ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Trabajo Fin de Grado

Desarrollo de un prototipo de plataforma de cómputo utilizando técnicas de reconfiguración parcial y dinámica

AUTOR: Santiago Córdoba Pellicer

DIRECTOR: Javier Garrigós Guerrero

Septiembre / 2015

Contenido

Introducción ... 3
Reconfiguración: Definición y funcionamiento ... 5
Flujo de trabajo ... 10
Puertos para reconfiguración parcial ... 15
Implementación de un sistema de filtros ... 20
Comprobación y resultados ... 30
Conclusiones ... 35
Anexo: Código de la aplicación FilterApp.c ... 37
Bibliografía ... 43
Agradecimientos ... 44
Exención de Responsabilidad ... 45

Introducción

La utilización de coprocesadores hardware para acelerar la ejecución de algoritmos de cómputo es una técnica comúnmente utilizada en las arquitecturas de las computadoras comerciales. Estos coprocesadores, sin embargo, son de propósito general, en tanto que aceleran las operaciones más comunes en los sistemas de cómputo convencionales. En ciertas ocasiones, no obstante, se hace conveniente disponer de aceleración por hardware de algoritmos de cómputos no convencionales. En estos casos se hace necesario el desarrollo de procesadores de propósito específico, especialmente adaptados a la naturaleza del problema y de los cálculos a realizar.

No obstante, la coexistencia de un número elevado de coprocesadores específicos en un chip se ve limitada por problemas de tamaño y consumo del propio circuito. Para intentar resolver este problema se han estado utilizando con éxito dispositivos programables tipo FPGA en algunas aplicaciones. La reprogramabilidad de las FPGAs nos hace posible tener almacenadas varias configuraciones en una memoria externa y programarlas a voluntad sobre la FPGA cuando convenga, alterando completamente la funcionalidad lógica del dispositivo.

Últimamente han aparecido dispositivos PSoCs (Programmable System on Chip) cuya capacidad de reconfiguración es más sofisticada, e incluso permiten cambiar tan sólo una parte del chip, utilizando lo que se conoce como reconfiguración parcial. Como además podemos reconfigurar una parte del dispositivo mientras el resto sigue su operación, a esta técnica también se le denomina reconfiguración parcial dinámica. La reconfiguración puede realizarla un dispositivo (microprocesador) implementado en el propio PSoC, ya sea en la zona estática o reconfigurable, sin que sea necesario ningún circuito o dispositivo externo adicional. Esta técnica es la denominada auto-reconfiguración parcial.

La (auto)reconfiguración parcial es útil por tanto para aplicaciones que requieren cargar diferentes diseños en la misma área del dispositivo o la flexibilidad para cambiar parte de un diseño sin hacer un reset o reconfigurar completamente el dispositivo. Con esta capacidad, nuevas áreas de aplicación son posibles, en particular, actualizaciones de hardware y reconfiguración en tiempo de ejecución. Estas características son de particular importancia en sistemas fundamentalmente paralelos y con carácter evolutivo, como las redes neuronales, los sistemas neuro-fuzzy y los algoritmos genéticos. También resultan de interés en aplicaciones donde es posible multiplexar en el tiempo la utilización de determinados recursos de forma similar las técnicas software de cambio de contexto que se utilizan en sistemas multitarea, para conseguir una reducción del tamaño del chip, y por tanto del costo y del consumo del mismo. En último extremo, gracias a la auto-reconfiguración, estos SoCs podrían permitir, no solo una mejora de rendimiento y de la eficiencia apreciable, sino una nueva generación de circuitos hardware capaces de re-adaptarse, evolucionar o auto-repararse.

Estas técnicas, no obstante, se encuentran en un temprano estadio de desarrollo, por lo que resulta conveniente evaluar el estado de la técnica y verificar hasta qué punto las herramientas estándar permiten el diseño de aplicaciones en diferentes ámbitos con ciclos de desarrollo relativamente cortos. Por ello, dentro de este marco, el objetivo principal del presente proyecto es el conocimiento de los procedimientos y la evaluación de las herramientas que permitan beneficiarse de las técnicas de auto-reconfiguración parcial en el desarrollo de aceleradores hardware para aplicaciones específicas. Para esto, se hará un estudio del flujo de trabajo propuesto para el uso de la reconfiguración parcial. Más adelante, y utilizando ese mismo flujo de trabajo, tendrá lugar el desarrollo de una arquitectura completa de un sistema de cómputo completo, y la posterior verificación de su funcionalidad

y de la versatilidad de la metodología de desarrollo propuesta. Por último, se concluirá con ejemplos y aplicaciones características que hagan uso de la técnica de la reconfiguración parcial relacionados con los sistemas de telecomunicaciones.

Reconfiguración: Definición y funcionamiento

Los dispositivos FPGA se pueden modelar de forma muy simple como dispositivos de dos capas: la capa lógica (logic layer) y la capa de memoria de configuración (configuration memory layer).

Siguiendo el esquema propuesto en la Figura 1, la memoria de configuración controla la función implementada en la capa lógica. Esta configuración es guardada mediante la carga de

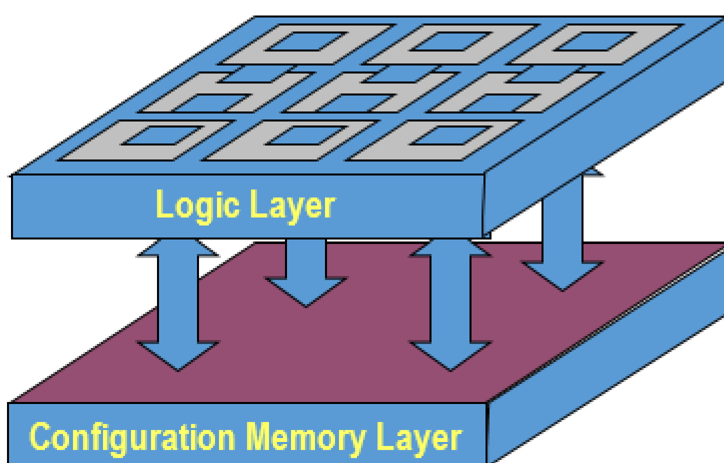

Figura 1: Esquema de capas

bitstreams a la memoria de configuración. Distintos bitstreams implementan distintas funciones lógicas, de modo que cargando a la memoria de configuración distintos bitstreams cambiaremos la función lógica implementada.

El modo de funcionamiento normal de un dispositivo FPGA consiste en la carga de un bitstream inicial al arrancar el dispositivo. Este bitstream cargado al inicio queda fijado durante todo el ciclo de trabajo de la FPGA. Este modo de funcionamiento es el más extendido en el flujo de trabajo tradicional.

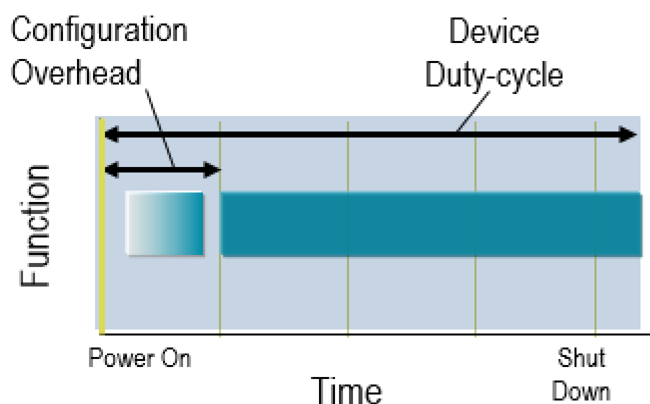

Figura 2: Cronograma de un dispositivo sin reconfiguración

La configuración de un dispositivo reconfigurable tiene distintas etapas, que podemos distinguir en el diagrama de la Figura 3:

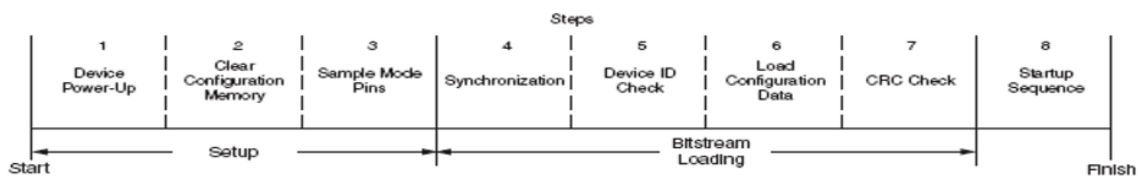

Figura 3: Etapas en la configuración de una FPGA

De forma resumida, los pasos que se siguen en la configuración de una FPGA son los siguientes:

1. Se enciende el dispositivo, activando la alimentación del mismo.
2. Se limpia la memoria de configuración, donde estaba alojada la anterior configuración del dispositivo.
3. Se comprueban los pines de modo (pines M0, M1 y M2), y según estén asignados se utiliza un modo u otro de configuración (JTAG, serial, Memory MAP)
4. Se sincroniza el bus de configuración y se alinean los datos de configuración entrantes con la lógica de configuración interna.
5. Se comprueba en la configuración entrante (bitstream) el ID de dispositivo para el cual ha sido formateado el bitstream, y evitar una configuración errónea.
6. Se carga el bitstream en la memoria de configuración
7. Tras la carga, se puede realizar una comprobación de CRC para comprobar que el bitstream ha sido transmitido a la memoria de configuración sin errores.
8. Por último, se inicia una secuencia de startup en la cual se habilita el modo de usuario

Si ocurre cualquier fallo, tanto en la comprobación del ID del dispositivo como en la comprobación CRC del bitstream transmitido, el dispositivo se bloquea y no permite la entrada al modo de usuario, quedando sin configurar hasta que se reinicie el dispositivo y se vuelva a iniciar el proceso de configuración.

Los dispositivos que permiten reconfiguración, por otro lado, no tienen el bitstream fijado durante su ciclo de trabajo, y pueden cargar nuevos bitstreams completos sin necesidad de reiniciar el dispositivo. No obstante, mientras la reconfiguración tiene lugar, toda la computación se detiene. Una vez cargado el nuevo bitstream el dispositivo continúa funcionando con la nueva función implementada en el bitstream.

Para efectuar tal reconfiguración, se suele partir de la FPGA en modo de usuario, y se activan ciertas señales del puerto de configuración (DONE, PROG), que se va a enviar una nueva configuración, ejecutando de nuevo el proceso de configuración.

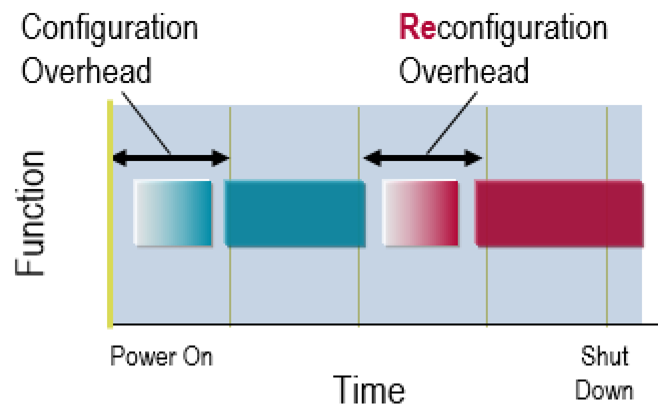

Figura 4: Cronograma de un dispositivo con reconfiguración

Algunos dispositivos permiten una reconfiguración parcial. Estos dispositivos permiten la carga de bitstreams parciales, los cuales alteran solo una parte de la capa lógica y mantienen sin modificar el resto de la capa. De nuevo, la computación se detiene durante la reconfiguración, pero la carga de bitstreams parciales es más rápida, en tanto que sólo se carga la parte del diseño que ha cambiado. Esto supone una ventaja frente a la reconfiguración completa, dado el ahorro de tiempo que supone.

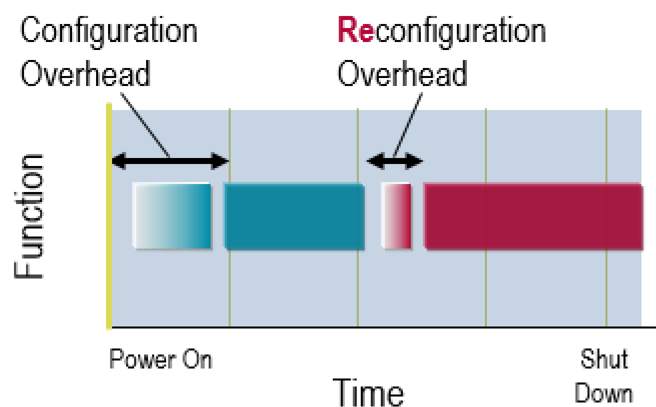

Figura 5: Cronograma de un dispositivo con reconfiguración parcial

Por último, la reconfiguración dinámica permite, al igual que la reconfiguración parcial, la carga de bitstreams parciales que están limitados a la parte de la lógica que se quiere cambiar, pero en este caso la computación únicamente se detiene en el área de la capa lógica que se está alterando, manteniendo en funcionamiento el resto de la capa sin interrupción.

En el caso de la reconfiguración dinámica, no se altera ninguna señal interna del dispositivo, puesto que hacerlo indicaría la transmisión de un bitstream completo, y no de un bitstream parcial, y detendría la computación en todo el dispositivo. En lugar de esto, el puerto de configuración detecta directamente la palabra de sincronización, y comienza el proceso de reconfiguración sin detener el funcionamiento del resto del dispositivo.

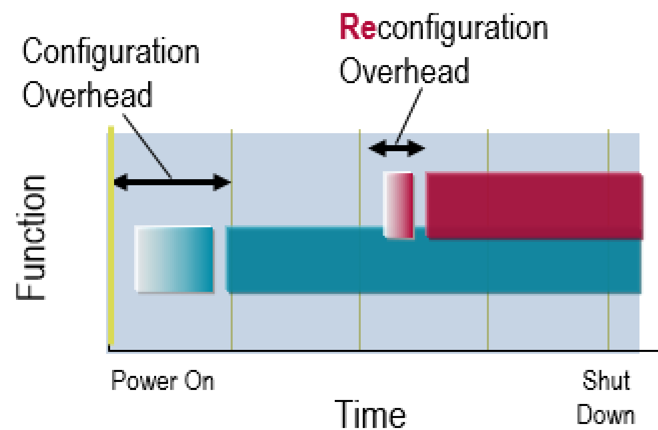

Figura 6: Cronograma de un dispositivo con reconfiguración parcial y dinámica

En este hecho reside el mayor problema asociado a la reconfiguración parcial. Cuando se configura una FPGA de forma normal, el modo de evitar dañar el dispositivo al cargar un bitstream que contenga errores consiste en no permitir que entre en modo de usuario. En la reconfiguración dinámica, la FPGA ya está en modo de usuario, y es posible cargar bitstreams parciales que contengan errores y dañar físicamente el dispositivo si no se detectan y corrigen rápidamente estos errores.

Los errores posibles al reconfigurar una partición generalmente son por transmitir un bitstream implementado para otro dispositivo, o un error en la transmisión del bitstream. En el primer caso, el diseñador es el responsable de que esto no ocurra. En el caso de que la transmisión del bitstream se haya corrompido, pueden ocurrir dos cosas: que el módulo reconfigurable no se haya transmitido bien y la partición quede mal reconfigurada, lo cual se puede arreglar reconfigurando de nuevo la partición, o que se transmita un error en las direcciones y que se reconfigure parte de la lógica estática. Este error, mucho más grave, debe solucionarse reconfigurando de nuevo todo el dispositivo. Es de vital importancia entender que una mala configuración en el dispositivo puede ocasionar cortocircuitos que dañen físicamente el dispositivo de forma irremediable.

Esta técnica de reconfiguración, aunque más restrictiva en cuanto al tipo y tamaño de las regiones reconfigurables, así como más exigente con el diseño del hardware (puesto que hay que definir cuidadosamente “fronteras” que permitan aislar cuidadosamente la parte estática de la que está siendo modificada durante la reconfiguración), es la más interesante y potente de las descritas, y abre la puerta a numerosas posibilidades muy beneficiosas para la implementación de nuevos dispositivos, que son cada vez más exigentes en términos de consumo energético y recursos de hardware.

Además de la evidente posibilidad de cambiar de función de forma dinámica o realizar actualizaciones en los dispositivos mientras se mantienen en funcionamiento, la reconfiguración dinámica permite una multiplexación del hardware en el tiempo. Con dicha multiplexación es posible implementar aplicaciones complejas en dispositivos relativamente pequeños, secuenciando las distintas funciones de la aplicación y realizando una parte de la aplicación cada vez. Los dispositivos pequeños tienen, además, un menor consumo energético y un menor coste de fabricación. Además, la reconfiguración dinámica permite desactivar bloques en momentos de baja actividad, reduciendo aún más el consumo de los dispositivos cuando no sea necesario un alto rendimiento.

Prácticamente todo en una FPGA es reconfigurable: LUTs, flip-flops, BRAMs o bloques DSP, por ejemplo. Pero existen ciertos elementos lógicos que deben mantenerse en estático: bloques modificadores de reloj, buffers de reloj globales, o componentes de E/S son un ejemplo de estos elementos.

En esencia, los proyectos de reconfiguración parcial destacan por tener un diseño estructurado que diferencia la lógica estática de la lógica reconfigurable. El diseño completo tiene una parte de lógica estática y una o más partes de lógica reconfigurable, conocidas como particiones reconfigurables. A la parte del diseño que ocupa las particiones reconfigurables se la conoce como módulo reconfigurables.

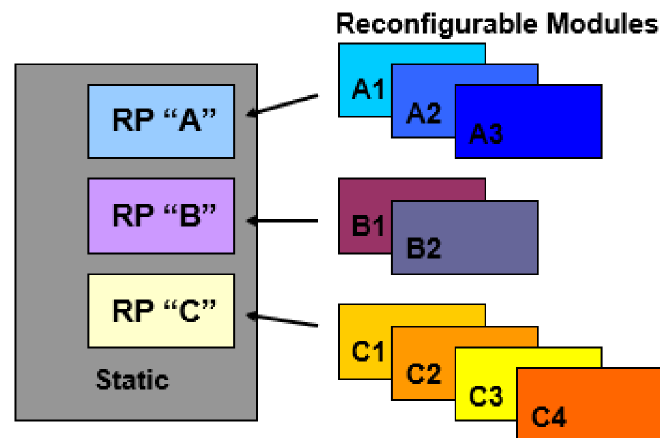

Figura 7: Estructura simple de un diseño con Reconfiguración Parcial

Como ilustra la Figura 7, las particiones reconfigurables (RP) están distinguidas de la lógica estática, y en cada partición puede haber más de un módulo reconfigurable. Estos módulos implementan las distintas funcionalidades que sean requeridas en el diseño. Además, los módulos reconfigurables de una misma partición deben tener la misma interfaz, es decir, mismas entradas y salidas con los mismos nombres, y del mismo ancho de bus.

El hecho de que cada partición reconfigurable permita más de un módulo reconfigurable da pie a distintas configuraciones, diseños completos consistentes en la lógica estática y un módulo reconfigurable para cada partición reconfigurable. Utilizando el caso que nos brinda la Figura 7, un ejemplo de configuración sería el formado por la lógica estática y las particiones reconfigurables "A","B" y "C" con los módulos reconfigurables A2, B1 y C4 respectivamente. Hay tantas configuraciones como combinaciones de módulos reconfigurables existan, incluyendo como se discutirá más adelante las particiones reconfigurables vacías o sin módulo.

Así, reconfigurar el dispositivo se podría considerar como cambiar de configuración, cambiando el módulo de una o varias particiones, pero manteniendo la lógica estática intacta y en funcionamiento, ya que en todas las configuraciones la lógica estática es común e idéntica.

Flujo de trabajo

Aunque los proyectos de reconfiguración parcial pueden desarrollarse con varios programas, tanto de Xilinx, como de terceros (por ejemplo, XST o Synplify), el presente documento se centra en el flujo de trabajo para hacer el desarrollo utilizando la suite de diseño Vivado, que es el software más actual y el recomendado por Xilinx. Por otro lado, el flujo de trabajo de reconfiguración parcial no tiene por el momento soporte en la interfaz de proyectos, y debe llevarse a cabo a través de comandos Tcl en la consola de Vivado.

Cuando trabajamos con reconfiguración dinámica, el flujo de trabajo utilizado es distinto al flujo tradicional. En los proyectos de flujo tradicional (top-down synthesis) solo hay un proyecto de síntesis en el que el diseño, aun teniendo distintas jerarquías entre los módulos, se aplana para optimizar al máximo la implementación. Por otro lado, en el flujo para la reconfiguración dinámica (bottom-up synthesis) existen distintas síntesis separadas, resultando cada una de ellas en una netlist distinta. Se sintetizan de forma separada los distintos bloques que componen el diseño (parte estática, módulos reconfigurables) y se unen sus netlist al acabar, por lo que no existe una optimización en la que se considere el diseño completo. La reconfiguración parcial requiere este tipo de síntesis porque la lógica estática debe ser exactamente la misma en cada una de las configuraciones para que así sean compatibles entre sí.

En los proyectos de reconfiguración parcial, se debe partir del diseño estructurado en lógica estática y particiones reconfigurables. En una primera instancia se hace el diseño del top-level, marcando las particiones lógicas como black boxes o cajas negras, y se sintetiza en lo que será la parte estática del diseño. Utilizando el ejemplo del a Figura 7, se haría una síntesis de la lógica estática, colocando cajas negras en los lugares (celdas) que ocupan las particiones reconfigurables. El resultado se guardará en un checkpoint que contiene, entre otras cosas, el netlist resultante de la síntesis.

Dado que los módulos que ocupan las particiones reconfigurables cambiarán según las necesidades de la aplicación, es de suma importancia crear un aislamiento entre la lógica estática y la parte reconfigurable, utilizando lo que se conoce como lógica de desacople. Usando registros en la parte estática se puede conseguir un adecuado aislamiento, aunque lo más conveniente sería utilizar registros a la entrada y la salida del módulo reconfigurable, creando así un delay registro a registro.

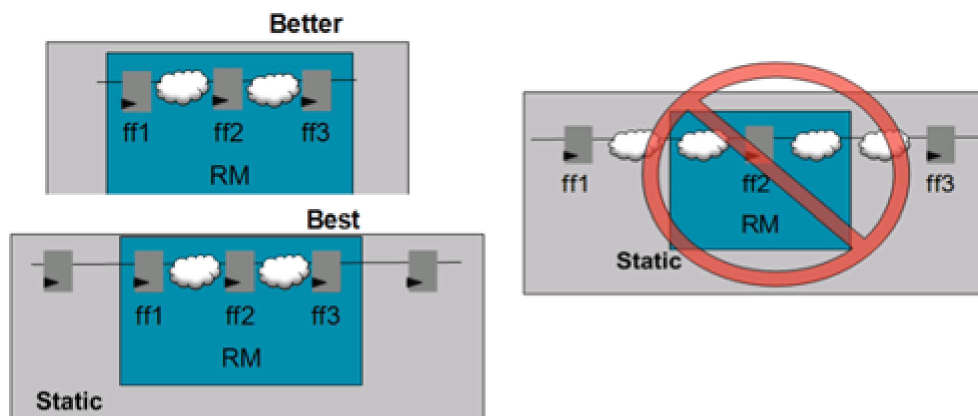

Figura 8: Recomendaciones para el diseño de la lógica de desacoplo

Siguiendo con el flujo de trabajo, se hace el diseño de los módulos de reconfiguración y se sintetizan de forma separada. Cabe destacar que, como se ha denotado antes, los módulos que estén destinados a la misma partición reconfigurable deben tener interfaces idénticas, y todas sus señales de entrada y señales de salida deben ser idénticas. Además, la síntesis para los módulos reconfigurables debe tener incluido un parámetro para que se realice out-of-context, un modo que no añade buffers de inserción en los puertos de entrada y salida, los cuales están ya contemplados en la síntesis del top-level. De nuevo, los resultados de estas síntesis se guardan en checkpoints para posteriormente incluirlos en el netlist general. Siguiendo con el ejemplo de la Figura 7, habría que sintetizar cada uno de los módulos reconfigurables, por separado y en modo *out-of-context*, y guardar el checkpoint de cada uno de ellos.

Antes de implementar el diseño, primero se deben marcar las cajas negras del top-level como particiones reconfigurables, y después definir los recursos físicos y delimitar la superficie en el dispositivo que utilizarán las particiones mediante pblocks. Estos pblocks, al estar vinculados a una partición reconfigurable, sólo pueden contener recursos reconfigurables. Si un pblock asignado a una partición reconfigurable contiene recursos no reconfigurables, dichos recursos no estarán disponibles para la implementación del módulo reconfigurable. Además, los bloques delimitados para cada partición deben contener suficientes recursos para los módulos reconfigurables de mayor complejidad. Existe la posibilidad de que la partición se resetee automáticamente después de la reconfiguración, utilizando la propiedad `RESET_AFTER_RECONFIG`. No obstante, esta propiedad requiere que el pblock que contiene la partición tenga el alto de una región de reloj completa. El ancho, por otro lado, es más flexible.

Una vez que todo está listo para la implementación, se carga en las celdas de las particiones reconfigurables el checkpoint de uno de sus módulos reconfigurables. En el caso de la Figura 7, la primera configuración podría estar formada, por ejemplo, por los módulos A1, B1 y C1. Una vez que se tenga una configuración cargada, se optimiza y se ejecuta el *place and route*.

Tras la primera implementación, se guarda un checkpoint de la configuración (en el ejemplo propuesto, y utilizando como esquema la Figura 9, `config_1.dcp`), que incluye entre otras cosas, la implementación completa de dicha configuración. Por otro lado, se guarda otro checkpoint de las particiones reconfigurables, que contendrán la implementación de los módulos reconfigurables seleccionados en la primera configuración (`RM_A1.dcp, RM_B1.dcp y RM_C1.dcp`).

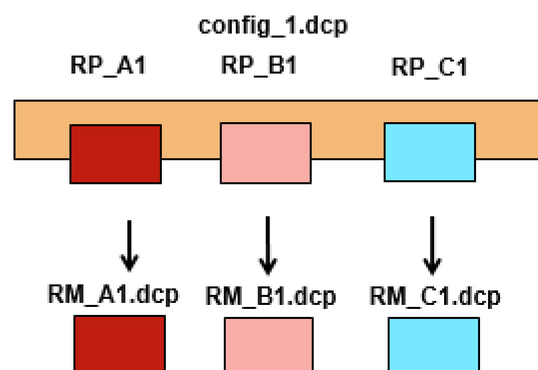

Figura 9: Implementación de la primera configuración

Lo que se pretende ahora es aislar la implementación de la parte estática, para reutilizarla en las implementaciones del resto de configuraciones. De modo que tras la implementación de la primera configuración, se marca de nuevo las particiones reconfigurables como *black boxes*,

se fija el diseño estático (bloqueando el enrutamiento) y se guarda la implementación de la parte estática en otro checkpoint (en la Figura 10, `static.dcp`).

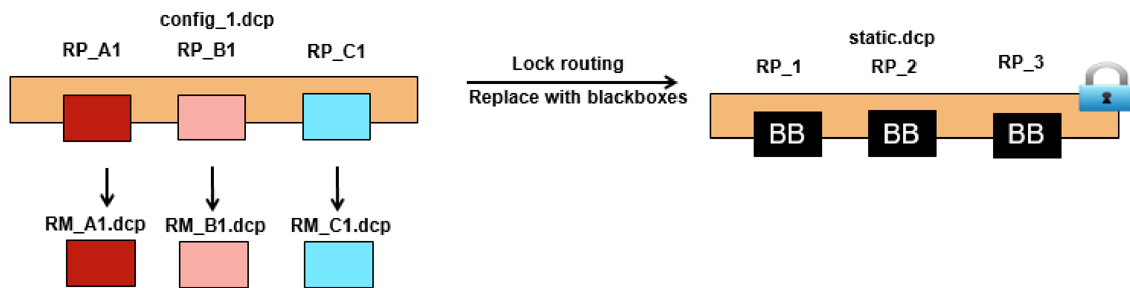

Figura 10: Extracción de la implementación de la lógica estática

Llegado a este punto, para implementar el resto de configuraciones se debe partir de la parte estática previamente aislada, utilizando el checkpoint que contiene la implementación, e insertar en las cajas negras los netlist de los módulos reconfigurables de la segunda configuración (por ejemplo, los módulos A2, B2 y C2). Al implementar esta nueva configuración, se mantiene la misma solución para la parte estática y sólo se efectuará la implementación de los módulos reconfigurables, que posteriormente se guardarán en sus respectivos checkpoints.

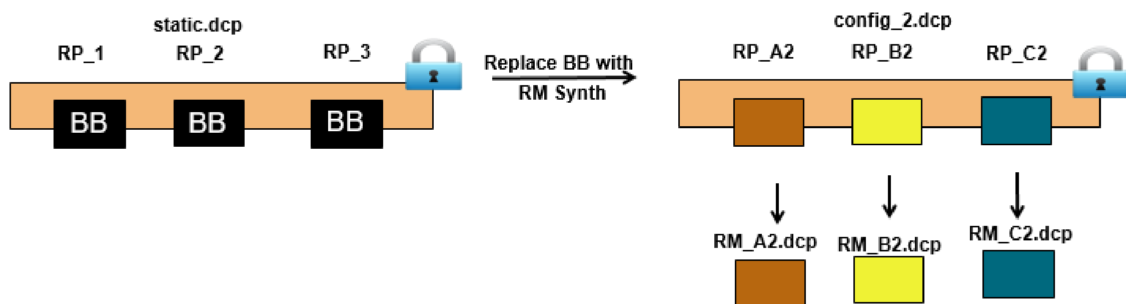

Figura 11: Implementación de la segunda configuración

Esta última operación debe repetirse con cada configuración, hasta tener una implementación de cada uno de los módulos reconfigurables de cada una de las particiones. Si en alguna configuración se repite algún módulo ya implementado, no es necesario volver a implementarlo, sino que se debe importar el checkpoint que contenga la solución para tal módulo. De este modo se mantiene la solución de los módulos reconfigurables entre configuraciones. Además dada la naturaleza de fuera de contexto de estos módulos, el hecho de fijar una solución previamente calculada se traduce en un ahorro de tiempo de cálculo, ya que en cada implementación que se ejecuta sólo se calculan las soluciones de los elementos que no han sido fijados (es decir, que no tienen una solución previa que haya sido importada).

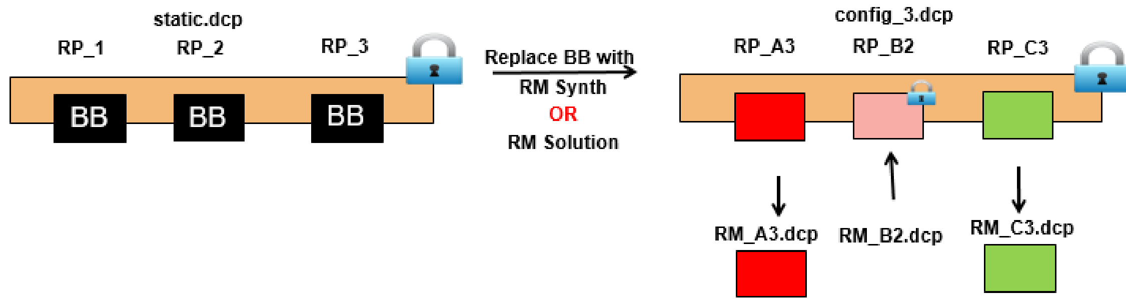

Figura 12: Implementación de la tercera configuración

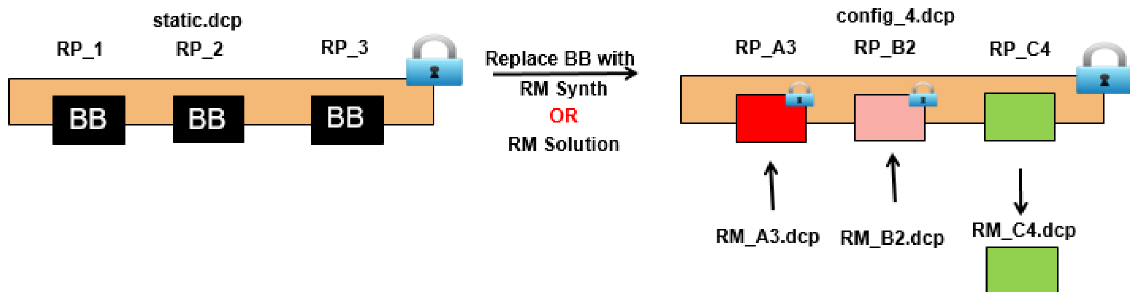

Figura 13: Implementación de la cuarta configuración

De aquí se puede deducir que el número mínimo de configuraciones que se deben implementar lo da el número máximo de módulos que tengan las particiones reconfigurables. Tomando como ejemplo la configuración de la Figura 7, serían necesarias 4 configuraciones para implementar totalmente el diseño, ya que la partición reconfigurable que más módulos tiene es 4.

Por último, se puede hacer una configuración adicional, en la que las particiones reconfigurables no tengan ningún módulo reconfigurable activo. Esta configuración se conoce como configuración vacía o configuración en blanco. Sirve para retirar o "borrar" una configuración previamente cargada en el dispositivo, principalmente para reducir el consumo energético en los momentos en los que los módulos no son necesarios. Esta configuración se implementa sustituyendo las cajas negras por módulos vacíos, de modo que las entradas al módulo queden conectadas a unas LUTs de bloqueo, que fijan la salida a 1 (o 0) independientemente del valor de las entradas, y las salidas estén dirigidas por un cero lógico (GND) o uno lógico (Vcc), según se desee.

Después de implementar todas las configuraciones, se debe verificar que la implementación de la parte estática es consistente y compatible con todas las configuraciones consideradas. El entorno de desarrollo Vivado otorga herramientas para esta operación (DRC o Design Rule Checker).

Tras verificar la compatibilidad entre todas las configuraciones implementadas, tan solo queda generar los bitstreams de las configuraciones y los bitstreams parciales para alterar las configuraciones lógicas. La generación de los bitstreams es automática, y no se distingue de la generación en los proyectos de síntesis tradicionales. La única diferencia es que al generar el bitstream de la configuración cargada en ese momento, además de generar el bitstream para la configuración completa, también se generará un bitstream parcial para cada uno de los módulos reconfigurables que forman dicha configuración. Si se generan los bitstreams de todas las configuraciones, se tendrán los bitstreams parciales de todos los módulos

reconfigurables del diseño. Si no se requiere el bitstream de alguna configuración completa, existe la opción de generar únicamente los bitstreams parciales de dicha configuración, sin generar el bitstream del top level. Cabe destacar que estos bitstreams, aun siendo resultado de un desarrollo de reconfiguración parcial, pueden encriptarse o comprimirse como cualquier otro bitstream.

Puertos para reconfiguración parcial

Finalmente, sólo queda cargar en el dispositivo el bitstream completo de una de las configuraciones, y según los requisitos de la aplicación, cargar los bitstreams parciales para reconfigurar las particiones como sea necesario. Los bitstreams parciales se cargan de la misma forma que los bitstreams completos, de modo que un microprocesador externo al dispositivo FPGA puede transmitir los bitstreams parciales desde un almacenamiento de memoria a través de uno de los puertos de configuración externos del dispositivo (Figura 14, sistema de la derecha) tales como Select MAP, serial o JTAG. Pero existe otra opción más interesante, y es que esta misma operación la efectúe una máquina de estado, un microcontrolador o cualquier otro sistema implementado en el mismo dispositivo FPGA, que transmita los bitstreams parciales desde la memoria donde se almacenen hacia algún puerto interno de configuración, ya sea ICAP o PCAP (este último únicamente disponible en los dispositivos Zynq). De este modo, no es necesaria la intervención de ningún sistema externo a la FPGA, y se estaría produciendo lo que se conoce como auto-reconfiguración parcial y dinámica (Figura 14, sistema de la izquierda).

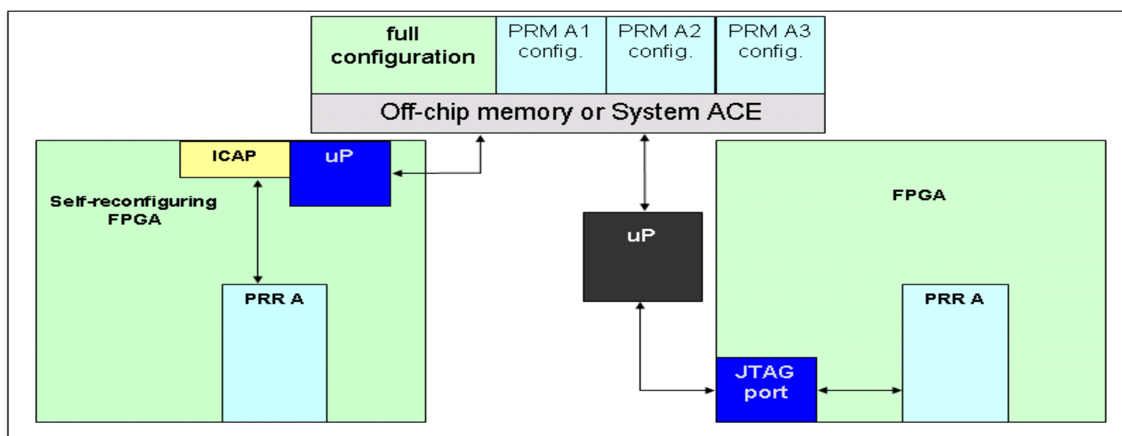

Figura 14: Soluciones para la carga de bitstreams parciales (self reconfiguration a la izq. y off-board reconfiguration a la drch.)

El tiempo de reconfiguración es el tiempo que se tarda en reconfigurar una partición reconfigurable, transmitiendo el bitstream parcial de uno de sus módulos reconfigurables a través de uno de los puertos de configuración del dispositivo. Este tiempo está definido, por un lado por el tamaño del bitstream parcial, y por otro lado por el ancho de banda del puerto de configuración que se utilice para la transmisión.

El tamaño del bitstream parcial depende del tamaño del pblock definido para alojar la partición reconfigurable. Se puede pensar en las particiones reconfigurables como FPGAs dentro de la FPGA: cuanto mayor es un dispositivo reconfigurable, mayor número de componentes y recursos, y mayor tamaño tienen sus bitstreams. Con las particiones reconfigurables sucede exactamente lo mismo, cuanto mayor sea la superficie que delimita el pblock, más recursos contendrá y mayor tamaño tendrá el archivo de configuración (bitstream parcial) de dicha superficie.

El ancho de banda del puerto utilizado depende del ancho de bus y de la frecuencia de reloj utilizada en la transmisión. La Figura 15 contiene una tabla con los puertos de

reconfiguración, tanto internos como externos, y sus características:

Configuration Mode	Max Clock Rate	Data Width	Max Bandwidth
SelectMap / ICAP	100 MHz	32-bit	3.2 Gbps
Serial Mode	100 MHz	1-bit	100 Mbps
JTAG	66 MHz	1-bit	66 Mbps

Figura 15: tabla de comparación entre puertos de configuración

Para hacer uso de la técnica más potente descrita hasta ahora en el documento, la auto-reconfiguración dinámica, será necesario utilizar uno de los puertos internos de configuración del dispositivo, a saber, ICAP o PCAP.

El puerto PCAP (Processor Configuration Access Port) es un puerto de configuración interna disponible solamente en los dispositivos Zynq, y accesible desde el Processing System (PS) con el que cuenta esta familia de dispositivos. El puerto PCAP es el responsable de la configuración de la FPGA (en los dispositivos Zynq, Lógica Programable o PL) cuando utilizamos una imagen de arranque desde una tarjeta SD. El bootloader del dispositivo (FSBL, First Stage BootLoader), se encarga de configurar la lógica programable con el bitstream indicado en la imagen de arranque. Por ello no es de extrañar que sea posible transmitir los bitstreams parciales directamente desde el microcontrolador con el que cuenta en el Processing System. Para ello, el entorno de desarrollo de software de Vivado cuenta con unas librerías bien documentadas, que permiten utilizar este puerto de configuración desde una aplicación.

Por otro lado, el puerto ICAP (Internal Configuration Access Port) no es exclusivo de la familia Zynq, sino que está disponible en muchos otros dispositivos. Es idéntico al puerto Memory MAP, pero accesible desde la lógica programable de la FPGA. Por ello, se puede usar por máquinas de estado o microprocesadores definidos en la lógica programable. También puede utilizarse con el Processing System, a través de alguna interfaz PS-PL (interfaz entre el Processing System y la lógica programable) y definiendo algún controlador para el ICAP.

La principal ventaja del ICAP frente al PCAP es que permite una reconfiguración utilizando un microprocesador implementado en la lógica programable (como MicroBlaze) o una máquina de estado, liberando de este modo al Processing System de las labores de reconfiguración. La desventaja, en este caso es obvia. Para utilizar el puerto ICAP es necesario hardware extra en la lógica programable, ya sea un controlador o máquina de estado, o sea una interfaz entre el Processing System y la lógica programable.

Como cabe esperar, las distintas opciones de hardware para utilizar el puerto ICAP permiten ajustar los diseños a las necesidades de la aplicación. La opción más simple es una máquina de estado que controle la escritura del bitstream parcial al puerto, como la de la Figura 16:

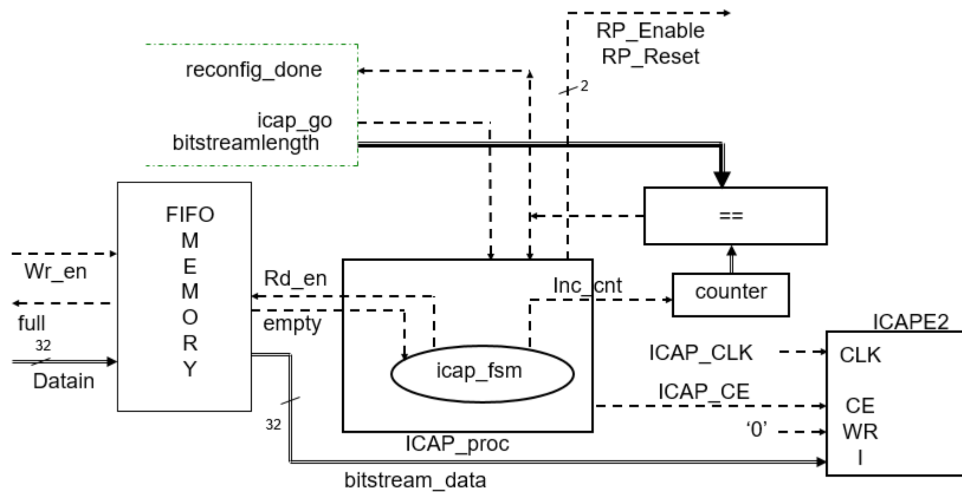

Figura 16: Máquina de estado para control del puerto ICAP

En esta solución, el bitstream se escribe en una FIFO y la máquina de estado controla el estado de la transmisión mediante un contador, que compara con la longitud del bitstream. Además, cuenta con señales de enable y reset para controlar, por un lado el correcto desacoplo de la lógica reconfigurable y la lógica estática durante la configuración, y una inicialización en un estado conocido por el otro. Esta solución, aunque muy eficiente y simple de implementar, necesita un mecanismo para transferir el bitstream a la FIFO y la longitud del mismo al registro habilitado para ello.

Como curiosidad, utilizar una solución como la de la máquina de estado de la Figura 16, es decir, un controlador ICAP personalizado, es la única forma de transmitir bistreams parciales que han sido encriptados con un algoritmo custom, y no con los algoritmos soportados por los puertos de configuración.

Adicionalmente, Xilinx otorga a los diseñadores que deseen utilizar el puerto ICAP en sus proyectos de Vivado unas soluciones a través de su IP Integrator. Por un lado, contamos con el Core AXI HWICAP, descrito en la Figura 17, que utilizando una conexión AXI4-Lite Slave se conecta al Processing Sysem del dispositivo Zynq y permite el uso del puerto ICAP desde la aplicación, haciendo uso de una librería (xhwicap.h) para interactuar con el Core.

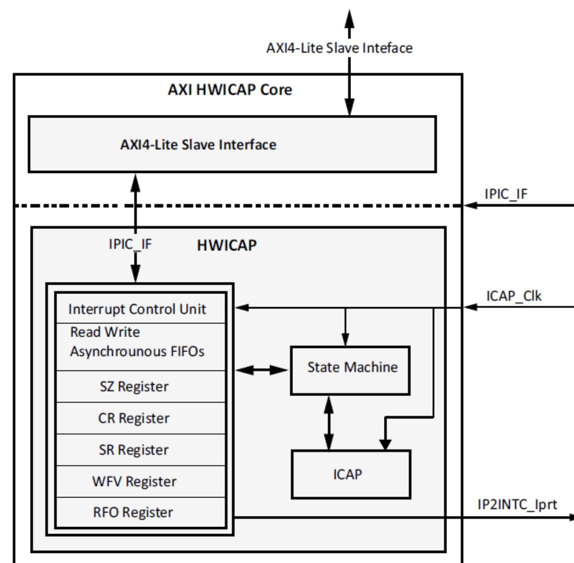

Figura 17: Esquema del Core AXI HWICAP

Por otro lado, y quizás la opción más interesante, desde la versión 2015.1 de Vivado está el Core Partial Reconfiguration Controller (PRC). Este Core tiene varias opciones de configuración, accesibles tanto desde la línea de comandos como desde la interfaz gráfica de Vivado.

Este controlador se hace cargo de todas las etapas de la reconfiguración parcial, desde la lectura del bitstream parcial hasta el desacoplo y reseteo necesarios para la reconfiguración de una partición reconfigurable. El controlador define lo que se conoce como un Virtual Socket (VS, descripción en la Figura 18), cada uno con su Virtual Socket Manager (VSM, ejemplo en la Figura 19), por cada partición reconfigurable. Se puede definir un máximo de 32 Virtual Sockets. Cada VSM puede controlar hasta 128 módulos reconfigurables, y hasta 512 triggers remapeables tanto de hardware como de software para controlar las reconfiguraciones. Además tiene interesantes opciones, como la interacción con el software (mediante líneas de interrupción hacia el microcontrolador del sistema) para informar de que ha ocurrido un evento de reconfiguración y así poder alterar el flujo del programa.

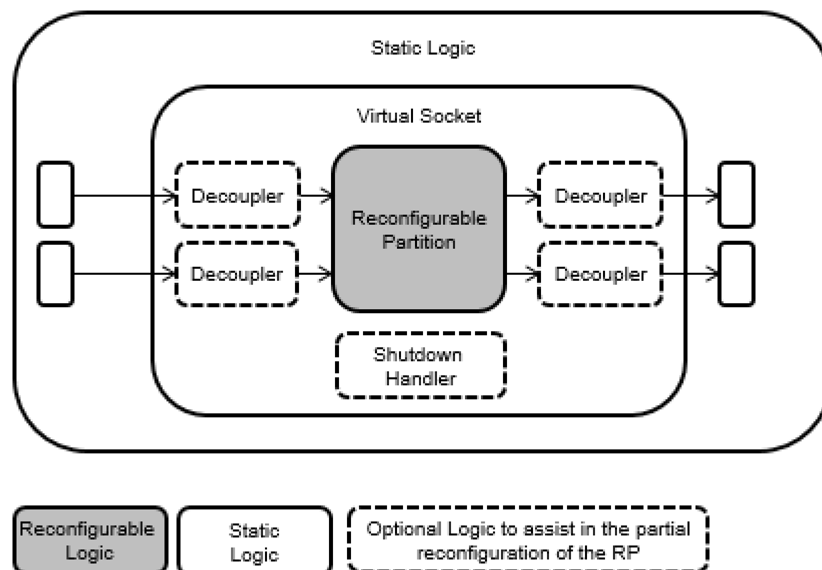

Figura 18: Esquema de un Virtual Socket

El PRC en sí se comporta como un pequeño microcontrolador, y cuenta con bancos de registros de configuración para cada uno de los VSM definidos, en los que la configuración y los triggers definidos se almacenan. Estos bancos de memoria son accesibles a través de una interfaz AXI4 (AXI4-Stream si lo que se quiere es utilizar los canales específicos de cada VSM, AXI4-Lite si se quiere utilizar la interfaz de registros), y se puede modificar la configuración de cada VSM por separado en tiempo de ejecución. Por último, una interfaz Master de AXI4-MM conecta el Core a la memoria donde están alojados los bitstreams, para así poder transmitirlos por ICAP.

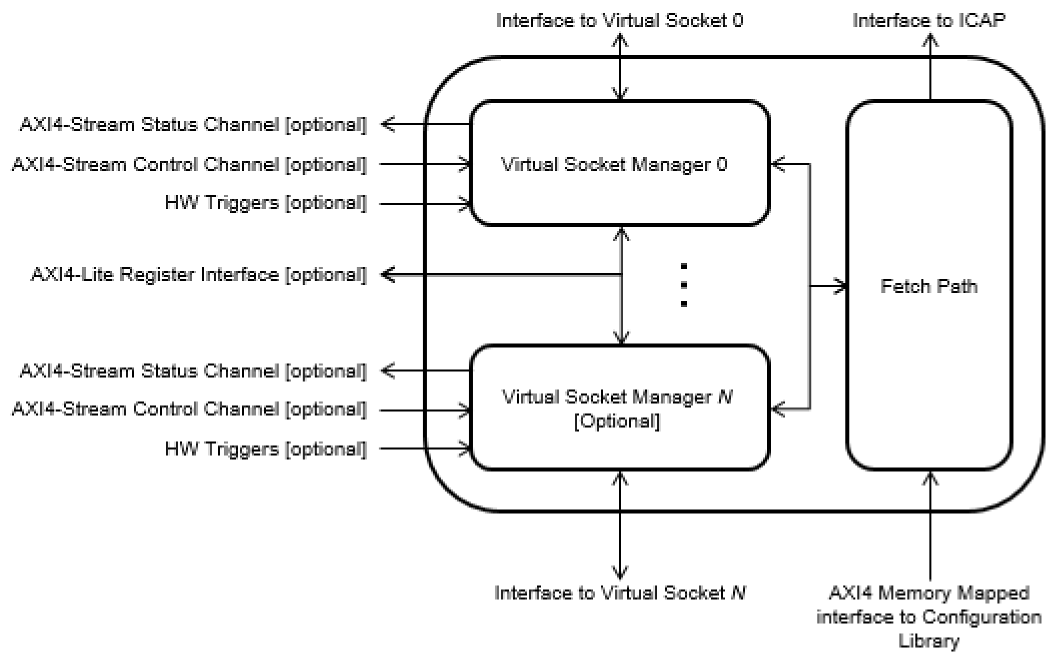

Figura 19: Diagrama de bloques del PRC

Implementación de un sistema de filtros

A continuación se documentará con detalle el proceso de desarrollo de un sistema de filtrado utilizando la técnica de reconfiguración parcial. Como el objetivo del presente documento radica en la técnica de reconfiguración parcial y no en la complejidad del tratamiento digital de señales, se ha considerado dos filtros sencillos para el sistema.

El hardware utilizado ha sido una placa de evaluación ZedBoard (Zynq Evaluation and Development Board), y el software para el flujo de trabajo ha sido la suite de diseño Vivado, tanto en modo de interfaz gráfica como en modo línea de comandos.

El diseño propuesto, ilustrado en la Figura 20, está compuesto por el sistema de procesado (PS) que contiene un ARM Cortex-A9, conectado a los periféricos de memoria DDR y periférico UART. Dos interfaces AXI (una en modo Slave y otra en modo Master) conectan el procesador a la lógica programable (PL). Es en esta lógica donde se alojará la partición reconfigurable que contendrá los filtros, así como un adaptador AXI IPIC, cuya función es interpretar las señales AXI y crear una interfaz para los filtros. Además, en la parte programable estará alojado el controlador de reconfiguración parcial (PRC), y los bloques de interconexión AXI necesarios.

En resumen, el diseño cuenta con un sistema de procesado, un controlador para la configuración parcial y una partición reconfigurable que contendrá dos ejemplos de filtros.

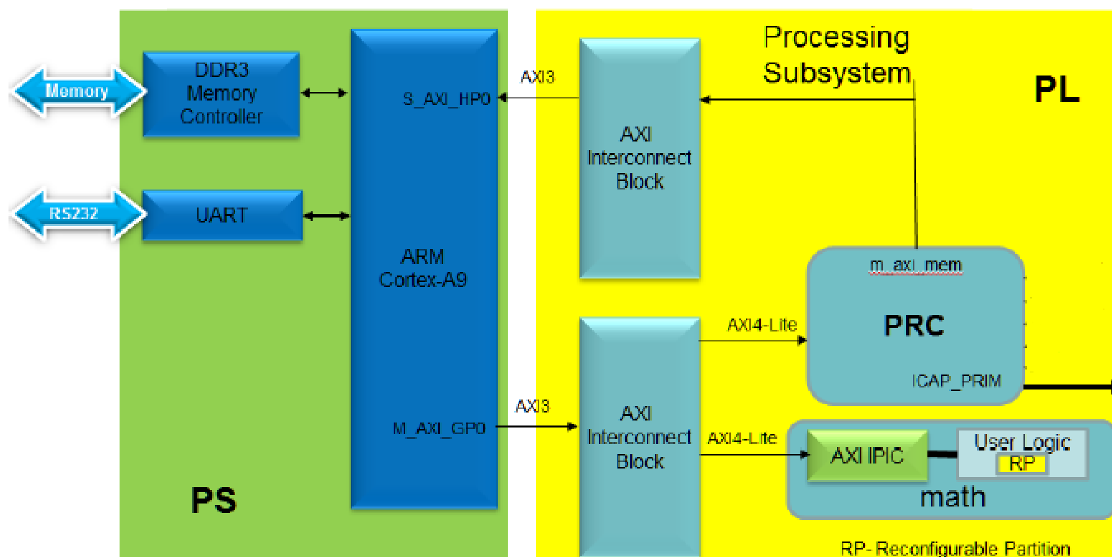

Figura 20: Esquema del diseño propuesto

Para los proyectos de reconfiguración parcial es recomendable tener una estructura de directorios determinada para facilitar el uso de los comandos y los scripts. Un ejemplo de una correcta estructura de carpetas se muestra en la Figura 21, en la que cada carpeta convenientemente nombrada según lo que contiene. En los casos en los que se tengan archivos correspondientes a la parte estática o a la reconfigurable, se separan en subdirectorios para facilitar la organización.

La carpeta demo es donde está alojado el proyecto de Vivado, que previamente hemos creado utilizando el asistente de creación de proyectos en la interfaz gráfica.

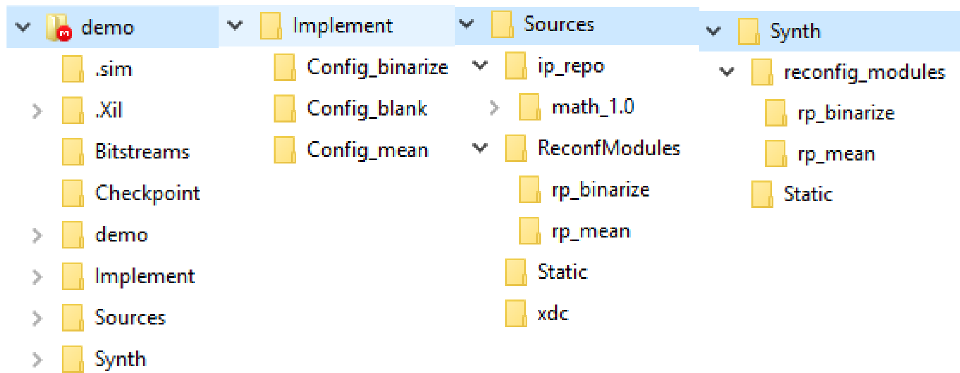

Figura 21: Estructura de carpetas en un proyecto de RP

Nota: para recrear todos los pasos descritos a partir de ahora basta con ejecutar el script `run_all.tcl` adjunto a este documento a través del comando

```
cd <ruta_hacia_la_carpeta_demo>
source ./run_all.tcl
```

Utilizando la interfaz gráfica de Vivado, creamos un nuevo diseño de bloques (opción *Create New Block Design*), añadimos nuestro repositorio de bloques IP alojado en la carpeta `./Sources/ip_repo` y creamos el diseño de bloques de la Figura 23.

Los bloques utilizados en el diseño son:

- ZYNQ Processing System: El bloque IP ha sido configurado para prescindir de todos los periféricos a excepción del UART y del controlador de la tarjeta SD, y se han habilitado las interfaces AXI `S_AXI_HP0` y `M_AXI_HP0` (Figura 22)

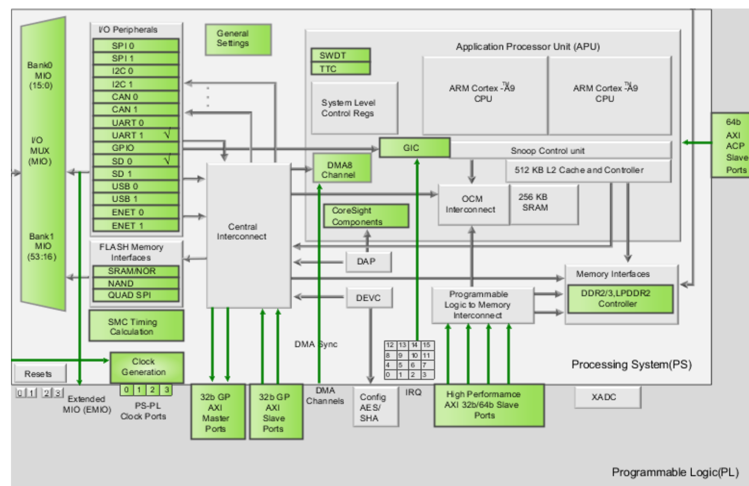

Figura 22: Configuración del ZYNQ Processing System

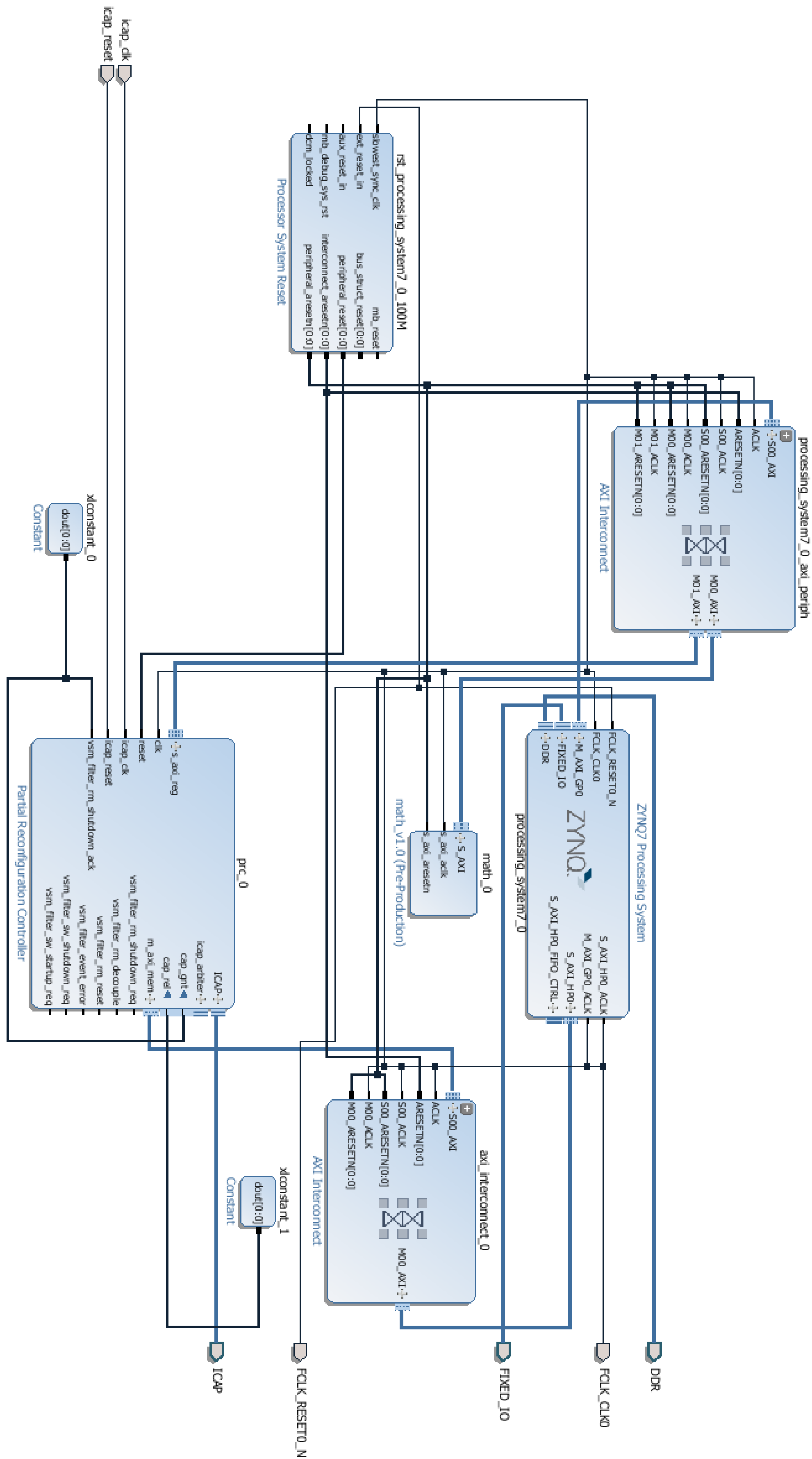

Figura 23: Diseño de Bloques completo

- Math_v1.0 (Pre-Production): Este bloque IP contiene el AXI IPIC necesario para transmitir los datos entre la interfaz AXI y la partición reconfigurable que hay en su interior. En la Figura 24 se muestra a un lado la jerarquía del bloque IP, y al otro lado la descripción de la interfaz del bloque reconfigurable (que está de momento sin definir).

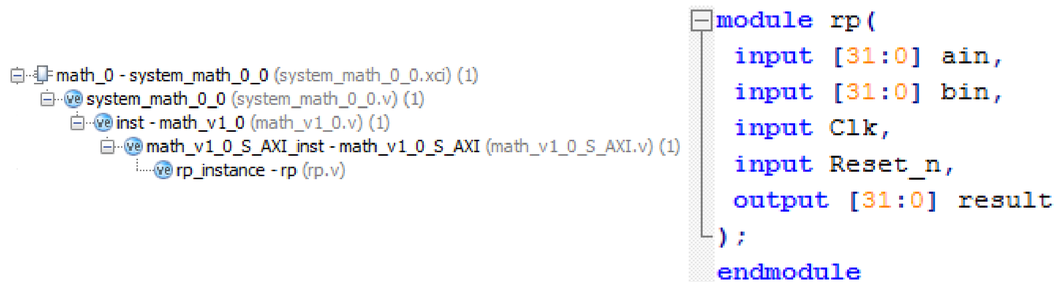

Figura 24: Estructura del bloque Math e interfaz de la RP

- Partial Reconfiguration Controller: Este bloque IP solo se encuentra disponible a partir de la versión 2015.1 de Vivado. Su configuración, bastante intuitiva, puede verse en la Figura 25. A la izquierda aparece la configuración global del PRC, en el que destacamos la habilitación de la interfaz AXI, y a la derecha aparecen las opciones de los VSM. Como sólo es necesaria una partición reconfigurable, creamos un solo VSM, llamado *filter*, que tendrá espacio para 4 módulos reconfigurables. Aunque solamente se generarán 3 módulos (los dos filtros y el módulo vacío), el número de RMs debe ser potencia de 2. Por último, en la parte de abajo se muestran las opciones de trigger del VSM. Estos valores, de nuevo, deber ser potencia de 2, e indican cuántos triggers se crearán para ese VSM, y cuántos de ellos serán triggers hardware.

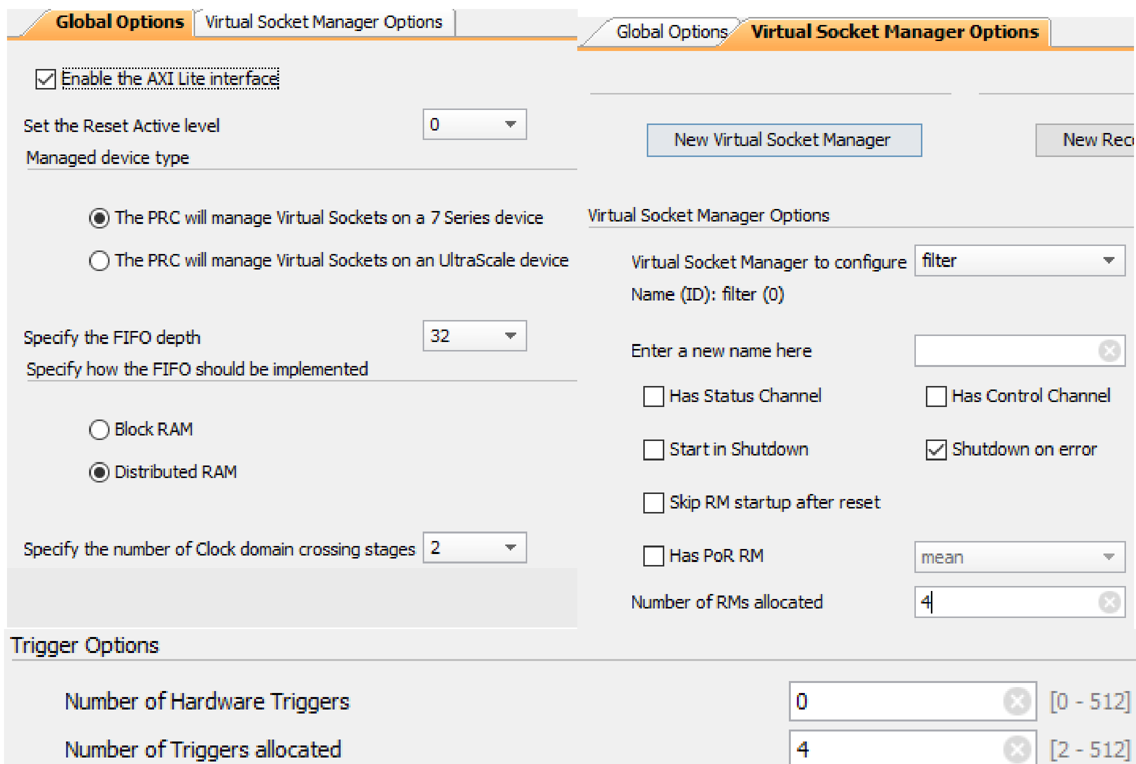

Figura 23: Ajustes Globales del PRC y opciones del VSM 'filter'

El resto de bloques (AXI Interconnect, Processor System Reset…) son creados por el asistente de conexiones, al que es posible acudir cuando se añaden nuevos bloques al diseño para que asigne correctamente las conexiones compatibles entre bloques.

Es necesario añadir los puertos que aparecen en el diagrama de bloques (`ICAP`, `icap_clk`, `icap_rst`, `fclk_reset_n`, etc.) puesto que son los puertos de entrada y salida del diseño de bloques. Estos puertos se conectarán a través del recurso ICAPE2, instanciado en el top level, al puerto de configuración ICAP, como se verá más adelante.

Por último, los bloques de constantes que se añaden al diseño son necesarios para el correcto funcionamiento del PRC en las circunstancias de la aplicación propuesta:

- 1 lógico en los puertos `cap_gnt` y `vsm_filter_rm_shutdown_ack`:
    - El puerto `cap_gnt` es uno de los puertos de arbitración para el puerto ICAP. Como se ha explicado anteriormente, los puertos PCAP e ICAP no pueden utilizarse al mismo tiempo, y este puerto es utilizado por el árbitro de ICAP para marcar si se tiene acceso o no al puerto de configuración. Puesto que no es necesario utilizar el puerto PCAP, el puerto ICAP estará siempre disponible, así que se puede asignar al puerto un 1 lógico constantemente.
    - El puerto `vsm_filter_rm_shutdown_ack` es utilizado por el VSM para saber cuándo tiene el permiso del sistema de procesamiento para iniciar la reconfiguración. Como en el sistema propuesto no será necesario hacer una desactivación del módulo en software, se puede asignar al puerto un 1 lógico constantemente para que efectúe la reconfiguración tan pronto como reciba un trigger.
- 0 lógico en el puerto `cap_rel`: De nuevo, este puerto forma parte del sistema de arbitraje del puerto ICAP. Concretamente, indica al PRC cuándo hay una petición de otro dispositivo para acceder a la configuración del dispositivo. Como en el diseño propuesto no ocurrirá tal cosa, se puede asignar al puerto un 0 lógico constantemente.

Una vez creado el diseño de bloques, se crea un wrapper que contenga el diseño y que generará un archivo Verilog con todo el diseño de bloques. A continuación se añade a los archivos de fuente el archivo `top.v`, el archivo Verilog que describe el top level del sistema propueso. Este archivo `top.v`, localizado en `./Sources/Static/top.v` simplemente contiene una instancia del diseño de bloques generado, y una instancia del recurso ICAP (Figura 26) que conectará los puertos del diseño de bloques con el puerto de configuración ICAP.

```
ICAPE2
    # (
      .DEVICE_ID(32'h23727093), // Device ID code for 7z020 of ZedBoard
      .ICAP_WIDTH(32)    // Input and output data width to be used with the ICAPE2.
    )
    ICAPE2_inst (
      .O(ICAP_o), // 32-bit output: Configuration data output bus
      .CLK(FCLK_CLK0), // 1-bit input: Clock Input
      .CSIB(ICAP_csib), // 1-bit input: Active-Low ICAP Enable
      .I(ICAP_i),     // 32-bit input: Configuration data input bus
      .RDWRB(ICAP_rdwrb) // 1-bit input: Read/Write Select input
  );
```

Figura 24: Instancia del recurso ICAPE2

Tras añadir el top level, el diseño está listo para sintetizar. Ejecutamos la síntesis, y guardamos un checkpoint que contiene el resultado de la síntesis a nuestro directorio de Checkpoints mediante el comando

```
write_checkpoint ./Checkpoint/top.dcp –force
```

Este comando escribe un archivo .dcp conteniendo el estado actual del proyecto (a nivel de síntesis, implementación…). El flag `–force` indica a la línea de comandos que tiene permiso para sobrescribir el archivo en el caso de que ya exista.

El próximo paso en el flujo de trabajo es sintetizar, de forma fuera de contexto, los distintos módulos reconfigurables de la partición reconfigurable. Para ello, ejecutamos el siguiente script:

```
read_verilog ./Sources/ReconfModules/rp_mean/rp_mean.v
synth_design -mode out_of_context -top rp -part xc7z020clg484-1
write_checkpoint Synth/reconfig_modules/rp_mean/filter_synth.dcp -
force
close_design
read_verilog ./Sources/ReconfModules/rp_binarize/rp_binarize.v
synth_design -mode out_of_context -top rp -part xc7z020clg484-1
write_checkpoint
./Synth/reconfig_modules/rp_binarize/filter_synth.dcp -force
close_design
```

En esencia, el script carga los archivos fuente de los filtros (un filtro de binarización y un filtro de media, en este caso), sintetiza el diseño marcando la celda `rp_instance` como top mediante el flag `–top rp` y activa la síntesis *out-of-context* mediante el flag `–mode out_of_context`. El flag `-part xc7z020clg484-1` indica el dispositivo para el cual se sintetizará el diseño. Finalmente, se guardan sendos checkpoints de los módulos sintetizados en su carpeta correspondiente.

El siguiente paso en el flujo de trabajo es marcar la celda `rp_instance` como partición reconfigurable y crear el pblock en el que estará contenida. Para ello, ejecutamos los comandos

```
open_checkpoint Synth/Static/top.dcp
read_checkpoint -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Synth/reconfig_modules/rp_binarize/filter_synth.dcp
```

```
set_property HD.RECONFIGURABLE 1 [get_cells
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance]
```

El primer comando abre el checkpoint del top level sintetizado. El siguiente lee el checkpoint de uno de los módulos reconfigurables sintetizados (en este caso, la del filtro de binarización) y lo aloja en la celda `rp_instance` con el flag `-cell`. Con este comando se combinan los netlist del top level y del módulo reconfigurable, y la celda `rp_instance` deja de ser una caja negra.

El último comando marca la celda `rp_instance` como reconfigurable. Es en este punto cuando la licencia para la característica de Partial Reconfiguration es verificada en el archivo de licencias.

Para marcar el pblock que contendrá la partición reconfigurable, no hay más que hacer click derecho en la celda de `rp_instance` en la ventana de netlists, y añadir un pblock a la celda. La forma más sencilla para definir el pblock es dibujándolo. Esta opción permite seleccionar sobre un esquema del dispositivo los recursos que debe incluir el pblock. Tras comprobar los recursos requeridos por cada uno de los módulos, se debe escoger un área que satisfaga las necesidades de todos los módulos reconfigurables.

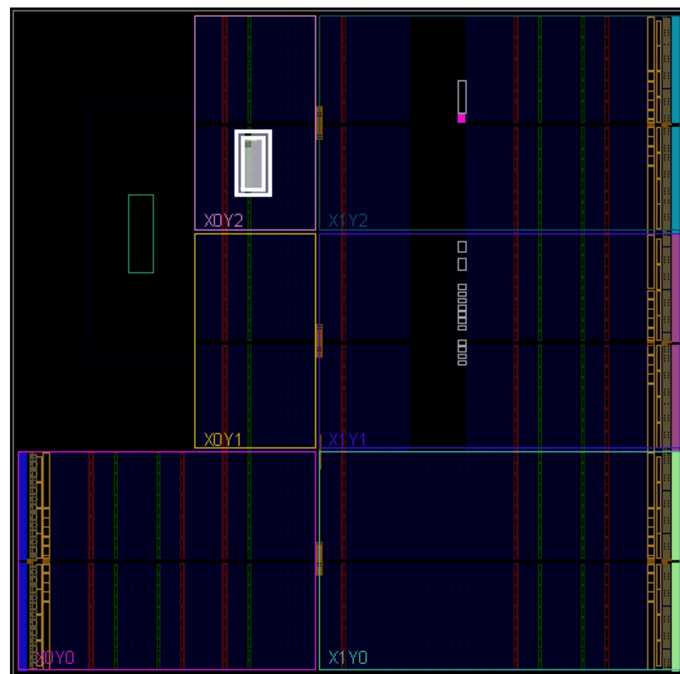

Figura 25: Área escogida en el diseño propuesto

Una vez definido el pblock, podemos implementar la configuración utilizando los comandos

```
opt_design
place_design
route_design
```

```
write_checkpoint -force
Implement/Config_binarize/top_route_design.dcp
report_utilization -file
Implement/Config_binarize/top_utilization.rpt
write_checkpoint -force -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Checkpoint/binarize_route_design.dcp
```

Como resultado, por un lado se obtiene un checkpoint de la implementación de toda la configuración y un informe de utilización de recursos, y por otro lado un checkpoint con la implementación únicamente de la celda `rp_instance`, para incluir en nuevas configuraciones.

Lo siguiente es extraer la implementación de la parte estática, añadiendo cajas negras y bloqueando el diseño. Esto se lleva a cabo mediante los comandos

```
update_design -cells
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance -black_box
lock_design -level routing
write_checkpoint -force Checkpoint/static_route_design.dcp
close_project
```

El primer comando sustituye la celda `rp_instance` por una caja negra, por el flag –`black_box`. Una vez sustituido, se fija el diseño a nivel de routing con el segundo comando y se guarda el checkpoint de la implementación de la lógica estática.

Para crear la segunda configuración y la configuración vacía se carga este último checkpoint de la lógica estática, se sustituye la caja negra por el filtro de media (segunda configuración) o por buffers (configuración vacía) y se sigue el mismo proceso que con la primera configuración.

Para la segunda configuración:

```
open_checkpoint Checkpoint/static_route_design.dcp
read_checkpoint -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Synth/reconfig_modules/rp_mean/filter_synth.dcp
opt_design
place_design
route_design
write_checkpoint -force Implement/Config_mean/top_route_design.dcp
```

```
write_checkpoint -force -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Checkpoint/mean_route_design.dcp
close_project
```

Para la configuración vacía:

```
open_checkpoint Checkpoint/static_route_design.dcp
update_design -buffer_ports -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
place_design
route_design
write_checkpoint -force Implement/Config_blank/top_route_design.dcp
```

Nótese el flag `–buffer_ports` en el comando `update_design` del script para la configuración vacía, que indica la inclusión de LUTs al diseño para dejar vacía la partición reconfigurable.

Una vez implementadas las configuraciones, comprobamos que son compatibles mediante el comando

```
pr_verify -initial Implement/Config_binarize/top_route_design.dcp -
additional {Implement/Config_mean/top_route_design.dcp
Implement/Config_blank/top_route_design.dcp}
```

que comprobará si las implementaciones son compatibles entre sí y no se han generado conflictos entre ellas.

Lo último que queda es generar los bitstreams, que se hacen con los siguientes comandos:

```
open_checkpoint Implement/Config_binarize/top_route_design.dcp
write_bitstream -file Bitstreams/Config_binarize.bit -force
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit
"up 0 Bitstreams/Config_binarize_pblock_rp_instance_partial.bit"
Bitstreams/binarize.bin -force
close_project
```

El segundo comando escribe los bitstreams generados por la utilidad, y el tercer comando formatea el bitstream parcial generado para que sea compatible con Select MAP (flag `–interface SMAPx32`) y se desactiva el swap a nivel de bit con el flag `–disablebitswap`, puesto que el puerto ICAP requiere swap a nivel de palabra.

Estos comandos se repiten para cada configuración, hasta que se generan los bitstreams de todas las configuraciones (en formato .bit) y los bitstreams parciales (formateados para SelectMap, en formato .bin) de todos los módulos reconfigurables, incluido el módulo vacío.

Comprobación y resultados

Una vez el diseño propuesto ha sido implementado, lo único que queda es escribir una aplicación para el sistema de procesamiento que utilice los periféricos diseñados. Partiendo del proyecto en Vivado, se debe exportar el hardware al SDK (*File > Export > Export Hardware*) y abrimos el SDK. Una vez en el entorno de desarrollo, hay que crear un paquete de soporte para la placa (*File > New > Board Support Package*) que incluya las librerías para sistemas de ficheros FAT. Utilizando este paquete de soporte, podemos desarrollar una aplicación sencilla para probar los filtros implementados. En el anexo se adjunta el código de la aplicación utilizada para comprobar el diseño.

Para ejecutar el código en la placa de desarrollo desde una tarjeta SD es necesario crear una imagen de arranque. Para ello, lo primero que hay que hacer es crear una aplicación de bootloader (desde el asistente de nueva aplicación, utilizando el preset de Zynq FSBL), y luego utilizar la herramienta de creación de imágenes de arranque para Zynq (*Xilinx Tools > Create Zynq Boot Image*). En esta herramienta se deben incluir los binarios del bootloader, el bitstream de la configuración inicial deseada, y los binarios de la aplicación de comprobación.

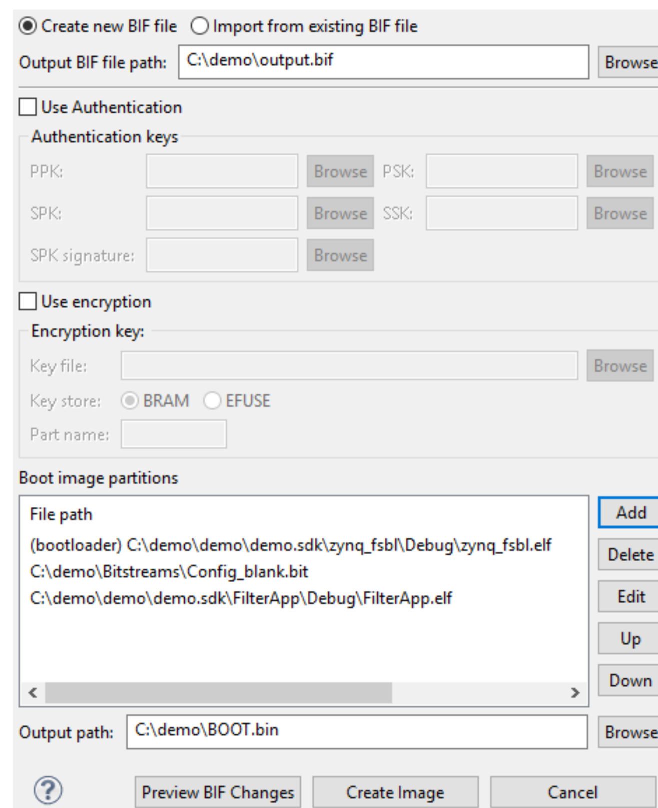

Figura 26: Herramienta de creación de imágenes de arranque

El resultado será un archivo `BOOT.bin` que se deberá copiar, junto con los bitstreams parciales, a la tarjeta SD.

La aplicación, en términos generales, sigue los pasos detallados a continuación. Lo que se incluye en las cajas de código es la confirmación a través del puerto serie del funcionamiento de la aplicación a medida que se ejecuta.

Los pasos que sigue la aplicación son los siguientes:

1. Transmite los bitstreams parciales alojados en la tarjeta SD (sistema de archivos FAT) hacia la memoria DDR utilizando la librería ff.h (FAT Filesystem).

```
Partial Binaries transferred successfully!
```

2. Desactiva el puerto PCAP, que es el puerto interno de configuración por defecto, para poder utilizar el puerto ICAP.
3. Pone el Virtual Socket Manager en modo shutdown para poder cambiar su configuración

```
Putting the Filter RP in Shutdown mode
Status: 0
Waiting for the shutdown to occur
Filter RP is shutdown
```

4. Escribe en los bancos de configuración del Virtual Socket Manager las direcciones donde están los bitstreams parciales y las longitudes de los bitstreams. Además, configura los triggers de software para poder reconfigurar desde la aplicación

```
Initializing RM bitstream address and size registers
Initializing RM trigger ID registers
Initializing RM address and control registers
```

5. Envía por UART un resumen de toda la configuración del VSM
   5.1. Primero, de las direcciones y tamaños de los bitstreams

```
Reading RP bitstreams address and size registers
Binarize RM address = 300000
Binarize RM size = D934
Mean RM address = 200000
Mean RM size = D934
Blank RM address = 400000
Blank RM size = D934
```

   5.2. Después, de los triggers y de los índices de los triggers

```
Reading RP Trigger and address registers
Binarize RM Trigger0 = 0
Binarize RM Address0 = 0
Mean RM Trigger1 = 0
Mean RM Address1 = 1
Blank RM Trigger2 = 0
Blank RM Address2 = 2
```

6. Reinicia el VSM para comenzar con el procesado de señales y lee el registro de status

```
Putting the Filter RP in Restart with Status mode
Reading the Math RP status=0
```

7. Carga el filtro de binarización a la partición reconfigurable y procesa la señal de ejemplo

```
Generating software trigger for Binarize reconfiguration
Starting Binarize Reconfiguration
Loading new RM
RM loaded
Binarize Reconfiguration Completed!
Filtering original signal with Binarize Filter......Done!
```

8. Carga el filtro de media a la partición reconfigurable y procesa la señal de ejemplo

```
Generating software trigger for Mean reconfiguration
Starting Mean Reconfiguration
Loading new RM
RM loaded
Mean Reconfiguration Completed!
Filtering original signal with Mean Filter......Done!
```

9. Transmite por UART la señal de ejemplo y las señales filtradas

```
========== Original signal ==========
[ 62213 53220 20175 45435 62272 50659 43491 38660 1904 48339…
========== Binarize signal ==========
[ 255 255 0 255 255 255 255 255 0 255 0 0 0 255 0 0 0 0 255 0…
========== Mean signal ==========
[ 18348 31955 55785 46581 39940 47464 38373 36098 33810 12357…
```

Las tres señales tienen 100 muestras cada una, y están formadas por enteros de máximo 16 bits.

Para comprobar que las salidas son correctas, se ha hecho un sencillo script en MATLAB que efectúa las mismas operaciones que hacen los bloques de filtros de los módulos reconfigurables:

- El filtro de binarización recibía por el puerto `ain[0:31]` una muestra de la señal original, y en el puerto `bin[0:31]` un valor entero que se utiliza como umbral. Si la muestra es mayor que el umbral, la salida `result[0:31]` es el valor entero 255, de lo contrario es un 0. En la aplicación de ejemplo se ha utilizado el valor de umbral de 35442, que coincide con la media de la señal de entrada.

```
module rp(
    input [31:0] ain,
    input [31:0] bin,
    input Clk,
    input Reset_n,
    output [31:0] result
    );
reg [31:0] result;

always @(posedge Clk)
    if(Reset_n==1'b0)
        result <= 32'b0;
    else
        begin
            if (ain > bin)
                result <= 255;
            else
                result <= 0;
        end
endmodule
```

Figura 27: Código Verilog del filtro binarizador

- El filtro de media calcula a la salida `result[0:31]` la media de los cuatro valores `ain[0:15]`, `ain[16:31]`, `bin[0:15]` y `bin[16:31]`, obviamente como el entero inferior. En la aplicación de ejemplo, estos cuatro valores son los dos valores anteriores y los dos valores siguientes de la muestra que se está evaluando. Si tales valores no existen, se suponen de valor 0.

```
module rp(
    input [31:0] ain,
    input [31:0] bin,
    input Clk,
    input Reset_n,
    output [31:0] result
    );
reg [31:0] result;

always @(posedge Clk)
    if(Reset_n==1'b0)
        result <= 32'b0;
    else
        result <= (ain[31:16] + ain[15:0]
        + bin[31:16] + bin[15:0]) / 4;
endmodule
```

Figura 28: Código Verilog del filtro de media

El script de MATLAB comprueba que las señales recibidas desde el sistema y las señales generadas mediante software son iguales. Además, el script genera una gráficas superponiendo la señal de entrada a las distintas salidas de los filtros. En dichas gráficas podemos ver el correcto funcionamiento de los filtros, suavizando los picos de la señal en el caso del filtro de media, y asignando correctamente los valores alto y bajo según la señal de entrada esté por encima o por debajo del umbral.

El filtro de media es útil en tratamientos de imagen para suavizar las imágenes, pero perdiendo por otro lado cierta definición y desenfocando la imagen. Es por tanto un filtro de desenfoque.

El filtro de binarización es útil en tratamiento de imagen para convertir una imagen de escala de grises a blanco y negro, asignando un color u otro dependiendo del nivel de gris que presente cada pixel. Estas imágenes binarizadas se utilizan para reconocer patrones en la imagen, como *trackers* para realidad aumentada, que se distinguen mejor si la imagen es tratada para ofrecer un alto contraste.

Aunque los filtros implementados sean simples, son filtros muy utilizados en las tareas modernas de tratamiento de imagen, y pueden pasar a formar parte de un banco de filtros para utilizar haciendo uso de la reconfiguración parcial en un sistema de tratamiento digital de la señal.

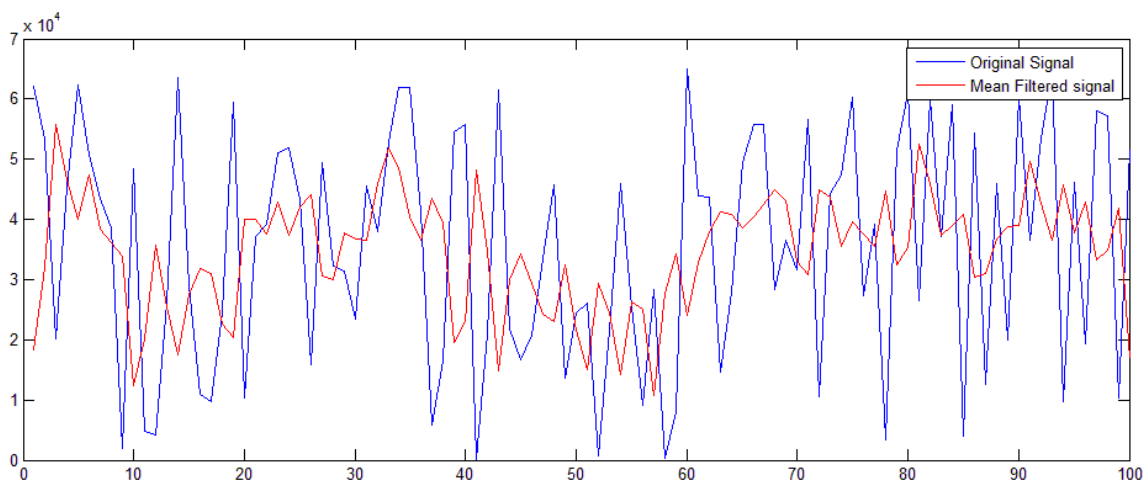

Figura 29: Señal original y salida del filtro de media

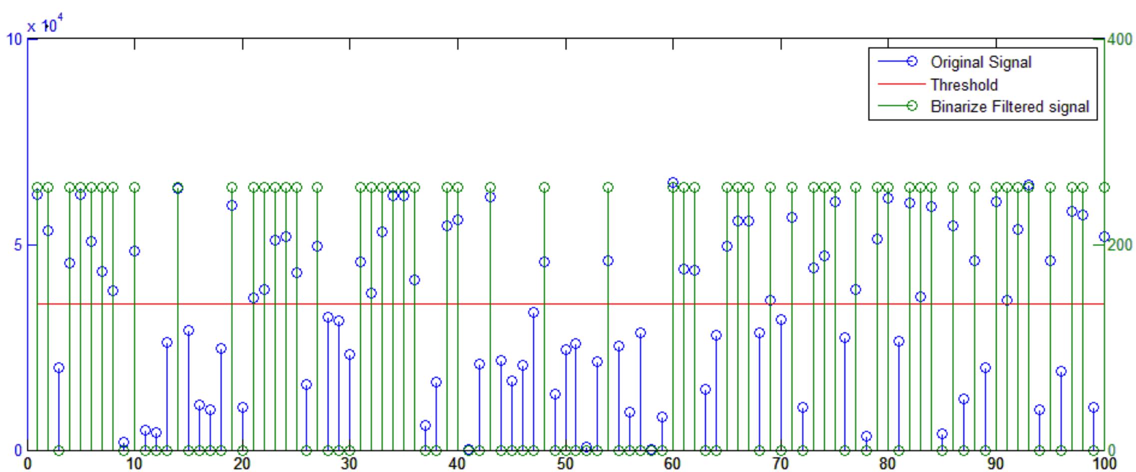

Figura 30: Señal original y salida del filtro de binarización

Conclusiones

En este proyecto se ha analizado el nuevo flujo de auto-reconfiguración parcial y dinámica propuesto por Xilinx recientemente, que sustituye a las técnicas anteriores, y que está destinado a los dispositivos reprogramables más avanzados disponibles en la actualidad, denominados SoRC (System on a Reprogrammable Chip), como las familias Zynq y UltraScale. Se han extraído las principales ventajas y limitaciones de esta tecnología, así como los distintos métodos, componentes y velocidades, con el objeto de proporcionar un marco claro que permita determinar la viabilidad de aplicar esta técnica a varios proyectos que se están desarrollando o se puedan desarrollar a corto plazo por los investigadores del Departamento.

Para verificar la metodología de trabajo, se ha aplicado la técnica de reconfiguración parcial a una sistema de ejemplo compuesto por una parte estática y una reconfigurable en la que se instancian alternativamente dos filtros simples utilizados frecuentemente en técnicas básicas de procesamiento de imágenes. Se ha implementado además un sistema de auto-reconfiguración parcial utilizando el controlador de reconfiguración desde una aplicación ejecutándose en un microprocesador en el chip, el caso de uso más sofisticado y de mayor utilidad de la PR. Como se ha comprobado en el capítulo anterior, efectivamente, las señales devueltas por la aplicación de prueba que se ejecuta en el microprocesador que envía y recibe los datos de los filtros, y las señales calculadas utilizando una versión solo-software del los algotirmos en MATLAB coinciden, lo que quiere decir que los filtros están funcionando correctamente y por tanto, la reconfiguración parcial que nos permite tener dos filtros en una sola partición reconfigurable.

Esta técnica puede lograr que en dispositivos considerablemente pequeños se puedan implementar algoritmos y aplicaciones específicas relativamente complejas, consiguiendo así reducir el consumo energético y el coste de fabricación, lo cual es siempre beneficioso en cualquier campo de la ingeniería. Pero aún más, esta técnica abre las puertas a grandes avances, como por ejemplo la actualización de sistemas críticos en caliente, o la disponibilidad de aceleradores hardware bajo demanda.

En el campo de telecomunicaciones, esta técnica puede tener un gran impacto en la industria y la investigación. Recientemente, los sistemas de telecomunicaciones por radio se están viendo desplazadas a un nuevo paradigma denominado Radio Definida por Software (Software Defined Radio, SDR). Los equipos de SDR, en lugar de construirse dedicados a una única modulación o tecnología, hacen todas las operaciones necesarias tratando la señal digitalmente. Esto quiere decir que el software define el funcionamiento de dicho transceptor, y un mismo equipo puede servir tanto para modulaciones analógicas (en amplitud, frecuencia, etc.) como para modulaciones digitales (QAM, PSK, etc.). Esta tecnología viene muchas veces limitada por la velocidad de cálculo de los equipos DSP que los componen, y es ahí donde entra en juego la técnica de reconfiguración parcial. Se han plataformas de radio definida por software y acelerada por hardware, en la que los procesos se hacen a alta velocidad gracias a los aceleradores de tareas específicas, y a la vez cuentan con una gran flexibilidad en cuanto a las diferentes técnicas de comunicación que se pueden implementar. Esto permite reunir una gran cantidad de estándares de telecomunicaciones con gran eficiencia en una misma plataforma.

Como ha quedado patente, el flujo de trabajo de un proyecto con reconfiguración parcial no dista mucho de un proyecto de síntesis *out of context*. Por lo tanto, los tiempos de producción

y de desarrollo de un proyecto de reconfiguración parcial se suponen similares. Así mismo, y como se comenta en el presente documento, actualmente el flujo de trabajo de la reconfiguración parcial requiere utilizar, casi en su totalidad, la línea de comandos de la suite de diseño Vivado, y no tiene soporte en la vista de proyecto. Cabe esperar, dada las grandes ventajas que esta técnica presenta, que en un futuro próximo el flujo de trabajo requerido para la reconfiguración parcial esté disponible en la vista de proyecto, facilitando y agilizando aún más los procesos de diseño y desarrollo.

Esta técnica puede ser utilizada en los más modernos dispositivos de Xilinx, a saber, dispositivos de la familia 7 Series (Virtex-7, Kintex-7, Artix-7, and Zynq-7000) y algunos dispositivos de la familia UltraScale, y es de esperar que cada vez más dispositivos, y obviamente las nuevas generaciones de dispositivos, estén disponibles para su uso en esquemas que hagan uso de la reconfiguración parcial. No sería extraño que con mejores soportes en el dispositivo y con mejoras en el flujo de trabajo, la reconfiguración parcial se convierta en una técnica popular entre investigadores, desarrolladores y fabricantes de productos.

Las líneas futuras al presente documento son muy amplias. Se podría plantear el uso de la técnica a través de un software o sistema operativo que, de forma totalmente transparente para el usuario, evaluara las necesidades del sistema y creara un acelerador hardware en instantes, para descongestionar el procesador y aligerar la carga de procesado. Además, esta técnica sería muy útil e interesante en aplicaciones espaciales o de difícil acceso al dispositivo, si se consigue que el sistema se auto-evalúe y en caso de avería genere nuevos periféricos para suplir las carencias causadas por la avería.

Si bien es cierto que es una técnica novedosa y potente, no todo es posible con la reconfiguración parcial. Las distintas opciones disponibles para efectuar la reconfiguración parcial (tanto en variedad de puertos como de hardware de control para llevarla a cabo) permiten ajustar el diseño a las necesidades de la aplicación y otorgan flexibilidad, pero hay aplicaciones que no se pueden implementar utilizando esta técnica. Concretamente, utilizar la reconfiguración parcial con aplicaciones que no se puedan separar en etapas para multiplexarlas en el tiempo tiene muy poco sentido. Además, planificar en un diseño la lógica de desacoplo entre la lógica estática y la lógica reconfigurable para una aplicación de altas prestaciones puede ser una ardua tarea si no se quiere poner en riesgo el rendimiento.

Anexo: Código de la aplicación FilterApp.c

A continuación se incluye el código utilizado para comprobar el diseño propuesto

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "xparameters.h"
#include "xil_printf.h"
#include "xil_cache.h"
#include "ff.h"
#include "xdevcfg.h"
#include "xil_io.h"
#include "xil_types.h"
#include "signal.h" //This header file contains a 100 random samples signal

// Parameters for Partial Reconfiguration
#define PARTIAL_BINARIZE_ADDR   0x300000
#define PARTIAL_MEAN_ADDR   0x200000
#define PARTIAL_BLANK_ADDR   0x400000
#define PARTIAL_BINARIZE_BITFILE_LEN  13901 // in number of words
#define PARTIAL_MEAN_BITFILE_LEN  13901 // in number of words
#define PARTIAL_BLANK_BITFILE_LEN  13901 // in number of words

// Table available after synthesis in PRC IP Core Customization Dialog
//| Virtual Socket Manager | Register      | Address |
//+------------------------+--------------+---------+
//| filter                 | STATUS        | 0X00000 |
//| filter                 | CONTROL       | 0X00000 |
//| filter                 | SW_TRIGGER    | 0X00004 |
#define filter_STATUS       XPAR_PRC_0_BASEADDR+0X00000
#define filter_CONTROL      XPAR_PRC_0_BASEADDR+0X00000
#define filter_SW_TRIGGER   XPAR_PRC_0_BASEADDR+0X00004
//| filter                 | TRIGGER0      | 0X00040 |
//| filter                 | TRIGGER1      | 0X00044 |
//| filter                 | TRIGGER2      | 0X00048 |
//| filter                 | TRIGGER3      | 0X0004C |
#define filter_TRIGGER0     XPAR_PRC_0_BASEADDR+0X00040
#define filter_TRIGGER1     XPAR_PRC_0_BASEADDR+0X00044
#define filter_TRIGGER2     XPAR_PRC_0_BASEADDR+0X00048
#define filter_TRIGGER3     XPAR_PRC_0_BASEADDR+0X0004C
//| filter                 | RM_BS_INDEX0 | 0X00080 |
//| filter                 | RM_CONTROL0  | 0X00084 |
//| filter                 | RM_BS_INDEX1 | 0X00088 |
//| filter                 | RM_CONTROL1  | 0X0008C |
#define filter_RM_INDEX0  XPAR_PRC_0_BASEADDR+0X00080
#define filter_RM_CONTROL0 XPAR_PRC_0_BASEADDR+0X00084
#define filter_RM_INDEX1  XPAR_PRC_0_BASEADDR+0X00088
#define filter_RM_CONTROL1 XPAR_PRC_0_BASEADDR+0X0008C
//| filter                 | RM_BS_INDEX2 | 0X00090 |
//| filter                 | RM_CONTROL2  | 0X00094 |
//| filter                 | RM_BS_INDEX3 | 0X00098 |
//| filter                 | RM_CONTROL3  | 0X0009C |
#define filter_RM_INDEX2  XPAR_PRC_0_BASEADDR+0X00090
#define filter_RM_CONTROL2 XPAR_PRC_0_BASEADDR+0X00094
#define filter_RM_INDEX3  XPAR_PRC_0_BASEADDR+0X00098
#define filter_RM_CONTROL3 XPAR_PRC_0_BASEADDR+0X0009C
```

```
//| filter                    | BS_ID0        | 0X000C0 |
//| filter                    | BS_ADDRESS0   | 0X000C4 |
//| filter                    | BS_SIZE0      | 0X000C8 |
//| filter                    | BS_ID1        | 0X000D0 |
//| filter                    | BS_ADDRESS1   | 0X000D4 |
//| filter                    | BS_SIZE1      | 0X000D8 |
#define filter_BS_ADDRESS0 XPAR_PRC_0_BASEADDR+0X000C4
#define filter_BS_SIZE0    XPAR_PRC_0_BASEADDR+0X000C8
#define filter_BS_ADDRESS1 XPAR_PRC_0_BASEADDR+0X000D4
#define filter_BS_SIZE1    XPAR_PRC_0_BASEADDR+0X000D8
//| filter                    | BS_ID2        | 0X000E0 |
//| filter                    | BS_ADDRESS2   | 0X000E4 |
//| filter                    | BS_SIZE2      | 0X000E8 |
//| filter                    | BS_ID3        | 0X000F0 |
//| filter                    | BS_ADDRESS3   | 0X000F4 |
//| filter                    | BS_SIZE3      | 0X000F8 |
#define filter_BS_ADDRESS2 XPAR_PRC_0_BASEADDR+0X000E4
#define filter_BS_SIZE2    XPAR_PRC_0_BASEADDR+0X000E8
#define filter_BS_ADDRESS3 XPAR_PRC_0_BASEADDR+0X000F4
#define filter_BS_SIZE3    XPAR_PRC_0_BASEADDR+0X000F8

// Constant strings for RM names
const char* reconfigurableModules[] = {"Binarize", "Mean", "Blank"};

// Read function for STDIN
extern char inbyte(void);

static FATFS fatfs;

// Driver Instantiations
static XDcfg_Config *XDcfg_0;
XDcfg DcfgInstance;
XDcfg *DcfgInstPtr;

int SD_Init()
{
	FRESULT rc;

	rc = f_mount(&fatfs, "", 0);
	if (rc) {
		xil_printf(" ERROR : f_mount returned %d\r\n", rc);
		return XST_FAILURE;
	}

	return XST_SUCCESS;
}

int SD_TransferPartial(char *FileName, u32 DestinationAddress, u32 ByteLength)
{
	FIL fil;
	FRESULT rc;
	UINT br;

	rc = f_open(&fil, FileName, FA_READ);
	if (rc) {
		xil_printf(" ERROR : f_open returned %d\r\n", rc);
		return XST_FAILURE;
	}
```

```
	rc = f_lseek(&fil, 0);
	if (rc) {
		xil_printf(" ERROR : f_lseek returned %d\r\n", rc);
		return XST_FAILURE;
	}

	rc = f_read(&fil, (void*) DestinationAddress, ByteLength, &br);
	if (rc) {
		xil_printf(" ERROR : f_read returned %d\r\n", rc);
		return XST_FAILURE;
	}

	rc = f_close(&fil);
	if (rc) {
		xil_printf(" ERROR : f_close returned %d\r\n", rc);
		return XST_FAILURE;
	}

	return XST_SUCCESS;
}

void reconfigurePartition(int trigger){
	xil_printf("Generating software trigger for %s reconfiguration\r\n",
reconfigurableModules[trigger]);
	int loading_done=0;
	int Status=Xil_In32(filter_SW_TRIGGER);
	if(!(Status&0x8000)) {
		xil_printf("Starting %s Reconfiguration\r\n",
reconfigurableModules[trigger]);
		Xil_Out32(filter_SW_TRIGGER,trigger);
	}
	loading_done = 0;
	while(!loading_done) {
		Status=Xil_In32(filter_STATUS)&0x07;
		switch(Status) {
			case 7 : print("RM loaded\r\n"); loading_done=1; break;
			case 6 : print("RM is being reset\r\n"); break;
			case 5 : print("Software start-up step\r\n"); break;
			case 4 : print("Loading new RM\r\n"); break;
			case 2 : print("Software shutdown\r\n"); break;
			case 1 : print("Hardware shutdown\r\n"); break;
		}
	}
	xil_printf("%s Reconfiguration Completed!\r\n",
reconfigurableModules[trigger]);
}

int main()
{
	int Status;

	// Flush and disable Data Cache
	Xil_DCacheDisable();

    // Initialize SD controller and transfer partials to DDR
	SD_Init();
	SD_TransferPartial("binarize.bin", PARTIAL_BINARIZE_ADDR,
(PARTIAL_BINARIZE_BITFILE_LEN << 2));
```

```
	SD_TransferPartial("mean.bin", PARTIAL_MEAN_ADDR, (PARTIAL_MEAN_BITFILE_LEN 
<< 2));
	SD_TransferPartial("blank.bin", PARTIAL_BLANK_ADDR, 
(PARTIAL_BLANK_BITFILE_LEN << 2));
	xil_printf("Partial Binaries transferred successfully!\r\n");

	// Invalidate and enable Data Cache
	Xil_DCacheEnable();

	// Initialize Device Configuration Interface
	DcfgInstPtr = &DcfgInstance;
	XDcfg_0 = XDcfg_LookupConfig(XPAR_XDCFG_0_DEVICE_ID) ;
	Status =  XDcfg_CfgInitialize(DcfgInstPtr, XDcfg_0, XDcfg_0->BaseAddr);
	if (Status != XST_SUCCESS) {
		return XST_FAILURE;
	}

	// De-select PCAP as the configuration device as we are going to use the 
ICAP
	XDcfg_ClearControlRegister(DcfgInstPtr, XDCFG_CTRL_PCAP_PR_MASK | 
XDCFG_CTRL_PCAP_MODE_MASK);

	// Display PRC status
	print("Putting the Filter RP in Shutdown mode\r\n");
	xil_printf("Status: %x\r\n",Xil_In32(filter_CONTROL));
	Xil_Out32(filter_CONTROL,0);
	print("Waiting for the shutdown to occur\r\n");
	while(!(Xil_In32(filter_STATUS)&0x80));
	print("Filter RP is shutdown\r\n");
	print("Initializing RM bitstream address and size registers\r\n");
	Xil_Out32(filter_BS_ADDRESS0,PARTIAL_BINARIZE_ADDR);
	Xil_Out32(filter_BS_ADDRESS1,PARTIAL_MEAN_ADDR);
	Xil_Out32(filter_BS_ADDRESS2,PARTIAL_BLANK_ADDR);
	Xil_Out32(filter_BS_SIZE0,PARTIAL_BINARIZE_BITFILE_LEN<<2);
	Xil_Out32(filter_BS_SIZE1,PARTIAL_MEAN_BITFILE_LEN<<2);
	Xil_Out32(filter_BS_SIZE2,PARTIAL_BLANK_BITFILE_LEN<<2);

	print("Initializing RM trigger ID registers\r\n");
	Xil_Out32(filter_TRIGGER0,0);
	Xil_Out32(filter_TRIGGER1,1);
	Xil_Out32(filter_TRIGGER2,2);

	print("Initializing RM address and control registers\r\n");
	Xil_Out32(filter_RM_INDEX0,0);
	Xil_Out32(filter_RM_INDEX1,1);
	Xil_Out32(filter_RM_INDEX2,2);
	Xil_Out32(filter_RM_CONTROL0,0);
	Xil_Out32(filter_RM_CONTROL1,0);
	Xil_Out32(filter_RM_CONTROL2,0);

	print("Reading RP bitstreams address and size registers\r\n");
	xil_printf("Binarize RM address = %x\r\n",Xil_In32(filter_BS_ADDRESS0));
	xil_printf("Binarize RM size = %x\r\n",Xil_In32(filter_BS_SIZE0));
	xil_printf("Mean RM address = %x\r\n",Xil_In32(filter_BS_ADDRESS1));
	xil_printf("Mean RM size = %x\r\n",Xil_In32(filter_BS_SIZE1));
	xil_printf("Blank RM address = %x\r\n",Xil_In32(filter_BS_ADDRESS2));
	xil_printf("Blank RM size = %x\r\n",Xil_In32(filter_BS_SIZE2));

	print("Reading RP Trigger and address registers\r\n");
```

```
xil_printf("Binarize RM Trigger0 = %x\r\n",Xil_In32(filter_TRIGGER0));
xil_printf("Binarize RM Address0 = %x\r\n",Xil_In32(filter_RM_INDEX0));
xil_printf("Mean RM Trigger1 = %x\r\n",Xil_In32(filter_TRIGGER1));
xil_printf("Mean RM Address1 = %x\r\n",Xil_In32(filter_RM_INDEX1));
xil_printf("Blank RM Trigger2 = %x\r\n",Xil_In32(filter_TRIGGER2));
xil_printf("Blank RM Address2 = %x\r\n",Xil_In32(filter_RM_INDEX2));

print("Putting the Filter RP in Restart with Status mode\r\n");
Xil_Out32(filter_CONTROL,2);
xil_printf("Reading the Math RP status=%x\r\n",Xil_In32(filter_STATUS));

int i;
int binarize[SIGNAL_SIZE];
int mean[SIGNAL_SIZE];

reconfigurePartition(0);
print("Filtering original signal with Binarize Filter...");
//Setting threshold to 0x8A72, which is the mean of our random signal
Xil_Out32(XPAR_MATH_0_S_AXI_BASEADDR+4,0x8A72);
//Writing original signal to filter
for(i=0; i<SIGNAL_SIZE; i++){
	Xil_Out32(XPAR_MATH_0_S_AXI_BASEADDR,signal[i]);
	binarize[i]=Xil_In32(XPAR_MATH_0_S_AXI_BASEADDR+8);
}
print("...Done!\r\n");
reconfigurePartition(1);
print("Filtering original signal with Mean Filter...");
//Processing 2 first samples from signal manually...
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR,0);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+2,0);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+4,signal[1]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+6,signal[2]);
mean[0]=Xil_In32(XPAR_MATH_0_S_AXI_BASEADDR+8);

Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR,0);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+2,signal[0]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+4,signal[2]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+6,signal[3]);
mean[1]=Xil_In32(XPAR_MATH_0_S_AXI_BASEADDR+8);

for(i=2; i<SIGNAL_SIZE-2; i++){
	Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR,signal[i-2]);
	Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+2,signal[i-1]);
	Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+4,signal[i+1]);
	Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+6,signal[i+2]);
	mean[i]=Xil_In32(XPAR_MATH_0_S_AXI_BASEADDR+8);
}

//Processing 2 last samples from signal manually...
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR,signal[SIGNAL_SIZE-4]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+2,signal[SIGNAL_SIZE-3]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+4,signal[SIGNAL_SIZE-1]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+6,0);
mean[SIGNAL_SIZE-2]=Xil_In32(XPAR_MATH_0_S_AXI_BASEADDR+8);

Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR,signal[SIGNAL_SIZE-3]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+2,signal[SIGNAL_SIZE-2]);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+4,0);
Xil_Out16(XPAR_MATH_0_S_AXI_BASEADDR+6,0);
```

```
        mean[SIGNAL_SIZE-1]=Xil_In32(XPAR_MATH_0_S_AXI_BASEADDR+8);
        print("...Done!\r\n");

        print("========== Original signal ==========\r\n");
        printSignal(signal,SIGNAL_SIZE);
        print("========== Binarize signal ==========\r\n");
        printSignal(binarize,SIGNAL_SIZE);
        print("========== Mean signal ==========\r\n");
        printSignal(mean,SIGNAL_SIZE);

    return 0;
}
```

Bibliografía

- The Zynq Book - Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC - Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz y Robert W. Stewart, University of Strathclyde.

- Zynq ZedBoard Concepts, Tools, and Techniques – Vivado 2014.2

- Xilinx UG909 - Vivado Design Suite User Guide - Partial Reconfiguration

- Xilinx PG193 - Partial Reconfiguration Controller - LogiCORE IP Product Guide

- Xilinx Workshop – Partial Reconfiguration (versiones 2014.x y 2015.x)

Agradecimientos

Quiero agradecer este proyecto

a mi familia, que siempre ha estado ahí y siempre estará,
a Fleming, que sin su apoyo todo habría sido más difícil,
a los inquilinos del loft de Cartagena por sus cafés a mitad de la noche,
a los habitantes del Spanish Village del CCC15 por descubrirme secretos que me animaron a acabarlo,
y a mi director de proyecto, por sus correos pasada la media noche.

Gracias de corazón.

Exención de Responsabilidad

Todos los nombres y productos registrados reflejados en el presente documento son propiedad de sus respectivas marcas.

Las figuras utilizadas pertenecientes a documentos de Xilinx son solo para su uso educativo - no comercial.