

# Síntesis de Unidades Funcionales para Soft-Cores desde un modelo C/C++

Borja Martínez Huerta, Màrius Montón Macián, Jordi Carrabina Bordoll

Dept. de Microelectrónica y Sistemas Electrónicos  
Universidad de Autònoma de Barcelona

{borja.martinez,marius.monton}@uab.es

**Resumen.** *La incorporación de unidades funcionales personalizadas específicamente para una aplicación en procesadores RISC genéricos permite incrementar las prestaciones de forma significativa. Mediante un proceso de perfilado se localizan las funciones críticas en el tiempo y éstas son reemplazadas por bloques de hardware que realizan la misma función a una velocidad muy superior. En este artículo se presenta una nueva metodología que permite la síntesis automática de la unidad funcional partiendo de un modelo o código C/C++ untimeed. Se describe el método para localizar las funciones candidatas a ser reemplazadas y los mínimos cambios necesarios para transformar el código en una descripción SystemC sintetizable de la función o sección de código del algoritmo a acelerar.*

## 1. Introducción

Las siglas ASIP (*Application Specific Instruction Set Processor*) se usan de una forma genérica para designar aquellos procesadores que tienen algún tipo de personalización arquitectural para el software que van a ejecutar. Ciertamente existe un gran número de modificaciones posibles sobre una arquitectura y un no menos importante número de metodologías para llevarlas a cabo.

Bajo esta definición de ASIP cabe un gran abanico de posibilidades [1]. Por un lado, la personalización puede ser genérica para un grupo o tipo de aplicaciones. En este sentido, un microcontrolador puede ser visto como un ASIP ya que incorpora instrucciones específicas para operaciones que involucran un único bit (un gran porcentaje de las tareas de un microcontrolador está dedicado a recibir estímulos de un sensor por una única pata del chip y actuar también de forma binaria sobre leds, relés y dispositivos similares).

De igual forma un DSP (*Digital Signal Processor*) incorpora unidades funcionales específicas y replicadas para realizar las operaciones típicas que aparecen en las aplicaciones de procesado de señal, como el filtrado o transformadas de Fourier. En el otro extremo existen metodologías extremadamente complejas que permiten generar la arquitectura completa desde una aplicación específica. Ejemplos podrían ser el core Xtensa [2] o el Gepard-Core [3]

Quizás la forma más sencilla y utilizada de acelerar algoritmos críticos en el tiempo es la incorporación de instrucciones específicas al repertorio estándar de un procesador. Con instrucciones específicas es posible reducir un conjunto complejo de instrucciones estándar a una única instrucción implementada en hardware. El punto a favor de esta metodología es que el uso como base de un procesador estándar permite simplificar enormemente el diseño. Como ejemplo el PRISM-Project [4] se basa en esta idea de aumentar el repertorio de instrucciones en este caso de un procesador RISC.

Hace tan solo unos años la posibilidad de desarrollar procesadores con instrucciones específicas quedaba en el dominio de los grandes fabricantes y diseñadores de procesadores [5][6]. Sin embargo, con la aparición de procesadores *Soft-Core*, el significativo aumento de capacidad de los dispositivos de lógica programables (que permiten la inclusión en un solo dispositivo de varios procesadores) y la aparición de nuevas metodologías y herramientas que simplifican enormemente el esfuerzo de diseño, ha reducido el tedioso trabajo que suponía generar un procesador a medida. En esta trabajo se presenta una nueva metodología que permite generar de forma sencilla unidades funcionales para *Soft-Cores* configurables que se apoya en las nuevas herramientas que permiten la síntesis de hardware

desde una descripción SystemC de alto nivel. Sin embargo, SystemC no es tomado como un lenguaje de modelado del sistema completo como es habitual. Todo lo contrario, nuestro punto de partida es el código C que se pretende acelerar, código que se supone ya validado funcionalmente. Mediante unos mínimos cambios se obtiene un código SystemC sintetizable equivalente a las funciones C que reemplazan.

Este artículo está dividido de la siguiente forma: la sección 2 explica de forma breve que es un *Soft-Core* y de forma un poco más detallada el procesador NiosII, utilizado para demostrar la validez de la metodología. La sección 3 trata sobre SystemC sintetizable y las herramientas utilizadas. La sección 4 presenta la metodología de forma detallada y en la sección 5 los ejemplos utilizados para validarla.

## 2. *Soft-Cores* e Instrucciones Específicas

### 2.1. *Soft-Cores* Configurables

Un procesador *Soft-Core* es una descripción sintetizable de un procesador, normalmente a nivel RTL, disponible en un lenguaje de descripción de hardware (HDL) que define la lógica que lo implementará. La principal ventaja de un *Soft-Core* es la flexibilidad: como el hardware no está aún sintetizado es posible hacer modificaciones sobre el HDL, o bien configurarlo de forma sencilla. Actualmente existen numerosos *cores* disponibles en el mercado como el DP8051CPU[7] o R8051XC [8] ambos basados en la arquitectura 8051. Normalmente en los casos comerciales el HDL no es accesible y por tanto únicamente ciertos parámetros arquitecturales pueden ser configurados (como el uso de multiplicadores, tamaño de memoria *cache* etc.) también existen *cores* de código abierto [9] y por tanto modificables a voluntad

### 2.2. Nios II y Custom Instructions

El procesador NiosII es un *Soft-Core* diseñado específicamente para los dispositivos lógicos programables de Altera. El NiosII está basado en un *core* RISC estándar. La configuración del NiosII parte de tres bases denominadas E (*Economical*) S (*Standard*) y F (*Fast*) que se diferencian en el uso o no de memoria *cache*, las

etapas del *pipeline* o la predicción de saltos dinámica. Sobre estas bases se configuran ciertos parámetros de la arquitectura, como el tamaño de las *caches* o la instancia de unidades funcionales prediseñadas (multiplicador, divisor, unidad de punto flotante...). El repertorio de instrucciones puede ser ampliado con hasta 256 instrucciones específicas

Las CIs (*Custom Instructions*) pueden ser vistas como bloques de lógica específica insertadas en el *datapath* de la CPU. Desde el punto de vista arquitectural existen tres tipos diferentes de CIs Figura 1

Combinational Custom Instructions: Son bloques lógicos capaces de completar su función en un ciclo de reloj. Es la manera más sencilla de insertar unidades funcionales, ya que no son necesarias señales de control

Multi Cycle Custom Instructions: Son bloques lógicos que requieren de más de un ciclo para completar una operación. El número de ciclos puede ser fijo o variable y las señales de control asociadas dependerán del caso.

External Interface Custom Instructions: El procesador NiosII permite añadir unidades funcionales a la CPU con acceso a lógica externa. Esta es una herramienta de gran potencia ya que a través de este interfaz el bloque lógico puede acceder a resultados intermedios, memorias ROM, *look-up tables* o parámetros presentes en lógica externa a la CPU.

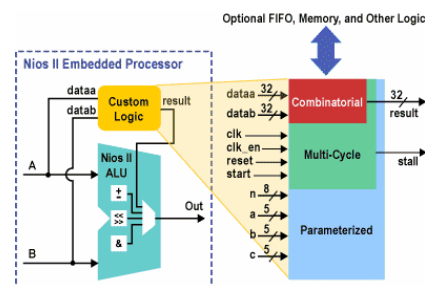


Figura 1. Nios II Custom Instructions.

## 3. Síntesis de SystemC

SystemC se está convirtiendo en un estándar en la verificación de sistemas heterogéneos. Son un conjunto de librerías C++ con capacidad para describir la concurrencia inherente al HW y las

comunicaciones entre diferentes módulos. Se puede usar SystemC para describir un sistema a muy alto nivel y refinar la descripción hasta llegar a un código RTL. Las metodologías clásicas aplicadas a SystemC lo utilizan para las descripciones a muy alto nivel, transformando luego de forma manual estas descripciones a lenguajes HDL.

Este paso hecho a mano puede ser una fuente importante de inserción de errores difíciles de detectar y solucionar, debido a la complejidad de las descripciones en sí así como las simulaciones mixtas SystemC/HDL.

Para evitar este paso manual se hace indispensable usar herramientas de síntesis automática de SystemC comportamental.[10][11] En nuestro caso concreto la herramienta utilizada ha sido el Cynthesizer 2.5 de ForteDS. La Figura 2 resume el flujo de diseño cuando se usa esta herramienta.

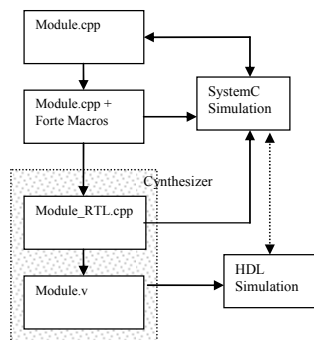


Figura 2. Flujo Diseño Forte.

La herramienta genera archivos Verilog a partir de código SystemC comportamental. Este código SystemC puede simularse con las herramientas habituales de SystemC, y una vez validado el módulo a este nivel, se prepara el código para ser leído por Cynthesizer. Esta preparación se hace necesaria para especificar ciertos tipos de datos (p.e.: enteros a enteros con un número determinado de bits) y también incluye macros de C para especificar optimizaciones tales como *loop unrolling*, *array flatten*, etc. Con toda esta información adicional, Cynthesizer genera un código SystemC RTL de nuestro módulo. Este módulo en RTL puede sustituir al original de más alto nivel en las simulaciones, permitiendo así validar de nuevo todo el sistema.

Si el código RTL generado cumple los requisitos, el siguiente paso es que Cynthesizer genere la traducción de los archivos a un Verilog equivalente. En este nivel Cynthesizer nos proporciona también los *wrappers* necesarios para hacer una simulación mixta SystemC/ Verilog con Modelsim. Tras este paso los archivos Verilog generados y validados, pueden ser usados como archivos de entrada cualquier herramienta de síntesis HDL clásica, en nuestro caso Precision de Mentor Graphics y QuartusII de Altera.

Por otro lado, este tipo de herramientas de síntesis comportamental no necesitan tener un código de entrada *cycle-accurated*. En general estas herramientas incluyen procesos automáticos de *scheduling* y *partitioning* de algoritmos *untimed* que los mapean directamente en HW. Sabiendo esto, podemos interpretar como código comportamental un código clásico C/C++. Con esta perspectiva, podemos rodear (*wrap*) un código *untimed* C/C++ con secciones SystemC para la entrada y salida del módulo (partes que si incorporan una descripción temporal) dejando a la herramienta la responsabilidad del *scheduling* de las partes complejas del algoritmo.

#### 4. Metodología

Las metodologías de SystemC tradicionales toman como punto de partida una descripción de muy alto nivel del sistema global. Tras la validación funcional se realiza la partición HW/SW que puede ser más o menos automática. Cuando esta partición queda fijada el HW puede ser sintetizado (en algunos casos directamente y en otros con la conversión previa a HDL) y el SW extraído de la descripción SystemC para ser luego compilado para un procesador concreto seleccionado.

Por el contrario, la metodología presentada en este trabajo toma como punto de partida un código C clásico compilable sobre un procesador tradicional. Existen dos motivos fundamentales para esta elección:

1. Como norma general, el diseñador de sistemas complejos intenta minimizar el número de funciones que se implementan por HW. Esto hace que habitualmente las primeras pruebas funcionales se realicen corriendo SW en un procesador. A partir de las pruebas funcionales que validan el modelo se realiza el proceso de perfilado con

el fin de localizar las secciones críticas de los algoritmos que deberían ser aceleradas por HW para cumplir con las especificaciones temporales.

2. A menudo los algoritmos que se quieren acelerar son directamente el denominado *Golden-Model* del sistema completo, lo que los convierte en un buen punto de partida para minimizar errores de interpretación. (Un buen ejemplo es el modelo MPEG que se utiliza en la siguiente sección de este trabajo)

En este punto existen dos variantes de la metodología:

#### 4.1. Método Directo

El repertorio de instrucciones de NiosII opera sobre cero, uno o dos operandos para obtener un resultado, por tanto una función con cero, uno o dos argumentos que devuelva un valor es una buena candidata para ser reemplazada por una unidad funcional específica. Figura 3

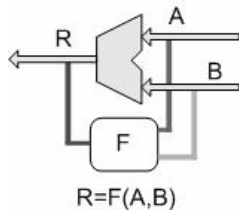


Figura 3. Método Directo de sustitución.

#### 4.2. Método Indirecto

Cuando una función es llamada pasando punteros como argumentos, sean puros o punteros a *arrays*, la unidad funcional los interpreta como posiciones de memoria. Gracias a la arquitectura extendida de las CIs de NiosII la unidad tiene acceso a hardware externo. Si este hardware externo es el propio *bus* de sistema la unidad funcional puede ver cualquier elemento de memoria que cuelgue del bus como hardware externo. De esta forma consigue acceso a variables de programa, parámetros almacenados en ROM o cualquier tipo de información almacenada en RAM.<sup>2</sup>

<sup>2</sup> No toda función puede ser acelerada por software. Por ejemplo, cuando una función llama a otra función no existe camino de retorno desde la unidad funcional al

Cuando las funciones candidatas son seleccionadas tras el proceso de perfilado, la unidad funcional equivalente ha de ser sintetizada, y posteriormente se ha de sustituir la llamada a la función el código C por una macro de llamada a la *Custom Intruction* (Figura 4)

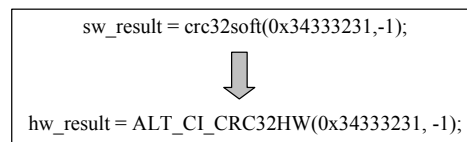


Figura 4. Substitución de función por macro

El primer paso para sintetizar la unidad es realizar los mínimos cambios necesarios para transformar los argumentos recibidos por la función en entradas de módulos SystemC. En el caso del método directo estos cambios son mínimos, tan sólo el tipo de los datos ha de ser reemplazado por tipos SystemC (Por ejemplo el tipo C *int* por el tipo SystemC *sc\_int*) y luego realizar las llamadas a los métodos SystemC que permiten hacer la lectura (*sc\_signal.read()* en la entrada y *sc\_signal.write()* para escribir el valor de salida)

Cuando la función recibe punteros es necesario realizar un poco más de trabajo ya que la unidad funcional ha de tener acceso al bus de sistema. Sin embargo, este trabajo extra se de realizar una sola vez, ya que cualquier elemento de memoria del sistema ha de cumplir con las especificaciones del *Avalon* (*Bus* de sistema del *Nios*). Así, desde el punto de vista de la unidad funcional este acceso se hace siempre de la misma forma.

Evidentemente, cuando estas transformaciones se realizan se pueden realizar las simulaciones pertinentes para validar que la CI generada realiza la función esperada. Esta simulación se realiza a alto nivel utilizando las herramientas SystemC, siendo posible también simular el código Verilog intermedio generado por Cynthesizer.

## 5. Ejemplos y Resultados

Los experimentos descritos a continuación han sido realizados sobre una plataforma de

procesador que permita realizar la llamada a la función software.

prototipado *Nios Development Kit Cyclone Edition*. Inicialmente se ha usado un NiosII sobre la configuración base denominada E, que se caracteriza por su simplicidad, ya que no utiliza ni *cache*, ni multiplicador HW ni predicción de saltos. El reloj de sistema es de 50 MHz. Otras características de la placa incluyen 1MByte de memoria SRAM para datos y programa.

La metodología seguida en los experimentos fue ejecutar y perfilar la aplicación sobre el NiosII para identificar las funciones más costosas en tiempo candidatas a ser pasadas a HW. Una vez localizadas se introdujo el código de estas funciones en nuestros *wrappers* SystemC de interfaz al *Bus Avalon* para sintetizar las unidades funcionales.

### 5.1. Primer Ejemplo: CRC32

La primera prueba realizada parte del código para el cálculo acumulado del algoritmo de CRC32 en palabras de 32 bits. En este ejemplo sólo hay una función con dos argumentos: la palabra a computar de 32 bits, y un *flag* que permite resetear el valor acumulado. La función por tanto calcula el CRC32 del parámetro sobre el valor acumulado devolviendo cada ejecución el valor actual.

El primer paso consistió en realizar las transformaciones de los tipos de datos descritas para el Método Directo (la función del CRC32 sólo tiene dos parámetros, un resultado y no necesita accesos externos). Los parámetros simplemente se pasan a través del *datapath* de la CPU y se recoge el resultado. En la parte de código SW principal, solo hay que cambiar la llamada a esa función por la macro para ejecutar la instrucción máquina específica (fig. 4).

Para este ejemplo se usaron directivas para que Cynthesizer hiciese automáticamente *loop unrolling* y *pipelining*.

Para comparar esta solución con una metodología clásica se implementó también un módulo equivalente escrito en VHDL. Para esta implementación se reutilizó una IP ya existente y testeada. Hay que destacar que esta IP está basada en un algoritmo diferente al código SW que portamos a *SystemC*, por tanto el resultado es únicamente orientativo de los valores que se pueden obtener siguiendo un método de diseño HW tradicional, pero no permite comparar las

diferencias en las herramientas de síntesis de HW desde VHDL y *SystemC*.

En la Figura 5 y la Tabla 1 se pueden ver los resultados de esta primera prueba. La implementación en VHDL resulta ser la más óptima en rendimiento y en ocupación de área. La versión SystemC generada casi automáticamente tiene un rendimiento 140 veces superior a la versión SW, aunque no llega a los resultados de la anterior. La gran ocupación del módulo generado a partir de SystemC puede deberse a dos factores:

- Las herramientas de síntesis aún no están suficientemente maduras.
- El algoritmo puramente SW que elegimos no es adecuado para una implementación HW.

La diferencia observada en los tiempos de ejecución de las dos implementaciones HW se debe la diferencia en los algoritmos empleados. La versión SystemC utiliza el mismo algoritmo que el modelo SW, mientras que la escrita en VHDL no se basa en un algoritmo SW y que se implementa con lógica combinatorial basada en cadenas de XORs, y que por tanto se ejecuta básicamente en un solo ciclo de reloj.

### 5.2. Un problema más complejo

Para el segundo ejemplo tomamos como punto de partida un código estándar de decodificador para MPEG2 que se toma como *Golden-Model* de la especificación. El perfilado de la aplicación muestra que dos funciones ocupaban el 15% del tiempo total de ejecución cada una: la función *idctcol* y la *idctrow*. Estas dos funciones son muy similares y calculan la transformada inversa del coseno (iDCT) usando el algoritmo de Chen-Wang [12] por columnas y por filas

En este caso los parámetros de la función a sustituir son punteros a la región de memoria donde está el *array* de valores sobre los que se aplica la iDCT. Al finalizar la función guarda los 8 valores de resultado en el mismo *array*. Por tanto para trasladar esta función hacia una unidad funcional se hizo necesario usar los *wrappers* de interfaz al bus, que permiten leer las posiciones de memoria adecuadas designadas por el puntero pasado como parámetro. Las 8 posiciones de memoria leídas se guardan en 8 registros, de manera que en el código SW original sólo hay que cambiar 8 accesos a variables por 8 accesos a registros.

Una vez obtenido el código SystemC con las partes *untimed* del código original, se usó Cynthesizer con los parámetros por defecto para obtener el módulo HW descrito en Verilog que permite sintetizar todo el sistema completo NiosII.

En la Tabla 1 están resumidos los resultados obtenidos. Hemos añadido también un prueba con NiosII-S (Versión *Standard*) con memoria *cache* de instrucciones y multiplicador HW, de forma que se puede comparar optimizaciones genéricas ofrecidas por Altera contra nuestra propuesta.

	IDCTCOL		IDCTROW	
	LEs	Cycles	LEs	Cycles
Nios-E	998	369177617	998	215894797
Nios-S	2001	48409014	2001	46275156
Nios-E+CI	4206	8997120	3911	7920000
Nios-S+CI	5209	6788744	4914	5766396

Tabla1. Resultados obtenidos para IDCTrow

En primer lugar se observa el hecho de que añadir unidades funcionales mejora el rendimiento en un factor 40 (función *idctcol*) y 30 (*idctrow*) respecto al NiosII más sencillo, mientras que en ocupación se aumenta en un factor 4 y 5 respectivamente.

En el caso de usar el NiosII-S (con *cache* y multiplicador) el rendimiento aumenta en un factor 7 mientras que el coste en LEs se multiplica por 2. Por tanto podemos concluir que el aumento de prestaciones al añadir unidades funcionales específicas es muy superior al obtenido con mejoras arquitecturales genéricas como puede ser el uso de *cache*. En este ejemplo concreto el incremento es casi 7 veces superior. Por completitud se ha añadido los resultados de la combinación de ambas soluciones conjuntamente y el resultado es una mejora de un factor 55 con respecto al caso básico.

Hay que destacar que se obtiene un importante incremento de prestaciones mediante un sencillo proceso en el que en ningún momento se hace necesario un conocimiento profundo del algoritmo original en sí, que no ha de ser transformado de forma manual ya que tan solo es necesario trabajar con los parámetros de entrada y salida de la función SW y la sustitución de las llamadas a ésta por macros de C en el código SW original.

## 6. Conclusiones

En este artículo se ha presentado una metodología para trasladar fácilmente funciones complejas hacia unidades funcionales de NiosII. De esta manera podemos generar un ASIP de una forma muy simple que permite un elevado incremento de las prestaciones en un tiempo de diseño mínimo.

Gracias a las nuevas herramientas de síntesis comportamental y mediante el uso de *wrappers* en SystemC ya testeados es posible portar a HW partes de código C/C++ estándar sin necesidad de conocer los algoritmos que describen.

Los resultados de síntesis obtenidos demuestran que la propuesta es viable y que al aumento de prestaciones es muy significativo. Más difícil de cuantificar es como la sencillez del método ahorra un enorme esfuerzo en tiempo y complejidad de diseño, si bien se han aportado algunos pequeños ejemplos para mostrar los mínimos cambios necesarios en el código original.

## Referencias

- [1] F. Vahid *The Softening of Hardware* IEEE Computer, April 2003
- [2] Tensilica Inc: [www.tensilica.com](http://www.tensilica.com). 2006
- [3] Austria Micro Systems [www.amsint.com](http://www.amsint.com) 00
- [4] P. M. Athanas *Processor reconfiguration through instruction-set metamorphosis*. Computer, 26(3): 11-18, March 1993.
- [5] A. Hoffman, et al. *A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs)* IEEE Tr. on Computer-Aided Design of Integrated Circuits and Systems, v.20, n.11, Nov. 2001.
- [6] O. Schliebush, A. Hoffman et al. *Architecture Implementation Using the Machine Description Language LISA*. Proceedings of the IEEE VLSID'02.
- [7] Digital Core Design <http://www.dcd.pl/>. 2006
- [8] Cast Inc. <http://www.cast-inc.com>. 2006
- [9] OpenRisc <http://www.opencores.org>, 2006
- [10] CoWare SystemC. [www.coware.com](http://www.coware.com)
- [11] Forte Design Systems [www.forteds.com](http://www.forteds.com)
- [12] Chen-Wang *Inverse two dimensional DCT*, in Proceedings of the IEEE ASSP-32, pp. 803-816, August 1984