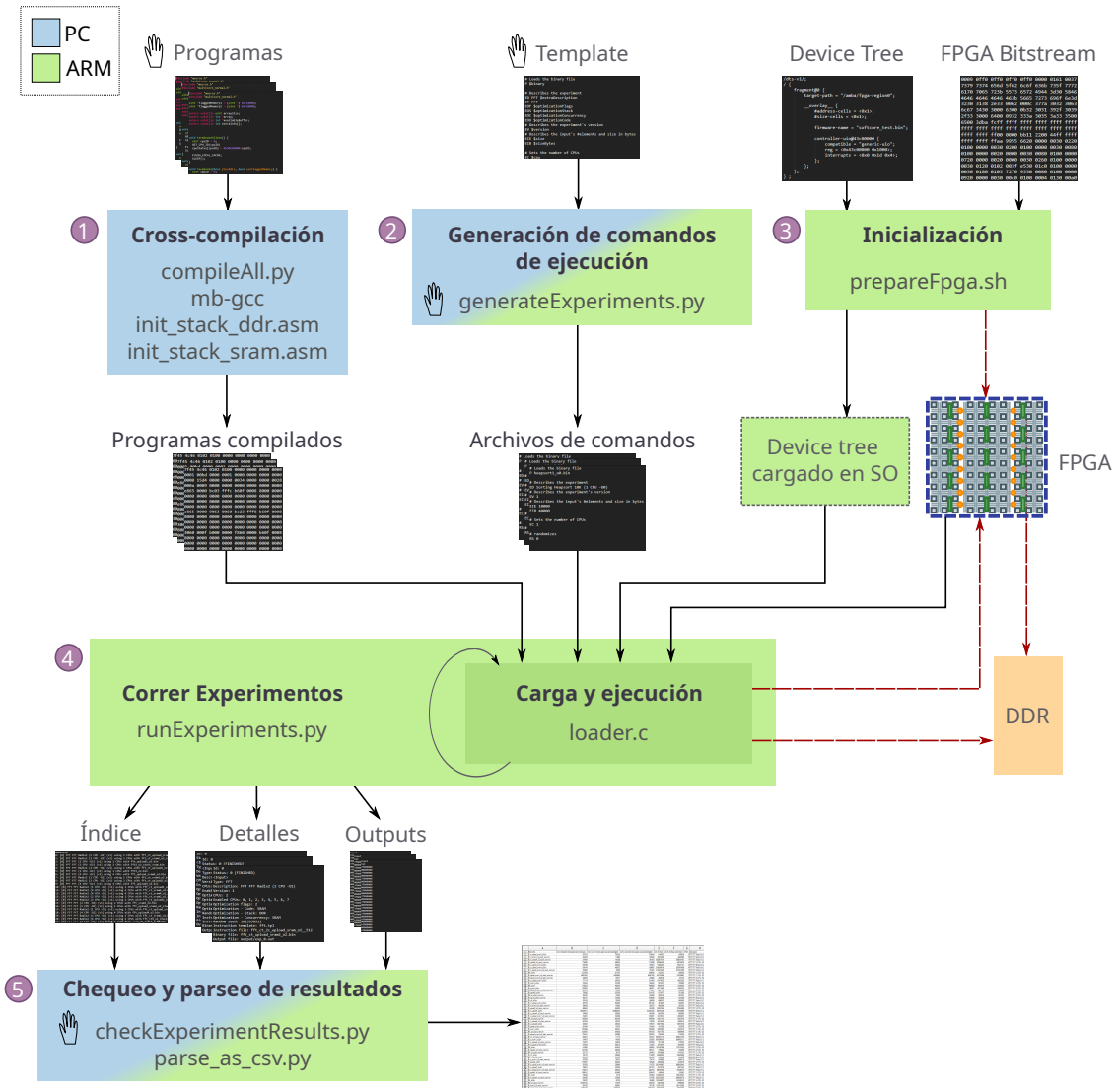


Figura 2.13: Procesos para crear y ejecutar experimentos. Las áreas azules (PC) y verdes (ARM) indican si el proceso puede ejecutarse en un ambiente o ambos. Las líneas rojas punteadas indican comunicación entre los componentes. Los procesos están enumerados de 1 a 5.
Se mencionan los principales *scripts* utilizados. Aquellos ítems que tengan el ícono de una mano requieren cambios a la hora de agregar un nuevo tipo de experimento.

Para facilitar la compilación de las variantes de los programas se cuenta con el *script* `compileAll.py` (1). Éste utiliza el cross-compilador *mb-gcc* (incluido en Vivado) para generar el código binario para MicroBlaze. Se opta por uno de dos *bootloaders* (`init_stack_ddr.asm` e `init_stack_sram.asm`) dependiendo de las optimizaciones aplicadas. En estos se definen la posición del *stack* y las variables necesarias del programa principal.

El *script* `generateExperiments.py` (2) tiene fijas todas las variantes a estudiar para cada tipo de experimento. Toma un archivo *template* y lo instancia a un archivo de comandos por cada una de estas variantes, incluyendo tamaños de entrada, cantidad de CPUs, optimizaciones utilizadas, distintas soluciones, etc.

Cualquiera sea el experimento, es necesario preparar la FPGA y el sistema operativo para ejecutar los experimentos. Para ello se utiliza `prepareFpga.sh` (3). Este paso es independiente del tipo de experimento y alcanza con correrlo una única vez luego de que el sistema operativo inicie.

Realizados estos tres procesos, mediante el *scriptZ* `runExperiments.py` se proceden a ejecutar todas las variantes preparadas ④, iterando sobre todos los archivos de comandos disponibles. Internamente, éste hace uso de *loader* para interpretar los comandos que ponen a correr los programas, y obtener sus métricas y resultados. Cada ejecución se registra en un índice de ejecuciones, y genera un archivo de salida y otro de resumen y mediciones.

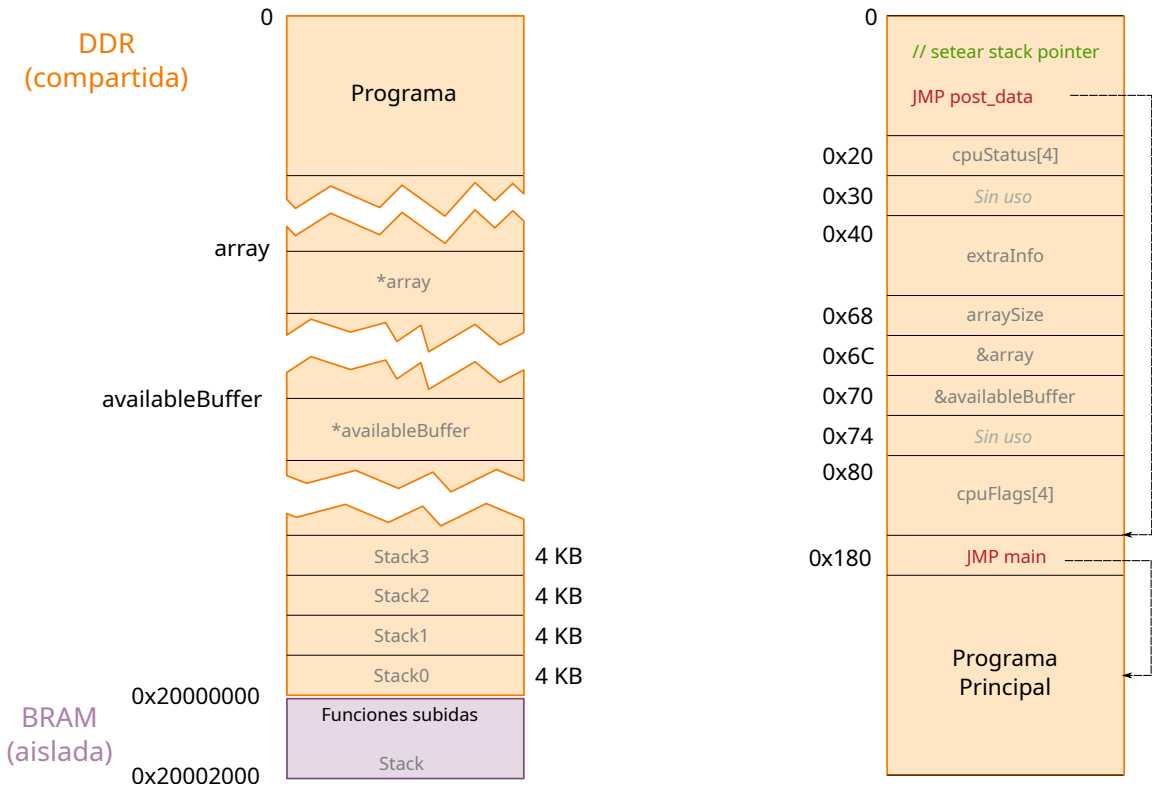
Finalmente, para experimentos con salida, éste se valida con `checkExperimentResults.py` ⑤. Si no hay discrepancias, se procede a utilizar `parse_as_csv.py` para agrupar las mediciones realizadas en los experimentos en un archivo *.csv* para su análisis posterior.

Más adelante explicaremos en detalle cada uno de estos procesos. Primero veremos cómo está estructurada la memoria para los programas que estaremos ejecutando en el `Sistema`.

2.3.1. Programas

Como hemos mencionado previamente, cada `MicroBlaze` direccionará a memoria principal a partir de la dirección `0`, por lo que allí cargaremos el programa a ejecutar (figura 2.14a).

Teniendo en cuenta que vamos a ejecutar código C compilado en un contexto *baremetal*, haremos uso de un *bootloader* para configurar el *stack* pointer antes de saltar al programa principal.



(a) Uso general de memoria. Stacks0-3 indican la posición de los *stacks* para los CPUs 0-3 en caso de usar la DDR. En caso contrario, se usará Stack (denotado en BRAM).

(b) Detalle de Programa, visto en (a). *Bootloader* y posición de variables utilizadas.

Figura 2.14: Detalle de uso de memoria para cada `MicroBlaze`.

En caso de querer usar la DDR, los *stacks* estarán a partir de la dirección más alta disponible, reservando 4 kB para cada CPU, en el orden que indica la figura 2.14a. En caso de querer usar la BRAM, el *stack* será configurado en la dirección más alta disponible de la BRAM. Como cada `MicroBlaze` tiene su BRAM exclusiva, no es necesario reservar espacio

para otros CPUs. De la misma manera, utilizaremos el inicio de la BRAM para subir las partes del programa que queramos ejecutar allí.

Definimos, además, dos posiciones de memoria útiles para la ejecución del programa: *array* y *availableBuffer*. Más adelante describiremos su función.

En la figura 2.14b podemos ver la composición del programa.

El *bootloader* será lo primero en ejecutar, y seteará el *stack* en la DDR o en la BRAM según el tipo de bootloader (`init_stack_ddr.asm` e `init_stack_sram.asm`). Tras hacer esto, hará un salto corto hacia el final del *bootloader* (luego de todas las estructuras de datos) y un segundo salto desde allí hacia la función `main` del programa principal. Este primer salto garantiza que la instrucción de salto sea de tamaño fijo y no desplace la dirección de los datos, algunos de las cuales serán configurados durante la carga del programa.

Estas estructuras de datos son:

- **cpuStatus**[**4**]: un array de cuatro elementos inicializado en `0` que será modificado por los CPUs a medida que finalizan su ejecución. Más adelante detallaremos los escenarios y valores posibles.
- **cpuFlags**[**4**]: un array de cuatro elementos inicializado en `0` que será leído y modificado por las funciones de concurrencia bloqueantes `waitJob(cpuID)` y `join()`, así como las funciones no bloqueantes `launchAll()`, `launchJob(cpuID)` y `finishJob(cpuID)` (ver algoritmo 1). Estos elementos están alineados a líneas de *cache* (64 B).
- **extraInfo**: un array que le permite al programa principal guardar información de *debug*, la cual aparecerá en el detalle de ejecución del experimento.
- **arraySize**: la cantidad de elementos en *array*.
- **&array**: un puntero al *Input* del programa principal.
- **&availableBuffer**: un puntero a memoria de uso seguro para resultados intermedios y/o *output*, dependiendo del experimento.

```
// flags alineados a 64 bytes
extern volatile __uint32_t cpuFlags[];
#define CPU_FLAGS(i) (cpuFlags[i * 16])

void launchAll() {
  for (i = 0; i < CPUS; i++) {
    CPU_FLAGS(i) = 0x0F;
  }
}
void launchJob(cpuID) {
  CPU_FLAGS(cpuID) = 0x0F;
}

void waitJob(cpuID) {
  while (!(*CPU_FLAGS(cpuID)));
}

void finishJob(cpuID) {
  CPU_FLAGS(cpuID) = 0;
}

void join(cpuID) {
  finishJob(cpuID);

  bool anyActive = 1;
  while (anyActive) {
    anyActive = 0;
    int i = 0;
    while (!anyActive && i < CPUS) {
      anyActive += CPUFLAGS(i);
      i ++;
    }
  }
}
```

Algoritmo 1: Funciones de Concurrencia utilizadas en el trabajo.

El programa principal estará escrito en *C* y contiene un *launcher* en su función `main`, compuesto de los siguientes pasos:

1. **Terminación rápida**: si el CPU no es necesario para la ejecución (su id es mayor o igual a la cantidad de CPUs habilitados), el programa terminará de manera rápida. Esto implica el seteo del **cpuStatus** del CPU en `0x0005DEAD` y el *flush* de la *cache*. El CPU entra en modo SLEEP.
2. **Subida de funciones**: si es necesaria la copia de parte del programa a BRAM, ésta se hace. En particular, las funciones de concurrencia siempre se suben, independientemente de si se van a usar o no.
3. **Ejecución**: se ejecuta el algoritmo/programa en sí.
4. **Terminación**: al finalizar, el CPU principal accede a una posición de memoria específica para notificar que la ejecución llegó a su fin. Tras hacer esto, se setea el **cpuStatus** del CPU en `0x0000DEAD` y se hace *flush* de la *cache*[3]. Si es el CPU principal, se accede a

[3]La finalización del algoritmo no necesariamente implica que los datos manipulados están en memoria

otra posición de memoria determinada para informar su terminación post-*cache flush*. El CPU entra en modo SLEEP.

2.3.2. Compilación

Para facilitar la compilación con las diversas variantes, desarrollamos el *script* denominado `compileAll.py`. Éste revisa todos los archivos *.c* dentro de una carpeta y procede a compilarlos para desde una a cuatro CPUs utilizando *mb-gcc* (parte de la *toolchain* de MicroBlaze).

Se supone que cada archivo *.c* maneja el define `CPUS` y genera la versión adecuada para el valor indicado. Es importante que los CPUs cuyo identificador sea mayor o igual al `define CPUS` terminen su ejecución rápidamente. En general, utilizaremos una, dos y cuatro CPUs, aunque hay un experimento donde también usaremos tres CPUs.

Además de compilar para los distintos números de CPUs, se compila una versión con el *stack* en la DDR (`init_stack_ddr.asm`) y una versión con el *stack* en la BRAM (`init_stack_sram.asm`). Para esto se linkea el archivo *.c* con el *bootloader* elegido.

El experimento tendrá distintas variantes de programa en forma de archivos *.c*. Estas variantes incluyen al menos cuatro casos: se suben las funciones de concurrencia a la BRAM, se sube parte del programa a la BRAM, ambas cosas se suben a BRAM, o ninguna se sube.

Adicionalmente se consideran las capacidades de nuestras instancias de MicroBlaze. Para que el código que sea generado utilice el máximo potencial de las instrucciones existentes para nuestro procesador fue necesario informar al compilador (*mb-gcc*) de nuestro *flavor*. Si esto no se hace, es posible que el código resultante termine simulando instrucciones por software, incurriendo en una pérdida innecesaria de rendimiento.

Para la configuración de MicroBlaze utilizada (vista en la figura 2.4), los *flags* seteados son:

- Barrel shifter: `-mxl-barrel-shift`.
- (EXTENDED) Floating Pointing Unit: `-mhard-float`, `-mxl-float-convert`, y `-mxl-float-sqrt`.
- (MUL64) Integer Multiplier: `-mno-xl-soft-mul` y `-mxl-multiply-high`.
- Integer Divider: `-mno-xl-soft-div`.
- Pattern Comparator: `-mxl-pattern-compare`.
- Otros: `-mcpu=v10.0` y `-mbig-endian`

En un contexto *baremetal*, el MicroBlaze solo estará ejecutando el código que le hayamos cargado. Esto significa que hay que tener presentes los vectores de interrupción. En nuestro caso, por la arquitectura descrita al inicio de este capítulo, no utilizamos interrupciones de MicroBlaze, por lo que podemos ignorarlos. Mediante los *flags* `-nostartfiles` y `-mxl-mode-novectors`, nos evitamos esta complejidad y su *overhead*.

En resumen, tenemos un *script* `compileAll.py` que se encarga de compilar los programas que queramos ejecutar. Por cada experimento tendremos al menos cuatro archivos *.c* (concurrencia en BRAM o no, parte del programa en BRAM o no). Cada una de estas variantes genera seis archivos de salida (tres distintas posibilidades de cantidad de CPUS y dos *bootloaders*).

principal, por esto es que se hace el *flush* de la *cache*.

Si bien en algunos casos agregamos la posibilidad de correr en tres CPUs, o tenemos otras variantes propias del experimento (distintas soluciones), en general tendríamos 24 archivos binarios resultantes por experimento.

Esta herramienta resultó fundamental al automatizar la compilación sin limitar la variabilidad de los experimentos.

2.3.3. Generación de Comandos

Cada experimento requiere un tipo de entrada distinto: *Heapsort*, por ejemplo, necesita una lista de valores para ordenar; Multiplicar Matrices requiere dos matrices; Fast Fourier Transform utiliza un arreglo de números complejos.

Queremos tener control del tamaño de estas entradas y sus valores, y nos interesa automatizar la carga del programa y la generación de la entrada de manera que:

- Se pueda repetir la ejecución de los experimentos para tener otorgar solidez estadística.
- Pueda cargarse un *Input* útil para cada experimento.
- Sea posible modificar casos de un experimento de manera rápida y sencilla.

Para automatizar la carga y ejecución de experimentos construimos la herramienta *loader* (compilada de *loader.c*) que se encarga de leer y procesar un archivo de comandos. En esta sección nos enfocaremos en la generación de estos archivos de comandos pero en la sección 2.3.5 hablaremos más en detalle sobre esta herramienta.

El *script* `generateExperiments.py` se encarga de generar todos los archivos de comandos a ejecutar del experimento. Se basa en un archivo de *template* del cual reemplaza "campos" de la forma `$nombre` por los finales. Por ejemplo, el campo `$size` denota la cantidad de elementos del *Input* del programa, y el campo `$timeout` marca el tiempo máximo de ejecución en segundos.

Algunos de estos campos son generales, presentes en todos los experimentos, y otros son específicos. Según los valores finales deseados (todos definidos para cada experimento dentro de `generateExperiments.py`), se generan tantos archivos de comandos como hagan falta.

A modo ilustrativo presentamos el archivo *template* para un experimento de *sorting*. Para comodidad del lector, se muestra dividido en cuatro partes:

- La primera se encarga de describir el experimento (qué binario utiliza, de qué se trata, qué optimizaciones utiliza, cuántas CPUs se quieren activas, etc).
- La segunda trata con el seteo de las variables o estructuras de datos vistas en la sección 2.3.1 y la carga del *Input*.
- La tercera inicializa el `Sistema` y pone a ejecutar el Programa.
- La cuarta guarda la salida del programa y finaliza el experimento.

En cada una detallaremos entre líneas lo que está ocurriendo:

```
# Comando P: Cargar el binario del programa
# Copia el contenido binario del archivo [binary] a memoria
P $binary

# Comandos E*: Detallar el experimento
# Descripcion
ED Sorting $extraDescription
# Tipo
ET Sorting
# Flags de optimizacion (en general es 2, de -O2)
EOF $optimizationFlags
# 1 tiene el stack en la BRAM, 0 si no
EOS $optimizationStack
# 1 si tiene las funciones de concurrencia en la BRAM, 0 si no
EOC $optimizationConcurrency
# 1 si sube parte del programa a la BRAM, 0 si no
EOU $optimizationCode

# Version del experimento (feature disponible pero no utilizada)
EV $version
# Entrada en cantidad de elementos y en bytes
EIE $size
EIB $sizeBytes

# Cantidad de CPUs activos
EC $cpu
# [Deprecado] Identificadores de CPUs activos (siempre 0,1,2,3,4,5,6,7)
EA $cpuNumbers
```

Algoritmo 2: Template de *Sorting* - Detalle de experimento

```
# Comandos R*: aleatoreidad
# La seed para iniciar el generador aleatorio
RS $random_seed

# Comandos L*: modificar memoria (en espacio de MicroBlaze)
# Cargar en la direccion 0x68 de memoria 1 valor de 32 bits (interpretado en decimal)
#   seguido del valor [size]
# Esto setea la variable arraySize del Programa
L 32 1 68
$size

# Cargar en la direccion 0x6C de memoria 1 valor de 32 bits interpretado en hexadecimal
#   seguido del valor "A00000"
# Esto setea la variable &array del Programa
Lh 32 1 6c
A00000

# Generar [size] enteros de 32 bits al azar entre [minNum] y [maxNum]
#   y guardarlos a partir de la posicion 0xA00000 de memoria
# Esto setea el contenido de array (puntero definido arriba)
R 32 $minNum $maxNum $size A00000

# Cargar en la direccion 0x70 de memoria 1 valor de 32 bits interpretado en hexadecimal
#   seguido del valor "B00000"
# Esto setea la variable &availableBuffer del Programa
Lh 32 1 70
B00000

# Comandos O*: escribir en el archivo de salida del experimento
# Escribir el string "Input" (con un <enter> al final)
OS Input
# Escribir el string [size] (con un <enter> al final)
OS $size
# Escribir [size] enteros de 32 bits desde la posicion de memoria 0xA00000
OM 32 $size A00000
```

Algoritmo 3: Template de *Sorting* - Inicialización

```
# Comando I: Preparar el Sistema para la ejecucion
I

# Comando S: Iniciar/resumir la ejecucion
# Iniciar la ejecucion por [timeout] segundos y pausar si se detecta un acceso a la
↪  direccion de memoria 0xF10000
# El programa accedera a esa direccion cuando haya terminado de correr
#   pero antes de hacer los flushes de cache
S $timeout F10000

# No se ejecutaran los comandos siguientes hasta que se detecte el acceso
# El timeout detendra la ejecucion del resto de las comandos

# Guarda los tiempos medidos del experimento hasta ahora
ES

# Resumir la ejecucion por [timeout] segundos y pausar si se detecta un acceso a la
↪  direccion de memoria 0xF20000
# El programa accedera a esa direccion cuando hecho flush de la cache
S $timeout F20000
```

Algoritmo 4: Template de *Sorting* - Ejecución

```
# Escribir el string "Output" (con un <enter> al final)
OS Output
# Escribir el string [size] (con un <enter> al final)
OS $size
# Escribir [size] enteros de 32 bits desde la posicion de memoria 0xA00000
# Para este caso, el programa debera escribir el resultado a partir de esa direccion
OM 32 $size A00000

# Comando F: Concluir el experimento
F
```

Algoritmo 5: Template de *Sorting* - Finalización

Un detalle a mencionar es que el *template* puede soportar más de un programa compilado que haga un *sorting*, ya sea *heapsort*, *quicksort*, u otro (siempre que esté adaptado a las especificaciones del trabajo). De la misma manera, variantes de un tipo de experimento pueden utilizar el mismo *template* para generar sus comandos de carga, difiriendo únicamente en el binario utilizado y algunos detalles de la descripción (todos estos valores definidos dentro del *script* `generateExperiments.py`).

2.3.4. Preparación para ejecución

Para poder correr los experimentos es necesario configurar la FPGA. Para esto requerimos de un *bitstream*, el resultado final de implementar nuestro diseño de hardware en Vivado. Podemos configurar la FPGA desde el ARM utilizando este *bitstream* y algunas herramientas mencionadas en la sección 1.5.4.

Una estructura de datos para describir los componentes de hardware es el *device tree*. Desde los buses y periféricos integrados hasta los CPUs disponibles, esta estructura puede encontrarse predefinida o generarse durante las primeras etapas de booteo. Algunos sistemas operativos modernos la utilizan para conocer el hardware donde están instalados, aunque otros lo descubren utilizando protocolos de autoconfiguración como ACPI.

Para el caso de nuestro trabajo, como parte de la inicialización agregamos al *device tree* un dispositivo que mapea las direcciones de memoria del Softcore Controller y los puertos de interrupción que utilizamos. Gracias al trabajo de Ichiro Kawazome en su repositorio `https://github.com/ikwzm/FPGA-SoC-Linux`, esto puede hacerse desde espacio de usuario mientras el sistema operativo está ejecutando.

Las interrupciones al ARM generadas desde el Softcore Controller merecen especial atencion: el *device tree* mencionado está configurado para registrar un dispositivo (`/dev/uio0`) que se encarga de recibir estas interrupciones y permitir que aplicaciones de espacio de usuario puedan, de manera bloqueante, esperarlas. Dicho esto, decidimos no hacer uso de estas interrupciones por hardware por *delays* introducidos durante el proceso de experimentación, detalle que explicaremos en breve.

El script `prepareFpga.sh` se encarga de hacer la preparación. Utiliza un `Rakefile` armado en base en uno de los ejemplos del repositorio para ejecutar tareas. Un `Rakefile` es un archivo de comandos muy similar a un `Makefile` y consiste en una serie de tareas que definen una lista comandos. En un `Rakefile`, al igual que en un `Makefile`, se construye un árbol de dependencias entre las tareas ejecutables.

Las tareas realizadas por el *script* son las siguientes:

- Se procesa el *bitstream* y se genera un archivo que puede usarse para hacer el *flash* de la FPGA (`fpga-bit-to-bin.py`, provisto por el repositorio arriba mencionado).
- Se hace el *flash* de la FPGA.
- Se elimina el *device tree* implementado y se instala el más reciente (usando `dtbocfg.rb`, también del repositorio).

El *script* `prepareFpga.sh` solo necesita ejecutarse una vez tras cada *restart* del sistema operativo. Si es necesario utilizar otro *bitstream*, o se modifica el *device tree*, este *script* deberá ser ejecutado nuevamente. En este trabajo siempre utilizamos el mismo *device tree* y el mismo *bitstream* para todos los experimentos.

2.3.5. Ejecución

Teniendo los programas compilados, sus archivos de comandos generados y la FPGA y ARM configurados, estamos en condiciones de realizar los experimentos.
El *script* `runExperiments.py` hace diez rondas de una iteración al azar de todos los archivos de comandos disponibles. Por cada uno ejecuta el *loader*, el cual se encarga de interpretar el archivo de comandos correspondiente para poner en marcha el experimento.

El orden de ejecución aleatorio en cada ronda merece una aclaración: de haber alguna dependencia de ejecución, es decir, si el resultado variara de alguna forma según en orden en que se ejecuten los experimentos, la ejecución al azar debería disminuir este impacto de manera estadística.

`runExperiments.py` imprime en pantalla el estado de la ejecución general: cuántos experimentos han corrido en esta ronda, el tiempo transcurrido hasta el momento, una estimación del tiempo restante (pasada la primera ronda). También muestra datos generales del experimento actual: tipo, binario utilizado y descripción, tiempo que demoró en segundos, si terminó sin problemas detectados, etc.

El *loader* es el responsable de la interacción con el `Softcore Controller`, y es quien carga los programas, sus *inputs*, y debe ser capaz de extraer el resultado de su ejecución.

Accede al `Softcore Controller` mediante un *driver* configurado durante la carga del *device tree*, mientras que el acceso a memoria se hace a través de `/dev/mem`. Para ambos casos utiliza `mmap`, ya sea para leer o modificar los registros del `Softcore Controller` o para manipular posiciones de memoria del `Sistema`.

Si bien la FPGA genera interrupciones para notificar eventos, desde el punto de vista del *loader*, no son capturadas (se ignoran). Esto se debe a que el mecanismo de interrupciones resultó ser más lento de lo esperado. Dada la cantidad de ejecuciones a realizar, este margen de espera se hizo significativo. Por esto, y siendo que solo afectaría el desempeño de los ARM (que no nos interesa), el *loader* utiliza *polling* para detectar el cambio de estado del `Softcore Controller`.

Cada ejecución del *loader* toma un archivo de comandos y genera o modifica tres archivos:

- El primero de ellos es `./results/index`. Su función es la de enumerar los experimentos ejecutados hasta el momento, con una breve descripción de cada uno. El número identificador del experimento (*id*) se define a partir de este archivo. Empieza en 0 y se incrementa en uno por cada experimento que haya finalizado. Interrumpir un experimento terminando el *script* de manera forzosa no incrementará el contador de *index*.

- El segundo es `./results/output/exp_[id].out`. Si el archivo de comandos genera un *output*, se escribirá en éste. Este archivo se utiliza para verificar que el resultado de correr el experimento es el correcto (si es un experimento de *sorting*, debería ordenar correctamente). Para los experimentos que necesiten esta verificación, el archivo de comandos se encargará de que se guarde tanto el *Input* como el *Output*.

- El último es `./results/exp_[id]`. Puede pensarse como el archivo de detalle del experimento. Contiene la totalidad de las métricas capturadas, junto con un memory dump de las variables vistas en la sección 2.3.1 y en la figura 2.14b.

A continuación mostramos un ejemplo del archivo resultado, con comentarios intercalados, y dividido en dos partes para una mejor integración en este documento:

```
Id: 11
Status: 0 (FINISHED)
<Input>
# varios de los campos siguientes vienen definidos en el archivo de comandos del
↪  experimento
Type: Sorting
Description: Sorting Heapsort 8K (2 CPU -O2)
Version: 1
CPUs: 2
Enabled CPUs: 0, 1, 2, 3, 4, 5, 6, 7
Optimization flags: 2
Optimization - Code: DDR
Optimization - Stack: DDR
Optimization - Concurrency: DDR
Random seed: 1613612624
Instruction template: sorting.tpl
Instruction file: heapsort_8K_2c_o2_stack_sram
Binary file: heapsort2_o2_stack_sram.bin
Output file: output/exp_11.out
Program size (bytes): 5832
Input size (bytes): 32768
Input size (elements): 8192
<Timings>
# las metricas a continuacion (los tiempos son en segundos y fueron medidos utilizando
↪  la funcion clock_gettime)
# tiempo de ejecucion del programa loader
Full time: 1.124088
# tiempo de ejecucion del experimento en MicroBlaze total
Experiment time (full): 0.264115
# tiempo de ejecucion del experimento en MicroBlaze exceptuando el flush de cache final
Experiment time (without cache flush): 0.232814
# CLK_W: ciclos de clock en los que el Sistema estuvo trabajando
# CLK_I: ciclos de clock en los que el Sistema estuvo en pausa
CLK_W (full): 25865875
CLK_W (without cache flush): 22816635
CLK_I (between interruptions and continues): 64807019
CLK_I (until each interruption was acknowledged): 545717
CLK_I (until 1st interruption was acknowledged): 464800
```

Algoritmo 6: Archivo de detalle de un experimento - Mediciones

```
<Memory dump>
<STATUS>
# valores (en hexadecimal) tomados de cpuStatus (0x20)
dead0000
dead0001
5dead002
5dead003
<OLD FLAGS>
# [deprecado] valores (en hexadecimal) tomados de 0x30
00000000
00000000
00000000
00000000
<NEW FLAGS>
# valores (en hexadecimal) tomados de cpuFlags (0x80, 0xC0, 0x100 y 0x140)
00000000
00000000
00000000
00000000
<EXTRA INFO>
# valores (en hexadecimal) tomados de extraInfo (0x40)
00002000
00a00000
00b00000
052cd7e0
01a5e808
00000000
00000000
00000000
00000000
00000000
```

Algoritmo 7: Archivo de detalle de un experimento - Volcado de memoria (*Memory Dump*)

2.3.6. Chequeo y parseo de resultados

Es importante poder verificar que el programa se ejecutó correctamente. Para ello utilizamos el *script* `checkExperimentResults.py`, el cual debe ser modificado a la hora de agregar un tipo de experimento nuevo.

El *script* busca en el directorio `results/` los archivos `results/exp_[id]` y parsea (según el experimento) su archivo de resultados en `results/output/exp_[id].out`.

Los experimentos se verificarán según lo guardado en el segundo archivo (el detalle), detectando el tipo de experimento mediante el campo *Instruction template* o el campo *Type*. Leyendo el *Input*, se corre el algoritmo correspondiente al experimento y se verifica contra la salida registrada.

La implementación de los algoritmos para verificar los resultados es nativa de `Python` o está incluida en la biblioteca `numpy`. En la sección correspondiente a cada experimento mencionaremos cómo se hizo este proceso.

Se reportan los resultados a medida que se hacen las verificaciones, indicando si tuvieron algún error y dónde.

Al finalizar el *script* se indica la cantidad de experimentos fallidos con errores identificados.

El script `plotExperimentResults.py` es una variante de `checkExperimentResults.py`. Además de verificar los resultados, genera un gráfico con el resultado de correr la FFT según el `Sistema`, `numpy`, y la diferencia entre ambos.

Se genera al menos uno por cada variación del *input* y los experimentos fallidos siempre generan un gráfico para una mejor visualización del problema.

Cuando todos los experimentos fueron verificados como correctos, lo siguiente es agrupar todas sus métricas en un archivo para su análisis. El *script* `parse_as_csv.py` se encarga de generar un archivo `.csv` con las mediciones de todos los archivos `results/exp_[id]`, tomando los campos como columnas.

Capítulo 3

Algoritmos y experimentos

En este capítulo detallaremos los experimentos realizados. Con el Framework construido en el capítulo anterior, nos resta diseñar programas y variantes de estos que puedan aprovecharse de la BRAM, así como definir las métricas sobre las cuáles haremos la comparación.

En primer lugar presentaremos la técnica que aplicaremos. Explicaremos de qué se trata y enumeraremos los tipos de uso que nos interesan.

Luego listaremos los algoritmos y variantes en los que será puesta a prueba y proveeremos una breve motivación por la que fueron elegidos.

Finalmente, entraremos en detalle de cada uno. Nos explayaremos en las variantes consideradas y explicaremos la estrategia de paralelización. Luego mostraremos los resultados junto con un análisis de los mismos y finalizaremos con una conclusión.

3.1. Técnica

Uno de los recursos más importantes es la memoria. Utilizar una memoria con mejor desempeño que la principal (sea más rápida o más cercana al CPU) debería mejorar los tiempos de ejecución del programa. En este sentido, hacer un traslado del código o los datos en uso permitiría reducir los tiempos ante accesos repetidos. El uso de *caches* es un ejemplo de esto, y es una solución de hardware transparente al programador.

En nuestro caso, la *cache* L1 de instrucciones está conectada por ACE a la L2. Para el caso de la L2, además, hay una latencia extra al tener que garantizar coherencia.

La técnica que utilizaremos consiste en que el programa direccione o copie parte de la memoria principal a una memoria local más rápida, aislada de la *cache*. Esta memoria será la BRAM mencionada anteriormente, conectada al MicroBlaze mediante puertos LMB (*Local Memory Bus*), uno para datos y otro para instrucciones. Si bien la L1 y la L2 también se implementan sobre BRAM, requieren un hardware más complejo para implementar su función de *cache*.

El LMB garantiza una lectura/escritura por ciclo del CPU[1].

Para todos los experimentos analizaremos las siguientes tres aplicaciones de esta técnica, dando a lugar ocho combinaciones posibles. Nos referiremos como **aplicaciones** o **aplicaciones de la técnica** a "aplicar" uno o varios de estos usos de BRAM:

[1] http://www.cs.columbia.edu/~soviani/cs4840hack/Microblaze_timing.html

Subir parte del código del programa a BRAM

Algunas partes del programa, como *loops* o ciertas funciones, tendrán mayor uso que otras. Moverlas a la BRAM puede llegar a mejorar el rendimiento al estar siempre disponibles y no tener que depender de la *cache*, quien podría desalojarlas.

Trataremos de subir la mayor parte de las funciones utilizadas por el programa que nos sea posible y que pueda entrar en la BRAM (de 8 kB). En ocasiones no será posible subir todo y será tarea del programador elegir qué funciones o partes del código conviene que estén en BRAM.

Subir parte del código de concurrencia a BRAM

`MicroBlaze` no implementa instrucciones de concurrencia por hardware [8]. Dado que no estamos usando hardware adicional para garantizar concurrencia entre *cores*, utilizaremos *spin-locks* para `wait` y `join`. Estos estarán accediendo a posiciones en memoria externa y haciendo saltos cortos hasta que su valor sea modificado. Mover estas instrucciones a BRAM puede llegar a reducir los *cache misses*.

Como la BRAM es aislada a cada `MicroBlaze` y la naturaleza de los datos es compartida, la dirección de memoria a la que acceden seguirá estando en el espacio de la memoria principal y quedará a merced de la *cache*.

Si bien aún requieren acceder a memoria principal para detectar cambios, al mantener las instrucciones en BRAM se evita que el procesador en *spin-lock* haga uso del *cache* de instrucciones.

Este es un ejemplo concreto de la aplicación mencionada arriba para un fragmento de código muy reducido.

Utilizar el *stack* en BRAM

De ser pocas, las variables locales se pueden traducir a registros de CPU. En caso contrario, se apilan en un *stack* y son direccionadas usando un puntero base. Cuando se modifica el *scope* en un programa (por ejemplo, al llamar a una función), el *stack* es alterado con nuevas variables locales. Al salir del *scope*, el *stack* vuelve a su posición anterior.

En esta aplicación de la técnica buscamos mejorar los tiempos de acceso a operaciones que utilizan variables locales. Para ello, el *stack* pointer se inicializa apuntando al área de memoria perteneciente a la BRAM. Esto permite que los accesos al *stack* utilicen una memoria más rápida que la principal, aislada de la *cache* y de los otros CPUs.

El código es generalmente de sólo lectura. Únicamente los *cache-misses* post desalojo generarían accesos extra a memoria principal. Los datos, en cambio, pueden ser leídos y/o escritos. Al mantener el *stack* en BRAM no es necesario enviar el dato a memoria principal, lo que debería reducir tanto el uso de la *cache* como eliminar una parte de las escrituras a memoria principal.

Cabe aclarar que esta aplicación de la técnica implica que los datos de la pila no pueden estar compartidos entre varios CPUs (la BRAM está aislada).

Algoritmos

Trivialmente paralelizable

Variantes: ninguna.
Entrada: WU y J (Enteros).
Salida: No aplica.

Sea `P` un programa y `WU` un número natural. Nos interesa correr `P` un total de `WU` veces (*Working Units*) entre varios CPUs e intercalar `J` *joins* durante las ejecuciones. Estos *joins* actuarán como puntos de sincronización de los CPUs. Definimos esta versión como `TrivialmenteParalelizable(P, WU_per_CPU, J, CPUs)`.

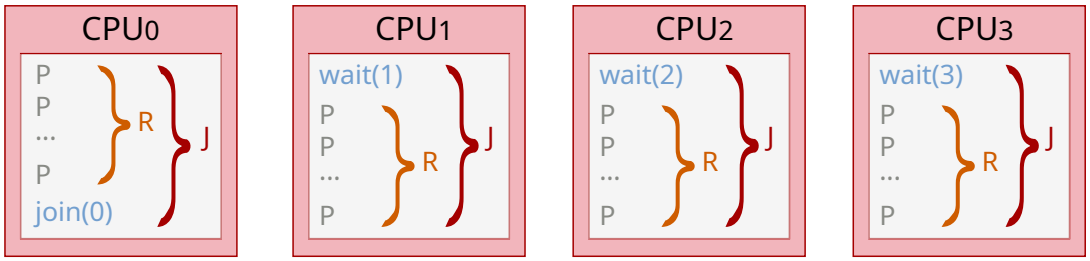


Figura 3.1: Ejemplo de particionamiento en `R` repeticiones de `J` *joins* de un esquema trivialmente paralelizable utilizando un programa `P` ($R * J * \#CPUs = WU$). Mostramos la diferencia entre el CPU0 que hace los *joins*, y los otros CPUS que esperan (*wait*) a ser "lanzados" (este "lanzado" y otras operaciones fueron omitidas para simplificar la visualización).

Figura 3.2: Ejemplo concreto de particionamiento de 16 `WU_per_CPU` en 1, 2 y 4 *joins* de un esquema trivialmente paralelizable. Mostramos la diferencia entre el CPU0 que hace los *joins*, y los otros CPUS que esperan (*wait*) a ser "lanzados" (este "lanzado" y otras operaciones fueron omitidas para simplificar la visualización).

Partiremos las `WU_per_CPU` ejecuciones en varias repeticiones `R` = `WU_per_CPU` / `J` con un *waitJob(cpuID)* al principio y un *join()* al final, tal como muestran las Figuras 3.1 y 3.2. Para `CPUs` = 4, `WU_per_CPU` = 4096 y `J` = 16, estaremos ejecutando un total de 16.384 veces el programa `P`, partido en 16 bloques de 256 repeticiones por cada uno de los cuatro CPUs.

Nuestro programa `P` será una doble ejecución del algoritmo de MD5 usando una entrada fija. La idea es que se trate de un programa determinístico que haga cálculo no trivial utilizando CPU y memoria.

Motivación MD5 es un algoritmo que involucra operaciones no triviales sobre la entrada. Más aún, puede ejecutarse utilizando el *stack* y sin accesos a memoria externa. Esto lo convierte en una opción interesante para la optimización de *stack*.

El problema trivialmente paralelizable elegido nos sirve para modelar una familia de programas cuya carga de trabajo puede partirse con facilidad y donde pueden requerir sincronizar sus CPUs en puntos intermedios de la ejecución.

Siendo parte de la entrada, podemos manipular el número de puntos de sincronización (`J`) y la distancia entre estos (`R`) previo a la ejecución. Variar estos parámetros nos permite medir el impacto en el desempeño seguń la cantidad de *joins*, así como la diferencia (si la hay) en el caso de subir las funciones de concurrencia a BRAM.

Considerando que la definición es `TrivialmenteParalelizable(P, WU_per_CPU, J, CPUs)`, cada vez que agregamos un CPU estamos agregando la carga de ese CPU (`WU_per_CPU`). Esto es conocido como **escalabilidad débil** y es algo que nos interesa analizar para aprender más sobre la arquitectura propuesta.

Multiplicación de matrices

Variantes: $A \times B$ y $A \times B^t$.
Entrada: N (entero) y matrices de Enteros A y B (de dimensión $N \times N$).
Salida: Matriz de Enteros.

Motivación La forma *naive* de multiplicar matrices recorre columnas y filas de manera completa, realizando una operación simple entre las celdas. Esto implica varios accesos a memoria con bajo tiempo de procesamiento entre cada uno. La diferencia entre multiplicar usando B^t es la posibilidad de recorrer ambas matrices por filas, reduciendo los *cache misses* dado el **principio de localidad**.

En definitiva, este algoritmo utiliza un patrón simple y repetitivo de acceso a memoria que es fácil de estudiar y analizar.

Heapsort

Variantes: ninguna.
Entrada: Arreglo de Enteros.
Salida: Arreglo de Enteros.

Motivación *Heapsort* sobre un arreglo implica una alta probabilidad de saltos de línea y, por lo tanto, *cache misses*. El paralelizado se hará dividiendo la entrada entre los CPUs disponibles, ejecutando *heapsort* en cada uno y haciendo merge-sorts al final. Esto implica un número constante de *wait* y *joins*. Siendo que la implementación es recursiva, el tope del *stack* estará tomando distintas posiciones de memoria a lo largo de su ejecución.

FFT - Fast Fourier Transform

Variantes: *iterativa* y *recursiva*.
Entrada: Arreglo de números complejos.
Salida: Arreglo de números complejos.

Motivación Explorar el uso de un algoritmo complejo que utiliza operaciones de punto flotante y funciones intrínsecas mediante la `libm` de `C`, así como analizar el comportamiento de sus variantes frente al aumento del tamaño de entrada y ante las distintas aplicaciones de la técnica.

Metodología experimental

Utilizaremos un único *bitstream* para todos los experimentos. Es decir, la FPGA estará implementando el mismo hardware. Como hemos mencionado previamente, los *softcores* empiezan a correr todos a la vez apenas inicia el `Sistema`.

Todos los experimentos fueron compilados con `-O2`.

Utilizaremos *spin-locks* (`waitJob(cpuID)` y `join()`) para sincronizar las operaciones (ver algoritmo 1 y figura 3.3). Este `join()` no desaloja ni desactiva ningún procesador, si no que solo funcionará como un `barrier`.

El procesador principal será el *softcore* con id 0 y será quien en general haga los *joins*. El resto de los *softcores* podrán llamar a `waitJob(cpuID)` para permanecer en *spin-lock* hasta que sean "lanzados" (`launchJob(cpuID)`) por algún *softcore* activo, para luego llamar a `finishJob(cpuID)`.

Este patrón de uso asegura que sólo un CPU estará leyendo *cpuFlags[i]* y sólo un CPU estará escribiendo *cpuFlags[i]*, siendo *i* un `cpuID`. El protocolo ACE nos garantiza que la lectura y escritura es coherente entre los CPUs.

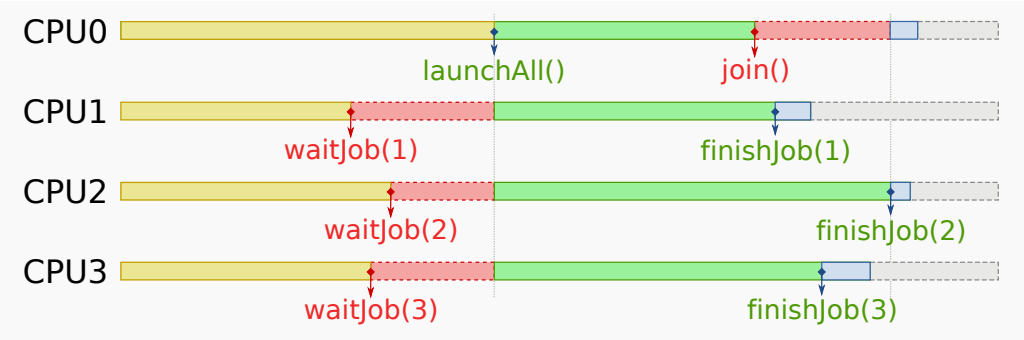


Figura 3.3: Ejemplo de sincronización entre procesadores utilizando las funciones definidas en el algoritmo 1. El color amarillo representa código de preparación. En rojo se muestra la espera activa (*spin-lock*). La ejecución del código aplicativo está en verde. Finalmente, en celeste, tenemos el código de finalización del programa. La longitud de las barras es meramente gráfica y no es necesariamente representativa del tiempo real.

Nuestra metodología se basa en estudiar el desempeño de las distintas configuraciones de software y hardware del `Sistema`. Para analizar los resultados de cada experimento elegimos el siguiente subconjunto de estas configuraciones:

- **CPUs**: Cantidad de CPUs utilizados.
- **experimento**: Tipo o variante de experimento.
- **code_upload**: Si parte del código del algoritmo se subió a la BRAM o si se encuentra todo en la DDR.
- **stack**: Si el *stack* está en la BRAM o en la DDR.
- **concurrencia**: Si las funciones de concurrencia se subieron a la BRAM o quedaron en la DDR (ver algoritmo 8).

- **Entrada**: Tamaño de entrada (la interpretación depende del experimento).

```
void waitJob(cpuID) {
  while (!(*CPU_FLAGS(cpuID)));
}

void join(cpuID) {
  finishJob(cpuID);

  bool anyActive = 1;
  while (anyActive) {
    anyActive = 0;
    int i = 0;
    while (!anyActive && i < CPUS) {
      anyActive += CPUFLAGS(i);
      i ++;
    }
  }
}
```

Algoritmo 8: Fragmento de funciones de Concurrencia, extraído de algoritmo 1. Las líneas marcadas indican las partes de las funciones que serán subidas a BRAM.

Sin importar el tiempo final del experimento, cada experimento será ejecutado 10 veces. Haremos el análisis utilizando el promedio de `CLK_W` y la relación porcentual del desvío estándar con éste:

- **CLK_W**: Ciclos de *clock* de `Sistema` en los que se estuvo ejecutando el experimento (previo al *flush* de *cache* final). Esto mide el tiempo desde que el programa inicia su ejecución. Es decir, incluye los tiempos de subida de código a BRAM.

Asumiendo que el comportamiento de la FPGA es determinístico, notamos que los experimentos pueden sufrir variaciones en el tiempo principalmente por el acceso a la memoria principal, la cual también puede estar siendo utilizada por el sistema operativo del ARM.

Finalmente, según el experimento puede ser necesario agregar o segmentar alguna configuración. Un ejemplo de esto es el experimento `Trivialmente Paralelizable`, cuya **Entrada** se divide en `WU` (*Working Units*) y `J` (*Joins*).

3.2. Algoritmos trivialmente paralelizables

Un buen punto de partida para el análisis de este trabajo sería implementar un algoritmo simple y con resultados predecibles. Tomando un programa `P` sencillo, si lo corremos repetidas veces, una tras otra, estaríamos multiplicando la carga del sistema:

Definimos 1 *WU* (*Working Unit*) como el trabajo realizado por una ejecución. R repeticiones en teoría resultarían en R *WU*, asumiendo que cada ejecución es independiente de las ejecuciones previas. Si cada CPU le hacemos correr R repeticiones, tendríamos un total de $\mathrm{R} \cdot \mathrm{CPUs}(WU)$ en todo el sistema.

Al tener varios CPUs, otro factor a analizar son las barreras de sincronización. Es decir, que los CPUs esperen a que los demás terminen su "cuota" antes de continuar. Agregando estas barreras entre bloques de ejecuciones podemos tener una idea de cómo impactan en los tiempos. Si tenemos J barreras (*joins*) de R repeticiones, cada CPU estaría manejando $\mathrm{J} \cdot \mathrm{R}(WU)$, dando un total de $\mathrm{T} = \mathrm{CPUs} \cdot \mathrm{J} \cdot \mathrm{R}(WU)$ para todo el sistema.

El pseudocódigo en Algoritmo 9 ilustra la idea.

Una *bag of tasks* donde se ejecuta `P` un total de T veces es trivialmente paralelizable entre varios CPUs utilizando el esquema anterior (eligiendo T de forma que sea divisible por los CPUs). Variando R y J podemos observar experimentalmente cómo se relacionan con el tiempo final las *WU* para casos con pocas ejecuciones entre *joins*, por ejemplo, o si la cantidad de *joins* es un factor significativo si están lo suficientemente distanciados.

```
void mainCPU_execute(J, R) {
    for (j = 0; j < J; j++) {
        launchJob(1);
        launchJob(2);

        for (r = 0; r < R; r++) {
          execute(0);
        }

        join(MAIN_CPU);
    }
}

void slaveCPU_execute(cpuID, J, R) {
    for (j = 0; j < J; j++) {
        waitJob(cpuID);
        if (cpuID == 1)
                launchJob(3); // CPU0 libera dos CPUs y CPU1 libera CPU3
                              // en vez de que CPU0 libere los tres CPUs
                              // esto ayuda a equilibrar los tiempos de
                              // inicio de ejecución

        for (r = 0; r < R; r++) {
          execute(cpuID);
        }

        finishJob(cpuID);
    }
}

void execute(cpuID) {
    f(value);          // "value" hardcodeado
}
```

Algoritmo 9: Algoritmo Trivialmente Paralelizable

3.2.1. Experimentos

El programa `P` elegido para utilizar como generador de carga será una implementación de MD5 adaptada para este trabajo[2].

Al mantener el tamaño de entrada de MD5 constante, el tiempo de ejecución debería ser constante sin importar qué entrada tenga.

Además de elegir el programa, necesitamos definir los valores J y WU que van a utilizarse en la ejecución.

Generamos experimentos a partir de las combinaciones posibles de $J \in \{1, 16, 128, 512\}$ y $WU \in \{512, 2048, 8192, 32768\}$, donde J son *joins* y WU es el trabajo para cada CPU. Llamaremos a MD5 dos veces por cada WU, repartiendo las *Working Units* en J sincronizaciones de R ejecuciones seriales ($WU = J \times R$).

Esperamos tiempos levemente mayores a medida que sube J debido a la sincronización pero no deberían ser significativos. Más aún, a medida que incremente la cantidad de CPUs utilizados, esperamos que los tiempos vayan en aumento para mismos valores de WU y J.

[2] `https://gist.github.com/creationix/4710780#file-md5-c-L160`

3.2.2. Resultados y Análisis

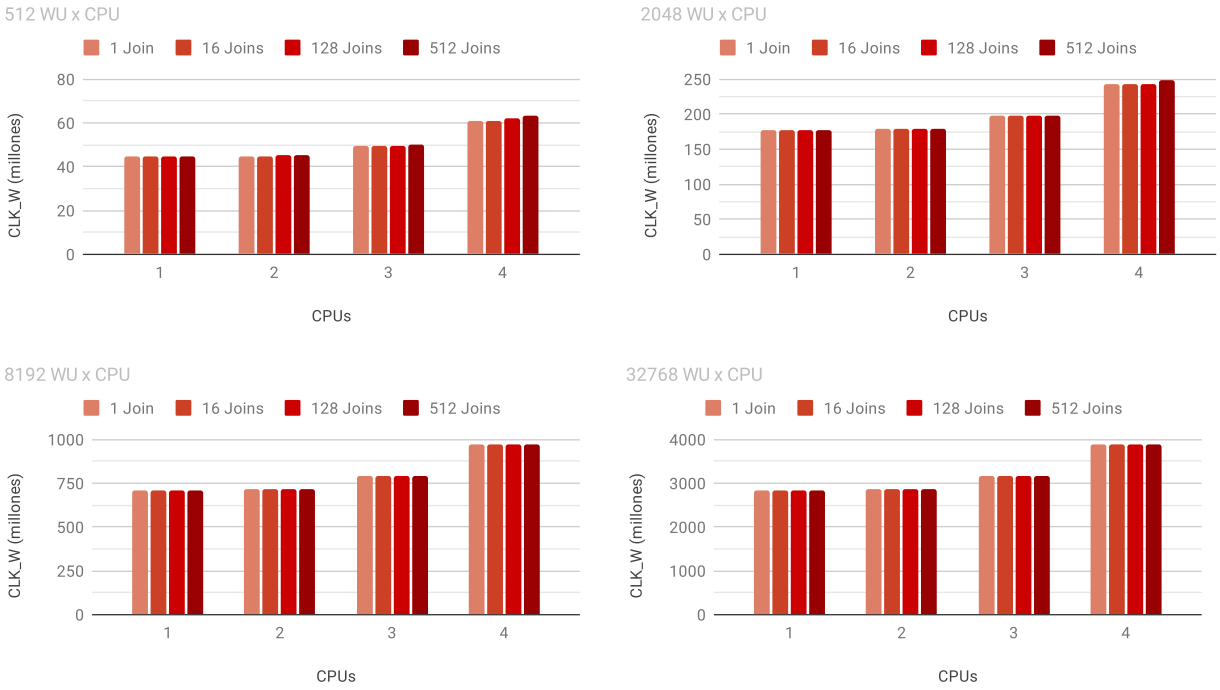


Figura 3.4: Ejecución del programa Trivialmente Paralelizable sin utilizar BRAM para tamaños (en WU) de 512, 2048, 8192 y 32,768.

En la figura 3.4 podemos ver cuatro gráficos, cada uno representando distintas variaciones de los argumentos del algoritmo base. Es decir, sin utilizar BRAM.

Notamos que los casos con una y dos CPUs parecen mantener valores similares de `CLK_W`. Con tres y cuatro CPUs, en cambio, se puede percibir un incremento significativo de esta magnitud. Respecto de 1 CPU, el incremento es del orden de entre 11 % a 12 % para tres CPUs y va de 37 % a 42 % para cuatro CPUs. El caso más visible es el de para 32,768 WU, en donde ésta pasa de menos de 3000 a más de 3000 millones y con cuatro CPUs, cerca de 4000 millones.

También se puede apreciar que los *joins* no introducen diferencias significativas en el desempeño del algoritmo para valores de WU suficientemente grandes:

- Un CPU: comparando con un *join* hay una diferencia general $\leq$ 0,08 % para 16, 128 y 512 J.
- Dos CPUs: la diferencia general es $\leq$ 0,18 %, y de 0,86 % para el caso de 512 WU y 512 J.
- Tres CPUs: $\leq$ 0,51 % en general, y 1,59 % para el mismo caso de arriba.
- Cuatro CPUs:
  - 8K y 32K de WU: difieren en menos de 0,09 %.
  - 2K WU: difieren en menos de 0,09 % para 16 y 128 J y 2,16 % para 512 J.
  - 512 WU: tenemos 0,18 % para 16 J, 2,22 % para 128 J y 4,13 % para 512 J.

Esto parece apuntar a que al incrementar los CPUs, tener sincronizaciones muy frecuentemente produce un detrimento en el rendimiento.

Finalmente, vemos que en los cuatro gráficos el incremento de las *Working Units* parece, a simple vista, afectar en forma lineal los tiempos de ejecución.

Variación por cantidad de CPUs

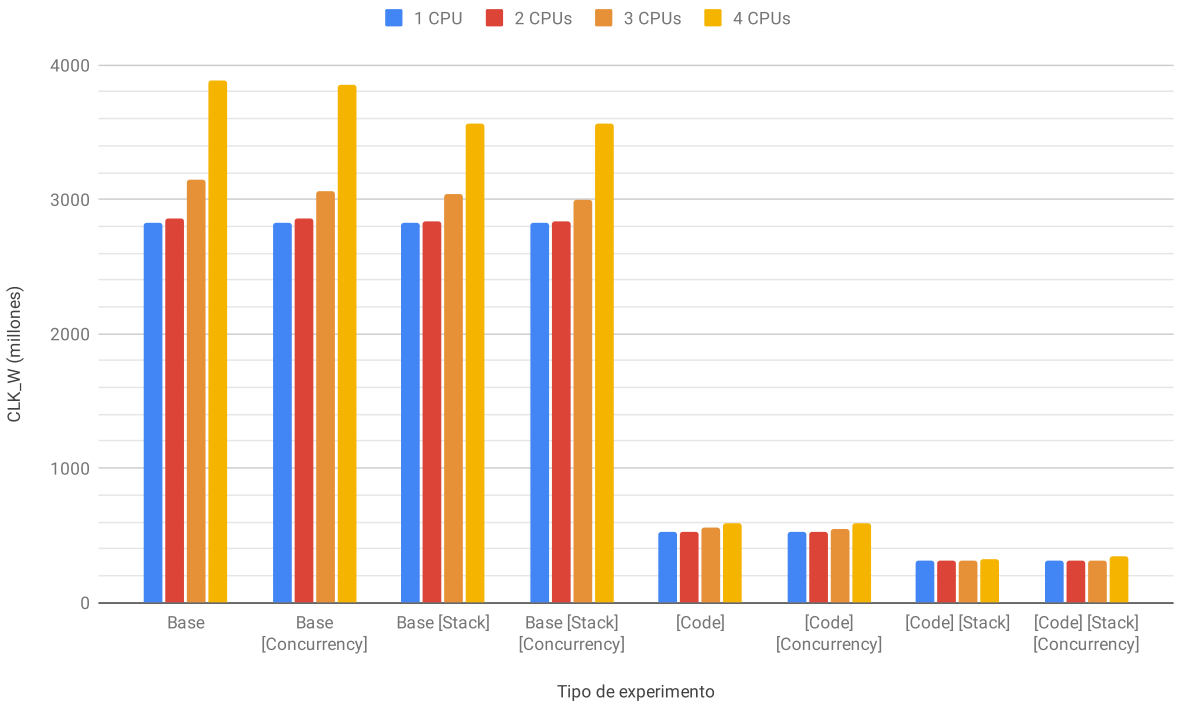


Figura 3.5: Ejecución del programa Trivialmente Paralelizable con 32K de WU (*Working Units*) y 512 Joins. Muestra la variación de tiempo por CPUs (barras de colores) para cada uso de BRAM.

Denotaremos los distintos usos de BRAM en la parte inferior de los gráfico para las siguientes figuras (por ejemplo, la figura 3.5).

> Indicamos **[Concurrency]**, **[Stack]** y **[Code]** cuando se utiliza BRAM para ese caso. De no estar mencionado el uso específico, éste no se encuentra activo. Por ejemplo, **Base [Stack]** indicaría que solo el *Stack* utilizaría BRAM, mientras que **[Code] [Stack]** indica que código y *stack* ambos usan BRAM.

El resto de los experimentos también hará uso de esta misma nomenclatura.

En este experimento, cada vez que se incrementa el numero de CPUs lo que se agrega no es solo el procesador, sino también una cantidad de trabajo para ese procesador. Si bien se mantiene la cantidad de trabajo para cada CPU, la total aumenta.

Observando la figura 3.5 y comparando el caso base, puede verse que para una y dos CPUs se dan resultados muy similares (dos CPUs tarda 1 % más). Agregar una CPU extra (tres CPUs) provoca un aumento del 10,41 % (11,52 % comparando con una CPU). Pero utilizar el máximo (cuatro CPUs) genera un incremento significativo del 23,30 % (37,51 % comparando con una CPU).

Este incremento puede explicarse por el uso común de los recursos del `Sistema`. Para dar un ejemplo, la cantidad de buses de memoria es un recurso limitado y a medida que se aumenta la cantidad de componentes que la utiliza se puede llegar a un cuello de botella.

De la misma manera, la L2 puede requerir desalojar ciertas líneas de *cache* ante algún pedido de acceso a memoria.

Variación por cantidad de Joins

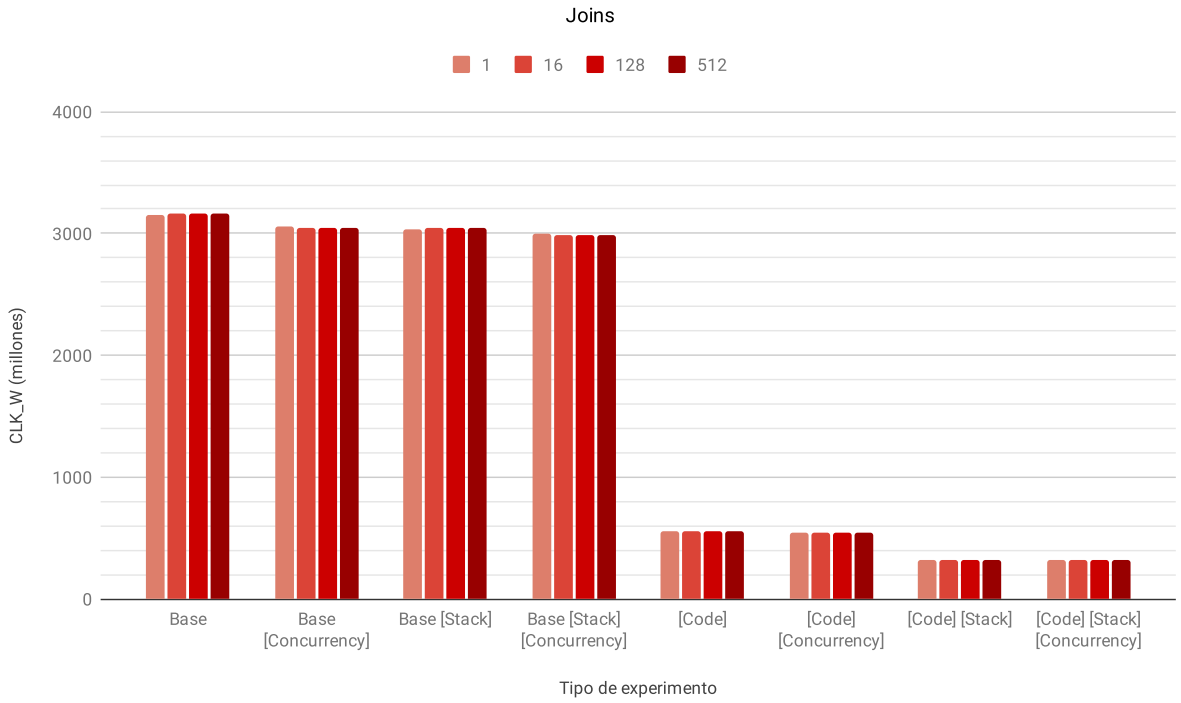


Figura 3.6: Ejecución del programa Trivialmente Paralelizable con 32K de WU (*Working Units*) sobre tres CPUs. Muestra la variación de tiempo por Join (barras de colores) para cada uso de BRAM.

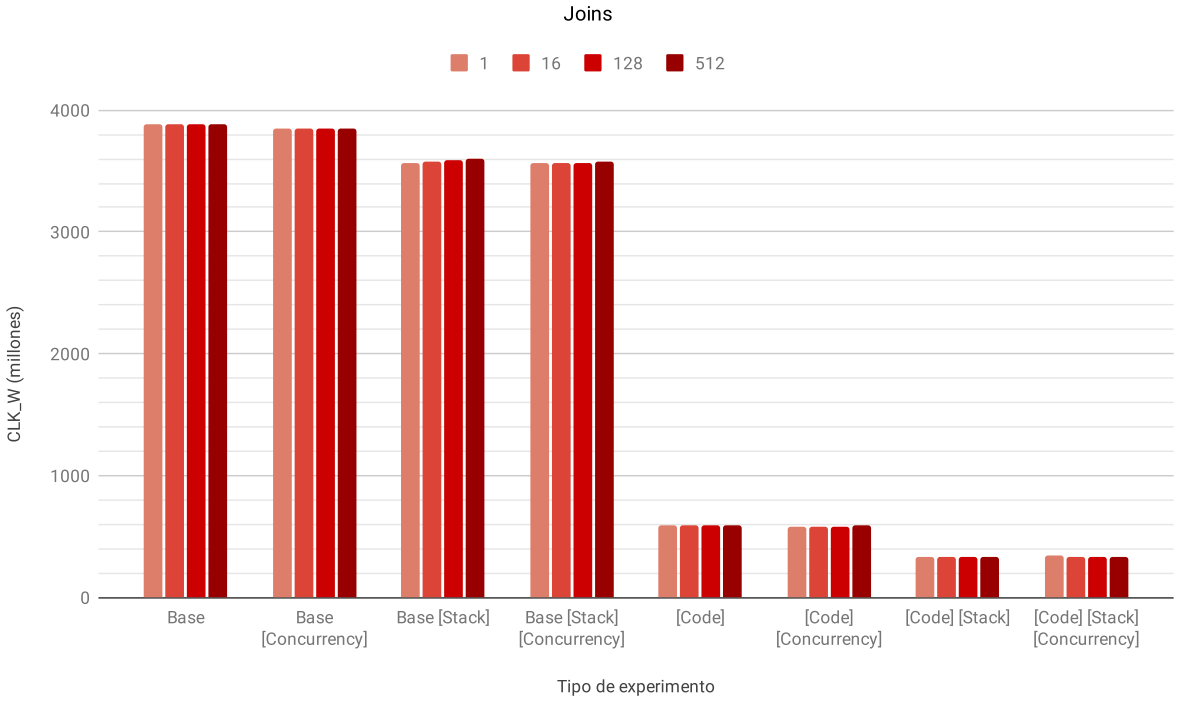


Figura 3.7: Ejecución del programa Trivialmente Paralelizable con 32K de WU (*Working Units*) sobre cuatro CPUs. Muestra la variación de tiempo por Join (barras de colores) para cada uso de BRAM.

Otra observación la podemos hacer sobre la variación que provoca la sincronización, es decir, el uso de *wait* y *joins* como barreras. Para una y dos CPUs, los resultados son muy similares. Más aún, la cantidad de sincronizaciones (*joins*) parecen no afectar su desempeño.

En las figuras 3.6 y 3.7 ilustramos los resultados para tres y cuatro CPUs. Para el primer caso, la diferencia entre comparar un *join* con 16, 128 y 512 es menor al 1,2 %. El caso de cuatro CPUs, ésta se mantiene por debajo del 1 %, con excepción del caso donde están en uso todas las aplicaciones de la técnica (usa/se sube todo a BRAM), donde un *join* resulta 5 % peor que 16, 128 o 512.

Respecto al desvío estándar, para tres CPUs éste se encuentra por debajo del 0,10 % en general; 0,26 % para un *join* con todas las aplicaciones de la técnica y 128 *joins* con sólo subir el código a BRAM; 1,02 % para un *join* con subir el código y *stack*.

Para cuatro CPUs hay más variedad: $\leq$ 0,07 % para variantes con *stack* en memoria principal; de 0,34 % a 0,51 % para un *join* con *stack* en BRAM sobre el caso base pero $\leq$ 0,17 % para los otros *joins*; 1,26 % para un *join* al subir código y *stack* y 5,94 % agregándole concurrencia; de 0,13 % a 0,57 % para 16 *joins* con código y *stack*; $\leq$ 0,06 % para el resto.

A menor cantidad de *joins*, menor es la varianza. A mayor cantidad de CPUs, ésta parece hacerse mas prominente. El caso más llamativo es el de un *join* para cuatro CPUs con todas las aplicaciones de la técnica, que es justamente el caso que resultó peor en la comparación.

Se puede observar que el tiempo que demora en sincronizar CPUs es muy chico. Esto se puede explicar por el procesamiento regular que ocurre para cada CPU (todos hacen correr el mismo algoritmo, al cual se le pasa la misma entrada).

Variante de rotación de CPU principal

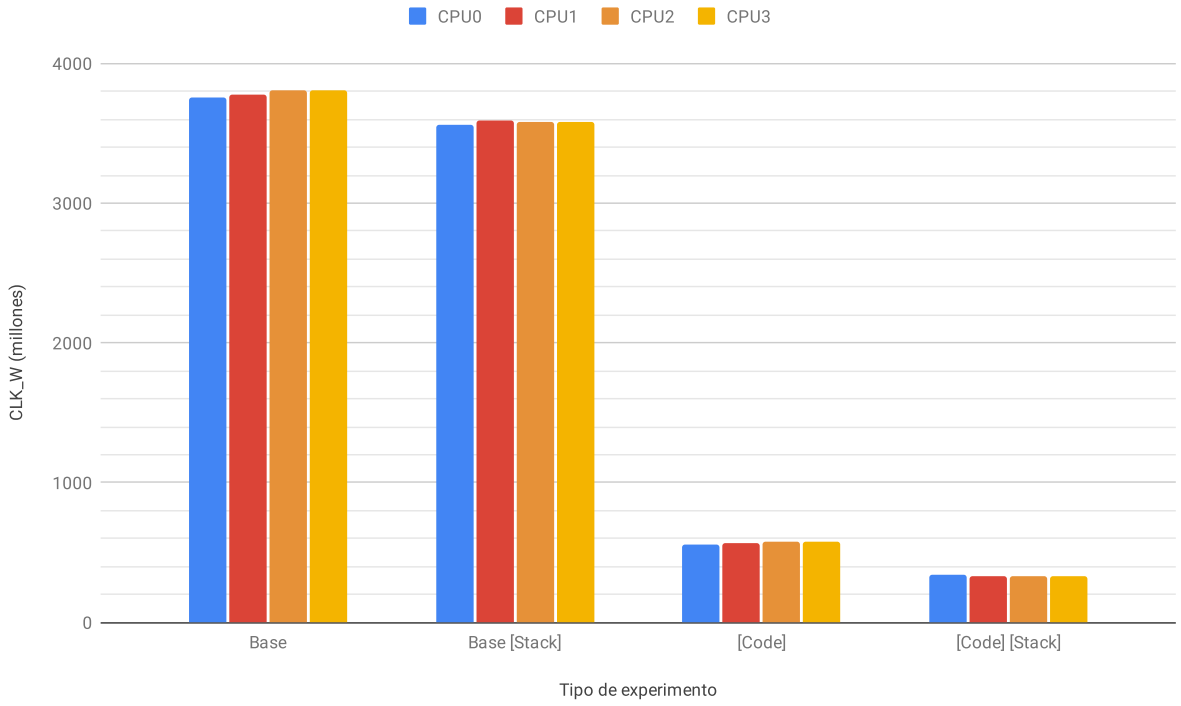


Figura 3.8: Ejecución del programa Trivialmente Paralelizable con 32K de WU (*Working Units*) sobre cuatro CPUs. Muestra la variación de tiempo según el CPU que se utilizó para medir (barras de colores) para cada aplicación (exceptuando concurrencia).

En una variación del experimento, uno de los CPUs es marcado como el principal y se registra el tiempo que demora en realizar 32K de WU, luego del cual queda esperando a que los demás terminen. Al contrario que los experimentos anteriores con al menos una sincronización o *join*, esta medición del tiempo no posee ninguna luego de que hayan empezado a correr.

El resto de los CPUs realizan el doble de trabajo (64K) para asegurar que continúen trabajando cuando el CPU principal finaliza. Al no haber sincronización, la aplicación de concurrencia fue ignorada. En la figura 3.8 podemos observar los resultados.

Este experimento nos permite ver que hay un cierto desvío entre los CPUs para el caso base. Comparando contra CPU0, se trata de un incremento en los tiempos del 0,58 % para CPU1, 1,58 % para CPU2, y 3,15 % para CPU3. Este incremento a medida que sube el id del CPU parece indicar una cierta priorización, la cual puede explicarse si observamos la figura 3.9 (que indica el orden de conexión de los puertos de la L2, siendo los primeros datos e instrucciones del CPU0, luego de CPU1, etc).

Es interesante ver que esto se repite para la técnica aplicada al código únicamente, (siendo los incrementos del 1,62 %, 3,30 % y 5,26 %).

Teniendo solo el *stack* en BRAM tenemos incrementos del 0,70 % (CPU1), 0,56 % (CPU2) y 0,84 % (CPU3).

El caso más interesante es donde ambos están en BRAM. Si bien tenemos un incremento del 0,10 % (CPU1), registramos mejoras del 0,96 % (CPU2) y 1,84 % (CPU3) por sobre el tiempo registrado para CPU0.

Tener el *stack* en BRAM, sin embargo muestra un desvío estándar no despreciable del 0,72 % (CPU0), 0,66 % (CPU1), 0,72 % (CPU2) y 1,19 % (CPU3) sobre el caso base. Con subir código y *stack*, éste es del 5,13 % (CPU0), 4,39 % (CPU1), 1,15 % (CPU2) y 2,76 % (CPU3). Por último, con código en BRAM y *stack* en memoria principal, el desvío estándar es $\leq$ 0,01 %.

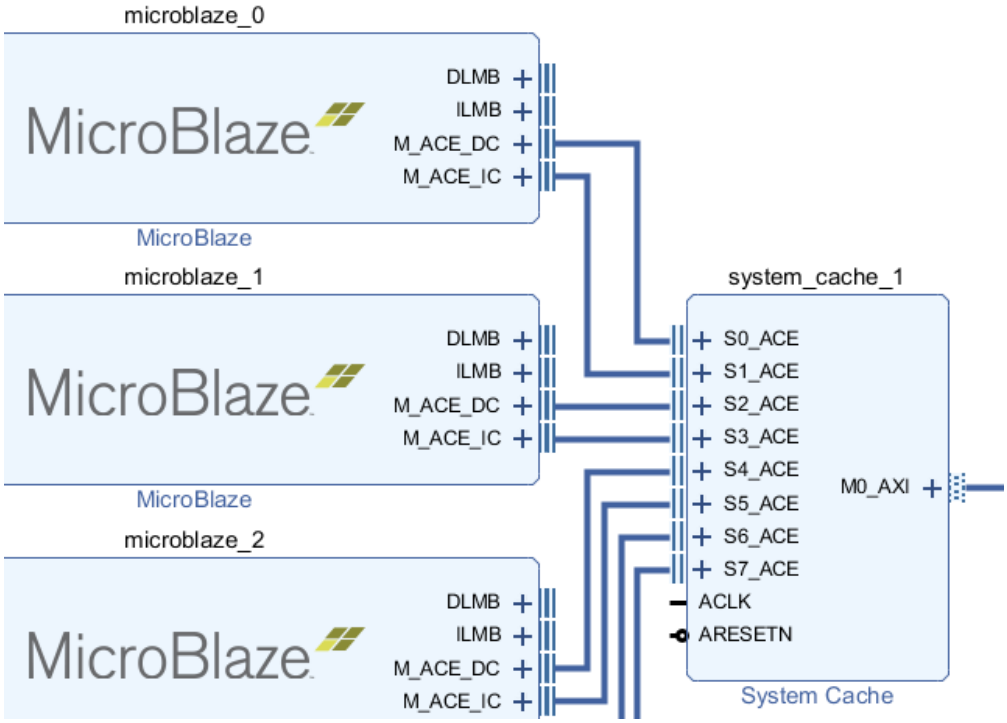


Figura 3.9: Conexión entre `MicroBlaze` y *cache* L2. Se omiten las otras conexiones para simplificar la visualización.

Ya sea se aplique alguna o ninguna de las técnicas, la diferencia de desempeño entre los CPUs parece ser menor al 4 %.

Al diseñar cualquier proyecto de hardware/software hay que tomar ciertas decisiones. En el caso de un bloque de hardware como una *cache* con varios puertos de entrada, una de ellas es qué hacer cuando se reciben pedidos de acceso desde varios de estos puertos. Si queremos una distribución equitativa, esto probablemente requiera más componentes y un mayor trabajo de diseño que si atendemos los pedidos en el orden de los puertos.

Es posible que la *cache* de L2 se haya implementado utilizando un procesamiento prioritario sobre los puertos. De ser este el caso, se explicaría por qué un programa enteramente en memoria principal termine más rápido cuando se hace la medición sobre el CPU con menor número de puerto.

El hardware sintetizado para la L2 (provisto por Xilnix) indica en su especificación [20] que el esquema de elección hace *round-robin* sobre los puertos a la hora de priorizar los accesos a memoria. Si el puerto elegido no tiene una transacción pendiente, se usará el primer puerto que la tenga (en el orden de conexión).

Mover código

En todas las figuras puede verse que la aplicación de la técnica sobre código es la que más diferencia genera comparando con el caso base. El uso de BRAM sobre el LMB explicaría la diferencia de desempeño cuando se utiliza esta variante.

Observando la figura 3.5 y comparando con el caso base, la técnica aplicada sobre el código genera una mejora del 81,5 % para uno y dos CPUs, 82,7 % para tres CPUs y 84,8 % para cuatro CPUs. En otras palabras, reduce los tiempos de un 15 % a 18 % de su valor usando memoria principal.

Considerando la substancial mejora de desempeño lograda por esta aplicación de la técnica, optamos por no compararla con las otras en esta sección. En cambio, hablaremos del resto en función de si la incluyen o no.

En este experimento se acerca lo más prácticamente posible el código a la CPU. Esto es similar a lo que hace la *cache*, pero nos aprovechamos de poder diseñar el hardware para proveer un acceso con mejor desempeño. La mejora de tiempos es significativa, más aún considerando que tenemos dos *caches* en uso.

Cualquier tipo de algoritmo que no sea muy extenso (que entre en la BRAM) podría aprovecharse de esta técnica.

Mover stack

Observando la figura 3.5, direccionar el *stack* a BRAM sin aplicar la técnica de código no mejora significativamente los tiempos con uno o dos CPUs (menor a 0,7 %). Presentando una reducción para tres y cuatro CPUs (3,62 % y 8,17 % respectivamente).

Sin embargo, el incremento en el desempeño es significativo cuando se usa en conjunto con la técnica aplicada al código. Allí se consigue una mejora del 40 % para un CPU, 40,89 % para dos CPUs, 43,16 % para tres CPUs y 44,5 % para cuatro CPUs (comparando con subir el código a BRAM y dejar el *stack* en memoria principal). Es decir, reduce los tiempos de 55 % a 60 % de su valor con solo almacenar el código en BRAM.

Nuevamente notamos una mejora sobre la *performance* al mover parte de la memoria en uso a la BRAM. Si bien no es tan significativa como cuando se usa en conjunto con la técnica aplicada al código, no es despreciable cuando hay tres o cuatro CPUs en el `Sistema`.

Esta discrepancia nos da una pauta de que el código es mucho más utilizado que los datos en este programa. Los algoritmos cortos que hacen procesamiento pesado sobre memoria local podrían aprovecharse de esta aplicación de la técnica (cuando los registros del CPU son insuficientes).

Mover Spin-locks

Esta aplicación de la técnica debería mejorar los casos donde haya barreras sobre procesos dispares. En este programa, el procesamiento que hace cada CPU es muy parejo. Esto puede explicar la diferencia poco significativa que se ve al mover los *spin-locks* a BRAM.

Más adelante mostraremos un experimento donde la aplicación muestra una mejora significativa. Resumiremos los resultados en la tabla 3.1:

Aplicaciones	Mejora con Concurrencia		
	2 CPU	3 CPUs	4 CPUs
Ninguna	-0,58 %	3,42 %	0,92 %
Stack	0,01 %	1,72 %	0,72 %
Código	-0,18 %	1,69 %	0,97 %
Código+Stack	0,04 %	0,76 %	-0,06 %

Tabla 3.1: Mejora de tiempos en porcentaje entre las distintas aplicaciones de la técnica al añadirles la subida de *spin-locks* a BRAM.

Al contrario que las otras aplicaciones de la técnica, la de concurrencia parece tener pocos casos donde incurre una diferencia no despreciable. Como se trata de un tipo de traslado de código mucho más específico, tiene sentido que tenga una utilidad reducida.

Para pocos procesadores, se nota un impacto negativo y en los casos donde es beneficioso se ve disminuido cuando se usa en conjunto con las otras aplicaciones de la técnica.

3.2.3. Conclusión

En esta sección tratamos con un problema "Trivialmente Paralelizable", donde un programa es paralelizado de manera sencilla y permitiéndonos interponer barreras que sincronizan los procesadores.

Como era esperado, agregar procesadores al `Sistema` sin reducir la carga de los otros *softcores* aumenta los tiempos de ejecución. La diferencia entre uno y dos CPUs es pequeña pero a partir de tres se hace significativa. Esto sería un indicador de que puede haber un cuello de botella al compartir la memoria con tres o más procesadores.

Algo a destacar es que hay un cierto desvío entre los procesadores. Esto puede explicarse por una priorización de los puertos en la L2.

Respecto a la aplicación de la técnica, es evidente que genera mejoras en varias instancias. Si bien quizás mover el código sea lo más performante, en programas más grandes habrá que elegir qué partes subir a BRAM, o ir alternándolas de ser posible.

Apuntar el *stack* a BRAM es más sencillo de implementar, mejora levemente los tiempos y no requiere modificar el programa en sí (solo el *bootloader*). El impacto es mayor cuando se usa en conjunto con la subida de código, en cuyo caso habrá que tener cuidado con el espacio que vayan a ocupar ambos en BRAM.

Mover los *spin locks* a BRAM, en cambio, no parece generar resultados significativos. Esto puede deberse a que la carga a sincronizar fue homogénea.

3.3. Multiplicación de matrices

La multiplicación de matrices es una de las operaciones más comunes en el cálculo científico. Existen distintos algoritmos que resuelven el problema de forma eficiente basándose en las características de la operación. Incluso, algunas soluciones se limitan a optimizar la operación para una familia particular de matrices con propiedades dadas.

En este trabajo no utilizamos un algoritmo eficiente u optimizado para un caso particular. Decidimos utilizar un algoritmo con un patrón de acceso simple (algoritmo 10) que nos permita estudiar más fácilmente cómo impacta el uso de BRAM. También nos limitaremos a matrices enteras (en vez de números de punto flotante o fijo), y cuadradas de dimensión potencia de dos. Esto último nos permite distribuir con facilidad el trabajo entre múltiples *cores*.

Partiendo de una implementación directa de la definición, la paralelizaremos haciendo uso de una técnica simple de particionado sobre la entrada.

```
void multiply(A, B, O, N) {
  elements = N * N;
  for (a_r = 0; a_r < elements; a_r += N) {
    for (b_c = 0; b_c < N; b_c++) {
      sum = 0;
      for (a_c = 0, b_r = 0; a_c < N; a_c++, b_r += N) {
        sum += A[a_r + a_c] * B[b_r + b_c];
      }

      i = a_r + b_c;
      O[i] = sum;
    }
  }
}
```

Algoritmo 10: Implementación directa de multiplicación de matrices A * B

```
void multiply(A, B, O, N) {
  elements = N * N;
  for (a_r = 0; a_r < elements; a_r += N) {
    for (a_c = 0, b_r = 0; a_c < N; a_c ++, b_r += N) {
      sum = 0;
      for (p = 0; p < N; p++) {
        sum += A[a_r + p] * B[b_r + p];
      }

      i = a_r + a_c;
      O[i] = sum;
    }
  }
}
```

Algoritmo 11: Implementación directa de multiplicación de matrices A * Bt

En una segunda versión del algoritmo calculamos $A * B^t$ (algoritmo 11). Esta es similar a la anterior, modificando el patrón de acceso a la memoria de B.

El primer algoritmo recorre B por columna, lo cual puede derivar en tiempos mayores dada la distancia entre los datos. El segundo recorre B por fila, permitiendo aprovechar el principio de localidad que ofrece la *cache*.

3.3.1. Técnica de paralelizado

Teniendo en cuenta que la multiplicación de matrices A y B se puede hacer multiplicando filas de A por columnas de B, podemos partir la entrada de forma que cada procesador utilice una fila distinta, intercalándose entre sí. Teniendo K procesadores disponibles, cada fila $F \in \{0, \dots N-1\}$ de la matriz A será recorrida por el CPU con $cpuID = F \ mod \ K$, $(cpuID \in [0, K-1])$.

Elegimos esta forma por su simpleza ante el análisis en comparación con una multiplicación por bloques.

```
void multiply(A, B, O, N , cpuID, K ) {
  elements = N * N;
  for (a_r =  cpuID * N ; a_r < elements; a_r +=  K * N ) {
    for (b_c = 0; b_c < N; b_c++) {
      sum = 0;
      for (a_c = 0, b_r = 0; a_c < N; a_c++, b_r += N) {
        sum += A[a_r + a_c] * B[b_r + b_c];
      }

      i = a_r + b_c;
      O[i] = sum;
    }
  }
}
```

Algoritmo 12: Multiplicación de matrices A*B paralelizada

```
void multiply(A, B, O, N , cpuID, K ) {
  elements = N * N;
  for (a_r =  cpuID * N ; a_r < elements; a_r +=  K * N ) {
    for (a_c = 0, b_r = 0; a_c < N; a_c ++, b_r += N) {
      sum = 0;
      for (p = 0; p < N; p++) {
        sum += A[a_r + p] * B[b_r + p];
      }

      i = a_r + a_c;
      O[i] = sum;
    }
  }
}
```

Algoritmo 13: Multiplicación de matrices A*Bt paralelizada

Analicemos el uso de memoria: por ejemplo, para cuatro CPUs, ambas variantes tendrán que acceder a cuatro filas distintas de la matriz A. Cada uno de estos accesos se asocia a una

línea de *cache* distinta para matrices a partir de 16×16 (las líneas de *cache* son de 64 B, y cada entero ocupa 4 B).

Como el caso no transpuesto debe recorrer una columna de B, debe hacer cuatro accesos a líneas distintas para poder multiplicar cuatro celdas, y 16 para multiplicar 16 celdas. En el mejor de los casos, estos accesos se comparten entre los CPUs, y tenemos un total de $4 + 16 = 20$ accesos a líneas distintas cada 16 elementos.

El caso transpuesto, en cambio, al recorrer B por filas, solo requiere un acceso adicional cada 16 elementos, logrando un total de $4+1 = 5$ accesos a líneas distintas cada 16 elementos.

En ambos casos, el patrón de escritura a la matriz resultante es el mismo.

Habrá una única barrera de sincronización luego del cálculo para finalizar el algoritmo cuando todos los procesadores hayan procesado la entrada. Los cambios para ambas variantes están ilustrados en los algoritmos 12 y 13, respectivamente.

3.3.2. Experimentos

Los valores de cada elemento en la matriz son generados al azar entre 1 y 3 inclusive. Las matrices son de $N \times N$, con $N \in \{4, 32, 128, 256, 512\}$.

Éste se trata de un algoritmo invariante a la entrada. En otras palabras, dadas matrices de entrada de tamaño $N \times N$, se ejecutarán la misma cantidad de instrucciones sin importar cuáles son los valores que contengan. Se espera que el tiempo de ejecución sea del orden de $O(N^3)$.

Dado que la paralelización es una partición de la entrada, esperamos que el *speed-up* a medida que se agregan procesadores sea próximo al máximo teórico. A pesar de tener la misma complejidad teórica, también esperamos observar diferencias entre la versión $A * B$ y $A * B^t$, obteniendo la segunda menores tiempos de ejecución.

3.3.3. Resultados

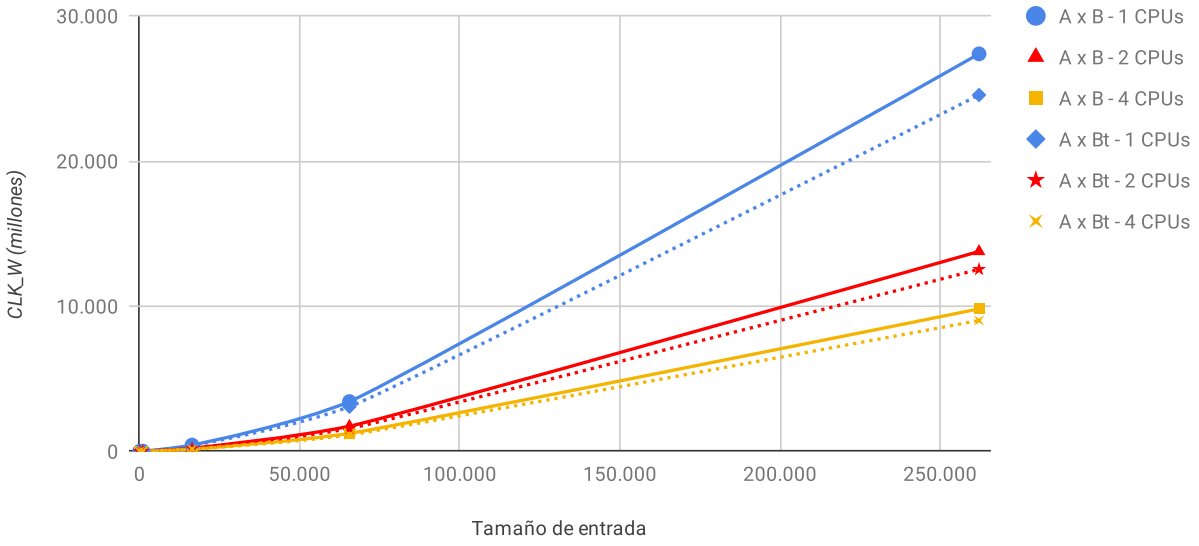


Figura 3.10: Ejecución del programa que multiplica matrices sobre uno, dos y cuatro procesadores sin utilizar BRAM. Incluye ambas variantes y muestra la variación de tiempo por procesador en función de la cantidad de elementos en las matrices.

Observando la figura 3.10 podemos ver que la variante transpuesta tiene mejores tiempos que su contraparte. También podemos ver que a mayor cantidad de CPUs, menor tiempo total.

Como los márgenes de los resultados difieren poco entre las distintas entradas, elegimos analizar un caso intermedio, con matrices de 128×128 (16,384 elementos). Para este caso, la mejora que presenta la variante $A * B^t$ con respecto a la version $A * B$ es de aproximadamente de 9 % a 10 % para uno, dos y cuatro procesadores.

> Calcularemos el *speed-up* al aumentar la cantidad de procesadores **en base a la misma configuración** con uno solo.
> Por ejemplo, para el caso de dos CPUs y una configuración donde solo se mueve el *stack* a BRAM, el *speed-up* será el cociente entre mover el *stack* a BRAM utilizando dos CPUs y mover el *stack* a BRAM utilizando un procesador.

Para dos CPUs el *speed-up* es de alrededor de 1,98 para $A * B$ y de 1,96 para $A * B^t$ (recordemos que el teórico es de 2). Para cuatro CPUs, éste es de 2,78 para $A * B$ y de 2,74 para $A * B^t$ (teórico de 4).

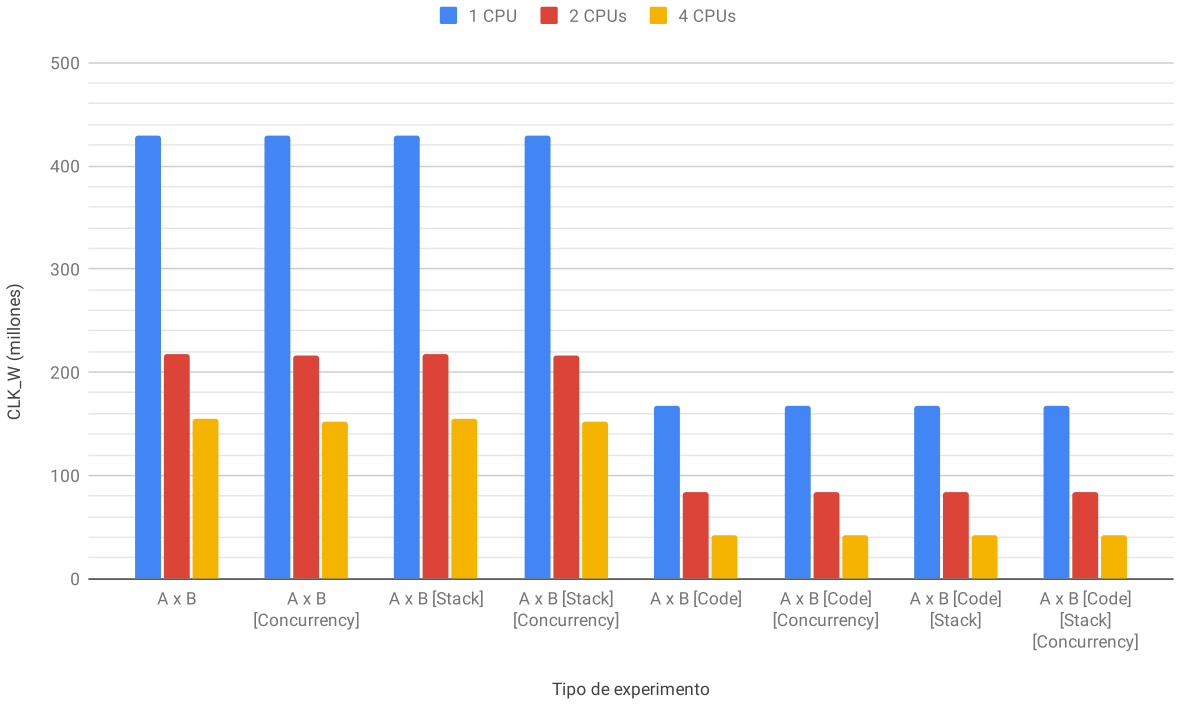


Figura 3.11: Ejecución del programa que multiplica matrices para matrices de 128×128 sobre uno, dos y cuatro procesadores. Incluye la variante $A * B$ y muestra la variación de tiempo por CPUs (barras de colores) para cada uso de BRAM.

En la figura 3.11 se comparan los usos de BRAM aplicados a la variante $A * B$. Se muestra para una entrada de 16,384 elementos (matrices de 128×128).

El desvío estándar obtenido para todos los casos es menor o igual al 0,28 %.

De la misma forma, en la figura 3.12 se comparan los usos de BRAM aplicados a la variante $A * B^t$. El desvío estándar obtenido para todos los casos es menor o igual al 0,30 %.

Algo a destacar es que mover el *stack* a BRAM no provoca diferencias significativas. Los tiempos variaron un máximo del 0,3 % para el caso $A * B$ y 0,25 % para $A * B^t$.

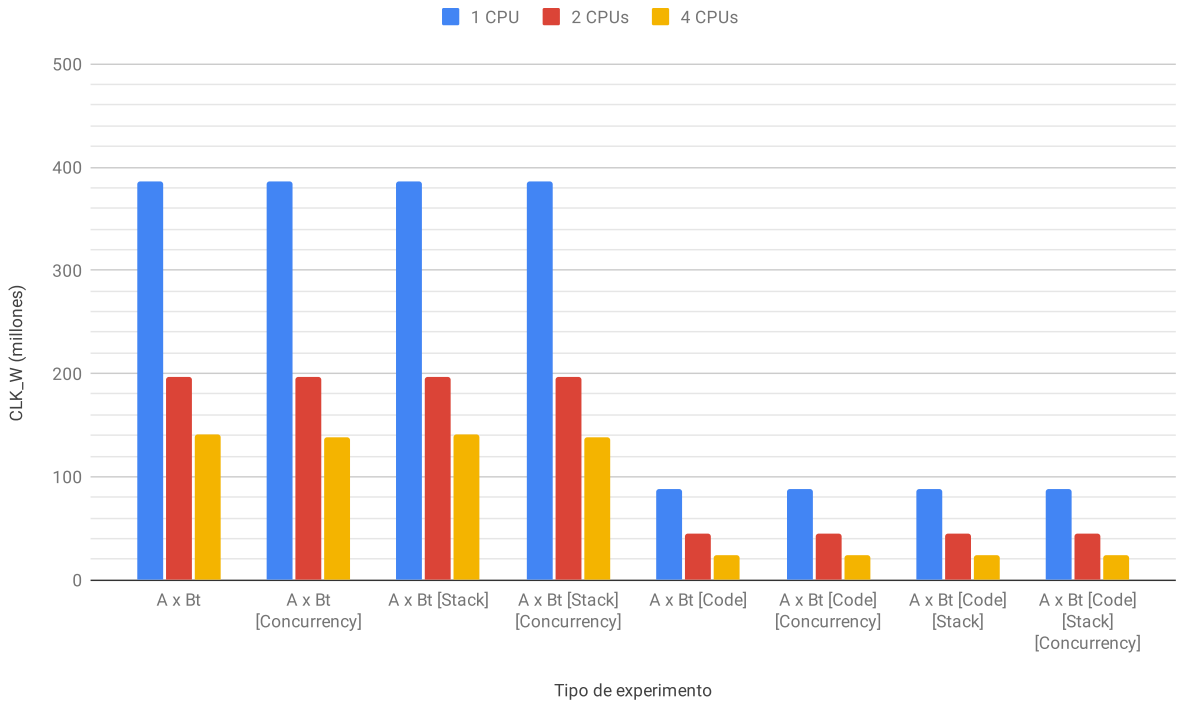


Figura 3.12: Ejecución del programa que multiplica matrices para matrices de 128×128 sobre uno, dos y cuatro procesadores. Incluye la variante $A * B^t$ y muestra la variación de tiempo por CPUs (barras de colores) para cada uso de BRAM.

Esto se puede explicar por la poca interacción que hay con el *stack*. Al tratarse de un algoritmo iterativo con pocas variables, éstas pueden almacenarse en los registros del CPU. Este es un ejemplo de un algoritmo gobernado por el acceso a la memoria.

Mover el código a BRAM, en cambio, genera mejoras del 61,1 % (una CPU), 61,6 % (dos CPUs) y 82,5 % (cuatro CPUs) para la variante $A * B$, y del 77,2 % (una CPU), 77,6 % (dos CPUs) y 85,0 % (cuatro CPUs) para $A * B^t$.

Parece que subir el código provoca un impacto más substancial cuando los datos se acceden aprovechando el principio de localidad.

Subir las funciones de concurrencia no genera diferencias significativas para uno o dos procesadores para ambas variantes (menor al 0,5 %). Para cuatro CPUs, de no subirse el código e independientemente de lo que se haga con el *stack*, hay una mejora del 1,7 % para la variante $A * B$, y del 2,0 % para la variante $A * B^t$. Al subirse el código e independientemente del *stack*, los tiempos empeoran un 0,2 %.

A pesar de tener una única barrera de sincronización, y a pesar de tener un tipo de procesamiento y una cantidad de trabajo similar entre los procesadores, esto parece hacer cuello de botella a medida que hay más involucrados.

Por otra parte, la variante $A * B^t$ presenta tiempos un 10,2 % mejor que su contraparte para un procesador, 9,5 % para dos CPUs y 9,3 % para cuatro CPUs. Subiendo el código a BRAM, ésta mejora es del 47,4 % para un CPU, 47,2 % para dos CPUs y 44,0 % para cuatro CPUs.

Esto nos indica que el acceso por columnas provoca una carga significativa sobre el `Sistema`. El hecho de que la diferencia se acrecente cuando el código está en BRAM da una pauta de que es uno de los puntos débiles del algoritmo.

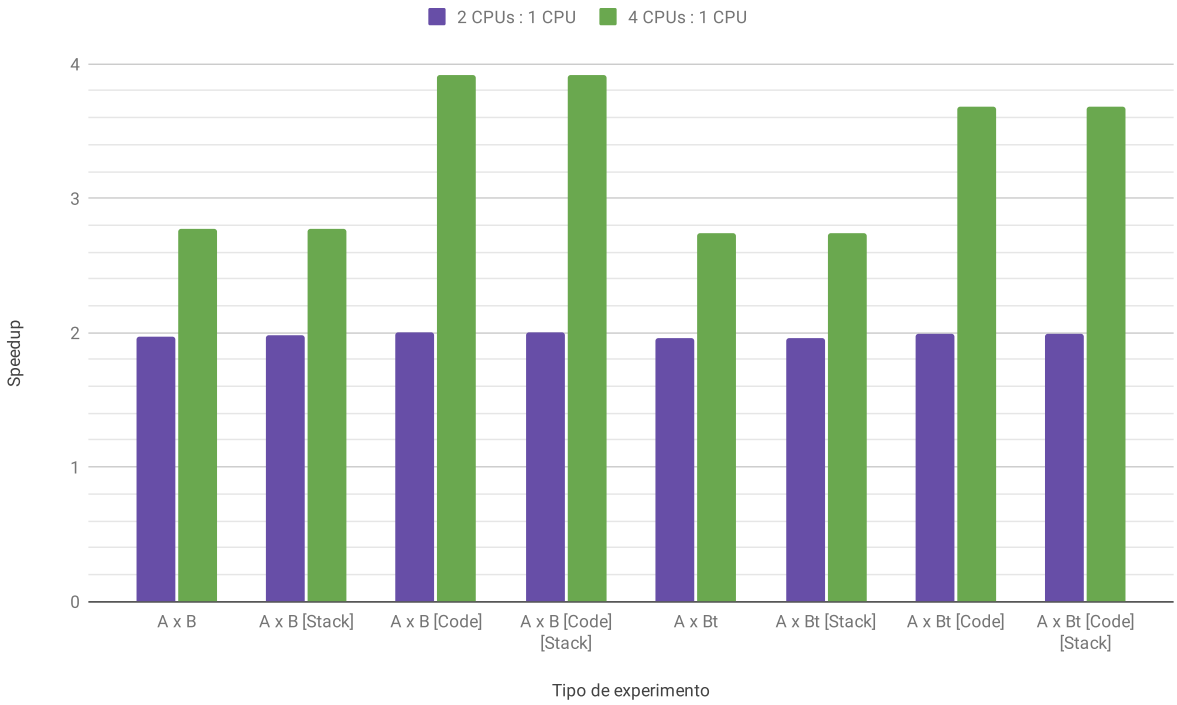


Figura 3.13: *Speed-up* registrado para dos y cuatro CPUs del programa que multiplica matrices para matrices de 128×128. Incluye la variante $A * B^t$ y muestra el *speed-up* (barras de colores) para cada uso de BRAM.

Finalmente, en la figura 3.13, observamos un *speed-up* de 1,98 $(A * B)$ y 1,96 $(A * B^t)$ para dos CPUs. Al subir el código, el *speed-up* pasa a ser de 2 $(A * B)$ y 1,99 $(A * B^t)$. Al tener todo en BRAM, el *speed-up* se mantiene igual.

Para cuatro CPUs, éste es del 2,77 $(A * B)$ y 2,74 $(A * B^t)$ sin subir el código, y del 3,92 $(A * B)$ y 3,68 $(A * B^t)$ subiéndolo. Al tener todo en BRAM, el *speed-up* pasa a ser del 3,92 $(A * B)$ y 3,67 $(A * B^t)$.

Usando dos procesadores, parece que el algoritmo se desempeña casi tan bien como se espera teóricamente. Con cuatro CPUs, la mejora es un 70 % más que con dos CPUs si el código continúa en memoria principal, y se incrementa de un 260 % a 290 % más subiéndolo a BRAM.

Esto nos da la pauta de que, con cuatro procesadores, se reduce bastante la competencia por los recursos utilizados al desplazar el código fuera de memoria principal.

3.3.4. Conclusión

Un punto a destacar es que hacer uso de la memoria aprovechando el principio de localidad puede provocar una mejora substancial de los tiempos. Esto depende del algoritmo y a veces no es posible.

Mover el *stack* no tuvo un impacto significativo y subir las funciones de concurrencia generó una leve mejora en los tiempos con cuatro procesadores solo cuando el código permanece en memoria principal.

Por otro lado, las mejoras obtenidas al subir el código indican que este último sigue siendo uno de los puntos más importantes de un programa. Si bien las *caches* tratan de reducir el impacto, evidentemente tener una memoria dedicada es una solución óptima.

Hay muchas formas mucho más eficientes de multiplicar matrices que la utilizada en esta sección, ya sea en bloque o aprovechándose de información sobre las matrices (si son esparzas, simétricas, etc). Implementamos una multiplicación *naive* para poder analizar los resultados con mayor facilidad, así como para no perder foco en el punto principal del trabajo, que es aplicar las técnicas mencionadas para aprovechar la existencia de la BRAM.

3.4. Heapsort

Heapsort es un algoritmo de *in-place sorting* basado en comparaciones que tiene complejidad $O(N\ log(N))$. El algoritmo se basa en las propiedades de una estructura de *heap*.

```
void heapify(arr[], n, i) {
    int largest = i;
    int l = 2*i + 1;        // nodo izquierdo
    int r = 2*i + 2;        // nodo derecho

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    // chequea la propiedad de heap
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapsort(arr[], int n) {
    // transformar a heap
    for (i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (i = n - 1; i >= 0; i--)
    {
        // extraer el mayor del tope, dejarlo al final
        swap(arr[0], arr[i]);

        // rearmar el heap
        heapify(arr, i, 0);
    }
}
```

Algoritmo 14: *Heapsort*

Un *heap* es un árbol casi completo donde el valor de cada nodo padre es **mayor** que el de sus hijos (*max-heap*).

Utilizando esta propiedad, se puede extraer el nodo raíz, volver a reorganizar el *input* restante en un *heap*, y repetir hasta obtener el valor final. Con esta idea se implementa el algoritmo de *heapsort*.

Construir el *heap* a partir de un arreglo tiene complejidad $O(N)$, siendo N la cantidad de elementos en la entrada. Si bien leer el valor de la raíz tiene complejidad constante, extraerlo y reorganizar los nodos restantes en un *heap* tiene complejidad $O(\log(N-1))$.

Debiendo extraer los N valores del arreglo original, queda que la complejidad de *heapsort* es entonces $O(N \log(N))$.

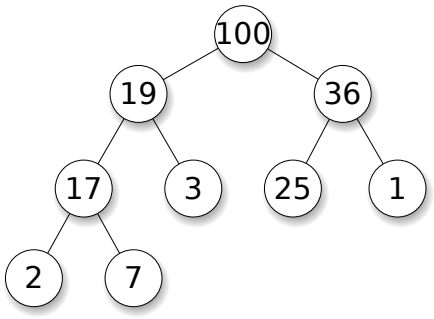


Figura 3.14: Un *max-heap*.

```
void merge(int vector[], int buffer[], int start, int mid, int end, int sortNElements) {
  int l = start;
  int r = mid;
  int i = 0;
  int stopAt = i + sortNElements;

  while (i < stopAt && l < mid && r < end) {
    if (vector[l] <= vector[r]) {
      buffer[i ++] = vector[l ++];
    } else {
      buffer[i ++] = vector[r ++];
    }
  }

  while (i < stopAt && l < mid)   buffer[i ++] = vector[l ++];
  while (i < stopAt && r < end)   buffer[i ++] = vector[r ++];
}

void reverseMerge(int vector[], int buffer[], int start, int mid, int end, int
↪  sortNElements) {
  int l = mid - 1;
  int r = end - 1;
  int i = (end - start) - 1;
  int stopAt = i - sortNElements;

  while (i >= stopAt && l >= start && r >= mid) {
    if (vector[l] > vector[r]) {
      buffer[i --] = vector[l --];
    } else {
      buffer[i --] = vector[r --];
    }
  }

  while (i >= stopAt && l >= start)   buffer[i --] = vector[l --];
  while (i >= stopAt && r >= mid)   buffer[i --] = vector[r --];
}
```

Algoritmo 15: Merge

3.4.1. Técnica de paralelización

Vamos a dividir *heapsort* en un *bag of tasks*, ya que el algoritmo no es naturalmente paralelizable. Si tenemos un único procesador, utilizamos el algoritmo de *heapsort* tal como es conocido. Con dos procesadores, vamos a dividir la entrada en dos, ordenar cada uno por *heapsort* y hacer un *merge* del resultado. De la misma forma, con cuatro procesadores haremos cuatro *heapsort* y un *merge* recursivo (figura 3.15).

Para aprovechar los procesadores, haremos un *merge* de dos direcciones (algoritmo 15).

En el *merge tradicional*, comparamos dos elementos de dos *inputs* ordenados I_{0_i} e I_{1_j}. Si el primero es mayor o igual lo guardamos en el *output* O_s, incrementamos i y s, y continuamos (viceversa con j si el segundo es mayor). Si llegamos al final de alguno, nos resta agregar el resto de los elementos del otro.

En el *merge* bidireccional, un CPU realiza el *merge* tradicional hasta que obtiene el elemento $s = N/2$. El otro CPU hace un *merge inverso*, comenzando por el final de ambos arreglos y completando el *output* al revés, también hasta llegar al elemento de la mitad.

Utilizando estas dos variaciones podemos implementar *Heapsort* de manera paralela. La figura 3.15 permite visualizar esto.

Destacamos el caso de cuatro procesadores: los *merges* se harán entre $O_{01} = merge(I_0, I_1)$ y $O_{23} = merge(I_2, I_3)$ utilizando todos los procesadores pero el *merge* final $O = O_{0123} = merge(O_{01}, O_{23})$ no podrá hacer uso de más de dos con esta técnica.

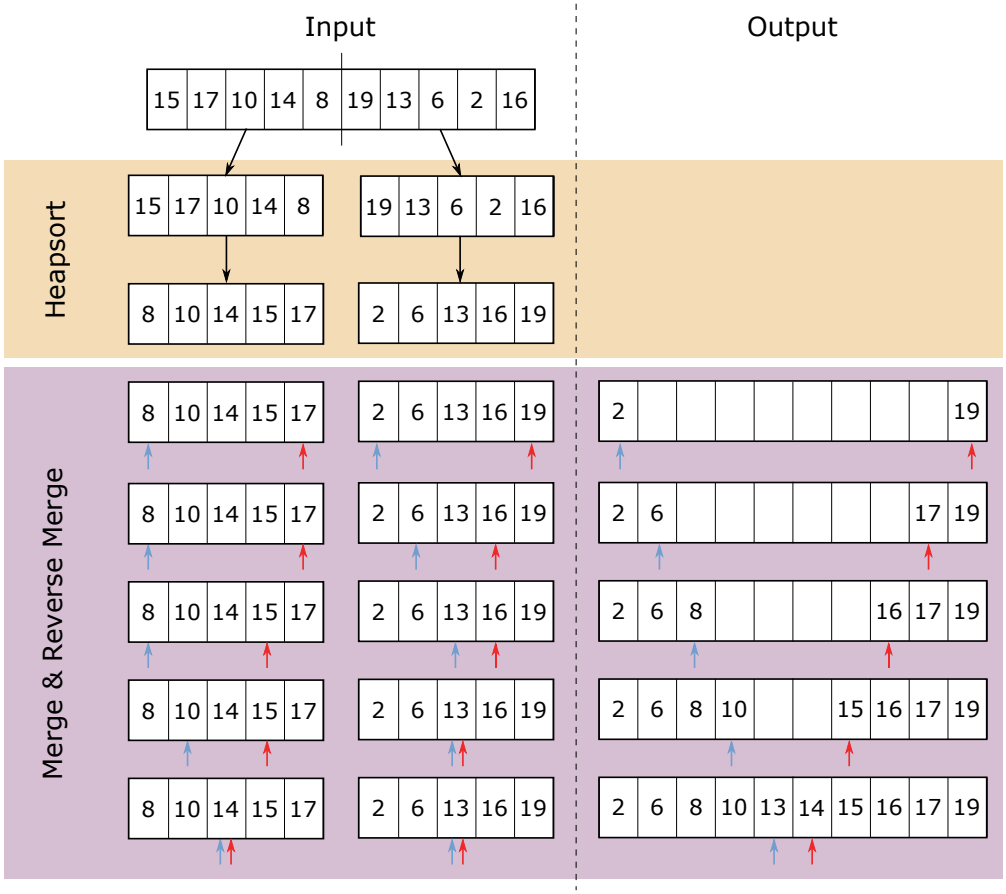


Figura 3.15: Visualización del algoritmo para dos procesadores con énfasis en los pasos del *merge* y *reverse merge* para un ejemplo de entrada de diez elementos. Las flechas azules indican los elementos indexados por el *merge* y las rojas los indexados por el *reverse merge*.

3.4.2. Experimentos

Los experimentos realizados utilizan un tamaño de entrada N tomando las siguientes opciones: 4096, 8192, 16,384, 32,768, 65,536 y 131,072. Los valores son generados al azar por el *loader* utilizando $rand(time(0))$ como semilla, obteniendo valores distintos en cada corrida.

Un factor a tener en cuenta es que *Heapsort* puede variar en tiempo según el **contenido** del *input* (más allá de su tamaño). Si bien es determinístico, el proceso de transformar el *input* en un *heap* (*heapify*) es recursivo condicionalmente (ver Algoritmo 14), lo cual implica que no todas las ramas son exploradas en su totalidad.

Esperamos ver un descenso en los tiempos de ejecución a medida que aumentamos el número de procesadores. También esperamos que el impacto de tener el *stack* en BRAM sea significativo, dada la naturaleza recursiva del algoritmo.

Notamos que tenemos pocos puntos de sincronización, y que dependen de la cantidad de procesadores disponible. Sin embargo, la carga no homogénea para cada procesador (por lo mencionado del contenido de la entrada para *heapsort*) tal vez puede beneficiarse de mover las funciones de sincronización a BRAM.

3.4.3. Resultados

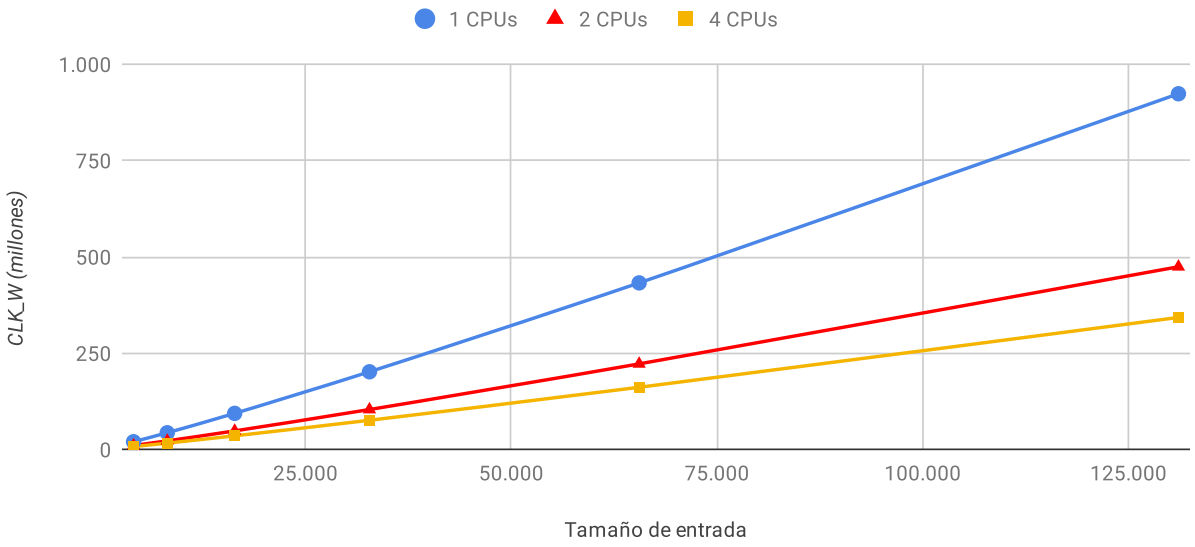


Figura 3.16: Ejecución del programa que hace *heapsort* sobre uno, dos y cuatro procesadores sin utilizar BRAM. Muestra la variación de tiempo por procesador en función de la cantidad de elementos a ordenar.

Observando la Figura 3.16 podemos ver que los tiempos se adhieren a la complejidad indicada.

También podemos ver que a mayor cantidad de procesadores, menor es el tiempo total. A 32,768 elementos (un caso intermedio), el *speed-up* para dos procesadores es de alrededor de 1,95 (cerca del teórico) y para cuatro de 2,69.

En la figura 3.17 se hace una comparación entre los usos de BRAM y la cantidad de procesadores utilizados para una entrada de 32,768 elementos. El desvío estándar obtenido es menor o igual al 0,28 % para todos los casos.

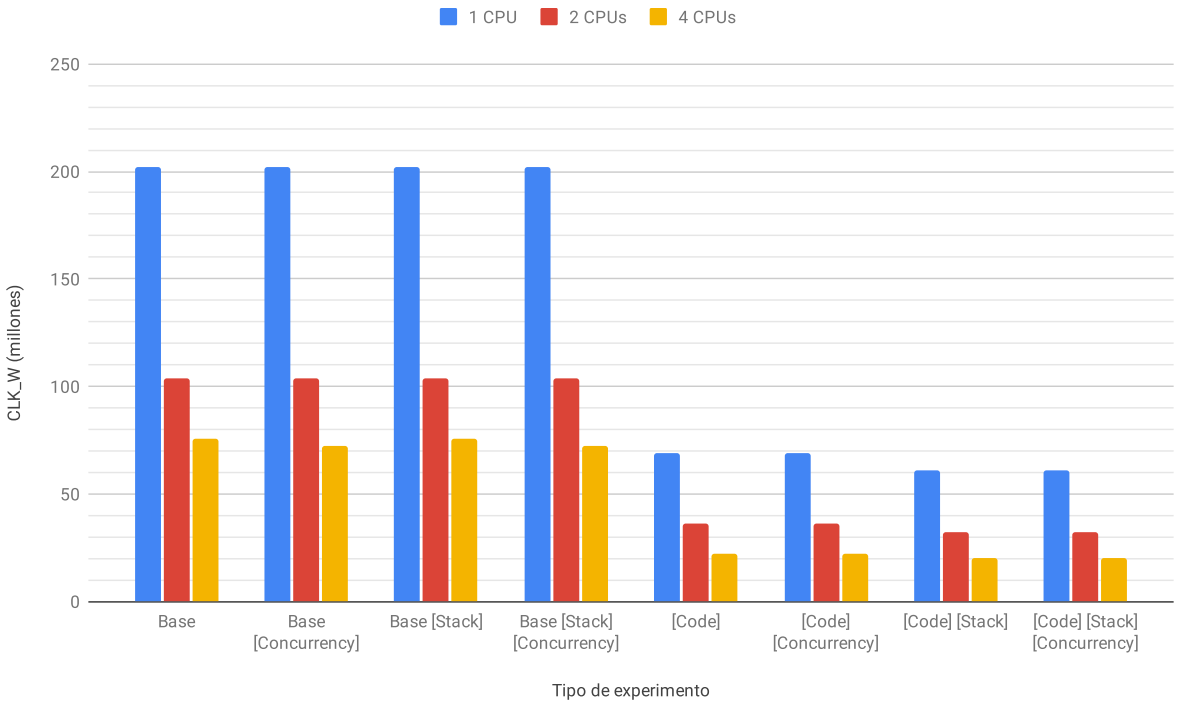


Figura 3.17: Ejecución del programa que hace *heapsort* para 32,768 elementos sobre uno, dos, y cuatro procesadores. Muestra la variación de tiempo por procesador (barras de colores) para cada uso de BRAM.

Tener el *stack* en la BRAM no presenta diferencias significativas cuando el código está en memoria principal (menos de 0,01 % para uno y dos procesadores, menos de 0,1 % para cuatro). Cuando éste está en BRAM, sin embargo, tener el *stack* allí también provoca mejoras del 11,9 % (un CPU), 11,2 % (dos CPUs) y 8,1 % (cuatro CPUs).

Contrario a las expectativas, mover el *stack* no parece tener impacto si el código permanece en memoria principal. Es interesante notar que la profundidad máxima de llamados a *heapify()* es del orden de $O(\log N)$. Para el caso de 32K de elementos, ésta sería de 15.

Mover el código a BRAM, genera mejoras en los tiempos del 65,8 % (un CPU), 65,2 % (dos CPUs) y 70,8 % (cuatro CPUs) comparando con el caso base. Con esto comprobamos nuevamente los beneficios de utilizar una memoria aislada y dedicada para el código.

Por otro lado, subir las funciones de concurrencia a BRAM no impacta significativamente en los tiempos para uno y dos procesadores (menor a 0,1 %). Para cuatro procesadores sin subir el código, presenta mejoras de un 4,9 % (sin *stack*) y 5,0 % (con *stack*). Al subir el código, esta mejora es de un 1,1 % (sin *stack*) y 1,4 % (con *stack*).

En este algoritmo no se garantiza una distribución equitativa de trabajo para cada procesador. En el caso de cuatro procesadores, cada partición de la entrada sobre la que se hace *heapsort* puede ejecutar un número distinto de operaciones. Esto se puede ver en *heapify()* (algoritmo 14), donde un chequeo puede detener las subsiguientes llamadas recursivas.

Junto con el cuello de botella que implica tener CPUs adicionales haciendo uso de la memoria, este desbalanceo de la carga puede causar que los procesadores finalicen su parte del trabajo de manera no simultánea. Al subir las funciones de concurrencia a BRAM, los procesadores que queden esperando estarán corriendo código en BRAM, haciendo únicamente accesos de solo lectura a `cpuFlags[]` (ver sección 2.3.1).

Esta mejora manifiesta un menor impacto cuando se sube el código a BRAM. Dado que

este caso demora menos que cuando se ejecuta código desde memoria principal, es posible que el desbalanceo de la carga impacte en menor medida.

De ser así, otra manera de ver esto es que sincronizar los procesadores dada esta carga desbalanceada provoca un aumento del tiempo de ejecución, el cual se reduce al mover las funciones de concurrencia a BRAM.

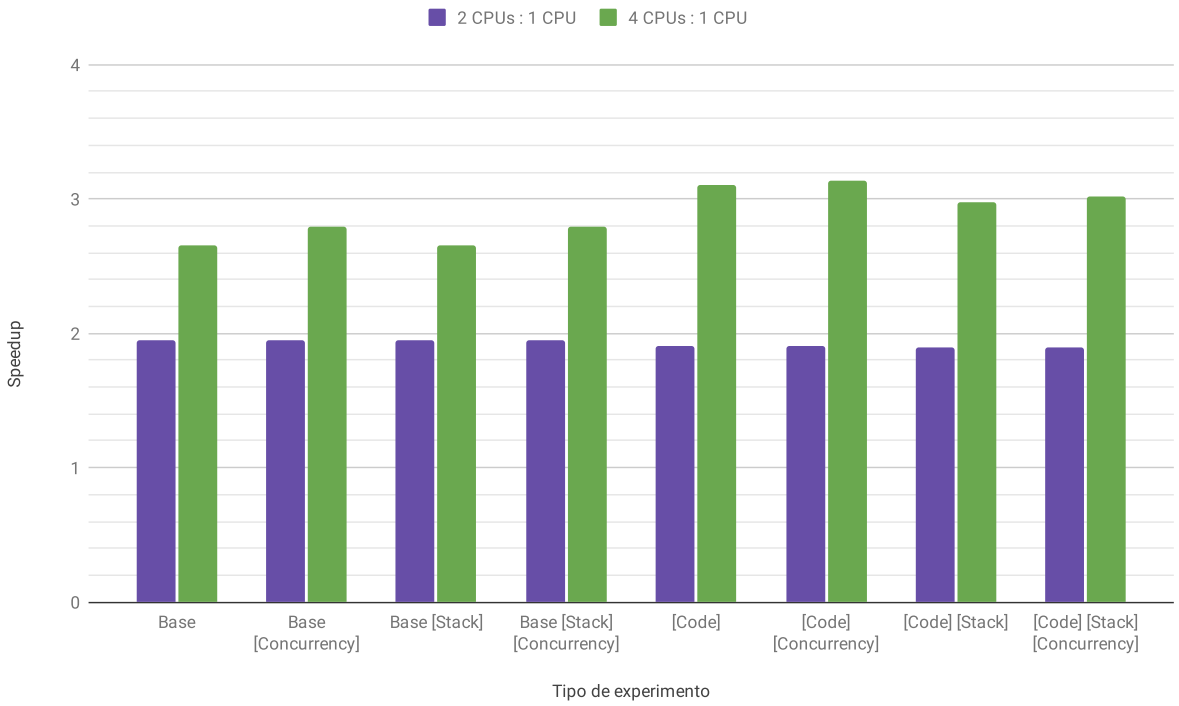


Figura 3.18: *Speed-up* registrado para dos y cuatro procesadores del programa que hace *heapsort* para 32,768 elementos. Muestra el *speed-up* (barras de colores) para cada uso de BRAM.

Respecto al *speed-up* (figura 3.18), observamos que es del 1,94 para dos procesadores moviendo o sin mover *stack* y funciones de concurrencia. Al subir el código, el *speed-up* pasa a ser de 1,91 (sin subir el *stack*) y 1,9 (subiéndolo). Este es un claro ejemplo del componente serial implícito en el código. Cuando el componente a paralelizar esta muy cerca de ser serial, es más ajustada la mejora que se puede obtener al paralelizarlo.

Para cuatro procesadores, el *speed-up* es del 2,66 moviendo o no el *stack* y 2,8 subiendo las funciones de concurrencia (con o sin *stack* en BRAM). Al subir el código, incrementa a 3,11 (sin *stack*, sin concurrencia), 3,14 (sin *stack*, con concurrencia), 2,98 (con *stack*, sin concurrencia) y 3,02 (con *stack* y concurrencia).

Recordamos que el *speed-up* resultante de 2,98 para la configuración con código y *stack* en BRAM es contra la versión de un procesador con la misma configuración. Comparar estos *speed-up* indica si el programa se beneficia más o menos al aumentar los procesadores. La discrepancia se puede explicar en que, al subir el *stack*, toma mayor protagonismo el conflicto de los datos a procesar en memoria (ya que el *stack* pasa a estar en una memoria aislada).

En este caso, la comparación contra el *speed-up* de 3,11 (donde solo se sube el código a BRAM) es indicativa de que dicha configuración reporta mayor beneficio al aumentar la cantidad de procesadores de uno a cuatro.

Es interesante ver que, si bien está muy cerca, no se llega al máximo teórico para dos procesadores (como ocurrió en la sección 3.3). Tampoco se llega al máximo para cuatro

procesadores, quedando a un 0,9 de éste.

En comparación con los resultados del *speed-up* para Multiplicación de Matrices, donde éste puede llegar al teórico, esto parece indicar que nuestra implementación del algoritmo (para el `Sistema` utilizado en este trabajo) no se está beneficiando tanto de utilizar más procesadores.

3.4.4. Conclusión

Si bien subir el *stack* a BRAM no generó diferencias contra el caso base, dio mejoras significativas en conjunto con mover el código. Este último es quien posee el mayor impacto, llegando a reducir severamente los tiempos, comparando con el caso base.

Mover las funciones de concurrencia parece tener mayores beneficios cuando el procesamiento puede estar desbalanceado entre los cuatro procesadores. Es importante recordar que se trata de mover una parte muy pequeña del código a BRAM. De tener un *IP Core* dedicado a la exclusión mutua, es posible que esta mejora en los tiempos sea incluso mayor, dado que eliminaría la necesidad de utilizar *flags* en memoria principal. También sería posible que este *IP Core* bloquee el CPU durante un *lock*, eliminando o reduciendo las instrucciones necesarias para sincronizar.

3.5. FFT - Fast Fourier Transform

3.5.1. Introducción

Una DFT (*Discrete Fourier Transform*) es una transformación que convierte una señal del dominio tiempo al dominio frecuencia (ver figura 3.19). La DFT es utilizada en análisis de circuitos, procesamiento de imágenes y procesamiento de señales, entre muchas aplicaciones [21].

El algoritmo de FFT es una optimización de la DFT que tiene un orden de complejidad $O(N \log N)$, siendo N la cantidad de muestras tomadas de una señal.

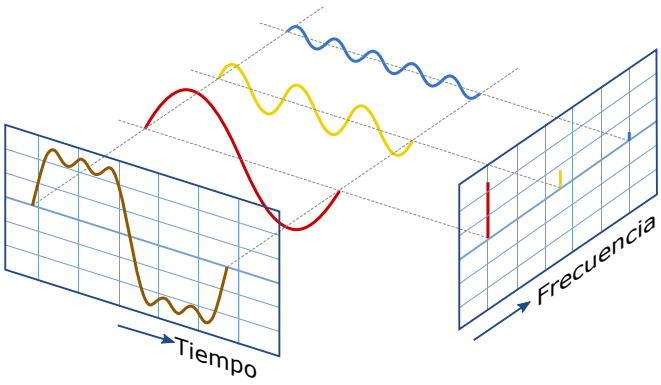


Figura 3.19: Transformación de una señal del dominio del tiempo a sus componentes armónicas en dominio de la frecuencia.

Para este trabajo se implementaron dos variantes de la FFT, una iterativa y una recursiva. Ambas resuelven el problema de manera muy diferente.

En particular, tienen un patrón de acceso y de procesamiento distinto. La variante recursiva siempre accede a memoria de manera continua, mientras que la iterativa accede haciendo saltos regulares (en potencias de dos). En términos del procesamiento, la variante recursiva hace muchas más operaciones trigonométricas que la iterativa (las cuales son costosas).

```
void butterfly(float* data, int nn) {
    unsigned long n = nn<<1;
    unsigned long i, j, m;
    // reverse-binary reindexing
    j=1;
    for (i=1; i<n; i+=2) {
        if (j>i) {
            swap(data[j-1], data[i-1]);
            swap(data[j], data[i]);
        }
        m = nn;
        while (m>=2 && j>m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
}
```

Algoritmo 16: Función butterfly()

Con esto en mente, nos parece interesante analizar el desempeño de ambas implementaciones en nuestra plataforma, y observar las diferencias que produce cada implementación ante los distintos usos de BRAM.

Las señales de entrada y salida se codifican como una serie de números complejos en punto flotante. Utilizamos la biblioteca *math* de C para implementar las funciones trigonométricas.

```
typedef struct _complexNum {
    float r;
    float i;
} complexNum;

void fft(complexNum data[], int N) {
    fft_recursive(data, N);
    butterfly((float*) data, N);
}

void fft_recursive(complexNum data[], int N) {
    int n2, k1, N1, N2;
    complexNum W, bfly[2];
    N1 = 2;
    N2 = N / 2;
    for (n2 = 0; n2 < N2; n2++) {
        float di = (float)((float) n2 / (float) N);
        float theta = -(TAU * di);
        W.r = cosf(theta);
        float sign = (theta >= 0.0f) ? 1.0f : -1.0f;
        W.i = sign * sqrtf(1.0f - W.r * W.r); // sin(theta)
        bfly[0].r = data[n2].r + data[N2 + n2].r;
        bfly[0].i = data[n2].i + data[N2 + n2].i;

        bfly[1].r =   (data[n2].r - data[N2 + n2].r) * W.r
                    - ((data[n2].i - data[N2 + n2].i) * W.i);
        bfly[1].i =   (data[n2].i - data[N2 + n2].i) * W.r
                    + ((data[n2].r - data[N2 + n2].r) * W.i);

        for (k1 = 0; k1 < N1; k1++) {
            data[n2 + N2*k1].r = bfly[k1].r;
            data[n2 + N2*k1].i = bfly[k1].i;
        }
    }

    if (N2 != 1) {
        for (k1 = 0; k1 < N1; k1++) {
            fft(&data[N2 * k1], N2);
        }
    }
}
```

Algoritmo 17: FFT Recursivo

3.5.2. Técnica de paralelización

A continuación veremos los algoritmos utilizados para ambas variantes y cómo fueron paralelizados. En ambos casos se utiliza la función $butterfly()$ que reordena la entrada para su procesamiento (o salida). Ésta se presenta en el algoritmo 16 y era originalmente parte

del algoritmo iterativo pero no del recursivo. Fue agregada a este último para que ambos generen el mismo resultado.

Cabe destacar que no se realizó ninguna modificación para paralelizar $butterfly()$, por lo que el código será ejecutado secuencialmente en todos los casos.

Versión recursiva

La versión recursiva puede verse en el Algoritmo 17. En este caso optamos por denotar los tipos de las variables dada la complejidad del algoritmo.

A simple vista, no podemos partir la entrada porque debe ser recorrido en su totalidad. Sin embargo, al tratarse de una recursión, podemos optar por partir la llamada recursiva entre los distintos *softcores*, reemplazando las líneas 37 a 40 de Algoritmo 17.

```
12  void fft_recursive(complexNum data[], int N , uint cpuID, int depth ) {
...
37      if (N2 != 1) {
38  #if CPUS > 1
39          if (depth == 0) {
40              N1 = 1;
41              launchJob(1);
42          }
43  #if CPUS > 2
44          else if (depth == 1) {
45              N1 = 1;
46              if (cpuID == MAIN_CPU) {
47                  launchJob(2);
48              } else {
49                  launchJob(3);
50              }
51          }
52    #endif
53    #endif
54
55          for (k1 = 0; k1 < N1; k1++) {
56              fft_recursive(&data[N2 * k1], N2, cpuID, depth + 1);
57          }
58      }
59  }
```

Algoritmo 18: Paralelizado de FFT Recursivo

Este cambio se puede ver en Algoritmo 18. Introducimos nuevos argumentos, `cpuID` y `depth`, por lo cual necesitamos modificar cómo se invoca a la función. La idea es que el procesador principal ejecute en solitario la primera parte de la FFT y luego se concentre en una de las ramas del paso recursivo, la otra rama sería ejecutada por el segundo procesador.

De haber cuatro procesasores, la segunda bifurcación repetirá el proceso para CPU0 y CPU1, cada uno corriendo en solitario, despertando a CPU2 y CPU3 y continuando con el procesamiento de una de las ramas de la nueva recursión. Luego de esto, cada procesador tendrá que explorar todas las sucesivas bifurcaciones que encuentre.

En el algoritmo 19 puede verse el código simplificado que maneja la inicialización y sincronización.

```
1
2  void run(complexNum input[], int inputSize, uint cpuID) {
3      if (cpuID == 0) {
4          fft_recursive(input, inputSize, cpuID, 0);
5          join(cpuID);
6
7          butterfly(input, inputSize);
8          return;
9      }
10
11     waitJob(cpuID);
12     if (cpuID == 1) {
13         fft_recursive(input + (inputSize * 2) / 4, inputSize / 2, cpuID, 1);
14     } else if (cpuID == 2) {
15         fft_recursive(input + (inputSize * 1) / 4, inputSize / 4, cpuID, 2);
16     } else {
17         fft_recursive(input + (inputSize * 3) / 4, inputSize / 4, cpuID, 2);
18     }
19     finishJob(cpuID);
20 }
21
```

Algoritmo 19: Pseudocódigo de inicialización de programa

Esta técnica es relativamente sencilla de aplicar. Una desventaja es que en los primeros llamados de función no se están aprovechando todos los *softcores*.

Versión iterativa

La versión iterativa[3] puede verse en el Algoritmo 20. Nuevamente optamos por dejar los tipos de las variables dada la complejidad del algoritmo.

Hay una diferencia en los argumentos entre ambas variantes (en términos de notación): en esta versión se recibe un *array* de *float* en vez de un *array* de *struct*. A efectos prácticos ambas variantes reciben la misma entrada en el mismo orden, siendo la diferencia en código meramente cosmética.

El proceso iterativo consiste en un *while* y dos *for*. El *for* más interno puede ser particionado entre los distintos procesadores. Dado que la implementación está modificando la entrada de manera *in situ*, es necesario sincronizar los *softcores* para que trabajen sobre el mismo "step" de entrada. Para esto agregamos barreras de sincronización antes del *for*, y luego en el *join*.

Además agregamos el *cpuID* como parámetro, que permite identificar la partición del problema que se asigna a cada CPU. Los cambios realizados sobre el código original pueden verse en el Algoritmo 21.

[3]Extraída de `https://www.drdobbs.com/cpp/a-simple-and-efficient-fft-implementatio/199500857`. Algoritmo originalmente implementado en W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P.Flannery. Numerical Recipes in C++. Cambridge university press, 2002, p.513

```
void fft_iterative(float data[], int N) {
    unsigned long n, mmax, m, j, istep, i;
    float wtemp, wr, wpr, wpi, wi, theta, tempr, tempi;

    n = N << 1;
    j = 1;

    butterfly((float*) data, N);

    mmax = 2;
    while (n > mmax) {
        istep = mmax << 1;
        theta = -(TAU / mmax);
        wtemp = sinf(0.5f * theta);
        wpr = -2.0f * wtemp * wtemp;
        wpi = sinf(theta);
        wr = 1.0f;
        wi = 0.0f;
        for (m = 1; m < mmax; m += 2) {
            for (i = m; i <= n; i += istep) {
                j = i + mmax;
                tempr = wr * data[j - 1] - wi * data[j];
                tempi = wr * data[j] + wi * data[j - 1];

                data[j - 1] = data[i - 1] - tempr;
                data[j] = data[i] - tempi;
                data[i - 1] += tempr;
                data[i] += tempi;
            }
            wtemp = wr;
            wr += wr * wpr - wi * wpi;
            wi += wi * wpr + wtemp * wpi;
        }
        mmax = istep;
    }
}
```

Algoritmo 20: FFT Iterativo

```
1   void fft_iterative(float data[], int N , int cpuID ) {
...
8       if (cpuID == MAIN_CPU) {
9         butterfly((float*) data, N);
10        launchAll();
11      }
12
13      // barrier
14      if (cpuID != MAIN_CPU) waitJob(cpuID);
15      finishJob(cpuID);
16
17      mmax = 2 ;
18      while (n > mmax) {
19          istep = mmax << 1;
20          theta = -(TAU / mmax);
21          wtemp = sinf(0.5f * theta);
22          wpr = -2.0f * wtemp * wtemp;
23          wpi = sinf(theta);
24          wr = 1.0f;
25          wi = 0.0f;
26          for (m = 1; m < mmax; m += 2) {
27              if (cpuID == MAIN_CPU) launchAll();
28              else                   waitJob(cpuID);
29
30              for (i=m + istep*cpuID; i <= n; i += istep * CPUS) {
31                  j = i + mmax;
32                  tempr = wr * data[j - 1] - wi * data[j];
33                  tempi = wr * data[j] + wi * data[j - 1];
34
35                  data[j - 1] = data[i - 1] - tempr;
36                  data[j] = data[i] - tempi;
37                  data[i - 1] += tempr;
38                  data[i] += tempi;
39              }
40              wtemp = wr;
41              wr += wr * wpr - wi * wpi;
42              wi += wi * wpr + wtemp * wpi;
43
44              if (cpuID == MAIN_CPU) join(cpuID);
45              else                   finishJob(cpuID);
46          }
47          mmax = istep;
48      }
49  }
```

Algoritmo 21: Paralelizado de FFT Iterativo

3.5.3. Funciones intrínsecas

Se dicen funciones intrínsecas a aquellas que exporta una biblioteca del compilador (más precisamente en el paso de generación de código). Ejemplos de esto son algunas de las funciones matemáticas de la *libm* (la biblioteca matemática de C). Si el procesador las implementa como instrucciones específicas, la *libm* puede utilizarlas. En caso contrario, la biblioteca deberá implementarlas usando instrucciones existentes (es decir, por software), generalmente incurriendo en mayores tiempos.

Informar al compilador de las extensiones que tiene nuestro MicroBlaze (FPU, multiplicación y división entera, etc) no es suficiente en este caso, ya que la *libm* se encuentra precompilada. Hay varios tipos de *libm* disponibles para utilizar, cada una precompilada utilizando un subconjunto de extensiones.

Cabe destacar que no todas las funciones de la *libm* tendrán una instrucción específica en el MicroBlaze. Algunas de ellas tendrán que ser simuladas parcialmente por software.

En particular nos interesan las funciones trigonométricas. Tomando la *libm* más completa y compatible con nuestro MicroBlaze, podemos evitar que sean simuladas por software **en su totalidad**.

Para el caso de la FFT, tanto $sinf(x)$ como $cosf(x)$ no existen como instrucciones de MicroBlaze. Dada cualquier *libm*, estas funciones estarán simulándose por software, aún eligiendo la *libm* precompilada con extensiones de punto flotante.

Una simulación parcial es mucho mejor que una total, ya que las operaciones de punto flotante necesarias para calcular $sinf(x)$ o $cosf(x)$ estarán aprovechando la FPU.

En contraste, $sqrtf(x)$ **existe** como instrucción de MicroBlaze y la estaremos utilizando para aprovechar el hardware.

Si bien MicroBlaze posee una FPU, en nuestro trabajo utilizamos *softcores* de 32 bits que solo implementan operaciones de punto flotante con precisión simple. Las operaciones flotantes de precisión doble deben ser simuladas por software. En un MicroBlaze de 64 bits, en cambio, las operaciones con ambos tipos de datos están implementadas por hardware [18].

Esto implica que no solo debemos utilizar las funciones intrínsecas para *floats*, si no que también debemos asegurarnos de que cada constante utilizada en el código sea interpretada como de precisión simple. De no hacer esto estaremos permitiendo que el compilador las interprete como *double* (el caso por defecto) e introduciendo operaciones simuladas por software de manera innecesaria.

3.5.4. Experimentos

Para los experimentos necesitaremos una lista de muestras de una señal de entrada (a intervalos regulares). En este caso haremos uso de una onda cuadrada de 32 Hz, cuya transformada de fourier es conocida. El muestreo se hizo sobre un segundo de duración, tomando 512, 1024, 4096 y 16,384 muestras.

Se espera que el tiempo de ejecución sea del orden de $O(N\ \log(N))$, siendo N la cantidad de muestreos.

Al contrario que en experimentos anteriores donde operamos con enteros y el resultado es una expresión exacta, al operar en punto flotante la aproximación del resultado nos obliga a considerar un margen de error entre la solución esperada y la obtenida.

El chequeo de resultados se hizo utilizando la implementación de FFT dada por *numpy*, verificando que el error relativo sea menor a 0,01. Considerando que estamos comparando

resultados de una implementación que utiliza precisión simple con otra que puede estar utilizando mayor precisión, este margen de error resulta suficiente.

Si bien para nuestra entrada el error está por debajo de este margen, una mayor cantidad de muestras implicará más operaciones y mayor error numérico. También es posible que la implementación contra la que se compara priorice tener menor error numérico antes que un menor tiempo de ejecución. Si bien el resultado para una onda cuadrada es conocido, decidimos armar un chequeo general.

Es interesante notar que al aumentar el número de muestras, la diferencia (error) entre los resultados continuará creciendo.

Por otro lado, respecto la BRAM y particularmente al tamaño elegido (8 kB), el uso de la *libm* nos presenta un problema.

Como la *libm* es grande, no entra en BRAM en su totalidad junto con el *stack*. Más aún, no podemos aislar de manera consistente las funciones trigonométricas y aquellas que éstas utilizan como para subir una porción de la *libm*.

Por lo tanto, optamos por subir a BRAM únicamente el código del algoritmo, dejando las implementaciones de $sinf(x)$ y $cosf(x)$ de la *libm* en memoria principal. Es muy posible que esto reduzca el impacto de los casos donde se sube el código a BRAM, comparado con los experimentos anteriores.

3.5.5. Resultados y análisis

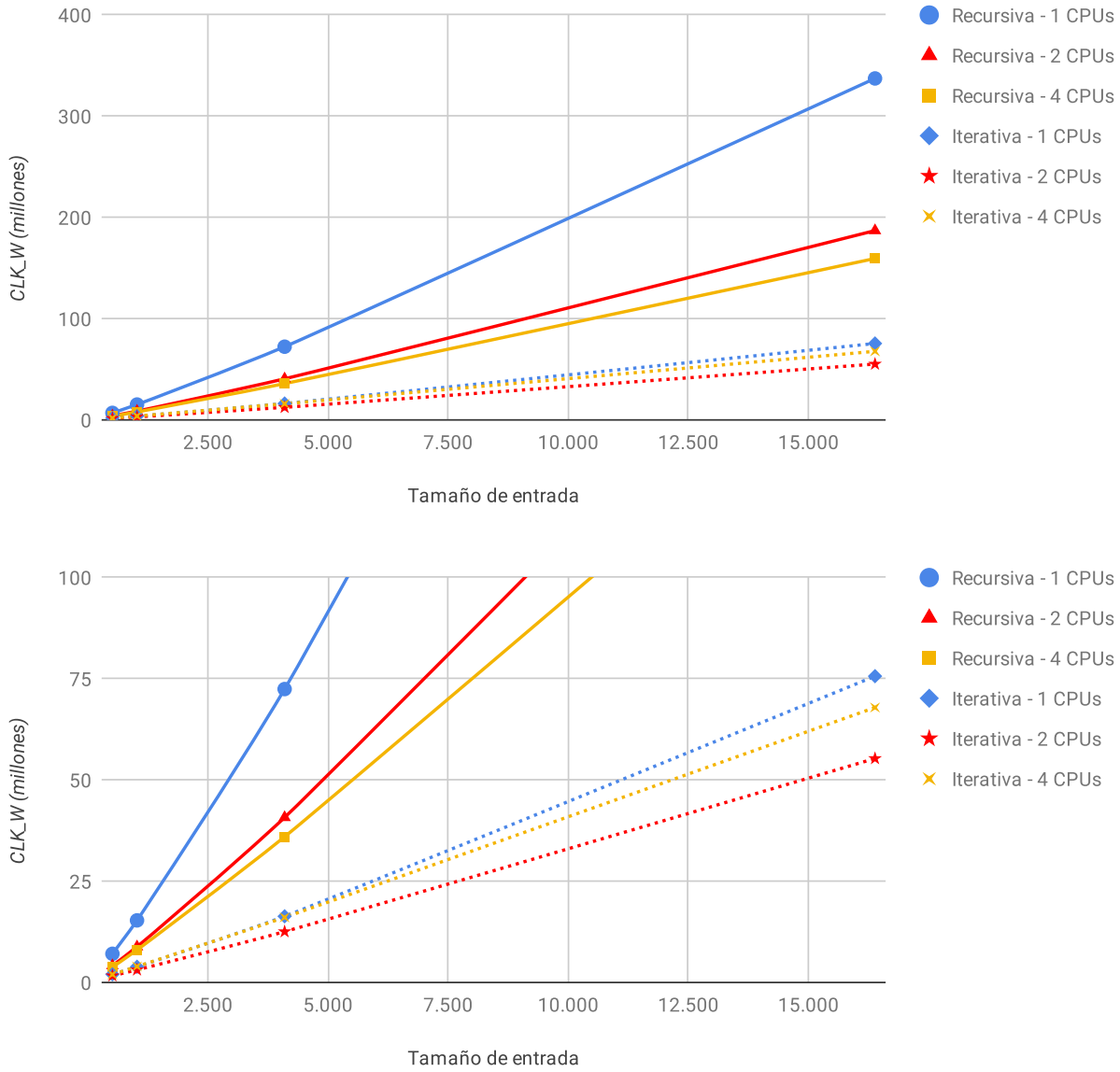


Figura 3.20: Ejecución del programa que calcula la FFT de una señal sobre uno, dos y cuatro procesadores sin utilizar BRAM. Incluye ambas variantes y muestra la variación de tiempo por procesador en función de la cantidad de muestras. El gráfico inferior hace *zoom* para enfocarse en la variante iterativa.

En la figura 3.20 podemos ver una clara diferencia entre las variantes para el caso base. El tiempo de ejecución de la variante iterativa es inferior a cualquiera de las variantes recursivas, independientemente del número de procesadores utilizados. Observamos que el mejor caso de la versión recursiva (cuatro procesadores) requiere cerca del doble de ciclos de *clock* que la variante iterativa utilizando un procesador. Por otro lado, vemos que la variante iterativa de cuatro procesadores tiene un rendimiento inferior a la de dos, consiguiendo un rendimiento cercano al de un único procesador.

Para explicar estas diferencias, analizaremos primero cada variante por su lado.

En la figura 3.21 vemos el resultado para 1024 muestras de la señal de entrada para la variante recursiva. El desvío estándar obtenido es menor o igual al 0,31 % para casi todos

los casos, siendo del 0,65 % para cuatro procesadores con solo subir el *stack* a BRAM y de 0,89 % subiendo al *stack*, código y funciones de concurrencia a BRAM.

Se destaca entre los resultados que subir el código no impacta tanto como en otros experimentos. Se trata de un 16 % (1 CPU), 16,5 % (2 CPUs) y 21,8 % (4 CPUs) de mejora en los tiempos. Esto se puede explicar por el hecho de que el experimento se limita a subir a BRAM el código generado (el algoritmo de FFT), excluyendo las funciones de biblioteca (las funciones trigonométricas).

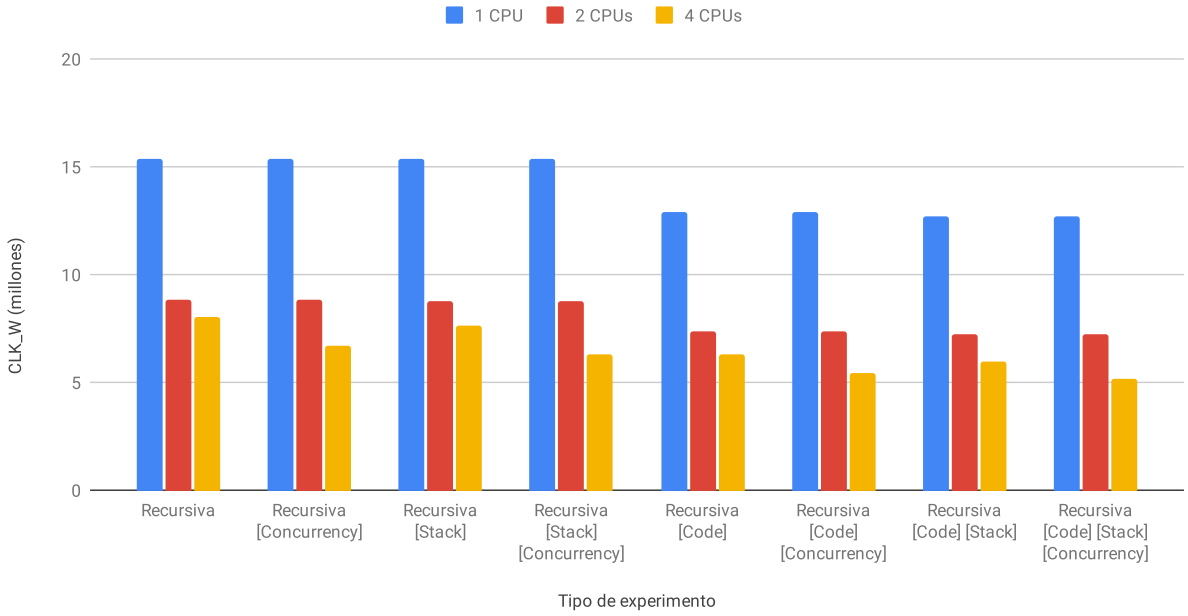


Figura 3.21: Ejecución del programa que calcula la FFT de una señal con 1024 muestras sobre uno, dos, y cuatro procesadores. Incluye la variante recursiva y muestra la variación de tiempo por procesador (barras de colores) para cada uso de BRAM.

Al subir el algoritmo a BRAM, no hace falta que esté alojado en la *cache*. Esto debería ayudar a evitar desalojos y podría explicar el incremento del rendimiento para cuatro procesadores comparado con dos.

Tener el *stack* en BRAM no parece tener impacto para el caso de un procesador, obteniendo mejoras en el tiempo de ejecución del 0,1 % (sin subir código) y 1,6 % (subiéndolo). Para dos procesadores, vemos una mejora del 0,6 % (sin subir código) y del 2,2 % (subiéndolo). Recién con cuatro procesadores notamos diferencias significativas del 5 % (sin subir código) y 4,9 % (subiéndolo).

Las funciones de concurrencia en BRAM no presentan diferencias para un procesador (no se usan) ni para dos procesadores, siendo en ambos casos menores a 0,1 % sin importar si es el caso base o se sube el código/*stack* a BRAM. Para cuatro procesadores, en cambio, hay una mejora del 16,8 % (vs caso base), 17,6 % (*stack* en BRAM), 13,7 % (código en BRAM) y 14,2 % (código y *stack* en BRAM).

Recordemos que a profundidad 0 de la recursión solo estará trabajando el CPU0, mientras que a profundidad 1 tendremos CPU0 y CPU1. Recién a profundidad 2 estarán utilizándose los cuatro procesadores. Es decir, existe un componente serial que limita el *speed-up*.

Este impacto en los tiempos al subir las funciones de concurrencia parece indicar que tener varios procesadores en *spin lock* sobre memoria principal es detrimental para el desempeño general.

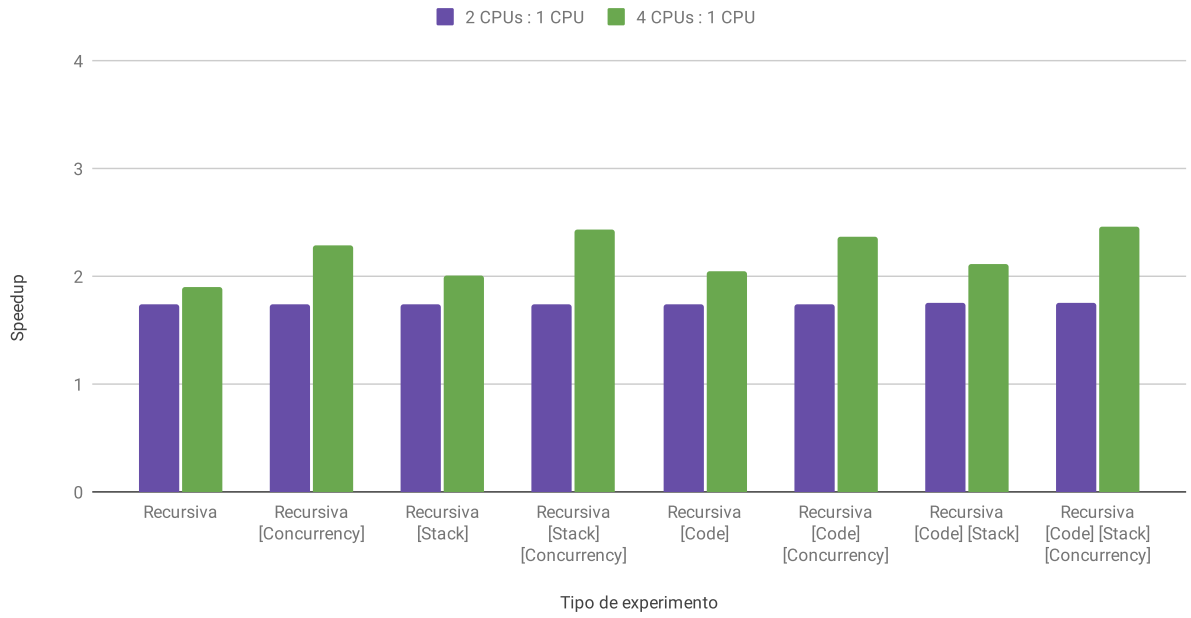


Figura 3.22: *Speed-up* registrado para dos y cuatro procesadores del programa que hace FFT de una señal con 1024 muestras. Incluye la variante recursiva y muestra el *speed-up* (barras de colores) para cada uso de BRAM.

Con respecto al *speed-up* (figura 3.22), vemos que para dos procesadores ronda entre 1,74 a 1,76, siendo menor para el caso base y aumentando ligeramente a medida que se suben cosas a BRAM.

Para cuatro procesadores, hay mucha más variedad:

Sin subir el código tenemos 1,91 (sin *stack* ni concurrencia), 2,29 (concurrencia), 2,01 (*stack*) y 2,43 (ambos) al comparar contra la misma configuración pero con un único procesador.

Subiendo el código, éste es del 2,04 (sin *stack* ni concurrencia), 2,37 (concurrencia), 2,12 (*stack*) y 2,47 (ambos).

De estos resultados podemos extraer que al subir las funciones de concurrencia se aprovechan mejor los recursos cuando se incrementa la cantidad de procesadores.

En la figura 3.23 vemos el resultado para 1024 muestras de la variante iterativa. El desvío estándar obtenido es menor o igual al 0,32 % para casi todos los casos, siendo la excepción un 0,58 % para cuatro procesadores con subir el *stack* y código a BRAM e independiente de lo que se haga con las funciones de concurrencia.

Lo primero que llama la atención es que para cuatro procesadores el tiempo de ejecución no solo es peor que para dos, sino que a veces llega superar el de un solo procesador. La única configuración que contrarresta esto es subir las funciones de concurrencia, aunque esta resulta en tiempos similares para cuatro y dos procesadores.

Subir el código genera mejoras de tiempo del 49,4 % (1 CPU), 30,2 % (2 CPUs) y 30,4 % (4 CPUs) comparando con el caso base. Esto parece indicar que las rutinas del código que no están cubiertas (la función $sinf(x)$ y sus funciones auxiliares que no se suben a BRAM) resultan un cuello de botella contra la memoria para dos o más procesadores.

Tener el *stack* en BRAM no parece tener impacto para un procesador, obteniendo mejoras en el tiempo de ejecución del 0,1 %. Para dos procesadores, esto se mantiene en general, habiendo una mejora del 0,2 % para el caso de subir todo a BRAM. Con cuatro procesadores, tampoco notamos diferencias significativas, siendo del 0,9 % (sin subir código ni concurren-

cia), 0,4 % (subiendo concurrencia) y 0,1 % (subiendo ambos).

La variante iterativa no requiere un uso extensivo de la memoria en el *stack*: las variables utilizadas entran en los registros disponibles y las únicas funciones que son llamadas son $sinf(x)$ y sus auxiliares. Es posible que estas funciones utilicen el *stack* para hacer los cálculos necesarios pero parece que tenerlo en una memoria más rápida no genera mucho impacto.

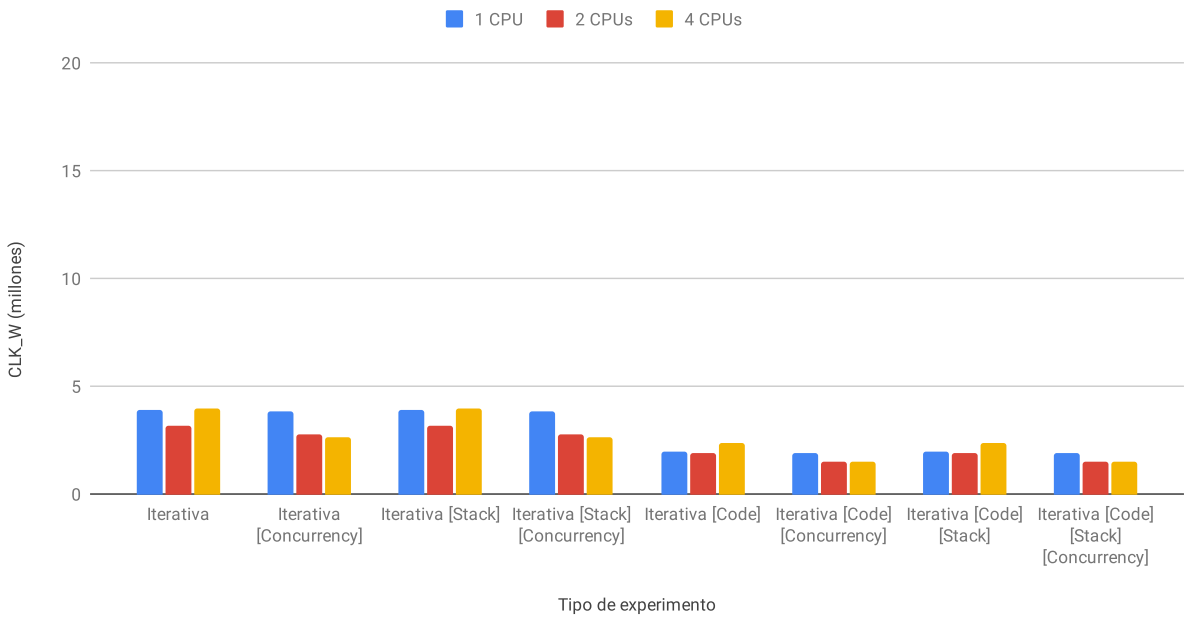


Figura 3.23: Ejecución del programa que calcula la FFT de una señal con 1024 muestras usando uno, dos, y cuatro procesadores. Incluye la variante iterativa y muestra la variación de tiempo por procesadores (barras de colores) para cada uso de BRAM.

Al subir las funciones de concurrencia notamos mejoras en los tiempos del 1,7 % (sin subir código) y 3,3 % (subiéndolo) para un procesador. Con dos procesadores esta mejora asciende al 11,4 % (sin código) y 18,7 % (con código). Para cuatro procesadores, ésta es del 33,8 % (sin código) y 37,8 % (con código).

Lo primero a destacar es que las funciones de concurrencia en BRAM están generando diferencias en los tiempos para un procesador. En algoritmos previos hemos evitado usarlas para el caso de un procesador mediante macros que ignoran esa parte del código. En esta variante, sin embargo, están presentes, y notamos que impactan de manera no trivial en los tiempos.

La llamada a estas funciones afecta negativamente el desempeño. Al estar ocurriendo incluso en el caso de un solo procesador, podemos concluir que no hace falta que el procesamiento se encuentre desbalanceado entre los procesadores. Revisando el algoritmo 1, vemos que las funciones de concurrencia para un procesador hacen una iteración sobre los *flags*. Es posible que la posición en memoria de estas funciones, junto con los accesos a estos *flags* estén desalojando líneas de cache que el programa continuará utilizando (ver algoritmo 21).

La L2 tiene 32 kB con dos vías de 64 B, lo que implica que cada múltiplo de 16 kB (`0x4000`) entraría en un set distinto. El binario del programa completo contiene más de 10,000 B (`0x2710`), y se ubica en memoria alineado a `0x4000` (la parte superior de los 512 MB de DDR). Las 1024 muestras como entrada del algoritmo implican 8 kB de datos (`0x2000`), los cuales están alineados a `0x4000` por estar en la posición `0x800000`.

Cada *stack* de 4 kB se encuentra en una posición alta de memoria. Los cuatro *stacks* en DDR (`0x1000*4 = 0x4000`) podrían estar ocupando un conjunto entero. Para el caso de un procesador, el *stack* en la posición más alta de memoria sería el único utilizado. Mover el stack a BRAM no presenta mejoras significativas en el desempeño para un procesador, probablemente porque el *stack* en uso no entra en conflicto con el resto de los accesos.

El código de las FFT iterativa base (`fft.c`) y con funciones de concurrencia (`fft_sram.c`) difieren solamente en el *multicore_*.h*. Estos `.h` suben las funciones de concurrencia a BRAM en `setupCore()` y solo difieren en cómo setean los punteros de las funciones de concurrencia, siendo a memoria de código (`multicore_normal.h`) o a memoria de BRAM (`multicore_sram.h`). Particularmente, el algoritmo de FFT solo haría llamados a la función `join()` para el caso de un procesador (no usaría `waitJob(cpuID)`).

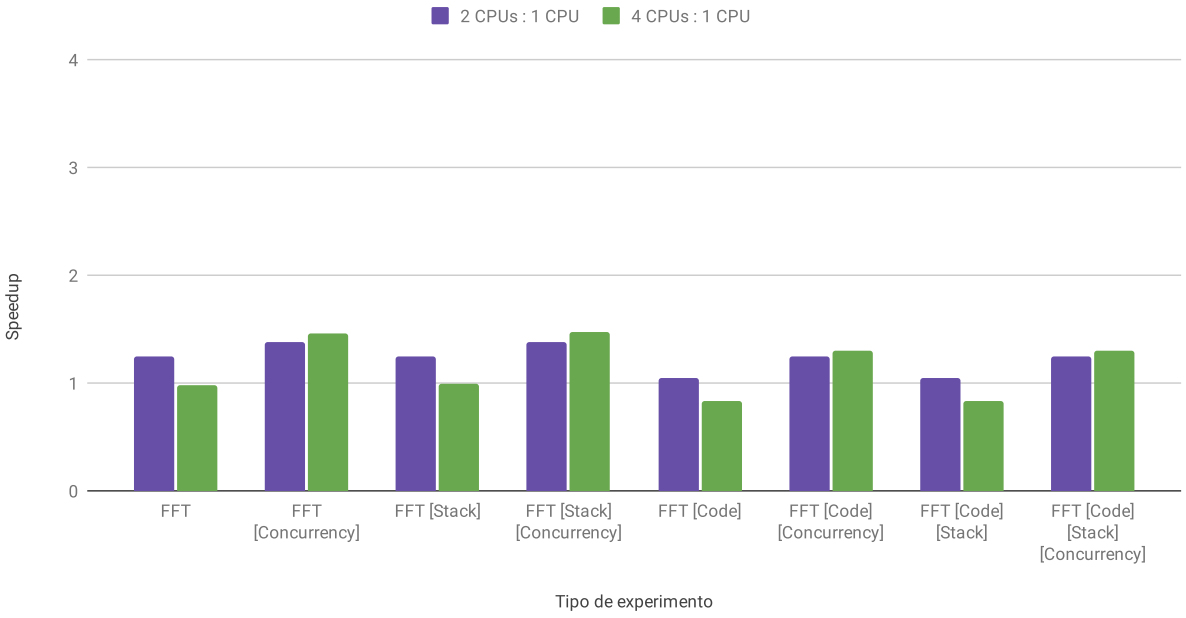


Figura 3.24: *Speed-up* registrado para dos y cuatro procesadores del programa que hace FFT de una señal con 1024 muestras. Incluye la variante iterativa y muestra el *speed-up* (barras de colores) para cada uso de BRAM.

Con respecto al *speed-up* (figura 3.24), vemos que para dos procesadores ronda entre 1,24 (base), 1,38 (concurrencia), 1,05 (código), 1,25 (código y concurrencia). Mover el *stack* no genera diferencias significativas (menor a 0,1 %).

Para cuatro procesadores, en algunos casos el *speed-up* resulta peor que con dos, o incluso un procesador. Este es de 0,99 (base), 1,46 (concurrencia), 0,83 (código) y 1,30 (código y concurrencia). Mover el *stack* tampoco muestra diferencias significativas (menor a 0,8 %).

Es evidente que la estrategia de paralelización elegida no es óptima para nuestro sistema. El hecho de que el desempeño sea peor comparando cuatro y dos procesadores, indica un cuello de botella en los recursos que no fue notado en los casos anteriores. Subir las funciones de concurrencia parece aliviar fuertemente este problema, llegando a superar la mejora que proporciona mover el código del algoritmo en el caso de cuatro procesadores.

Mover pequeñas partes del código como los *spin-locks* puede generar tanto impacto como mover el algoritmo completo.

Al comparar ambas variantes (figuras 3.21 y 3.23), podemos ver que el peor de los casos de la versión iterativa para un procesador tarda menos en ejecutar que el mejor de los casos de la versión recursiva para cuatro.

También podemos destacar que la recursiva escala mejor con más procesadores, habiendo muy poca diferencia para la variante iterativa entre los mejores casos de dos y cuatro. Esta diferencia puede explicarse por la cantidad de trabajo total y la diferente estrategia de paralelización.

Un punto a destacar es que la variante recursiva realiza $O(N \log(N))$ llamados a la función $cosf(x)$. La iterativa, en cambio, tiene $O(\log(N))$ llamados a $sinf(x)$.

Recordemos que las funciones trigonométricas vienen implementadas en la `libm`, la cual no subimos a BRAM, y deben ser parcialmente implementadas por software ya que `MicroBlaze` no provee una instrucción específica (como sí es el caso para $sqrtf(x)$). Éste puede ser el principal causante de la diferencia de tiempos entre ambas variantes.

Considerando los algoritmos desde el punto de vista de llamados a funciones trigonométricas (las más caras), el recursivo tiene una mayor cantidad de trabajo que el iterativo.

Por otra parte, la estrategia de paralelización de la versión recursiva es simple y solo requiere un `join()` o un `waitJob(cpuID)` por cada procesador. Una vez que los procesadores empiezan a trabajar, no requieren sincronizar el trabajo. El CPU0 hace el `join()` para poder finalizar el programa con la salida completa. En la versión iterativa, en cambio, tenemos $O(N)$ sincronizaciones.

Como experimento de control reemplazamos los llamados a funciones trigonométricas por valores constantes. Notamos que los tiempos de la variante recursiva para cualquier cantidad de procesadores resultaron mejores que los de la variante iterativa para cualquier cantidad de ellos.

Esto nos confirma dos cosas: por un lado, nos da una pauta de que si pudiéramos implementar eficientemente las funciones trigonométricas, es posible que la variante recursiva tenga mejor rendimiento que la iterativa. Por el otro, nos indica que la estrategia de paralelización de la variante iterativa es un cuello de botella.

Otro punto a notar es que la variante iterativa toma datos de memoria haciendo saltos que muchas veces caen en otra línea de *cache*. En cambio, la versión recursiva trabaja la memoria de manera secuencial desde los bordes del arreglo hacia el interior, lo cual permite aprovechar el principio de localidad.

3.5.6. Conclusión

La estrategia de paralelización del caso iterativo solo obtiene beneficios significativos al pasar de uno a dos procesadores, mientras que la recursiva podría seguir beneficiándose de procesadores adicionales.

Uno de los puntos a destacar es que subir las funciones de concurrencia produce una de las mejoras más significativas para la variante iterativa. Esto puede deberse a que la estrategia de paralelización no es óptima y que el subir los *spin locks* a BRAM simplemente alivia los *cache-misses*. El no subir las funciones trigonométricas (utilizadas más frecuentemente en la variante recursiva) parece haber limitado la *performance* en general. Subir únicamente el código del algoritmo no tuvo tanto impacto como en otros experimentos, siendo menor en el caso recursivo.

En comparación, mover el *stack* a BRAM genera mejoras que se hacen significativas a medida que la cantidad de procesadores aumenta. Estas mejoras suelen tener mayor impacto cuando también se sube el código. Para el resto de los casos, y en particular para la versión iterativa, la ubicación del *stack* no mostró diferencias significativas.

3.6. Resumen

En este capítulo hemos analizado los resultados de utilizar la BRAM de maneras distintas para cuatro algoritmos, algunos con más de una variante.

Para el caso general, notamos que utilizar la BRAM para mantener el *stack* o copiar y ejecutar parte del código de programa no provoca un impacto negativo en los tiempos totales de ejecución. Por el contrario, su uso suele generar mejoras en la mayoría de los casos.

Esto se debe principalmente a dos motivos. En primer lugar, acceder a BRAM mediante el LMB toma el mínimo tiempo posible, comparado con acceder a una línea de *cache* en la L2 (varios ciclos de *clock*, aún más si hay un *cache miss*). En segundo lugar, evitar el acceso a memoria principal alivia el uso de las *cache*, reduciendo la probabilidad de que los datos sean desplazados.

De todos los experimentos corridos, notamos que subir parte del código a BRAM es la mejor forma de reducir los tiempos. Hacer esto implica una menor constante de tiempo por unidad de trabajo (instrucciones ejecutadas más rápidamente) y una reducción en la competencia de accesos a memoria (los *fetch* de instrucciones de cada procesador no se hacen contra memoria principal). Por ambos motivos también permite paralelizar mejor.

Dependiendo del algoritmo, subir el código puede utilizar gran parte de la BRAM. Comparada con la memoria principal, la disponibilidad de BRAM es muy reducida (8 kB en nuestro caso). Subir el código del algoritmo de FFT iterativo, por ejemplo, utiliza alrededor de 2 kB de BRAM.

Observamos que subir el *stack* a BRAM es generalmente la segunda mejor forma de mejorar el rendimiento. Hacer esto, sin embargo, limita la BRAM disponible. Aunque la longitud del *stack* sea dinámica, debe reservarse una porción de la misma a la hora de diseñar la partición de memoria. Esta porción estará relacionada con el programa a ejecutar.

Los algoritmos que no hacen llamados a funciones o que poseen pocas variables no se beneficiarán de tener un *stack* en BRAM.

Por otro lado, subir las funciones de *spin lock* a la BRAM solo genera una mejora significativa cuando son llamadas muchas veces, o cuando hay varios procesadores activos que deban esperar a otros.

Se tratan de un caso especial de subir el código a BRAM, donde solo se usan unas pocas instrucciones. Esto indica que no es realmente necesario subir el algoritmo completo. Subir las partes de uso más frecuente puede ser suficiente para generar una mejora significativa.

Otro punto a mencionar es que hay que prestar atención a cómo se resuelven las funciones intrínsecas (en nuestro caso, las trigonométricas). En otras palabras, tenemos que ser conscientes de las capacidades del *softcore* que implementamos y asegurarnos de que el compilador esté al tanto de ellas y use la biblioteca más adecuada. Hacer esto puede eliminar las simulaciones por software innecesarias y producir una mejora significativa en el rendimiento del programa.

Las mejoras vistas se aplican a nuestro `Sistema` compuesto por `MicroBlaze` donde puede agregarse una BRAM con facilidad. Es posible que otros procesadores tengan una infraestructura nativamente *multi-core* que resulte en un mejor desempeño que nuestro caso donde armamos un SMP con varios `MicroBlaze`. También podrían tener otros recursos que no implementamos en nuestro `Sistema`, como instrucciones de exclusión mutua para *multi-core* (que `MicroBlaze` no soporta), o donde quizás agregar una BRAM no resulte tan beneficioso.

Capítulo 4

Conclusiones

Una solución informática se puede analizar separándola en niveles de abstracción: desde niveles bajos de lógica digital, pasando por componentes de hardware como procesadores, hasta niveles altos que comprenden complejas herramientas de software.

En este trabajo hicimos un recorrido por varios niveles de abstracción. Particularmente, implementamos procesadores en FPGA y pusimos a ejecutar código escrito en C sobre ellos, obteniendo métricas que nos permitieron comparar distintos escenarios de hardware y software.

Nuestro objetivo es avanzar en técnicas que permitan el uso eficiente de múltiples *softcores* dentro de un FPGA. Más específicamente, optamos por analizar distintas formas en la que estos *softcores* pueden utilizar la BRAM.

Las FPGAs son hardware programable. Algunas de ellas, como la utilizada en este trabajo, están implementadas en un chip junto con un procesador tradicional.

Propusimos y desarrollamos una infraestructura que permite medir el desempeño de varios algoritmos en un `Sistema` de procesamiento implementado en una FPGA. Dividimos esta infraestructura en un `Framework` de hardware y un conjunto de herramientas de software. La infraestructura se caracteriza por ser una solución genérica que permite testear distintos procesadores sin requerir mayores cambios para su puesta en marcha.

El `Framework` de hardware se destaca por los requerimientos del `Sistema` a testear, siendo estos un puerto AXI para acceder a memoria principal, y señales de `CLK` y `RST` para su ejecución. También mide y reporta tiempos de ejecución en términos de ciclos de *clock*.

Por otro lado, las herramientas de software desarrolladas permiten correr cualquier tipo de programa que sea soportado por el `Sistema` elegido, generando los valores de entrada para cada *test run* con la precisión requerida, registrando un conjunto de métricas relevantes para cada ejecución, y permitiendo validar su resultado.

Elegimos implementar como `Sistema` un SMP de cuatro *softcores* `MicroBlaze` conectados mediante ACE a una *cache* L2, con sincronización por software y siendo la DDR la única memoria compartida. Esta arquitectura de *softcores* se podría utilizar como parte de una solución de tipo acelerador o coprocesador.

Mediante los algoritmos "Multiplicación de matrices", *Heapsort*, y FFT (y algunas variantes interesantes de ellos), evaluamos la *performance* de tres formas de utilizar la BRAM bajo esta arquitectura (un total de seis combinaciones de uso): usar *stack* en BRAM, mover y correr código del algoritmo en BRAM, mover y correr ódigo de funciones *spin lock* en BRAM. Esto implica estar ejecutando procesos cuyos accesos a memoria utilizan distintas memorias, cada una con características distintas (BRAM y *Cache*+DDR).

Basándonos en que una ejecución tiene mejor *performance* que otra si la cantidad de ciclos de *clock* registrados durante su ejecución es menor, observamos que utilizar cualquiera

de estos usos de BRAM no induce un impacto negativo en la *performance*.

Respecto a las tres formas de uso de BRAM propuestas, notamos que mover y correr código de funciones *spin lock* en BRAM no genera mejoras significativas salvo en casos donde hay muchos llamados o varios procesadores en espera. De todas formas, esto nos da un indicio de que no es necesario subir el algoritmo completo: subir partes de uso frecuente puede ser suficiente para reducir los tiempos de ejecución sin invertir mucho espacio de BRAM.

Dependiendo del algoritmo y su implementación, usar el *stack* en BRAM puede ir desde tener poco efecto a tener un impacto significativo en el desempeño, por encima de mover los *spin-locks* a BRAM. En ciertos casos, el impacto puede ser menor porque el cuello de botella está en la ejecución de código. En casos con muchos procesadores, donde hay mayor probabilidad de colisiones en *cache*, subir el *stack* puede aliviarla.

Finalmente, mover y correr código del algoritmo en BRAM genera el mayor impacto positivo en la mejora de *performance*. Este impacto puede verse reducido si hay partes del código de uso frecuente que no se han movido a BRAM y continúan en memoria principal. Al tener además el *stack* en BRAM, ambas alternativas pueden generar una mejora significativa adicional a su uso independiente.

Al adentrarse en el mundo de las FPGAs, nuestras posibilidades se amplían. Tener control del hardware nos permite implementar soluciones que se ajusten al problema a resolver. Trabajar desde una capa de abstracción baja implica que los desarrolladores deben manejar detalles de muy bajo nivel, como los procesadores mismos, su interconexión, y la distribución de memoria utilizando distintos tipos de memoria.

Sacarles el máximo beneficio no será fácil. Es posible que en el futuro construyamos herramientas que nos permitan facilitar el desarrollo y el uso de estas tecnologías, o incluso unificar el desarrollo de software y hardware.

En cualquier caso, optimizar implica explorar estos detalles. En un software de aplicación esto incluye el software propio, el sistema operativo, el procesador. En una FPGA, se suman el hardware que compone el procesador, los *IP Cores*, los bloques disponibles en la FPGA misma.

Cuando se amplían las posibilidades también se amplía la complejidad.

4.1. Futuro trabajo

Continuando la línea de este trabajo, podríamos considerar el consumo de energía como una de las métricas a estudiar. Más específicamente, se podría medir la diferencia de energía modificando las distintas variables, analizando la relación entre el consumo y el desempeño obtenido.

Si bien en este trabajo la BRAM es aislada a cada `MicroBlaze`, se puede estudiar el caso de tener una o varios BRAMs compartidas entre varios *softcores*. Esto abriría la puerta al análisis de compartir datos en memoria rápida para algoritmos que se puedan beneficiar de ello.

Por otro lado, Vivado provee una librería de *IP Cores* que podrían permitir implementar una solución más eficiente. Uno de ellos es un Mutex que ayudaría a garantizar la exclusión mutua en multi-procesadores sin depender de la memoria principal. Se podrían evaluar los experimentos de este trabajo para un `Sistema` donde sea incluido. Su uso ayudaría a reducir los *cache misses* que puede generar un *spin-lock* al acceder a *flags* en memoria principal.

Alternativamente se puede utilizar el `Framework` presentado para evaluar el desempeño de algoritmos en distintos `Sistemas`, por ejemplo utilizando un `Sistema` implementando RISC-V en vez de un SMP con `MicroBlaze`.

Bibliografía

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution, in: Proceedings of the IEEE Symposium on Security and Privacy (SP), Institute of Electrical and Electronics Engineers, Inc, 2019, pp. 1–19. doi:10.1109/SP.2019.00002.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, R. Strackx, Meltdown: Reading kernel memory from user space, Communications of the ACM 63 (6) (2020) 46–56. doi:10.1145/3357033.

[3] Z. Hajduk, An fpga embedded microcontroller, Microprocessors and Microsystems 38 (1) (2014) 1–8. doi:10.1016/j.micpro.2013.10.004.

[4] R. Garcia-Ramirez, A. Chacon-Rodriguez, R. Molina-Robles, R. Castro-Gonzalez, E. Solera-Bolanos, G. Madrigal-Boza, M. Oviedo-Hernandez, D. Salazar-Sibaja, D. Sanchez-Jimenez, M. Fonseca-Rodriguez, J. Arrieta-Solorzano, R. Rimolo-Donadio, A. Arnaud, M. Miguez, J. Gak, Siwa: A custom RISC-V based system on chip (SOC) for low power medical applications, Microelectronics Journal 98 (2020) 104753. doi:10.1016/j.mejo.2020.104753.

[5] J. Echanobe, I. del Campo, K. Basterretxea, M. V. Martinez, F. Doctor, An fpga-based multiprocessor-architecture for intelligent environments, Microprocessors and Microsystems 38 (7) (2014) 730–740. doi:10.1016/j.micpro.2014.07.005.

[6] S. Nolting, G. Payá-Vayá, F. Giesemann, H. Blume, S. Niemann, C. Müller-Schloer, Dynamic self-reconfiguration of a mips-based soft-core processor architecture, Journal of Parallel and Distributed Computing 133 (2019) 391–406. doi:10.1016/j.jpdc.2017.09.013.

[7] X. Liu, H. Allah Ounifi, A. Gherbi, Y. Lemieux, W. Li, A hybrid GPU-FPGA-based computing platform for machine learning, Procedia Computer Science 141 (2018) 104–111, the 9th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2018) / The 8th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2018) / Affiliated Workshops. doi:10.1016/j.procs.2018.10.155.

[8] E. Matthews, L. Shannon, A. Fedorova, Polyblaze: From one to many. bringing the microblaze into the multicore era with linux smp support, in: Proceedings of the 22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Institute of Electrical and Electronics Engineers, Inc, 2012, pp. 224–230. doi:10.1109/FPL.2012.6339185.

[9] ARM, AMBA AXI and ACE Protocol Specification, last checked on 2 de octubre de 2023(2013).
URL https://developer.arm.com/documentation/ihi0022/e/

[10] M. Bečvář, T. Brabec, B10: Implementation of configurable system on a chip with microblaze processor core, IFAC Proceedings Volumes 37 (20) (2004) 222–227, proceedings of the IFAC Workshop on Programmable Devices and Systems - PDS 2004, Cracow, Poland, November 18-19, 2004. doi:10.1016/S1474-6670(17)30600-6.

[11] M. Cvikl, A. Zemva, Fpga-oriented hw/sw implementation of ecg beat detection and classification algorithm, Digital Signal Processing 20 (1) (2010) 238–248. doi:10.1016/j.dsp.2009.05.008.

[12] M. Makni, M. Baklouti, S. Niar, M. W. Jmal, M. Abid, A comparison and performance evaluation of fpga soft-cores for embedded multi-core systems, in: Proceedings of the 11th International Design Test Symposium (IDT), Institute of Electrical and Electronics Engineers, Inc, 2016, pp. 154–159. doi:10.1109/IDT.2016.7843032.

[13] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, R. W. Stewart, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc, Strathclyde Academic Media, Glasgow, GBR, 2014.

[14] R. P. Weicker, Dhrystone: A synthetic systems programming benchmark, Commun. ACM 27 (10) (1984) 1013–1030. doi:10.1145/358274.358283.
URL https://doi.org/10.1145/358274.358283

[15] S. M. Trimberger, Three ages of fpgas: A retrospective on the first thirty years of fpga technology, Proceedings of the IEEE 103 (3) (2015) 318–331. doi:10.1109/JPROC.2015.2392104.

[16] M. Turki, M. Abid, Z. Marrakchi, H. Mehrez, Routability driven placement for mesh-based fpga architecture, in: Proceedings of the 2010 5th International Design and Test Workshop, Institute of Electrical and Electronics Engineers, Inc, 2010, pp. 85–90. doi:10.1109/IDT.2010.5724414.

[17] Z. Marrakchi, M. Hayder, U. Farooq, H. Mehrez, Fpga interconnect topologies exploration, International Journal of Reconfigurable Computing 2009 (2009) 259837, selected Papers from ReCoSoc 2008. doi:10.1155/2009/259837.

[18] AMD Xilinx, MicroBlaze Processor Reference Guide, uG984 (May 2022).
URL https://docs.xilinx.com/v/u/en-US/ug984-vivado-microblaze-ref

[19] J. Blanchard, Microblaze configuration for an rtos, last checked on 2022-11-12 (Feb. 2019).
URL https://www.jblopen.com/microblaze-configuration-part-1/

[20] Xilinx, System Cache v4.0 - LogiCORE IP Product Guide, last checked on 2 de octubre de 2023(Apr. 2017).

[21] F. Franchetti, M. Puschel, Y. Voronenko, S. Chellappa, J. M. Moura, Discrete fourier transform on multicore, IEEE Signal Processing Magazine 26 (6) (2009) 90–102. doi:10.1109/MSP.2009.934155.