# Arquitectura de módulos reutilizables para DSP con orientación a objetos en VHDL

Álvaro Gámez Machado, Javier Gismero Menoyo, Alberto Asensio López [1]

*Resumen*— **Este artículo presenta una arquitectura genérica de etapas de procesado digital en VHDL haciendo uso de conceptos prestados de la Programación Orientada a Objetos. Mediante el encapsulado de las señales a procesar y haciendo uso de los recursos de polimorfismo del lenguaje VHDL, se simplifican los interfaces de puertos de entrada/salida de cada módulo digital, permitiendo su reubicación en cualquier punto de la cadena de procesado. Algunas herramientas accesorias simplifican la escritura de código redundante para simplificar el trabajo del diseñador de módulos de procesado de señal. Al establecer un interfaz común de puertos de entrada/salida para todos los módulos, todos ellos pueden ser substituidos por otras etapas que empleen esa interfaz; estando esta posibilidad de intercambio no limitada al momento preciso del diseño sino que facilita la reconfiguracin parcial dinámica del sistema.**

*Palabras clave*—**arquitectura, DSP, FPGA, POO, radar, reconfiguracin parcial dinámica, VHDL,**

## I. Introducción

EL procesado de señales en tiempo real es un problema computacional cada vez más habitual. Si bien la capacidad de procesado de los procesadores de propósito general así como de los dedicados al procesado de señal (DSP) es cada vez mayor, la tecnología de conversión analógico-digital ha mejorado también notablemente; hasta el punto de que existen componentes comerciales de apenas unas decenas de euros de coste que permiten frecuencias de muestreo superiores a los 2 GHz con resoluciones de 8 bits o superiores, existiendo conversores de hasta 4 GHz en desarrollo[1]. En otras ocasiones, la frecuencia de muestreo utilizada no es tan grande, siendo mucho más habituales frecuencias del orden del centenar de MHz; aunque en estas ocasiones la resolución suele incrementarse hasta los 14 ó 16 bits, haciendo uso, a veces, de dos canales de muestreo, típicamente I y Q. En ambos casos, sin embargo, es imprescindible el uso de procesadores dedicados al procesado de señal (DSP) e incluso el tratamiento de señal en FPGA para los sistemas más veloces o con estrictos requisitos de temporización.

Por otro lado, las posibles cadenas de procesado existentes son tan ilimitadas como los sistemas, modulaciones e ideas de los ingenieros; por lo que la tendencia es a desarollar arquitecturas muy específicas que cubran satisfactoriamente las necesidades de cada sistema. Debido a esto, no sólo es habitual diseñar una arquitectura hardware para cada proyecto, sino que dificulta además la reutilización de los *cores* diseñados con anterioridad. El proceso de testeo y verificación se hace también dificultoso, en tanto y cuanto cada módulo de procesado requiere de su propio banco de pruebas diseñado ad hoc.

En este trabajo se propone la utilización de una arquitectura hardware inspirada en la programación orientada a objetos, la encapsulación de señales, la definición de interfaces y el polimorfismo. Al mantener esta disciplina en la creación de los *cores*, se facilita enormemente la reutilización de los diseños, la recolocación de cada módulo de procesado en la cadena completa, y la configurabilidad del sistema completo. Aunque ésta no es la primera aproximación al paradigma de los objetos, algunas propuestas [2], [3] no han tenido gran aceptación por lo, aunque sea muy levemente, disruptivas; requiriendo extensiones al lenguaje y complicados preprocesadores. Algunas nuevas propuestas [4] requieren del aprendizaje de nuevos lenguajes y de encapsulación y doble encapsulación de los *cores* existentes o a diseñar. Si bien estos sistemas son altamente versátiles, presentan el problema del tiempo requerido para aprender y dominar el nuevo lenguaje de especificación de los módulos encapsulados.

Nuestra aproximación, por el contrario, se basa en la disciplina y políticas de desarrollo apoyadas por plantillas genéricas y algunas librerías de software orientadas a facilitar su uso. De este modo, los diseñadores de VHDL no tienen que emplear tiempo en aprender nuevos paradigmas y todo el código existente es reutilizable y puede ir adaptándose poco a poco a los interfaces propuestos. Por último, y aunque es aún una posibilidad por estudiar en profundidad, la definición de los interfaces se ha hecho teniendo en mente la gran versatilidad que supondría la reconfiguración parcial dinámica.

## II. Selección de interfaces

Proponemos la definición a priori de un interfaz común al que deben conformarse todos los módulos que realizan procesado sobre la señal. Haciendo uso de la capacidad de agrupación o encapsulación de señales en un tipo predefinido como `record`, es posible concentrar un bus de datos que represente la señal sobre la que se está operando. Cuanto mayor sea la agrupación de señales, tanto más sencillo será el interfaz general.

[1]Grupo de Microondas y Radar, Dpto. Señales, Sistemas y Radiocomunicaciones, ETSI Telecomunicación, Universidad Politécnica de Madrid. a.gamez@upm.es, javier@gmr.ssr.upm.es, vera@gmr.ssr.upm.es

```
entity processing_module is
    port (
        i_rst_clk : in rst_clk_t;
        i_data    : in data_t;
        o_data    : out data_t
    );
end processing_module;
```

Fig. 1. Definición de interfaz común a todos los módulos

El interfaz propuesto (Fig. 1) está compuesto por únicamente tres conjuntos de señales (tipo `record`) que se describen cualitativamente a continuación y en detalle más adelante:

- La primera de ellas, de tipo `rst_clk_t`, agrupa el conjunto de relojes, señales de reset y señales de activación de reloj (*chip enable* o `CE`).
- La segunda señal, de tipo `data_t`, contiene no solo los datos en si mismos sino también todas las señales que deban ir sincronizadas con cada muestra. Dentro de estas señales contamos con algunas esenciales, como una que indique la validez del dato.
- La tercera señal es del mismo tipo `data_t`; y el hecho de que la entrada y la salida sean del mismo tipo garantiza que los módulos puedan situarse en cualquier punto de la cadena de procesado.

Los convenios habituales para dotar a un módulo VHDL de la capacidad de ser reemplazado por otro mientras el sistema se encuentra en funcionamiento (reconfiguración parcial dinámica) exigen que las señales de entrada y salida se encuentren registradas en cada módulo[5], de manera tal que el rutado y temporización de cada etapa pueda realizarse por separado y de forma independiente al resto de módulos.

Así pues, al interfaz de puertos propuesto se añaden unas líneas propias de la arquitectura que sintetizan los registros de entrada y salida de los datos (Fig. 2).

```
architecture reconfigurable of
    processing_module is
    signal si_data, so_data : data_t;
    -- [...]
begin
    si_data <= i_data when rising_edge(
        i_rst_clk.c) and i_rst_clk.r='0' and
         i_rst_clk.ce='1';
    o_data <= so_data when rising_edge(
        i_rst_clk.c) and i_rst_clk.r='0' and
         i_rst_clk.ce='1';
    -- [...]
end reconfigurable;
```

Fig. 2. Registro de los datos de entrada/salida

El fichero así resultante puede emplearse como plantilla para todos los módulos del sistema. Asociada a esta plantilla es posible diseñar un *testbench* que sirva, de forma genérica, a cada etapa de procesado; puesto que a todas ellas se les exige que sean capaces de tratar con el mismo formato de señal, ancho de palabra, señales de control, etc.

## III. Señales encapsuladas

Aunque los interfaces propuestos pueden emplearse de forma genérica para la mayoría de los sistemas de procesado de señal, es evidente que cada uno tiene sus propias características y peculiaridades. Éstas, sin embargo, tienen cabida en el modelo propuesto, en tanto y cuanto el tipo `record` descrito anteriormente puede dar cobijo a multitud de señales. En cualquier caso, no obstante, consideramos la estructura de la Fig. 3 obligatoria para los diseños que realizamos en el Grupo de Microondas y Radar.

```
type chan_t is record
    data : std_logic_vector(WORD_WIDTH-1
        downto 0);
    en : std_logic;
end record;
type chan_array is array(natural range <>)
    of chan_t;

type meta_t is record
    valid : std_logic;
    ...
end record;

type data_t is record
    chan : chan_array(CHAN_NUMBER-1 downto
        0);
    meta : meta_t;
end record;
```

Fig. 3. Definición de los tipos *record* empleados

En este modelo contemplamos dos grados de libertad en concepto de ancho de palabra y de número de canales. El motivo de ser de la primera variable es evidente: cada sistema tiene sus propios requisitos de ruido de cuantificación y, por tanto, se debe contemplar este fenómeno para permitir la reusabilidad de los módulos mediante la especificación del ancho de palabra utilizado. La segunda variable, el número de canales, es útil especialmente en sistemas que trabajen con los canales procedentes de un demodulador I/Q. Incluso si éste no es el caso, operaciones como la FFT son comunes y requieren siempre de una salida compleja, cuando no también la entrada.

Aunque cada canal cuenta con un bit que indica si ese canal se encuentra activo o no, la señal de tipo `meta_t` incluye también una señal denominada `valid`, que indica si el conjunto completo definido por el `record data_t` es válido y debe ser procesado. Esta señal es esencial cuando suceden cambios de dominio de reloj a través de FIFOs asíncronas, puesto que tanto a la entrada como a la salida de tales FIFOs puede haber un lapso de tiempo (medido en ciclos de reloj de escritura/lectura) en los que no haya datos disponibles; pues su procedencia se encuentra en un dominio de reloj diferente y la relación $f_{in}/f_{out}$ determinará en qué medida hay ciclos muertos entre una muestra y otra.

Así pues, conectando la señal `si_data.meta.valid` a la señal `wr_en` de una de estas FIFOs, guardaremos en ella tan solo los datos válidos; mientras que los podremos extraer y marcar como válidos para el segundo dominio de reloj conectando la señal `not empty` a las señales

`rd_en` y `so_data.meta.valid` de la propia FIFO y del módulo en cuestión respectivamente. La Fig. 4 ilustra esta situación.

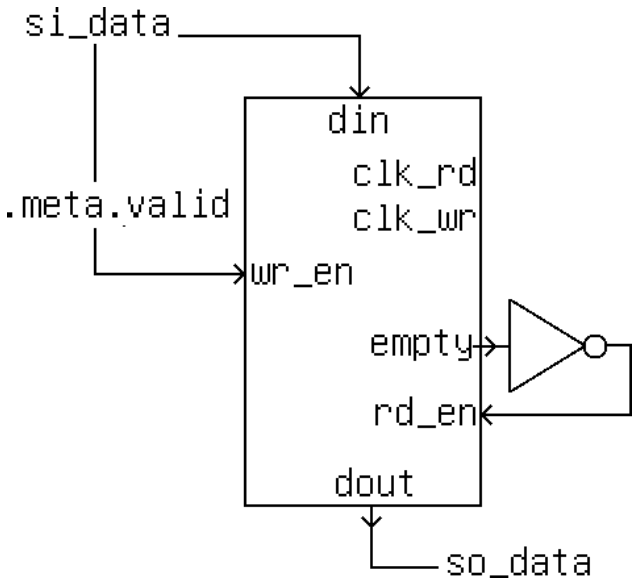


Fig. 4. Uso del bit de muestra valida en una FIFO asíncrona

Además de estas señales definidas como se ha explicado, se contempla de forma general la posibilidad de que el campo `meta` albergue diverso número de señales. Aquí es donde se hace máxima la variabilidad entre unos sistemas y otros, pues la diferencia no radica en rasgos generales como es la precisión binaria; sino que implica conceptos a nivel del significado de las operaciones y del procesado en sí mismo. Un ejemplo de esto son los sistemas que requieren conocer del momento preciso en que comenzó la captura por parte del conversor analógico/digital. Esta señal de `trigger` puede ser incluida para marcar la primera muestra entregada por el conversor. En tal caso, el `record meta_t` debería incluir una señal de tipo `std_logic` y nombre `trigger`.

Otro caso habitual consiste en marcar alguna característica de la muestra tomada, como por ejemplo la configuración del ADC, su escala o la frecuencia de muestreo en sistemas multitasa. En estos casos, es factible incluir al campo `meta` una señal que indique estos valores, a través de señales de un solo bit o de varios, de tipo `std_logic_vector` o incluso `unsigned`.

## IV. Polimorfismo

El apartado anterior planteaba una posibilidad – la del cambio de dominio de reloj– obviando intencionadamente el problema que plantea el hecho de trabajar con tipos `record`. En efecto, cuando se trata de utilizar algún *core* como las FIFOs asíncronas proporcionadas por terceros, no es posible situar a su entrada señales tan complejas como un `record` de la forma que parece indicar la Fig. 4.

Este problema, aunque real, tiene solución. La solución pasa por el diseño de determinadas funciones que permitan convertir las complicadas señales empleadas en algo estándar desde el punto de vista de los *cores* habitualmente empleados. Aunque el número de señales descrito es limitado, es posible pensar en una expansión de la arquitectura que contemple un número mayor y muy variable de definición de nuevos tipos. En este caso, la programación manual de todas y cada una de las funciones de conversión se hace muy tediosa, propensa a errores y de difícil ajuste a los cambios.

A modo de ejemplo, la Fig. 5 presenta el prototipo o interfaz de las funciones de conversión del tipo `data_t`.

```
subtype slv_data_t is std_logic_vector(
    C_DATA_T_SIZE-1 downto 0);
function to_slv(d: data_t) return slv_data_t
    ;
function from_slv(d: slv_data_t) return
    data_t;
function fill_others(d: std_logic) return
    data_t;
```

Fig. 5. Interfaz de las funciones de conversión

El significado de las dos primeras funciones es evidente por el nombre; se trata de funciones que convierten la agrupación de señales en una señal `std_logic_vector` y viceversa. Haciendo uso de ambas es posible inyectar la agrupación en una FIFO, enviarlas a memoria RAM o trabajar con ellas con *cores* ajenos que no contemplan la arquitectura propuesta. La tercera función cumple el objetivo de simular la construcción del lenguaje VHDL `(others => value)`, permitiendo por ejemplo poner a 0 todas las señales y subseñales, sin más que escribir una asignación tradicional; por ejemplo: `so_data <= fill_others('0')`.

La Fig. 6 muestra la forma de solucionar el problema que inspira la creación de las funciones de conversión citadas.

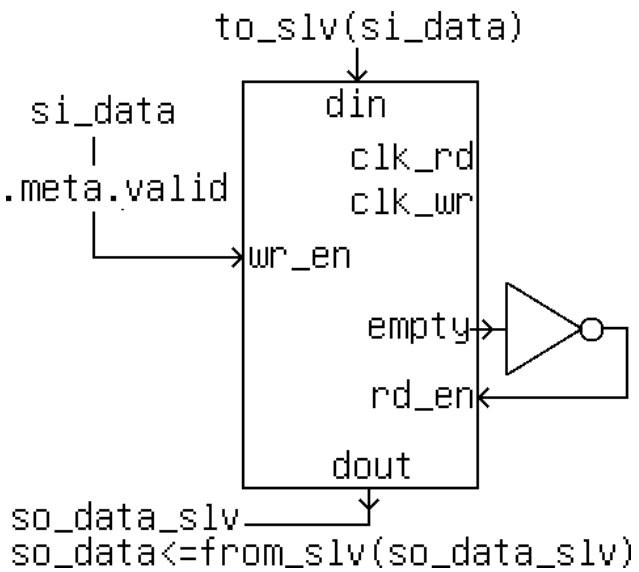


Fig. 6. Uso de las funciones de conversión a tipos básicos

### A. Generación automática del código

Al hacer uso de nombres de funciones idéntico para todos los tipos de datos definidos, pero cada uno

de ellos con sus propios tipos de entrada y salida, podemos aplicar la capacidad de polimorfismo del lenguaje VHDL para simplificar la tarea del diseñador.

Como ya comentábamos, no obstante, la codificación de estas funciones hace todo lo contrario: dificultad la tarea del diseñador. Así pues, la solución propuesta inclye no solo la definición de las funciones de conversión, sino también una librería desarrollada en Python que, analizando la definición de los tipos utilizados, genera dichas funciones. De esta forma, la labor del diseñador se reduce a definir los tipos requeridos y ejecutar la utilidad de generación de funciones (o bien dejar que se ejecute de forma automática en caso de utilizar sintetizadores con capacidad de *scripting*, como es el caso de las herramientas de Xilinx y su soporte del lenguaje `TCL`.).

El funcionamiento de la librería se basa en la interpretación del código VHDL original, tomando como entrada ficheros de las características del listado de la Fig. 3; de donde es posible extraer las características de cada tipo, la longitud de cada una de sus señales internas cuando se trata de tipos básicos (`std_logic`, `std_logic_vector`) y también de las señales que son a su vez otros tipos `record` definidos previamente. Analizando de forma recursiva cada una de las expresiones interiores, el software genera los subtipos `std_logic_vector` de la longitud adecuada y las funciones de conversión. Haciendo uso del operador de concatenación `&` en el caso de la conversión a `std_logic_vector` y mediante el direccionamiento de los bits necesarios para la conversión desde `std_logic_vector`, el script es capaz de generar funciones que actúan de manera consistente, sin errores y sin intervención del diseñador.

No obstante, las limitaciones del software desarrollado no son pocas. El lenguaje VHDL presenta una sintaxis complicada, con alta variabilidad y de difícil análisis. Por este motivo, el software no realiza un análisis sintáctico completo; sino que se limita a reconocer los elementos descriptivos del código: declaración de tipos, subtipos y arrays, tipos `record`, definición de funciones y procedimientos; así como de la definición de entidades, nombre de las mismas y puertos de entrada y salida.

A pesar de las limitaciones del software, su capacidad actual permite no solo generar el código más repetitivo de forma automática (como las funciones de conversión a y desde `std_logic_vector`), sino que también es capaz de generar módulos en VHDL que interconecten varios de los submódulos diseñados, siempre y cuando sigan un patrón de señales común como el descrito en la Fig. 1, evitando así el tedio y posibles errores en la interconexión de señales.

## V. Caso de ejemplo: procesado radar

El sistema de desarrollo propuesto se encuentra ya en uso en un sistema de procesado radar desarrollado por los autores. La Fig. 7 representa la cadena de procesado simplificada del sistema de procesado en una FPGA Virtex5.

El sistema propuesto parte de la digitalización de la señal a través de un ADC conectado directamente a la FPGA y controlado por ella. A la salida del módulo se encadenan los elementos de procesado de la señal. Un filtro elimina la componente continua, y un módulo que contiene a su vez otros submódulos con el mismo interfaz realiza la parte de procesado correspondiente al dominio del tiempo. A continuación, tras realizar el enventanado de la señal se calcula su FFT mediante un *core* de Xilinx, previamente encapsulado en un módulo que ofrece los interfaces definidos en 1. Si hasta este momento la señal procesada hacía uso de un único canal, ahora utiliza dos para poder representar la salida real e imaginaria de la FFT. Para mantener el ancho de palabra, este *core* ha sido configurado con la apropiada estrategia de redondeo que minimiza los lóbulos laterales de la señal enventanada. Tras este proceso, un nuevo subconjunto de procesado en el dominio de la frecuencia termina de realizar el procesado de la señal.

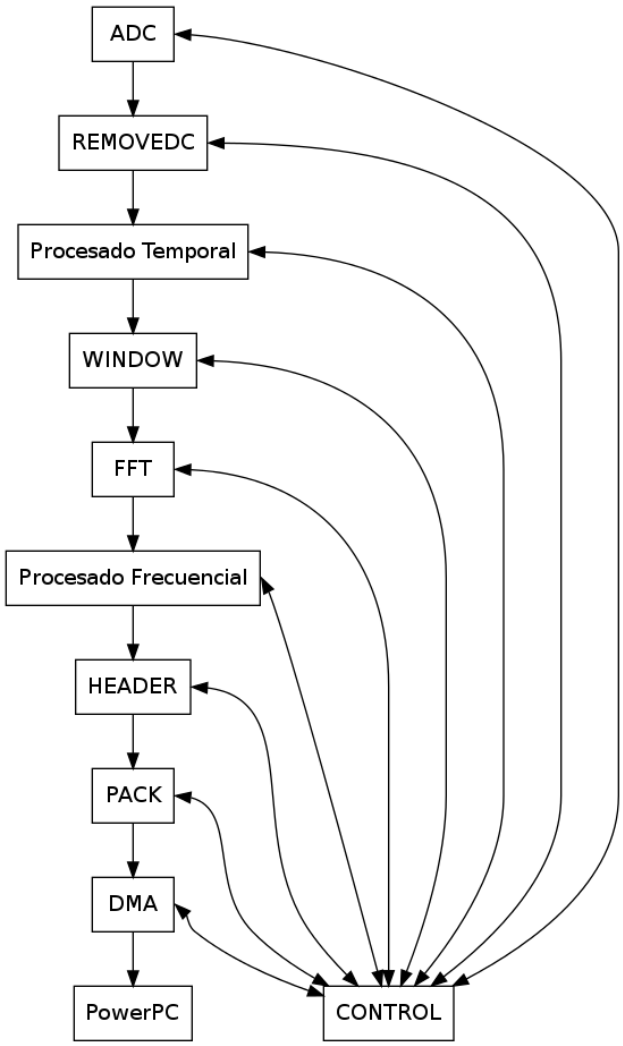


Fig. 7. Arquitectura de ejemplo: cadena de procesado radar

Finalmente, debido a que la información procesada va a transferirse a un procesador de propósito general, un PowerPC, en el que se realizarán otras tareas de procesado de temporización menos crítica, se sitúa un módulo denominado `HEADER`. Este módulo intercala entre las tramas que se desean transferir,

en los momentos adecuados, una serie de cabeceras que ayudarán al procesador a alinear los datos recibidos, identificar su formato y poder convertir una trama de bytes en una estructura en memoria a la que accederá un programa escrito en lenguaje C.

A este módulo le sigue una etapa de preparación para transferencia por DMA hacia la memoria del PowerPC. Dado que las transferencias DMA se realizan haciendo uso de un bus de ancho de 64 bits, es necesario convertir los datos agrupados en señales de tipo `record` a señales `std_logic_vector` de este mismo ancho. De esto se encarga el módulo `PACK` y, haciendo uso de nuevo de la señal `meta.valid`, se identifican las muestras que han de ser empaquetadas y escritas una FIFO asíncrona que permite transferir los datos desde el dominio de reloj de procesado al dominio de reloj de la transferencia DMA a través de un bus PCI.

En esta arquitectura se han empleado algunos *cores* privativos proporcionados por el mismo fabricante del hardware empleado. En concreto, el módulo `CONTROL` permite al PowerPC acceder a registros de la FPGA para modificar la configuración de cada uno de los módulos de procesado. Evidentemente, estas señales de control modifican el interfaz propuesto al principio del artículo y suponen el principal conflicto con la arquitectura propuesta. Sin embargo, es posible encontrar diferentes formas de solventar este problema para mantener la independencia de los módulos y su capacidad de reubicación en cualquier parte de la cadena de procesado y en cualquier zona, físicamente hablando, de la FPGA. En concreto, haciendo uso de un bus tipo *daisy chain*, en el que cada módulo contiene, además de los puertos definidos anteriormente, un puerto de entrada y otro de salida (posiblemente también tipos `record`) que conecten en anillo todos los módulos junto al módulo `CONTROL`, se permite al módulo `CONTROL` y al PowerPC que accedan a los registros de configuración internos a los módulos de procesado sin perder ninguna de la ventajas que proporcionan la arquitectura ideada. Tan solo es necesario un pequeño controlador en el interior de cada módulo que sea capaz de identificar las instrucciones de control que van dirigidas a él, para procesarlas y devolver una respuesta, o las que van destinadas a otro módulo y deben entonces propagarlas sin interferir en ellas.

## VI. Conclusiones y resultados

El sistema de desarrollo propuesto cubre las necesidades elementales para el desarrollo de múltiples cadenas de procesado, permitiendo la reutilización de cada una de las etapas en otros sistemas que sigan el mismo convenio de desarrollo. La previsión de diferentes anchos de palabra y extensibilidad del campo que almacena los metadatos de cada muestra hacen de la arquitectura general un sistema versátil que puede utilizarse en un elevado rango de sistemas de procesado de señal.

El caso de ejemplo explicado, si bien se basa en un sistema radar concreto, recoge lo que bien puede ser el esquema general de un sistema de procesado de señal. Las etapas más habituales, ADC, filtrado inicial y el interfaz con otros dispositivos del sistema (marcado de cabeceras, transferencia por DMA o a través de otro canal) pueden ser en general idénticos o muy similares entre sistemas. La fase de conversión al dominio de frecuencia también es habitual en sistemas que requieren de procesado espectral además de temporal; por lo que pueden generarse diversos sistemas de procesado siguiendo la misma arquitectura y alterando (o suprimiendo) únicamente los bloques no descritos en detalle: el procesado temporal y el procesado frecuencial.

Aunque aún no exploradas las posibilidades de reconfiguración parcial dinámica, el sistema está preparado para aplicarla; en tanto y cuanto la reconfiguración en tiempo de diseño es inmediata al seguir todos los módulos interfaces comunes (y, en el caso de no serlo, las herramientas software diseñadas facilitan la tarea de integración), y se han tomado las precauciones necesarias para permitir el rutado y localización independiente de cada uno de los submódulos. La profundización en esta capacidad y la extensión de la librería software y sus aplicaciones a la reconfiguración parcial dinámica serán las futuras líneas de trabajo.

## Agradecimientos

Este trabajo ha sido financiado a través de los proyectos TEC2008-02148 y TEC2011-28683 del Ministerio de Ciencia e Innovación.

## Referencias

[1] K. Poulton, R. Neff, A. Muto, W. Liu, A. Burstein, and M. Heshami, "A 4 gsample/s 8b adc in 0.35 mu;m cmos," in *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, vol. 1, 2002, pp. 166 –457 vol.1.

[2] S. Swamy, A. Molin, and B. Covnot, "Oo-vhdl. object-oriented extensions to vhdl," *Computer*, vol. 28, no. 10, pp. 18 –26, oct 1995.

[3] P. Ashenden, P. Wilsey, and D. Martin, "Suave: painless extension for an object-oriented vhdl," in *VHDL International Users' Forum, 1997. Proceedings*, oct 1997, pp. 60 –67.

[4] M. Sánchez Marcos, Á. Herrero, and M. López-Vallejo, "xhdl: Extending vhdl to improve core parameterization and reuse," in *Advances in Design and Specification Languages for SoCs*, P. Boulet, Ed. Springer US, 2005, pp. 217–235, 10.1007/0-387-26151-6-16.

[5] Xilinx Inc, "Xilinx partial reconfiguration user guide," 2010.