UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica
Máster Universitario en Computación Grid y Paralelismo

TRABAJO FIN DE MÁSTER

Aceleración de cálculos para la selección de genes en la clasificación del cáncer mediante paralelismo y FPGAs

José Luis Cerrada Barrios
Julio, 2015

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica
Máster Universitario en Computación Grid y Paralelismo

TRABAJO FIN DE MÁSTER

Aceleración de cálculos para la selección de genes en la clasificación del cáncer mediante paralelismo y FPGAs

Autor: José Luis Cerrada Barrios
Fdo.:

Director: Juan Antonio Gómez Pulido
Fdo.:

Tribunal Calificador:

Presidente:
Fdo:

Secretario:
Fdo:

Vocal:
Fdo:

CALIFICACIÓN:

FECHA:

Índice.

Índice de Figuras.

Índice de Tablas.

1 Resumen.

El cáncer es una enfermedad altamente variable que presenta cambios genéticos y epigenéticos heterogéneos. Es por ello que los estudios funcionales son esenciales para la comprensión de la complejidad y el polimorfismo de esta enfermedad. En la actualidad se utilizan diferentes metodologías a la hora de detectar y clasificar el cáncer. No obstante, en los últimos años ha cobrado especial popularidad el uso de *microarrays de expresión génica*, sobre todo en artículos de investigación científica [Perez2000] [Wang2005] [Russo2003], debido en gran parte a los buenos resultados que aporta. Se trata de una nueva herramienta para el estudio de las bases moleculares a una escala a la que no sería posible llegar utilizando el análisis convencional. Esta técnica hace posible el estudio de miles de genes de forma simultánea.

El uso de *microarrays de expresión génica* presenta, no obstante, varios problemas, siendo el principal de ellos la necesidad de seleccionar el pequeño subconjunto de genes que contribuye a la enfermedad de entre los miles de genes analizados en el microarray, que no constituyen sino ruido [Russo2003]. La mayor parte de los trabajos anteriores aborda este problema de forma manual, lo cual resulta tedioso e ineficiente.

El objetivo de este proyecto es mejorar el tiempo de computación en la automatización de la selección de un pequeño, pero relevante, subconjunto de genes presentes en células cancerosas, de entre el vasto conjunto de genes que componen el *microarray de expresión génica*. Esta automatización utiliza un algoritmo genético recursivo, en el que el cálculo del *fitness* de cada individuo puede ser acelerado utilizando circuitos de diseño específico mediante hardware reconfigurable o *FPGA*, disminuyendo así el tiempo necesario para realizar la clasificación.

Los resultados experimentales obtenidos demuestran que el uso de este subconjunto de genes representativos produce una clasificación más precisa que la obtenida con los métodos clásicos. Además, la inclusión de una FPGA como elemento hardware de aceleración ha contribuido a reducir apreciablemente el tiempo necesario para realizar dicha clasificación, al comparar con el uso de los modernos procesadores de propósito general.

2 Introducción.

2.1 Ámbito del problema.

El cáncer constituye una de las principales causas de muerte de la sociedad actual. Sólo en España, en 2012[1] se diagnosticaron 215.534 casos nuevos y hubo 102.762 muertes a causa de dicha enfermedad. Teniendo en cuenta que el total de defunciones en España durante el mismo periodo de tiempo fue de 402.950 [Ministerio2015] personas, llegamos a la conclusión de que 1 de cada 4 muertes en nuestro país tiene su causa en el cáncer. Si miramos más allá de nuestras fronteras, las cifras no mejoran; en ese mismo año se registraron 8.2 millones de muertes a nivel mundial a causa del cáncer y se diagnosticaron 14.1 millones de casos nuevos [CancerUK2015].

Si bien es cierto que el origen del cáncer es una alteración en el funcionamiento normal de las células que las lleva a multiplicarse de forma descontrolada, también es cierto que dichas alteraciones pueden verse propiciadas por factores externos, tales como: nuestro estilo de vida, entorno medioambiental en el que nos encontramos, etc.

Con el paso de los años han ido surgiendo cada vez más estudios en los que se hace referencia a cómo ciertos cambios en nuestro estilo de vida podrían ayudar a prevenir esta enfermedad [Anand2008] [Martínez2005]; dichos cambios pasan por seguir una dieta saludable, hacer ejercicio de forma regular, consumir alcohol de forma moderada y no fumar, entre otros. Sólo para que nos hagamos una idea acerca de la importancia de seguir estas recomendaciones, se estima que el 90% de las muertes por cáncer de pulmón en hombres y entre el 75%-80% en mujeres, tiene su origen en el tabaco; si tenemos en cuenta que este tipo de cáncer es el responsable de una cuarta parte de las muertes por esta enfermedad, entonces nos hacemos una idea de cuán importante es nuestro papel en la prevención de esta alteración celular.

Ahora bien, la prevención no siempre es suficiente; muchas veces no podemos hacer nada al respecto, pues el cáncer es una enfermedad de origen idiopático de la cual sólo conocemos los factores de riesgo, pero no el motivo de dichas alteraciones celulares. En estos casos un diagnóstico precoz y un tratamiento personalizado para cada paciente son claves para la superación de la enfermedad.

2.2 Un enfoque computacional al problema.

Muchas son las técnicas biomédicas que abordan el problema de la prevención y detección precoz del cáncer. Además, lo habitual es que estas técnicas se apoyen en modelos matemáticos e ingentes cantidades de datos que requieren, a su vez, de complejos algoritmos para su tratamiento computacional.

Una de las vías de estudio más importantes se centra en la detección y clasificación del cáncer. En este ámbito, nace el proyecto que sustenta el presente

[1] Corresponde al último año del que se tienen cifras oficiales.

Trabajo fin de Máster (**TFM**): crear un entorno hardware/software que ayude a detectar y clasificar el cáncer de forma rápida y precisa. El objetivo final es mejorar herramientas que permiten a los médicos estar en condiciones de proporcionar a cada paciente el tratamiento que mejor se ajuste a su condición, mejorando así las expectativas de éxito, que no son otras que la supervivencia del enfermo.

Son muchas las metodologías hasta ahora utilizadas para la detección y clasificación del cáncer. Nosotros nos hemos basado en una que ha adquirido cierta popularidad en los últimos años y que ofrece muy buenos resultados en estos aspectos: el uso de **microarrays de expresión génica**. La característica más destacada de esta tecnología es que permite analizar miles de genes de forma simultánea y además extraer gran cantidad de información de ellos [Ambroise2002][Mohamad2007]. Dicha información puede ser posteriormente analizada por computadores que, en base a ciertos marcadores, determinarán si el tejido o genes analizados tienen cáncer o no [Ryu2002]. Esto, además de facilitar, automatizar y simplificar el proceso de detección del cáncer, también elimina la subjetividad en la interpretación de los resultados por parte del facultativo.

Por otro lado, el uso de esta tecnología presenta ciertos inconvenientes, entre los cuales destaca la elevada cantidad de información a analizar, con el añadido de que la mayor parte de los genes procesados no constituyen sino ruido, es decir, no aportan valor al resultado final y lo único a lo que contribuyen es a ralentizar el proceso de análisis. Como solución a este problema se han publicado multitud de artículos en el pasado reciente que proponen el uso de *Algoritmos Genéticos* (**AGs**) y otras heurísticas basadas en la Computación Bioinspirada, como herramienta para reducir la cantidad de genes a analizar [Huerta2006][Mohamad2008][Mohamad2009].

2.3 Planteamiento del trabajo.

El objetivo de este proyecto es construir una herramienta que, dado como entrada un microarray de expresión génica, determine qué genes son los más representativos a la hora de detectar y clasificar un determinado tipo cáncer. Para ello dividiremos los datos de entrada en una serie de cromosomas (array de genes) a los que le aplicaremos una función fitness, que nos indicará cómo de importantes son esos genes a la hora de detectar un determinado cáncer. Estos datos podrán ser posteriormente utilizados para construir sistemas de diagnóstico y clasificación de cáncer automatizados.

La aportación de nuestro trabajo no proviene del tratamiento algorítmico en sí mismo, que no es especialmente novedoso, pues se basa en técnicas ya publicadas en la literatura científica. Existen estudios previos que ya han hecho uso de algoritmos genéticos con objeto de discernir qué genes son relevantes y cuáles no para la detección del cáncer. De hecho, este trabajo tiene sus bases en varios de ellos. Lo que sí podemos aportar en este trabajo es una primera aproximación a la aceleración computacional de ciertos cálculos aritméticos que están involucrados en los algoritmos de clasificación de genes, y que son los que originan el mayor coste computacional, pues son operaciones muy repetitivas en el algoritmo. Para esta aceleración diseñaremos circuitos específicos programados mediante lenguajes de descripción hardware de alto nivel (HDLs) e implementados en hardware mediante la tecnología de la *Computación Reconfigurable* (**CR**), que se basa en los dispositivos *Field Programmable Gate Arrays* (**FPGA**).

2.3.1 Aceleración de la función de fitness.

Se ha comprobado que en la mayoría de ocasiones la parte más costosa (en tiempo de computación) de un algoritmo genético es el cálculo de la función de aptitud o fitness de cada uno de los elementos de la población [Baraglia2001] [Emam2003] [Hidalgo2014] [Mostafa2004]. Por tanto, si lo que pretendemos es mejorar el rendimiento y la eficiencia del algoritmo, lo más lógico sería comenzar con esta parte (acelerar el cálculo de la función de fitness). La aceleración de la función de fitness es muy importante, ya que esta función se evalúa numerosas veces dentro de una misma generación del algoritmo (tantas como individuos tenga la población) y suele presentar cálculos aritméticos en la mayoría de los casos.

La idea de utilizar la tecnología basada en FPGAs como herramienta aceleradora, se debe a la naturaleza altamente paralelizable del cálculo de la función fitness en la población, ya que se trata de una función que hay que aplicar a todos y cada uno de los individuos de la población en una generación dada, y cuyos resultados son independientes entre sí. Este hecho hace que podamos aprovechar las cualidades del paralelismo real que proporciona una FPGA para calcular en paralelo el fitness de toda la población, o al menos hacerlo por subconjuntos de la misma, reduciendo así el tiempo necesario para diagnosticar y clasificar el cáncer. Por otro lado, la paralelización de las operaciones aritméticas propias de la misma función de fitness permite incrementar el rendimiento global, al tratarse de un paralelismo "de grano fino". Es en este nivel del paralelismo donde hemos centrado los esfuerzos del diseño hardware.

2.3.2 Algoritmo de optimización.

Como ya hemos comentado anteriormente, este proyecto tiene sus bases en varios artículos científicos en los que se estudia cómo obtener los genes más significativos indicadores de la existencia de un determinado tipo de cáncer a partir de una muestra de tejido mapeada en forma de microarray de expresión génica, cuyo tamaño imposibilita o dificulta la consecución de esta tarea de forma manual.

Para lograr este cometido, estos estudios proponen una combinación de Algoritmos Genéticos y clasificadores del tipo Support Vector Machine (SVM). De esta forma, mediante el AG se selecciona un subconjunto de genes que posteriormente el clasificador SVM evalúa durante el proceso de clasificación. En la Figura 1 puede observarse el flujo de ejecución completo de este proceso.

Partiendo de un microarray de expresión génica, se selecciona un número de genes, *ns*, del conjunto total de genes *M*. A partir de los genes seleccionados se genera una población inicial donde cada individuo de la población (cromosoma) representa un subconjunto de genes de *ns*. A continuación se aplica la función fitness sobre cada uno de estos cromosomas y se selecciona un determinado número de ellos con una probabilidad proporcional a su fitness. Sobre los individuos seleccionados se aplica una serie de cruces, que dan como resultado un conjunto de cromosomas nuevos sobre los que se aplica una mutación en base a una probabilidad determinada. A continuación se sustituyen aquellos individuos menos aptos de la población (con el valor de fitness más bajo) por los que han surgido como resultado de los cruces. Llegados a este punto, si el subconjunto de genes obtenidos es óptimo, damos por terminado el proceso de búsqueda; si por el contrario dicho subconjunto no es suficientemente bueno, volvemos a comenzar el mismo proceso desde el punto de selección de cromosomas para su cruce.

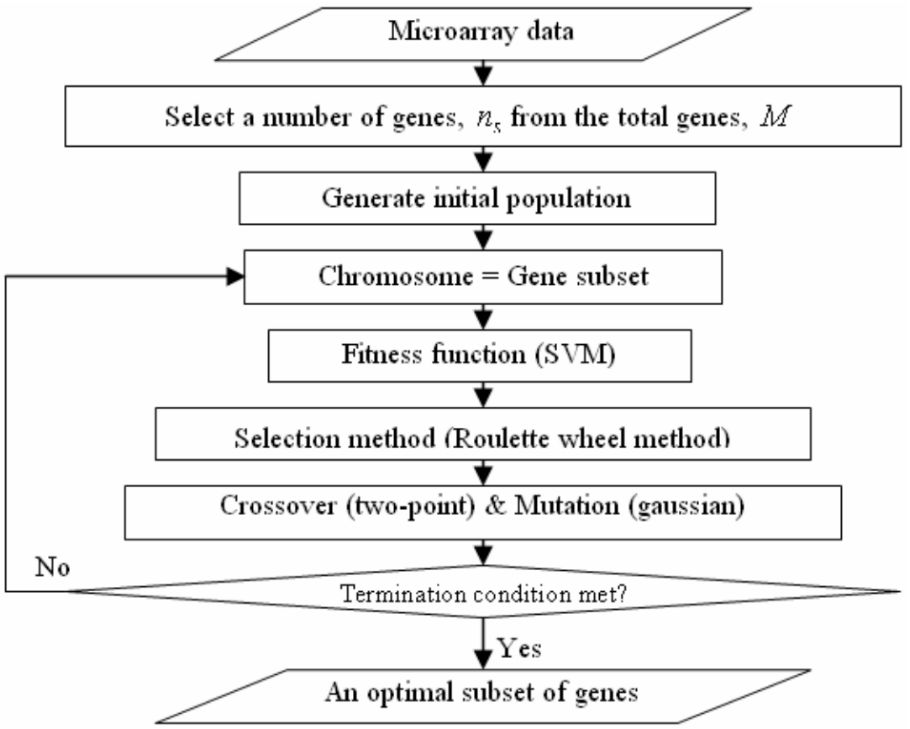

Figura 1. Flujo de ejecución del algoritmo de optimización.

Como podemos observar, el proceso descrito anteriormente posee la estructura de cualquier algoritmo genético, donde podemos encontrar las siguientes fases:

(1) Generación de la población inicial.

(2) Evaluación de cada uno de los individuos de la población para obtener su fitness.

(3) Selección de los individuos que van a formar los cruces.

(4) Recombinación (cruce) de los individuos seleccionados y mutación de los individuos resultantes.

(5) Reemplazo de los individuos menos aptos por los nuevos y vuelta al punto 2 hasta que se cumpla la condición de término.

Este es, a grandes rasgos, el algoritmo que pretendemos implementar en nuestro proyecto. No obstante, y para una mejor comprensión de los resultados obtenidos, en el capítulo "Programación de un algoritmo genético" explicamos de manera más detallada y punto por punto las peculiaridades de cada una de las fases que componen nuestro algoritmo genético, así como la estructura de cada cromosoma.

3 Circuito acelerador.

En este capítulo se expone el diseño de un circuito para la aceleración del cómputo de la función de fitness, utilizando la tecnología basada en hardware reconfigurable.

3.1 Motivación.

Como se ha mencionado anteriormente, se ha investigado mucho sobre metodologías computacionales que, partiendo de un microarray de expresión génica, tratan de determinar qué genes son los más representativos respecto a la detección y clasificación de un determinado tipo de cáncer.

Algunos de estos algoritmos utilizan técnicas basadas en la computación bioinspirada o computación evolutiva, donde un conjunto de individuos (población) evoluciona a lo largo de varias generaciones. En las diversas técnicas algorítmicas bioinspiradas siempre hay un elemento común: la evaluación de la función de fitness a cada individuo de la población. Así, siempre encontraremos una operación repetitiva, tanto dentro de la población como en las sucesivas generaciones, que consiste en aplicar la misma función de fitness a cada individuo, con el propósito de obtener un índice de la importancia de los genes a la hora de detectar un determinado tipo de cáncer.

Es en esta operación repetitiva donde se puede obtener provecho al paralelizar sus tareas, ya que en la misma unidad de tiempo que tarda en computarse la función de fitness, se puede evaluar el fitness de un determinado número de individuos (tantos como el área de la FPGA permita para el circuito paralelo).

Podemos paralelizar no solo la evaluación del fitness de los individuos de la población (aplicar en paralelo la función de fitness a varios individuos), sino también las propias operaciones aritméticas internas de la función de fitness (lo que constituye un paralelismo de "grano fino"). Esta paralelización en varios niveles multiplica el posible rendimiento, que también se multiplica por la sucesión de generaciones. Esto, a priori, induce a pensar que podría esperarse una aceleración de los algoritmos dedicados a la detección de genes. Esta hipótesis es la que ha motivado el trabajo de modelación e implementación de la función de fitness para ser ejecutada en una FPGA.

Numerosos trabajos han probado la adecuación de acelerar la función de fitness mediante FPGAs. Como se mencionó anteriormente, es habitual que la evaluación del fitness consuma la mayor parte del tiempo de computación de un algoritmo evolutivo, por lo que esta evaluación sería prioritaria a la hora de desarrollar circuitos aceleradores.

3.2 Tecnologías utilizadas.

3.2.1 Hardware reconfigurable.

La computación reconfigurable es un área de la computación que combina parte de la flexibilidad que proporcionan las soluciones software con el alto rendimiento de las soluciones puramente hardware, todo ello mediante el uso de los dispositivos de lógica reconfigurable tipo field-programmable gate arrays (FPGAs).

Tradicionalmente han existido dos opciones a la hora de resolver un problema computacional; a nivel hardware (Very-large-scale integration, VLSI, application-specific integrated circuit, ASIC, gate-arrays...) y a nivel software (DSPs, microcontroladores, microprocesadores embebidos o de propósito general...). La introducción de los dispositivos FPGA hace aproximadamente 20 años dio lugar al nacimiento de lo que se conoce como Computación Reconfigurable. Si bien las primeras generaciones de FPGAs fueron bastante limitadas en sus capacidades, en la actualidad las más modernas tecnologías de fabricación de circuitos integrados permiten disponer de dispositivos no sólo con millones de puertas de lógica programable, sino también con recursos hardware específicos, con una amplísima gama de soluciones de conectividad, haciendo posible el diseño de complejos y potentes sistemas en un único chip (concepto SOC, System On a Chip) y permitiendo un gran salto hacia delante en la capacidad para procesar datos a mayor velocidad y con menor coste.

El hardware reconfigurable ofrece un compromiso entre los circuitos integrados de aplicación específica (ASICs) y los procesadores de propósito general (GPPs). Como los ASICs, las FPGAs implican implementación hardware y con ello paralelismo y alta capacidad de procesamiento digital; como los GPPs, proporcionan reconfigurabilidad y, en consecuencia, flexibilidad y rapidez de prototipado. Su principal diferencia en comparación con el uso de microprocesadores ordinarios es la capacidad de realizar cambios significativos en el propio datapath, así como en el flujo de control. Por otro lado, su diferencia más significativa en relación al hardware personalizado, es decir, con respecto a los circuitos integrados de aplicación específica (ASIC), es la posibilidad de adaptar el hardware en tiempo de ejecución mediante la "carga" de un nuevo circuito en el dispositivo reconfigurable.

La estructura interna de las FPGAs las convierte en dispositivos perfectamente adecuados para realizar en paralelo tareas elementales de procesamiento digital, lo cual constituye el principal motivo de haberlas seleccionado en este proyecto como herramienta aceleradora de nuestro algoritmo.

Los dispositivos basados en FPGAs ofrecen un rendiemiento bastante notorio [DeHon1999][Ashenden1990], multiplicando en muchos casos por más de 100 el rendimiento de las soluciones puramente software. Por otro lado, si comparamos dicho rendimiento con el que ofrecen las soluciones hardware, entonces ya no sale tan bien parado, lo cual tampoco supone una sorpresa, ya que se trata de un hardware que ha sido específicamente diseñado para resolver un problema en concreto y sólo uno, al contrario de lo que sucede con las FPGAs, que deben ofrecer una mayor flexibilidad.

En las siguientes figuras se muestra cómo cada una de las tecnologías comentadas anteriormente puede ser utilizada para resolver un mismo problema. Para ello utilizaremos la siguiente ecuación de segundo grado como problema a resolver: $y = Ax^2 + Bx + C$ [DeHon1999].

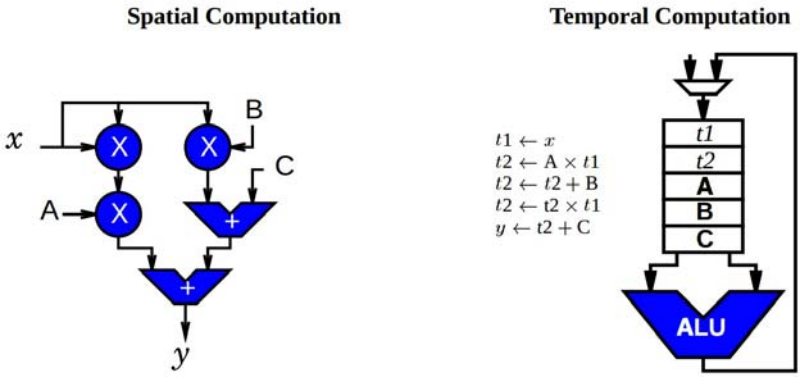

Figura 2. Computación espacial frente a computación temporal para la expresión $y = Ax^2 + Bx + C$

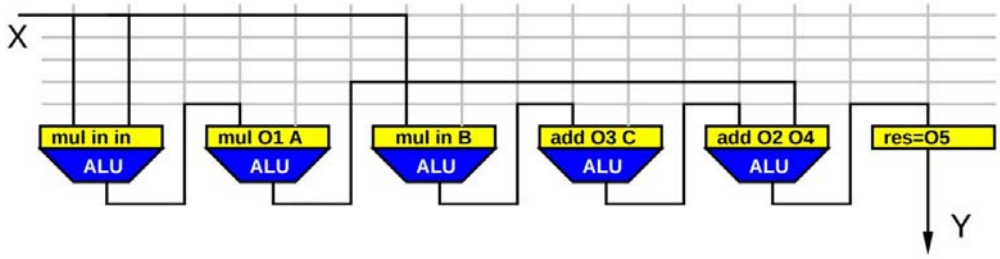

Figura 3. Implementación espacialmente configurable de la expresión $y = Ax^2 + Bx + C$

La Figura 2 muestra las diferencias entre computación espacial y temporal. En la computación espacial, cada operador existe en un punto diferente del espacio, explotando al máximo el paralelismo inherente a ciertos problemas, aumentando el rendimiento y disminuyedo la latencia. Por otro lado, en las implementaciones temporales, un pequeño número de de recursos computacionales más generales son reutilizados en el tiempo, permitiendo una implementación del problema más compacta.

Además, se muestra que cuando sólo tenemos estas dos opciones, implícitamente conectamos procesamiento espacial con computación espacial y procesamiento temporal con computación software. El beneficio clave de las FPGAs, y más ampliamente de los dispositivos reconfigurables, es que introducen una clase de dispositivos configurables post-fabricación que soportan computación espacial, lo cual nos da una nueva estructura organizacional en este espacio. Por otro lado, la Figura 3 muestra una arquitectura espacialmente configurable que puede ser comparada con las mostradas en la Figura 2. Los dispositivos reconfigurables tienen el claro beneficio del paralelismo espacial, permitiendo llevar a cabo más operaciones por ciclo. De hecho, a menudo las FPGAs ofrecen mayor capacidad de cómputo que muchos procesadores modernos debido a sus altas posibilidades de paralelización.

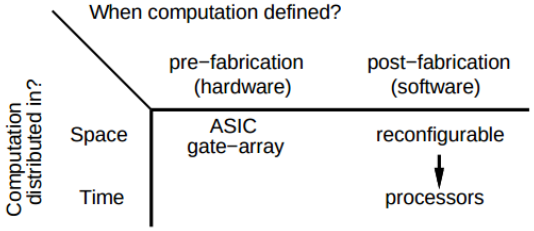

Figura 4. Estructura hardware según el tipo de computación y el momento de su diseño.

3.2.2 Lenguajes de descripción hardware: VHDL y Handel-C.

A la hora de programar una FPGA tenemos varias opciones, siendo las más conocidas y usadas las siguientes: VHDL y Verilog a un nivel de abstracción bajo, más cercano al "lenguaje máquina", y lenguajes HDL basados en C a un nivel de abstracción más alto. Para el desarrollo de este proyecto utilizaremos dos de estos lenguajes, uno de cada nivel: VHDL y Handel-C. El primero de ellos lo utilizaremos para la creación de los distintos componentes que conformarán el circuito (multiplicadores, sumadores...), así como para definir las conexiones entre los mismos, mientras que el segundo lo utilizaremos para implementar el propio algoritmo de cálculo del fitness de los individuos de una población. Podríamos haber implementado el algoritmo al completo en la FPGA utilizando un único lenguaje de programación (VHDL); sin embargo, la implementación e Handel-C proporcionaba una via más rápida para el prototipado, donde principalmente nos interesaba saber si el rendimiento es susceptible de ser acelerado en relación a un procesador de propósito general. Así, decidimos aprovechar el mayor nivel de abstracción que proporciona Handel-C para implementar la parte más compleja del problema, que no es otra que la lógica del mismo (función fitness, la única que se pretendía implementar en la FPGA).

VHDL

VHDL es un lenguaje de descripción hardware utilizado para describir sistemas electrónicos digitales. Surgió del programa ***Very High Speed Integrated Circuits*** (**VHSIC**) del Gobierno de los Estados Unidos, que se inició en 1980. Durante el transcurso de este programa, quedó claro que existía la necesidad de un lenguaje estándar para la descripción de la estructura y función de los circuitos integrados (**ICs**). Fue entonces cuando se inició el desarrollo de **VHDL** (***VHSIC Hardware Description Language***), que posteriormente sería adoptado como un estándar por el **IEEE** (***Institute of Electrical and Electronics Engineers***) en los EE.UU..

Un sistema electrónico digital puede ser descrito como un módulo con entradas y/o salidas. Los valores eléctricos en las salidas son el resultado de la aplicación de alguna función a los valores de entrada. La Figura 5(a) muestra un ejemplo de este punto de vista en un sistema digital. El módulo F tiene dos entradas, A y B, y una salida Y. Utilizando la terminología VHDL, llamamos al módulo F entidad (***entity***) del diseño, y a las entradas y salidas se las denomina puertos (***ports***).

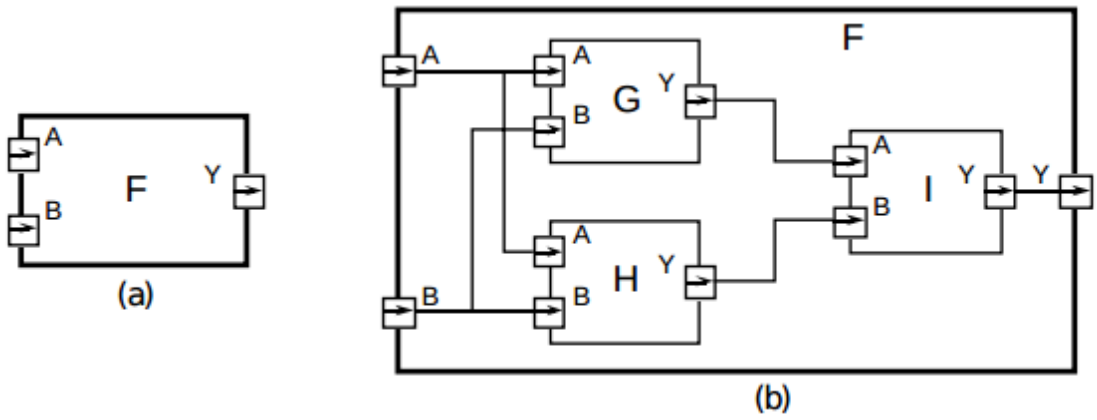

Figura 5. Diseño a alto (**a**) y bajo (**b**) nivel de una entidad **F**.

Una forma de describir la función de un módulo es describir el modo en que se compone de sub-módulos. Cada uno de los sub-módulos es una instancia (***instance***) de una entidad, y los puertos de dichas instancias están conectados mediante señales (***signals***). La Figura 5(b) muestra cómo la entidad F podría estar compuesta por instancias de las entidades *G, H* e *I*. A este tipo de descripción se le denomina descripción estructural (***structural description***). A su vez, cada una de las entidades *G, H* e *I* también podría estar compuesta de instancias de otras entidades, es decir, tener una descripción estructural propia.

Sin embargo, en muchos casos no es apropiado describir un módulo estructuralmente. Uno de estos casos podría ser el de un módulo que está en la parte inferior de la jerarquía de alguna otra descripción estructural. Por ejemplo, si estamos diseñando un sistema que utiliza paquetes de circuitos integrados que han sido comprados, no es necesario describir su estructura interna. En dichos casos, lo que se requiere es una descripción de la función realizada por el módulo, sin referencia a su estructura interna real. Tal descripción se denomina descripción funcional o conductual (***functional*** o ***behavioural description***).

Para ilustrar esto, supongamos que la función de la entidad F en la Figura 5(a) es la función exlusive-or (xor). Una descripción del comportamiento de F podría ser la función booleana siguiente:

$$F = \neg A \cdot B + A \cdot \neg B$$

Hay comportamientos más complejos que no pueden ser descritos simplemente como una función de entradas. En sistemas con retroalimentación, las salidas también dependen del tiempo. VHDL resuelve este problema permitiendo la descripción de comportamientos (***behaviours***) en forma de programa ejecutable.

Una vez se han especificado la estructura y el comportamiento de un módulo, es posible simular su funcionamiento mediante la ejecución de su descripción comportamental (***behavioural description***). Esto se consigue mediante la simulación del paso del tiempo en pasos discretos. En cierto momento de la simulación, la entrada de un módulo puede ser estimulada cambiando el valor de alguno de sus puertos de entrada. El módulo reacciona ejecutando el código de su descripción comportamental y programando nuevos valores que serán asignados a las señales conectadas a sus puertos de salida en algún punto posterior de la simulación. Esto se conoce como planificación de una transacción (***transaction***) en dicha señal. Si el nuevo valor de la señal es diferente al que tenía previamente, se produce un evento (***envent***), pudiendo dar lugar a la activación de otros módulos cuyos puertos de entrada se hallen conectados a esta señal.

La simulación comienza con una fase de inicialización, y luego continúa mediante la repetición de un ciclo de simulación de dos etapas. En la fase de inicialización, todas las señales se resetean a sus valores iniciales, el tiempo de simulación se pone a cero, y se ejecuta el programa comportamental de cada uno de los módulos. Esto normalmente resulta en la planificación de transacciones sobre las señales de salida para algún punto posterior de la simulación.

En la primera etapa de cada ciclo de simulación, el tiempo se hace avanzar al punto más cercano sobre el que una transacción ha sido planificada. Todas las transacciones programadas para ese momento se ejecutan, lo que puede provocar ciertos eventos en algunas señales.

En la segunda etapa, todos los módulos que reaccionan a los eventos producidos en la primera etapa proceden a ejecutar su programa comportamental. Estos programas suelen planificar nuevas transacciones en sus señales de salida. Cuando todos los programas han finalizado su ejecución, el ciclo de simulación se repite de nuevo, volviendo a la primera etapa. La simulación finaliza cuando no hay más transacciones programadas.

El objetivo de la simulación es reunir información sobre los cambios en el estado del sistema a lo largo del tiempo. Esto se puede conseguir mediante la ejecución de la simulación bajo el control de un monitor de simulación. El monitor permite que las señales y otra información de estado puedan ser vistos o almacenados en un log para su posterior análisis. También permitir interactuar paso a paso durante el proceso de simulación, al igual que los depuradores de otros lenguajes de programación.

Tipos de datos

El tipo de un objeto identifica los valores que dicho objeto puede tomar. Cuando se asigna un valor a una señal, dicho valor se comprueba para estar seguro de que se encuentra dentro del conjunto de valores permitido. Si no es así, se emite un mensaje de error. Esto es particularmente útil para los valores asignados a partir de un cálculo aritmético. VHDL proporciona dos tipos de valores: escalares y compuestos. [Ashenden1990][VHDL1995]

Tipos escalares

Los tipos escalares (sin estructura) incluyen números, enumerados y tipos de objetos físicos. Estos tipos se componen de números reales, enteros, cantidades con unidades físicas asociadas tales como tiempos y objetos que se componen de literales o identificadores.

Integer

Los integers son el conjunto no acotado de números enteros positivos y negativos.

- Formato:
  - `TYPE type_name IS RANGE int_range_constraint;`
- Ejemplo:
  - `TYPE day IS RANGE 1 TO 31;`

Floating point

Los floating point son el conjunto no acotado de números positivos y negativos, que contienen un punto decimal.

- Formato:
  - `TYPE type_name IS RANGE int_range_constraint;`
- Ejemplo:
  - `TYPE real IS RANGE -1.79769E308 TO 1.79769E308;`

Enumeration

Lista de identificadores o literales.

- Formato:
  - `TYPE type_name IS (enumeration_ident_list);`

- Ejemplos:
  - `TYPE bit IS (`0','1');`
  - `TYPE boolean IS (false,true);`
  - `TYPE character IS (`a','b','c',...);`

Physical

Describe los objetos en términos de una unidad base, múltiplos de dicha unidad, y en un rango especificado.

- Formato:
  -
```
TYPE type_name IS RANGE range_constraints
UNITS base_unit;
       [ -- multiples;]
END UNITS;
```
- Ejemplo:
  -
```
TYPE time IS RANGE -2**(31-1) TO 2**(31-1)
UNITS
       fs; --femtosecond =10-15sec
       ps = 1000 fs; --picosecond =10-12sec
       ns = 1000 ps; --nanosecond =10-9sec
       us = 1000 ns; --microsecond =10-6sec
       ms = 1000 us; --millisecond =10-3sec
       sec = 1000 ms; --second
       min = 60 sec; --minute
       hr = 60 min; --hour
END UNITS;
```

Tipos compuestos

Hay dos clases de tipos de compuestos: arrays y registros.

Array

Múltiples valores de un mismo tipo bajo único identificador y que puede tener una o más dimensiones.

- Formato:
  - `TYPE array_type_name IS ARRAY (range_constraints) OF type;`
- Ejemplos:
  - `TYPE string IS ARRAY (positive RANGE <>) OF character;`
  -
```
TYPE Column IS RANGE 1 TO 80;
TYPE Row IS RANGE 1 TO 24;
TYPE Matrix IS ARRAY (Row,Column) OF boolean;
```

Record

Los registros son tipos compuestos heterogéneos; es decir, los elementos de un registro pueden ser de varios tipos. Una definición de un registro especifica uno o más elementos, donde cada elemento tiene un nombre diferente y, posiblemente, un tipo distinto.

- Formato:
  - RECORD
    element_declaration
    {element_declaration}
    END RECORD;
- Ejemplo:

```
TYPE Opcode IS (Add,Add_with_carry,Sub,Sub_with_carry);
TYPE Address IS RANGE 16#0000# TO 16#FFFF#;
TYPE Instruction IS
RECORD
       Op_field :Opcode;
       Operand_1 :Address;
       Operand_2 :Address;
END RECORD;
```

HANDEL-C

Handel-C es un lenguaje de programación diseñado para permitir la implementación de programas en hardware síncrono. Handel-C no es un lenguaje de descripción de hardware, sino más bien es un lenguaje de programación orientado a la compilación de algoritmos de alto nivel directamente en hardware a nivel de puertas lógicas. Al igual que con los lenguajes de alto nivel convencionales, Handel-C está diseñado para permitir la implementación de algoritmos sin la necesidad de tener que entender cómo funciona el hardware interno que lo ejecuta. Esta filosofía hace de Handel-C un lenguaje de programación en lugar de un lenguaje de descripción de hardware. En algunos sentidos, Handel-C es al hardware lo que los lenguajes de alto nivel convencionales son al lenguaje ensamblador. Es importante tener en cuenta que el diseño hardware que Handel-C produce es exactamente el hardware especificado en el código fuente. No hay ninguna capa intermedia como ocurre en el caso del lenguaje ensamblador cuando se programa para microprocesadores de propósito general.

Al igual que cualquier otro lenguaje convencional, Handel-C proporciona instrucciones para controlar el flujo de un programa. Por ejemplo, parte del código puede ser ejecutado de forma condicional en función del valor de alguna expresión, o un bloque de código se puede repetir un número de veces utilizando instrucciones de bucle.

Dado que el código generado por Handel-C interactúa con el hardware a bajo nivel, es posible conseguir grandes mejoras en el rendimiento mediante el uso de paralelismo. Aunque Handel-C es intrínsecamente secuencial, es posible (y de hecho esencial para la creación programas eficientes) indicar al compilador que construya el hardware necesario para ejecutar sentencias en paralelo. El paralelismo de Handel-C es paralelismo real, no es ese pseudo-paralelismo propio de los procesadores de propósito general en los que se asigna porciones de tiempo a los distintos procesos del sistema. En otras palabras, cuando se indica la ejecución de dos instrucciones en paralelo, dichas instrucciones se ejecutarán exactamente en el mismo instante en el tiempo por elementos separados de hardware.

Cuando se encuentra un bloque paralelo, el flujo de ejecución se divide al inicio de dicho bloque y cada rama del mismo se ejecuta de forma simultánea. Este flujo de

ejecución se unifica al final del bloque, cuando todas las ramas han completado sus instrucciones. Todas las líneas de ejecución deben esperar a la finalización de aquella más lenta para poder continuar. Este hecho se ilustra en el siguiente diagrama:

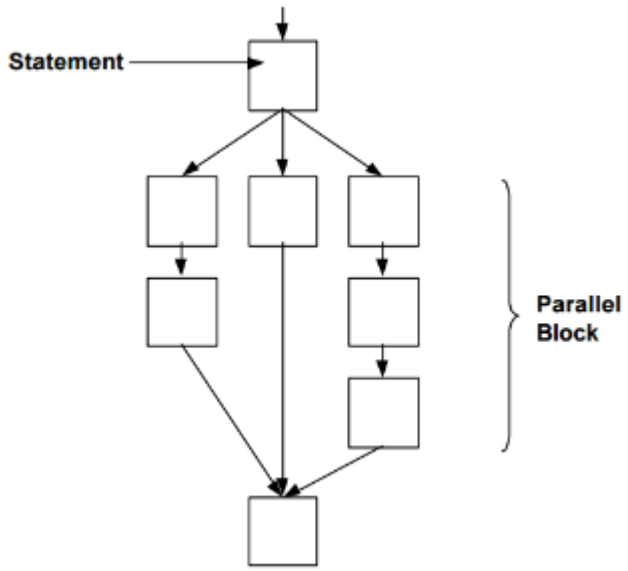

Figura 6. Ejemplo de estructura de un programa escrito en ***Handel-C***.

Este diagrama ilustra la ramificación y reunificación del flujo de ejecución. Las dos ramas más a la izquierda deben esperar para asegurarse de que todas las líneas de ejecución han finalizado antes de que la siguiente instrucción al bloque paralelo pueda ser ejecutada.

Canales de comunicación

Los canales de comunicación proporcionan un vínculo entre las distintas ramas paralelas. Una rama envía datos a uno de estos canales y el resto de las ramas los leen. Los canales pueden ser construidos con y sin capacidades FIFO:

- Canales construidos sin capacidad FIFO:

  Estos canales también proporcionan sincronización entre ramas paralelas, ya que la transferencia de datos sólo puede darse por finalizada cuando ambas partes están preparados para ello. Si el transmisor no está preparado para la comunicación, entonces el receptor debe esperar a que lo esté y viceversa. Este vínculo entre distintas ramas paralelas queda ilustrado en la Figura 7 de más abajo:

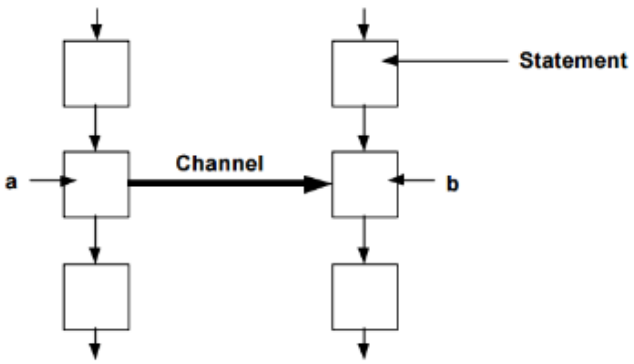

Figura 7. Ejemplo de comunicación entre dos ramas paralelas mediante el uso de un ***channel***.

Aquí, el canal se muestra enviando datos de la rama izquierda a la de la derecha. Si la primera alcanza el punto 'a' antes de que la segunda llegue al punto 'b', aquella debe esperar en dicho punto a que la rama de la derecha alcance el punto de comunicación para poder iniciar la transmisión.

- Canales construidos sin capacidad FIFO:

  Un canal puede ser construido como una cola FIFO. En este caso, los datos se escriben a la cabeza de la FIFO y se leen de la cola. Si la FIFO está llena, las escrituras se bloquean hasta que un elemento procede a leer de la FIFO. Si por el contrario, la cola FIFO está vacía, todas las lecturas quedan bloqueadas hasta que haya datos listos para ser leídos.

  En el ejemplo de más arriba, si en su lugar utilizamos un canal FIFO, la rama izquierda no tendrá que esperar en el punto 'a' si hay espacio en la FIFO. En su lugar, puede escribir en la cola una vez por ciclo de reloj hasta que la FIFO se llene. Sólo entonces tendrá que esperar. Por otro lado, cada vez que la rama derecha lea de la cola en el punto 'b', los datos situados a la cabeza de la misma son eliminados, y el siguiente fragmento de datos se convierte en el nuevo top. La rama derecha debe esperar si la FIFO está vacía.

Estructura de un programa

Al igual que cualquier programa C convencional, una aplicación Handel-C consiste en una serie de instrucciones que se ejecutan secuencialmente. Estas instrucciones están contenidas dentro de una función main () para informar al compilador de dónde comienza el programa. El cuerpo de la función principal se puede dividir en un número determinado de bloques usando las llaves {...}, las cuales permiten romper el programa en trozos legibles y restringen el ámbito de las variables y de los identificadores.

Handel-C también tiene variables y expresiones similares a las que podemos encontrar en C, aunque con una serie de restricciones, ya que ciertas operaciones no son adecuadas para su implementación hardware. De igual modo, también disponemos de ciertas funcionalidades adicionales debido al bajo nivel con el que trabajamos con el hardware.

Algo que diferencia Handel-C de C (hablando en términos de sintaxis, en funcionalidad son muy diferentes), es que dispone de declaraciones para ejecutar código en paralelo. Esta característica es fundamental cuando estamos diseñando el comportamiento hardware de una FPGA, ya que el paralelismo es el principal medio del que disponemos a la hora de incrementar su rendimiento.

La estructura general de un programa Handel-C es la siguiente:

```
Global Declarations
void main(void) {
    Local Declarations
    Body Code

}
```

Nótese que la función `main ()` no tiene argumentos de entrada y tampoco devuelve nada. Esto está en consonancia con la implementación hardware, donde no se reciben argumentos por línea de comandos, ni hay entorno al que devolver valores. Los parámetros `argc`, `argv` y `envp`, y el valor retorno típicos de C, pueden ser

reemplazados por comunicaciones explícitas con un sistema externo (por ejemplo, con un microprocesador host) dentro del cuerpo del programa.

Tipos de datos

Int

Handel-C únicamente dispone de un tipo fundamental para las variables: int. Además, éste tipo tipo puede ser calificado con la palabra clave unsigned para indicar que la variable sólo contiene números enteros positivos. Por ejemplo:

```
int 5 x;
unsigned int 13 y;

unsigned z;
```

Estas tres líneas declaran 3 variables: una de 5 bits con signo -x-, otra de 13 bits sin signo -y-, y otra sin signo -z- en la que es el compilador el que infiere el tamaño de la variable. Nótese que el rango de un entero con signo de 8 bits es -128 a 127 mientras que el rango de un entero sin signo de 8 bits es de 0 a 255 ambos inclusive. Esto se debe a que los enteros con signo utilizan la representación de complemento a 2.

Handel-C también proporciona soporte para portar aplicaciones de C al permitir los tipos `char`, `short` y `long`.

```
unsigned char w;
short y;

unsigned long z;
```

El tamaño inferido para cada uno de estos tipos es el siguiente:

Type	Width
char	8 bits
short	16 bits
long	32 bits

Tabla 1. Ancho en bits de los tipos básicos soportados por ***Handel-C***.

Arrays

Handel-C nos permite declarar arrays de la misma forma en que lo haríamos en C, es decir:

```
int 6 x[7];
```

Esto declara 7 registros, teniendo cada uno de los cuales 6 bits de ancho. El acceso a los distintos elementos del array es exactamente igual que en C, mediante el uso de un índice, el cual también comienza en 0. Por ejemplo, para acceder al quinto elemento del array, escribiríamos lo siguiente:

```
x[4] = 1;
```

También es posible crear arrays multidimensionales:

```
unsigned int 6 x[4][5][6];
```

Esto crea un array de 4x5x6=120 variables de 6 bits cada una.

Cuando se accede a un array, el índice debe ser conocido en tiempo de compilación, es decir, no puede ser aleatorio o generado durante la ejecución del programa.

Channels

Handel-C proporciona canales para la comunicación entre ramas paralelas. Una rama escribe en un canal y una segunda rama lee de él. La comunicación sólo se produce cuando ambas hilos están listos para la transferencia, momento en el que los datos son transferidos de una rama a otra.

La palabra clave utilizada para crear estos canales es chan.

```
chan int 7 link;
```

Como con las variables, el compilador de Handel-C puede inferir el tamaño de los canales si este ha sido especificado como undefined.

```
set intwidth = undefined;

chan int Link1;
chan unsigned undefined Link2;
chan Link3;
```

RAM y ROM

Los tipos ram y rom pueden ser construidos a partir de la lógica proporcionada en la FPGA utilizando las palabras clave ram y rom. Por ejemplo:

```
ram int 6 a[43];
rom int 16 b[4] = { 23, 46, 69, 92 };
```

En este ejemplo se construye una RAM que consta de 43 entradas cada una de las cuales de 6 bits de ancho y una ROM que consta de 4 entradas donde cada una de las cuales es de 16 bits de ancho. La ROM se inicializa con las constantes especificadas en la lista, más o menos de la misma manera en que se inicializa un array en C.

El compilador de Handel-C también puede inferir los anchos, tipos y el número de entradas de ram y rom de su uso. Por lo que también podemos declararlos de la siguiente forma:

```
ram int undefined a[123];
ram int 6 b[];
ram c[43];
ram d[];
```

Los tipos ram y rom son accedidos de igual forma que los arrays en C, es decir, mediante un índice que comienza en 0.

```
ram int 6 b[56];
b[7] = 4;
```

Nótese que las RAMs difieren de los arrays en que éstos equivalen a declarar un número de variables igual a su tamaño. Cada entrada en un array puede ser utilizada exactamente como una variable individual con tantas lecturas y escrituras por ciclo de reloj como se requiera. Las RAMs, por otro lado, son normalmente más eficientes de implementar en términos de recursos hardware y además también permiten el uso de índices no conocidos en tiempos de compilación, aunque en contraposición, no permiten el acceso a más de uno de sus elementos por ciclo de reloj.

Ejemplos

A continuación se muestran una serie de ejemplos que ilustran la sintaxis y estructura de un programa en Handel-C sencillo.

Ejemplo 1

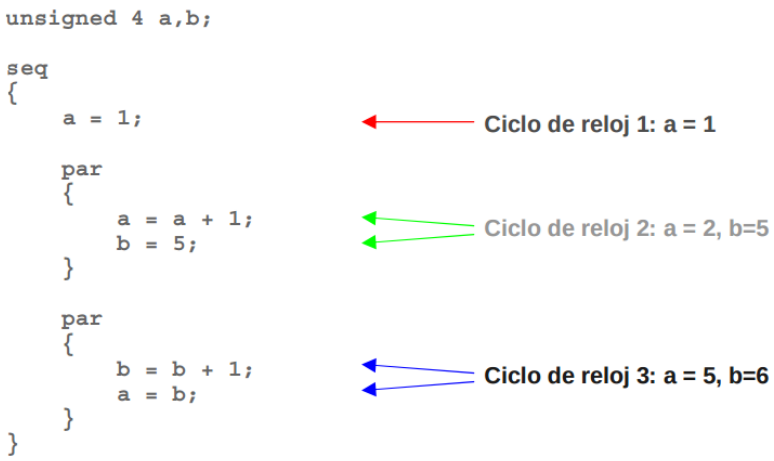

Figura 8. Ejemplo de programa con un bloque principal secuencial y dos secciones internas paralelas.

En este primer ejemplo, vemos cómo dentro de un bloque secuencial, indicado mediante la palabra clave `seq` (la cual es opcional), se encuentran una instrucción y dos bloques paralelos, marcados con la palabra clave `par`. En el primer ciclo de reloj se ejecuta la instrucción:

```
a = 1;
```

En el segundo ciclo de reloj se ejecutan todas las instrucciones del primer bloque paralelo:

```
a = a + 1;
b = 5;
```

Y en el tercer y último ciclo de reloj se ejecutan las instrucciones del segundo bloque paralelo:

```
b = b + 1;
a = b;
```

Ejemplo 2

Figura 9. Ejemplo de programa con un bloque principal paralelo y una sección interna secuencial.

En este segundo ejemplo, tenemos justamente lo contrario: un bloque principal paralelo en el que dentro se encuentra una instrucción básica y un bloque secuencial. De manera que en el primer ciclo de reloj se ejecuta la siguiente instrucción:

```
a--;
```

Por defecto y si no se especifica lo contrario, en Handel-C todas las instrucciones se ejecutan en secuencial.

En el segundo ciclo de reloj se ejecutan ta primera instrucción del bloque paralelo junto a la primera instrucción del bloque secuencial al que engloba:

```
b++;
a++;
```

En el tercer ciclo de reloj se ejecuta la segunda y última instrucción del bloque secuencial:

```
a = b;
```

Y en el cuarto y último ciclo de reloj se ejecuta la instrucción posterior al bloque paralelo:

```
b--;
```

3.2.3 Xilinx ISE 14.6.

El entorno de desarrollo que se ha utilizado para diseñar y simular el circuito a implementar en la FPGA para el cálculo del fitness en el algoritmo genético ha sido la aplicación Xilinx ISE, en su versión 14.6.

Xilinx ISE (Integrated Synthesis Environment) es una herramienta software desarrollada por Xilinx para la síntesis y análisis de los diseños HDL, la cual permite al desarrollador sintetizar ("compilar") sus diseños, realizar análisis de tiempo, examinar diagramas RTL, simular el comportamiento de su diseño ante diferentes estímulos y configurar el dispositivo de destino entre otras cosas.

Una de las limitaciones de esta herramienta es que está diseñada para trabajar con las FPGAs producidas por la propia Xilinx, lo que significa que está fuertemente acoplada a la arquitectura de dichos chips, por lo que no puede ser utilizada con FPGAs de otros proveedores. Xilinx ISE se utiliza principalmente para la síntesis y diseño de circuitos integrados, mientras que el simulador ModelSim se utiliza para las pruebas a nivel de sistema. Otros componentes suministrados junto a Xilinx ISE incluyen **EDK** (Embedded Development Kit), **SDK** (Software Development Kit) y **ChipScope Pro**.

Desde 2012, el software Xilinx ISE ha sido abandonado en favor de **Vivado Design Suite**, que sirve a los mismos propósitos, pero que además ofrece características adicionales para el diseño de chips. La última versión de esta herramienta, la 14.7, fue lanzada en Octubre de 2013, a la cual se sigue ofreciendo soporte pero sobre la que nos se realizarán desarrollos nuevos.

INTERFAZ DE USUARIO.

La interfaz de usuario principal de ISE es el navegador de proyectos, que incluye la jerarquía de diseño (Sources), un editor de código fuente (Workspace), una consola de salida (Transcript), y un árbol de procesos (Processes).

La jerarquía de diseño se compone de módulos (archivos de diseño, valga la redundancia), cuyas dependencias son interpretadas por Xilinx y que dan como resultado una estructura arbórea. El diseño de un chip puede estar compuesto por un módulo principal que a su vez incluye otros módulos secundarios; algo similar a lo que ocurre con la subrutina main() de C++ y sus llamadas a otras funciones previamente definidas. Las restricciones de diseño se especifican en los módulos e incluyen la configuración de los pines y el mapping.

En el árbol de procesos se describen las operaciones que Xilinx realizará en el módulo activo en ese momento. Esta jerarquía incluye las funciones de compilación, funciones de dependencias y otras herramientas. Esta pestaña también indica problemas o errores que surgen en cada una de las funciones.

La consola de salida proporciona el estado de las operaciones que se están ejecutando, e informa a los desarrolladores sobre problemas en el diseño. Estos problemas pueden ser filtrados para mostrar únicamente advertencias, errores, o ambos.

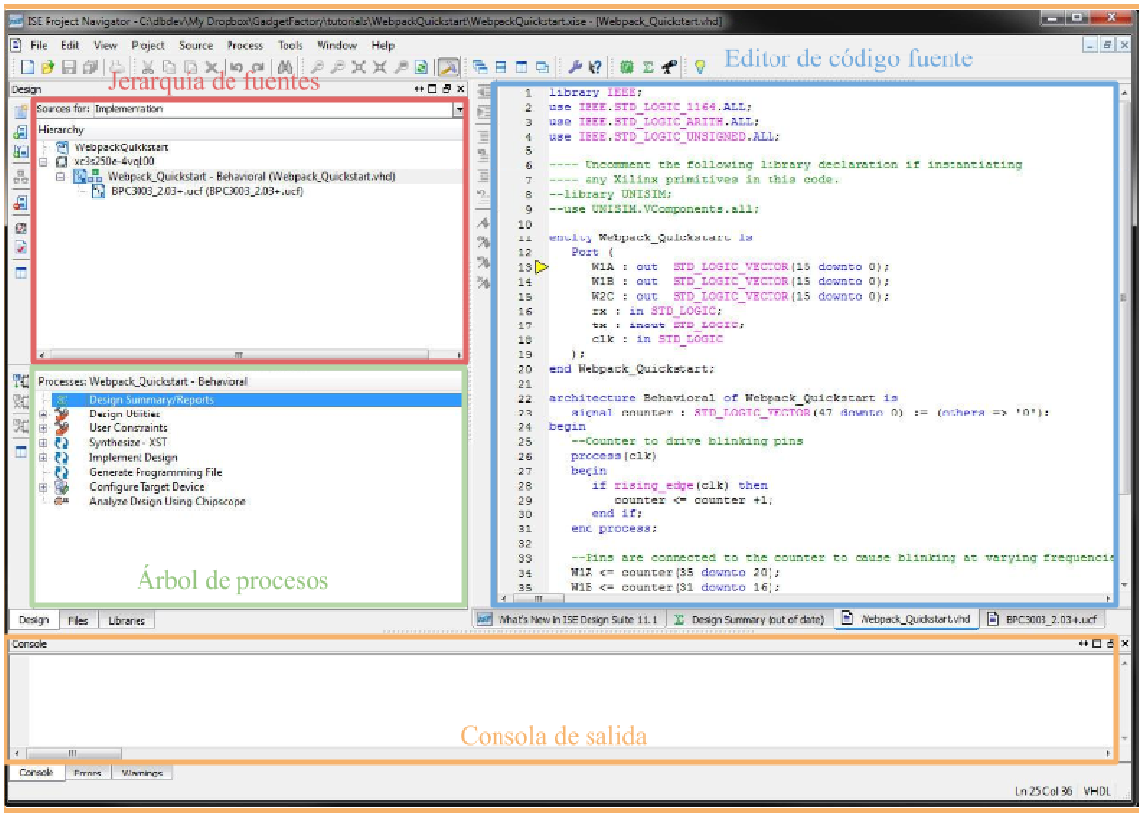

Figura 10. Interfaz principal con las distintas áreas de trabajo del software Xilinx ISE.

SIMULACIÓN.

Las pruebas a nivel de sistema pueden llevarse a cabo con el simulador lógico **ModelSim**. Este tipo de simulaciones también están escritas en lenguaje HDL. Los tests pueden incluir desde señales de entrada en forma de onda, hasta monitores que observan y verifican las salidas del dispositivo que se está probando.

ModelSim puede ser utilizado para realizar los siguientes tipos de simulación:

- **Verificación lógica**, para asegurar que el módulo produce los resultados esperados.
- **Verificación comportamental**, para verificar problemas lógicas y de temporización.
- **Simulación post-instalación** (Post-place simulation), para verificar el comportamiento del módulo después de haber sido colocado en la lógica reconfigurable de la FPGA.

SÍNTESIS.

Xilinx ha patentado algoritmos para la síntesis que permiten que sus diseños corran un 30% más rápido que aquellos generados por los programas de sus competidores, lo cual permite una mayor densidad de lógica y con ello una reducción del coste del proyecto.

Asimismo, debido a la creciente complejidad de la estructura de las FPGAs (incluyendo bloques de memoria y bloques de I/O), se han desarrollado algoritmos de

síntesis más complejos que separan módulos no relacionados en "compartimentos" separados, reduciendo con ello los errores post-instalación.

Xilinx, así como otros proveedores, ofrece **IP Cores** (Intellectual Property Core) para implementar funciones a nivel de sistema, tales como procesamiento de señales digitales (DSP), interfaces de bus, protocolos de red, procesamiento de imágenes, procesadores embebidos y periféricos. El papel de Xilinx ha sido fundamental para pasar de la implementación basada en ASIC a la implementación basada en FPGA.

DISPOSITIVOS SOPORTADOS.

Xilinx ofrece dos versiones de su herramienta software: una gratuita (**ISE Webpack**) y otra de pago por suscripción (**ISE Design Suite**). La principal diferencia entre ambas versiones es el número de dispositivos FPGA que soportan, estando los más potentes únicamente disponibles en la versión comercial.

La siguiente tabla lista las FPGAs soportadas por ambas versiones:

	ISE Webpack	ISE Design Suite
Virtex FPGA	Virtex-4 LX: XC4VLX15, XC4VLX25 SX: XC4VSX25 FX: XC4VFX12 Virtex-5 LX: XC5VLX30, XC5VLX50 LXT: XC5VLX20T - XC5VLX50T FXT: XC5VFX30T Virtex-6 XC6VLX75T	Virtex-4 LX: Todas SX: Todas FX: Todas Virtex-5 LX: Todas LXT: Todas SXT: Todas FXT: Todas Virtex-6 Todas
Spartan FPGA	Spartan-3 XC3S50 - XC3S1500 Spartan-3A Todas Spartan-3AN Todas Spartan-3A DSP XC3SD1800A Spartan-3E Todas Spartan-6 XC6SLX4 - XC6SLX75T XA (Xilinx Automotive) Spartan-6 Todas	Spartan-3 Todas Spartan-3A Todas Spartan-3AN Todas Spartan-3 DSP Todas Spartan-3E Todas Spartan-6 Todas XA (Xilinx Automotive)
Coolrunner PLA Coolrunner-II CPLD Coolrunner-IIA CPLD	Todas	
XC9500 Series CPLD	Todas excepto la familia 9500XV	

Tabla 2. Listado de FPGAs disponibles con cada versión del software.

3.3 Planteamiento.

3.3.1 Circuito para la implementación del fitness.

Planteamos el diseño de un circuito específico para la implementación de la función de fitness (en adelante, **F**), consistente en una serie de operaciones aritméticas que combinan pasos de ejecución secuenciales y paralelos.

Implementar F implica poner en juego un conjunto de operadores aritméticos (sumas, multiplicaciones, etc). La clave del diseño es implementar también estas operaciones de la forma lo más paralela posible, para obener un mayor rendimiento.

Por otro lado, es importante saber que algunos operandos pueden ser enteros, otros en punto flotante, y otras operaciones (como la división) pueden dar lugar a valores en punto flotante. Hay que tener esto en cuenta porque las operaciones con enteros son muchísimo más rápidas que las mismas en punto flotante (es más rápido 4x3 que 4.0x3.0). De esta forma, siempre que sea posible, utilizamos operaciones con enteros. En el caso, por ejemplo, de realizar 4/3, habrá que convertir 4 y 3 a punto flotante utilizando un core específico (generado a partir de *Xilinx Core Generator*) para la división en coma flotante. También hay que tener en cuenta que, si realizamos N operaciones paralelas con un determinado operador en coma flotante, necesitaremos N módulos del mismo operador (lo que tiene un impacto en el área ocupada de la FPGA).

Para la implementación de F utilizamos la fórmula dada en [Mohamad2008], que es utilizada en un AG para seleccionar genes en la clasificación del cáncer:

$$\mathbf{F(x) = w1 \cdot A(x) + (w2 \cdot (M - R(x)) / M)}$$

donde:

- $A(x) \in [0,1]$.
- $w1 \in [0.1,0.9]$.
- $w2 \in 1 - w1$.
- M = nº total de genes.
- R(x) = nº de genes seleccionados en x.

3.3.2 Circuito para la evaluación del rendimiento.

Una vez diseñado el circuito que implementa F, para medir su rendimiento (en tiempo de computación) diseñamos una arquitectura donde varias unidades F puedan trabajar en paralelo, mediante un circuito de prueba que controla la ejecución paralela de la evaluación del fitness de varios individuos.

El procesamiento de varias unidades F en paralelo permite acelerar el sistema, que tendrá más rendimiento cuantas más unidades paralelas F podamos integrar (lo que depende del área disponible de la FPGA). Por ejemplo, si tenemos 4 módulos F, los podemos ejecutar al mismo tiempo, lo cual equivaldría en una CPU de propósito general a procesar un bucle FOR de 4 iteraciones.

Finalmente, destacamos que, como el objetivo fundamental es implementar F y evaluar su rendimiento en un entorno paralelo, es indiferente el valor de los operandos de entrada, que pueden ser cualesquiera.

3.4 Implementación: Versión 1.

Para la evaluación del rendimiento del circuito acelerador del cálculo de la función de fitness, hemos creado un prototipo cuyo único propósito es medir el tiempo de computación del fitness. Este circuito ha sido modelado con lenguajes de descripción hardware VHDL y Handel-C e implementado sobre distintas FPGAs.

3.4.1 Módulo de alto nivel: top.

En la Figura 11 se observa la visión en alto nivel del circuito diseñado para medir el tiempo de computación. Este circuito tiene las siguientes señales:

- Entradas:
  - *clk*: señal de reloj.
  - *start*: señal que indica, mediante un pulso, la activación del circuito para realizar el cálculo.
- Salidas:
  - *rdy*: señal de 1 bit que indica (poniéndose a 1) si la operación se ha completado.
  - *y*: señal de 32 bit con el valor en coma flotante del mejor fitness evaluado.

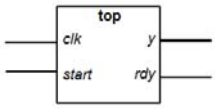

Figura 11. Visión en alto nivel del prototipo para evaluación del rendimiento del circuito acelerador del cálculo de la función de fitness.

Este circuito consta, como puede verse en la Figura 12 de cinco módulos:

- 1 registro con el valor del operando M en coma flotante (32 bit).
- 1 registro con el valor del operando R en coma flotante (32 bit).
- 1 registro con el valor del operando w1 en coma flotante (32 bit).
- 1 registro con el valor del operando w2 en coma flotante (32 bit).
- 1 unidad de control, llamada *parfcomp*, que controla el proceso de evaluar varias unidades paralelas de fitness.

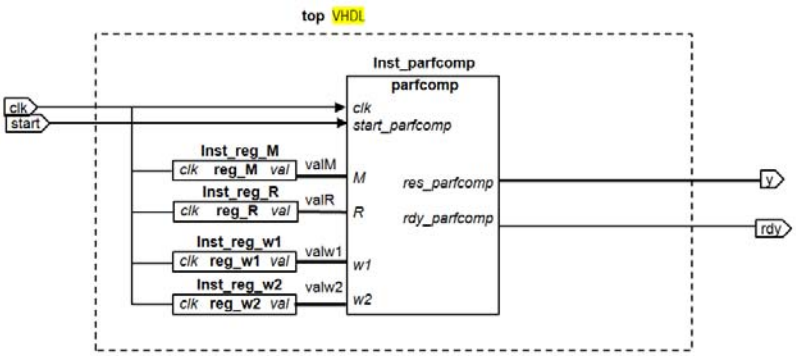

Figura 12. Componentes del módulo top: 4 registros de solo lectura y el módulo parfcomp para la computación paralela de las operaciones del fitness.

El código VHDL de este módulo se describe en la Figura 13. Es un código de tipo estructural, que tan solo define la estructura del módulo de alto nivel.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
------------------------------------------------
-- I/O section
--                  |----------|
--       clk ---1--> |   top    | --32--> y
--     start ---1--> |          | ---1--> rdy
--                  |----------|
-----------------------------
ENTITY top IS
  PORT
  (
    clk   : IN STD_LOGIC;
    start : IN STD_LOGIC;
            y     : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
    rdy   : OUT STD_LOGIC
  );
END top;

ARCHITECTURE estructural OF top IS
  --//-----------------------------------------//
  --//SIGNALS
  --//-----------------------------------------//
  SIGNAL valM   : STD_LOGIC_VECTOR (19 DOWNTO 0);
  SIGNAL valR   : STD_LOGIC_VECTOR (19 DOWNTO 0);
  SIGNAL valw1  : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL valw2  : STD_LOGIC_VECTOR (31 DOWNTO 0);

  --//-----------------------------------------//
  --//COMPONENTS
  --//-----------------------------------------//
  COMPONENT reg_M
    PORT
    (
    clk : in  STD_LOGIC;
    val : out STD_LOGIC_VECTOR (19 downto 0)
    );
  END COMPONENT;
  --//-----------------------------------------//
  COMPONENT reg_R
    PORT
    (
    clk : in  STD_LOGIC;
    val : out STD_LOGIC_VECTOR (19 downto 0)
    );
  END COMPONENT;
  --//-----------------------------------------//
  COMPONENT reg_w1
    PORT
    (
    clk : in  STD_LOGIC;
    val : out STD_LOGIC_VECTOR (31 downto 0)
    );
  END COMPONENT;
  --//-----------------------------------------//
  COMPONENT reg_w2
    PORT
    (
    clk : in  STD_LOGIC;
    val : out STD_LOGIC_VECTOR (31 downto 0)
    );
  END COMPONENT;
  --//-----------------------------------------//
  COMPONENT parfcomp
    PORT
    (
    clk            : IN  STD_LOGIC;
    start_parfcomp : IN  STD_LOGIC;
    M              : IN  STD_LOGIC_VECTOR (19 DOWNTO 0);
    R              : IN  STD_LOGIC_VECTOR (19 DOWNTO 0);
    w1             : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
    w2             : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
    res_parfcomp   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
    rdy_parfcomp   : OUT STD_LOGIC
    );
  END COMPONENT ;
  --//-----------------------------------------//
BEGIN
  --//-----------------------------------------//
  Inst_reg_M: reg_M
  PORT MAP
  (
    clk => clk,
    val => valM
  );
  --//-----------------------------------------//
  Inst_reg_R: reg_R
  PORT MAP
  (
    clk => clk,
    val => valR
  );
  --//-----------------------------------------//
  Inst_reg_w1: reg_w1
  PORT MAP
  (
    clk => clk,
    val => valw1
  );
  --//-----------------------------------------//
  Inst_reg_w2: reg_w2
  PORT MAP
  (
    clk => clk,
    val => valw2
  );
  --//-----------------------------------------//
  Inst_parfcomp: parfcomp
  PORT MAP
    (
    clk            => clk,
    start_parfcomp => start,
    M              => valM,
    R              => valR,
    w1             => valw1,
    w2             => valw2,
    res_parfcomp   => y,
    rdy_parfcomp   => rdy
    );
  --//-----------------------------------------//

END estructural;
```

Figura 13. Código VHDL de la unidad top.

3.4.2 Módulo de control paralelo.

El módulo *parfcomp* tiene como propósito:

- Ejecutar varias unidades de fitness F en paralelo.
- Suministrar los valores de entrada a las unidades F.
- Comparar los resultados de las unidades F paralelas, determinando el valor mínimo, que sería el valor óptimo de la población en una generación dada.

Este módulo se define mediante un código VHDL estructural: *parfcomp.vhd*. En la Figura 14 se observa la estructura de este módulo, que consta de estos elementos:

- Señales de entrada:
  - *clk* (1 bit): Señal de reloj.

  - *start_parfcomp* (1 bit): Inicia el proceso de cálculos.
  - *M* (20 bit): Valor entero que representa el número total de genes.
  - *R* (20 bit): valor entero que representa el número de genes seleccionados.
  - *w1* (32 bit): Valor en coma flotante del parámetro w1.
  - *w2* (32 bit): Valor en coma flotante del parámetro w2.
- Señales de salida:
  - *res_parfcomp*: (32 bit): Resultado del mejor fitness encontrado (valor en coma flotante).
  - *rdy_parfcomp* (1 bit): Avisa (al ponerse a 1) de que la obtención del mejor fitness ya se ha producido.
- *parfcomp_core_top*: Es un circuito que controla el proceso de la evaluación paralela del fitness de varios individuos de la población. Está diseñado en Handel-C (*parfcomp_core.hcc*), a partir del cual la herramienta DK4 genera el correspondiente código VHDL (*parfcomp_core_top.vhd*), que es el que se instancia en la estructura del módulo *parfcomp*.
- *fitness*: Es el circuito que implementa la función de fitness F. Existen varios de estos módulos paralelos en el circuito parfcomp. Está definido mediante un código VHDL estructural, *fitness.vhd*.
- *fp_lt*: Es un circuito que permite la comparación de dos valores de entrada (*a* y *b*) en coma flotante, conde el criterio de comparación es "menor que". Estos módulos son utilizados para comparar los resultados de las unidades F; como hay varias de estas unidades, necesitaremos un determinado número de unidades *fp_lt* que operen en paralelo y en secuencial para determinar el valor mínimo de todos los fitness. Este módulo se genera a partir de la herramienta Xilinx Core Generator.

Hemos desarrollado diversas versiones del módulo *parfcomp*, según el número de unidades paralelas de fitness (***NF***) y el correspondiente número necesario de unidades paralelas de comparadores (***NC***). Estas implementaciones tienen el nombre de archivo *parfcomp_NF-X_NC-Y.vhd*, donde X indica el número de unidades paralelas de fitness, e Y indica el número de unidades paralelas de comparación. Por ejemplo, *parfcomp_NF-8_NC-4.vhd* tiene 8 unidades paralelas de fitness y 4 unidades paralelas de comparación. En total, hemos probado las siguientes configuraciones:

- NF=8, NC=4.
- NF=16, NC=8.
- NF=32, NC=16.
- NF=64, NC=32.
- NF=100, NC=50.
- NF=128, NC=64.
- NF=256, NC=128.

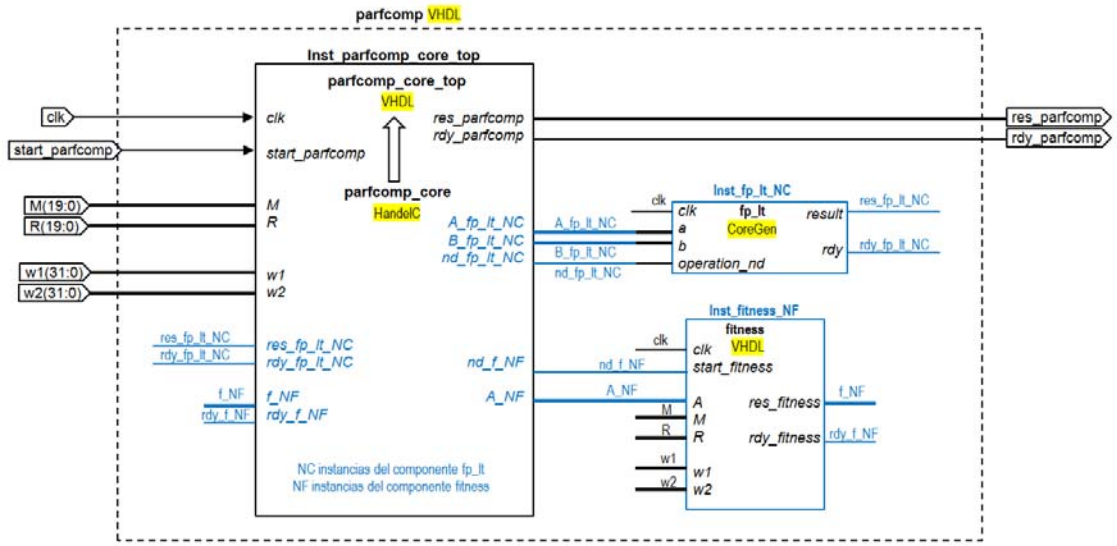

Figura 14. Componentes del módulo parfcomp: un controlador parfcomp_core_top (que controla el proceso de la evaluación paralela del fitness de varios individuos de la población) y varios componentes del tipo fitness (que implementa la función de fitness).

A su vez, esto implica tener también diversas configuraciones del controlador, para lo cual se han programado las mismas configuraciones anteriores según los códigos parfcomp_core_NF-X_NC-Y.hcc.

Para que no ocupe demasiado espacio, en la Figura 15 se observa el código estructural de la unidad *parfcomp* para la primera configuración, NF=8, NC=4, *parfcomp_NF-8_NC-4.vhd*; y en la Figura 16 se muestra el código Handel-C del controlador correspondiente a dicha configuración (*parfcomp_core_NF-8_NC-4.hcc*).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--------------------------------------------------
-- I/O section
--                      |----------|
--         clk ---1--> |          | -32--> res_parfcomp
-- start_parfcomp -1-> |          | --1--> rdy_parfcomp
--                      | parfcomp |
--             M -20--> |          |
--             R -20--> |          |
--                      |          |
--            w1 -32--> |          |
--            w2 -32--> |          |
--                      |----------|
-----------------------------
ENTITY parfcomp IS
PORT
  (
    clk            : IN  STD_LOGIC;
    start_parfcomp : IN  STD_LOGIC;
    M              : IN  STD_LOGIC_VECTOR (19 DOWNTO 0);
    R              : IN  STD_LOGIC_VECTOR (19 DOWNTO 0);
    w1             : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
    w2             : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
    res_parfcomp   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
    rdy_parfcomp   : OUT STD_LOGIC
  );
END parfcomp;

ARCHITECTURE estructural OF parfcomp IS

  --//-------------------------------------------//
  --//SIGNALS
  --//-------------------------------------------//
  --// float comparators
  SIGNAL A_fp_lt_0   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL B_fp_lt_0   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL nd_fp_lt_0  : STD_LOGIC;
  SIGNAL res_fp_lt_0 : STD_LOGIC_VECTOR(0 DOWNTO 0);
  SIGNAL rdy_fp_lt_0 : STD_LOGIC;
  SIGNAL A_fp_lt_1   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL B_fp_lt_1   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL nd_fp_lt_1  : STD_LOGIC;
  SIGNAL res_fp_lt_1 : STD_LOGIC_VECTOR(0 DOWNTO 0);
  SIGNAL rdy_fp_lt_1 : STD_LOGIC;
  SIGNAL A_fp_lt_2   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL B_fp_lt_2   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL nd_fp_lt_2  : STD_LOGIC;
  SIGNAL res_fp_lt_2 : STD_LOGIC_VECTOR(0 DOWNTO 0);
  SIGNAL rdy_fp_lt_2 : STD_LOGIC;
  SIGNAL A_fp_lt_3   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL B_fp_lt_3   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL nd_fp_lt_3  : STD_LOGIC;
  SIGNAL res_fp_lt_3 : STD_LOGIC_VECTOR(0 DOWNTO 0);
  SIGNAL rdy_fp_lt_3 : STD_LOGIC;
  --// fitness
  SIGNAL nd_f_0      : STD_LOGIC;
  SIGNAL A_0         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_0         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_0     : STD_LOGIC;
  SIGNAL nd_f_1      : STD_LOGIC;
  SIGNAL A_1         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_1         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_1     : STD_LOGIC;
  SIGNAL nd_f_2      : STD_LOGIC;
  SIGNAL A_2         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_2         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_2     : STD_LOGIC;
  SIGNAL nd_f_3      : STD_LOGIC;
  SIGNAL A_3         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_3         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_3     : STD_LOGIC;
  SIGNAL nd_f_4      : STD_LOGIC;
  SIGNAL A_4         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_4         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_4     : STD_LOGIC;
  SIGNAL nd_f_5      : STD_LOGIC;
  SIGNAL A_5         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_5         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_5     : STD_LOGIC;
  SIGNAL nd_f_6      : STD_LOGIC;
  SIGNAL A_6         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_6         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_6     : STD_LOGIC;
  SIGNAL nd_f_7      : STD_LOGIC;
  SIGNAL A_7         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL f_7         : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL rdy_f_7     : STD_LOGIC;
  --//-------------------------------------------//
  --//COMPONENTS
```

```
  --//-------------------------------------------//
  COMPONENT fitness
  PORT
  (
    clk           : IN STD_LOGIC;
    start_fitness : IN STD_LOGIC;
    A             : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    M             : IN STD_LOGIC_VECTOR (19 DOWNTO 0);
    R             : IN STD_LOGIC_VECTOR (19 DOWNTO 0);
    w1            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    w2            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    res_fitness   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
         rdy_fitness   : OUT STD_LOGIC
  );
  END COMPONENT;
  --//-------------------------------------------//
  COMPONENT parfcomp_core_top
  PORT
  (
    A_0 : OUT std_logic_vector(31 DOWNTO 0);
    A_1 : OUT std_logic_vector(31 DOWNTO 0);
    A_2 : OUT std_logic_vector(31 DOWNTO 0);
    A_3 : OUT std_logic_vector(31 DOWNTO 0);
    A_4 : OUT std_logic_vector(31 DOWNTO 0);
    A_5 : OUT std_logic_vector(31 DOWNTO 0);
    A_6 : OUT std_logic_vector(31 DOWNTO 0);
    A_7 : OUT std_logic_vector(31 DOWNTO 0);
    A_fp_lt_0 : OUT std_logic_vector(31 DOWNTO 0);
    A_fp_lt_1 : OUT std_logic_vector(31 DOWNTO 0);
    A_fp_lt_2 : OUT std_logic_vector(31 DOWNTO 0);
    A_fp_lt_3 : OUT std_logic_vector(31 DOWNTO 0);
    B_fp_lt_0 : OUT std_logic_vector(31 DOWNTO 0);
    B_fp_lt_1 : OUT std_logic_vector(31 DOWNTO 0);
    B_fp_lt_2 : OUT std_logic_vector(31 DOWNTO 0);
    B_fp_lt_3 : OUT std_logic_vector(31 DOWNTO 0);
    clk : IN std_logic;
    f_0 : IN std_logic_vector(31 DOWNTO 0);
    f_1 : IN std_logic_vector(31 DOWNTO 0);
    f_2 : IN std_logic_vector(31 DOWNTO 0);
    f_3 : IN std_logic_vector(31 DOWNTO 0);
    f_4 : IN std_logic_vector(31 DOWNTO 0);
    f_5 : IN std_logic_vector(31 DOWNTO 0);
    f_6 : IN std_logic_vector(31 DOWNTO 0);
    f_7 : IN std_logic_vector(31 DOWNTO 0);
    M : IN std_logic_vector(19 DOWNTO 0);
    nd_f_0 : OUT std_logic;
    nd_f_1 : OUT std_logic;
    nd_f_2 : OUT std_logic;
    nd_f_3 : OUT std_logic;
    nd_f_4 : OUT std_logic;
    nd_f_5 : OUT std_logic;
    nd_f_6 : OUT std_logic;
    nd_f_7 : OUT std_logic;
    nd_fp_lt_0 : OUT std_logic;
    nd_fp_lt_1 : OUT std_logic;
    nd_fp_lt_2 : OUT std_logic;
    nd_fp_lt_3 : OUT std_logic;
    R : IN std_logic_vector(19 DOWNTO 0);
    rdy_parfcomp : OUT std_logic;
    rdy_f_0 : IN std_logic;
    rdy_f_1 : IN std_logic;
    rdy_f_2 : IN std_logic;
    rdy_f_3 : IN std_logic;
    rdy_f_4 : IN std_logic;
    rdy_f_5 : IN std_logic;
    rdy_f_6 : IN std_logic;
    rdy_f_7 : IN std_logic;
    rdy_fp_lt_0 : IN std_logic;
    rdy_fp_lt_1 : IN std_logic;
    rdy_fp_lt_2 : IN std_logic;
    rdy_fp_lt_3 : IN std_logic;
    res_fp_lt_0 : IN std_logic_vector(0 DOWNTO 0);
    res_fp_lt_1 : IN std_logic_vector(0 DOWNTO 0);
    res_fp_lt_2 : IN std_logic_vector(0 DOWNTO 0);
    res_fp_lt_3 : IN std_logic_vector(0 DOWNTO 0);
    res_parfcomp : OUT std_logic_vector(31 DOWNTO 0);
    start_parfcomp : IN std_logic;
    w1 : IN std_logic_vector(31 DOWNTO 0);
    w2 : IN std_logic_vector(31 DOWNTO 0)
  );
  END COMPONENT;
  --//-------------------------------------------//
  COMPONENT fp_lt
  PORT
  (
    clk          : IN STD_LOGIC;
    a            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    b            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    operation_nd : IN STD_LOGIC;
    result       : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
    rdy          : OUT STD_LOGIC
  );
  END COMPONENT;
  --//-------------------------------------------//

BEGIN

  --//-------------------------------------------//
  Inst_fp_lt_0: fp_lt
  PORT MAP
  (
    clk          => clk,
    a            => A_fp_lt_0,
    b            => B_fp_lt_0,
    operation_nd => nd_fp_lt_0,
    result       => res_fp_lt_0,
    rdy          => rdy_fp_lt_0
  );
  --//-------------------------------------------//
  Inst_fp_lt_1: fp_lt
  PORT MAP
  (
    clk          => clk,
    a            => A_fp_lt_1,
    b            => B_fp_lt_1,
    operation_nd => nd_fp_lt_1,
    result       => res_fp_lt_1,
    rdy          => rdy_fp_lt_1
  );
  --//-------------------------------------------//
  Inst_fp_lt_2: fp_lt
  PORT MAP
  (
    clk          => clk,
    a            => A_fp_lt_2,
    b            => B_fp_lt_2,
    operation_nd => nd_fp_lt_2,
    result       => res_fp_lt_2,
    rdy          => rdy_fp_lt_2
  );
  --//-------------------------------------------//
  Inst_fp_lt_3: fp_lt
  PORT MAP
  (
    clk          => clk,
    a            => A_fp_lt_3,
    b            => B_fp_lt_3,
    operation_nd => nd_fp_lt_3,
    result       => res_fp_lt_3,
    rdy          => rdy_fp_lt_3
  );
  --//-------------------------------------------//
  Inst_fitness_0:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_0,
    A             => A_0,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
    res_fitness   => f_0,
    rdy_fitness   => rdy_f_0
  );
  --//-------------------------------------------//
  Inst_fitness_1:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_1,
    A             => A_1,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
    res_fitness   => f_1,
    rdy_fitness   => rdy_f_1
  );
  --//-------------------------------------------//
  Inst_fitness_2:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_2,
    A             => A_2,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
    res_fitness   => f_2,
    rdy_fitness   => rdy_f_2
  );
  --//-------------------------------------------//
  Inst_fitness_3:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_3,
    A             => A_3,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
    res_fitness   => f_3,
    rdy_fitness   => rdy_f_3
  );
  --//-------------------------------------------//
  Inst_fitness_4:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_4,
    A             => A_4,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
```

```
    res_fitness   => f_4,
    rdy_fitness   => rdy_f_4
  );
  --//-----------------------------------------//
  Inst_fitness_5:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_5,
    A             => A_5,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
    res_fitness   => f_5,
    rdy_fitness   => rdy_f_5
  );
  --//-----------------------------------------//
  Inst_fitness_6:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_6,
    A             => A_6,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
    res_fitness   => f_6,
    rdy_fitness   => rdy_f_6
  );
  --//-----------------------------------------//
  Inst_fitness_7:fitness
  PORT MAP
  (
    clk           => clk,
    start_fitness => nd_f_7,
    A             => A_7,
    M             => M,
    R             => R,
    w1            => w1,
    w2            => w2,
    res_fitness   => f_7,
    rdy_fitness   => rdy_f_7
  );
--//-----------------------------------------//
  Inst_parfcomp_core_top: parfcomp_core_top
  PORT MAP
  (
    A_0            => A_0,
    A_1            => A_1,
    A_2            => A_2,
    A_3            => A_3,
    A_4            => A_4,
    A_5            => A_5,
    A_6            => A_6,
    A_7            => A_7,
    A_fp_lt_0      => A_fp_lt_0,
    A_fp_lt_1      => A_fp_lt_1,
    A_fp_lt_2      => A_fp_lt_2,
    A_fp_lt_3      => A_fp_lt_3,
    B_fp_lt_0      => B_fp_lt_0,
    B_fp_lt_1      => B_fp_lt_1,
    B_fp_lt_2      => B_fp_lt_2,
    B_fp_lt_3      => B_fp_lt_3,
    clk            => clk,
    f_0            => f_0,
    f_1            => f_1,
    f_2            => f_2,
    f_3            => f_3,
    f_4            => f_4,
    f_5            => f_5,
    f_6            => f_6,
    f_7            => f_7,
    M              => M,
    nd_f_0         => nd_f_0,
    nd_f_1         => nd_f_1,
    nd_f_2         => nd_f_2,
    nd_f_3         => nd_f_3,
    nd_f_4         => nd_f_4,
    nd_f_5         => nd_f_5,
    nd_f_6         => nd_f_6,
    nd_f_7         => nd_f_7,
    nd_fp_lt_0     => nd_fp_lt_0,
    nd_fp_lt_1     => nd_fp_lt_1,
    nd_fp_lt_2     => nd_fp_lt_2,
    nd_fp_lt_3     => nd_fp_lt_3,
    R              => R,
    rdy_parfcomp   => rdy_parfcomp,
    rdy_f_0        => rdy_f_0,
    rdy_f_1        => rdy_f_1,
    rdy_f_2        => rdy_f_2,
    rdy_f_3        => rdy_f_3,
    rdy_f_4        => rdy_f_4,
    rdy_f_5        => rdy_f_5,
    rdy_f_6        => rdy_f_6,
    rdy_f_7        => rdy_f_7,
    rdy_fp_lt_0    => rdy_fp_lt_0,
    rdy_fp_lt_1    => rdy_fp_lt_1,
    rdy_fp_lt_2    => rdy_fp_lt_2,
    rdy_fp_lt_3    => rdy_fp_lt_3,
    res_fp_lt_0    => res_fp_lt_0,
    res_fp_lt_1    => res_fp_lt_1,
    res_fp_lt_2    => res_fp_lt_2,
    res_fp_lt_3    => res_fp_lt_3,
    start_parfcomp => start_parfcomp,
    w1             => w1,
    w2             => w2,
    res_parfcomp   => res_parfcomp
  );
  --//-----------------------------------------//,
END estructural;
```

Figura 15. Código estructural VHDL del módulo parfcomp, para la configuración NF=8, NC=4: parfcomp_NF-8_NC-4.vhd.

```
// parfcomp_core.hcc
/*=====================================================*/
/* FUNCTION DECLARATION section                        */
unsigned int 1 fp_lt_0(unsigned int 32 op1,unsigned int
32 op2);
unsigned int 1 fp_lt_1(unsigned int 32 op1,unsigned int
32 op2);
unsigned int 1 fp_lt_2(unsigned int 32 op1,unsigned int
32 op2);
unsigned int 1 fp_lt_3(unsigned int 32 op1,unsigned int
32 op2);
//----//
unsigned int 32 fitness_0(unsigned int 32 op);
unsigned int 32 fitness_1(unsigned int 32 op);
unsigned int 32 fitness_2(unsigned int 32 op);
unsigned int 32 fitness_3(unsigned int 32 op);
unsigned int 32 fitness_4(unsigned int 32 op);
unsigned int 32 fitness_5(unsigned int 32 op);
unsigned int 32 fitness_6(unsigned int 32 op);
unsigned int 32 fitness_7(unsigned int 32 op);

/*=====================================================*/
/* DEFINES section                                     */
#define NF 8
#define NC 4

/*=====================================================*/
/* GLOBAL VARIABLES section                            */
/*=====================================================*/
// I/O section
//                      |-----------------|
//             clk -1-> |              |-32->res_parfcomp
//start_parfcomp  --1-> |              |-1->rdy_parfcomp
//                      | parfcomp_core   |
//              M -20-> |                 |
//              R -20-> |                 |
//                      |                 |
//             w1 -32-> |                 |
//             w2 -32-> |                 |
//                      |                 |
//    res_fp_lt_x --1-> | x (NC=16) ports |-32->A_fp_lt_x
//    rdy_fp_lt_x --1-> | para el         |-32->B_fp_lt_x
//                      | componente      |-1->nd_fp_lt_x
//                      | fp_lt           |
//                      |                 |
//           f_y -32-> | y (NF=32) ports | --1-> nd_f_y
//       rdy_f_y --1-> | para el         | -32-> A_y
//                      | componente      |
//                      | fitness         |
//                      |-----------------|
//
/*-----------------------------*/
/* INPUTS */
interface port_in (unsigned int 1  clk)  ClockPort();
set clock = internal ClockPort.clk;
interface port_in (unsigned int 1 start_parfcomp)
Read_start_parfcomp();
interface port_in (unsigned int 20 M)
Read_M()           with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 20 R)
Read_R()           with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 w1)
Read_w1()          with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 w2)
Read_w2()          with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 res_fp_lt_0)
Read_res_fp_lt_0() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_0)
Read_rdy_fp_lt_0();
interface port_in (unsigned int 1 res_fp_lt_1)
Read_res_fp_lt_1() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_1)
Read_rdy_fp_lt_1();
interface port_in (unsigned int 1 res_fp_lt_2)
Read_res_fp_lt_2() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_2)
Read_rdy_fp_lt_2();
interface port_in (unsigned int 1 res_fp_lt_3)
Read_res_fp_lt_3() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_3)
```

```
Read_rdy_fp_lt_3();
//----//
interface port_in (unsigned int 32 f_0)
Read_f_0()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_0)
Read_rdy_f_0();
interface port_in (unsigned int 32 f_1)
Read_f_1()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_1)
Read_rdy_f_1();
interface port_in (unsigned int 32 f_2)
Read_f_2()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_2)
Read_rdy_f_2();
interface port_in (unsigned int 32 f_3)
Read_f_3()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_3)
Read_rdy_f_3();
interface port_in (unsigned int 32 f_4)
Read_f_4()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_4)
Read_rdy_f_4();
interface port_in (unsigned int 32 f_5)
Read_f_5()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_5)
Read_rdy_f_5();
interface port_in (unsigned int 32 f_6)
Read_f_6()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_6)
Read_rdy_f_6();
interface port_in (unsigned int 32 f_7)
Read_f_7()       with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_f_7)
Read_rdy_f_7();
//----//
//-----------------------------//
// OUTPUTS
//-----------------------------//
unsigned int 32 res_parfcomp;
interface port_out () out_res_parfcomp (unsigned int 32
res_parfcomp=res_parfcomp)
with {vhdl_type="std_logic_vector"};
//----//
unsigned int 1 rdy_parfcomp;
interface port_out () out_rdy (unsigned int 1
rdy_parfcomp=rdy_parfcomp);
//----//
unsigned int 32 A_fp_lt_0;
interface port_out () out_A_fp_lt_0 (unsigned int 32
A_fp_lt_0=A_fp_lt_0) with {vhdl_type="std_logic_vector"};
unsigned int 32 A_fp_lt_1;
interface port_out () out_A_fp_lt_1 (unsigned int 32
A_fp_lt_1=A_fp_lt_1) with {vhdl_type="std_logic_vector"};
unsigned int 32 A_fp_lt_2;
interface port_out () out_A_fp_lt_2 (unsigned int 32
A_fp_lt_2=A_fp_lt_2) with {vhdl_type="std_logic_vector"};
unsigned int 32 A_fp_lt_3;
interface port_out () out_A_fp_lt_3 (unsigned int 32
A_fp_lt_3=A_fp_lt_3) with {vhdl_type="std_logic_vector"};
//----//
unsigned int 32 B_fp_lt_0;
interface port_out () out_B_fp_lt_0 (unsigned int 32
B_fp_lt_0=B_fp_lt_0) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_lt_1;
interface port_out () out_B_fp_lt_1 (unsigned int 32
B_fp_lt_1=B_fp_lt_1) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_lt_2;
interface port_out () out_B_fp_lt_2 (unsigned int 32
B_fp_lt_2=B_fp_lt_2) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_lt_3;
interface port_out () out_B_fp_lt_3 (unsigned int 32
B_fp_lt_3=B_fp_lt_3) with {vhdl_type="std_logic_vector"};
//----//
unsigned int 1 nd_fp_lt_0;
interface port_out () out_nd_fp_lt_0
(unsigned int 1 nd_fp_lt_0=nd_fp_lt_0);
unsigned int 1 nd_fp_lt_1;
interface port_out () out_nd_fp_lt_1
(unsigned int 1 nd_fp_lt_1=nd_fp_lt_1);
unsigned int 1 nd_fp_lt_2;
interface port_out () out_nd_fp_lt_2
(unsigned int 1 nd_fp_lt_2=nd_fp_lt_2);
unsigned int 1 nd_fp_lt_3;
interface port_out () out_nd_fp_lt_3
(unsigned int 1 nd_fp_lt_3=nd_fp_lt_3);
//----//
unsigned int 1 nd_f_0;  interface port_out () out_nd_f_0
(unsigned int 1 nd_f_0=nd_f_0);
unsigned int 1 nd_f_1;  interface port_out () out_nd_f_1
(unsigned int 1 nd_f_1=nd_f_1);
unsigned int 1 nd_f_2;  interface port_out () out_nd_f_2
(unsigned int 1 nd_f_2=nd_f_2);
unsigned int 1 nd_f_3;  interface port_out () out_nd_f_3
(unsigned int 1 nd_f_3=nd_f_3);
unsigned int 1 nd_f_4;  interface port_out () out_nd_f_4
(unsigned int 1 nd_f_4=nd_f_4);
unsigned int 1 nd_f_5;  interface port_out () out_nd_f_5
(unsigned int 1 nd_f_5=nd_f_5);
unsigned int 1 nd_f_6;  interface port_out ()
out_nd_f_6 (unsigned int 1 nd_f_6=nd_f_6);
unsigned int 1 nd_f_7;  interface port_out ()
out_nd_f_7 (unsigned int 1 nd_f_7=nd_f_7);
//----//
unsigned int 32 A_0;  interface port_out ()
out_A_0 (unsigned int 32 A_0=A_0)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_1;  interface port_out ()
out_A_1 (unsigned int 32 A_1=A_1)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_2;  interface port_out ()
out_A_2 (unsigned int 32 A_2=A_2)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_3;  interface port_out ()
out_A_3 (unsigned int 32 A_3=A_3)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_4;  interface port_out ()
out_A_4 (unsigned int 32 A_4=A_4)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_5;  interface port_out ()
out_A_5 (unsigned int 32 A_5=A_5)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_6;  interface port_out ()
out_A_6 (unsigned int 32 A_6=A_6)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_7;  interface port_out ()
out_A_7 (unsigned int 32 A_7=A_7)
with {vhdl_type="std_logic_vector"};
//-----//

/*=================================================*/
unsigned int 1 fp_lt_0(unsigned int 32 op1,
unsigned int 32 op2)
    {
    unsigned int 1 result;
    par { A_fp_lt_0  = op1; B_fp_lt_0  = op2;
          nd_fp_lt_0 = 1; }
    nd_fp_lt_0 = 0;
    while(Read_rdy_fp_lt_0.rdy_fp_lt_0 == 0);
    result = Read_res_fp_lt_0.res_fp_lt_0;
    return(result);
    }
unsigned int 1 fp_lt_1(unsigned int 32 op1,
unsigned int 32 op2)
    {
    unsigned int 1 result;
    par { A_fp_lt_1  = op1; B_fp_lt_1  = op2;
          nd_fp_lt_1 = 1; }
    nd_fp_lt_1 = 0;
    while(Read_rdy_fp_lt_1.rdy_fp_lt_1 == 0);
    result = Read_res_fp_lt_1.res_fp_lt_1;
    return(result);
    }
unsigned int 1 fp_lt_2(unsigned int 32 op1,
unsigned int 32 op2)
    {
    unsigned int 1 result;
    par { A_fp_lt_2  = op1; B_fp_lt_2  = op2;
          nd_fp_lt_2 = 1; }
    nd_fp_lt_2 = 0;
    while(Read_rdy_fp_lt_2.rdy_fp_lt_2 == 0);
    result = Read_res_fp_lt_2.res_fp_lt_2;
    return(result);
    }
unsigned int 1 fp_lt_3(unsigned int 32 op1,
unsigned int 32 op2)
    {
    unsigned int 1 result;
    par { A_fp_lt_3  = op1; B_fp_lt_3  = op2;
          nd_fp_lt_3 = 1; }
    nd_fp_lt_3 = 0;
    while(Read_rdy_fp_lt_3.rdy_fp_lt_3 == 0);
    result = Read_res_fp_lt_3.res_fp_lt_3;
    return(result);
    }
/*=================================================*/
unsigned int 32 fitness_0(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_0 = op; nd_f_0 = 1; }
    nd_f_0 = 0;
    while(Read_rdy_f_0.rdy_f_0 == 0);
    result = Read_f_0.f_0;
    return(result);
    }
unsigned int 32 fitness_1(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_1 = op; nd_f_1 = 1; }
    nd_f_1 = 0;
    while(Read_rdy_f_1.rdy_f_1 == 0);
    result = Read_f_1.f_1;
    return(result);
    }
unsigned int 32 fitness_2(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_2 = op; nd_f_2 = 1; }
    nd_f_2 = 0;
    while(Read_rdy_f_2.rdy_f_2 == 0);
    result = Read_f_2.f_2;
    return(result);
    }
unsigned int 32 fitness_3(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_3 = op; nd_f_3 = 1; }
    nd_f_3 = 0;
    while(Read_rdy_f_3.rdy_f_3 == 0);
```

```
    result = Read_f_3.f_3;
    return(result);
    }
unsigned int 32 fitness_4(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_4 = op; nd_f_4 = 1; }
    nd_f_4 = 0;
    while(Read_rdy_f_4.rdy_f_4 == 0);
    result = Read_f_4.f_4;
    return(result);
    }
unsigned int 32 fitness_5(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_5 = op; nd_f_5 = 1; }
    nd_f_5 = 0;
    while(Read_rdy_f_5.rdy_f_5 == 0);
    result = Read_f_5.f_5;
    return(result);
    }
unsigned int 32 fitness_6(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_6 = op; nd_f_6 = 1; }
    nd_f_6 = 0;
    while(Read_rdy_f_6.rdy_f_6 == 0);
    result = Read_f_6.f_6;
    return(result);
    }
unsigned int 32 fitness_7(unsigned int 32 op)
    {
    unsigned int 32 result;
    par { A_7 = op; nd_f_7 = 1; }
    nd_f_7 = 0;
    while(Read_rdy_f_7.rdy_f_7 == 0);
    result = Read_f_7.f_7;
    return(result);
    }
/*=====================================================*/
void main(void)
{
unsigned int 20 lM,lR;
unsigned int 32 lw1,lw2;
unsigned int 32 f[NF], A[NF];
unsigned int  1 lt[NC];
unsigned int 32 Abest[NC],fmin[NC];
//----------------------------- //
while(1)
  {
  //-------------------- //Espera mientras start OFF
  while(Read_start_parfcomp.start_parfcomp == 0);
  //                     //Resetea salidas y las entradas
  par {
      rdy_parfcomp = 0;
      res_parfcomp = 0b00000000000000000000000000000000;
      lM  = Read_M.M;
      lR  = Read_R.R;
      lw1 = Read_w1.w1;
      lw2 = Read_w2.w2;
      }
  //-----------------------------
  // Mismo valor A (solo para pruebas de rendimiento)
  par {
      A[0]  = 0b01000000001110011001100110011010;
      A[1]  = 0b01000000001110011001100110011010;
      A[2]  = 0b01000000001110011001100110011010;
      A[3]  = 0b01000000001110011001100110011010;
      A[4]  = 0b01000000001110011001100110011010;
      A[5]  = 0b01000000001110011001100110011010;
      A[6]  = 0b01000000001110011001100110011010;
      A[7]  = 0b01000000001110011001100110011010;
      }
  //----------------------------- //
  par {
      f[0]  = fitness_0(A[0]);
      f[1]  = fitness_1(A[1]);
      f[2]  = fitness_2(A[2]);
      f[3]  = fitness_3(A[3]);
      f[4]  = fitness_4(A[4]);
      f[5]  = fitness_5(A[5]);
      f[6]  = fitness_6(A[6]);
      f[7]  = fitness_7(A[7]);
      }
  //                          -- // Fase 1 comparac.
  par {
      lt[0] =fp_lt_0(f[0],f[1]);
      lt[1] =fp_lt_0(f[1],f[2]);
      lt[2] =fp_lt_0(f[2],f[3]);
      lt[3] =fp_lt_0(f[3],f[4]);
      }
  par {
      { if(lt[0]==1)
            { fmin[0]=f[0];  Abest[0]=A[0]; }
        else
            { fmin[0]=f[1];  Abest[0]=A[1]; } }
      { if(lt[1]==1)
            { fmin[1]=f[2];  Abest[1]=A[2]; }
        else
            { fmin[1]=f[3];  Abest[1]=A[3]; } }
      { if(lt[2]==1)
            { fmin[2]=f[4];  Abest[2]=A[4]; }
        else
            { fmin[2]=f[5];  Abest[2]=A[5]; } }
      { if(lt[3]==1)
            { fmin[3]=f[6];  Abest[3]=A[6]; }
        else
            { fmin[3]=f[7];  Abest[3]=A[7]; } }
      }
  //----------------------------- //
  par {
      lt[0]=fp_lt_0(fmin[0],fmin[1]);
      lt[1]=fp_lt_1(fmin[2],fmin[3]);
      }
  par {
      { if(lt[0]==1)
            { fmin[0]=fmin[0];  Abest[0]=Abest[0]; }
        else
            { fmin[0]=fmin[1];  Abest[0]=Abest[1]; } }
      { if(lt[1]==1)
            { fmin[1]=fmin[2];  Abest[1]=Abest[2]; }
        else
            { fmin[1]=fmin[3];  Abest[1]=Abest[3]; } }
      }
  //----------------------------- //
  par {
      lt[0]=fp_lt_0(fmin[0],fmin[1]);
      }
  par {
      { if(lt[0]==1)
            { fmin[0]=fmin[0];  Abest[0]=Abest[0]; }
        else
            { fmin[0]=fmin[1];  Abest[0]=Abest[1]; } }
      }
  //----------------------------- //
  res_parfcomp = fmin[0];
  res_parfcomp = Abest[0];
  rdy_parfcomp = 1;   //Indica el fin del calculo
  }
}
```

Figura 16. Código Handel-C del módulo controlador parfcomp_core, para la configuración más sencilla, NF=8, NC=4: parfcomp_core_NF-8_NC-4.hcc. Se resaltan los bloques de ejecución paralela.

3.4.3 Módulo fitness.

El módulo *fitness* tiene como propósito implementar las operaciones aritméticas de la función F. En la Figura 17 puede observarse la arquitectura de este módulo, que está definido mediante un código VHDL estructural, *fitness.vhd* (Figura 18). Este módulo se compone de los siguientes elementos:

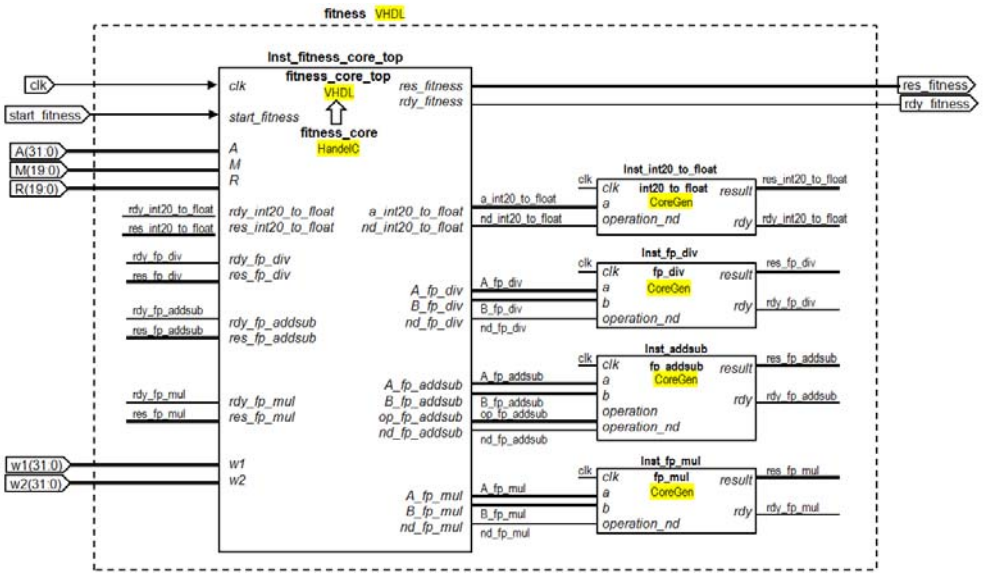

Figura 17. Componentes del módulo fitness: un controlador fitness_core_top (que controla los pasos del cálculo paralelo de la función de fitness) y varios componentes de operadores aritméticos en coma flotante necesarios para dicho cálculo: multiplicación.

- Entradas:
  - *clk* (1 bit): Señal de reloj.
  - *start_fitness* (1 bit): Inicio del cálculo de F.
  - *A* (32 bit): Valor en coma flotante del operando A.
  - *M* (20 bit): Valor entero del operando M.
  - *R* (20 bit): Valor entero del operando R.
- Salidas:
  - *res_fitness* (32 bit): Resultado del cálculo de la función F.
  - *rdy_fitness* (1 bit): Avisa de que el resultado está disponible (al ponerse a 1).
- fitness_core: Controlador del proceso de cálculo de la función F. Incluye un conjunto de pasos secuenciales y paralelos. Se comunica con los módulos de los operadores aritméticos en coma flotante (que se describen a continuación), necesarios para realizar las operaciones aritméticas. Este módulo está escrito en lenguaje Handel-C, en el archivo *fitness_core.hcc* (Figura 19), el cual es compilado a VHDL en el archivo *fitness_core_top.vhd*.
- *int20_to_float:* Módulo encargado de convertir valores enteros de 20 bit de resolución, a coma flotante (necesario en algunos pasos de los cálculos). Este módulo ha sido generado a partir de la herramienta *Xilinx Core Generator*.

- *fp_div*: Módulo que implementa el operador de división en coma flotante. Este módulo ha sido generado a partir de la herramienta *Xilinx Core Generator*.
- *fp_addsub*: Módulo que implementa el operador de suma/resta en coma flotante. Este módulo ha sido generado a partir de la herramienta *Xilinx Core Generator*.
- *fp_mul*: Módulo que implementa el operador de multiplicación en coma flotante. Este módulo ha sido generado a partir de la herramienta *Xilinx Core Generator*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
----------------------------------------------------
-- I/O section
--                          |----------|
--           clk --1--> |          | --32-> res_fitness
-- start_fitness --1--> |          | ---1-> rdy_fitness
--                          | fitness  |
--              A -32--> |          |
--              M -20--> |          |
--              R -20--> |          |
--                          |          |
--             w1 -32--> |          |
--             w2 -32--> |          |
--                          |----------|
------------------------------
ENTITY fitness IS
  PORT
  (
    clk           : IN STD_LOGIC;
    start_fitness : IN STD_LOGIC;
    A             : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    M             : IN STD_LOGIC_VECTOR (19 DOWNTO 0);
    R             : IN STD_LOGIC_VECTOR (19 DOWNTO 0);
    w1            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    w2            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    rdy_fitness   : OUT STD_LOGIC;
    res_fitness   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
  );
END fitness;

ARCHITECTURE estructural OF fitness IS

  --//-----------------------------------------------//
  --//SIGNALS
  --//-----------------------------------------------//
  --// integers to float
  SIGNAL a_int20_to_float:STD_LOGIC_VECTOR (19 DOWNTO 0);
  SIGNAL nd_int20_to_float  : STD_LOGIC;
  SIGNAL rdy_int20_to_float : STD_LOGIC;
SIGNAL res_int20_to_float:STD_LOGIC_VECTOR (31 DOWNTO 0);
  --// adder-subtractor floating point
  SIGNAL A_fp_addsub   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL B_fp_addsub   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL nd_fp_addsub  : STD_LOGIC;
  SIGNAL op_fp_addsub  : STD_LOGIC_VECTOR (5 DOWNTO 0);
  SIGNAL rdy_fp_addsub : STD_LOGIC;
  SIGNAL res_fp_addsub : STD_LOGIC_VECTOR (31 DOWNTO 0);
  --// multiplier
  SIGNAL A_fp_mul   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL B_fp_mul   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL nd_fp_mul  : STD_LOGIC;
  SIGNAL rdy_fp_mul : STD_LOGIC;
  SIGNAL res_fp_mul : STD_LOGIC_VECTOR (31 DOWNTO 0);
  --// divider
  SIGNAL A_fp_div   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL B_fp_div   : STD_LOGIC_VECTOR (31 DOWNTO 0);
  SIGNAL nd_fp_div  : STD_LOGIC;
  SIGNAL rdy_fp_div : STD_LOGIC;
  SIGNAL res_fp_div : STD_LOGIC_VECTOR (31 DOWNTO 0);

  --//-----------------------------------------------//
  --//COMPONENTS
  --//-----------------------------------------------//
  COMPONENT int20_to_float
    PORT
    (
      clk          : IN STD_LOGIC;
      a            : IN STD_LOGIC_VECTOR (19 DOWNTO 0);
      operation_nd : IN STD_LOGIC;
      result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      rdy          : OUT STD_LOGIC
    );
  END COMPONENT;
  --//-----------------------------------------------//
  COMPONENT fp_addsub
    PORT
    (
      clk          : IN STD_LOGIC;
      a            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      b            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      operation    : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
      operation_nd : IN STD_LOGIC;
      result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      rdy          : OUT STD_LOGIC
    );
  END COMPONENT;
  --//-----------------------------------------------//
  COMPONENT fp_mul
    PORT
    (
      clk          : IN STD_LOGIC;
      a            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      b            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      operation_nd : IN STD_LOGIC;
      result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      rdy          : OUT STD_LOGIC
    );
  END COMPONENT;
  --//-----------------------------------------------//
  COMPONENT fp_div
    PORT
    (
      clk          : IN STD_LOGIC;
      a            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      b            : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      operation_nd : IN STD_LOGIC;
      result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      rdy          : OUT STD_LOGIC
    );
  END COMPONENT;
  --//-----------------------------------------------//
  COMPONENT fitness_core_top
    PORT
 (
 clk               : IN  STD_LOGIC;
 start_fitness     : IN  STD_LOGIC;
 A                 : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
 M                 : IN  STD_LOGIC_VECTOR (19 DOWNTO 0);
 R                 : IN  STD_LOGIC_VECTOR (19 DOWNTO 0);
 rdy_int20_to_float: IN  STD_LOGIC;
 res_int20_to_float: IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
 rdy_fp_addsub     : IN  STD_LOGIC;
 res_fp_addsub     : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
 rdy_fp_mul        : IN  STD_LOGIC;
 res_fp_mul        : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
 rdy_fp_div        : IN  STD_LOGIC;
 res_fp_div        : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
 w1                : IN  std_logic_vector (31 DOWNTO 0);
 w2                : IN  std_logic_vector (31 DOWNTO 0);
 res_fitness       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
 rdy_fitness       : OUT STD_LOGIC;
 a_int20_to_float  : OUT STD_LOGIC_VECTOR (19 DOWNTO 0);
 nd_int20_to_float : OUT STD_LOGIC;
 A_fp_addsub       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
 B_fp_addsub       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
 nd_fp_addsub      : OUT STD_LOGIC;
 op_fp_addsub      : OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
 A_fp_mul          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
 B_fp_mul          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
 nd_fp_mul         : OUT STD_LOGIC;
 A_fp_div          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
 B_fp_div          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
 nd_fp_div         : OUT STD_LOGIC
 );
  END COMPONENT;
  --//-----------------------------------------------//

BEGIN

  --//-----------------------------------------------//
  Inst_int20_to_float: int20_to_float
  PORT MAP
  (
    clk          => clk,
    a            => a_int20_to_float,
    operation_nd => nd_int20_to_float,
    result       => res_int20_to_float,
    rdy          => rdy_int20_to_float
  );
  --//-----------------------------------------------//
  Inst_fp_addsub:fp_addsub
  PORT MAP
  (
    clk          => clk,
    a            => A_fp_addsub,
    b            => B_fp_addsub,
    operation    => op_fp_addsub,
    operation_nd => nd_fp_addsub,
    result       => res_fp_addsub,
```

```
    rdy          => rdy_fp_addsub
  );
  --//-------------------------------------------//
  Inst_fp_mul: fp_mul
  PORT MAP
  (
    clk          => clk,
    a            => A_fp_mul,
    b            => B_fp_mul,
    operation_nd => nd_fp_mul,
    result       => res_fp_mul,
    rdy          => rdy_fp_mul
  );
  --//-------------------------------------------//
  Inst_fp_div: fp_div
  PORT MAP
  (
    clk          => clk,
    a            => A_fp_div,
    b            => B_fp_div,
    operation_nd => nd_fp_div,
    result       => res_fp_div,
    rdy          => rdy_fp_div
  );
  --//-------------------------------------------//
  Inst_fitness_core_top: fitness_core_top
  PORT MAP
  (
    clk               => clk,
    start_fitness     => start_fitness,
    A                 => A,
    M                 => M,
    R                 => R,
    rdy_int20_to_float => rdy_int20_to_float,
    res_int20_to_float => res_int20_to_float,
    rdy_fp_addsub     => rdy_fp_addsub,
    res_fp_addsub     => res_fp_addsub,
    rdy_fp_mul        => rdy_fp_mul,
    res_fp_mul        => res_fp_mul,
    rdy_fp_div        => rdy_fp_div,
    res_fp_div        => res_fp_div,
    w1                => w1,
    w2                => w2,
    res_fitness       => res_fitness,
    rdy_fitness       => rdy_fitness,
    a_int20_to_float  => a_int20_to_float,
    nd_int20_to_float => nd_int20_to_float,
    A_fp_addsub       => A_fp_addsub,
    B_fp_addsub       => B_fp_addsub,
    nd_fp_addsub      => nd_fp_addsub,
    op_fp_addsub      => op_fp_addsub,
    A_fp_mul          => A_fp_mul,
    B_fp_mul          => B_fp_mul,
    nd_fp_mul         => nd_fp_mul,
    A_fp_div          => A_fp_div,
    B_fp_div          => B_fp_div,
    nd_fp_div         => nd_fp_div
  );
  --//-------------------------------------------//
END estructural;
```

Figura 18. Código VHDL estructural del módulo fitness (archivo fitness.vhd).

```
// fitness_core.hcc
/*=========================================================*/

/*=========================================================*/
/* FUNCTION DECLARATION section                            */
unsigned int 32 fp_mul(unsigned int 32 op1,unsigned int 32 op2);
unsigned int 32 fp_div(unsigned int 32 op1,unsigned int 32 op2);
unsigned int 32 fp_addsub(unsigned int 32 op1,unsigned int 32 op2,unsigned int 6 operation);
unsigned int 32 int20_to_float(unsigned int 20 op);

/*=========================================================*/
/* DEFINES section                                         */
/*=========================================================*/
/* GLOBAL VARIABLES section                                */
/*=========================================================*/
// I/O section
//                              |--------------|
//                clk ---1--> |              | --32--> res_fitness
//       start_fitness ---1--> |              | ---1--> rdy_fitness
//                              | fitness_core |
//                   A -32--> |              |
//                   M -20--> |              |
//                   R -20--> |              |
//                              |              |
//   rdy_int20_to_float --1--> |              | --20--> a_int20_to_float
//   res_int20_to_float -32--> |              | ---1--> nd_int20_to_float
//                              |              |
//                              |              |
//        rdy_fp_addsub --1--> |              | --32--> A_fp_addsub
//        res_fp_addsub -32--> |              | --32--> B_fp_addsub
//                              |              | ---1--> nd_fp_addsub
//                              |              | ---6--> op_fp_addsub
//                              |              |
//           rdy_fp_mul --1--> |              | --32--> A_fp_mul
//           res_fp_mul -32--> |              | --32--> B_fp_mul
//                              |              | ---1--> nd_fp_mul
//                              |              |
//           rdy_fp_div --1--> |              | --32--> A_fp_div
//           res_fp_div -32--> |              | --32--> B_fp_div
//                              |              | ---1--> nd_fp_div
//                   w1 -32--> |              |
//                   w2 -32--> |              |
//                              |--------------|
//
/*------------------------------*/
/* INPUTS */
interface port_in (unsigned int 1  clk)  ClockPort();
set clock = internal ClockPort.clk;
interface port_in (unsigned int 1 start_fitness) Read_start_fitness();
interface port_in (unsigned int 32 A) Read_A() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 20 M) Read_M() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 20 R) Read_R() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_int20_to_float) Read_rdy_int20_to_float();
interface port_in (unsigned int 32 res_int20_to_float) Read_res_int20_to_float() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_fp_addsub) Read_rdy_fp_addsub();
interface port_in (unsigned int 32 res_fp_addsub) Read_res_fp_addsub() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_fp_mul) Read_rdy_fp_mul();
interface port_in (unsigned int 32 res_fp_mul) Read_res_fp_mul() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_fp_div) Read_rdy_fp_div();
interface port_in (unsigned int 32 res_fp_div) Read_res_fp_div() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 w1) Read_w1() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 w2) Read_w2() with {vhdl_type="std_logic_vector"};
//----//
//-----------------------------//
// OUTPUTS
//-----------------------------//
unsigned int 32 res_fitness;
interface port_out () out_res_fitness (unsigned int 32 res_fitness=res_fitness) with {vhdl_type="std_logic_vector"};
```

```
//----//
unsigned int 1 rdy_fitness;
interface port_out () out_rdy_fitness (unsigned int 1 rdy_fitness=rdy_fitness);
//----//
unsigned int 20 a_int20_to_float;
interface   port_out   ()   out_a_int20_to_float   (unsigned   int   20   a_int20_to_float=a_int20_to_float)   with
{vhdl_type="std_logic_vector"};
unsigned int 1 nd_int20_to_float;
interface port_out () out_nd_int20_to_float (unsigned int 1 nd_int20_to_float=nd_int20_to_float);
//----//
unsigned int 32 A_fp_addsub;
interface port_out () out_A_fp_addsub (unsigned int 32 A_fp_addsub=A_fp_addsub) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_addsub;
interface port_out () out_B_fp_addsub (unsigned int 32 B_fp_addsub=B_fp_addsub) with {vhdl_type="std_logic_vector"};
unsigned int 1 nd_fp_addsub;
interface port_out () out_nd_fp_addsub (unsigned int 1 nd_fp_addsub=nd_fp_addsub);
unsigned int 6 op_fp_addsub;
interface port_out () out_op_fp_addsub (unsigned int 6 op_fp_addsub=op_fp_addsub) with {vhdl_type="std_logic_vector"};
//----//
unsigned int 32 A_fp_mul;
interface port_out () out_A_fp_mul (unsigned int 32 A_fp_mul=A_fp_mul) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_mul;
interface port_out () out_B_fp_mul (unsigned int 32 B_fp_mul=B_fp_mul) with {vhdl_type="std_logic_vector"};
unsigned int 1 nd_fp_mul;
interface port_out () out_nd_fp_mul (unsigned int 1 nd_fp_mul=nd_fp_mul);
//----//
unsigned int 32 A_fp_div;
interface port_out () out_A_fp_div (unsigned int 32 A_fp_div=A_fp_div) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_div;
interface port_out () out_B_fp_div (unsigned int 32 B_fp_div=B_fp_div) with {vhdl_type="std_logic_vector"};
unsigned int 1 nd_fp_div;
interface port_out () out_nd_fp_div (unsigned int 1 nd_fp_div=nd_fp_div);
//-----//
/*=========================================================*/
unsigned int 32 int20_to_float(unsigned int 20 op)
{
    unsigned int 32 result;
    par
    {
        a_int20_to_float  = op;
        nd_int20_to_float = 1;
    }
    nd_int20_to_float = 0;
    while(Read_rdy_int20_to_float.rdy_int20_to_float == 0);
    result = Read_res_int20_to_float.res_int20_to_float;
    return result;
}
/*=========================================================*/ // (op1 +/- op2)
unsigned int 32 fp_addsub(unsigned int 32 op1,unsigned int 32 op2,unsigned int 6 operation)
{
    unsigned int 32 result;
    par
    {
        A_fp_addsub  = op1;
        B_fp_addsub  = op2;
        op_fp_addsub = operation;
        nd_fp_addsub = 1;
    }
    nd_fp_addsub = 0;
    while(Read_rdy_fp_addsub.rdy_fp_addsub == 0);
    result = Read_res_fp_addsub.res_fp_addsub;
    return result;
}
/*=========================================================*/ // (op1 * op2)
unsigned int 32 fp_mul(unsigned int 32 op1,unsigned int 32 op2)
{
    unsigned int 32 result;
    par
    {
        A_fp_mul  = op1;
        B_fp_mul  = op2;
        nd_fp_mul = 1;
    }
    nd_fp_mul = 0;
    while(Read_rdy_fp_mul.rdy_fp_mul == 0);
    result = Read_res_fp_mul.res_fp_mul;
    return result;
}
/*=========================================================*/ // (op1 / op2)
unsigned int 32 fp_div(unsigned int 32 op1,unsigned int 32 op2)
{
    unsigned int 32 result;
    par
    {
        A_fp_div  = op1;
        B_fp_div  = op2;
        nd_fp_div = 1;
    }
    nd_fp_div = 0;
    while(Read_rdy_fp_div.rdy_fp_div == 0);
    result = Read_res_fp_div.res_fp_div;
    return result;
}
/*=========================================================*/

void main(void)
{
    unsigned int 32 f1,f2;  //Variables intermedias.
    unsigned int 32 lA,lw1,lw2;     //Variables para leer las entradas
    unsigned int 20 lR,lM, i1;
    unsigned int 32 fM, fMR;      //Variable para manejar el valor K como punto flotante


    //fitness(x) = w1 × A(x) + ( w2 ( M - R(x)) / M )
```

```
    while(1)
    {
        //---------------------------- //Esperamos mientras start_fitness es OFF
        while(Read_start_fitness.start_fitness == 0);
        //---------------------------- //Reseteamos las salidas
        par
        {
            rdy_fitness = 0;            //inicializamos las salidas
            res_fitness = 0b00000000000000000000000000000000;

            lA  = Read_A.A;   //Leemos las entradas
            lR  = Read_R.R;
            lM  = Read_M.M;

            lw1  = Read_w1.w1;
            lw2  = Read_w2.w2;
        }
        par
        {
            i1 = lM - lR;
            fM = int20_to_float(lM); // pasamos M a punto flotante
        }
        par
        {
            fMR = int20_to_float(i1); // pasamos (M -R(x)) a punto flotante
            f1  = fp_div(lw2,fM);     // dividimos w2/M
            f2  = fp_mul(lw1,lA);     // multiplicamos w1*A(x)
        }
        f1  = fp_mul(f1,fMR); // multiplicamos (w2 / M) * (M - R(x))

        res_fitness  = fp_addsub(f2,f1,0);  // sumamos (w1 * A(x)) + ((w2 / M) * (M - R(x)))

        rdy_fitness = 1;   //El controlador indica el fin del calculo
    }
}
```

Figura 19. Código Handel-C del controlador fitness_core (archivo fitness_core.hcc). Se han resaltado los bloques de ejecución paralela.

3.5 Implementación: Versión 2.

Los parámetros M, w1 y w2 son constantes, mientras que A y R son particulares de cada individuo en la población. En algunos artículos se indica que R es común a todos los individuos y su valor se fija al inicio del algoritmo (R es el número de genes seleccionado dentro de cada individuo).

En la implementación anterior (que llamamos Versión 1), teníamos A como variable. Hemos desarrollado una segunda implementación (que llamamos Versión 2) donde, además de A, dejamos también R como variable.

En los resultados experimentales, hemos observado que la versión 2 ofrece rendimientos bastante similares a la versión 1, en algunos casos mejores y en otros no, dependiendo del número de unidades paralelas de fitness, área ocupada, FPGA, etc.

3.5.1 Módulo de alto nivel: top2.

En la Figura 20 se observa la visión en alto nivel del circuito diseñado para medir el tiempo de computación. Este circuito tiene las mismas señales de E/S y componentes que en la versión 1, excepto el registro con el valor del operando R.

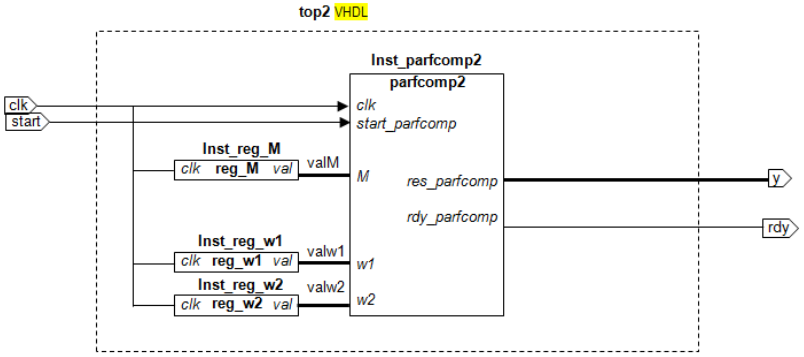

Figura 20. Componentes del módulo top2: 3 registros de solo lectura y el módulo parfcomp2 para la computación paralela de las operaciones del fitness.

El código VHDL de este módulo se describe en la Figura 13. Es un código de tipo estructural, que tan solo define la estructura del módulo de alto nivel.

3.5.2 Módulo de control paralelo: parfcomp2.

El módulo *parfcomp2* tiene el mismo propósito que el de la versión 1, y se define mediante un código VHDL estructural: *parfcomp2.vhd*. En la Figura 21 se observa la estructura de este módulo, que incorpora nuevos componentes: un divisor en coma flotante y un conversor de entero a coma flotante.

La razón de esta modificación es para optimizar el módulo de fitness, ya que ahora la operación aritmética w2/M se realiza en la unidad controladora, liberando de repetir en cada unidad de fitness esta operación. De esta forma, las nuevas unidades de fitness serán más rápidas y ocuparán menos área de la FPGA.

En la Figura 22 se muestra el código Handel-C del controlador correspondiente a la configuración más sencilla (*parfcomp2_core_NF-8_NC-4.hcc*).

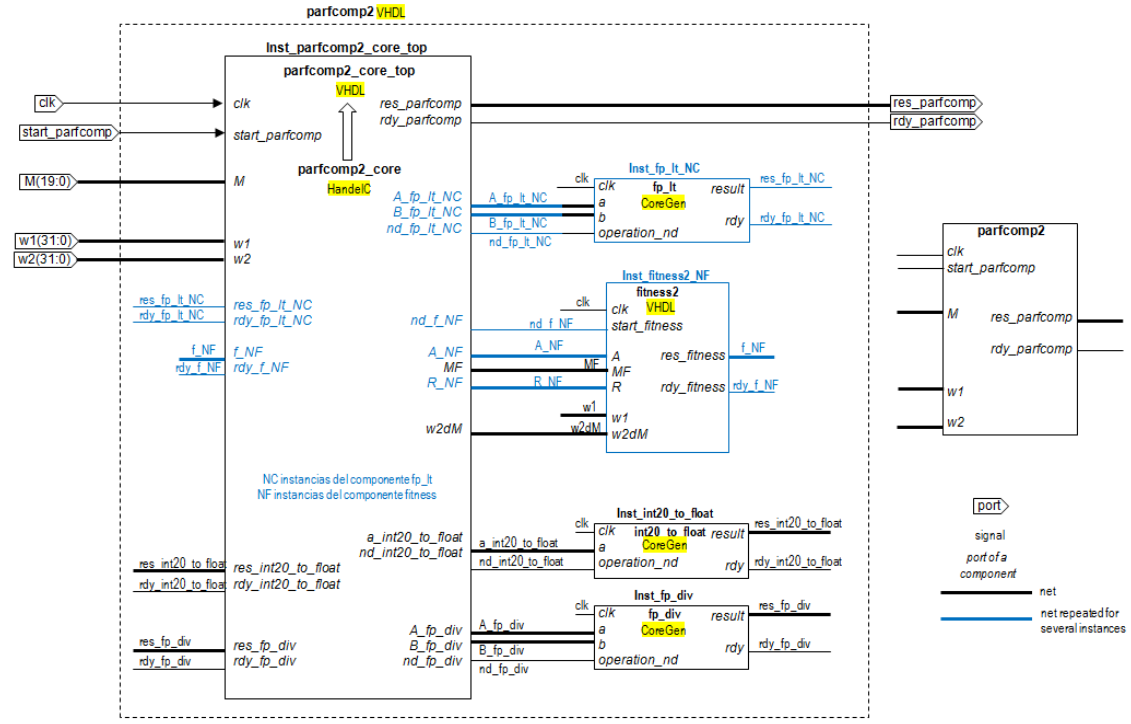

Figura 21. Componentes del módulo parfcomp2.

```
// parfcomp2_core.hcc
/*=====================================================*/
/* FUNCTION DECLARATION section                        */
unsigned int 32 fp_div(unsigned int 32 op1,
unsigned int 32 op2);
unsigned int 32 int20_to_float(unsigned int 20 op);
unsigned int 1 fp_lt_0(unsigned int 32 op1,
unsigned int 32 op2);
unsigned int 1 fp_lt_1(unsigned int 32 op1,
unsigned int 32 op2);
unsigned int 1 fp_lt_2(unsigned int 32 op1,
unsigned int 32 op2);
unsigned int 1 fp_lt_3(unsigned int 32 op1,
unsigned int 32 op2);
unsigned int 32 fitness_0(unsigned int 32 op1,
unsigned int 20 op2);
unsigned int 32 fitness_1(unsigned int 32 op1,
unsigned int 20 op2);
unsigned int 32 fitness_2(unsigned int 32 op1,
unsigned int 20 op2);
unsigned int 32 fitness_3(unsigned int 32 op1,
unsigned int 20 op2);
unsigned int 32 fitness_4(unsigned int 32 op1,
unsigned int 20 op2);
unsigned int 32 fitness_5(unsigned int 32 op1,
unsigned int 20 op2);
unsigned int 32 fitness_6(unsigned int 32 op1,
unsigned int 20 op2);
unsigned int 32 fitness_7(unsigned int 32 op1,
unsigned int 20 op2);
/*=====================================================*/
/* DEFINES section                                     */
#define NF 8
#define NC 4
/*=====================================================*/
/* GLOBAL VARIABLES section                            */
/*=====================================================*/
// I/O section
//                      |----------------|
//          clk -1-> |                |-32->res_parfcomp
//start_parfcomp -1-- |                |-1-> rdy_parfcomp
//                    | parfcomp2_core |
//          M -20--> |                |
//                    |                |
//         w1 -32--> |                |
//         w2 -32--> |                |
//                    |                |
// res_fp_lt_x --1--> | x (NC=4) ports | -32-> A_fp_lt_x
// rdy_fp_lt_x --1--> | para el        | -32-> B_fp_lt_x
//                    | componente     | -1-> nd_fp_lt_x
//                    | fp_lt          |
//                    |                |
//        f_y -32--> | y (NF=8) ports | --1-> nd_f_y
//     rdy_f_y --1--> | para el        | -32-> A_y
//                    | componente     | -32-> MF
//                    | fitness        | -32-> R_y
//                    |                |
//                    |                | -32-> w2dM
//                    |                |
//res_int20_to_float-32->|        |-20->a_int20_to_float
//rdy_int20_to_float--1->|        |--1->d_int20_to_float
//                    |                |
//  res_fp_div -32--> |                | -32-> A_fp_div
//  rdy_fp_div --1--> |                | -32-> B_fp_div
//                    |                | -1--> nd_fp_div
//                    |----------------|
/*------------------------------*/
/* INPUTS */
interface port_in (unsigned int 1  clk)  ClockPort();
set clock = internal ClockPort.clk;
//----//
interface port_in (unsigned int 1 start_parfcomp)
      Read_start_parfcomp();
//----//
interface port_in (unsigned int 20 M)
Read_M()  with {vhdl_type="std_logic_vector"};
//----//
interface port_in (unsigned int 32 w1)
Read_w1()  with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 w2)
Read_w2()  with {vhdl_type="std_logic_vector"};
//----//
interface port_in (unsigned int 1 res_fp_lt_0)
Read_res_fp_lt_0() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_0)
Read_rdy_fp_lt_0();
interface port_in (unsigned int 1 res_fp_lt_1)
Read_res_fp_lt_1() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_1)
Read_rdy_fp_lt_1();
interface port_in (unsigned int 1 res_fp_lt_2)
Read_res_fp_lt_2() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_2)
Read_rdy_fp_lt_2();
interface port_in (unsigned int 1 res_fp_lt_3)
Read_res_fp_lt_3() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1 rdy_fp_lt_3)
Read_rdy_fp_lt_3();
//----//
interface port_in (unsigned int 32 f_0)
Read_f_0()       with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_0) Read_rdy_f_0();
interface port_in (unsigned int 32 f_1)
Read_f_1()       with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_1) Read_rdy_f_1();
interface port_in(unsigned int 32 f_2)
Read_f_2() with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_2) Read_rdy_f_2();
interface port_in (unsigned int 32 f_3)
Read_f_3() with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_3) Read_rdy_f_3();
interface port_in(unsigned int 32 f_4)
Read_f_4() with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_4) Read_rdy_f_4();
interface port_in(unsigned int 32 f_5)
Read_f_5() with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_5) Read_rdy_f_5();
interface port_in(unsigned int 32 f_6)
```

```
Read_f_6() with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_6) Read_rdy_f_6();
interface port_in(unsigned int 32 f_7)
Read_f_7() with {vhdl_type="std_logic_vector"};
interface port_in(unsigned int 1 rdy_f_7) Read_rdy_f_7();
//----//
interface port_in(unsigned int 1  rdy_int20_to_float)
 Read_rdy_int20_to_float();
interface port_in(unsigned int 32 res_int20_to_float)
 Read_res_int20_to_float() with
{vhdl_type="std_logic_vector"};
//----//
interface port_in (unsigned int 1  rdy_fp_div)
 Read_rdy_fp_div();
interface port_in (unsigned int 32 res_fp_div)
 Read_res_fp_div() with {vhdl_type="std_logic_vector"};
//-----------------------------//
// OUTPUTS
//-----------------------------//
unsigned int 32 res_parfcomp; interface port_out ()
out_res_parfcomp (unsigned int 32
res_parfcomp=res_parfcomp) with
 {vhdl_type="std_logic_vector"};
//----//
unsigned int 1 rdy_parfcomp; interface port_out ()
out_rdy (unsigned int 1  rdy_parfcomp=rdy_parfcomp);
//----//
unsigned int 32 A_fp_lt_0;
interface port_out () out_A_fp_lt_0 (unsigned int 32
A_fp_lt_0=A_fp_lt_0) with {vhdl_type="std_logic_vector"};
unsigned int 32 A_fp_lt_1;
interface port_out () out_A_fp_lt_1 (unsigned int 32
A_fp_lt_1=A_fp_lt_1) with {vhdl_type="std_logic_vector"};
unsigned int 32 A_fp_lt_2;
interface port_out () out_A_fp_lt_2 (unsigned int 32
A_fp_lt_2=A_fp_lt_2) with {vhdl_type="std_logic_vector"};
unsigned int 32 A_fp_lt_3;
interface port_out () out_A_fp_lt_3 (unsigned int 32
A_fp_lt_3=A_fp_lt_3) with {vhdl_type="std_logic_vector"};
//----//
unsigned int 32 B_fp_lt_0;
interface port_out () out_B_fp_lt_0 (unsigned int 32
B_fp_lt_0=B_fp_lt_0) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_lt_1;
interface port_out () out_B_fp_lt_1 (unsigned int 32
B_fp_lt_1=B_fp_lt_1) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_lt_2;
interface port_out () out_B_fp_lt_2 (unsigned int 32
B_fp_lt_2=B_fp_lt_2) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_lt_3;
interface port_out () out_B_fp_lt_3 (unsigned int 32
B_fp_lt_3=B_fp_lt_3) with {vhdl_type="std_logic_vector"};
//----//
unsigned int 1 nd_fp_lt_0;
interface port_out () out_nd_fp_lt_0 (unsigned int 1
nd_fp_lt_0=nd_fp_lt_0);
unsigned int 1 nd_fp_lt_1;
interface port_out () out_nd_fp_lt_1 (unsigned int 1
nd_fp_lt_1=nd_fp_lt_1);
unsigned int 1 nd_fp_lt_2;
interface port_out () out_nd_fp_lt_2 (unsigned int 1
nd_fp_lt_2=nd_fp_lt_2);
unsigned int 1 nd_fp_lt_3;
interface port_out () out_nd_fp_lt_3 (unsigned int 1
nd_fp_lt_3=nd_fp_lt_3);
//----//
unsigned int 1 nd_f_0;
interface port_out () out_nd_f_0 (unsigned int 1
nd_f_0=nd_f_0);
unsigned int 1 nd_f_1;
interface port_out () out_nd_f_1 (unsigned int 1
nd_f_1=nd_f_1);
unsigned int 1 nd_f_2;
interface port_out () out_nd_f_2 (unsigned int 1
nd_f_2=nd_f_2);
unsigned int 1 nd_f_3;
interface port_out () out_nd_f_3 (unsigned int 1
nd_f_3=nd_f_3);
unsigned int 1 nd_f_4;
interface port_out () out_nd_f_4 (unsigned int 1
nd_f_4=nd_f_4);
unsigned int 1 nd_f_5;
interface port_out () out_nd_f_5 (unsigned int 1
nd_f_5=nd_f_5);
unsigned int 1 nd_f_6;
interface port_out () out_nd_f_6 (unsigned int 1
nd_f_6=nd_f_6);
unsigned int 1 nd_f_7;
interface port_out () out_nd_f_7 (unsigned int 1
nd_f_7=nd_f_7);
//----//
unsigned int 32 A_0;
interface port_out () out_A_0 (unsigned int 32 A_0=A_0)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_1;
interface port_out () out_A_1 (unsigned int 32 A_1=A_1)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_2;
interface port_out () out_A_2 (unsigned int 32 A_2=A_2)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_3;
interface port_out () out_A_3 (unsigned int 32 A_3=A_3)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_4;
interface port_out () out_A_4 (unsigned int 32 A_4=A_4)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_5;
interface port_out () out_A_5 (unsigned int 32 A_5=A_5)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_6;
interface port_out () out_A_6 (unsigned int 32 A_6=A_6)
with {vhdl_type="std_logic_vector"};
unsigned int 32 A_7;
interface port_out () out_A_7 (unsigned int 32 A_7=A_7)
with {vhdl_type="std_logic_vector"};
//-----//
unsigned int 32 MF;
interface port_out () out_MF (unsigned int 32 MF=MF) with
{vhdl_type="std_logic_vector"};
//-----//
unsigned int 20 R_0;
interface port_out () out_R_0 (unsigned int 20 R_0=R_0)
with {vhdl_type="std_logic_vector"};
unsigned int 20 R_1;
interface port_out () out_R_1 (unsigned int 20 R_1=R_1)
with {vhdl_type="std_logic_vector"};
unsigned int 20 R_2;
interface port_out () out_R_2 (unsigned int 20 R_2=R_2)
with {vhdl_type="std_logic_vector"};
unsigned int 20 R_3;
interface port_out () out_R_3 (unsigned int 20 R_3=R_3)
with {vhdl_type="std_logic_vector"};
unsigned int 20 R_4;
interface port_out () out_R_4 (unsigned int 20 R_4=R_4)
with {vhdl_type="std_logic_vector"};
unsigned int 20 R_5;
interface port_out () out_R_5 (unsigned int 20 R_5=R_5)
with {vhdl_type="std_logic_vector"};
unsigned int 20 R_6;
interface port_out () out_R_6 (unsigned int 20 R_6=R_6)
with {vhdl_type="std_logic_vector"};
unsigned int 20 R_7;
interface port_out () out_R_7 (unsigned int 20 R_7=R_7)
with {vhdl_type="std_logic_vector"};
//----//
unsigned int 32 w2dM;
interface port_out () out_w2dM (unsigned int 32
w2dM=w2dM) with {vhdl_type="std_logic_vector"};
//----//
unsigned int 20 a_int20_to_float;
interface port_out () out_a_int20_to_float (unsigned int
20 a_int20_to_float=a_int20_to_float) with
{vhdl_type="std_logic_vector"};
unsigned int 1 nd_int20_to_float;
interface port_out () out_nd_int20_to_float (unsigned int
1 nd_int20_to_float=nd_int20_to_float);
//----//
unsigned int 32 A_fp_div;
interface port_out () out_A_fp_div (unsigned int 32
A_fp_div=A_fp_div) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_div;
interface port_out () out_B_fp_div (unsigned int 32
B_fp_div=B_fp_div) with {vhdl_type="std_logic_vector"};
unsigned int 1 nd_fp_div;
interface port_out () out_nd_fp_div (unsigned int 1
nd_fp_div=nd_fp_div);

/*================================================*/
unsigned int 32 int20_to_float(unsigned int 20 op)
{
  unsigned int 32 result;
  par { a_int20_to_float = op; nd_int20_to_float = 1; }
  nd_int20_to_float = 0;
  while(Read_rdy_int20_to_float.rdy_int20_to_float == 0);
  result = Read_res_int20_to_float.res_int20_to_float;
  return result;
}
/*====================================*/ // (op1 / op2)
unsigned int 32 fp_div(unsigned int 32 op1,unsigned int
32 op2)
{
  unsigned int 32 result;
  par { A_fp_div = op1; B_fp_div = op2; nd_fp_div = 1; }
  nd_fp_div = 0;
  while(Read_rdy_fp_div.rdy_fp_div == 0);
  result = Read_res_fp_div.res_fp_div;
  return result;
}
/*================================================*/
unsigned int 1 fp_lt_0(unsigned int 32 op1,unsigned int
 32 op2)
  {
  unsigned int 1 result;
  par { A_fp_lt_0=op1; B_fp_lt_0=op2; nd_fp_lt_0=1; }
  nd_fp_lt_0 = 0;
  while(Read_rdy_fp_lt_0.rdy_fp_lt_0 == 0);
  result = Read_res_fp_lt_0.res_fp_lt_0; return(result);
  }
unsigned int 1 fp_lt_1(unsigned int 32 op1,unsigned int
32 op2)
  {
  unsigned int 1 result;
  par { A_fp_lt_1=op1; B_fp_lt_1=op2; nd_fp_lt_1=1; }
  nd_fp_lt_1 = 0;
  while(Read_rdy_fp_lt_1.rdy_fp_lt_1 == 0);
  result = Read_res_fp_lt_1.res_fp_lt_1; return(result);
  }
unsigned int 1 fp_lt_2(unsigned int 32 op1,unsigned int
```

```
32 op2)
  {
  unsigned int 1 result;
  par { A_fp_lt_2= p1; B_fp_lt_2=op2; nd_fp_lt_2= ; }
  nd_fp_lt_2 = 0;
  while(Read_rdy_fp_lt_2.rdy_fp_lt_2 == 0);
  result = Read_res_fp_lt_2.res_fp_lt_2; return(result);
  }
unsigned int 1 fp_lt_3(unsigned int 32 op1,unsigned int
32 op2)
  {
  unsigned int 1 result;
  par { A_fp_lt_3= p1; B_fp_lt_3=op2; nd_fp_lt_3=1; }
  nd_fp_lt_3 = 0;
  while(Read_rdy_fp_lt_3.rdy_fp_lt_3 == 0);
  result = Read_res_fp_lt_3.res_fp_lt_3; return(result);
  }
/*=====================================================*/
unsigned int 32 fitness_0(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_0 = op1; R_0 = op2; nd_f_0 = 1; }
  nd_f_0 = 0;
  while(Read_rdy_f_0.rdy_f_0 == 0);
  result = Read_f_0.f_0;  return(result);
  }
unsigned int 32 fitness_1(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_1 = op1; R_1 = op2; nd_f_1 = 1; }
  nd_f_1 = 0;
  while(Read_rdy_f_1.rdy_f_1 == 0);
  result = Read_f_1.f_1;  return(result);
  }
unsigned int 32 fitness_2(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_2 = op1; R_2 = op2; nd_f_2 = 1; }
  nd_f_2 = 0;
  while(Read_rdy_f_2.rdy_f_2 == 0);
  result = Read_f_2.f_2;  return(result);
  }
unsigned int 32 fitness_3(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_3 = op1; R_3 = op2; nd_f_3 = 1; }
  nd_f_3 = 0;
  while(Read_rdy_f_3.rdy_f_3 == 0);
  result = Read_f_3.f_3;  return(result);
  }
unsigned int 32 fitness_4(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_4 = op1; R_4 = op2; nd_f_4 = 1; }
  nd_f_4 = 0;
  while(Read_rdy_f_4.rdy_f_4 == 0);
  result = Read_f_4.f_4;  return(result);
  }
unsigned int 32 fitness_5(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_5 = op1; R_5 = op2; nd_f_5 = 1; }
  nd_f_5 = 0;
  while(Read_rdy_f_5.rdy_f_5 == 0);
  result = Read_f_5.f_5;  return(result);
  }
unsigned int 32 fitness_6(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_6 = op1; R_6 = op2; nd_f_6 = 1; }
  nd_f_6 = 0;
  while(Read_rdy_f_6.rdy_f_6 == 0);
  result = Read_f_6.f_6;  return(result);
  }
unsigned int 32 fitness_7(unsigned int 32 op1,unsigned
int 20 op2)
  {
  unsigned int 32 result;
  par { A_7 = op1; R_7 = op2; nd_f_7 = 1; }
  nd_f_7 = 0;
  while(Read_rdy_f_7.rdy_f_7 == 0);
  result = Read_f_7.f_7;  return(result);
  }
/*=====================================================*/
void main(void)
{
unsigned int 20 lM, R[NF];
unsigned int 32 lw1,lw2,f[NF],A[NF],Abest[NC],fmin[NC];
unsigned int  1 lt[NC];
//---------------------------- //
while(1)
    {
    //---------------- //Espera mientras start OFF
    while(Read_start_parfcomp.start_parfcomp == 0);
    //----------------- //Resetea salidas y lee entradas
    par {
       rdy_parfcomp = 0;
       res_parfcomp = 0b00000000000000000000000000000000;
       lM  = Read_M.M;
       lw1 = Read_w1.w1;
       lw2 = Read_w2.w2;
       }
    //----------------------------
    // Para pruebas de rendimiento, mismo valor A y R
    par {
        A[0]  = 0b01000000001110011001100110011010;
        A[1]  = 0b01000000001110011001100110011010;
        A[2]  = 0b01000000001110011001100110011010;
        A[3]  = 0b01000000001110011001100110011010;
        A[4]  = 0b01000000001110011001100110011010;
        A[5]  = 0b01000000001110011001100110011010;
        A[6]  = 0b01000000001110011001100110011010;
        A[7]  = 0b01000000001110011001100110011010;
        R[0]  = 15;     R[1]  = 15;
        R[2]  = 15;     R[3]  = 15;
        R[4]  = 15;     R[5]  = 15;
        R[6]  = 15;     R[7]  = 15;
        }
    //---------------------------- // Calcula MF y w2dM
    MF   = int20_to_float(lM); // MF (float)=M (integer)
    w2dM = fp_div(lw2,MF);     // w2/MF
    //---------------------------- //
    par {              // f(y) = w1*A(y) + w2*(M-R(y))/M
        f[0]  = fitness_0(A[0],R[0]);
        f[1]  = fitness_1(A[1],R[1]);
        f[2]  = fitness_2(A[2],R[2]);
        f[3]  = fitness_3(A[3],R[3]);
        f[4]  = fitness_4(A[4],R[4]);
        f[5]  = fitness_5(A[5],R[5]);
        f[6]  = fitness_6(A[6],R[6]);
        f[7]  = fitness_7(A[7],R[7]);
        }
    //----------------- // Primera fase de comparaciones
    par {
        lt[0] =fp_lt_0(f[0],f[1]);
        lt[1] =fp_lt_0(f[1],f[2]);
        lt[2] =fp_lt_0(f[2],f[3]);
        lt[3] =fp_lt_0(f[3],f[4]);
        }
    par {
        { if(lt[0]==1) { fmin[0]=f[0];  Abest[0]=A[0]; }
          else         { fmin[0]=f[1];  Abest[0]=A[1]; } }
        { if(lt[1]==1) { fmin[1]=f[2];  Abest[1]=A[2]; }
          else         { fmin[1]=f[3];  Abest[1]=A[3]; } }
        { if(lt[2]==1) { fmin[2]=f[4];  Abest[2]=A[4]; }
          else         { fmin[2]=f[5];  Abest[2]=A[5]; } }
        { if(lt[3]==1) { fmin[3]=f[6];  Abest[3]=A[6]; }
          else         { fmin[3]=f[7];  Abest[3]=A[7]; } }
        }
    //---------------------------- //
    par {
        lt[0]=fp_lt_0(fmin[0],fmin[1]);
        lt[1]=fp_lt_1(fmin[2],fmin[3]);
        }
    par {
        { if(lt[0]==1)
              { fmin[0]=fmin[0]; Abest[0]=Abest[0]; }
          else  fmin[0]=fmin[1]; Abest[0]=Abest[1]; } }
        { if(lt[1]==1)
              { fmin[1]=fmin[2]; Abest[1]=Abest[2]; }
          else { fmin[1]=fmin[3]; Abest[1]=Abest[3]; } }
        }
    //---------------------------- //
    par {
        lt[0]=fp_lt_0(fmin[0],fmin[1]);
        }
    par {
        { if(lt[0]==1)
              { fmin[0]=fmin[0]; Abest[0]=Abest[0]; }
          else { fmin[0]=fmin[1]; Abest[0]=Abest[1]; } }
        }
    //---------------------------- //
    res_parfcomp = fmin[0];
    res_parfcomp = Abest[0];
    rdy_parfcomp = 1;   //Indica el fin del calculo
    }
}
```

Figura 22. Código Handel-C del módulo controlador parfcomp2_core, para la configuración más sencilla, NF=8, NC=4: parfcom2p_core_NF-8_NC-4.hcc. Se resaltan los bloques de ejecución paralela.

3.5.3 Módulo fitness2.

El módulo *fitness2* tiene como propósito implementar las operaciones aritméticas de la función F, pero está más optimizado que en la versión 1, ya que ahorra operaciones aritméticas que se realizan en el controlador, y que son comunes a todos los cálculos del fitness independientemente del individuo sobre el que se aplique. En la Figura 23 puede observarse la arquitectura de este módulo, que está definido mediante un código VHDL estructural, *fitness2.vhd*.

En la Figura 24 se puede observar el código Handel-C del módulo controlador del fitness2, *fitness2_core*, que regula las operaciones del mismo, y en el que se han resaltado los bloques de ejecución paralela.

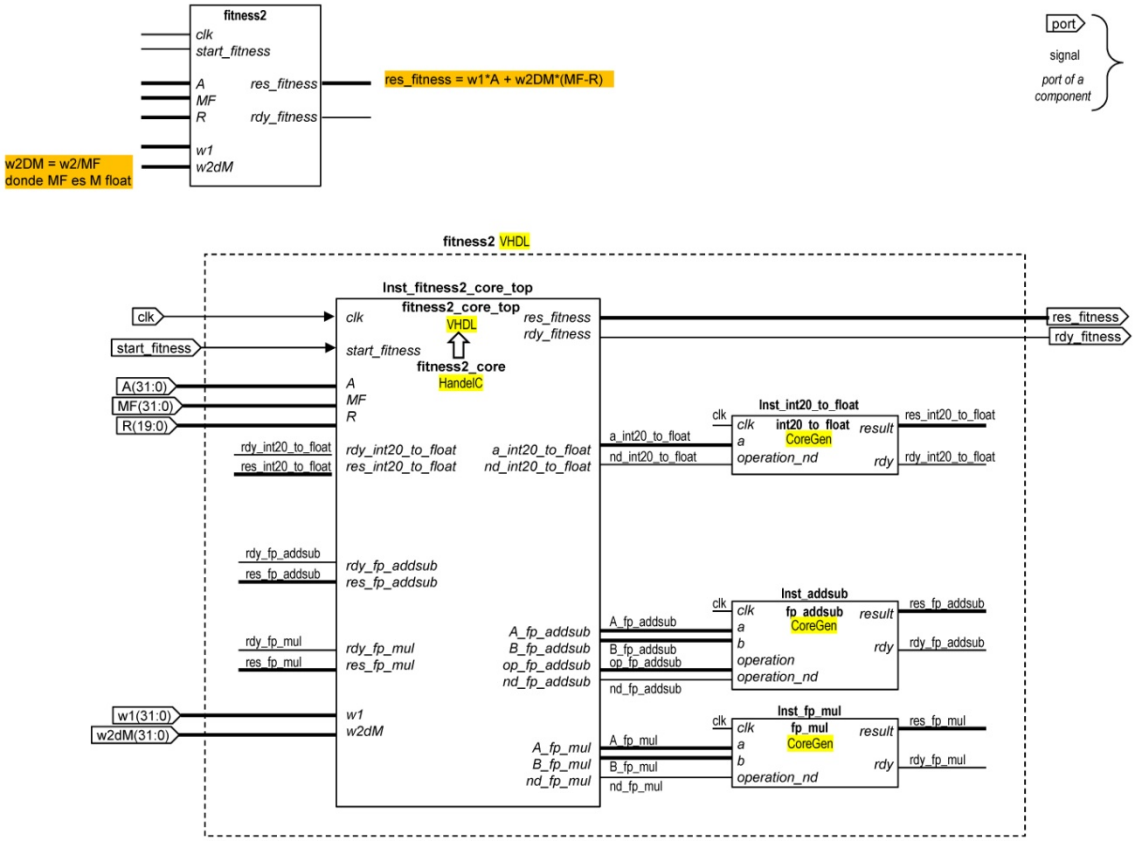

Figura 23. Componentes del módulo fitness2.

```
// fitness2_core.hcc
/*=====================================================*/

/*=====================================================*/
/* FUNCTION DECLARATION section                        */
unsigned int 32 fp_mul(unsigned int 32 op1,unsigned int 32 op2);
unsigned int 32 fp_div(unsigned int 32 op1,unsigned int 32 op2);
unsigned int 32 fp_addsub(unsigned int 32 op1,unsigned int 32 op2,unsigned int 6 operation);
unsigned int 32 int20_to_float(unsigned int 20 op);

/*=====================================================*/
/* DEFINES section                                     */
/*=====================================================*/
/* GLOBAL VARIABLES section                            */
/*=====================================================*/
// I/O section
//                            |---------------|
//                clk ---1--> |       -       | --32--> res_fitness
//       start_fitness ---1--> |               | ---1--> rdy_fitness
//                            | fitness2_core |
//                   A -32--> |               |
//                  MF -32--> |               |
//                   R -20--> |               |
//                            |               |
//   rdy_int20_to_float --1--> |               | --20--> a_int20_to_float
//   res_int20_to_float -32--> |               | ---1--> nd_int20_to_float
//                            |               |
```

```
//                              |               |
//        rdy_fp_addsub --1--> |               | --32--> A_fp_addsub
//        res_fp_addsub -32--> |               | --32--> B_fp_addsub
//                              |               | ---1--> nd_fp_addsub
//                              |               | ---6--> op_fp_addsub
//                              |               |
//           rdy_fp_mul --1--> |               | --32--> A_fp_mul
//           res_fp_mul -32--> |               | --32--> B_fp_mul
//                              |               | ---1--> nd_fp_mul
//                    w1 -32--> |               |
//                  w2dM -32--> |               |
//                             |---------------|
//
/*----------------------------*/
/* INPUTS */
interface port_in (unsigned int 1  clk)  ClockPort();
set clock = internal ClockPort.clk;
interface port_in (unsigned int 1 start_fitness) Read_start_fitness();
interface port_in (unsigned int 32 A) Read_A() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 MF) Read_MF() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 20 R) Read_R() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_int20_to_float) Read_rdy_int20_to_float();
interface port_in (unsigned int 32 res_int20_to_float) Read_res_int20_to_float() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_fp_addsub) Read_rdy_fp_addsub();
interface port_in (unsigned int 32 res_fp_addsub) Read_res_fp_addsub() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 1  rdy_fp_mul) Read_rdy_fp_mul();
interface port_in (unsigned int 32 res_fp_mul) Read_res_fp_mul() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 w1) Read_w1() with {vhdl_type="std_logic_vector"};
interface port_in (unsigned int 32 w2dM) Read_w2dM() with {vhdl_type="std_logic_vector"};
//----------------------------//
// OUTPUTS
//----------------------------//
unsigned int 32 res_fitness;
interface port_out () out_res_fitness (unsigned int 32 res_fitness=res_fitness) with {vhdl_type="std_logic_vector"};
unsigned int 1 rdy_fitness;
interface port_out () out_rdy_fitness (unsigned int 1 rdy_fitness=rdy_fitness);
unsigned int 20 a_int20_to_float;
interface   port_out   ()   out_a_int20_to_float   (unsigned   int   20   a_int20_to_float=a_int20_to_float)   with
{vhdl_type="std_logic_vector"};
unsigned int 1 nd_int20_to_float;
interface port_out () out_nd_int20_to_float (unsigned int 1 nd_int20_to_float=nd_int20_to_float);
unsigned int 32 A_fp_addsub;
interface port_out () out_A_fp_addsub (unsigned int 32 A_fp_addsub=A_fp_addsub) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_addsub;
interface port_out () out_B_fp_addsub (unsigned int 32 B_fp_addsub=B_fp_addsub) with {vhdl_type="std_logic_vector"};
unsigned int 1 nd_fp_addsub;
interface port_out () out_nd_fp_addsub (unsigned int 1 nd_fp_addsub=nd_fp_addsub);
unsigned int 6 op_fp_addsub;
interface port_out () out_op_fp_addsub (unsigned int 6 op_fp_addsub=op_fp_addsub) with {vhdl_type="std_logic_vector"};
unsigned int 32 A_fp_mul;
interface port_out () out_A_fp_mul (unsigned int 32 A_fp_mul=A_fp_mul) with {vhdl_type="std_logic_vector"};
unsigned int 32 B_fp_mul;
interface port_out () out_B_fp_mul (unsigned int 32 B_fp_mul=B_fp_mul) with {vhdl_type="std_logic_vector"};
unsigned int 1 nd_fp_mul;
interface port_out () out_nd_fp_mul (unsigned int 1 nd_fp_mul=nd_fp_mul);
/*=========================================================*/
unsigned int 32 int20_to_float(unsigned int 20 op)
{
    unsigned int 32 result;
    par
    {
        a_int20_to_float  = op;
        nd_int20_to_float = 1;
    }
    nd_int20_to_float = 0;
    while(Read_rdy_int20_to_float.rdy_int20_to_float == 0);
    result = Read_res_int20_to_float.res_int20_to_float;
    return result;
}
/*=========================================================*/ // (op1 +/- op2)
unsigned int 32 fp_addsub(unsigned int 32 op1,unsigned int 32 op2,unsigned int 6 operation)
{
    unsigned int 32 result;
    par
    {
        A_fp_addsub  = op1;
        B_fp_addsub  = op2;
        op_fp_addsub = operation;
        nd_fp_addsub = 1;
    }
    nd_fp_addsub = 0;
    while(Read_rdy_fp_addsub.rdy_fp_addsub == 0);
    result = Read_res_fp_addsub.res_fp_addsub;
    return result;
}
/*=========================================================*/ // (op1 * op2)
unsigned int 32 fp_mul(unsigned int 32 op1,unsigned int 32 op2)
{
    unsigned int 32 result;
    par
    {
        A_fp_mul  = op1;
        B_fp_mul  = op2;
        nd_fp_mul = 1;
    }
    nd_fp_mul = 0;
    while(Read_rdy_fp_mul.rdy_fp_mul == 0);
    result = Read_res_fp_mul.res_fp_mul;
    return result;
}


/*=========================================================*/
void main(void)
```

```
{
    unsigned int 32 f1,f2,fR;  //Variables intermedias.
    unsigned int 32 lA,lMF,lw1,lw2dM;     //Variables para leer las entradas
    unsigned int 20 lR;


    //fitness(x) = w1 × A(x) + ( w2 ( M - R(x)) / M )

    while(1)
    {
        //----------------------------- //Esperamos mientras start_fitness es OFF
        while(Read_start_fitness.start_fitness == 0);
        //----------------------------- //Reseteamos las salidas
        par
        {
            rdy_fitness = 0;     //inicializamos las salidas
            res_fitness = 0b00000000000000000000000000000000;
            lA    = Read_A.A;   //Leemos las entradas
            lR    = Read_R.R;
            lMF   = Read_MF.MF;
            lw1   = Read_w1.w1;
            lw2dM = Read_w2dM.w2dM;
        }

        fR = int20_to_float(lR); // pasamos R(x) a punto flotante
        par
        {
            f1 = fp_mul(lw1,lA);      // w1*A(x)
            f2 = fp_addsub(lMF,fR,1); // MF-R
        }
        f2  = fp_mul(lw2dM,f2); // w2dM*(MF-R)
        res_fitness  = fp_addsub(f1,f2,0);  // w1*A(x) + w2dM*(MF-R(x))
        rdy_fitness = 1;   // Indica el fin del calculo
    }
}
```

Figura 24. Código Handel-C del controlador fitness2_core (archivo fitness2_core.hcc). Se han resaltado los bloques de ejecución paralela.

3.6 Síntesis e implementación del circuito.

El circuito se sintetiza e implementa en Xilinx ISE mediante la siguiente sucesión:

- Fase de síntesis.
- Fase de implementación.
  - Traslación (*translate*),
  - Correspondencia (*map*).
  - Emplazamiento y encaminado (*place and route*).

En esta última fase, mediante el proceso "*Generate Post-Place & Route Simulation Model*", se genera un informe donde se especifica cuál es la frecuencia de reloj máxima (y por tanto, cuál es el periodo de reloj mínimo) a la que puede operar el circuito.

Al finalizar el proceso de implementación, se genera un informe global (un resumen de un conjunto de informes individuales) donde podemos obtener la información del grado de ocupación de la FPGA.

Específicamente, los datos más importantes que recogemos de los informes serán:

- *slice registers* (%).
- *slice LUTs* (%).
- *occupied slices*(%).
- *min.period* (ns).
- *max.freq*. (MHz).

donde los tres primeros hacen referencia al área ocupada, y los dos últimos a la velocidad de operación del circuito.

El ajuste de los parámetros del proceso de síntesis puede derivar en distintos valores obtenidos del área y la velocidad del circuito en la FPGA. Por tanto, hemos repetido cada síntesis e implementación tres veces por experimento, para cada una de las siguientes opciones de síntesis:

- Opción DEF: *Default*.
- Opción TPP: *Timing Prformance - Performance with Physical Synthesis*.
- Opción TPN: *Timing Performance - Performance without IOB packing*.

Para finalizar, destacamos que cada proceso de síntesis e implementación tarda mucho tiempo, incluso en CPUs muy potentes en ordenadores con mucha memoria. Este tiempo ha variado entre 30 minutos y 10 horas, dependiendo del número de unidades paralelas de fitness, NF, que se haya considerado en el experimento. Esto nos ha obligado a utilizar varios ordenadores simultáneamente, donde cada uno lanzaba la implementación de una configuración distinta de los experimentos.

3.6.1 Simulación y medida del tiempo.

Para simular el diseño (en ambas versiones) y obtener la medida del tiempo de computación, creamos un banco de pruebas o *testbench* para generar las simulaciones en Xilinx ISE, a través del archivo en código VHDL "*test_top.vhd*" (Figura 25). Sobre este *testbench*, hemos de hacer las siguientes consideraciones:

- La simulación del circuito mediante este testbench se hace después de sintetizar el circuito, para hacer no solo una simulación funcional, sino también de tiempos.
- El valor de la constante *clk_period* es fijado de acuerdo con el informe de síntesis de ISE, donde se obtiene la máxima frecuencia de reloj (y por tanto el mínimo periodo de reloj) al que puede operar el circuito.
- Este *testbench* genera un pulso de un ciclo para iniciar el circuito con la señal *start*.
- Este *testbench* genera un pulso de un ciclo para iniciar el circuito con la señal *start*.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY test_top IS
END test_top;

ARCHITECTURE behavior OF test_top IS

    -- Component Declaration for Unit Under Test (UUT)
    COMPONENT top
    PORT(
         clk : IN  std_logic;
         start : IN  std_logic;
         y : OUT  std_logic_vector(31 downto 0);
         rdy : OUT  std_logic
        );
    END COMPONENT;

   --Inputs
   signal clk : std_logic := '0';
   signal start : std_logic := '0';
     --Outputs
   signal y : std_logic_vector(31 downto 0);
   signal rdy : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: top PORT MAP (
          clk => clk,
          start => start,
          y => y,
          rdy => rdy
        );

   -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

      wait for clk_period*10;

      -- insert stimulus here -----------------
        start <= '0';
        start <= '1';
        wait for clk_period;
        --wait for 10 ns;
        start <= '0';
      ------------------------ -----------------
      wait;
   end process;

END;
```

Figura 25. Código del testbench para simulaciones: test_top.vhd. El valor de la constante clk_period se fijado según el informe de síntesis, donde se obtiene el mínimo periodo de reloj al que puede operar el circuito.

En la Figura 26 se puede observar un ejemplo de simulación, concretamente de una unidad de fitness. En este ejemplo, el *testbench* se simuló fijando un valor *de clk_period* de **3.9 ns**, que fue el reportado en la síntesis. Una vez simulado, el tiempo de cálculo de F se mide desde la activación de la señal *start* (*start*=1) hasta la activación de la señal *rdy* (*rdy*=1): =392.25ns - 139ns = **253.25ns**.

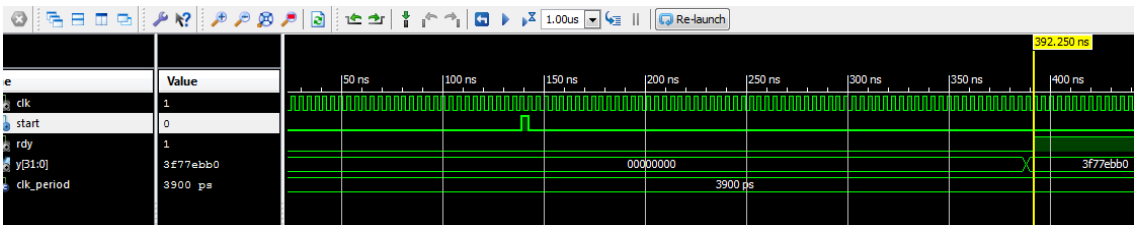

Figura 26. Ejemplo de simulación de la unidad fitness.

3.6.2 Medida del consumo energético.

El consumo de energía del circuito se realiza durante la fase de implementación, mediante la herramienta Xilinx Xpower Analyzer (Figura 27). Esta herramienta toma los valores de ocupación, emplazado y rutado de los bloques de lógica configurable de la FPGA para calcular la energía consumida por el circuito.

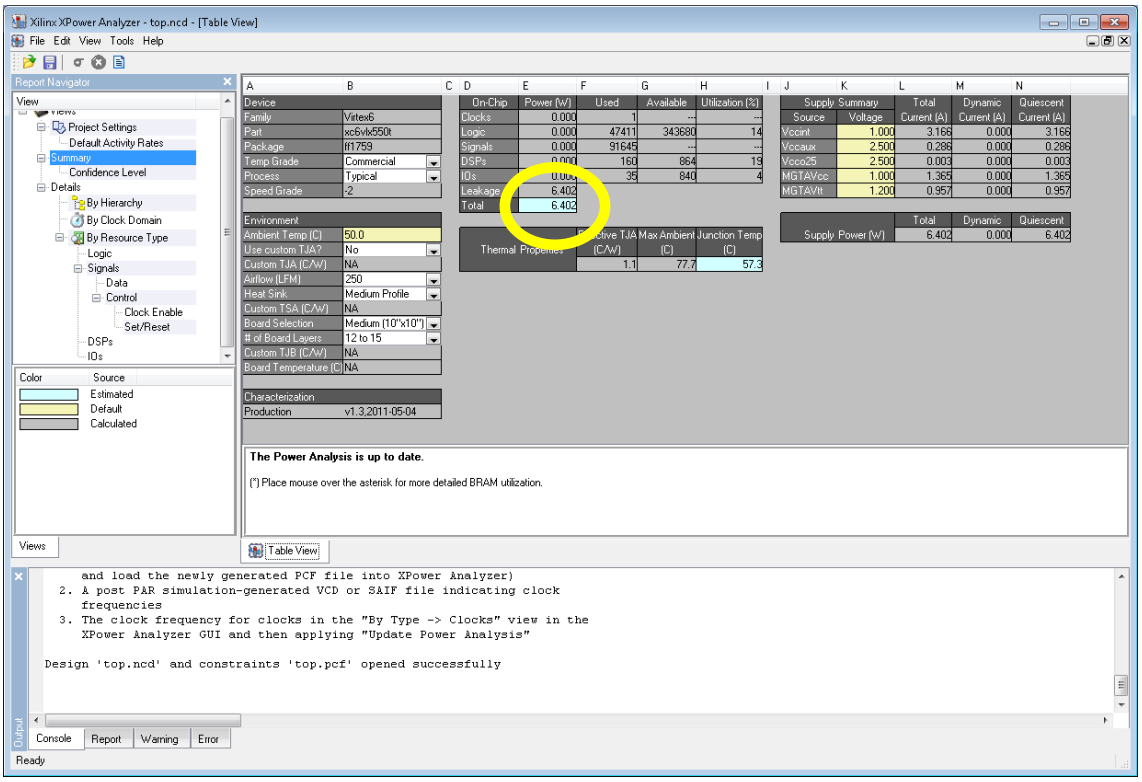

Figura 27. Herramienta Xilinx Xpower Analyzer. Se señala el valor del consumo energético, expresado en watios, del circuito implementado en una FPGA determinada.

3.7 Experimentos de implementación y síntesis.

Se han realizado numerosos experimentos para calcular el rendimiento, según cuatro grupo de configuraciones:

- Grupo 1:
  - Implementación versión 1.
  - Implementación versión 2.
- Grupo 2:
  - NF=8, NC=4.
  - NF=16, NC=8.
  - NF=32, NC=16.
  - NF=64, NC=32.
  - NF=100, NC=50.
  - NF=128, NC=64.
  - NF=256, NC=128.
- Grupo 3:
  - FPGA xc5vlx330-1ff1760.
  - FPGA xc6vlx550t-2ff1759
  - FPGA xc6slx150-3fgg676
- Grupo 4:
  - Opción de síntesis DEF.
  - Opción de síntesis TPP.
  - Opción de síntesis TPN.

Por tanto, se han realizado 2x7x3x3 = **126 experimentos**, donde cada experimento implica un proceso de síntesis e implementación en ISE. Para afrontar este elevado número de experimentos, cuyas síntesis pueden durar horas, se han utilizado, además de computadores personales, dos servidores: **hprc.unex.es** y **arcolab.unex.es**, controlados por escritorio remoto.

Para cada uno de estos experimentos, se han anotado los valores de:

- slice registers (%).
- slice LUTs (%).
- occupied slices(%).
- min.period (ns).
- max.freq. (MHz).

En base a la ocupación, calculamos una aproximación al número de posibles unidades paralelas. Además, para cada experimento, realizamos una simulación, ajustando en el archivo *testbench* el periodo de reloj al mínimo periodo reportado

anteriormente, y calculamos el tiempo de cálculo de F (tiempo desde que la start=1 hasta que rdy=1).

En algunos casos (dependiendo de la FPGA y del número de unidades paralelas) se han generado los módulos aritméticos. en coma flotante sin la opción de utilización de los DSPs internos para ahorrar espacio. Estos elementos internos aumentan la velocidad, pero ocupan mucho espacio, por lo que hubo que desactivarlos para cierts configuraciones. Aun así, algunas de las últimas configuraciones y para FPGAs de bajas prestaciones, los diseños exigen más área de la proporcionada por la FPGA, por lo que no se han podido completar los experimentos.

En la Tabla 3 se muestran los resultados obtenidos con la configuración experimental que proporciona los mejores valores. Los resultados de esta configuración los utilizaremos para compararlos con los obtenidos por una implementación software que efectúa las mismas operaciones que el circuito acelerador, con el propósito de evaluar el rendimiento de éste.

Unidades fitness paralelas (NF)	128
Unidades comparadoras paralelas (NC)	64
Versión de la implementación	1
Dispositivo FPGA	xc6vlx550t-2ff1759
Opción de síntesis ISE	Timing Performance with Physical Synthesis
Registros ocupados (%)	35%
Memorias LUTs ocupadas (%)	52%
Slices ocupados (%)	69%
DSP48Es ocupados[2] (%)	74%
Periodo mínimo de reloj (ns)	7.8
Máxima frecuencia de reloj (MHz)	128
Tiempo de computación[3] (ns)	936
Energía consumida (W)	6.4

Tabla 3. Configuración experimental con los mejores resultados.

[2] Estos elementos son utilizados por los cores aritméticos en coma flotante generados mediante la herramienta Core Generator.

[3] Tiempo de computación del bloque de NF unidades paralelas del cálculo del fitness.

3.8 Versión software.

Con el objeto de evaluar el rendimiento del circuito acelerador, se ha desarrollado una versión software del cálculo paralelo del fitness que reproduce las mismas operaciones efectuadas por el circuito, de forma que puedan establecerse comparativas adecuadas.

3.8.1 Software secuencial.

En la primera columna de la Figura 28 se muestra el código de la versión software del circuito, de ejecución secuencial en la CPU de un computador, según el archivo *selgencancer_main.c*. Este código C ejecuta un número de iteraciones dado, *contmax*, de un **bloque** de instrucciones que calculan NF funciones de fitness (Figura 29). El cálculo NF veces de la función de fitness (un bloque) emula la operación de NF módulos paralelos en el circuito; como este cálculo es muy rápido, lo repetimos muchas veces para poder medir el tiempo en segundos. Es decir:

- *TPC* = tiempo (s) software de *contmax* iteraciones.
- *XPC* = *TPC*/*contmax* x 10e9 = tiempo (ns) de una sola iteración software.
- *Y* = tiempo (ns) obtenido en las simulaciones del circuito tras su síntesis e implementación.
- Factor de aceleración (*speedup*) FPGA vs CPU = *XPC* / Y.

Para el cálculo del tiempo se ha utilizado la función *clock*() de ANSI C.

3.8.2 Software paralelo.

En la segunda columna de la Figura 28 se muestra la versión de ejecución paralela del mismo código, programado mediante las librerías OpenMP [OpenMP2015] para utilizar hilos de ejecución paralela en las CPUs multinúcleo: *selgencancer-OpenMP_main.c*.

Se ha utilizado una CPU con las siguientes características:

- Procesador: Intel(R) Core(TM) i7-3517U CPU @ 1.90GHz
- Número de núcleos 2
- Número de hilos: 4
- Velocidad de reloj: 1.9 GHz
- Caché: 4 MB.
- TDP[4]: 17 W.

Este código ejecuta *contmax*/4 iteraciones, lanzándose en cada una de ellas 4 hilos de ejecución del bloque de NF cálculos de fitness (Figura 30). Por ejemplo, si *contmax* = 10^7 = 10.000.000 ejecuciones del bloque en la versión secuencial, en la versión paralela se ejecutan $10^7/4$ = 2.500.000 ejecuciones de 4 bloques paralelos.

Para calcular el tiempo no hemos utilizado clock(), porque cuenta el tiempo acumulado por todas las CPUs. En su lugar, utilizamos la función omp_get_wtime() de OpenMP.

[4] TDP = potencia de diseño térmico (*thermal design power*). Es la potencia máxima requerida para disipar el calor generado por la CPU.

C secuencial (selgencancer_main_seq.c)

```
//====================================//
// fitness
//====================================//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

//====================================//
#define NF 8
//Probar NF con: 8, 16, 32, 64, 100, 128
#define M 48
#define W1 0.2
#define W2 (1 - W1)
//====================================//
float getFitness(float A,float R);
//====================================//
int main(void)
{
clock_t      t_start,t_finish;
             // Marcas de medidas de tiempo
float        t_elapsed;
unsigned int cont,contmax;

int i;

//-------------------//
float A[NF],R[NF];
float fitness[NF];
float fitness_min,A_best;
//-------------------//
contmax=1E+7;

t_start=clock();


//-------------------//
for(cont=0; cont<contmax; cont++)
{
//contmax iteraciones del cálculo del
//conjunto de NF unidades fitness

    for(i=0; i<NF; i++)
    //doy entradas a las NF unidades fitness
    {
        A[i]=3.4;
        R[i]=4.07;
    }

    for(i=0; i<NF; i++)
    //calculo los NF fitness
    {
        fitness[i] = getFitness(A[i],R[i]);
    }

    fitness_min=fitness[0];
    //determino fitness mínimo

    for(i = 1; i < NF; i++)
    {
        if(fitness[i] < fitness_min)
        {
            fitness_min=fitness[i];
        }
    }
}
//-------------------//
t_finish=clock();
t_elapsed=(float)(t_finish-t_start)/CLOCKS_PER_SEC;

printf("fitness_min=%f\n",fitness_min);
printf("Time = %g seconds\n",t_elapsed);
//-------------------//
printf("=== FIN ===\n");
return(1);
}
//====================================//
float getFitness(float A,float R)
//Fitness = F(x) = w1·A(x)+(w2·(M-R(x))/M)
{
float  fitness = 0;
fitness = (W1 * A) + (W2 * (M - R)) / M;
return fitness;
}
```

C paralelo (selgencancer_main_par.c)

```
//====================================//
// fitness
//====================================//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <omp.h>
//====================================//
#define NF 8
//Probar NF con: 8, 16, 32, 64, 100, 128
#define M 48
#define W1 0.2
#define W2 (1 - W1)
//====================================//
float getFitness(float A,float R);
//====================================//
int main(void)
{
double       t_start,t_finish, t_elapsed;
             // Marcas de medidas de tiempo
unsigned int cont,contmax;
unsigned int numExecutions;
int i, j, tid;

//-------------------//
float A[NF],R[NF];
float fitness[NF];
float fitness_min,A_best;
//-------------------//
contmax = 1E+7;

t_start = omp_get_wtime();

#pragma omp parallel for private(i, fitness, A,
R) lastprivate(fitness_min)
//-------------------//
for(cont=0; cont<contmax; cont++)
{
//contmax iteraciones del cálculo del
//conjunto de NF unidades fitness

    for(i = 0; i < NF; i++)
    //doy entradas a las NF unidades fitness
    {
        A[i] = i / NF;
        R[i] = i % NF;
    }

    for(i = 0; i < NF; i++)
    //calculo los NF fitness
    {
        fitness[i] = getFitness(A[i], R[i]);
    }

    fitness_min=fitness[0];
    //determino fitness mínimo

    for(i = 1; i < NF; i++)
    {
        if(fitness[i] < fitness_min)
        {
            fitness_min=fitness[i];
        }
    }
}
//-------------------//
t_finish = omp_get_wtime();
t_elapsed = t_finish - t_start;

printf("fitness_min = %f\n",fitness_min);
printf("Time = %g seconds\n",t_elapsed);
//-------------------//
printf("=== FIN ===\n");
return(1);
}
//====================================//
float getFitness(float A,float R)
//Fitness = F(x) = w1·A(x)+(w2·(M-R(x))/M)
{
float  fitness = 0;
fitness = (W1 * A) + (W2 * (M - R)) / M;
return fitness;
}
```

Figura 28. Versiones C secuencial y C paralelo (OpenMP).

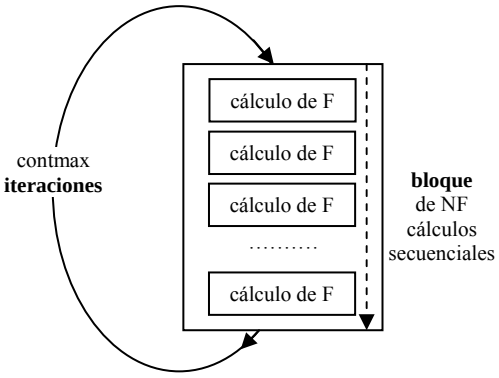

Figura 29. Iteraciones de ejecución de un bloque de NF cálculos de la función de fitness.

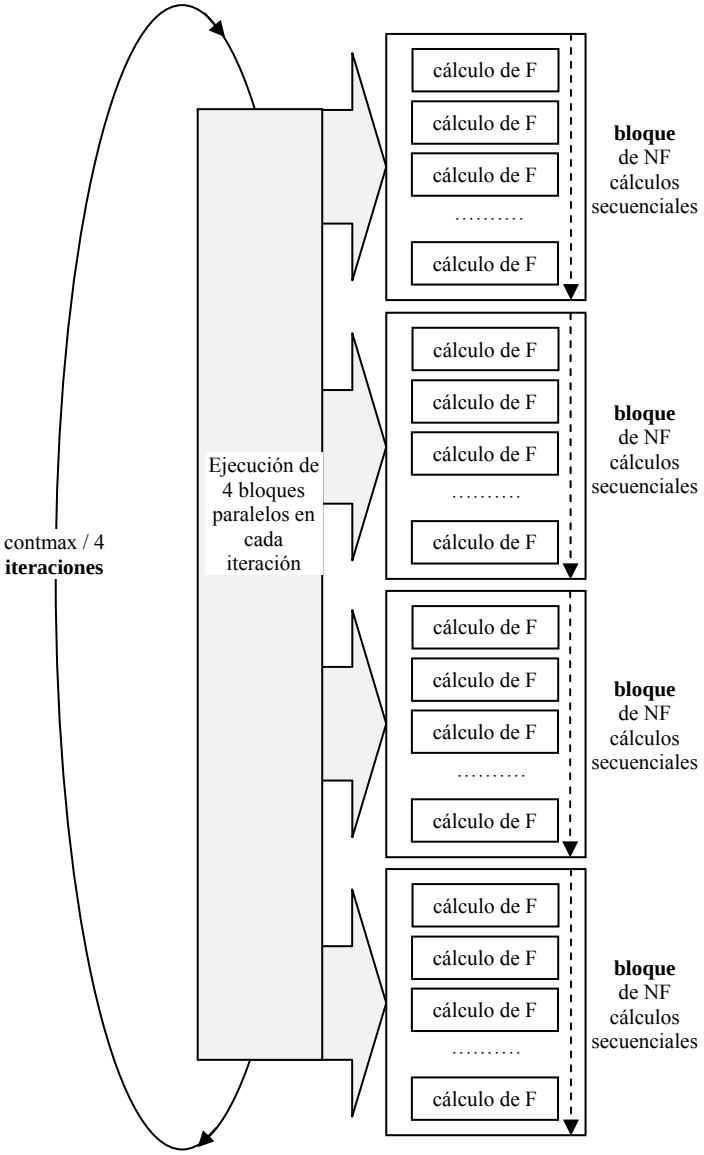

Figura 30. Iteraciones de ejecución de 4 bloques de NF cálculos de fitness mediante 4 hilos de ejecución paralela.

3.8.3 Tiempo de computación.

En la Tabla 4 se muestran los tiempos de computación del software, tanto en su versión secuencial (C) como paralela (OpenMP). Se han efectuado 10 ejecuciones de cada experimento (NF=8, 16, 32, 64, 100, 128), para encontrar los valores promedios, que se muestran como tabla en la Tabla 5 y gráficamente en la Figura 31.

	NF = 8		NF = 16		NF = 32	
	Tiempos (s)		Tiempos (s)		Tiempos (s)	
Ejecución	C	OpenMP	C	OpenMP	C	OpenMP
1	1.3910	0.8454	2.6126	1.4463	5.2853	2.7194
2	1.3687	0.8014	2.6296	1.4232	5.2689	2.9103
3	1.3976	0.7780	2.6866	1.6162	5.3602	2.7585
4	1.3538	0.7676	2.8993	1.4516	5.4003	2.6869
5	1.3704	0.8507	2.7086	1.5283	5.5562	2.6519
6	1.3339	0.7967	2.7192	1.5313	5.3647	2.6556
7	1.3220	0.8114	2.7332	1.4591	5.2538	2.6923
8	1.3591	0.9239	2.6392	1.5267	5.3089	2.7279
9	1.3508	0.7992	2.6734	1.5508	5.2303	2.7704
10	1.3374	0.8043	2.7688	1.5790	5.2192	2.7711
Average	**1.3585**	**0.8179**	**2.7070**	**1.5112**	**5.3248**	**2.7344**
	NF = 64		**NF = 100**		**NF = 128**	
	Tiempos (s)		Tiempos (s)		Tiempos (s)	
Ejecución	C	OpenMP	C	OpenMP	C	OpenMP
1	10.9328	5.1855	17.6001	8.1520	21.7555	10.0318
2	10.9244	5.4145	17.3837	8.1982	21.7957	9.8743
3	10.9183	5.1285	17.5355	8.1823	21.4827	9.8682
4	10.8937	5.3039	17.6133	8.2920	21.8357	9.7584
5	11.1980	5.0228	17.6287	8.1907	21.6928	9.6074
6	10.7866	5.1245	17.7007	8.4703	21.7558	9.7769
7	10.7243	5.0907	17.7479	8.3310	21.8934	9.8351
8	10.8268	5.0960	17.5593	8.1436	21.8002	10.0601
9	10.7163	5.0945	17.9382	8.2904	21.6157	9.9228
10	11.0396	5.0456	17.9231	8.2504	21.8459	10.0446
Average	**10.8961**	**5.1506**	**17.6631**	**8.2501**	**21.7473**	**9.8779**

Tabla 4. Ejecuciones de las versiones secuencial y paralela.

NF	C		OpenMP	
	Tiempo (s)	Consumo (W)	Tiempo (s)	Consumo (W)
8	1.3585	29.02	0.8179	32.96
16	2.7070	27.28	1.5112	31.51
32	5.3248	28.82	2.7344	31.56
64	10.8961	27.52	5.1506	32.38
100	17.6631	27.33	8.2501	31.90
128	21.7473	27.22	9.8779	31.62

Tabla 5. Tiempos promedios y energía consumida en las ejecuciones de las versiones secuencial y paralela.

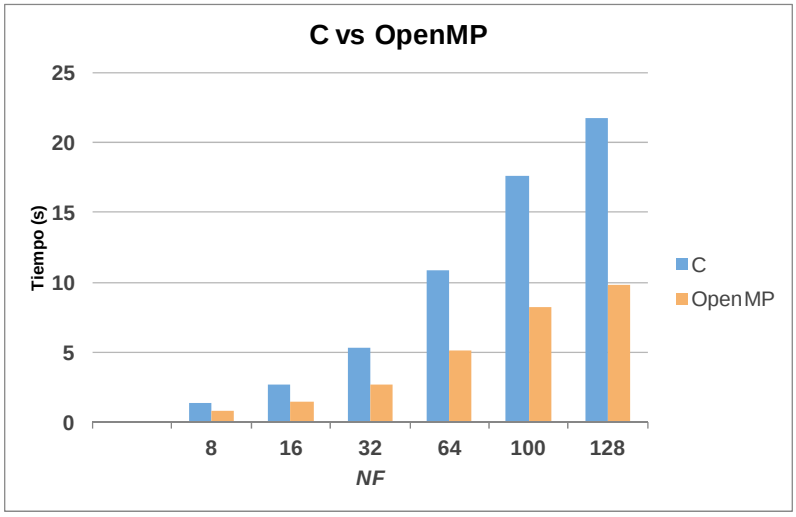

Figura 31. Tiempos promedios de las ejecuciones de las versiones secuencial y paralela.

3.8.4 Consumo energético.

Para evaluar el rendimiento del circuito acelerador respecto a una CPU de propósito general, además del tiempo de computación es interesante comparar el consumo energético, que es un parámetro de gran interés hoy dia en el ámbito de la computación.

Para medir el impacto energético en la CPU producido por la ejecución del software, se ha utilizado la herramienta Powerstat para el S.O. Ubuntu 13. Esta utilidad mide el consumo de la CPU de un portátil a partir de la información ACPI[5] de la batería.

Los resultados del consumo energético de la CPU durante la ejecución del software, tanto en la versión secuencial como paralela, se muestran tabularmente en la Taqbla 5, y de forma gráfica en la Figura 32. Se puede apreciar cómo el consumo de la versión paralela es superior, debido a que se utilizan los dos núcleos de la CPU.

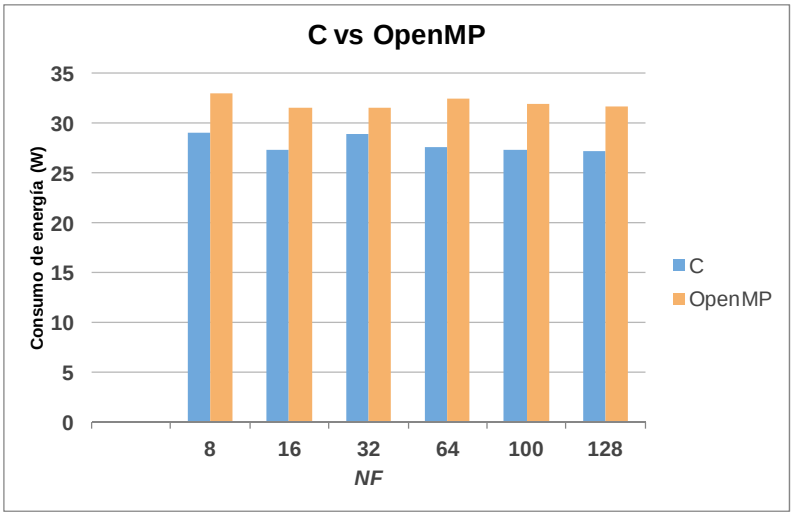

Figura 32. Consumos energéticos en las ejecuciones de las versiones secuencial y paralela.

[5] ACPI (Advanced Configuration and Power Interface) es una interfaz hardware que controla el funcionamiento de la BIOS proporcionando mecanismos para la gestión de la energía.

3.9 Rendimiento del circuito acelerador.

En esta sección comparamos los resultados obtenidos para la misma configuración experimental, según los valores mencionados en apartados anteriores:

- Configuración experimental:
  - NF=128.
  - NC = 64.
- Hardware:
  - FPGA Virtex6.
  - CPU i7.

3.9.1 Tiempo de computación.

En la Figura 33 se muestra el grado de aceleración de la FPGA frente a la CPU. A la vista del resultado, se ve que la aceleración es buena respecto de una versión secuencial del software. Cuando la implementación software aprovecha los hilos de ejecución que proporcionan los núcleos de la CPU, el rendimiento de la FPGA baja, hasta igualar al tiempo de computación de la CPU. En este caso no se puede decir que hay una mejoría, aunque sí podemos decir que la FPGA mantiene un atractivo de computación, teniendo en cuenta que se enfrenta a una CPU de muy altas prestaciones programada en paralelo.

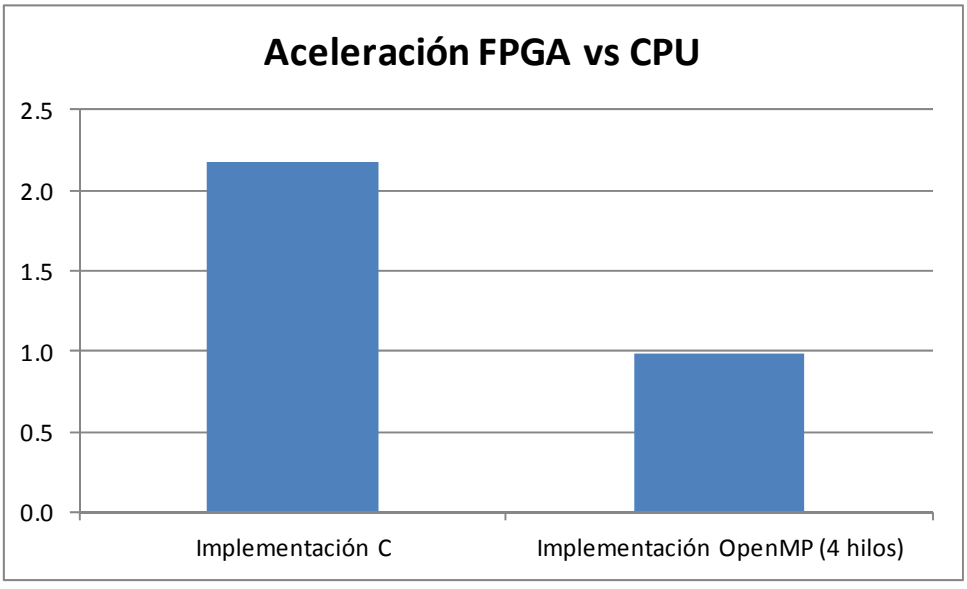

Figura 33. Aceleración FPGA vs. CPU.

3.9.2 Consumo energético.

En la Figura 34 se muestran juntos los resultados del consumo energético de las implementaciones hardware y software. En este caso sí podemos decir que el rendimiento es más que aceptable cuantitativamente. La inclusión del consumo energético como medida de rendimiento de un sistema computacional radica en el interés que suscita hoy dia la demanda de bajo potencia de los sistemas informáticos,

especialemente en aquellos casos en que, por razones del problema abordado, los algoritmos están procesando grandes cantidades de datos durante mucho tiempo, lo cual tiene un impacto económico considerable, teniendo en cuenta no solo la demanda de energía de los circuitos, sino también el gasto en los sistemas de refrigeración que son necesarios para mantener en buen estado los equipos informáticos.

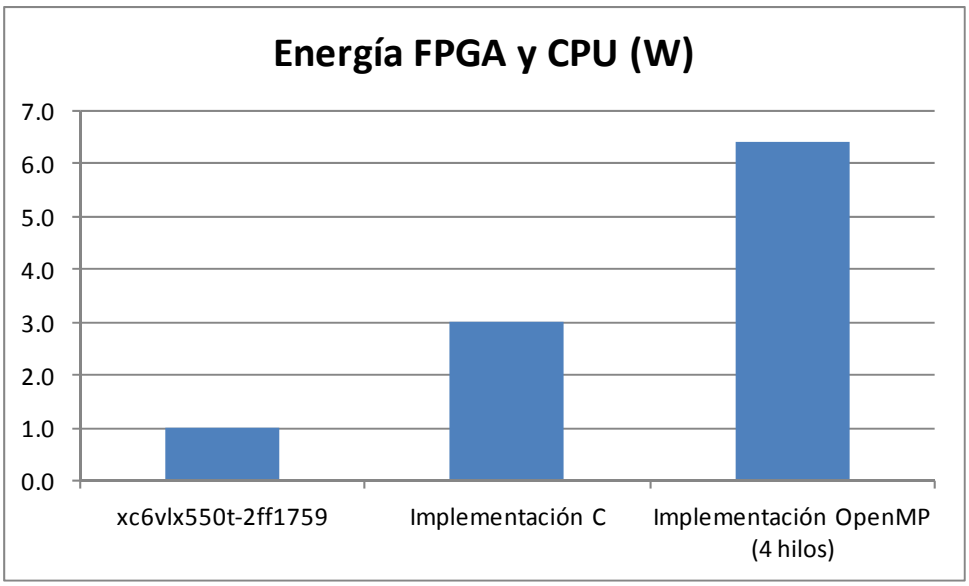

Figura 34. Consumo energético FPGA Virtex 6 (mejor configuración experimental) y CPU Intel i7 (para las implementaciones software secuencial y paralela).

3.10 Aumento del rendimiento.

El principal objetivo fijado para este trabajo fue evaluar la adecuación de una implementación FPGA de la función de fitness para acelerar los cálculos involucrados en la clasificación y selección de genes del cáncer, dado que se ha demostrado que la aplicación de la función de fitness a todos los indivicuos de una población es, habitualmente, la parte más costosa en tiempo de computación en el tratamiento algorítmico basado en computación evolutiva.

Para este objetivo, y a la vista de los resultados experimentales anteriormente expuestos, podemos concluir que el rendimiento del circuito acelerador es aceptable (sobre todo desde el punto de vista energético), pero no suficiente para suscitar interés en desarrollar soluciones HW/SW que traten el problema de forma práctica.

En este sentido, creemos que es fundamental mejorar el rendimiento del circuito acelerador implementado en la FPGA. Para ello, existen diversas posibilidades por explorar, que se dejan como líneas de trabajo propuestas para el futuro:

- Implementar la función de fitness y los controladores en lenguaje VHDL. Algunos estudios [Stu2008] han demostrado que el diseño de circuitos con VHDL ofrece mayor velocidad de reloj y menor área ocupada que el mismo diseño utilizando Handel-C. También se puede plantear el uso de lenguajes basados en C más recientes (System-C, Impulse-C, Vivado, etc).
- Rediseñar la función de fitness y el controlador para introducir un mayor grado de paralelismo.
- Opimizar el proceso de sínteis e implementación mediante ISE o Vivado para obtener frecuencias de reloj más elevadas.
- Utilizar librerías más modernas de CoreGenerator para generar cores aritméticos en coma flotante más optimizados.
- Aplicar dispositivos FPGAs más modernos que proporcionan rendimientos muy superiores (Virtex-7, Kyntex-7, Virtex-UltraScale, etc).

4 Programación de un algoritmo genético.

Dado que la complejidad de desarrollar el algoritmo al completo en la FPGA era demasiado elevada, decidimos que la parte menos crítica, es decir, aquella en la que el algoritmo invierte menos tiempo, podría estar desarrollada en C y que la parte más importante y que más recursos consume se mantendría en la FPGA con el objetivo de acelerar lo máximo posible dicha sección del algoritmo. No obstante, y con el objetivo de poder conocer con exactitud la ganancia en rendimiento que supone el uso de una FPGA como herramienta aceleradora, finalmente decidimos que el primer paso en nuestro camino sería implementar el algoritmo completo en C y posteriormente plantear su integración en una FPGA. De esta forma podríamos comprobar con precisión milimétrica la diferencia de tiempos entre la versión desarrollada exclusivamente en C y la versión híbrida implementada en su mayor parte en C pero siendo la FPGA la que resuelve la parte más crítica del algoritmo, el cálculo del fitness.

Como ya comentamos al inicio del documento, este proyecto tiene como base un algoritmo genético propuesto en varios artículos científicos, donde cada uno de ellos añade pequeñas modificaciones al mismo. En el siguiente apartado se explica detalladamente y punto por punto la configuración que hemos utilizado para implementar el mencionado algoritmo genético, así como su estructura.

4.1 Configuración del algoritmo genético.

Los Algoritmos Genéticos (AGs) son métodos adaptativos que pueden usarse para resolver problemas de búsqueda y optimización. Están basados en el proceso genético de los organismos vivos. A lo largo de las generaciones, las poblaciones evolucionan en la naturaleza de acorde con los principios de la selección natural y la supervivencia de los más fuertes, postulados por Darwin (1859). Por imitación de este proceso, los Algoritmos Genéticos son capaces de ir creando soluciones para problemas del mundo real. La evolución de dichas soluciones hacia valores óptimos del problema depende en buena medida de una adecuada codificación de las mismas.

Todo AG necesita una codificación o representación del problema, que resulte adecuada al mismo. Además se requiere una función de ajuste o adaptación al problema (***función fitness***), la cual asigna un número real a cada posible solución codificada (***cromosoma***) de una población dada (***población***). Durante la ejecución del algoritmo, los padres deben ser seleccionados para la reproducción (***selección***), a continuación dichos padres seleccionados se cruzarán generando dos hijos (***cruce***), sobre cada uno de los cuales actuará un operador de mutación (***mutación***). El resultado de la combinación de las anteriores funciones será un conjunto de individuos (posibles soluciones al problema), que en la evolución del AG formarán la siguiente población (***reemplazo***).

A continuación se explica en detalle la configuración particular seleccionada en cada una de las fases que componen nuestro AG: selección, cruce, mutación y reemplazo; así como su estructura, desde la definición de sus cromosomas, hasta la generación de la población inicial, pasando por la descripción de su función fitness.

4.1.1 Estructura de un cromosoma.

Cada individuo o cromosoma representa un subconjunto de genes de longitud variable, tal y como se presenta en la Figura 35 de más abajo.

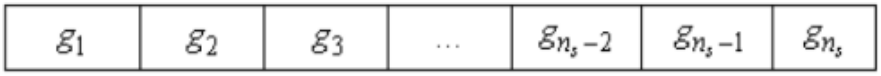

Figura 35. Estructura de un cromosoma.

Donde:

- **r**: es la longitud de este cromosoma en particular ($1 \leq r \leq n_s$).
- **i**: representa el i-ésimo gen dentro de un cromosoma ($1 \leq i \leq r$).
- $\mathbf{g_i}$: indica qué genes de entre el conjunto total de ellos son necesarios para formar este cromosoma. Por ejemplo: $g_i = 5$ indica que el gen 5 del conjunto total de genes forma parte de este cromosoma ($1 \leq g_i \leq M$).

4.1.2 Generación de la población inicial.

Tanto el tamaño como la forma de generar la población inicial de un algoritmo genético son claves para la obtención de unos resultados óptimos en un tiempo razonable. Se ha demostrado que un tamaño de población demasiado pequeño [Diaz2007][RYLANDER2002] da como resultado soluciones poco óptimas, mientras que poblaciones muy grandes no suponen mejora alguna sobre otras de tamaño más comedido y que además resultan más eficientes, por lo que resulta de vital importancia encontrar ese término medio. Por otro lado, la forma de generar la población inicial en un algoritmo genético influye enormemente en los resultados obtenidos; se han realizado numerosos estudios acerca de este hecho [DiazGomez2007] en los cuales se afirma que la población inicial idónea es aquella que presenta una mayor diversidad entre sus individuos, es decir, una población lo más heterogénea posible, porque de lo contrario el algoritmo tiende a converger a lo que es conocido como soluciones óptimas locales [Rocha1999], que son aquellas generadas sin la intervención de gran parte de la población en todo el proceso.

Con objeto de encontrar los parámetros óptimos del algoritmo, en este proyecto se probarán distintos tamaños de población y la generación de la población inicial se realizará de forma aleatoria. De manera que para una población de tamaño P se generarán P cromosomas de longitud variable donde los genes de cada uno de ellos serán seleccionados de forma aleatoria de entre el conjunto total de genes, siguiendo cada cromosoma la estructura descrita en el apartado anterior.

4.1.3 Evaluación de los individuos de la población.

Uno de los puntos más importantes, si no el que más, de un algoritmo genético es el cálculo del fitness de los individuos de la población, es decir, evaluar cuán apto es cada uno de ellos de cara al problema que se pretende resolver. Utilizar la función fitness adecuada para cada contexto es clave de cara a la obtención de unos resultados óptimos, pues dicha función va a determinar con qué individuos nos quedamos y cuáles descartamos.

La función fitness que aquí se implementará será la misma que la utilizada en los estudios sobre los que este proyecto tiene su base. Dicha función es la siguiente:

$$fitness(X) = w_1 A(x) + w_2\big(M - R(x)\big)/M$$

Donde:

- $A(x) \in [0, 1]$ es el la precisión resultante de aplicar el algoritmo *leave-one-out-cross-validation* (**LOOCV**) sobre los datos de entrenamiento utilizando únicamente los valores de expresión de los genes seleccionados en el subconjunto *x*.
- *R(x)* es el número de genes seleccionados en *x*.
- *M* es el número total de genes.
- w_1 y w_2 denotan dos pesos correspondientes a la importancia de la precisión y del número de genes seleccionados, respectivamente. Donde $w_1 \in [0.1, 0.9]$ y $w_2 = 1 - w_1$.

Esta fórmula está diseñada de manera que promocione aquellos individuos con mayor precisión LOOCV y menor número de genes seleccionados.

4.1.4 Selección.

A la hora de implementar un algoritmo genético, una de las dudas que nos puede surgir es qué método de selección utilizar. Existen multitud de ellos: *Roulette Wheel Selection* (RWS), *Stochastic Universal Sampling* (SUS), *Linear Rank Selection* (LRS), *Exponential Rank Selection* (ERS) y *Tournament Selection* (TOS). En realidad, todos estos métodos, a excepción del último (TOS), son variaciones del primero (RWS), en el que los individuos se seleccionan con una probabilidad proporcional a su fitness.

En este proyecto implementaremos el primer método de selección nombrado, Roulette Wheel Selection, pues éste ha sido el utilizado por los diferentes estudios en los que se basa este trabajo. Este hecho nos permitirá poder realizar diferentes comparativas de tiempo en un futuro.

El funcionamiento de este método de selección se ilustra en la Figura 36 de más abajo. Se crea una ruleta con los cromosomas presentes en una generación. Cada cromosoma tendrá una parte de esa ruleta mayor o menor en función al fitness que tenga cada uno. Se hace girar la ruleta y se selecciona el cromosoma que se encuentre bajo el selection point una vez ha dejado de girar. Obviamente el cromosoma con mayor puntuación saldrá con mayor probabilidad. En caso de que las probabilidades difieran mucho, este método de selección dará problemas puesto que si un cromosoma tiene un 90% de posibilidades de ser seleccionado, el resto apenas saldrá lo que reduciría la diversidad genética.

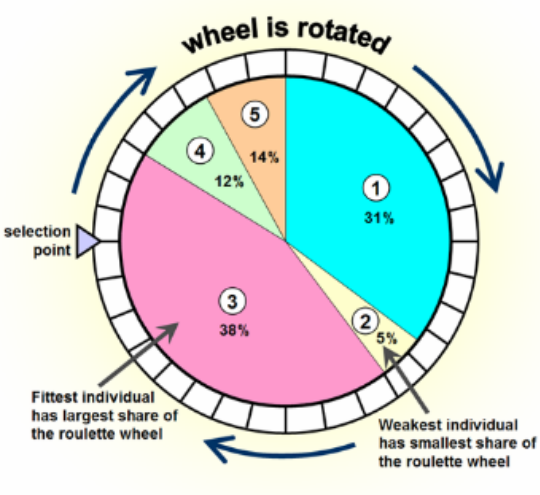

Figura 36. Funcionamiento del metodo de selección Roulette Wheel.

4.1.5 Cruce.

Al igual que ocurre en la fase de selección de cromosomas, en la de cruce también nos encontramos con múltiples métodos, con la diferencia de que en este caso las diferencias entre ellos son mayores, influyendo en mayor medida en la evolución del algoritmo. Los métodos de cruce que nos podemos encontrar son los siguientes: *One-Point Crossover*, *Two-Point Crossover*, *Cut and Splice*, *Uniform Crossover* y *Half Uniform Crossover* entre otros.

Para la ejecución de este proyecto implementaremos el *Two-Point Crossover*, pues al igual que en el caso anterior, éste es el método elegido por los estudios sobre los que se basa este trabajo. En la Figura 37 de más abajo podemos ver una ilustración que muestra el funcionamiento de este método de cruce:

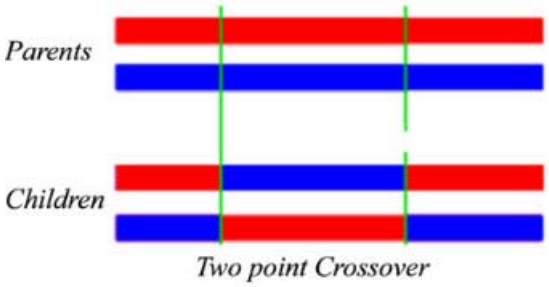

Figura 37. Cruce utilizando el algoritmo Two point Crossover.

Se trata de una generalización del cruce de 1 punto. En vez de cortar por un único punto los cromosomas de los padres como en el caso anterior, se realizan dos cortes. Deberá tenerse en cuenta que ninguno de estos puntos de corte coincida con el extremo de los cromosomas para garantizar que se originen tres segmentos. Para generar la descendencia se escoge el segmento central de uno de los padres y los segmentos laterales del otro padre.

Generalizando, se pueden añadir más puntos de cruce dando lugar a algoritmos de cruce multipunto. Sin embargo, existen estudios que desaconsejan esta técnica [DeJong1992]. Aunque se admite que el cruce de 2 puntos aporta una sustancial mejora con respecto al cruce de un solo punto, el hecho de añadir un mayor número de puntos de cruce reduce el rendimiento del AG. El problema principal de añadir nuevos puntos de cruce radica en que es más fácil que los segmentos originados sean corruptibles, es

decir, que por separado quizás pierdan las características de bondad que poseían conjuntamente. Sin embargo no todo son desventajas y añadiendo más puntos de cruce se consigue que el espacio de búsqueda del problema sea explorado más a fondo.

4.1.6 Mutación.

La mutación es otro de los puntos claves en el éxito o fracaso de un algoritmo genético, pues evita en cierta medida que algunos caminos del espacio de búsqueda queden inexplorados, evitando además la convergencia prematura a soluciones subóptimas. No obstante, hay que encontrar el valor de probabilidad óptimo con que se producen estas mutaciones, pues un valor demasiado bajo podría eternizar la llegada a una solución óptima, estancamiento en resultados subóptimos; mientras que un valor alto podría llegar a corromper buenas soluciones con demasiada frecuencia, alargando de igual modo la consecución de soluciones óptimas.

En este proyecto y al igual que los artículos científicos en los que se basa, utilizaremos una probabilidad de mutación del 15%, es decir, 15 de cada 100 cromosomas generados por medio de cruces sufrirán una mutación.

Ahora bien, existen múltiples tipos de mutación: *Bit String Mutation*, *Flip Bit, Boundary, Non Uniform, Uniform* y *Gaussian*, entre otros. Para este proyecto se implementará una mutación del tipo *Uniform*, pues es la que pensamos que mejor resultados aporta de cara a evitar una convergencia prematura en soluciones subóptimas. En este tipo de mutación se sustituye el valor de un gen dentro del cromosoma elegido de forma aleatoria por un valor también aleatorio seleccionado entre los límites superior e inferior del conjunto total de genes.

4.1.7 Reemplazo.

La última fase de todo algoritmo genético es la de reemplazo de algunos miembros de la población actual por aquellos generados durante las fases de cruce y mutación. El método seleccionado es muy importante, ya que las diferencias entre ellos hacen que los resultados obtenidos y el tiempo empleado en ello sean muy distintos. Existen multitud de métodos de reemplazo, entre los que se encuentran: *Replace worst, Replace Random, Replace parent, Replace most similar* y *Replace Best*. De su nombre podemos deducir su funcionamiento y cómo influyen en el devenir del algoritmo.

En este proyecto y con objeto de seguir la línea de los estudios anteriormente citados, utilizaremos una versión del *Replace parent* un tanto modificada. En este tipo de reemplazo los hijos generados durante las fases de cruce y mutación siempre sustituyen a sus padres, no importando el valor de sus fitness. Con objeto de paliar este hecho y que se descarten aquellos individuos más aptos, introduciremos en este proceso lo que se conoce como *Elitism*, mediante el cual un porcentaje de la población más apta es salvaguardada al inicio de cada generación y recuperada al final de la misma. Esta estrategia garantiza que la calidad de las soluciones obtenidas por el algoritmo genético no disminuirá de una generación a la siguiente. Para este trabajo se ha utilizado un elitismo del 15% en consonancia a los artículos previamente mencionados.

4.2 Implementación del algoritmo genético.

Ahora que ya conocemos las peculiaridades de nuestro algoritmo genético, ha llegado el momento de proceder a la explicación de su implementación. Para ello, se ha elaborado un diagrama de clases UML en el que se muestran las distintas clases presentes en nuestra aplicación, así como las relaciones que mantienen entre sí.

4.2.1 Diagrama de clases UML.

La Figura 38 muestra el diagrama de clases UML usado para desarrollar el AG.

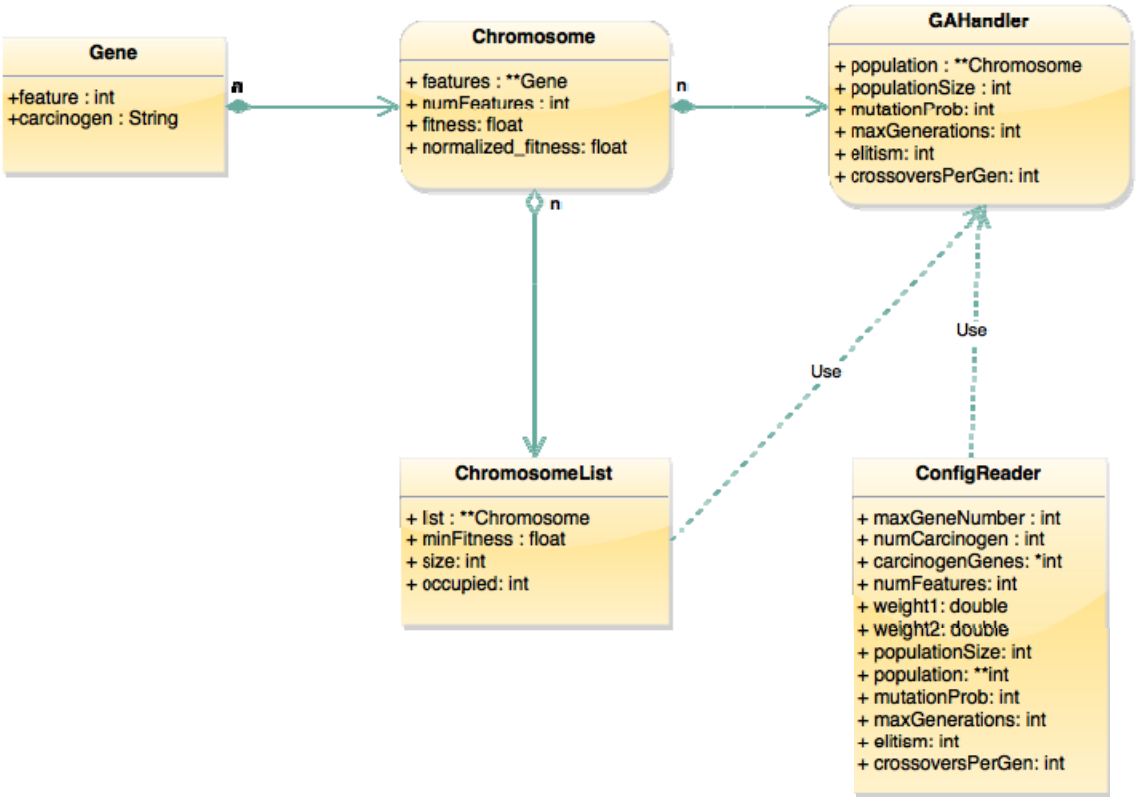

Figura 38. Diagrama de clases del algoritmo genético desarrollado.

Como podemos observar, tenemos un total de 5 clases: *Gene*, *Chromosome*, *ChromosomeList*, *ConfigReader* y *GAHandler*. La más importante de todas ellas, o la que por decirlo de algún modo "lo mueve todo" es *GAHandler* que hace uso de objetos del tipo *ChromosomeList* y *ConfigReader*, y además se compone de una matriz de objetos *Chromosome* que forman una población. Por otro lado, la clase *ChromosomeList* se compone de un array de *Chromosome*, los cuales a su vez están formados por un conjunto de *Gene*, que constituyen la clase más básica del sistema.

En el siguiente apartado describiremos más en detalle la función de cada una de estas clases, así como la de sus atributos.

4.2.2 Descripción de clases.

Antes de comenzar a describir cada una de las clases implementadas, hemos de decir que, dado que todo ha sido desarrollado en C para un mayor rendimiento y

compatibilidad, hemos perdido la capacidad de utilizar clases como tal, ya que este lenguaje no las soporta. No obstante, hemos intentado (y creo que hemos conseguido) que nuestro desarrollo se asemeje lo máximo posible a uno propio de la POO (*Programación Orientada a Objetos*), con ficheros cabecera en los que se definen las estructuras ("clases") y sus funciones, y ficheros fuente donde se encuentra la implementación de dichas funciones.

GENE.

Esta clase es la representación software de los genes y sus funciones dentro del algoritmo.

Atributos:

- ***feature***: contiene el valor correspondiente al gen que representa (ej. 5).
- ***carcinogen***: informa sobre si este gen es carcinógeno o no.

CHROMOSOME.

Esta clase es la representación software de los chromosomas y sus funciones dentro del algoritmo.

Atributos:

- ***features***: array de genes que forman este chromosoma.
- ***numFeatures***: informa sobre el número de genes que forman este cromosoma.
- ***fitness***: contiene el valor fitness asignado por la función de evaluación a este cromosoma.
- ***normalized_fitness***: contiene el valor fitness normalizado a 1 en relación al resto de individuos de la población.

CHROMOSOMELIST.

Esta clase representa una lista de cromosomas con una serie de funciones asociadas a los mismos.

Atributos:

- ***list***: representa el array de cromosomas presentes en esta lista.
- ***minFitness***: contiene el valor de fitness más bajo presente entre los cromosomas de la lista.
- ***size***: informa sobre el número máximo de cromosomas que caben esta lista.
- ***occupied***: informa sobre el número de cromosomas que un momento dado hay en la lista.

CONFIGREADER.

Esta clase es la encargada de traducir la información presente en el archivo de configuración a un formato "entendible" por el programa.

Atributos:

- ***maxGeneNumber***: contiene el valor máximo que puede contener un gen.
- ***numCarcinogen***: contiene el número de genes cancerosos presentes.

- ***carcinogenGenes***: array que contiene los genes cancerosos presentes en una ejecución dada.
- ***numFeatures***: informa sobre el número de genes que va a tener cada cromosoma.
- ***weight1***: contiene el valor correspondiente a la variable $_1$ de la función fitness.
- ***weight2***: contiene el valor correspondiente a la variable $_2$ de la función fitness.
- ***populationSize***: número de cromosomas que va a contener la población.
- ***population***: matriz de cromosomas que formarán la población inicial.
- ***mutationProb***: probabilidad de que se produzca una mutación sobre cada uno de los cromosomas hijos resultados de un cruce.
- ***maxGenerations***: máximo número de generación que el algoritmo genético puede completar sin llegar a una solución óptima.
- ***elitism***: almacena qué porcentaje de la población se va a mantener como élite en cada generación.
- ***crossoversPerGen***: contiene el número de cruces que se debe producir en cada generación del algoritmo genético.

GAHANDLER.

Esta clase es la encargada de gestionar todas y cada una de las fases del algoritmo genético, desde la creación de la población inicial, hasta la generación de una solución óptima; pasando por todas y cada una de las fases intermedias que componen un algoritmo genético.

Atributos:

- ***population***: almacena los cromosomas correspondientes a la generación actual. En la primera iteración contendrá los cromosomas especificados en el archivo de configuración. A partir de ahí, la población irá variando en cada generación con los cromosomas generados frutos de los cruces.
- ***populationSize***: informa sobre el tamaño de la población de la generación actual. Este parámetro permanecerá invariable durante la ejecución del algoritmo, pues todas las poblaciones mantendrán el tamaño de la población original.
- ***mutationProb***: informa sobre la probabilidad que tiene un cromosoma generado como consecuencia de un cruce de sufrir una mutación.
- ***maxGenerations***: indica el número máximo de generaciones que se pueden llegar a completar sin encontrar una solución óptima. Si esta solución es hallada antes de alcanzar tal número, el algoritmo se da por finalizado igualmente.
- ***elitism***: indica qué porcentaje de la población va a pasar de la generación actual a la siguiente sin verse afectada durante la fase de reemplazo.
- ***crossoversPerGen***: informa sobre el número de cruces que se van a realizar en cada iteración del algoritmo.

4.2.3 FIchero de configuración.

Con objeto de facilitar la ejecución del algoritmo a todas aquellas personas ajenas al mundo de la informática y más concretamente al de la programación, decidimos que lo mejor sería que los diferentes parámetros del algoritmo se pudiesen modificar a través de un archivo de configuración que fuese fácil de entender y modificar. Este archivo tiene la extensión `.cfg` y debe denominarse `gaselcancer` (`gaselcancer.cfg`).

A continuación se muestra la estructura interna de este archivo de configuración, así como los valores por defecto de los distintos parámetros del algoritmo:

```
// GENE VARIABLES
GENE_VARS =
{
  MAX_GENE_NUMBER = 100; //Maximum number of genes available. It's the M defined in
the function above.
  CARCINOGEN_GENES = [3, 4, 7, 9, 12, 15, 19, 22, 28, 36, 41, 49, 53, 57, 64, 65,
70, 73, 81, 97];
}

// CHROMOSOME VARIABLES
// Fitness function f(x) = A(x)*w1 + w2*(M - R(x))/M
CHROMOSOME_VARS =
{
  NUM_FEATURES = 10; //Number of features per chromosome
  WEIGHT_1 = 0.65; //Weights defined in the function above
  WEIGHT_2 = 0.35;
}

// GENETIC ALGORITHM VARIABLES
GA_VARS =
{
  MUTATION_PROBABILITY = 20; //Mutation probability in percentage
  MAX_GENERATIONS = 100;
  ELITISM = 15; //Percentage of the population that will be part of the elite
  CROSSOVERS_PER_GEN = 10;
  POPULATION =
  (
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
    [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
    [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
    [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
    [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
    [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
    [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
    [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
  )
}
```

Figura 39. Estructura del archivo de configuración.

El formato del fichero de configuración es muy similar a un JSON, por lo que resulta fácilmente comprensible y modificable. Como podemos observar, en este archivo también se pueden insertar comentarios, basta con utilizar la notación "//" y todo lo que venga a continuación en esa línea será considerado como un comentario.

De la Figura 39 anterior podemos distinguir tres secciones bien diferenciadas, cuyos nombres dan una idea acerca de los elementos a los que afectan dentro del algoritmo: ***GENE_VARS***, ***CHROMOSOME_VARS*** y ***GA_VARS***.

GENE_VARS.

Contiene los parámetros relacionados con los genes en el algoritmo genético.

Parámetros:

- ***MAX_GENE_NUMBER***: informa sobre el número máximo de genes presentes en la muestra. Este número es necesario para calcular el fitness de los cromosomas. Se corresponde a la variable **M** presente en dicha función.
- ***CARCINOGEN_GENES***: array que indica cuáles de estos genes son canceros.

CHROMOSOME_VARS.

Contiene las variables relativas a los cromosomas en el algoritmo genético.

Parámetros:

- ***NUM_FEATURES***: indica el número de genes que va a contener cada cromosoma.
- ***WEIGHT_1***: parámetro utilizado en el cálculo del fitness de un cromosoma y que se corresponde con la variable w_1 presente en dicha función.
- ***WEIGHT_2***: al igual que WEIGHT_1, se trata de una variable utilizada en el cálculo del fitness de un cromosoma.

GA_VARS.

Contiene los parámetros propios de todo algoritmo genético.

Parámetros:

- ***MUTATION_PROBABILITY***: indica la probabilidad de que un cromosoma fruto de un cruce sufra una mutación.
- ***MAX_GENERATIONS***: indica el número máximo de generaciones que se pueden llegar a completar sin alcanzar una solución óptima.
- ***ELITISM***: informa sobre qué porcentaje de la población se va a mantener como élite y que por tanto pasarán de una iteración dada a la siguiente sin ser reemplazados.
- ***CROSSOVERS_PER_GEN***: indica el número de cruces que se van a realizar en cada iteración/generación del algoritmo.
- ***POPULATION***: matriz de genes (números enteros) que representa los distintos cromosomas que van a formar parte de la generación inicial del algoritmo genético.

4.2.4 Descripción de la función principal.

Considero que explicar todas y cada una de las funciones que componen nuestro algoritmo genético no tiene mucho sentido, pues ésto incrementaría notablemente la extensión del documento y no aportaría demasiado. No obstante, sí que conviene explicar el flujo de control que sigue la función más importante del algoritmo, que no es otra que la que se encarga gestionar el propio algoritmo genético, desde su inicio, hasta su consecución.

A esta función la hemos denominado `applyGA` y su estructura es la siguiente:

```
void applyGA(GAHandler *gaHandler)
{
  int i, j, k, *parents,
      numParents  = 2,
      numChildren = 2;
  Chromosome     **children;
  ChromosomeList  *elite;

  printPopulation(gaHandler); [1]
  for(k = 0; k < gaHandler->maxGenerations; ++k) { [2]
    printf("\n=================================================\n");
    printf("Generation: %d\n", k + 1);
    elite = elitism(gaHandler); [2.1]
    for(i = 0; i < gaHandler->crossoversPerGen; ++i) { [2.2]
      parents  = selection(gaHandler, numParents); [2.2.1]
      children = crossoverPhase(gaHandler,gaHandler->population[parents[0]],
        gaHandler->population[parents[1]]); [2.2.2]
      for(j = 0; j < numChildren; ++j) { [2.2.3]
        mutation(gaHandler, children[j]); [2.2.3.1]
        calculateFitness(children[j]); [2.2.3.2]
        freeChromosome(gaHandler->population[parents[j]]);
        gaHandler->population[parents[j]] = children[j]; [2.2.3.3]
      }
      free(children);
      free(parents);
    }
    restoreElite(gaHandler, elite); [2.3]
    printPopulation(gaHandler); [2.4]

    freeChromosomeList(elite);
  }
}
```

Figura 40. Código del método principal del algoritmo genético (applyGA).

El flujo de ejecución de la función anterior es la siguiente:

1. **Imprimimos** por pantalla la población inicial. Este paso no es necesario, pero ayuda a entender el devenir de los resultados.
2. Para **cada generación** realizamos lo siguiente:
   - 2.1. Guardamos aquellos **cromosomas** que van a conformar la **élite** de la generación actual.
   - 2.2. Para **cada** uno de los cruces a realizar en la **iteración** actual llevamos a cabo los siguientes pasos:
     - 2.2.1. **Seleccionamos** los cromosomas que van a formar parte del cruce. Se escoge un total de dos cromosomas.
     - 2.2.2. Se lleva a cabo el **cruce** de los cromosomas seleccionados, lo que da lugar a otros dos cromosomas (hijos).
     - 2.2.3. Para **cada** uno de estos **cromosomas hijo** realizamos lo siguiente:
       - 2.2.3.1. Llevamos a cabo una **mutación** en función de la probabilidad indicada en el archivo de configuración.
       - 2.2.3.2. Calculamos su **fitness**.

2.2.3.3. **Sustituimos** uno de los cromosomas padre por éste en la población actual. Si recordamos bien, según el algoritmo, los cromosomas hijos siempre reemplazan a sus padres.

2.3. **Recuperamos** aquellos cromosomas que formaban parte de la **élite** de la generación actual y si durante la fase de reemplazo fueron sustituidos como resultado de un cruce, los recuperamos en detrimento de aquellos cromosomas de la población menos aptos (fitness más bajo).

2.4. Imprimimos por pantalla la población resultante. Como ya comentamos al inicio, este paso no es necesario, pero ayuda a seguir la evolución del algoritmo.

Esta es a grandes rasgos la implementación del algoritmo genético que hemos llevado a cabo. Queda a disposición del lector la implementación completa de la aplicación en el [Ambroise2002] de este documento.

Como comentamos al inicio de este capítulo, la idea era implementar la parte menos crítica del algoritmo en C y dejar el cálculo del fitness para la FPGA. Pues bien, como podemos apreciar en el punto *2.2.3.2* del algoritmo, el cálculo del fitness se ha llevado a cabo, junto con el resto de la aplicación, en C. Debido a limitaciones de tiempo, finalmente no hemos llegado a implementar este modelo híbrido que combina un procesador de propósito general con una FPGA, pero de hacerlo en una revisión futura de este trabajo, ese sería el punto en el que se produciría la comunicación entre ambos sistemas.

4.2.5 Código fuente.

El código fuente correspondiente a la implementación del algoritmo en C se puede encontrar en el [Ambroise2002].

5 Conclusiones y trabajos futuros.

5.1 Conclusiones.

Este proyecto nació al conocer la necesidad de reducción de los tiempos de diagnóstico del cáncer, así como la mejora de su clasificación. Es un hecho objetivo que un diagnóstico precoz y un tratamiento adecuado son claves en la lucha contra esta enfermedad, y por tanto, cualquier esfuerzo investigador en la mejora en cualquiera de estos aspectos es bienvenido.

Los diferentes artículos científicos que constituyen la base de este proyecto ya demostraron en el momento de su presentación que una mejora en ambos campos (reducción del tiempo y mejora del diagnóstico) era posible. Estos trabajos probaron que el uso de algoritmos genéticos en las técnicas de diagnóstico suponía mejoras significativas sobre los métodos existentes en el momento.

Bajo estas condiciones y gracias a la popularización de las FPGAs como dispositivos donde implementar circuitos de procesamiento paralelo, vimos una oportunidad de intentar combinar ambos aspectos y conseguir con ello una reducción aún mayor de los tiempos de diagnóstico. Los algoritmos genéticos son a menudo procesos altamente paralelizables, pues en numerosas ocasiones una misma operación debe realizarse sobre todos (o parte) de los individuos de la población, lo cual encaja a la perfección con la naturaleza altamente paralelizable de las FPGAs.

A la hora de embarcarnos en un nuevo proyecto es importante tener las metas claras y definir los distintos pasos a seguir para una buena ejecución del mismo. El primer objetivo que nos marcamos para este proyecto fue implementar la función fitness del algoritmo genético en una FPGA, pues como se ha demostrado en varios artículos previamente citados, esta parte de algoritmo suele ser la más costosa en tiempo de computación. Durante la implementación de esta función nos encontramos con un problema, y es que el cálculo aplicado para obtener el fitness de un cromosoma incluye el uso de un clasificador SVM, el cual y como podemos leer en la tesis "*FPGA implementation of a support vector machine based classification system and its potential application in smart grid*" [Song2014] no resulta nada trivial y puede constituir un proyecto por sí solo. Llegados a este punto teníamos dos opciones: dar por finalizado ahí nuestro proyecto o continuar a pesar de este contratiempo y comprobar si el uso de una FPGA como herramienta aceleradora en nuestro algoritmo tenía sentido. Decidimos optar por la segunda opción y simular los datos provenientes del SVM, pues si conseguíamos mejorar la eficiencia del algoritmo a pesar de este hecho, estaría justificado y además habría razones de peso para en un futuro realizar la implementación completa en la FPGA.

A la vista de los resultados aportados en el captulo 3.9, podemos concluir lo siguiente:

- La FPGA duplica el rendimiento de un procesador de propósito general cuando éste utiliza un único hilo para ejecutar el software.

- Cuando la CPU hace uso de varios hilos para ejecutar la aplicación, el rendimiento de la FPGA resulta ligeramente inferior en comparación.
- En todos los casos, el consumo energético de la FPGA es notablemente inferior que el de sus homólogos ejecutados en la CPU; concretamente de ⅓ de la versión de un único hilo y de ⅙ de la versión multi-hilo.

¿Tiene por tanto sentido utilizar una FPGA como herramienta aceleradora en nuestro algoritmo? La respuesta es "depende". Si un bajo consumo energético resulta de vital importancia, entonces la respuesta es "Sí". Si por el contrario, el tema energético no es nuestra prioridad, yo diría que "No", siempre y cuando estemos trabajando con CPUs capacitadas para utilizar más de un core, pues resulta mucho más fácil y rápido implementar una versión del algoritmo para un procesador de propósito general utilizando C y OpenMP, que para una FPGA utilizando VHDL, ya que se trata de lenguajes diseñados con un mayor nivel de abstracción.

Es importante mencionar que hemos utilizado una FPGA de la serie Virtex-6 con tecnología CMOS de 45 nm, cuando las modernas arquitecturas FPGA Kintex y Virtex UltraScale utilizan tecnología de 16nm. Con estas modernas FPGA sería esperable obtener tiempos de computación en FPGA mucho mejores que los obtenidos en este proyecto, para la implementación de la función de fitness ya diseñada. También, hay que tener en cuenta que el rendimiento podría mejorarse sustancialmente utilizando VHDL en lugar de Handel-C.

Por último, decir que también se ha realizado una implementación completa del algoritmo en C, cuyo propósito era el de ser integrado con la FPGA, de manera que al calcular el fitness de los individuos de una población, éste se computara en la propia FPGA y no en el procesador de propósito general. El esfuerzo y tiempo que necesitaría una implementación completa del AG en una arquitectura hardware/software nos obliga a plantearla como posible línea de trabajo futuro.

5.2 Líneas de trabajo futuras.

Durante el desarrollo de este proyecto nos hemos encontrado con multitud de problemas, unos más sencillos y otros más complejos. Buena parte de estos contratiempos ha sido abordada con solvencia, pero otra pequeña parte se ha quedado sin resolver a la espera de una investigación futura más profunda.

Comenzamos implementando en la FPGA únicamente la función fitness del algoritmo y comprobamos que dicha elección no suponía una gran mejora en el rendimiento global de la solución implementada (siempre y cuando utilizásemos OpenMP y no C puro), aunque ganábamos en eficiencia energética. También cabe decir que el uso del SVM quedó excluido en ambas implementaciones.

Como se ha mencionnado anteriormente, la posibilidad de utilizar FPGAs modernas de 16m y programación eficiente en VHDL hace que merezca la pena seguier explorando la implementación hardware de la función de fitness.

También, una línea de trabajo futura podría ser la implementación del algoritmo completo en la FPGA, utilizando múltiples unidades paralelas de fitness, pues aunque una sola función fitness no ofrezca un gran rendiminento, lo más razonable es que una implementación del algoritmo completo sí lo ofrezca, teniendo en cuenta la naturaleza altamente paralelizable de los algoritmos genéticos. Por ejemplo, en cada iteración del algoritmo podríamos realizar las siguientes tareas en paralelo:

- Seleccionar parejas de cromosomas. Si bien en este punto debemos tener cuidado, pues aunque la probabilidad sea baja, puede ocurrir que seleccionemos dos veces la misma pareja, lo cual no quiere decir que produzcan los mismos hijos, pues el tipo de cruce implementado es el "Two Point Crossover"con selección de puntos de corte aleatorios. Además también tenemos que tener en cuenta las posibles mutaciones, por lo que seleccionar dos veces la misma pareja tampoco es un problema grave, aunque sí resulta incorrecto.
- Realizar los cruces de todas las parejas seleccionadas. Este paso puede ser completamente paralelizable, ya que el resultado de un cruce no tiene influencia alguna sobre los restantes.
- Realizar las posibles mutaciones sobre los cromosomas resultantes. Al igual que la fase anterior, este punto también es completamente paralelizable, pues las mutaciones son independientes entre sí.
- Realizar evaluaciones de fitness de los individuos en paralelo. Según la capacidad de la FPGA, o número de FPGAs disponibles, y el número de individuos de la población, se podrían evaluar todos los fitness en paralelo, o bien un determinado número de fitness en paralelo, repitiéndo secuencialmente esta evaluación paralela para evaluar a toda la población.

El único aspecto que no puede ser paralelizado es la fase de reemplazo, pues podría ocurrir que dos procesos paralelos encontrasen al mismo tiempo el cromosoma con menor fitness, de manera que el primero en sustituir a este cromosoma sería a su vez reemplazado por el segundo, pensando éste que está sustituyendo al cromosoma con fitness más bajo, no sabiendo que ya ha sido reemplazado.

Como podemos observar, todavía hay bastante margen de mejora, lo cual no quiere decir que vaya a ser fácil, más bien todo lo contrario. Implementar un algoritmo genético en una FPGA no es una algo trivial, sino que resulta una tarea bastante compleja. No obstante, la esperable mejora del tendimiento, junto con una alta eficiencia energética, hace que merezca la pena seguir investigando e implementar todo el algoritmo en la FPGA.

Por otro lado, si la eficiencia energética no es nuestra prioridad y teniendo en cuenta el buen rendimiento ofrecido por la versión ***OpenMP***, podríamos aprovechar y hacer uso de esta tecnología para implementar el algoritmo genético al completo. La ventaja más significativa de seguir este camino es que podemos utilizar la implementación C que hemos desarrollado e incluir ciertas directivas de *OpenMP* para crear una versión paralela del algoritmo. Esto tiene como principales puntos positivos la sencillez de la implementación y la garantía de un buen rendimiento frente a la versión secuencial del algoritmo, lo cual quedó demostrado durante las pruebas de rendimiento realizadas sobre la función fitness.

Otra línea de trabajo futura podría ser llevar la implementación del algoritmo a una nueva tecnología de creciente popularidad que está ganando cada vez más adeptos en el campo de la programación paralela. La tecnología de la que hablamos es ***CUDA*** [Nvidia2015], una creación de ***Nvidia*** nacida en el año 2007 cuyo propósito es intentar aprovechar el gran paralelismo, y el alto ancho de banda de la memoria en las GPUs en aplicaciones con un gran coste aritmético frente a realizar numerosos accesos a memoria principal, lo que podría actuar de cuello de botella. Esta tecnología ha demostrado ofrecer un gran rendimiento en aplicaciones altamente paralelizables y en particular en algoritmos genéticos [Pospichal2010], por lo que resulta una buena opción si lo que buscamos es una mejora en el rendimiento de nuestro algoritmo. Además, al estar

basado en C, la curva de aprendizaje para cualquier programador es mucho menos pronunciada que la de aprender VHDL, ya que este es un lenguaje completamente diferente.

En definitiva, merece la pena explorar cada uno de estos caminos si con ello conseguimos mejorar la la eficiencia y rendimiento del algoritmo planteado, pues no debemos olvidar que la finalidad última de éste es la de ayudar en el avance del conocimiento para su aplicabilidad en el diagnóstico y clasificación del cáncer, permitiendo mejorar las expectativas de salud de la población.

6 Referencias.

[Ambroise2002] Ambroise, C., McLachlan, G.J.: Selection bias in gene extraction on the basis of microarray gene-expression data. Proceedings of the National Academy of Sciences of the United States of America 99 (2002) 6562–6566

[Anand2008] Anand, P., Kunnumakara, A. B., Sundaram, C., Harikumar, K. B., Tharakan, S. T., Lai, O. S., ⋯ Aggarwal, B. B. (2008). Cancer is a Preventable Disease that Requires Major Lifestyle Changes. Pharmaceutical Research, 25(9), 2097 - 2116. doi:10.1007/s11095-008-9661-9

[Ashenden1990] Ashenden, P. (1990). The VHDL Cookbook. 1st ed. [ebook] South Australia. Available at: http://www.ics.uci.edu/~alexv/154/VHDL-Cookbook.pdf [Accessed 5 Feb. 2015].

[Baraglia2001] Baraglia, R., R. Perego, et al. (2001). A Parallel Compact Genetic Algorithm for Multi-FPGA Partitioning. Ninth Euromicro Workshop on Parallel and Distributed Processing (PDP '01).

[Buell2007] Buell, D., El-Ghazawi, T., Gaj, K., Kindratenko, V.: High-Performance Reconfigurable Computing. Computer 40 (2007) 23-27.

[CancerUK2015] Cancer Research UK, (2015). Worldwide cancer statistics. [online] Available at: http://www.cancerresearchuk.org/health-professional/cancer-statistics/worldwide-cancer [Accessed 2 Feb. 2015].

[DeHon1999] DeHon, A. and Wawrzynek, J. (1999). Reconfigurable computing: what, why, and implications for design automation. Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361).

[DeJong1992] De Jong, K. and Spears, W. (1992). A formal analysis of the role of multi-point crossover in genetic algorithms. Annals of Mathematics and Artificial Intelligence, 5(1), pp.1-26.

[Diaz2007] Diaz-Gomez, P. A., & Hougen, D. F. (2007, January). Initial Population for Genetic Algorithms: A Metric Approach. In GEM (pp. 43-49).

[DiazGomez2007] Diaz-Gomez, P. A., & Hougen, D. F. (2007). Empirical Study: Initial Population Diversity and Genetic Algorithm Performance. Artificial Intelligence and Pattern Recognition, 2007, 334-341.

[Emam2003] Emam, H., M. A. Ashour, et al. (2003). Introducing an FPGA based - genetic algorithms in the applications of blind signals separation. The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC'03).

[Hauck2008] Hauck, S., DeHon, A.: Reconfigurable Computing, The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann (2008).

[Hidalgo2014] Hidalgo, J. I., J. M. Colmenar, et al. (2014). Solving GA-Hard Problems with EMMRS and GPGPUs. GECCO 2014.

[Huerta2006] Huerta, E.B., Duval, B., Hao, J.-K.: A Hybrid GA/SVM Approach for Gene Selection and Classification of Microarray Data. Lecture Notes in Computer Science Vol. 3907. Springer (2006) 34-44.

[Jebari2013] Jebari, K., & Madiafi, M. (2013). Selection methods for genetic algorithms. International Journal of Emerging Sciences, 3(4).

[Martínez2005] Martínez, M. E. (2005). Primary prevention of colorectal cancer: lifestyle, nutrition, exercise. In Tumor Prevention and Genetics III (pp. 177-211). Springer Berlin Heidelberg.

[Ministerio2015] Ministerio de Sanidad y Consumo, (2015). La situación del cáncer en España. [online] Available at: http://www.msssi.gob.es/ciudadanos/enfLesiones/enfNoTransmisibles/docs/situacionCancer.pdf [Accessed 2 Feb. 2015].

[Mohamad2007] Mohamad, M.S., Omatu, S., Deris, S., Zaiton, S., Hashim, M.: A model for gene selection and classification of gene expression data. Artificial Life and Robotics 11 (2007) 219-222.

[Mohamad2008] Mohamad, M. S., S. Omatu, et al. (2008). A Recursive Genetic Algorithm to Automatically Select Genes for Cancer Classification. 2nd International Workshop on Practical Applications of Computational Biology and Bioinformatics (IWPACBB 2008). J. M. C. e. al. Berlin Heidelberg, Springer. 49: 166-174.

[Mohamad2009] Mohamad, M. S., S. Omatu, et al. (2009). "A multi-objective strategy in genetic algorithms for gene selection of gene expression data." Artif Life Robotics 13: 410-413.

[Mostafa2004] Mostafa, H. E., A. I. Khadragi, et al. (2004). Hardware Implementation of Genetic Algorithm on FPGA. 21th National Radio Science Conference (NRSC2004).

[Nvidia2015] Nvidia.es, (2015). Procesamiento paralelo CUDA. [online] Available at: http://www.nvidia.es/object/cuda-parallel-computing-es.html [Accessed 9 Jul. 2015].

[OpenMP2015] The OpenMP® API specification for parallel programming (2015): http://openmp.org/

[Perez2000] Perez-Diez, A., Morgun, A., & Shulzhenko, N. (2000). Microarrays for Cancer Diagnosis and Classification.

[Pospichal2010] Pospichal, P., Jaros, J., & Schwarz, J. (2010). Parallel genetic algorithm on the cuda architecture. In Applications of Evolutionary Computation (pp. 442-451). Springer Berlin Heidelberg.

[Ramamritham2004] Ramamritham, K., Arya, K.: System Software for Embedded Applications. 17th IEEE International Conference on VLSI Design IEEE Press (2004) 22.

[Rocha1999] Rocha, M., & Neves, J. (1999). Preventing premature convergence to local optima in genetic algorithms via random offspring generation. In Multiple Approaches to Intelligent Systems (pp. 127-136). Springer Berlin Heidelberg.

[Russo2003] Russo, G., Zegar, C., & Giordano, A. (2003). Advantages and limitations of microarray technology in human cancer. Oncogene, 22(42), 6497-6507.

[RYLANDER2002] RYLANDER, S. G. B. (2002). Optimal population size and the genetic algorithm. Population, 100(400), 900.

[Ryu2002] Ryu, J., Cho, S.-B.: Towards Optimal Feature and Classifier for Gene Expression Classification of Cancer. Advances in Soft Computing - AFSS 2002. Springer (2002) 310–317.

[Song2014] Song, X., Wang, H., & Wang, L. (2014, April). FPGA implementation of a support vector machine based classification system and its potential application in smart grid. In Information Technology: New Generations (ITNG), 2014 11th International Conference on (pp. 397-402). IEEE.

[Stu2008] Stu, Mesnier; Jain, Raj (2008). Measuring the Effectiveness of FPGA Programming Languages. Project Report. Department of Computer Science & Engineering, Washington University: www.cse.wustl.edu/~jain/cse567-08/ftp/fpgaprog/index.html

[Vahid2007] Vahid, F., Lysecky, R.: VHDL for Digital Design Wiley (2007).

[VHDL1995] VHDL Data Types. (1995). 1st ed. [ebook] Chicago: University of Illinois. Available at: http://www.ece.uic.edu/~dutt/courses/ece368/lect-notes/vhdl-attrs-data-types.pdf [Accessed 5 Feb. 2015].

[Wang2005] Wang, Y., Tetko, I. V., Hall, M. A., Frank, E., Facius, A., Mayer, K. F., & Mewes, H. W. (2005). Gene selection from microarray data for cancer classification—a machine learning approach. Computational biology and chemistry, 29(1), 37-46.

[Xilinx2015] Xilinx Inc., http://www.xilinx.com.

Anexos

Anexo I. Aplicación C

GENE.H

```
#ifndef GENE_H
#define GENE_H

#include "Commons.h"

int MAX_GENES;
int NUM_CARCINOGEN;
int *CARCINOGEN_GENES;

typedef struct Gene {
  int        feature;  //array  containing  the
features of the chromosome
  bool   carcinogen;
} Gene;

Gene* createGene(int feature);
int   getFeature(Gene *gene);
bool  isCarcinogen(Gene *gene);
void  setCarcinogen(Gene *gene);
void  freeGene(Gene *gene);

#endif
```

GENE.C

```
#include "Gene.h"

Gene* createGene(int feature)
{
        Gene *gene = my_malloc(sizeof(Gene));

        gene->feature = feature;
        setCarcinogen(gene);

        return gene;
}

int getFeature(Gene *gene)
{
        return gene->feature;
}

bool isCarcinogen(Gene *gene)
{
  return gene->carcinogen;
}

void setCarcinogen(Gene *gene)
{
  int i;

  gene->carcinogen = false;
  for(i = 0; i < NUM_CARCINOGEN; i++) {
    if(CARCINOGEN_GENES[i] == gene->feature) {
      gene->carcinogen = true;
      break;
    }
  }
}

void freeGene(Gene *gene)
{
        free(gene);
}
```

CHROMOSOME.H

```
#ifndef CHROMOSOME_H
#define CHROMOSOME_H

#include "Gene.h"

int   NUM_FEATURES;
float WEIGHT_1;
float WEIGHT_2;

typedef struct Chromosome {
  Gene     **features; //array containing the features of the chromosome
  int     numFeatures;
  float   fitness;
  float   normalized_fitness;
} Chromosome;

Chromosome*    createChromosome(int  *features, int num_features);
Chromosome*          cloneChromosome(Chromosome *chromosome);
bool            alreadyInChromosome(Chromosome* chromosome, Gene* gene, int position);
int*                  featuresToInt(Chromosome* chromosome);
Chromosome**  crossover(Chromosome  *parent_1, Chromosome *parent_2);
void         mutate(Chromosome *chromosome);
float               calculateFitness(Chromosome *chromosome);
float           setNormalizedFitness(Chromosome *chromosome,    float    totalFitness,    float aggregatedFitness);
float                     getFitness(Chromosome *chromosome);
float           getNormalizedFitness(Chromosome *chromosome);
void                printChromosome(Chromosome *chromosome);
void                 freeChromosome(Chromosome *chromosome);

#endif
```

CHROMOSOME.C

```
#include <string.h>
#include "Chromosome.h"

extern int MAX_GENES;

Chromosome*   createChromosome(int   *features, int num_features)
{
  int i;
  Chromosome *chromosome;

  chromosome           =           (Chromosome *)my_malloc(sizeof(Chromosome));
  chromosome->features               =    (Gene **)my_malloc(num_features * sizeof(Gene *));

  chromosome->numFeatures                     = num_features;
  chromosome->fitness            = 0;
  chromosome->normalized_fitness = 0;
  for(i = 0; i < num_features && features != NULL; i++) {
    chromosome->features[i]                   = createGene(features[i]);
  }

  return chromosome;
}

Chromosome*          cloneChromosome(Chromosome *chromosome)
{
  int *features;
  Chromosome *cloneChromosome;

  features        = featuresToInt(chromosome);
  cloneChromosome = createChromosome(features, chromosome->numFeatures);

  cloneChromosome->fitness    =    chromosome->fitness;

  free(features);

  return cloneChromosome;
}

bool            alreadyInChromosome(Chromosome* chromosome, Gene* gene, int position)
{
  int i;
  bool exists = false;

  for(i = 0; i < position && exists == false; ++i) {
    if(getFeature(gene)                     == getFeature(chromosome->features[i])) {
      exists = true;
```

```
    }
  }

  return exists;
}

int* featuresToInt(Chromosome* chromosome)
{
  int i, *features;

  features = (int *)my_malloc(chromosome->numFeatures * sizeof(int));
  for(i = 0; i < chromosome->numFeatures; i++) {
    features[i] = getFeature(chromosome->features[i]);
  }

  return features;
}

/**
* Two-point crossovers
*/
Chromosome** crossover(Chromosome *parent_1, Chromosome *parent_2)
{
  int i, num_children, num_features, point_1, point_2, aux;
  Chromosome **children;
  num_features = parent_1->numFeatures;
  num_children = 2;

  children = (Chromosome **)my_malloc(num_children * sizeof(Chromosome *));
  for(i = 0; i < num_children; i++) {
    children[i] = createChromosome(NULL, num_features);
  }

  point_1 = rand() % num_features;
  do {
    point_2 = rand() % num_features;
  } while(point_1 == point_2);
  if(point_1 > point_2) {
    aux = point_1;
    point_1 = point_2;
    point_2 = aux;
  }

  for (i = 0; i < point_1; ++i) {
    children[0]->features[i] = createGene(getFeature(parent_1->features[i]));
    children[1]->features[i] = createGene(getFeature(parent_2->features[i]));
  }
  for (; i < point_2; ++i) {
    children[0]->features[i] = createGene(getFeature(parent_2->features[i]));
    children[1]->features[i] = createGene(getFeature(parent_1->features[i]));
  }
  for (; i < num_features; ++i) {
    children[0]->features[i] = createGene(getFeature(parent_1->features[i]));
    children[1]->features[i] = createGene(getFeature(parent_2->features[i]));
  }

  /* If we use static arrays to store the genes, the code below gives better performance
  memcpy(children[0]->features, parent_1->features, point_1 * sizeof(struct Gene));
  memcpy(children[1]->features, parent_2->features, point_1 * sizeof(struct Gene));
  memcpy(children[0]->features + point_1, parent_2->features + point_1, (point_2 - point_1) * sizeof(struct Gene));
  memcpy(children[1]->features + point_1, parent_1->features + point_1, (point_2 - point_1) * sizeof(struct Gene));
  memcpy(children[0]->features + point_2, parent_1->features + point_2, (num_features - point_2) * sizeof(struct Gene));
  memcpy(children[1]->features + point_2, parent_2->features + point_2, (num_features - point_2) * sizeof(struct Gene));
  */

  return children;
}

void mutate(Chromosome *chromosome)
{
  int gene, position;

  gene = rand() % MAX_GENES;
  position = rand() % chromosome->numFeatures;

  freeGene(chromosome->features[position]);
  chromosome->features[position] = createGene(gene);
}

/**
* f(x) = A(x)*w1 + w2*(M - R(x))/M --> A(x) ->suma de elementos cancerígenos, M--> total genes, R(X)--> genes seleccionados
*/
float calculateFitness(Chromosome *chromosome)
{
  int i, A, R, M;

  A = 0;

  for (i = 0; i < chromosome->numFeatures; ++i) {
    if(isCarcinogen(chromosome->features[i]) == true && alreadyInChromosome(chromosome, chromosome->features[i], i) == false) {
      A++;;
```

```
    }
  }

  M = MAX_GENES;
  R = chromosome->numFeatures - A; //Number of
selected features - Carcinogen genes selected

  chromosome->fitness = (A * WEIGHT_1) +
(WEIGHT_2 * (M - R))/M;

  return chromosome->fitness;
}

float          setNormalizedFitness(Chromosome
*chromosome,    float    totalFitness,    float
aggregatedFitness)
{
  chromosome->normalized_fitness             =
(chromosome->fitness    /    totalFitness)    +
aggregatedFitness;

  return chromosome->normalized_fitness;
}

float getFitness(Chromosome *chromosome)
{
  return chromosome->fitness;
}

float          getNormalizedFitness(Chromosome
*chromosome)
{
  return chromosome->normalized_fitness;
}

void printChromosome(Chromosome *chromosome)
{
  int i, numFeatures = chromosome->numFeatures
- 1;
  printf("[");
  for(i = 0; i < numFeatures; ++i) {
    printf("%d,    ",    getFeature(chromosome-
>features[i]));
  }
  if(chromosome->numFeatures > 0) {
    printf("%d",         getFeature(chromosome-
>features[numFeatures]));
  }
  printf("] -> %f", chromosome->fitness);
}

void freeChromosome(Chromosome *chromosome)
{
  int i;

  for(i = 0; i < chromosome->numFeatures; i++)
{
    freeGene(chromosome->features[i]);
  }
  free(chromosome->features);
  free(chromosome);
}
```

CHROMOSOMELIST.H

```
#ifndef CHROMOSOMELIST_H
#define CHROMOSOMELIST_H

#include "Chromosome.h"

typedef struct ChromosomeList {
  struct Chromosome **list;
  float               minFitness;
  int                 size;
  int                 occupied;
} ChromosomeList;

ChromosomeList*       createChromosomeList(int
size);
void
addChromosomeToList(ChromosomeList
*chromosomeList, Chromosome *chromosome);
Chromosome*         chromosomeAt(ChromosomeList
*chromosomeList, int index);
float              getMinFitness(ChromosomeList
*chromosomeList);
int                  getListSize(ChromosomeList
*chromosomeList);
int              getListOccupied(ChromosomeList
*chromosomeList);
void
printChromosomeList(ChromosomeList
*chromosomeList);
void
freeChromosomeList(ChromosomeList
*chromosomeList);

#endif
```

CHROMOSOMELIST.C

```
/*
 * ChromosomeList.c
 *
 *  Created on: 02/02/2015
 *      Author: jose
 */

#include "ChromosomeList.h"

ChromosomeList* createChromosomeList(int size)
{
  ChromosomeList       *chromosomeList       =
(ChromosomeList
*)my_malloc(sizeof(ChromosomeList));
  chromosomeList->list           = (Chromosome
**)my_malloc(size * sizeof(Chromosome*));
  chromosomeList->minFitness     = 0;
  chromosomeList->occupied       = 0;
  chromosomeList->size           = size;

  return chromosomeList;
}

void         addChromosomeToList(ChromosomeList
*chromosomeList, Chromosome *chromosome)
{
  int i, position;

  for(i = 0; i < chromosomeList->occupied &&
getFitness(chromosomeList->list[i])          <
getFitness(chromosome); ++i);
  position = (i == chromosomeList->size ? (i -
1) : i);

  if(chromosomeList->occupied                <
chromosomeList->size) {
    for(i  =  chromosomeList->occupied;  i  >
position; --i) {
      chromosomeList->list[i]                =
chromosomeList->list[i - 1];
    }
    chromosomeList->occupied++;
  }
  else {
    freeChromosome(chromosomeList->list[0]);
    for(i = 0; i < position; ++i) {
      chromosomeList->list[i]                =
chromosomeList->list[i + 1];
    }
  }

  chromosomeList->list[position] = chromosome;
  chromosomeList->minFitness                 =
getFitness(chromosomeList->list[0]);
}
```

```
Chromosome*         chromosomeAt(ChromosomeList
*chromosomeList, int index)
{
        return chromosomeList->list[index];
}

float              getMinFitness(ChromosomeList
*chromosomeList)
{
  return chromosomeList->minFitness;
}

int                  getListSize(ChromosomeList
*chromosomeList)
{
  return chromosomeList->size;
}

int              getListOccupied(ChromosomeList
*chromosomeList)
{
  return chromosomeList->occupied;
}

void         printChromosomeList(ChromosomeList
*chromosomeList)
{
  int i;

  for(i  =  0;  i  <  chromosomeList->occupied;
++i) {
    printf("\n");
    printChromosome(chromosomeList->list[i]);
  }
  printf("\n");
}

void          freeChromosomeList(ChromosomeList
*chromosomeList)
{
  int i;

  for(i  =  0;  i  <  chromosomeList->occupied;
++i) {
    freeChromosome(chromosomeList->list[i]);
  }
  free(chromosomeList->list);

  free(chromosomeList);
}
```

CONFIGREADER.H

```
#ifndef CONFIGREADER_H
#define CONFIGREADER_H

#include "Commons.h"
#include "libconfig/libconfig.h"

#define CONFIG_ERROR -1
#define CONFIG_RIGHT  1

typedef struct ConfigReader {
  //Gene vars
  int maxGeneNumber;
  int numCarcinogen;
  int *carcinogenGenes;

  //Chromosome vars
  int   numFeatures;
  double weight1;
  double weight2;

  //GA vars
  int   populationSize;
  int** population;
  int   mutationProb;
  int   maxGenerations;
  int   elitism;
  int   crossoversPerGen;

} ConfigReader;

ConfigReader* createConfigReader();
int            readGeneVarsConfig(ConfigReader*
configReader, config_t *cfg);
int
readChromosomeVarsConfig(ConfigReader*
configReader, config_t *cfg);
int              readGAVarsConfig(ConfigReader*
configReader, config_t *cfg);
int                readConfigFile(ConfigReader*
configReader, char* configFile);
int            getCRMaxGeneNumber(ConfigReader*
configReader);
int
getCRNumCarcinogenGenes(ConfigReader*
configReader);
int*
getCRCarcinogenGenes(ConfigReader*
configReader);
int
getCRFeaturesPerChromosome(ConfigReader*
configReader);
float                getCRWeight1(ConfigReader*
configReader);
float                getCRWeight2(ConfigReader*
configReader);
int
getCRPopulationSize(ConfigReader*
configReader);
int**             getCRPopulation(ConfigReader*
configReader);
int
getCRMutationProbability(ConfigReader*
configReader);
int
getCRMaxGenerations(ConfigReader*
configReader);
int                  getCRElitism(ConfigReader*
configReader);
int
getCRCrossoversPerGeneration(ConfigReader*
configReader);
void             freeConfigReader(ConfigReader*
configReader);

#endif
```

CONFIGREADER.C

```
#include "ConfigReader.h"

ConfigReader* createConfigReader()
{
  int i, j;
  ConfigReader *configReader = (ConfigReader *)my_malloc(sizeof(ConfigReader));

  //Gene vars
  configReader->maxGeneNumber   = 100;
  configReader->numCarcinogen   = 20;
  configReader->carcinogenGenes     =     (int *)my_malloc(configReader->numCarcinogen       * sizeof(int));
  for (i = 0; i < configReader->numCarcinogen; ++i) {
    configReader->carcinogenGenes[i] = rand() % configReader->maxGeneNumber;
  }

  //Chromosome vars
  configReader->numFeatures = 10;
  configReader->weight1     = 0.65;
  configReader->weight2               =   1   - configReader->weight1;

  //GA vars
  configReader->mutationProb     = 15;
  configReader->maxGenerations   = 100;
  configReader->elitism          = 20;
  configReader->crossoversPerGen = 10;
  configReader->populationSize   = 10;
  configReader->population             =  (int **)my_malloc(configReader->populationSize     * sizeof(int *));
  for(i = 0; i < configReader->populationSize; ++i)
  {
    configReader->population[i]     =     (int *)my_malloc(configReader->numFeatures         * sizeof(int));
    for (j = 0; j < configReader->numFeatures; ++j)
    {
      configReader->population[i][j] = (10 * i) + j;
    }
  }

  return configReader;
}

int           readGeneVarsConfig(ConfigReader* configReader, config_t *cfg)
{
  int i;
  config_setting_t *setting;

  setting = config_lookup(cfg, "GENE_VARS");
  if(setting != NULL)
  {
    if(!(config_setting_lookup_int(setting, "MAX_GENE_NUMBER",               &configReader->maxGeneNumber)))
      return CONFIG_ERROR;

    setting                                  = config_setting_get_member(setting, "CARCINOGEN_GENES");
      if(setting != NULL)
      {
        configReader->numCarcinogen          = config_setting_length(setting);
        for(i   =   0;   i   <   configReader->numCarcinogen; ++i)
        {
          configReader->carcinogenGenes[i]    = config_setting_get_int_elem(setting, i);
        }
      }
      else
        return CONFIG_ERROR;
  }
  else
    return CONFIG_ERROR;

  return CONFIG_RIGHT;
}

int     readChromosomeVarsConfig(ConfigReader* configReader, config_t *cfg)
{
  config_setting_t *setting;

  setting         =         config_lookup(cfg, "CHROMOSOME_VARS");
  if(setting != NULL)
  {
    if(!(config_setting_lookup_int(setting, "NUM_FEATURES", &configReader->numFeatures)
         &&
config_setting_lookup_float(setting, "WEIGHT_1", &configReader->weight1)
         &&
config_setting_lookup_float(setting, "WEIGHT_2", &configReader->weight2)))
      return CONFIG_ERROR;
  }
  else
    return CONFIG_ERROR;
  return CONFIG_RIGHT;
}

int              readGAVarsConfig(ConfigReader* configReader, config_t *cfg)
{
```

```
  int i, j, numFeatures;
  config_setting_t *setting, *chromosome;

  setting = config_lookup(cfg, "GA_VARS");
  if(setting != NULL)
  {
    if(!(config_setting_lookup_int(setting,
"MUTATION_PROBABILITY",          &configReader-
>mutationProb)
         &&  config_setting_lookup_int(setting,
"MAX_GENERATIONS",               &configReader-
>maxGenerations)
         &&  config_setting_lookup_int(setting,
"ELITISM", &configReader->elitism)
         &&  config_setting_lookup_int(setting,
"CROSSOVERS_PER_GEN",            &configReader-
>crossoversPerGen)))
      return CONFIG_ERROR;

    setting                                  =
config_setting_get_member(setting,
"POPULATION");
    if(setting != NULL)
    {
      configReader->populationSize           =
config_setting_length(setting);
      for(i    =    0;    i    <    configReader-
>populationSize; ++i)
      {
        chromosome                           =
config_setting_get_elem(setting, i);
        numFeatures                          =
config_setting_length(chromosome);
        if(numFeatures     !=     configReader-
>numFeatures)
          return CONFIG_ERROR;
        for(j = 0; j < numFeatures; ++j)
        {
          configReader->population[i][j]     =
config_setting_get_int_elem(chromosome, j);
        }
      }
    }
    else
      return CONFIG_ERROR;
  }
  else
    return CONFIG_ERROR;
  return CONFIG_RIGHT;
}

int readConfigFile(ConfigReader* configReader,
char* configFile)
{
  config_t cfg;

  config_init(&cfg);

  /* Read the file. If there is an error,
report it and exit. */
  if(! config_read_file(&cfg, configFile))
  {
    fprintf(stderr,     "%s:%d     -     %s\n",
config_error_file(&cfg),
            config_error_line(&cfg),
config_error_text(&cfg));
    config_destroy(&cfg);
    return CONFIG_ERROR;
  }
  if(readGeneVarsConfig(configReader, &cfg) ==
CONFIG_ERROR
      ||
readChromosomeVarsConfig(configReader,    &cfg)
== CONFIG_ERROR
      ||  readGAVarsConfig(configReader,  &cfg)
== CONFIG_ERROR)
  {
   config_destroy(&cfg);
   return CONFIG_ERROR;
  }
  config_destroy(&cfg);
  return CONFIG_RIGHT;
}

int            getCRMaxGeneNumber(ConfigReader*
configReader)
{
  return configReader->maxGeneNumber;
}

int       getCRNumCarcinogenGenes(ConfigReader*
configReader)
{
  return configReader->numCarcinogen;
}

int*         getCRCarcinogenGenes(ConfigReader*
configReader)
{
  return configReader->carcinogenGenes;
}

int    getCRFeaturesPerChromosome(ConfigReader*
configReader)
{
  return configReader->numFeatures;
}

float getCRWeight1(ConfigReader* configReader)
{
  return (float)configReader->weight1;
}

float getCRWeight2(ConfigReader* configReader)
{
  return (float)configReader->weight2;
}

int           getCRPopulationSize(ConfigReader*
configReader)
{
```

```
  return configReader->populationSize;
}

int**             getCRPopulation(ConfigReader*
configReader)
{
  return configReader->population;
}

int      getCRMutationProbability(ConfigReader*
configReader)
{
  return configReader->mutationProb;
}

int           getCRMaxGenerations(ConfigReader*
configReader)
{
  return configReader->maxGenerations;
}

int getCRElitism(ConfigReader* configReader)
{
  return configReader->elitism;
}

int  getCRCrossoversPerGeneration(ConfigReader*
configReader)
{
  return configReader->crossoversPerGen;
}

void             freeConfigReader(ConfigReader*
configReader)
{
  int i;

  free(configReader->carcinogenGenes);

  for    (i    =    0;    i    <    configReader-
>populationSize; ++i) {
    free(configReader->population[i]);
  }
  free(configReader->population);

  free(configReader);
}
```

GAHANDLER.H

```
#ifndef GAHANDLER_H
#define GAHANDLER_H

#include "ChromosomeList.h"
#include "ConfigReader.h"

#define CONFIG_FILE "gaselcancer.cfg"

int   BEST_POSIBLE_FITNESS;

typedef struct GAHandler {
  struct Chromosome **population;
  int populationSize;
  int mutationProb;
  int maxGenerations;
  int elitism;
  int crossoversPerGen;

} GAHandler;

GAHandler*      createGAHandler();
Chromosome**            getPopulation(GAHandler
*gaHandler);
Chromosome*         getBestChromosome(GAHandler
*gaHandler);
int
getWeakestChromosomeIndex(GAHandler
*gaHandler);
ChromosomeList* elitism(GAHandler *gaHandler);
int*                        selection(GAHandler
*gaHandler, int numParents);
Chromosome**           crossoverPhase(GAHandler
*gaHandler,  Chromosome  *parent_1,  Chromosome
*parent_2);
void             mutation(GAHandler *gaHandler,
Chromosome *chromosome);
void                     restoreElite(GAHandler
*gaHandler, ChromosomeList *elite);
void            applyGA(GAHandler *gaHandler);
void                  printPopulation(GAHandler
*gaHangler);
void                     freeGAHandler(GAHandler
*gaHandler);

#endif
```

GAHANDLER.C

```
#include "GAHandler.h"

extern int   MAX_GENES;
extern int   NUM_FEATURES;
extern float WEIGHT_1;
extern float WEIGHT_2;
extern int   NUM_CARCINOGEN;
extern int   *CARCINOGEN_GENES;

GAHandler* createGAHandler()
{
  int i, *carGenes, numGenes, **population;
  ConfigReader *config = createConfigReader();
  GAHandler     *gaHandler     =     (GAHandler
*)my_malloc(sizeof(GAHandler));

  //Reading Config file
  if(readConfigFile(config,   CONFIG_FILE)   ==
CONFIG_ERROR) {
    fprintf(stderr, "Error trying to read the
configuration file: %s\n", CONFIG_FILE);
    freeConfigReader(config);
    free(gaHandler);
    return NULL;
  }

  // Initializing seed
  srand(time(NULL));

  MAX_GENES      = getCRMaxGeneNumber(config);
  NUM_FEATURES                                =
getCRFeaturesPerChromosome(config);
  WEIGHT_1       = getCRWeight1(config);
  WEIGHT_2       = getCRWeight2(config);
  NUM_CARCINOGEN                              =
getCRNumCarcinogenGenes(config);
  carGenes                                    =
getCRCarcinogenGenes(config);
  CARCINOGEN_GENES            =            (int
*)my_malloc(NUM_CARCINOGEN * sizeof(int));
  for(i = 0; i < NUM_CARCINOGEN; ++i) {
    CARCINOGEN_GENES[i] = carGenes[i];
  }

  BEST_POSIBLE_FITNESS     =     (WEIGHT_1     *
NUM_FEATURES) + WEIGHT_2;

  gaHandler->populationSize                   =
getCRPopulationSize(config);
  gaHandler->mutationProb                     =
getCRMutationProbability(config);
  gaHandler->maxGenerations                   =
getCRMaxGenerations(config);
  gaHandler->elitism                          =
getCRElitism(config);
  gaHandler->crossoversPerGen                 =
getCRCrossoversPerGeneration(config);
```

```
  gaHandler->population          = (Chromosome
**)my_malloc(gaHandler->populationSize       *
sizeof(Chromosome *));


  // Generating initial population with all
the genes
  numGenes       =  MAX_GENES  /  gaHandler-
>populationSize;
  population = getCRPopulation(config);
  for (i = 0; i < gaHandler->populationSize;
++i) {
    gaHandler->population[i]                =
createChromosome(population[i], numGenes);
    calculateFitness(gaHandler-
>population[i]);
  }


  freeConfigReader(config);


  return gaHandler;
}


Chromosome**           getPopulation(GAHandler
*gaHandler)
{
  return gaHandler->population;
}


Chromosome*        getBestChromosome(GAHandler
*gaHandler)
{
  int i;
  struct Chromosome *chromosome;
  float             fitness, bestFitness = -1;


  for (i = 0; i < gaHandler->populationSize &&
bestFitness < BEST_POSIBLE_FITNESS; ++i) {
    fitness       =       getFitness(gaHandler-
>population[i]);
    if(fitness > bestFitness) {
      bestFitness = fitness;
      chromosome = gaHandler->population[i];
    }
  }


  return chromosome;
}


int        getWeakestChromosomeIndex(GAHandler
*gaHandler)
{
  int   i, position = 0;
  float fitness       = getFitness(gaHandler-
>population[position]);


  for(i  =  1;  i  <  gaHandler->populationSize;
i++) {
    if(getFitness(gaHandler->population[i])   <
fitness) {
      position = i;
      fitness       =    getFitness(gaHandler-
>population[i]);
    }
  }


  return position;
}


ChromosomeList* elitism(GAHandler *gaHandler)
{
  int  i,  elite_size  =  (gaHandler->elitism  *
gaHandler->populationSize) / 100;
  Chromosome*     chromosome;
  ChromosomeList*           elite           =
createChromosomeList(elite_size);


  if(elite->size > 0) {
    for(i  =  0;  i  <  gaHandler->populationSize;
++i) {
      if(getMinFitness(elite)                <
getFitness(gaHandler->population[i])        ||
getListOccupied(elite) < getListSize(elite)) {
        chromosome                             =
cloneChromosome(gaHandler->population[i]);
        addChromosomeToList(elite,
chromosome);
      }
    }
  }


  return elite;
}


/**
 * Roulette Wheel Selection
 */
int*   selection(GAHandler   *gaHandler,   int
numParents)
{
  int     i,     j,     *parents     =     (int
*)my_malloc(numParents * sizeof(int));
  float normalizedFitness,
                totalFitness      = 0,
                aggregatedFitness = 0;


  for(i  =  0;  i  <  gaHandler->populationSize;
++i) {
    totalFitness    +=    getFitness(gaHandler-
>population[i]);
  }


  //Normalizing fitness
  for(i  =  0;  i  <  gaHandler->populationSize;
++i) {
    aggregatedFitness                         +=
setNormalizedFitness(gaHandler->population[i],
totalFitness, aggregatedFitness);
  }


  //Roulette wheel selection
  for(i = 0; i < numParents; ++i) {
```

```
    do {
      normalizedFitness = (rand() % 1000 ) / 1000.0; // We generate a random number between 0.000 and 0.999
      for(j                =                0; getNormalizedFitness(gaHandler->population[j]) < normalizedFitness; ++j);
      parents[i] = j;
    } while(i != 0 && parents[0] == j);
  }

  return parents;
}

Chromosome**          crossoverPhase(GAHandler *gaHandler, Chromosome *parent_1, Chromosome *parent_2)
{
  Chromosome **children = crossover(parent_1, parent_2);

  return children;
}

void mutation(GAHandler *gaHandler, Chromosome *chromosome)
{
  int mutateRate = rand() % 100;

  if(mutateRate < gaHandler->mutationProb) {
          mutate(chromosome);
  }
}

void     restoreElite(GAHandler     *gaHandler, ChromosomeList *elite)
{
  int             i,    position,    size    = getListOccupied(elite);
  float fitness;
  Chromosome *chromosome;

  for(i = 0; i < size; ++i) {
    position                                  = getWeakestChromosomeIndex(gaHandler);
    fitness         =     getFitness(gaHandler->population[position]);
    for(;       i       <       size       && (getFitness(chromosomeAt(elite,     i))     <= fitness); ++i);
    if(i < size) {
      freeChromosome(gaHandler->population[position]);
      chromosome                              = cloneChromosome(chromosomeAt(elite, i));
      gaHandler->population[position]         = chromosome;
    }
  }
}
```

```
void applyGA(GAHandler *gaHandler)
{
  int i, j, k, *parents,
      numParents  = 2,
      numChildren = 2;
  Chromosome     **children;
  ChromosomeList  *elite;

  printPopulation(gaHandler);
  for(k  =  0;  k  <  gaHandler->maxGenerations; ++k) {

printf("\n=================================================\n");
    printf("Generation: %d\n", k + 1);
    elite = elitism(gaHandler);
    for(i     =     0;     i     <     gaHandler->crossoversPerGen; ++i) {
      parents         =     selection(gaHandler, numParents);
      children                                  = crossoverPhase(gaHandler,gaHandler->population[parents[0]],             gaHandler->population[parents[1]]);
      for(j = 0; j < numChildren; ++j) {
        mutation(gaHandler, children[j]);
        calculateFitness(children[j]);
        freeChromosome(gaHandler->population[parents[j]]);
        gaHandler->population[parents[j]]     = children[j]; // Children always replace their parents
      }
      free(children);
      free(parents);
    }
    restoreElite(gaHandler, elite);
    printPopulation(gaHandler);

    freeChromosomeList(elite);
  }
}

void printPopulation(GAHandler *gaHangler)
{
  int i;

  printf("\n##############            Population ##############\n");
  for(i  =  0;  i  <  gaHangler->populationSize; ++i) {
    printf("Chromosome %d: ", i);
    printChromosome(gaHangler->population[i]);
    printf("\n");
  }
}

void freeGAHandler(GAHandler *gaHandler)
{
  int i;
```

```
  for (i = 0; i < gaHandler->populationSize;
++i) {
    freeChromosome(gaHandler->population[i]);
  }
  free(gaHandler->population);
  free(CARCINOGEN_GENES);

  free(gaHandler);
}
```

COMMONS.H

```
#ifndef COMMONS_H
#define COMMONS_H

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <stdbool.h>

void* my_malloc(size_t size);

#endif
```

COMMONS.C

```
#include "Commons.h"

void *my_malloc(size_t size)
{
  void *ptr;
  if(size<=0) size=1; /* for AIX compatibility
*/
  ptr=(void *)malloc(size);
  if(!ptr) {
    perror ("Out of memory!\n");
    exit (1);
  }
  return(ptr);
}
```