***Títol:*** Integración de un entorno de compilación automática para FPGAs

***Volum:*** 1/1

***Alumne:*** Sara García Fernández

***Director/Ponent:*** Carlos Álvarez Martínez

***Departament:*** AC

***Data:*** 27/01/2009

**DADES DEL PROJECTE**

*Títol del Projecte:* Integración de un entorno de compilación automática para FPGAs

*Nom de l'estudiant:* Sara García Fernández
*Titulació:* Ingeniería Informática
*Crèdits:* 37.5
*Director/Ponent:* Carlos Álvarez Martínez
*Departament:* AC

**MEMBRES DEL TRIBUNAL** *(nom i signatura)*

*President:* Daniel Jiménez González

*Vocal:* Ana Cristina Zoltán Torres

*Secretari:* Carlos Álvarez Martínez

**QUALIFICACIÓ**

*Qualificació numèrica:*
*Qualificació descriptiva:*

*Data:*

**AGRADECIMIENTOS:**

A Carlos por ayudarme en todo momento.
A mi familia por aguantarme y apoyarme durante todo este tiempo.
A David por estar siempre a mi lado.

# Índice general

**1. Introducción** **3**
1.1. Objetivos y Planteamiento . . . . . 6
1.2. Organización . . . . . 8

**2. Las FPGAs y el hardware programable** **9**
2.1. Arquitectura de una FPGA . . . . . 13
2.2. Las FPGAs y los supercomputadores . . . . . 17
2.3. Entorno de trabajo . . . . . 20
2.3.1. Altix . . . . . 20
2.3.2. FPGApc . . . . . 22
2.3.3. Ordenador personal . . . . . 23

**3. Desarrollo** **25**
3.1. Funcionamiento de la FPGA . . . . . 25
3.2. Entender el código C . . . . . 27
3.3. Entender el código Verilog . . . . . 29
3.4. Automatización . . . . . 34
3.5. Ampliaciones . . . . . 40
3.6. Optimizaciones . . . . . 49

**4. Resultados** **53**
4.1. Limitaciones . . . . . 54
4.2. Contador de tiempos . . . . . 55

**5. Análisis temporal y económico** **67**
5.1. Análisis temporal . . . . . 67
5.2. Análisis económico . . . . . 70
5.2.1. Coste personal . . . . . 70
5.2.2. Coste del hardware . . . . . 71
5.2.3. Coste del software . . . . . 72
5.2.4. Coste total . . . . . 73

**6. Conclusiones** **75**
6.1. Trabajo futuro . . . . . 76

**Bibliografía** **79**

**A. Código Verilog** **83**

**B. Contenido del CD** **91**

# Capítulo 1

# Introducción

Las FPGAs se están volviendo más y más populares en la computación de alto rendimiento gracias a que con ellas se pueden conseguir buenos tiempos de ejecución en algunas aplicaciones y al mismo tiempo ahorrar mucha energía. Sin embargo, este tipo de dispositivos son todavía difíciles de programar.

En este trabajo se ha pretendido facilitar el trabajo de los programadores que quieren realizar aplicaciones para ser ejecutadas en las FPGAs.

Hoy en día se está llegando a los límites de cálculo de una CPU, pero con la llegada de los procesadores multihilos, han llegado también los problemas de temperatura. Además, estos procesadores multihilos son de propósito general y, por tanto, no están diseñados para resolver de forma óptima algunos tipos de problemas clásicos.

Como posible solución a estos problemas, han aparecido los aceleradores de hardware o co-procesadores, con los que podemos acelerar las aplicaciones gracias a su paralelismo. Algunos de ellos son hardware especializado que ha sido diseñado para computar algunas funciones específicas (los circuitos

ASIC). Por tanto un ASIC es un circuito integrado hecho a la medida para un uso particular, y no para un propósito general [2].

Hay diferentes tipos de aceleradores hardware según los métodos que utilizan. En ocasiones se usan procesadores multinúcleo heterogéneos donde se usan simultáneamente, uno o varios núcleos de propósito general y otros núcleos como ASIC, FPGA o GPU.

Un ejemplo de procesador multinúcleo es tener una o más GPUs (Graphics Processor Unit) como co-procesador. Con este tipo de aceleradores se usa la GPU para acelerar partes de la aplicación. Estas GPUs son muy paralelas y son muy buenas para datos con coma flotante. Usando GPUs se puede conseguir aumentar la velocidad considerablemente en muchos problemas de computación a tiempo real. En la figura 1.1 se puede ver la típica arquitectura de una GPU, en concreto una GeForce 8800 GT [3].

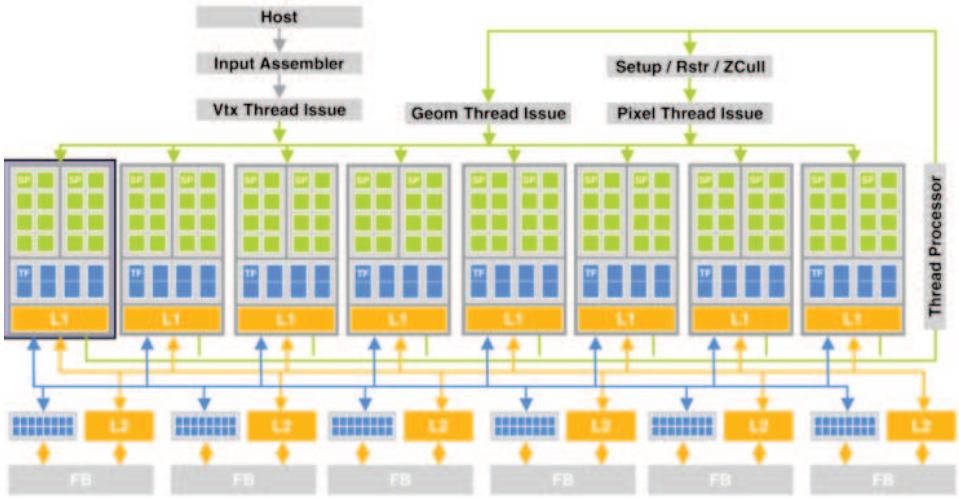



Figura 1.1: Arquitectura de una GPU [6]

Otro tipo de aceleradores son las FPGA (Field Programmable Gate Arrays, matriz de puertas programables), que contienen bloques de lógica programa-

ble e interconexiones programables que permiten a un modelo de FPGA ser usada en muchas aplicaciones distintas [7]. Con estos computadores reconfigurables podemos aumentar mucho la velocidad y ahorrar mucha energía comparado con computadores de propósito general. En la figura 1.2 se puede ver una arquitectura de una procesador usando una FPGA.

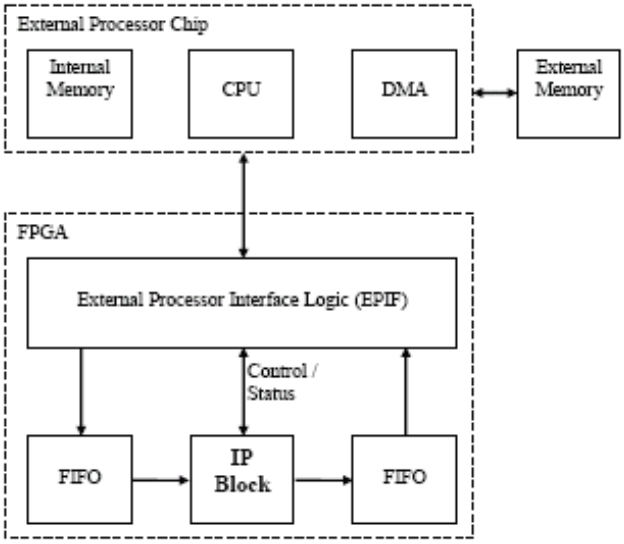



Figura 1.2: Arquitectura usando una FPGA [1]

Programar una FPGA es un trabajo complicado, ya que los programadores deben tener conocimientos sobre los lenguajes de descripción de hardware (HDL) como Verilog o VHL para describir la funcionalidad de estos dispositivos. Además, los lenguajes HDL son lenguajes de muy bajo nivel y son difíciles de debugar [4].

Este proyecto pretende evaluar y mejorar las ayudas técnicas a la implementación de algoritmos en C mediante el uso de hardware reconfigurable (FPGAs) y lenguaje HDL.

Para realizar esta tarea, se han estudiado las diferentes herramientas de la traducción de lenguaje C a lenguaje Verilog (HDL), así como sus limitaciones a la hora de trabajar con máquinas reales.

Como resultado de esta evaluación se ha decidido crear una herramienta que escriba un algoritmo en lenguaje HDL automáticamente a partir de un algoritmo en lenguaje C.

El lenguaje HDL elegido para este proyecto es Verilog, ya que es uno de los lenguajes HDL más extendido y además utiliza una sintaxis muy similar a la que utiliza el lenguaje C [5].

## 1.1. Objetivos y Planteamiento

El objetivo final del proyecto es crear una herramienta en Python que a través del código C (no cualquier código, sino uno destinado a ser usado en una FPGA) cree el código Verilog asociado y que al compilarlo y ejecutarlo en la FPGA funcione correctamente.

Para llegar a este objetivo final, ha sido necesario pasar por unos objetivos parciales.

Primero, ha sido necesario familiarizarse con el entorno de trabajo; tanto lo necesario para conectarse desde el exterior a la FPGA y ejecutar un código, como lo necesario para conectarse a un ordenador y compilar el código que, una vez compilado, se ejecuta en la FPGA. Además, ha sido necesario aprender a utilizar programas como Eclipse y Subversion, así como un aprendizaje previo sobre Python.

Uno de los objetivos iniciales ha sido aprender a compilar el código Verilog

y ejecutar el código resultante en la FPGA usando el código C.

Otro de los objetivos iniciales ha sido entender el código C que se usa para comunicar con la FPGA, y que tiene una parte especial y única en el caso las FPGAs. Además, otra tarea ha sido entender el código Verilog que se ejecuta en la FPGA. El problema es que no hay demasiados ejemplos, así que entender un ejemplo inicial (y simple) ha sido un objetivo muy importante para la realización de este trabajo.

El objetivo siguiente ha sido empezar a automatizar el proceso de creación del código en Verilog a través del Código C. En este paso se ha automatizado el ejemplo inicial, para obtener el mismo código Verilog del ejemplo (o equivalente) de forma automática. Así, automáticamente se podía obtener el código en Verilog del ejemplo sencillo a través del código C.

El siguiente problema que se ha planteado es que no hay más ejemplos de códigos Verilog correctos, con lo que para seguir con el proyecto se han tenido que crear códigos Verilog para usar como muestra.

Así, durante la última parte del proyecto se ha seguido un bucle en los objetivos, que ha consistido en tres pasos. Primero, crear un código Verilog correcto para usar como muestra con un nuevo caso o algoritmo para probar. Segundo, añadir ese nuevo caso programa en Python para que se cree automáticamente ese caso en Verilog. Tercero y último, probar casos similares para comprobar que efectivamente se crea el código Verilog correcto para cada caso concreto.

Además, para comprobar el resultado del proyecto, durante la última etapa se han medido los tiempos de ejecución de algunos algoritmos, comparando el tiempo que tardan en ejecutarse en la FPGA por hardware y en

software.

Y por último, para acabar el proyecto se ha redactado este documento para explicar todo el proceso.

## 1.2. Organización

Los siguientes capítulos del documento están estructurados como se explica a continuación:

- **Capítulo 2:** Describe los conceptos generales sobre las FPGAs y el hardware programable. Además, lista los elementos que han sido necesarios para realizar este trabajo.
- **Capítulo 3:** Muestra el desarrollo del proyecto.
- **Capítulo 4:** Muestra las evaluaciones y resultados del trabajo y los compara con los objetivos.
- **Capítulo 5:** Estudio del coste temporal y económico de este proyecto.
- **Capítulo 6:** Muestra las conclusiones del trabajo y lo que se podría llevar a cabo en un posible trabajo futuro.

# Capítulo 2

# Las FPGAs y el hardware programable

En el mundo de los ordenadores y la electrónica, estamos acostumbrados a dos formas diferentes de computación: hardware y software. En el caso de la computación hardware, los circuitos integrados para aplicaciones específicas (ASICs) proporcionan unos recursos altamente optimizados para realizar tareas críticas muy rápidamente, pero se configuran permanentemente para una única aplicación mediante un costoso diseño y grandes esfuerzos de fabricación. Por otro lado, en el caso del software, proporciona la flexibilidad de cambiar las aplicaciones y realizar un gran número de tareas diferentes, pero en órdenes de magnitud peores que los ASIC en términos de rendimiento y uso de energía.

Las FPGAs (del inglés *Field Programmable Gate Array*) son unos dispositivos revolucionarios que unen los beneficios del hardware y el software [22]. Las FPGAs contienen bloques de lógica cuya interconexión y funcionalidad se pueden programar. En la figura 2.1 se puede ver la arquitectura genérica de una FPGA.

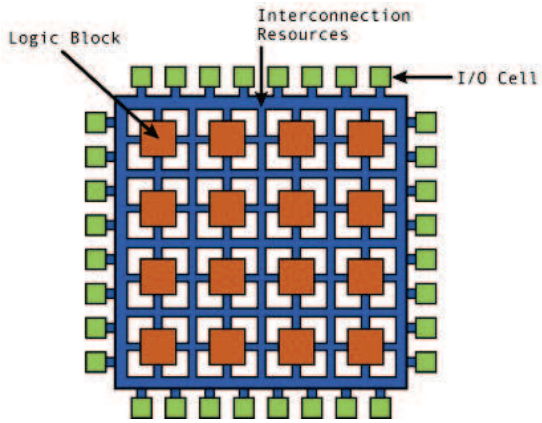



Figura 2.1: Arquitectura genérica de una FPGA [9]

Esta jerarquía de interconexiones programables permite a los bloques lógicos de una FPGA (figura 2.2) ser interconectados según la necesidad del diseñador del sistema (figura 2.3).

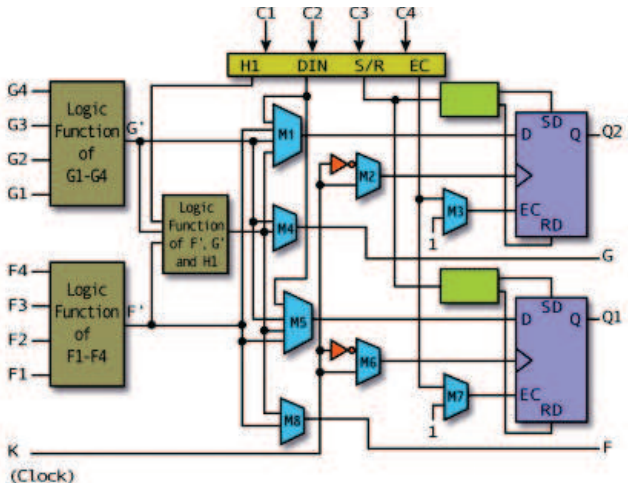



Figura 2.2: Bloque lógico configurable de una FPGA [9]

La FPGA puede desempeñar cualquier función lógica necesaria gracias a que los bloques lógicos e interconexiones pueden ser programados después del proceso de manufactura por el usuario/diseñador. Además, las entradas y salidas de una FPGA son totalmente configurables, como se puede ver en

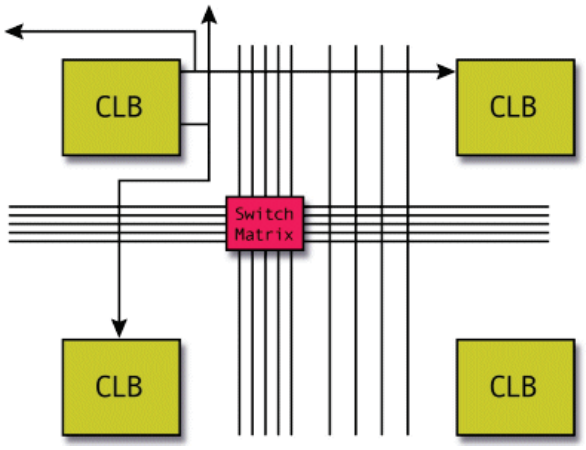



Figura 2.3: Interconexión programable de una FPGA [9]

la figura 2.4.

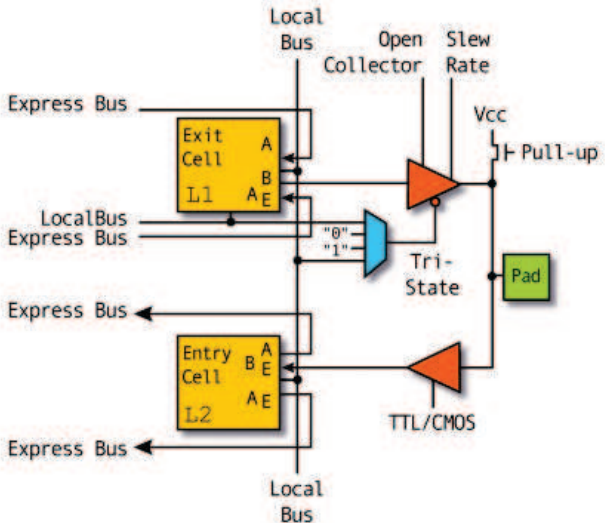



Figura 2.4: Bloque E/S configurable de una FPGA [9]

Muchas FPGA modernas soportan la reconfiguración parcial del sistema, permitiendo que una parte del diseño sea reprogramada, mientras las demás partes siguen funcionando. Este es el principio de la idea de la computación reconfigurable.

La computación reconfigurable (RC) está definida como un computador

que contiene hardware que se puede reconfigurar para implementar funciones específicas. Los sistemas RC combinan microprocesadores y dispositivos lógicos programables en un solo sistema. Un tipo de dispositivo lógico programable que se usa en los computadores reconfigurables es la FPGA. Como se puede ver en la figura 2.5, donde la FPGA sería el *Reconfigurable Element*.

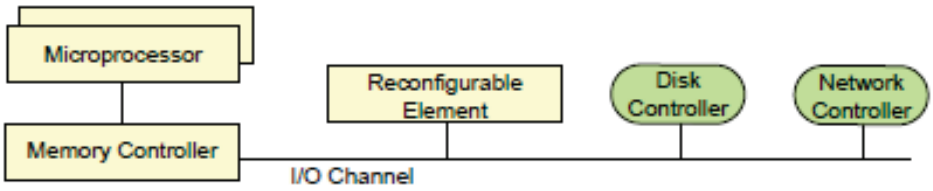



Figura 2.5: Computador Reconfigurable

Algunos ejemplos de aplicaciones en las que se puede usar una FPGA son: el procesamiento de señal digital, imágenes médicas, visión por computador, reconocimiento de voz, criptografía y bioinformática. En general, cualquier aplicación donde se haga uso del paralelismo que las FPGAs ofrecen gracias a su arquitectura. Una de ellas es el desciframiento de códigos, por fuerza bruta, de los algoritmos criptográficos [7].

Para poder programar una FPGA es necesario traducir manualmente el algoritmo que se quiere ejecutar a un lenguaje de programación especial llamado HDL (en inglés Hardware Description Language), que permite documentar las interconexiones y el comportamiento de un circuito electrónico. Esto no es una tarea sencilla ya que los programadores de software típicamente escriben programas secuenciales que explotan la habilidad de los microprocesadores de ir a través de una serie de instrucciones. En cambio, un diseño de calidad para una FPGA requiere pensar sobre el paralelismo es-

pacial, es decir, usar simultáneamente múltiples recursos repartidos por todo el chip para que el programa rinda bien computacionalmente.

Entre los lenguajes HDLs más utilizados destacan VHDL, Verilog y ABEL [4]. Aunque en este proyecto se ha elegido el lenguaje Verilog, existen compiladores que traducen el código en lenguaje Verilog a código en lenguaje VHDL, por lo tanto los lenguajes Verilog y VHDL son equivalentes.

El algoritmo en lenguaje HDL debe contener todas las interconexiones y registros necesarios para programar la FPGA, tales como: los registros para las variables de entrada y salida del algoritmo (con el tamaño de la variable y dirección concreta de memoria), los registros necesarios para los datos que se leen y se escriben de memoria, registros con las direcciones de memoria de lectura y escritura, etc.

Además el algoritmo en lenguaje HDL debe contener el código para leer y escribir en memoria correctamente (especificando las instrucciones que debe realizar el algoritmo en cada uno de los ciclos del reloj), así como la operación (u operaciones) que realiza el algoritmo original traducida a lenguaje HDL.

## 2.1. Arquitectura de una FPGA

Para realizar un buen programa en una FPGA es imprescindible conocer los entresijos de su funcionamiento. En esta sección del documento se van a explicar los componentes hardware principales típicos de una FPGA.

Una FPGA juega el mismo rol en una plataforma de computación reconfigurable que una CPU en un computador.

En términos generales, hay solo dos tipos de recursos en una FPGA: lógi-

cos y de interconexión. Los lógicos son donde se hacen las cosas aritméticas, 1+1=2, y las funciones lógicas, if (cierto) x=1 else x=0. Los de interconexión es como obtenemos los datos (como los resultados del anterior cálculo) desde un nodo de cálculo a otro [22].

Cualquier cálculo puede ser representado como una ecuación Booleana, y además, cualquier ecuación Booleana puede ser expresada como una tabla de verdad. Con estos métodos se pueden construir estructuras más complejas que pueden hacer cálculos aritméticos, como sumadores y multiplicadores, además de estructuras de toma de decisión que pueden evaluar condiciones, como un if-then-else. Combinando esto, se puede decir que se puede expresar cualquier algoritmo usando tablas de verdad.

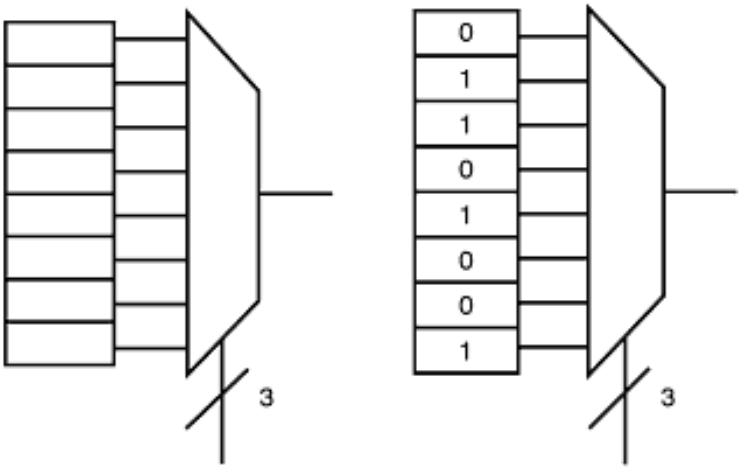



Figura 2.6: 3-LUT (izquierda) y la tabla de verdad de una XOR (derecha)

Desde un punto de vista sencillo, se puede ver la tabla de verdad como el corazón de cálculo de una FPGA. Más concretamente, un elemento hardware que puede implementar fácilmente una tabla de verdad es la tabla de búsqueda (del inglés *lookup table*) o LUT. De manera sencilla, una LUT enumera

una tabla de verdad. Por lo tanto, usar LUTs da a las FPGAs la libertad de poder implementar cualquier función lógica. La figura 2.6 muestra una LUT como las que se pueden encontrar hoy en día en las FPGAs.

Una LUT puede implementar cualquier función de N entradas simplemente programando la tabla de búsqueda con la tabla de verdad de la función que se quiere implementar.

Por lo tanto, la mayoría de FPGAs que se venden hoy en día tienen LUTs como su bloque lógico básico.

Además de las LUTs, es necesario un elemento para mantener el estado y permitir la implementación secuencial. Para ello, se añade un elemento donde guardar un bit en el bloque lógico, llamado D flip-flop. En la figura 2.7 se muestra como queda el bloque lógico. El multiplexor de salida elige entre el resultado generado por la LUT o el guardado en el D flip-flop.

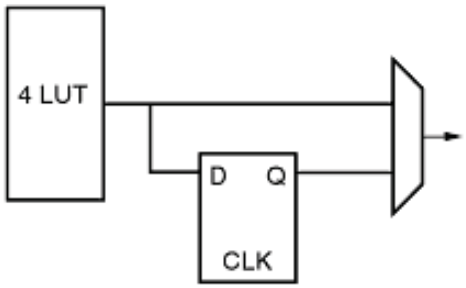



Figura 2.7: Bloque lógico

Ahora que se ha explicado el bloque lógico, lo siguiente es entender como estos bloques lógicos se conectan entre ellos.

La mayoría de FPGAs actuales implementan la arquitectura llamada estilo-isla. Como se puede ver en la figura 2.8 los bloques lógicos se colocan en un vector de dos dimensiones (matriz) y se interconectan entre ellos. Los

bloques lógicos forman las islas que flotan en el mar de las interconexiones.

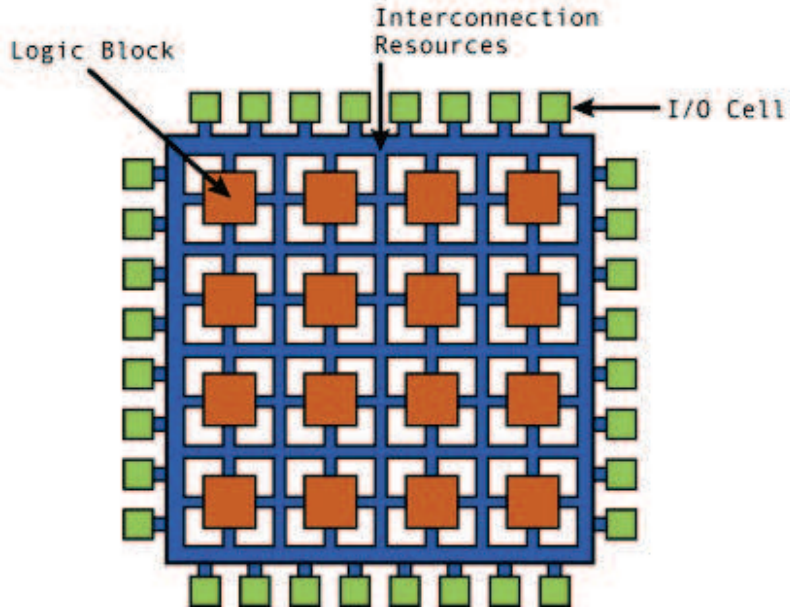



Figura 2.8: La arquitectura en forma de isla de una FPGA

Con esta arquitectura de matriz, los cálculos se realizan espacialmente en la estructura de la FPGA. Los cálculos grandes se dividen en cálculos más pequeños de tamaño 4-LUTs y son colocados en los bloques lógicos de la matriz. La interconexión se configura para fijar el camino de las señales entre los bloques lógicos de forma correcta. Con bloques lógicos suficientes, se puede hacer que una FPGA haga cualquier tipo de cálculo.

Además de estos elementos, las FPGAs necesitan memorizar los datos en algún tipo de memoria. Según el tipo de memorias que utilicen las FPGAs se pueden dividir en dos diferentes grupos [7].

Por un lado, las memorias pueden ser volátiles, basadas en RAM. En este tipo de memorias la programación se pierde al quitar la alimentación

y además requieren una memoria externa no volátil para configurarlas al arrancar (antes o durante el reset).

Por otro lado, las memorias pueden ser no volátiles, basadas en ROM. Hay dos tipos de memorias no volátiles, las reprogramables y las no reprogramables.

Las reprogramables, están basadas en memorias EPROM o flash. Estas memorias se pueden borrar y volver a reprogramar, pero con un límite de unos 10.000 ciclos.

Las no reprogramables, basadas en fusibles. Este tipo de memorias solo se pueden programar una vez, lo que las hace poco recomendables para trabajos en laboratorio.

## 2.2. Las FPGAs y los supercomputadores

Hoy en día la mayoría de FPGAs se usan en supercomputadores. Es posible que en un futuro no muy lejano en todos los ordenadores se utilice una (o más de una) FPGA, como hoy en día que prácticamente todos los ordenadores disponen de una GPU para realizar los cálculos gráficos, pero por el momento se ha desarrollado muy poco sobre el tema de FPGAs y habrá que esperar para su comercialización en ordenadores de sobremesa.

Un ejemplo de supercomputador que utiliza FPGAs es el JANUS, un supercomputador que se inauguró en mayo del 2008 y que está en la Universidad de Zaragoza [16]. En la figura 2.9 se puede ver el exterior de Janus, un armario de unos 2 metros y medio de altura [17].

Figura 2.9: Exterior del supercomputador Janus

En su interior, Janus dispone de 16 placas [18]. En la figura 2.10 se puede observar una de estas placas.

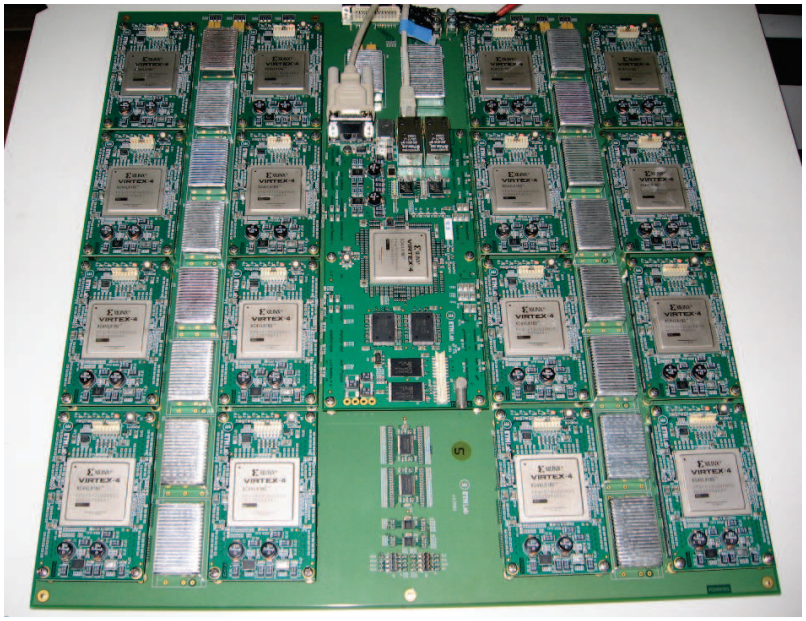


Figura 2.10: Una placa del supercomputador Janus

Cada una de estas placas está compuesta a su vez por 16 FPGAs. Como se puede ver la figura 2.11, las FPGAs de Janus son Xilinx Virtex-4, exactamente el mismo modelo de FPGA que se ha usado en este proyecto.

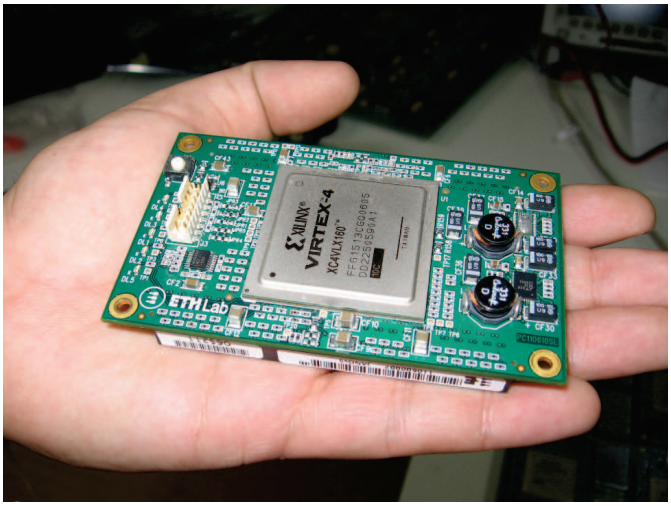



Figura 2.11: Una FPGA del supercomputador Janus

Además, Janus dispone de 256 procesadores para hacer simulaciones. Su potencia en los problemas para los que ha sido diseñado es el equivalente a unas 90 billones de operaciones por segundo, es decir, el equivalente a unos 90.000 procesadores convencionales.

El consumo de corriente de Janus es en proporción muy bajo: un máximo de 15 kilowatios cuando trabaja a máxima potencia. Esto es gracias a una de las ventajas de las FPGAs, su bajo consumo energético. Gracias a esto, Janus es uno de los ordenadores verdes más eficientes, realizando 8750 millones de operaciones por watio consumido.

Gracias a todo esto, Janus ha podido realizar un cálculo intensivo de unos de los problemas abiertos más difíciles de resolver: el estudio de los vidrios de spin; en concreto, del llamado modelo de Edwards-Anderson. Para obtener resultados fiables que aclaren algunas cuestiones en discusión en la comunidad científica, harían falta del orden de 100 años de trabajo en ordenadores convencionales.

## 2.3. Entorno de trabajo

Para realizar este proyecto ha sido necesario utilizar tres máquinas: un supercomputador Altix que contiene una FPGA en uno de sus blades (al que se llamará Altix a partir de ahora), un ordenador personal con mucha memoria (8Gb) donde se compila el código (al que se llamará a partir de ahora FPGApc) y mi ordenador personal.

### 2.3.1. Altix

La máquina Altix es un supercomputador que contiene una FPGA y es donde se ejecuta el código final obtenido de una compilación anterior en otro ordenador. Además, también se compila (si es necesario) el código C que se utiliza para ejecutar el algoritmo en la FPGA.

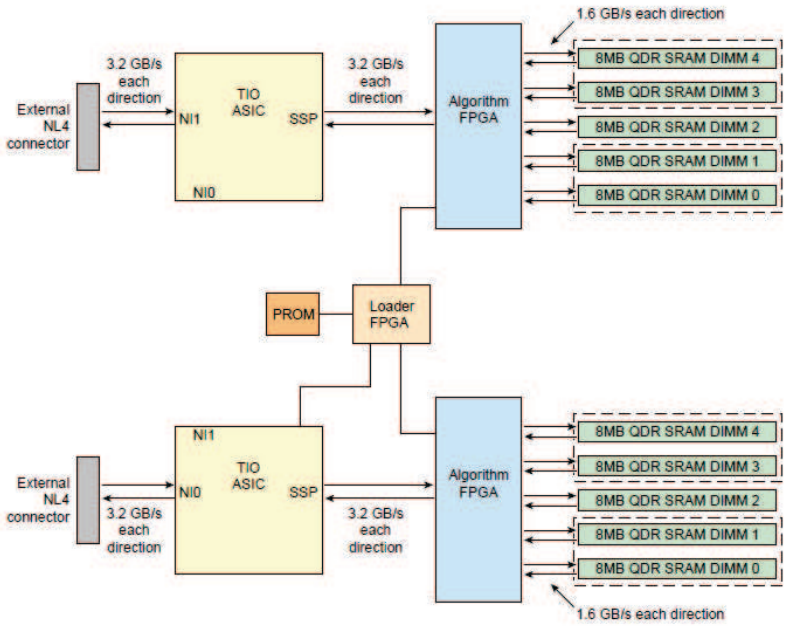



Figura 2.12: Hardware de un Blade de una FPGA [1]

La FPGA que se ha usado para este proyecto pertenece al BSC (Barcelona Supercomputing Center) [10].

El modelo concreto de la FPGA con la que se ha trabajado es una SGI Altix 4700. En las figuras 2.12 y 2.13 se puede ver el hardware de este modelo concreto de FPGA.

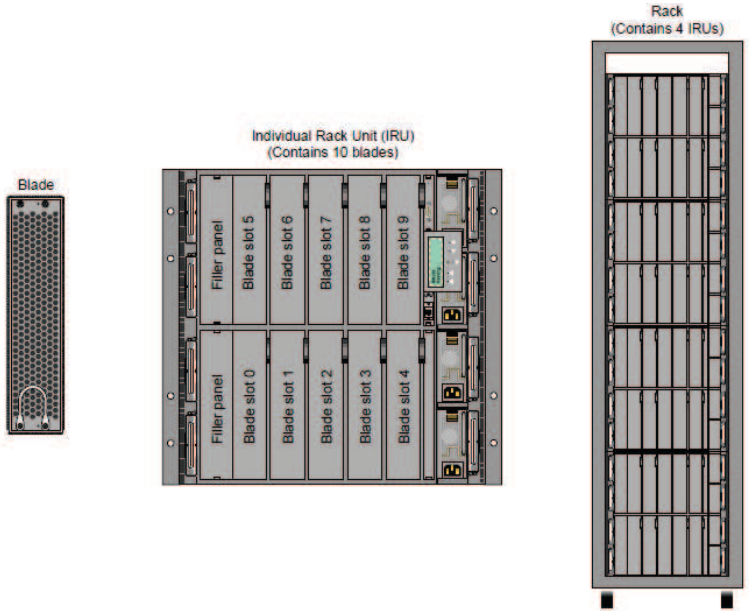



Figura 2.13: Altix 4700 Blade, Individual Rack Unit, y Rack [1]

La SGI Altix es una máquina con memoria compartida, con una arquitectura cc-NUMA (Cache Coherent Non-Uniform Memory Access). La configuración de su hardware es la siguiente [10]:

- 128 CPUs Dual Core Montecito (IA-64). Cada uno de estos 256 cores trabaja a 1,6 GHz.
- 8 MB L3 caché y 533 MHz Bus.
- 2,5 TB RAM.
- Rendimiento máximo: 819,2 Gflops.

- 2 discos SAS internos de 146 GB a 15000 RPMs.
- 12 discos SAS externos de 300 GB a 10000 RPMs.

En la Altix, aparte de ejecutar el código, se compila el código C, y para ello se usa el compilador gcc con la siguiente versión: 4.1.2 20070115 (SUSE Linux).

### 2.3.2. FPGApc

El FPGApc se ha utilizado en este proyecto para compilar el código Verilog que luego se ejecuta en la FPGA. Para realizar esta tarea se necesita un ordenador potente ya que el proceso es muy costoso.

Para ello se ha usado un ordenador perteneciente al laboratorio AC de la Universidad Politécnica de Cataluña (UPC) [11].

La configuración hardware del FPGApc es la siguiente:

- 2 CPUs Intel Core 2 6600. Cada uno de estos cores trabaja a 2,4 GHz.
- 8 MB caché.
- 8 GB RAM.
- 128 GB disco duro.

Este ordenador tiene un compilador de FPGA instalado llamado xst, que es el compilador integrado en la ISE (del inglés *Integrated Software Enviroment*) para FPGAs de Xilinx. La versión de el compilador xst que dispone el FPGApc es la 10.1.03.

### 2.3.3. Ordenador personal

En mi ordenador personal se ha trabajado en la creación de la herramienta automática en Python.

La configuración hardware de mi ordenador personal es la siguiente:

• CPU Intel Core 2 Duo T5500. Cada uno de estos cores trabaja a 1,66 GHz.

• 4 MB caché.

• 2 GB RAM.

• 120 GB disco duro.

Para realizar la herramienta automática se ha usado el programa Eclipse [12], en su versión 3.4.2. Para poder trabajar con Python en Eclipse, ha sido necesario instalar Pydev (el entorno de trabajo de Python dentro de Eclipse) [13]. La versión concreta de Pydev que se ha usado es la 1.4.6.2788. En la figura 2.14 se puede observar el programa Eclipse funcionando con el complemento Pydev.

Además, para mantener un control de versiones del proyecto se ha usado un complemento que añade Subversion al Eclipse, llamado Subclipse [14]. La versión en concreto que se ha instalado para este proyecto es la 1.4.8. En la figura 2.15 se puede ver el programa Eclipse funcionando con el complemento Subclipse. Las diferentes versiones se han subido al repositorio que la Facultad de Informática de Barcelona (FIB) [15] reserva para cada estudiante: "https://svn.fib.upc.es/svn/nombreUsuario".

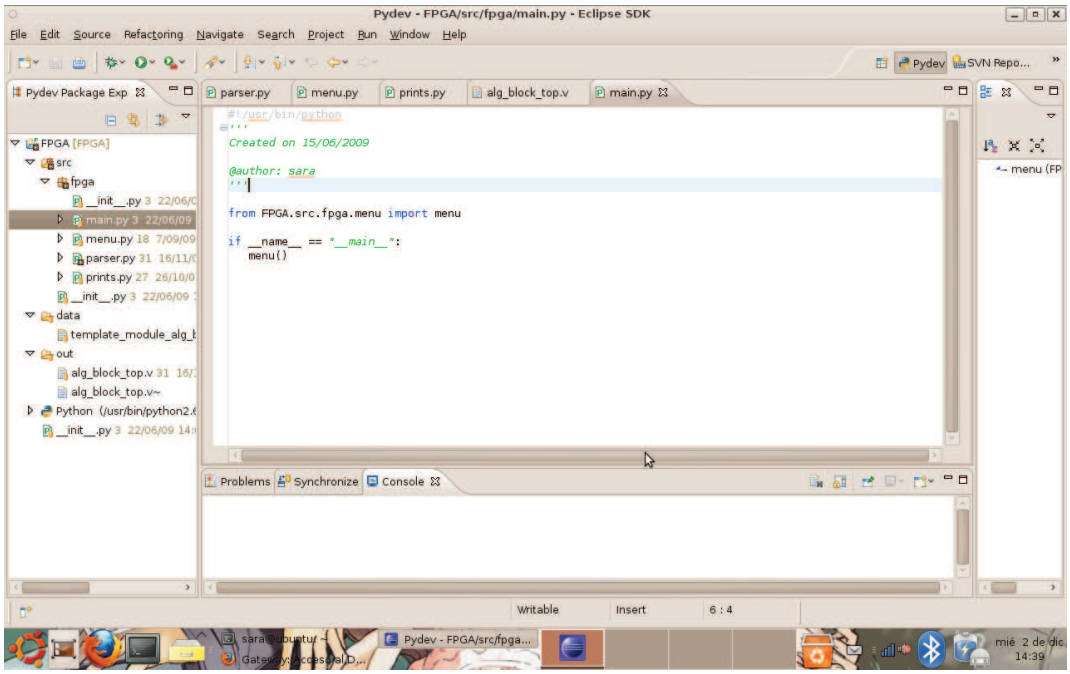



Figura 2.14: Eclipse con el complemento para Python

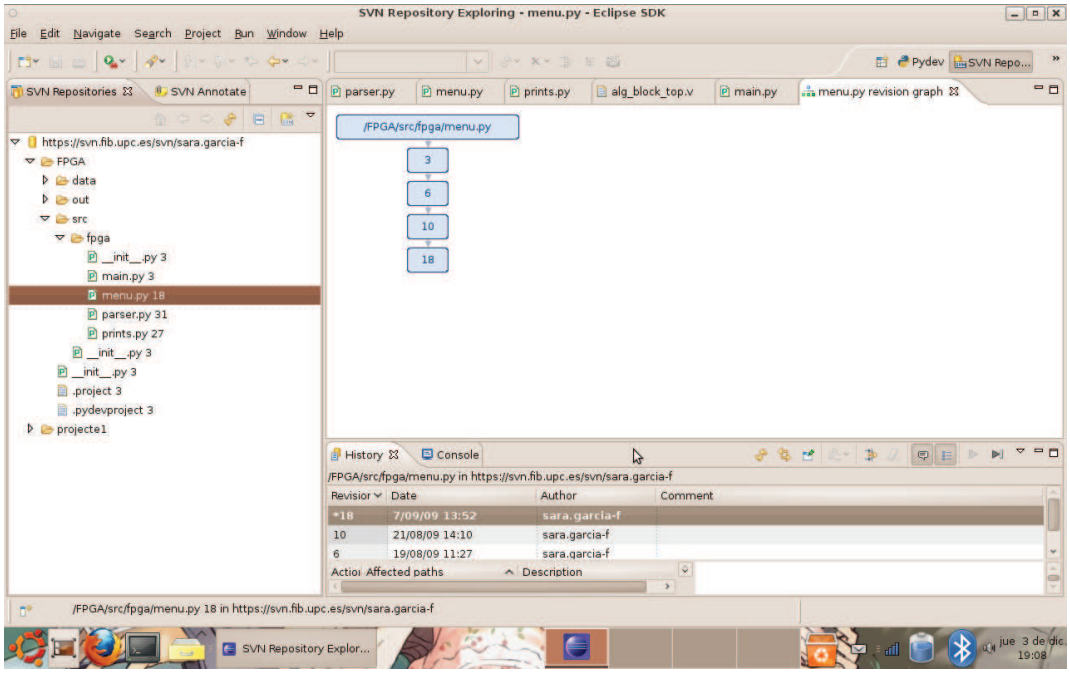



Figura 2.15: Eclipse con el complemento para Subversion

# Capítulo 3

# Desarrollo

En este capítulo se van a explicar los pasos que se han seguido para llevar a cabo este proyecto.

## 3.1. Funcionamiento de la FPGA

Para empezar a realizar el proyecto, fue necesario entender el funcionamiento de la FPGA y los elementos que necesita para funcionar correctamente.

Como ya se ha explicado anteriormente, para ejecutar un algoritmo en la FPGA es necesario programar un código C y un código Verilog. El fichero en código Verilog contiene mucha de la información que necesita la FPGA para funcionar, como las señales, los registros y el código que se debe ejecutar en la FPGA.

En primer lugar se compila el código Verilog (junto con más archivos que ya vienen preparados y que no hay que modificar) y se obtienen tres archivos: dos archivos de configuración y un archivo binario. En la figura 3.1 se puede

ver el funcionamiento de la implementación de una FPGA. Este proceso se lleva a cabo en el FPGApc.

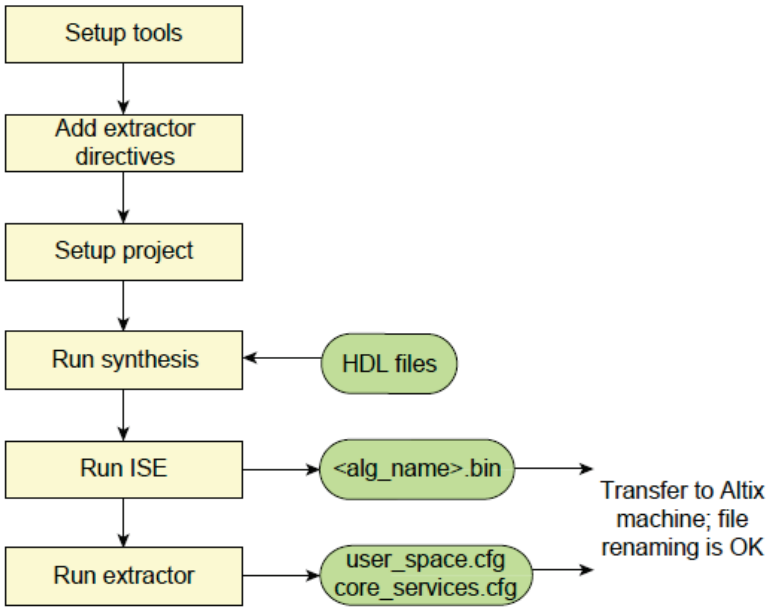



Figura 3.1: Flujo de implementación de una FPGA [1]

Una vez que se han obtenido estos tres archivos, primero se han de transferir al ordenador que contiene la FPGA y después cargar en la misma. A partir de este momento la FPGA ya está lista para ejecutar el algoritmo.

Por último, se compila (si es necesario) y ejecuta el código C en el sistema que contiene la FPGA. Este código en C es el encargado de transferir los datos hacia y desde la FPGA y de iniciar y terminar el funcionamiento de ésta.

## 3.2. Entender el código C

El código C que se usa para ejecutar en la FPGA tiene unas características especiales, ya que para comunicarse con la FPGA es necesario usar unas funciones propias de la librería de la FPGA (llamada RASClib).

En el código que se puede ver en la figura 3.2 se muestra un ejemplo sencillo de código C en el que, para simplificar, se ha omitido el control de errores.

```
unsigned long A[SIZE],B[SIZE],C[SIZE];
main() {

    int al_desc,
    int num_devices = N;
    int res;

    res = rasclib_resource_reserve(num_devices, NULL);

    res = rasclib_resource_configure("compute", num_devices, NULL);

    al_desc = rasclib_algorithm_open("compute");

    rasclib_algorithm_send(al_desc, "input_one", A,
      SIZE *sizeof(unsigned long));
    rasclib_algorithm_send(al_desc, "input_two", B,
      SIZE *sizeof(unsigned long));

    rasclib_algorithm_go(al_desc);

    rasclib_algorithm_receive(al_desc, "output", C,
      SIZE *sizeof(unsigned long));

    rasclib_algorithm_commit(al_desc);

    rasclib_algorithm_wait(al_desc);

    rasclib_resource_return("compute", num_devices);
    rasclib_resource_release(num_devices, NULL);
}
```

Figura 3.2: Ejemplo de código C

La descripción de las funciones más importantes de la librería RASClib es la siguiente [1]:

**rasclib_resource_reserve:** Envia una petición al administrador de dispositivos para reservar un número de dispositivos.

**rasclib_resource_configure:** Carga el bitstream en el dispositivo y lo marca como dispositivo en uso.

**rasclib_algorithm_open:** Notifica a la librería RASCLIB que el usuario quiere usar todos los dispositivos reservados con el bitstream asociado como un único dispositivo lógico.

**rasclib_algorithm_send:** Encola un comando que mueve datos desde la memoria del host a la entrada del dispositivo identificado con el nombre dado en el segundo parámetro.

**rasclib_algorithm_go:** Encola un comando para empezar la computación.

**rasclib_algorithm_receive:** Encola un comando que mueve datos desde la salida del dispositivo hasta la memoria del host.

**rasclib_algorithm_commit:** Provoca a la librería RASCLIB enviar todos los comandos encolados al driver del kernel para la ejecución en el dispositivo.

**rasclib_algorithm_wait:** Bloquea la ejecución (espera) hasta que todos los comandos que se han enviado al dispositivo han completado su ejecución.

**rasclib_resource_return:** Devuelve un dispositivo configurado a su fondo de reserva.

**rasclib_resource_release:** Mueve los dispositivos del fondo de reserva al fondo libre.

Las dos funciones más interesantes son **rasclib_algorithm_send** y **ras-**

**clib_algorithm_receive**, que son las encargadas de enviar y recibir los datos respectivamente.

Como se puede ver, en el código C no hay información sobre que operaciones se van a realizar en el algoritmo, pero ese es un problema que se solucionará más adelante. Por tanto, se puede decir que el código C usado en los algoritmos para FPGAs es bastante genérico. En concreto, este código C serviría para todos los algoritmos con 2 entradas y 1 salida (con variables del tipo unsigned long[SIZE]).

## 3.3. Entender el código Verilog

Como se ha comentado anteriormente en esta memoria, Verilog es un lenguaje HDL que se usa para programar la FPGA.

Para entender mejor el funcionamiento de este lenguaje, se empezó por entender un ejemplo sencillo como el que se muestra en el apéndice A.

Lo primero que se puede ver en el código, son los denominados "pragmas"(o extractors). Estos pragmas sirven para aportar datos al algoritmo en código Verilog.

En el primer ejemplo se pueden ver los siguientes extractors:

```
// extractor VERSION: 6.3
// extractor CS: 2.2
// extractor SRAM:a_in  2048  64 sram[0] 0x0000 in  u stream
// extractor SRAM:b_in  2048  64 sram[0] 0x4000 in  u stream
// extractor SRAM:c_in  2048  64 sram[0] 0x8000 in  u stream
// extractor SRAM:d_out 2048  64 sram[1] 0x0000 out u stream
//
// extractor REG_IN:op_length1 10 u alg_def_reg[0][9:0]
```

La explicación de los extractors más usados [1]:

- Extractor VERSION: Para identificar la versión del algoritmo.
- Extractor CS: Para identificar la versión del Core Services.
- Extractor SRAM: Vectores de entrada y salida del algoritmo.
- Extractor REG_IN: Registros de entrada usados en el algoritmo.
- Extractor REG_OUT: Registros de salida usados en el algoritmo.
- Extractor STREAM_IN: Etiqueta de stream de entrada.
- Extractor STREAM_OUT: Etiqueta de stream de salida.

Por tanto este ejemplo consta de tres variables de entrada y una de salida, las cuatro con 2048 elementos de 64 bits cada uno.

Una vez vistos los pragmas del ejemplo, el código sigue con la declaración del módulo, en este caso llamado alg_block_top. El módulo consta de tres partes; la definición del módulo y sus señales, la descripción de la dirección del puerto que van a usar las señales, y la inicialización de las señales. En esta parte del código se declaran tanto las señales como los registros. La diferencia entre las señales y los registros, es que las señales se pueden consultar desde fuera del programa y los registros se pueden consultar solo desde el propio programa Verilog.

De esta parte del código, lo más interesante es la parte donde se inicializa el acceso a memoria:

```
assign mem_0_rd_addr = {alg_mem_0_offset[23:16],req_abc,rd_index,4'b0};
assign mem_1_wr_addr = {alg_mem_1_offset[23:14],        wr_index,4'b0};
```

La primera línea significa que el algoritmo lee de una dirección memoria

que depende de la variable que quiere leer (req_abc), del índice de lectura (rd_index) y que además, leemos de 16 en 16 (4'b0) ya que cada vez que se lee de memoria se leen 128 bits.

La segunda línea, al tener solo un elemento de escritura, la dirección se calcula únicamente teniendo en cuenta el índice de escritura (wr_index).

Después de las declaraciones e inicializaciones se pasa a la parte del código propiamente dicho. En este ejemplo aparecen cuatro bucles, como se especifica a continuación:

• Bucle 1: Inicializa la longitud de las operaciones. Los parámetros pueden variar según el algoritmo.

```
always @(posedge clk)
  begin
      op_length_bound <= rst ? 10'h3ff : alg_def_reg0[9:0];
  end
```

• Bucle 2: Determina el elemento que se ha de leer de memoria.

```
always @ (posedge clk)
    begin: read_mem
     // reset memory read then start reading
     if (rst)
      begin
       rd_cmd_vld <= 1'b0;
       rd_index   <= 10'h00;
       req_abc    <= 2'b0;
       read_done  <= 1'b0;
      end // if (rst)
     else
      begin
       if (!read_done)
        begin
         rd_cmd_vld <= 1'b1;
         if (rd_cmd_vld)
          begin
```

```
          if (req_abc == 2'b01 && rd_index == op_length_bound)
           begin
            read_done <= 1'b1; // Next read is last one
           end
          if (req_abc == 2'b10)
           begin
            req_abc <= 2'b00;
            rd_index <= rd_index + 10'b1;
           end
          else
           begin
            req_abc <= req_abc + 1'b1;
           end
         end
       end // if (!read_done)
      else
       begin
        rd_cmd_vld <= 1'b0;
       end // else: !if(!read_done)
     end
   end // block: read_mem
```

Lo más importante es que en este bucle se actualiza el valor del registro que indica qué variable hay que leer (req_abc) y además se incrementa el índice de lectura (rd_index).

• Bucle 3: Lee un elemento de memoria y lo guarda en un registro.

```
always @ (posedge clk)
    begin:flop_read_data

     rd_data <= mem_0_rd_data;
     if (rst)
      begin
       rd_dvld   <= 1'b0;
       read_abc  <= 2'b0;
      end // if (rst)
     else
      begin
       rd_dvld   <= mem_0_rd_data_vld;
       if (rd_dvld)
        begin
         if (read_abc == 2'b10)
          begin
```

```
          read_abc <= 2'b00;
         end
        else
         begin
          read_abc <= read_abc + 2'b01;
         end
        if (read_abc == 2'b00)
          temp_a <= rd_data;
        if (read_abc == 2'b01)
          temp_b <= rd_data;
       end
     end // else: !if(rst)
   end // block: flop_read_data
```

En este bucle, usando el valor de req_abc, se guardan los valores leídos de memoria en registros temporales (temp_a y temp_b en este caso). El valor de la tercera variable no es necesario guardarlo en un registro ya que se consulta directamente de rd_data.

• Bucle 4: Calcula las operaciones que se deben realizar sobre los elementos leídos y escribe el resultado.

```
always @ (posedge clk)
    begin: write_result_mem

      // flop write data prior to sending to ram
      wr_data <= (temp_a & temp_b) | rd_data;

      // reset memory write
      if (rst)
       begin
        wr_index <= 10'h0;
        wr_dvld  <= 1'b0;
        alg_done <= 1'b0;
       end // if (rst == 1'b1)
      else
       begin
        // when C is vaild enable sram write
        wr_dvld <= rd_dvld & (read_abc == 2'b10);

        // increment index after write
        if (wr_dvld)
```

```
          wr_index <= wr_index + 10'h001;

        // all done assert alg_done
        done <= (wr_index == op_length_bound);
        alg_done <= done & wr_dvld;
       end // else: !if(rst)

    end // block: write_result_mem
```

En este bucle se realiza el cálculo de la operación con los valores leídos en los bucles anteriores y se actualiza el índice de escritura (sólo si la escritura se ha realizado con éxito). Hay que tener en cuenta que se leen de memoria 128 bits y los elementos son de 64 bits, esto significa que estamos calculando dos elementos en cada operación.

En resumen, en este ejemplo se leen tres variables y se calcula la siguiente operación:

```
D = (A & B) | C
```

Es importante hacer notar que en este proyecto no nos vamos a preocupar de la operación o algoritmo que se implementa en el código tratado. Se trata, en cambio, de automatizar la gestión de la memoria para dicho algoritmo de forma que un programador de verilog pueda incluir automáticamente su código en una máquina Altix sin tener que reprogramarlo.

## 3.4. Automatización

Una vez entendidos tanto el código C como el código Verilog, se empezó a automatizar el proceso de creación del código Verilog.

Una de las primeras cosas que se pudo ver, como ya se ha comentado anteriormente, es que el código C no contiene todos los datos que son necesarios en el código Verilog. Por ello es necesario incorporar manualmente al código C los elementos que faltan. Estos elementos se han añadido como comentarios en forma de pragma al código C.

Los pragmas que se han utilizado en este proyecto siguen el patrón de OpenMP [19] (Open Multi-Proccessing), una API que permite añadir concurrencia a las aplicaciones mediante paralelismo con memoria compartida [20].

Los pragmas de OpenMP siguen el siguiente patrón:

```
# pragma omp <directiva> [cláusula [ , ...] ...]
```

En un reciente artículo [21] se propone una extensión de los pragmas de OpenMP para arquitecturas heterogéneas, como el caso de la FPGA. En este artículo se propone un nuevo pragma para especificar a qué dispositivo se está dirigiendo. Este nuevo pragma es de la forma siguiente:

```
#pragma omp target device (nombre-disp.) [cláusula [ , ...] ...]
```

Este artículo propone, además, unas cláusulas opcionales para usar junto con el pragma device para especificar las variables que se van a usar:

```
copy_in (lista-variables)
copy_out (lista-variables)
```

Así, en este proyecto se han utilizado unos pragmas que siguen el patrón OpenMP y que además utilizan las extensiones propuestas en el artículo mencionado anteriormente.

Por tanto, los pragmas que se han utilizado en este proyecto son los siguientes:

• Pragma 1:

```
#pragma omp target device (FPGA) implements (<nombre_algoritmo>)
 copy_in (<var_entrada_1, var_entrada_2, ... , var_entrada_n>)
 copy_out (<var_salida_1, var_salida_2, ... , var_salida_n>)
{
<operación_1>
<operación_2>
...
<operación_n>
}
```

Este pragma sirve para indicar las variables de entrada y salida del algoritmo y las operaciones que debe realizar.

• Pragma 2:

```
#pragma omp target device (FPGA) version (<version>)
```

Este pragma se usa para indicar la versión del algoritmo.

• Pragma 3:

```
#pragma omp target device (FPGA) latency (<latencia>)
```

Este pragma indica la latencia de las operaciones del algoritmo.

• Pragma 4:

```
#pragma omp target device (FPGA) temp (<var_temp_1, var_temp_2, ... ,
var_temp_n)
```

Este pragma se usa para especificar las variables temporales que se usan en las operaciones.

Para empezar el proceso de automatización, lo primero fue insertar automáticamente los extractors en el código Verilog.

Para ello se siguen los siguientes pasos:

1. Se inserta el pragma de la versión del Core Services. Normalmente se usará la versión 2.20, ya que es la última:

```
// extractor CS:2.2
```

2. Se inserta el pragma de la versión del algoritmo:

```
// extractor VERSION:1.0
```

3. Se buscan en el código C funciones de entrada de datos: rasclib_algorithm_send. Por cada una de ellas, se inserta un pragma SRAM de tipo in:

Código C:

```
res = rasclib_algorithm_send(algorithm_id, "a_in", A,
 sizeof(unsigned long long) * (128*2048));
res = rasclib_algorithm_send(algorithm_id, "b_in", B,
 sizeof(unsigned long long) * (128*2048));
res = rasclib_algorithm_send(algorithm_id, "c_in", C,
 sizeof(unsigned long long) * (128*2048));
```

Código Verilog:

```
// extractor SRAM:a_in 2048 64 sram[0] 0x0000 in u stream
// extractor SRAM:b_in 2048 64 sram[0] 0x4000 in u stream
// extractor SRAM:c_in 2048 64 sram[0] 0x8000 in u stream
```

Para simplificar, se leerá de la SRAM[0].

El nombre de la variable (A, B y C en este ejemplo), el número de elementos (2048) y el tamaño de cada elemento (64), se guardan en una estructura de datos para posteriores consultas.

4. Se buscan en el código C las funciones de salida de datos: rasclib_algorithm_receive. Por cada una de ellas, se inserta un pragma SRAM de tipo out:

Código C:

```
res = rasclib_algorithm_receive(algorithm_id, "d_out", D,
 sizeof(unsigned long long) * (128*2048));
```

Código Verilog:

```
// extractor SRAM:d_out 2048 64 sram[1] 0x0000 out u stream
```

Para simplificar, se escribirá en la SRAM[1].

El nombre de la variable (D en este ejemplo), el número de elementos (2048) y el tamaño de cada elemento (64), se guardan en una estructura de datos para posteriores consultas.

Para saber el número de elementos y el tamaño de cada uno, se utiliza el programa "objdump" que proporciona información sobre ficheros objeto

(ficheros con extensión .o). En concreto la opción "-t" imprime la tabla de símbolos. Un ejemplo de la salida del programa "objdump" con los ejemplos anteriores:

```
...
0000000000004000 O *COM* 0000000000000008 A
0000000000004000 O *COM* 0000000000000008 B
0000000000004000 O *COM* 0000000000000008 C
0000000000004000 O *COM* 0000000000000008 D
...
```

Donde la primera columna representa el tamaño total de las variables en bytes y la cuarta columna representa el tamaño de cada elemento también en bytes.

Después, para automatizar la declaración del módulo alg_bock_top hay que tener en cuenta que es todo prácticamente igual en todos los algoritmos, excepto el tamaño de algunas señales o registros, así que no se le dará más importancia a esta parte.

Para automatizar el código de los cuatro bucles, en este primer caso únicamente se tuvo en cuenta este primer ejemplo. Así, simplemente se tenía que conseguir que automáticamente se creara un código Verilog equivalente para este ejemplo y una vez que se había entendido el código Verilog esto fue una tarea sencilla.

Hasta este punto se tuvieron también en cuenta diferentes aspectos para que no solo se creara el código Verilog para este ejemplo en concreto, sino que además la operación que se realiza pudiera ser cualquiera de tipo bit a bit (ya que se realiza la operación para dos variables a la vez) siempre que las variables de entrada y salida tuvieran el mismo tamaño entre sí. Además,

también se tenía en cuenta que las variables pudieran tener cualquier número de elementos (en el primer ejemplo eran 2048 elementos) siempre que fueran todas las variables del mismo tamaño. Así, hasta este punto, se aceptaban tres variables de entrada y una variable de salida, las cuatro variables del tamaño indicado en el primer ejemplo (64 bits cada elemento de la variable), y que realizaran una sola línea de operaciones, es decir, cuyo cálculo se realizará en un ciclo de reloj de la FPGA.

## 3.5. Ampliaciones

Para que el programa fuera más genérico y tuviera en cuenta más tipos de algoritmos, fue necesario primero crear códigos Verilog con diferentes tipos de algoritmos para ver el funcionamiento correcto de los algoritmos y posteriormente poder automatizar el proceso.

La primera ampliación que se aplicó al programa fue que se pudieran tener un número de variables de entrada diferente de tres.

Para más de dos variables de entrada, el código Verilog no cambia significativamente en comparación con el caso de tres variables de entrada. Lo único que hay que tener en cuenta es que hay que añadir un extractor del tipo SRAM más por cada variable de entrada y guardar los elementos que se van leyendo de cada variable en un registro temporal, con lo que se necesitarán tantos registros temporales como variables de entrada menos una (ya que la última variable se consulta directamente sobre el registro que lee de memoria).

Para el caso de una sola variable de entrada, el código Verilog cambia un poco más. En este caso, la dirección de memoria de lectura, ya no es necesario calcularla dependiendo de la variable que se quiere leer (registro *req*) ya que solo hay una, y por tanto quedaría de la siguiente forma:

```
assign  mem_0_rd_addr = {alg_mem_0_offset[23:12], rd_index, 4'b0};
```

Por tanto, todas las apariciones del registro *req* desaparecen, simplificando considerablemente todo el código.

Además, al solo haber una variable, no es necesario tener registros temporales para guardar los elementos que se van leyendo.

En caso de no tener variables de entrada, el código se simplifica mucho, ya que no son necesarios los dos bucles que se usan para leer las variables.

La siguiente ampliación que se realizó fue para poder utilizar todo tipo de operaciones (y no solo operaciones bit a bit). Esta limitación se debe a que al leer siempre 128 bits, si las variables son de menos de este tamaño, no trata cada elemento por separado, sino que se realiza la operación para todos los elementos que haya leído en 128 bits.

En este caso, no se cambió nada en el código Verilog, no obstante, en el código C hay que transformar los elementos de las variables, sean del tamaño que sean, a elementos de 128 bits. De esta manera, se lee y escribe un solo elemento en cada lectura y escritura, pudiendo usar por tanto operaciones que no son de tipo bit a bit.

La siguiente ampliación que se incluyó al programa fue poder escribir varias líneas de operaciones en un mismo algoritmo. Esta ampliación va muy de la mano con otra que se realizó un poco más adelante para poder usar latencias, así que para simplificar se explican conjuntamente.

Hasta este punto de desarrollo del proyecto, se habían usado algoritmos con una sola línea de operaciones, con lo que la latencia era siempre 1, es decir, no había que esperar para escribir el resultado de la operación. Al permitir algoritmos con varias líneas de operaciones, lo más normal es que el cálculo de una operación dependa del resultado de una operación anterior. Por tanto, para que funcione todo correctamente, es necesario esperar algunos ciclos de reloj para que todo esté calculado y entonces poder escribir el resultado correcto.

En esta ampliación el cambio más importante que se produjo fue en el bucle donde se realizan las escrituras. Para explicar esta parte usaremos un algoritmo un poco más complejo que realiza varias operaciones con dependencia de datos entre ellas como las que se muestran en la figura 3.3.

```
temp_0 <= tempRD_0 + tempRD_1;

temp_1 <= tempRD_0 - rd_data;

temp_2 <= temp_0 + temp_1;

wr_data <= temp_2 + 10;
```

Figura 3.3: Ejemplo de código con dependencia de datos

Para realizar varias líneas de operaciones, hay que guardar las operaciones

intermedias en registros temporales (llamados *temp_n*), para así disponer de su valor en ciclos siguientes.

Por tanto, en el ejemplo anterior se realizan las siguientes operaciones:

```
temp_0 = A + B;
temp_1 = A - C;
temp_2 = temp_0 + temp_1;
D = temp_2 + 10;
```

Esto significa que hay que esperar tres ciclos desde que se han acabado de leer todas las variables hasta que D obtiene el resultado correcto de todas las operaciones anteriores, como se indica en la figura 3.4.

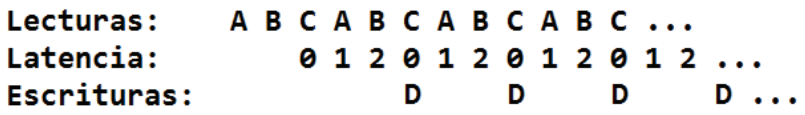



Figura 3.4: Ejemplo de un algoritmo con latencia 3

Comparando un algoritmo con latencia 3 con un algoritmo con latencia 1, como el de la figura 3.5, se puede observar que las escrituras del algoritmo con latencia 3 se realizan con tres ciclos de retraso con respecto al de latencia 1.

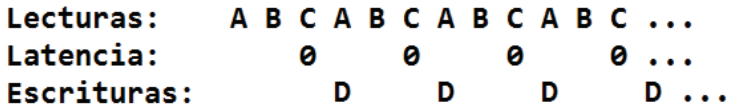



Figura 3.5: Ejemplo de un algoritmo con latencia 1

```
always @ (posedge clk)
  begin: write_result_mem
   temp_0 <= tempRD_0 + tempRD_1;
   temp_1 <= tempRD_0 - rd_data;
   temp_2 <= temp_0 + temp_1;
   wr_data <= temp_2 + 10;

   if (rst)
    begin
     wr_index <= 11'h0;
     wr_dvld  <= 1'b0;
     alg_done <= 1'b0;
     lat      <= 3'b0;
     wr_first <= 1'b1;
    end // if (rst == 1'b1)
   else
    begin
     wr_dvld <= ((lat == 3'b100) & (wr_first == 1'b1)) |
                    ((wr_first == 1'b0) & (lat == 3'b10));

     if (wr_first == 1'b1 & rd_dvld)
      begin
       if (lat == 3'b100)
        begin
         wr_first <= 1'b0;
         lat <= 3'b0;
        end
       else  lat <= lat +3'b1;
      end
     else if (wr_first == 1'b0)
      begin
       if (lat == 3'b10)  lat <= 3'b0;
       else               lat <= lat + 3'b1;
      end

     if (wr_dvld)  wr_index <= wr_index + 11'b1;

     done <= (wr_index == op_length_bound);
     alg_done <= done & wr_dvld;
    end // else: !if(rst)
  end // block: write_result_mem
```

Figura 3.6: Ejemplo de un algoritmo con latencia 3

Para simplificar, se puede decir que la primera escritura es la que se realiza con tres ciclos de retraso y que las demás se realizan como en el caso de latencia 1, cada tres ciclos.

Por tanto, fue necesario añadir dos nuevas variables para controlar este tiempo de espera; una para contar la espera entre escritura y escritura (llamada *lat*) y otra para saber si estamos en la primera escritura (llamada *wr_first*). Así, el bucle donde se realizan las escrituras para el ejemplo de latencia 3 quedaría como se muestra en la figura 3.6.

La última ampliación que se incluyó al programa fue la posibilidad de utilizar diversas variables de salida. Esta ampliación resultó ser la más compleja ya que requiere diversos cambios en el código Verilog.

Primeramente, en los pragmas o extractors del código Verilog hay que añadir un pragma del tipo SRAM por cada variable de salida, exactamente igual que con las variables de entrada. Para un algoritmo con tres variables de escritura (o salida) los extractors quedarían de la forma siguiente:

```
// extractor SRAM:a_in  2048  128 sram[0] 0x0000 in  u stream
// extractor SRAM:b_in  2048  128 sram[0] 0x8000 in  u stream
// extractor SRAM:c_in  2048  128 sram[0] 0x10000 in  u stream
// extractor SRAM:d_out 2048  128 sram[1] 0x0000 out u stream
// extractor SRAM:e_out 2048  128 sram[1] 0x8000 out u stream
// extractor SRAM:f_out 2048  128 sram[1] 0x10000 out u stream
```

Además, fue necesario crear un nuevo registro para saber que variable se está escribiendo en memoria en cada momento (llamado *req_wr*). Por tanto el cálculo de la dirección de memoria de escritura se calcula de la forma siguiente:

```
assign  mem_1_wr_addr = {alg_mem_1_offset[23:14], req_wr, wr_index, 4'b0};
```

En este caso fue necesario modificar sobretodo el bucle donde se realizan las escrituras. Para entender esta parte del código, primero es importante entender que, debido a que solo se puede escribir una variable en cada ciclo, hay que guardar las variables de salida en registros temporales (llamados *tempWR_n*)para poder escribirlas una en cada ciclo, como se muestra en la figura 3.7.

```
Lecturas:    A B C A B C A B C A B C ...
Latencia:        0 1 2 0 1 2 0 1 2 0 1 2 ...
Escrituras:            D E F D E F D E F D E F ...
```

Figura 3.7: Algoritmo con latencia 3 y tres variables de salida

Por tanto, cada variable tiene que esperar a que se haya escrito la anterior para poder ser escrita, y se debe guardar el valor de la variable en cada ciclo hasta que llega el ciclo en el que se escribe esa variable. Además es necesario un registro nuevo para saber que variable se va escribir (llamado *write*). El código de las operaciones queda de la siguiente forma:

```
temp_0 <= tempRD_0 + tempRD_1;

temp_1 <= tempRD_0 - rd_data;

temp_2 <= temp_0 + temp_1;

wr_data <= temp_2 + 10;

tempWR_0 <= temp_2 + 20;

tempWR_1 <= temp_2 + 30;
```

```
if (write == 2'b1)
  begin
    wr_data <= tempWR_0;
    tempWR_1 <= tempWR_1;
  end
if (write == 2'b10)
  begin
    wr_data <= tempWR_1;
  end
```

Las operaciones que se realizan en este ejemplo son las siguientes:

```
temp_0 = A + B;

temp_1 = A - C;

temp_2 = temp_0 + temp_1;

D = temp_2 + 10;

E = temp_2 + 20;

F = temp_2 + 30;
```

Es necesario por tanto, actualizar el valor del registros *req_wr* y *write* en el bucle donde se realizan las escrituras. El código final del bucle de escritura quedaría, para este ejemplo, de la siguiente forma:

```
always @ (posedge clk)
  begin: write_result_mem
   // flop write data prior to sending to ram
   temp_0 <= tempRD_0 + tempRD_1;
   temp_1 <= tempRD_0 - rd_data;
   temp_2 <= temp_0 + temp_1;
   wr_data <= temp_2 + 10;
   tempWR_0 <= temp_2 + 20;
   tempWR_1 <= temp_2 + 30;

   if (write == 2'b1)
    begin
     wr_data <= tempWR_0;
     tempWR_1 <= tempWR_1;
    end
```

```
if (write == 2'b10)
 begin
  wr_data <= tempWR_1;
 end

// reset memory write
if (rst)
 begin
  wr_index <= 11'h0;
  wr_dvld  <= 1'b0;
  alg_done <= 1'b0;
  lat      <= 3'b0;
  wr_first <= 1'b1;
  req_wr   <= 2'b0;
  write    <= 2'b0;
 end // if (rst == 1'b1)
else
 begin
  // when the last item to read is vaild enable sram write
  wr_dvld <= ((lat == 3'b100) & (wr_first == 1'b1)) |
         ((wr_first == 1'b0) & (lat == 3'b10)) | (write != 2'b0);

  if (wr_first == 1'b1 & rd_dvld)
   begin
    if (lat == 3'b100)
     begin
      wr_first <= 1'b0;
      lat <= 3'b0;
     end
    else
      lat <= lat +3'b1;
   end
  else if (wr_first == 1'b0)
   begin
    if (lat == 3'b10)
      lat <= 3'b0;
    else
      lat <= lat + 3'b1;
   end

  if (((lat == 3'b100) & (wr_first == 1'b1)) |
      ((wr_first == 1'b0) & (lat == 3'b10)) | (write != 2'b0))
   begin
    if (write == 2'b10)
      write <= 2'b0;
    else
      write <= write + 2'b1;
   end
```

```
    // increment index after write
    if (wr_dvld)
     begin
      if (req_wr == 2'b10)
       begin
        req_wr <= 2'b0;
        wr_index <= wr_index + 11'b1;
       end
      else
        req_wr <= req_wr + 2'b1;
     end

    // all done assert alg_done
    done <= (wr_index == op_length_bound);
    alg_done <= done & wr_dvld;
   end // else: !if(rst)
 end // block: write_result_mem
```

Como nota final, añadir que cada ampliación realizada en este trabajo conserva todas las ampliaciones realizadas anteriormente, es decir, en el momento de realizar una nueva ampliación se tuvieron en cuenta todas las ampliaciones realizadas hasta ese momento para no perder el trabajo realizado.

## 3.6. Optimizaciones

La limitación de leer y escribir siempre 128 bits provoca, en caso de que los elementos a leer y escribir sean menores de ese tamaño, una gran pérdida de espacio y tiempo. Para solucionar esta limitación, se planteó aplicar una optimización al programa que permitiera leer y escribir varios elementos a la vez, siempre aprovechando al máximo las lecturas y escrituras que suelen ser el cuello de botella en este tipo de aplicaciones.

Para aplicar esta optimización se tiene que tener en cuenta la diferen-

cia entre el tamaño de los elementos de las variables y el tamaño de lectura/escritura. Por tanto, para variables de 64 bits, se deben leer y escribir dos elementos cada vez. Este cambio se puede ver en el siguiente código:

```
wr_data[63:0] <= (temp_0[63:0] & temp_1[63:0]) | rd_data[63:0];
wr_data[127:64] <= (temp_0[127:64] & temp_1[127:64]) | rd_data[127:64];
```

De esta manera, además, se separan las operaciones para los dos elementos que se han leído a la vez y se pueden utilizar operaciones que no sean tipo bit a bit (ya que cada elemento se calcula por separado).

Un ejemplo con sumas y restas y elementos de tamaño 64 bits que ya funcionó en este punto del desarrollo:

```
wr_data[63:0] <= (temp_0[63:0] + temp_1[63:0]) - rd_data[63:0];
wr_data[127:64] <= (temp_0[127:64] + temp_1[127:64]) - rd_data[127:64];
```

Con esta optimización, como se ha podido ver en los ejemplos anteriores, el tamaño de los elementos de las variables es un dato más importante, ya que según este tamaño, se deben leer y escribir más o menos elementos cada vez.

Por lo tanto, esta optimización provocó algunos cambios más en el código Verilog. Hay muchas señales y registros que se deben cambiar de tamaño según el tamaño de las variables, pero en este documento no entraremos en tanto detalle.

Los cambios importantes se hicieron en los extractors y en las operaciones.

En los extractors del tipo SRAM se debe reflejar el tamaño de las variables, por lo que con un ejemplo con elementos de tamaño 32 bits (y no 64 bits) quedarían de la siguiente forma:

```
// extractor SRAM:a_in  2048  32 sram[0] 0x0000 in  u stream
// extractor SRAM:b_in  2048  32 sram[0] 0x2000 in  u stream
// extractor SRAM:c_in  2048  32 sram[0] 0x4000 in  u stream
// extractor SRAM:d_out 2048  32 sram[1] 0x0000 out u stream
```

Así, se señala que los elementos de las variables son de 32 bits y además el tamaño total de cada variable es 0x2000 (8192 bytes = 65536 bits = 2048 * 32), es decir, la mitad que en el caso de 64.

Además, también se realizaron cambios importantes en las operaciones, ya que como se ha comentado anteriormente siempre se leen 128 bits, y al ser elementos de 32 bits, se leen cuatro elementos a la vez en cada lectura. Por lo tanto, en las operaciones hay que calcular cuatro elementos cada vez por separado y quedarían de la forma siguiente:

```
wr_data[31:0] <= (temp_0[31:0] & temp_1[31:0]) | rd_data[31:0];
wr_data[63:32] <= (temp_0[63:32] & temp_1[63:32]) | rd_data[63:32];
wr_data[95:64] <= (temp_0[95:64] & temp_1[95:64]) | rd_data[95:64];
wr_data[127:96] <= (temp_0[127:96] & temp_1[127:96]) | rd_data[127:96];
```

# Capítulo 4

# Resultados

El resultado de este proyecto ha sido un programa en Python que a través de un código C que se va a ejecutar en una FPGA, con los pragmas necesarios para saber los datos que el código C no contiene (ver apartado 4.5.), crea el código Verilog necesario para ejecutar correctamente el algoritmo deseado.

Inicialmente se quería que el programa aceptara todo tipo de entradas y salidas, pero por falta de tiempo no se ha podido realizar todo lo que se pretendía.

Los algoritmos permitidos en el programa son aquellos que cumplen las siguientes características:

✓ Variables con elementos de diversos tamaños.

✓ Todo tipo de operaciones (+,-,*,/,&,...).

✓ Varias líneas de operaciones.

✓ Diversas latencias.

✓ Varias variables de lectura.

✓ Varias variables de escritura.

Además, se han aplicado mejoras con respecto al código Verilog original:

✓ Paralelismo en las lecturas y escrituras.

✓ Segmentación de las lecturas y escrituras.

✓ Todo tipo de operaciones (+,-,*,/,&,...).

## 4.1. Limitaciones

Como ya se ha dicho anteriormente, al inicio del proyecto se pretendió hacer una herramienta automática para crear un código Verilog que funcionara en una FPGA para cualquier tipo de entradas y salidas, teniendo únicamente el código C. Finalmente no ha podido ser así, y el programa resultante de este proyecto tiene algunas limitaciones.

Una de las limitaciones más importantes es que todas las variables deben de ser del mismo tamaño y sus elementos también, tanto las variables de lectura como las variables de escritura.

Otra de las limitaciones de este proyecto es que pese a que la FPGA permite leer y escribir 64 bits en vez de 128 bits, no ha sido posible añadir esta optimización por falta de tiempo. De todas formas, debido a que en este proyecto se ha añadido una optimización para realizar las lecturas y escrituras de forma segmentada, poder leer y escribir 64 bits no supone ninguna ventaja respecto a leer y escribir 128 bits.

Otra de las limitaciones más importantes es que los algoritmos únicamente pueden contener operaciones, y por tanto no pueden contener otro tipo de instrucciones que no sean operaciones. Es decir, no pueden tener ni expresiones de decisión (como *if*, *else* o *switch*) ni expresiones de iteración (como *for*, *while*, *do-while*, *break* o *continue*) [23]. Solo hay una excepción a esta

limitación y es en el caso de que las variables sean vectores, que entonces se pueden realizar operaciones sobre todos sus elementos (del tipo A = B + C) conteniendo una expresión de iteración en su equivalente en código C, como en el ejemplo siguiente:

```
for(i=0; i<SIZE; i++) {
    A[i] = B[i] + C[i];
}
```

No obstante, esto no era un objetivo de este proyecto, ya que en ningún momento se ha pretendido implementar un traductor de lenguaje C a lenguaje HDL.

## 4.2. Contador de tiempos

Una vez acabado el proyecto, para comprobar los resultados obtenidos la mejor manera es medir los tiempos de ejecución de algunos algoritmos. Con ello se pueden observar las mejoras obtenidas con las optimizaciones añadidas al programa y además comparar la ejecución de los algoritmos en la FPGA mediante hardware o software.

Primero se van a comparar tres algoritmos, ejecutados mediante hardware en la FPGA, sin las optimizaciones realizadas en este proyecto con los mismos tres algoritmos con las mejoras incluidas. Estas optimizaciones consisten en la segmentación de las lecturas y escrituras, de manera que se leen y escriben varios elementos en un solo ciclo de reloj (para más información leer la sección 3.6. Optimizaciones).

El primer algoritmo ejecuta la siguiente operación:

```
D = (A & B) | C;
```

Las cuatro variables son vectores de 2048 elementos y cada elemento de 64 bits.

El segundo algoritmo realiza la misma operación que el anterior, los cuatro vectores tienen 2048 elementos, pero cada elemento es de 32 bits de tamaño en vez de 64.

El tercer algoritmo tiene dos variables de entrada en vez de tres como los dos ejemplos anteriores. La operación que realiza es la siguiente:

```
C = A & B;
```

Las tres variables son vectores de 2048 elementos (como los dos ejemplos anteriores) y cada elemento es de 32 bits.

Como se puede ver en la figura 4.1, en los tres casos la mejora del tiempo es notable, mejorando unos 250 microsegundos la versión optimizada con respecto a la versión sin optimizar. Teniendo en cuenta que la ejecución de un algoritmo sin optimizar ronda los 2.350 milisegundos, esto supone una mejora casi del 11 % en estos tres algoritmos optimizados con respecto a la versión sin optimizar.

Cabe destacar que la mayor parte del tiempo de ejecución de un algoritmo ejecutado en la FPGA se corresponde al tiempo de transferencia de las variables, tanto de entrada como de salida. Por tanto, el tiempo de cálculo de un algoritmo es una pequeña parte del tiempo total de ejecución.

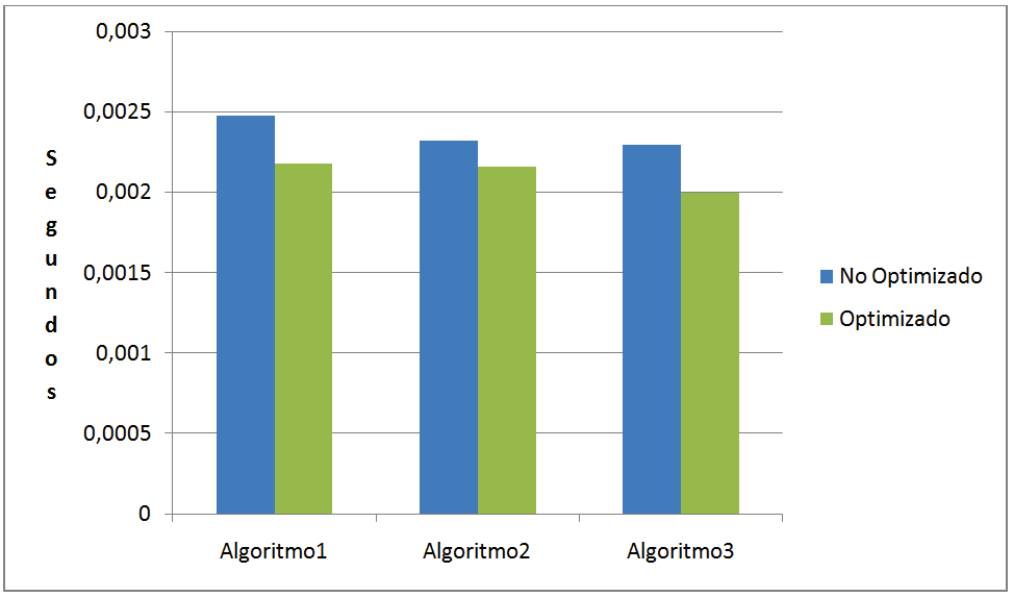



Figura 4.1: Comparación de tiempos de ejecución de tres algoritmos (sin optimizaciones y con optimizaciones)

La siguiente comparación se va a realizar sobre los tiempos de ejecución de un mismo algoritmo con diferentes tamaños de variables. El algoritmo ejecuta la siguiente operación:

```
D = (A & B) | C;
```

Las cuatro variables son vectores de elementos de tamaño 64 bits cada uno, y en cada ejemplo se han variado los números de elementos de estos vectores, desde 2048 a 32. Como se puede ver en la figura 4.2, con la optimización, los tiempos mejoran si hay menos elementos. En cambio, midiendo los tiempos para estos mismos ejemplos sin la optimización realizada en este proyecto, el tiempo no mejora pese a cambiar el número de elementos y además el tiempo es muy superior a los mismos pero optimizados, como se puede ver en la figura 4.2.

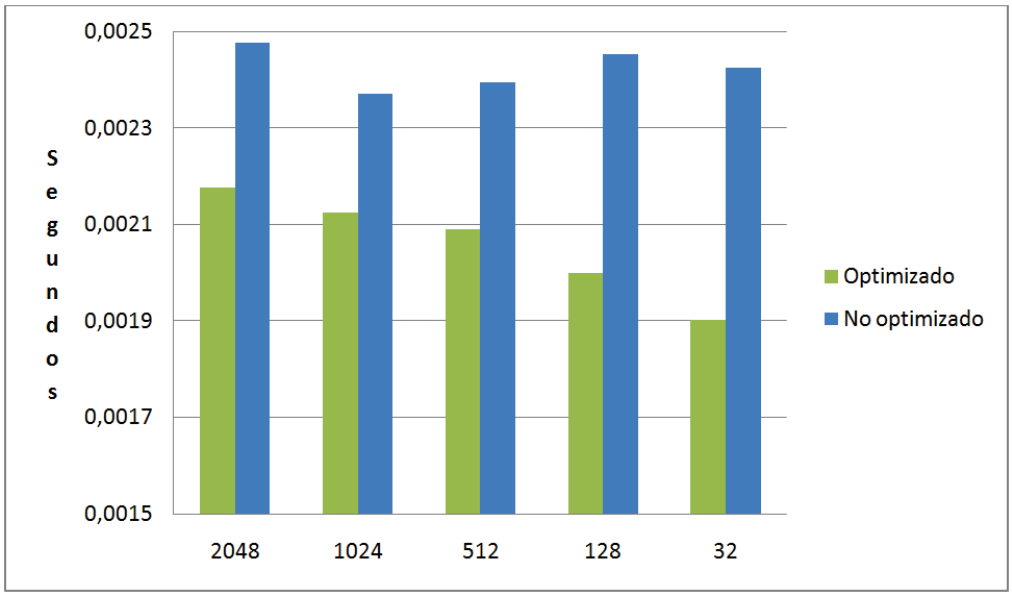



Figura 4.2: Comparación de tiempos de ejecución de un algoritmo con diferentes tamaños de vectores

La mayor parte de estas mejoras del tiempo de ejecución en los algoritmos optimizados se deben a que cuantos menos elementos hay en los vectores, el tiempo de transferencia es menor.

Algo parecido pasa si para el mismo algoritmo del ejemplo anterior, en vez de cambiar el número de elementos, se cambia el tamaño de cada elemento. Así, con 2048 elementos de tamaños entre 8 octetos y 1 octeto, el tiempo con la optimización también mejora con elementos más pequeños, como se puede ver en la figura 4.3.

Como en el caso anterior, los tiempos de los ejemplos sin optimizar no mejoran teniendo elementos más pequeños y en todos los casos son tiempos muy superiores a los optimizados.

En este caso los tiempos de los algoritmos optimizados no mejoran tanto

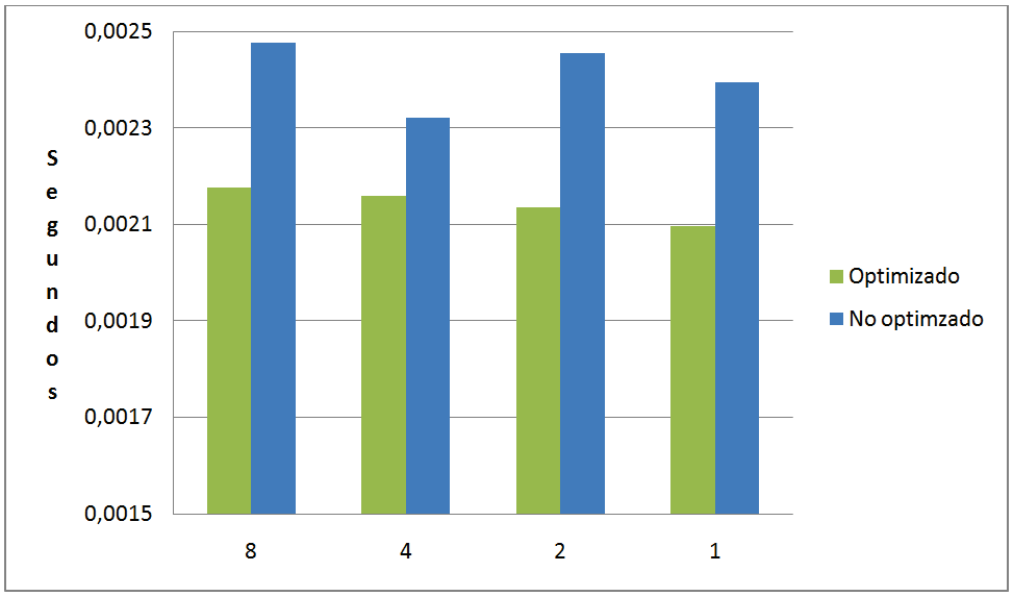



Figura 4.3: Comparación de tiempos de ejecución de un algoritmo con diferentes tamaños de elementos

como en el caso anterior debido al tiempo de transferencia. En este caso el número de elementos de los vectores se mantiene y por tanto el tiempo de transferencia es el mismo.

La última comparación se va a realizar sobre los tiempos de ejecución de algunos algoritmos optimizados y ejecutados mediante hardware (en la FPGA), con los mismos algoritmos ejecutados en software (en código C, también en la FPGA). En la figura 4.4 se muestran los tiempos de ocho algoritmos ejecutados por hardware en la FPGA, mientras que en la figura 4.5 se muestran los mismos algoritmos ejecutados por software en la FPGA.

El algoritmo número uno realiza la siguiente operación:

```
D = (A & B) | C;
```

Las cuatro variables son vectores de 2048 elementos y cada elemento de 64

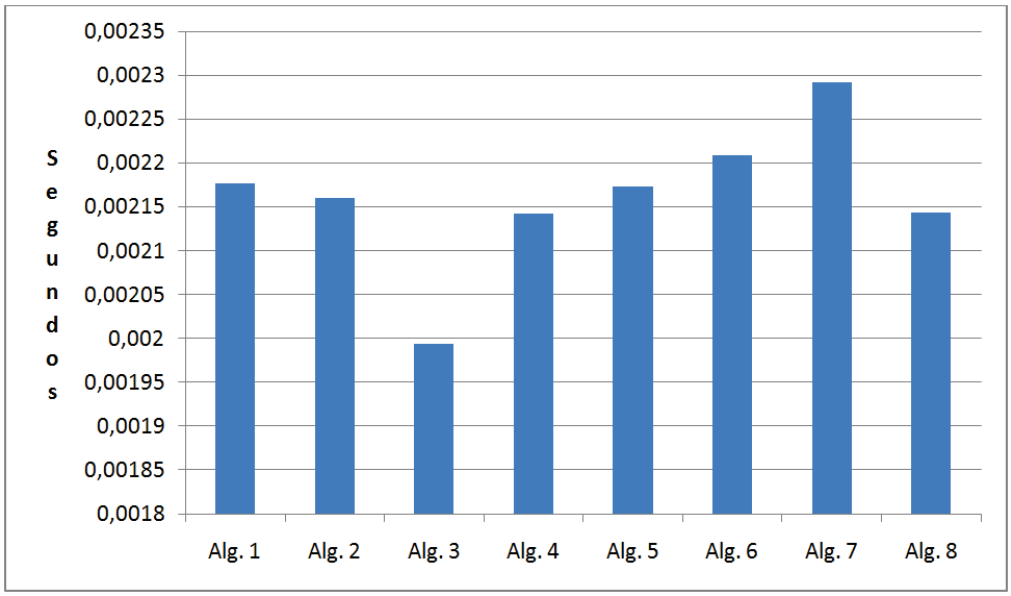



Figura 4.4: Comparación de tiempos de ejecución de siete algoritmos (ejecución hardware con optimizaciones)

bits.

El segundo algoritmo realiza la misma operación que el anterior, con 2048 elementos pero cada elemento con 32 bits.

El tercer algoritmo realiza la siguiente operación:

```
C = A & B;
```

Las tres variables son vectores de 2048 elementos y cada elemento de 32 bits.

El algoritmo número cuatro realiza la siguiente operación:

```
C = A + B;
```

Las tres variables son vectores de 2048 elementos y cada elemento de 32 bits.

El quinto algoritmo realiza las siguientes operaciones:

```
temp0 = A + B;
```

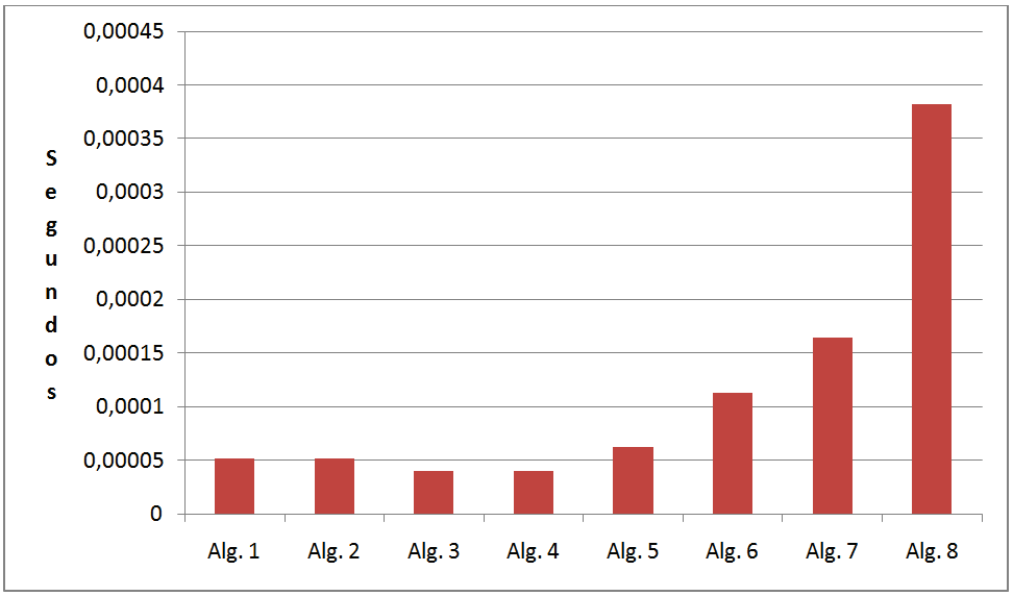



Figura 4.5: Comparación de tiempos de ejecución de siete algoritmos (ejecución software)

```
temp1 = A - C;

temp2 = temp0 + temp1;

D = temp2 + 10;
```

Las cuatro variables son vectores de 2048 elementos y cada elemento es de 32 bits. Además es necesario añadir una latencia de 3 para que las operaciones se realicen correctamente.

El sexto algoritmo realiza las siguientes operaciones:

```
temp0 = A + B;

temp1 = A - C;

temp2 = temp0 + temp1;

D = temp2 + 10;

E = temp2 + 20;
```

Las cinco variables son vectores de 2048 elementos y cada elemento es de 32 bits. En este caso también se necesita añadir una latencia de 3 y hay dos variables de escritura.

El séptimo algoritmo realiza las siguientes operaciones:

```
temp0 = A + B;
temp1 = A - C;
temp2 = temp0 + temp1;
D = temp2 + 10;
E = temp2 + 20;
F = temp2 + 30;
```

Las seis variables son vectores de 2048 elementos y cada elemento es de 32 bits. En este caso también se necesita añadir una latencia de 3 y hay tres variables de escritura.

El octavo y último algoritmo es el más complejo de todos los expuestos hasta ahora. Se trata de un algoritmo que, aunque es mucho más sencillo, intenta usar el mismo tipo de operaciones que los algoritmos de codificación tipo DES (del inglés *Data Encryption Standard*) [24]. Este octavo algoritmo realiza las siguientes operaciones:

```
temp0 = A << 3;
temp1 = B >> 5;
temp2 = C << 7;
temp3 = D << 9;

temp4 = (temp0 & 15) | (temp1 & 240);
```

```
temp5 = (temp2 & 15) & (temp4 & 240);


temp6 = temp4 << 3;
temp7 = temp5 << 1;


E = temp6 ^ temp2;
F = temp6 ^ temp1;
G = temp7 ^ temp2;
H = temp7 ^ temp1;
```

Las ocho variables son vectores de 4096 elementos y cada elemento es de 16 bits. En este caso se necesita añadir una latencia de 4.

Lo primero que llama la atención es que la ejecución por software es del orden de 10 veces más rápida que por hardware, ya que mientras que por hardware el algoritmo se ejecuta en unos 2.100 microsegundos, por software se ejecuta en unos 200 microsegundos. Esto se debe principalmente a que el procesado en la FPGA paga un tiempo extra de transferencia de datos entre la memoria principal y la memoria de la FPGA para enviar los datos de entrada, y entre la memoria de la FPGA y la principal para leer los datos de salida. Este tiempo de transferencia es imposible de ocultar con algoritmos tan sencillos como los aquí presentados pero se compensa para algoritmos más complejos gracias a la capacidad de explotar paralelismo de datos de la FPGA.

Por otro lado, parece que por software, al complicarse el algoritmo, como en los algoritmos 5, 6, 7 y 8, el coste temporal aumenta muy rápidamente (casi aumenta exponencialmente). Por hardware, en cambio, el coste temporal

en los algoritmos 5, 6, 7 y 8 en proporción aumenta mucho menos. Entonces, se puede decir que ejecutando un algoritmo más complejo por hardware tardará casi lo mismo que uno más sencillo (algoritmos 1, 2, 3 y 4) mientras que ejecutando un algoritmo complejo por software tarda tres veces más que uno menos complejo.

Otro dato que se puede observar al estudiar los tiempos de los ocho algoritmos es que cambiar el tamaño de los elementos no afecta el tiempo de ejecución en software (algoritmos 1 y 2). Además, tampoco afecta al tiempo de ejecución software el tipo de operación que se realice, al menos para operaciones sencillas (algoritmos 3 y 4).

En cambio, dependiendo de las operaciones que se realicen el tiempo cambia en la ejecución hardware (algoritmos 3 y 4). En concreto, la FPGA trabaja mejor con algoritmos que no implican acarreo (algoritmo 3) con respecto a los que si implican acarreo (algoritmo 4). Esto se debe a que con algoritmos sin acarreo la FPGA puede usar al máximo su paralelismo.

En el caso del algoritmo 8, se ha intentado crear un algoritmo que sea favorable a este paralelismo para que la FPGA trabaje a pleno rendimiento. En este caso se ha podido comprobar que pese a ser un algoritmo más complejo que, por ejemplo, el algoritmo 7, la FPGA tarda menos en ejecutar el algoritmo 8. Esto es debido a que el algoritmo 8 no realiza ninguna operación que implique acarreo y por tanto la FPGA puede utilizar al máximo su paralelismo. Además, el tiempo de ejecución del algoritmo 8 por software aumenta considerablemente si se compara con un algoritmo menos complejo (como el algoritmo 7) mientras que por hardware se mantiene similar a los otros algoritmos.

Por tanto, la FPGA es más eficiente con algoritmos más complejos y que se puedan ejecutar utilizando paralelismo.

Todos los algoritmos utilizados en estas pruebas se encuentran en el CD adjunto a este documento. Para más información, leer el apéndice B (Contenido del CD).

# Capítulo 5

# Análisis temporal y económico

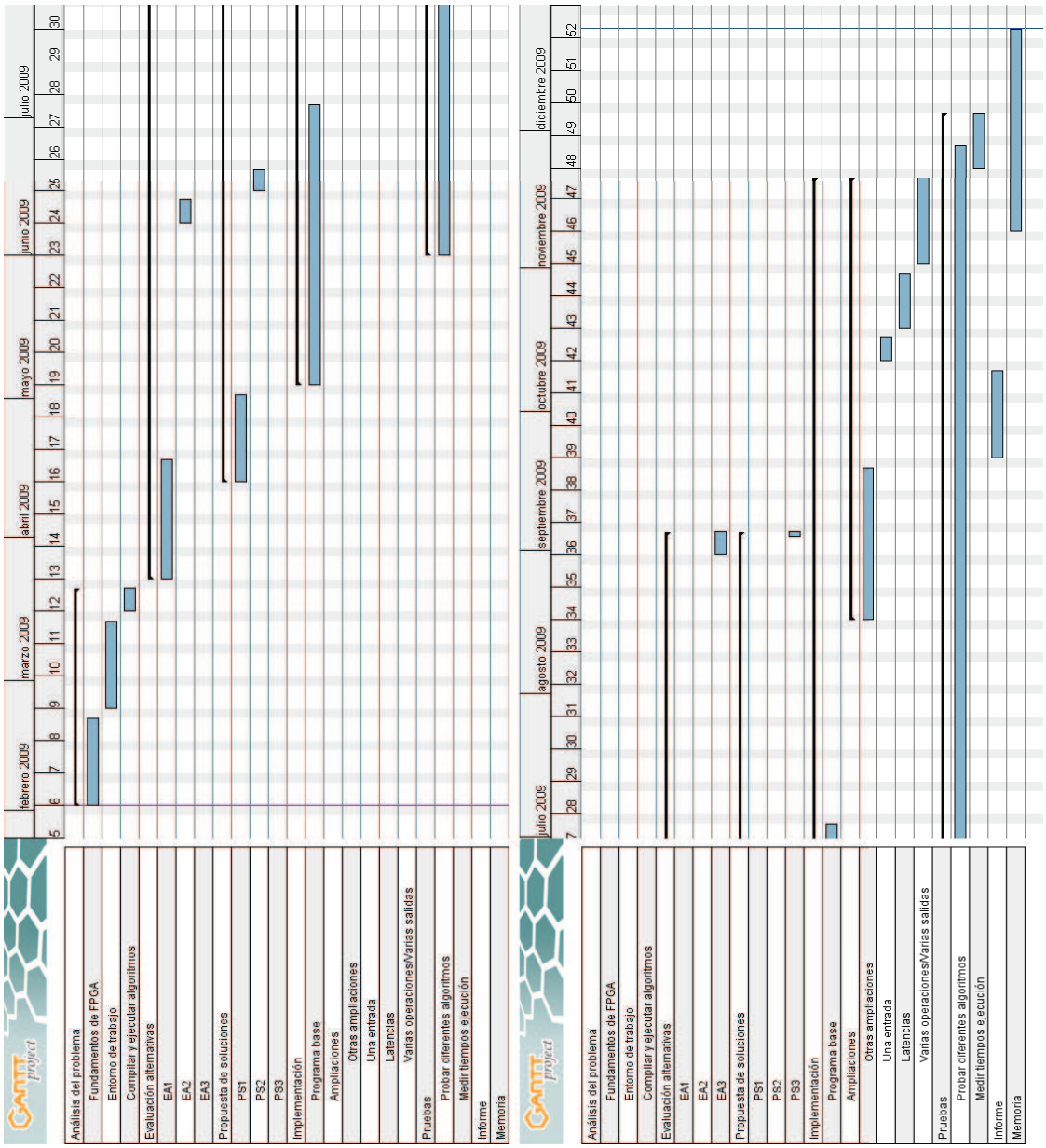
En este capítulo del documento se va a realizar un estudio del coste temporal y económico del proyecto.

## 5.1. Análisis temporal

En este apartado se muestra la planificación inicial del proyecto y la comparación con el tiempo final dedicado al mismo. En la planificación inicial se indica el tiempo que se preveía dedicar a cada parte del proyecto. Además también se muestra un análisis concreto de las horas que se han dedicado a cada tarea.

En el diagrama de Gantt de la página siguiente se puede ver la planificación inicial realizada para este proyecto desglosada por tareas y donde se puede ver fácilmente el tiempo estimado que se pretendía dedicar a cada una de ellas.

Como se puede ver, la primera parte del proyecto consistió en analizar el problema y entender todo lo relacionado con las FPGAs. Esta parte del


GANTT project
febrero 2009
marzo 2009
abril 2009
mayo 2009
junio 2009
julio 2009
Análisis del problema
Fundamentos de FPGA
Entorno de trabajo
Compilar y ejecutar algoritmos
Evaluación alternativas
EA1
EA2
EA3
Propuesta de soluciones
PS1
PS2
PS3
Implementación
Programa base
Ampliaciones
Otras ampliaciones
Una entrada
Latencias
Varias operaciones/Varias salidas
Pruebas
Probar diferentes algoritmos
Medir tiempos ejecución
Informe
Memoria
julio 2009
agosto 2009
septiembre 2009
octubre 2009
noviembre 2009
diciembre 2009

proyecto ocupó unas siete semanas.

Durante toda la duración del proyecto se fueron proponiendo soluciones para los problemas que fueron surgiendo.

La parte más importante del proyecto fue la implementación, que fue la que requirió más tiempo. En concreto la tarea de realizar varias salidas se retrasó bastante y no se pudo finalizar hasta la segunda semana de diciembre. Esto provocó un retraso en las tareas que se debían realizar a continuación. Es decir, las pruebas se retrasaron hasta la tercera semana de diciembre. La redacción de esta memoria no se vio afectada por el retraso ya que se le adjudicó inicialmente más tiempo de lo que finalmente ha sido necesario.

En total han sido 11 meses de proyecto y un total de unas 750 horas, repartidas por tareas como se puede ver en la figura 5.1.

| Tarea | Duración (Horas) |
|---|---|
| Análisis del problema | 67 |
| Evaluación de alternativas | 25 |
| Propuesta de soluciones | 12 |
| Implementación | 474 |
| Pruebas | 126 |
| Documentación | 42 |
| Total | 746 |

Figura 5.1: Tiempo dedicado a cada tarea

## 5.2. Análisis económico

Una vez definidas las horas que se requieren para cada tarea, se va a pasar a definir el coste total del proyecto, y para ello se va a dividir el estudio económico en 3 partes:

- Coste personal
- Coste del hardware
- Coste de software

### 5.2.1. Coste personal

Se considera que para la realización de este proyecto se requieren dos perfiles diferentes:

1. Analista programador experto en lenguaje Verilog
2. Programador en Python

El salario medio de un Analista programador son unos 22.000 euros al año, con lo que a la hora supone unos 11 euros más el coste de la seguridad social (un 30 % más aproximadamente). El cálculo del coste por hora se puede ver en la figura 5.2.

| Anual | Mensual | Semanal | Diaria | Hora | Coste SS | Total |
|---|---|---|---|---|---|---|
| 22.000 € | 1.833,33 € | 458,33 € | 91,67 € | 11,46 € | 3,44 € | 14,9 € |

Figura 5.2: Coste de un Analista programador

Por otro lado, el salario medio de un Programador son unos 19.800 euros al año, y con el mismo cálculo anterior (que se puede ver en la figura 5.3), supone unos 10 euros a la hora más el coste de la seguridad social (un 30 % más aproximadamente).

| Anual | Mensual | Semanal | Diaria | Hora | Coste SS | Total |
|---|---|---|---|---|---|---|
| 19.800 € | 1.650,00 € | 412,50 € | 82,50 € | 10,31 € | 3,09 € | 13,4 € |

Figura 5.3: Coste de un Programador

La cantidad de horas que se debe dedicar a cada trabajo se muestran en la figura 5.4.

| Trabajo | Horas |
|---|---|
| Analista programador experto en lenguaje Verilog | 406 |
| Programador en Python | 340 |

Figura 5.4: Horas dedicadas por cada trabajo

Por lo tanto, los costes de personal son unos 10.600 euros, como se puede ver en la figura 5.5.

| Trabajo | Horas | Coste / hora | Total |
|---|---|---|---|
| Analista progr. experto en lenguaje Verilog | 406 | 14,9 | 6.049,40 € |
| Programador en Python | 340 | 13,4 | 4.556,00 € |
| Total costes de personal | | | 10.605,40 € |

Figura 5.5: Costes totales de personal

### 5.2.2. Coste del hardware

A los equipos se les suele considerar una vida útil de unos 3 años, por lo que se va a amortizar la parte proporcional a la realización del proyecto

(unos 11 meses).

En la figura 5.6 se puede ver el coste total de cada equipo y la parte proporcional que se puede imputar a este proyecto, en total los costes ascienden a unos 3.000 euros.

| Equipo | Coste total | Duración proyecto | Coste imputable |
|---|---|---|---|
| FPGA | 6.000,00 € | 11 meses | 1.833,33 € |
| FPGApc | 2.500,00 € | 11 meses | 763,89 € |
| Equipo de trabajo | 1.300,00 € | 11 meses | 397,22 € |
| | | Total costes de hardware | 2.994,44 € |

Figura 5.6: Costes totales de hardware

### 5.2.3. Coste del software

Una vez se ha calculado el coste de los equipos, se puede calcular el coste de las licencias software utilizadas en cada uno de ellos.

No obstante, prácticamente todo el software utilizado en el proyecto es Open Source gratuito, el único software propietario que se ha utilizado es el compilador xst integrado en la ISE para FPGAs de Xilinx.

Igual que en el caso de los costes de hardware, se va a considerar únicamente la parte proporcional de la realización del proyecto (unos 11 meses) con respecto a la vida útil del equipo (unos 3 años).

En la figura 5.7 se puede ver el coste total de software y la parte proporcional que se puede imputar a este proyecto. En total los costes ascienden a algo más de 600 euros.

| Software | Coste total | Duración proyecto | Coste imputable |
|---|---|---|---|
| ISE Xilinx | 2.066,80 € | 11 meses | 631,52 € |
| | | **Total costes de software** | **631,52 €** |

Figura 5.7: Costes totales de software

### 5.2.4. Coste total

Una vez detallados todos los costes, se puede calcular el coste total que supone el proyecto. Además de los costes calculados hasta este momento, se han añadido unos costes adicionales, ya que siempre pueden surgir imprevistos que pueden encarecer el proyecto.

En la figura 5.8 se muestran los costes totales de este proyecto, que ascienden a unos 15.200 euros.

| Estudio económico | Coste |
|---|---|
| **Coste de personal** | 10.605,40 € |
| **Coste de hardware** | 2.994,44 € |
| **Coste de software** | 631,52 € |
| **Gastos adicionales** | 1.000,00 € |
| **Total coste proyecto** | **15.231,36 €** |

Figura 5.8: Costes totales del proyecto

# Capítulo 6

# Conclusiones

En este proyecto se ha implementado un programa que ayuda en la programación en FPGAs, creando el código Verilog automáticamente a partir del código C.

Además se ha comprobado que el programa funciona con diversos algoritmos distintos y que, efectivamente, crea un código Verilog que funciona correctamente.

Al no existir demasiados ejemplos de códigos Verilog para FPGA, se han tenido que crear diversos códigos para comprobar su funcionamiento y así poder automatizar posteriormente el proceso. La creación de estos códigos Verilog para utilizar como ejemplos ha sido una de las tareas más importantes del proyecto y además ha requerido mucho tiempo.

Se ha comprobado, tomando los tiempos de ejecución de diversos algoritmos, que la FPGA es más lenta ejecutando un algoritmo que ejecutándolo directamente en código C. No obstante, con algoritmos más complejos la FPGA se comporta mejor, ya que tarda prácticamente el mismo tiempo en ejecutar un algoritmo sencillo que uno muy complejo. En C, en cambio, un algoritmo complejo tarda prácticamente tres veces más que un algoritmo

más sencillo. Se ha comprobado también que la FPGA se comporta mejor con algoritmos que no impliquen acarreo y que por tanto permiten utilizar paralelismo.

Además se ha comprobado también, midiendo los tiempos de ejecución de diversos algoritmos, que las optimizaciones realizadas en este proyecto ahorran más de un 10 % del tiempo de ejecución respecto a los mismos algoritmos sin optimizar.

Por último, se ha realizado un estudio del coste temporal y económico de este proyecto.

## 6.1. Trabajo futuro

Hay diversas contribuciones que se han llevado a cabo en este proyecto. Pero, por otro lado, aún hay muchas características que se podrían añadir en un futuro.

Se ha testeado el programa con un número limitado de algoritmos. Por ello, no es seguro al 100 % que el programa funcione con todo tipo de algoritmos. Por tanto, habría que probar con muchos más algoritmos para comprobar el correcto funcionamiento del programa en todos los casos.

Además, solo se ha podido probar el programa con una única FPGA, con lo que sería interesante probarlo con más FPGAs.

Otra característica importante, como se ha comentado anteriormente en este documento, es que todas las variables del algoritmo deben tener el mismo tamaño y sus elementos también. Un posible trabajo futuro interesante sería realizar una ampliación para que las variables fueran independientes entre

sí y cada una pudiera ser de un tamaño diferente.

Otro elemento que se debería mejorar es que pese a que la FPGA permite leer y escribir 64 bits además de 128 bits, no ha sido posible añadir esta optimización por falta de tiempo. Por tanto, en el futuro se podría añadir esta posibilidad al programa y estudiar si es una mejora leer y escribir 64 bits y no 128 para algunos algoritmos.

Por último, sería conveniente juntar el programa con un compilador de lenguaje C a lenguaje HDL de forma que se pudieran utilizar en los algoritmos tanto expresiones de decisión (como *if*, *else* o *switch*) como expresiones de iteración(como *for*, *while*, *do-while*, *break* o *continue*) [23].

Incluyendo todas estas características ya se podría decir que el programa acepta todo tipo de algoritmos.

# Bibliografía


[1] *Reconfigurable Application-Specific Computing User's Guide* [en línea] SGI. 28 febrero 2008. http://techpubs.sgi.com/library/manuals/4000/007-4718-007/pdf/007-4718-007.pdf [Consulta: 2 diciembre 2009]

[2] Wikipedia: *ASIC* [en línea] http://es.wikipedia.org/wiki/Asic [Consulta: 15 noviembre 2009]

[3] Wikipedia: *GPU* [en línea] http://es.wikipedia.org/wiki/GPU [Consulta: 16 noviembre 2009]

[4] Wikipedia: *Hardware description language* [en línea] http://en.wikipedia.org/wiki/Hardware_description_language [Consulta: 16 noviembre 2009]

[5] Wikipedia: *Verilog* [en línea] http://en.wikipedia.org/wiki/Verilog [Consulta: 16 noviembre 2009]

[6] Tom Goddard, *GPU Computing* [en línea] 15 diciembre 2008. http://www.cgl.ucsf.edu/chimera/group-meeting-dec2008/gpu.html [Consulta: 23 noviembre 2009]

[7] Wikipedia: *FPGA* [en línea] http://es.wikipedia.org/wiki/FPGA [Consulta: 18 noviembre 2009]

[8] Altera Robert Cottrell, *FPGA Coprocessors: Hardware IP for Software Engineers* [en línea] Design And Reuse. High Wycombe, UK. http://www.design-reuse.com/articles/6733/fpga-coprocessors-hardware-ip-for-software-engineers.html [Consulta: 24 noviembre 2009]

[9] Bob Zeidman, *Back to the Basics: All about FPGAs* [en línea] Programable Logic DesignLine. 21 marzo 2006. http://www.embedded.com/columns/technicalinsights/183701900 [Consulta: 24 noviembre 2009]

[10] Barcelona Supercomputing Center. http://www.bsc.es/ [Consulta: 28 noviembre 2009]

[11] Departamento AC de la Universidad Politécnica de Cataluña http://www.ac.upc.edu/ [Consulta: 28 noviembre 2009]

[12] Eclipse página web oficial http://www.eclipse.org/ [Consulta: 29 noviembre 2009]

[13] Pydev página web oficial http://pydev.org/ [Consulta: 29 noviembre 2009]

[14] Subclipse página web oficial http://subclipse.tigris.org/ [Consulta: 29 noviembre 2009]

[15] Facultad de Informática de Barcelona http://www.fib.upc.edu/fib/ [Consulta: 30 noviembre 2009]

[16] Universidad de Zaragoza http://www.unizar.es/index.html [Consulta: 6 diciembre 2009]

[17] Blog Janus http://www.janus-computer.com/category/fotos-janus/index.html [Consulta: 6 diciembre 2009]

[18] El País.com: *Janus, un supercomputador para Zaragoza* http://www.elpais.com/articulo/internet/Janus/supercomputador/ Zaragoza/elpeputec/20080527elpepunet_10/Tes [Consulta: 7 diciembre 2009]

[19] OpenMP página web oficial http://openmp.org/wp/ [Consulta: 1 diciembre 2009]

[20] Wikipedia: *OpenMP* http://en.wikipedia.org/wiki/OpenMP [Consulta: 1 diciembre 2009]

[21] Eduard Ayguade, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, Jose M. Perez y Enrique S. Quintana-Ort, *A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures.* Junio 2009.

[22] Scott Hauck y Andre Dehon, *Reconfigurable Computing: The theory and practice of FPGA-based computation.* Morgan Kaufmann publications, United States, 1ª edición, 2008.

[23] Wikilibros. *Programación en C++: Iteraciones y decisiones.* [en línea] http://es.wikibooks.org/wiki/Programación_en_C%2B%2B/Iteraciones_y_decisiones [Consulta: 4 diciembre 2009]

[24] Wikipedia: *Data Encryption Standard* http://en.wikipedia.org/wiki/Data_Encryption_Standard [Consulta: 12 diciembre 2009]

# Apéndice A

# Código Verilog

En este apéndice se muestra un código completo Verilog para una FPGA.

```
////////////////////////////////////////////////////////////
//       PLACE USER EXTRACTOR STATEMENTS HERE              //
////////////////////////////////////////////////////////////
// extractor VERSION: 6.3
// extractor CS: 2.2
// extractor SRAM:a_in  2048  64 sram[0] 0x0000 in  u stream
// extractor SRAM:b_in  2048  64 sram[0] 0x4000 in  u stream
// extractor SRAM:c_in  2048  64 sram[0] 0x8000 in  u stream
// extractor SRAM:d_out 2048  64 sram[1] 0x0000 out u stream
//
// extractor REG_IN:op_length1 10 u alg_def_reg[0][9:0]
//
// this is the top level module

`include "alg.h"


module alg_block_top

    (clk,                     // input core clock
     rst,                     // input core reset
     mem_0_rd_data,           // input sram 0 read data bus
     mem_0_rd_data_vld,       // input sram 0 read data valid
     mem_0_rd_busy,           // input ram 0 read busy
     mem_0_rd_fifo_cred,      // input ram 0 read FIFO credit
     mem_0_wr_busy,           // input ram 0 write busy
     mem_0_wr_fifo_cred,      // input ram 0 write FIFO credit
     mem_0_error,             // input ram 0 memory error
     alg_mem_0_offset,        // input PIO sram0 address offset 10-bit
     mem_0_rd_addr,           // output sram 0 read adddress bus 24-bit
     mem_0_rd_cmd_vld,        // output sram 0 read command valid
```

```
    mem_0_wr_addr,          // output sram 0 write address bus
    mem_0_wr_be,            // output sram 0 write byte enable bus 16-bit
    mem_0_wr_data,          // output sram 0 write data bus
    mem_0_wr_cmd_vld,       // output sram 0 write command data valid
    mem_1_rd_data,          // input sram 1 read data bus
    mem_1_rd_data_vld,      // input sram 1 read data valid
    mem_1_rd_busy,          // input ram 1 read busy
    mem_1_rd_fifo_cred,     // input ram 1 read FIFO credit
    mem_1_wr_busy,          // input ram 1 write busy
    mem_1_wr_fifo_cred,     // input ram 1 write FIFO credit
    mem_1_error,            // input ram 1 memory error
    alg_mem_1_offset,       // input PIO ram1 address offset 10-bit
    mem_1_rd_addr,          // output sram 1 read adddress bus 24-bit
    mem_1_rd_cmd_vld,       // output sram 1 read command valid
    mem_1_wr_addr,          // output sram 1 write address bus
    mem_1_wr_be,            // output sram 1 write byte enable bus 16-bit
    mem_1_wr_data,          // output sram 1 write data bus
    mem_1_wr_cmd_vld,       // output sram 1 write command data valid
    alg_def_reg0,           // input PIO write register 0 bus 64-bit
    alg_def_reg1,           // input PIO write register 1 bus 64-bit
    alg_def_reg2,           // input PIO write register 2 bus 64-bit
    alg_def_reg3,           // input PIO write register 3 bus 64-bit
    alg_def_reg4,           // input PIO write register 4 bus 64-bit
    alg_def_reg5,           // input PIO write register 5 bus 64-bit
    alg_def_reg6,           // input PIO write register 6 bus 64-bit
    alg_def_reg7,           // input PIO write register 7 bus 64-bit
    alg_def_reg_updated,    // Alg defined reg written by host
    alg_def_reg_polled,     // Alg defined reg read by host

    alg_done,               // output algorithm complete signal
    alg_events,             // Event triggers for CS interrupt generation
    debug0,                 // output debug ports  0 64-bit
    debug1,                 // output debug ports  1 64-bit
    debug2,                 // output debug ports  2 64-bit
    debug3,                 // output debug ports  3 64-bit
    debug4,                 // output debug ports  4 64-bit
    debug5,                 // output debug ports  5 64-bit
    debug6,                 // output debug ports  6 64-bit
    debug7,                 // output debug ports  7 64-bit
    debug8,                 // output debug ports  8 64-bit
    step_flag_out,          // output algorithm step complete
    alg_def_reg0_wr_data,   // output PIO write value register 0 bus 64-bit
    alg_def_reg1_wr_data,   // output PIO write value register 1 bus 64-bit
    alg_def_reg2_wr_data,   // output PIO write value register 2 bus 64-bit
    alg_def_reg3_wr_data,   // output PIO write value register 3 bus 64-bit
    alg_def_reg4_wr_data,   // output PIO write value register 4 bus 64-bit
    alg_def_reg5_wr_data,   // output PIO write value register 5 bus 64-bit
    alg_def_reg6_wr_data,   // output PIO write value register 6 bus 64-bit
    alg_def_reg7_wr_data,   // output PIO write value register 7 bus 64-bit
```

```
    alg_def_reg_write         // Indicate a write to a given alg defined reg
    );

// interface description
// module port direction definition
    input          clk, rst;
    input          mem_0_rd_data_vld;
    input [127:0]  mem_0_rd_data;
    input [23:14]  alg_mem_0_offset;
    input          mem_0_rd_busy;
    input          mem_0_wr_busy;
    input          mem_0_rd_fifo_cred;
    input          mem_0_wr_fifo_cred;
    input          mem_0_error;
    input     mem_1_rd_data_vld;
    input [127:0]  mem_1_rd_data;
    input [23:14]  alg_mem_1_offset;
    input          mem_1_rd_busy;
    input          mem_1_wr_busy;
    input          mem_1_rd_fifo_cred;
    input          mem_1_wr_fifo_cred;
    input          mem_1_error;
    input [63:0]   alg_def_reg0;
    input [63:0]   alg_def_reg1;
    input [63:0]   alg_def_reg2;
    input [63:0]   alg_def_reg3;
    input [63:0]   alg_def_reg4;
    input [63:0]   alg_def_reg5;
    input [63:0]   alg_def_reg6;
    input [63:0]   alg_def_reg7;
    input [`ADR_REG_NUM-1:0] alg_def_reg_updated;
    input [`ADR_REG_NUM-1:0] alg_def_reg_polled;

    output         alg_done;
    output [1:0]   alg_events; // Event triggers for CS interrupt generation
    output [63:0]  debug0;
    output [63:0]  debug1;
    output [63:0]  debug2;
    output [63:0]  debug3;
    output [63:0]  debug4;
    output [63:0]  debug5;
    output [63:0]  debug6;
    output [63:0]  debug7;
    output [63:0]  debug8;
    output [23:0]  mem_0_rd_addr;
    output [23:0]  mem_0_wr_addr;
    output [15:0]  mem_0_wr_be;
    output [127:0] mem_0_wr_data;
    output         mem_0_wr_cmd_vld;
```

```
output          mem_0_rd_cmd_vld;
output [23:0]   mem_1_rd_addr;
output [23:0]   mem_1_wr_addr;
output [15:0]   mem_1_wr_be;
output [127:0] mem_1_wr_data;
output          mem_1_wr_cmd_vld;
output          mem_1_rd_cmd_vld;
output          step_flag_out;
output [63:0]   alg_def_reg0_wr_data;
output [63:0]   alg_def_reg1_wr_data;
output [63:0]   alg_def_reg2_wr_data;
output [63:0]   alg_def_reg3_wr_data;
output [63:0]   alg_def_reg4_wr_data;
output [63:0]   alg_def_reg5_wr_data;
output [63:0]   alg_def_reg6_wr_data;
output [63:0]   alg_def_reg7_wr_data;
output [`ADR_REG_NUM-1:0] alg_def_reg_write;

wire            mem_0_rd_cmd_vld;
wire [23:0]     mem_0_rd_addr;

wire            mem_1_wr_cmd_vld;
wire [23:0]     mem_1_rd_addr;
wire [15:0]     mem_1_wr_be;
wire [23:0]     mem_1_wr_addr;
wire [127:0]    mem_1_wr_data;

wire [63:0]   debug0;
reg  [63:0]   debug1;
reg  [63:0]   debug2;
reg  [63:0]   debug3;
reg  [63:0]   debug4;
reg  [63:0]   debug5;
reg  [63:0]   debug6;
reg  [63:0]   debug7;
reg  [63:0]   debug8;

reg           done;
reg           alg_done;
wire [1:0]    alg_events;


wire [63:0]   alg_def_reg0_wr_data, alg_def_reg1_wr_data;
wire [63:0]   alg_def_reg2_wr_data, alg_def_reg3_wr_data;
wire [63:0]   alg_def_reg4_wr_data, alg_def_reg5_wr_data;
wire [63:0]   alg_def_reg6_wr_data, alg_def_reg7_wr_data;

reg  [9:0]    op_length_bound; //in Qwords; value = length in qwords-1
```

```
    reg          rd_cmd_vld;
    reg  [9:0]   rd_index;
    reg          rd_dvld;
    reg  [127:0] rd_data;
    reg  [1:0]   req_abc;
    reg  [1:0]   read_abc;
    reg          read_done;
    reg          wr_dvld;
    reg  [9:0]   wr_index;
    reg  [127:0] wr_data;

    reg  [127:0] temp_a, temp_b;


// set step increment to one clock
    assign  step_flag_out = 1'b1;
    assign  alg_events = 2'b0;

// alg does not write mem 0
    assign  mem_0_wr_cmd_vld = 1'b0;
    assign  mem_0_wr_addr    = 24'b0;
    assign  mem_0_wr_be      = 16'h0;
    assign  mem_0_wr_data    = 128'b0;
// alg does not read mem 1
    assign  mem_1_rd_cmd_vld = 1'b0;
    assign  mem_1_rd_addr    = 24'b0;

// assign reset values to def regs

    assign  alg_def_reg0_wr_data = 64'h0000_0000_0000_03ff; // [9:0] OP_LENGTH-1
    assign  alg_def_reg1_wr_data = 64'h3333_2222_1111_0000;
    assign  alg_def_reg2_wr_data = 64'h0;
    assign  alg_def_reg3_wr_data = 64'h0;
    assign  alg_def_reg4_wr_data = 64'h0;
    assign  alg_def_reg5_wr_data = 64'h0;
    assign  alg_def_reg6_wr_data = 64'h0;
    assign  alg_def_reg7_wr_data = 64'h0;

    assign  alg_def_reg_write = 8'b0;

    assign  mem_0_rd_cmd_vld = rd_cmd_vld;
    assign  mem_0_rd_addr = {alg_mem_0_offset[23:16], req_abc, rd_index, 4'b0};

    assign  mem_1_wr_cmd_vld = wr_dvld;
    assign  mem_1_wr_addr = {alg_mem_1_offset[23:14],          wr_index, 4'b0};
    assign  mem_1_wr_be   = 16'hffff;
    assign  mem_1_wr_data = wr_data;
```

```
// Internally flop MMR values
always @(posedge clk)
  begin
      op_length_bound <= rst ? 10'h3ff : alg_def_reg0[9:0];
  end


// read memory
always @ (posedge clk)
    begin: read_mem
// reset memory read then start reading
if (rst)
  begin
    rd_cmd_vld <= 1'b0;
    rd_index   <= 10'h00;
    req_abc    <= 2'b0;
    read_done  <= 1'b0;
  end // if (rst)
else
  begin
    if (!read_done)
begin
    rd_cmd_vld <= 1'b1;
    if (rd_cmd_vld)
         begin
    if (req_abc == 2'b01 && rd_index == op_length_bound)
      begin
read_done <= 1'b1; // Next read is last one
      end
    if (req_abc == 2'b10)
      begin
req_abc <= 2'b00;
rd_index <= rd_index + 10'b1;
      end
    else
      begin
req_abc <= req_abc + 1'b1;
      end
 end
end // if (!read_done)
    else
begin
    rd_cmd_vld <= 1'b0;
end // else: !if(!read_done)
  end
    end // block: read_mem
```

```
always @ (posedge clk)
    begin:flop_read_data

rd_data <= mem_0_rd_data;
if (rst)
  begin
      rd_dvld   <= 1'b0;
      read_abc  <= 2'b0;
  end // if (rst)
else
  begin
      rd_dvld   <= mem_0_rd_data_vld;
      if (rd_dvld)
begin
  if (read_abc == 2'b10)
    begin
      read_abc <= 2'b00;
    end
  else
    begin
      read_abc <= read_abc + 2'b01;
    end
  if (read_abc == 2'b00)
    temp_a <= rd_data;
  if (read_abc == 2'b01)
    temp_b <= rd_data;
end
    end // else: !if(rst)
    end // block: flop_read_data

//
// write results in memory 1
//

always @ (posedge clk)
    begin: write_result_mem

      // flop write data prior to sending to ram
      wr_data <= (temp_a & temp_b) | rd_data;

      // reset memory write
      if (rst)
begin
    wr_index <= 10'h0;
    wr_dvld  <= 1'b0;
    alg_done <= 1'b0;
end // if (rst == 1'b1)
      else
begin
```

```
    // when C is vaild enable sram write
    wr_dvld <= rd_dvld & (read_abc == 2'b10);

    // increment index after write
    if (wr_dvld)
      wr_index <= wr_index + 10'h001;

    // all done assert alg_done
    done <= (wr_index == op_length_bound);
    alg_done <= done & wr_dvld;
end // else: !if(rst)

    end // block: write_result_mem

endmodule
```

# Apéndice B

# Contenido del CD

El CD adjunto con este documento contiene el código que se ha desarrollado en el proyecto y algunos algoritmos de ejemplo.

En el directorio "FPGA" del CD, se encuentra el código del programa que se ha desarrollado durante el proyecto.

Para utilizar el programa se debe llamar al archivo main.py ubicado en "FPGA/src/fpga" de la siguiente manera:

```
./fpga.py [-o fichero_destino / --output=fichero_destino][-s
size / --size=tamaño_lect_escr] fichero_origen
```

- El fichero_origen debe tener extensión .c
- Opciones -h y –help para imprimir la ayuda
- En caso de no especificar una ruta destino, se usará la ruta por defecto (en el directorio ../out/ y con nombre "alg_block_top.v")
- En caso de no especificar un tamaño de lectura y escritura, se asignará 128 por defecto

AVISO: En caso de no existir el .o llamado igual que fichero_origen, se intenta compilar el fichero_origen, dando un error si no se puede compilar y acabando el programa.

En el directorio “ejemplos” se encuentran algunos algoritmos de ejemplo. Todos los ejemplos contienen el código C (.c) y sus compilados (.o y ejecutable), el MAKEFILE para compilar el .c, el código VERILOG, y los dos archivos de configuración (.cfg) y el archivo binario (.bin) necesarios para ejecutar el cogido en la FPGA.

Este directorio está dividido en dos subdirectorios: “NoOptimizado” y “Optimizado”.

En el directorio “NoOptimizado” se encuentran los algoritmos sin optimizar.

En el directorio “Optimizado” se encuentran los algoritmos optimizados.