Actas de las IX jornadas de computación reconfigurable y aplicaciones

Universidad de Alcalá. Departamento de Electrónica
Alcalá de Henares, 9-11 Septiembre, 2009

Título: Actas de las IX jornadas de computación reconfigurable y aplicaciones

I.S.B.N.: 978-84-8138-832-9
Depósito Legal: M-32241-2009

Editores:

D. Raúl Mateos Gil
D. Ignacio Bravo Muñoz

Departamento de Electrónica
Escuela Politécnica
Universidad de Alcalá
28871 Alcalá de Henares

Diseño de Cubierta:

Juan José Villadangos Pombar

Imagen de portada:

© Ivan Cholakov | Dreamstime.com

Impresión:

Servicio de Publicaciones de la Universidad de Alcalá

Escuela Politécnica
Universidad de Alcalá
28871 Alcalá de Henares

Este libro ha sido editado utilizando LaTeX.

Índice general

Usando Python como HDL: Estudio comparativo de resultados basado en el desarrollo de un periférico real

Villar J.I.[1], Juan J.[1], Bellido M.J.[1], Ruiz-de-Clavijo P.[1], Guerrero D.[1], Muñoz A.[1]

[1] Grupo ID2, Dpto. de Tecnología Electrónica, E.T.S. Ing. Informática, Univ. de Sevilla, España,
{jose, jjchico, bellido, paulino, guerre, amrivera}@dte.us.es
http://www.dte.us.es/

Abstract. A muchos, el desarrollo de software y de hardware les pueden parecer disciplinas diferentes. Sin embargo existen grandes similitudes entre ellas que se han evidenciado con la aparición de extensiones para lenguajes de programación de propósito general orientadas a su uso como lenguajes de descripción de hardware. En este artículo analizamos la aproximación al uso de Python como HDL que propone el paquete MyHDL mediante un estudio comparativo de los resultados obtenidos al aplicar, de un modo independiente, flujos de diseño basados en Verilog y en Python para desarrollar un periférico real. El uso de este tipo lenguajes se revela tras esta experiencia como una potente y prometedora herramienta no solo debido a los sorprendentes resultados obtenidos, sino también a los nuevos horizontes que se abren con respecto al desarrollo de nuevas técnicas de modelado y verificación utilizando toda la potencia de un lenguaje de programación tan versátil como Python.

1 Introducción

El diseño de sistemas electrónicos digitales, desde sus inicios, ha estado marcado por la tónica de un constante incremento tanto de las prestaciones de las tecnologías de implementación como de la complejidad de los desarrollos a los que se enfrenta. Hace décadas que este hecho puso de manifiesto que las técnicas de diseño a muy bajo nivel no serían viables a largo plazo y que, por tanto, sería necesario el desarrollo y adopción de nuevas metodologías eficientes, que a un nivel de abstracción mayor, permitiesen abordar la creciente complejidad de las labores de modelado, verificación e implementación. Como respuesta a estas necesidades surgieron lenguajes de modelado [1] inspirados en los lenguajes de programación. Estos lenguajes de descripción de hardware o HDL (hardware description language) pueden ser definidos como lenguajes de programación con especiales capacidades para describir la naturaleza concurrente de la lógica digital y de los sistemas electrónicos. Los HDLs modelan la estructura y la evolución del hardware en las dimensiones del tiempo y del espacio.

Los diseños en HDL son utilizados para generar especificaciones del comportamiento y descripciones de la estructura del hardware. Estas especificaciones tienen una doble utilidad: simulación de las descripciones y síntesis orientada a la implementación sobre dispositivos físicos. Cuando estas especificaciones son utilizadas para simulación, se generan porciones de software ejecutable, que emulan, junto con las herramientas de simulación el comportamiento del hardware que se pretende especificar, para de este modo, poder verifi-

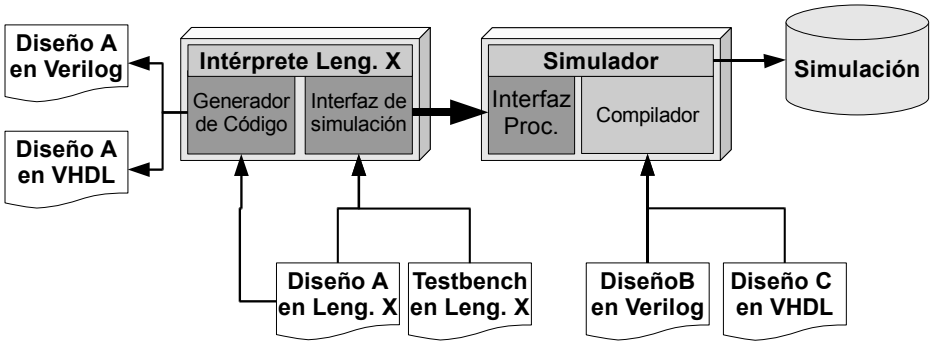

Fig 1. Adaptación genérica de un lenguaje interpretado de propósito general X para su uso como lenguaje de descripción de hardware

car su correcta operación. Estas porciones de código ejecutable son en definitiva las responsables de que los HDLs puedan ser vistos como lenguajes de programación. Por otra parte las herramientas de síntesis toman estas descripciones basadas en HDL para otro fin: inferir a partir del código fuente, elementos extrapolables a hardware real, para así generar descripciones a más bajo nivel de tal modo que tras una serie de etapas se llegue a un diseño implementable sobre un dispositivo físico. El hecho de que no todas las construcciones sean extrapolables a hardware hace que los HDLs tengan una capacidad expresiva mayor para simulación que para síntesis, por lo que es necesario que definan un subconjunto sintetizable bien especificado [2].

En la práctica, el mercado ha elevado a la categoría de estándar a dos HDLs: VHDL [3] y Verilog [4]. Actualmente su omnipresente soporte por parte de los distintos fabricantes así como la falta de argumentos definitivos de uno sobre el otro, hace que ambos constituyan el puente por el que cualquier diseño debe pasar para llegar a la tecnología de síntesis e implementación, independientemente de cual haya sido su HDL primitivo. Este hecho tiene como consecuencia que cualquier lenguaje de programación que quisiera tener capacidades de HDL, tan solo siendo capaz de generar código Verilog o VHDL a partir de su subconjunto sintetizable podría abstraerse de las herramientas y etapas de más bajo nivel.

Por otro lado, los simuladores tanto de Verilog como de VHDL han implementado interfaces procedimentales (VPI [4] en Verilog y VHPI [5] en VHDL), a través de las cuales, procesos software externos, desarrollados en cualquier lenguaje de programación para el que existan los correspondientes enlaces, podrían conectarse y gobernar el comportamiento de algunas señales en función del estado observado. Es en este hecho en el que se basa la posibilidad de simular el comportamiento de un sistema hardware utilizando únicamente rutinas software realizadas en cualquier lenguaje de programación conectadas al simulador mediante VPI o VHPI.

Tomando como base los dos puntos anteriores, parece factible transformar un lenguaje de programación de muy alto nivel en un HDL [6] [7] [8] siguiendo un esquema similar al que se muestra en la figura 1, de modo que éste sea válido tanto para su implementación final sobre un dispositivo físico utilizando VHDL o Verilog como lenguajes intermedios que lo independicen de la tecnología subyacente, como para su uso como HVL (Lenguaje de verificación de hardware), capaz de realizar cosimulación con otro módulos escritos en VHDL y Verilog, aportando toda la potencia expresiva de la que disponga el lenguaje bajo adaptación: construcciones de alto nivel, orientación a objetos, librerías, etc...

Este artículo presenta la adaptación concreta del lenguaje interpretado Python [9] basada en el paquete MyHDL [8] para transformarlo en una alternativa viable para el desarrollo

de hardware. Para caracterizar esta viabilidad se ha realizado un estudio comparativo de resultados mediante la realización del diseño de un periférico real utilizando de un modo independiente flujos basados en Verilog y MyHDL. De este modo, se pretende llegar a tener una primera impresión de la potencia de Python como herramienta de modelado de hardware, así como ser capaces de valorar tanto su potencia expresiva como la calidad del diseño generado automáticamente. Debemos hacer notar que aunque en algunos aspectos, puedan existir coincidencias entre el enfoque de MyHDL y el de otras soluciones como System C o System Verilog existe un rasgo que los diferencia de un modo inequívoco: mientras que estas otras soluciones son lenguajes cuyo propósito específico es la descripción de hardware a nivel de sistemas, Python es un lenguaje cuyo desarrollo ha sido totalmente ajeno a los requisitos específicos de un HDL y al que mediante una librería externa a su núcleo se le ha dado capacidad para modelar diseños hardware.

El artículo presenta la siguiente organización: en el apartado dos se analiza el funcionamiento de MyHDL resaltando sus principales características, ventajas e inconvenientes. En el apartado tres se presenta el diseño del periférico sobre el que se ha realizado la comparativa. El apartado cuatro detalla la metodología llevada a cabo en la comparativa así como los parámetros sobre los que se han medido ambas soluciones. En el apartado cinco se presentan y analizan los resultados obtenidos utilizando ambos flujos de desarrollo. Finalmente el apartado seis presenta las conclusiones más relevantes obtenidas a partir de esta experiencia.

2 Python como HDL

El desarrollo de HDLs basados en lenguajes de propósito general implica una serie de decisiones de diseño relativas al modo en que serán modeladas las características diferenciadoras del hardware y sobre cómo se harán explícitas las relaciones de concurrencia entre las distintas operaciones. A este respecto, las características particulares del lenguaje de programación bajo estudio juegan un papel fundamental, con un gran impacto en aspectos tan importantes como las diferentes técnicas de modelado o la generación de código. Otro aspecto de vital importancia sobre el lenguaje a adaptar, radica en si éste es de naturaleza compilada o interpretada, pues los lenguajes que utilizan intérpretes aportan una gran flexibilidad debido a su capacidad de autoanálisis y automodificación en tiempo de ejecución mediante las llamadas técnicas de introspección [10], que como veremos en los siguientes apartados, son de gran utilidad para este cometido. El lenguaje Python se encuentra, por su propia filosofía en un lugar de excepción para su aplicación como HDL. A lo largo de su desarrollo ha evolucionando profundamente para dar respuesta a las principales filosofías de diseño, constituyéndose como el lenguaje en que los distintos paradigmas: imperativo, funcional, orientado a objetos, etc.. se reconcilian para aglutinar a una de las comunidades de desarrolladores más ecléctica hoy día.

Siendo esta versatilidad y capacidad de afrontar casi a cualquier filosofía de diseño el signo diferenciador de Python, no resulta extraño que finalmente éste, se haya acercado al diseño de hardware a través de un conjunto de librerías que forman el paquete llamado MyHDL. La finalidad de este proyecto, como sus creadores la definen es “aportar a los diseñadores de hardware la simplicidad y la elegancia del lenguaje Python” [8]. Para mapear los elementos de un flujo de diseño de hardware, trataremos por separado las tres cuestiones principales: el modelado de hardware en Python, la simulación y verificación, y por úl-

timo la generación de código VHDL o Verilog utilizando técnicas de análisis introspectivo.

2.1 Modelado de hardware

La idea principal de MyHDL es el uso de generadores [11] y decoradores [12][13] para modelar la concurrencia. Los generadores son un tipo especial de función. Se diferencian de las funciones comunes en que son capaces de devolver valores recordando el punto del flujo de ejecución en el que se encuentran para continuar en las subsiguientes llamadas. Aparecieron por primera vez en el lenguaje CLU [14] y actualmente están siendo implementados en un creciente número de lenguajes entre los que se encuentra Python.

Los módulos hardware son modelados como funciones Python que devuelven un conjunto de generadores. De este modo, se aprovecha la semántica y estructura de las funciones Python para soportar características como jerarquías arbitrarias de componentes, asociación de puertos basados en su nombre, etc... Además de lo anterior, MyHDL define nuevas clases para modelar conceptos del hardware no existentes en el núcleo del lenguaje como las señales, que son utilizadas para establecer la comunicación entre los distintos generadores, los tipos enumerados con codificación configurable, para generar conjuntos de estados en FSMs o clases con operaciones orientadas al trabajo con números con o sin signo a nivel de bits.

2.2 Simulación y verificación

Si bien Python puede ser una herramienta útil para la implementación, es en el area de simulación y verificación donde ofrece un mayor incremento de beneficios con respecto a VHDL o Verilog.

En el ciclo tradicional de desarrollo de un diseño hardware, los costes relacionados con la verificación varían entre un 30% y un 80% [15], lo que tiene como consecuencia que los avances en este área tengan un gran impacto sobre el coste total. Python, al tratarse de un lenguaje de propósito general especialmente pensado para desarrollo rápido de aplicaciones [16], aporta toda su potencia haciendo que el desarrollo de la fase de pruebas sea una tarea realmente económica en tiempo y en coste.

Por otra parte el apartado de simulación no cuenta con ninguna restricción inherente al subconjunto sintetizable, por lo que se dispone de la ingente cantidad de funciones de Python y de todas sus extensiones. Esto abre la puerta a metodologías de pruebas provenientes del mundo del software [17] como los tests unitarios y el desarrollo de pruebas con la capacidad de interactuar con elementos de red, con otros procesos software, con capacidad de generar y analizar datos mediante el uso paquetes para calculo científico como SciPy [18], etc...

MyHDL implementa un simulador empotrado capaz de generar visualizaciones de las formas de onda generando archivos en el formato estándar VCD [4]. Además, puede ser utilizado como HVL [8] (hardware verification language) junto con simuladores externos para verificar diseños en Verilog mediante el uso de sus funciones de cosimulación.

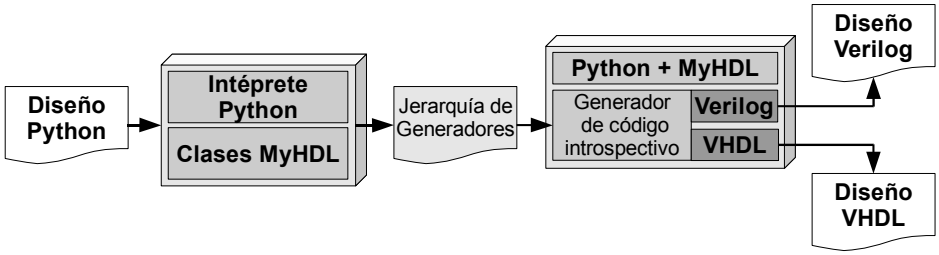

Fig 2. Proceso de generación de código Verilog y VHDL a partir de un diseño en Python con MyHDL

2.3 Generación de Verilog y VHDL a partir de Python

Al igual que en el resto de lenguajes de descripción de hardware, en MyHDL, la generación de código sintetizable no esta exenta de ciertas limitaciones. Dado que el convertidor de código genera Verilog o VHDL, el límite de la capacidad expresiva de MyHDL se circunscribe al subconjunto de elementos sintetizables comunes a ambos lenguajes. Sin embargo, esta capacidad expresiva resulta suficiente para definir un subconjunto convertible bastante más extenso que el subconjunto sintetizable estándar. Esto es posible debido a la aplicación previa a la conversión de código de una fase de elaboración del diseño y a que el convertidor realiza de manera automática labores como el análisis de la dirección (in, out o inout) de las señales, manejo transparente de aritmética con y sin signo, inferencia de tipos y tamaño de registros entre otras.

El convertidor sigue el esquema de funcionamiento ilustrado en la figura 2. En un primer paso realiza una elaboración del diseño en la que no se aplican las limitaciones de convertibilidad para obtener así una jerarquía de generadores Python sobre los que posteriormente, el generador de código hará una traducción a VHDL o Verilog. De este modo la limitaciones del subconjunto sintetizable se circunscriben únicamente al código interno de los generadores permitiendo toda la potencia de Python fuera de éstos. Esta aproximación a la generación de código permite modelar hardware usando elementos como listas de instancias, instanciación condicional, etc...

3 Descripción del diseño bajo estudio

Para realizar una comparativa de dos flujos de diseño cuyos resultados sean significativos, el desarrollo sobre el que ambos flujos serán contrastados debe ser representativo de la mayoría de desarrollos hardware e incluir por tanto los elementos más comunes dentro del modelado: bloques combinacionales y secuenciales, memoria, máquinas de estados finitos, etc...

Para elegir el diseño sobre el que se realiza la comparativa se han evaluado diversas aplicaciones en distintos rangos de complejidad. Uno de los más atractivos para este fin ha sido un controlador para displays de caracteres LCD compatible con los chips Sitronix ST7066U [19], Samsung S6A0069X o KS0066U, Hitachi HD44780 y SMOS SED1278 como el que se puede ver en figura 3. Este controlador resulta idóneo debido a que existen dos modos de comunicación entre el controlador y el chip, uno basado en ocho lineas de datos y otro más complejo basado tan solo en cuatro. Se ha elegido el protocolo de comu-

Fig 3. Fotografía del dispositivo en funcionamiento para el que se ha realizado el controlador sobre el que se basa la comparativa

nicación de cuatro líneas para dar cierta complejidad al diseño e incluir una máquina de estados anidada que gestione la transmisión. Por otra parte, estos chips necesitan de una fase previa de inicialización y configuración que también ha sido implementada como una máquina de estados anidada. De este modo, el núcleo del diseño está formado por una máquina de estados principal que invoca a dos submáquinas en distintos momentos, una para la inicialización y otra para la transmisión de datos.

Sobre el interfaz con el sistema, se han implementado dos soluciones a nivel de protocolo, y dos a nivel de almacenamiento. Sobre los protocolos, existe la opción de utilizar el controlador conectándolo directamente un bus de memoria. Como alternativa se ha desarrollado un módulo que envolviendo al anterior, ofrece un interfaz basado en el estándar Wishbone [20].

En lo que respecta al almacenamiento de los caracteres también se han implementado dos alternativas: almacenarlos en una memoria RAM interna al controlador con lo que las transacciones tardan un solo ciclo de reloj, o almacenarlos en la memoria del chip LCD que tiene una latencia del orden de microsegundos. Estas dos formas de almacenamiento, a las que hemos denominado RAM y RAMless respectivamente, en combinación con los dos protocolos de comunicación dan lugar a un espacio de cuatro diseños diferentes sobre los que comparar resultados.

4 Metodología de desarrollo y comparación

Para realizar la comparativa se ha establecido previamente una metodología tanto para el desarrollo de los diseños a comparar como para la extracción de los parámetros sobre los que serán medidos.

Se han tomado como punto de partida las especificaciones del chip a controlar para desarrollar un primer diseño funcional en Verilog basado en RAM interna y carente de todo tipo de optimizaciones. Sobre este diseño se han ido aplicando optimizaciones iterativamente hasta llegar a un punto de quiescencia en el que sin cambios en el protocolo no se han conseguido mejoras. Partiendo de este diseño se ha desarrollado una traducción directa a Python para obtener un desarrollo equivalente en lo que respecta a la estructura de módulos y de las máquinas de estados finitos. A partir de este punto se trabajará con ambos flujos de un modo independiente. Sobre el diseño en Python se inicia de nuevo un procedimiento iterativo de optimización que al igual que se hizo con el core en Verilog continuará hasta no obtener mejoras en iteraciones sucesivas.

Una vez obtenido los diseños óptimos en Verilog y Python para la versión con memoria RAM integrada, se han realizado las modificaciones necesarias para realizar el almacenamiento de caracteres en el chip externo del display LCD, suprimiendo así la necesidad de implementar una memoria en el interior de nuestros diseños. Con las versiones RAMless

se inicia un nuevo proceso de optimización independiente para cada diseño, hasta llegar otra vez a un punto de estancamiento de las mejoras en los parámetros medidos.

Otro aspecto bajo estudio ha sido el impacto que provoca añadir una capa de recubrimiento para modificar el interfaz del core con el exterior. Se ha desarrollado un módulo de recubrimiento que implementa un interfaz Wishbone tanto en Verilog como en Python. Sobre todos los diseños se han tomado medidas utilizando el core controlador con el interfaz mapeado en memoria y con el interfaz Wishbone. De este modo se pretende medir cómo afecta este recubrimiento al diseño final. Tomar una medida de la ocupación debida a la capa externa, con MyHDL representa un problema, debido a que éste genera el código instanciando toda la jerarquía en un solo módulo. De este modo, las herramientas de síntesis no pueden ofrecer datos pormenorizados referentes a cada módulo al contrario de lo que sí ocurre utilizando un diseño modular en Verilog. En estos casos se ha optado por medir el impacto del recubrimiento Wishbone de un modo indirecto, estimando la ocupación de la capa como la diferencia de la ocupación total entre la versión con y sin ésta.

Para cada uno de los tests se ha comprobado la equivalencia funcional con respecto al resto de diseños mediante la simulación y el contraste de las formas de onda observables desde el exterior almacenadas en archivos VCD.

De las tres dimensiones en las que clásicamente se comparan los diseños: ocupación, velocidad y consumo, nos hemos centrado exclusivamente en la de ocupación de recursos como medida indirecta de la capacidad expresiva de Python. Para ello se ha utilizado como herramienta de síntesis XST [21] con directivas de optimización de área con un nivel de esfuerzo 2 y generando el diseño para una FPGA Spartan 3E con medio millón de puertas equivalentes. De cada uno de los tests se han tomado medidas del uso de: LUTs, FFs y BRAMs.

Con esta metodología se pretenden comparar ambos flujos de diseño a partir de su aplicación a un problema real. Ambos son comparados desde la perspectiva de la capacidad de optimizar los recursos utilizando las diferentes técnicas de modelado que ofrece cada uno. De este modo se quiere determinar la calidad del código autogenerado por MyHDL así como su capacidad para hacer que las optimizaciones realizadas en Python tengan una repercusión directa en el diseño final. Asimismo se evalúan tanto la capacidad expresiva de Python como lenguaje de modelado, como la calidad del código desde el punto de vista de la legibilidad, tamaño y posibles beneficios con respeco al código generado por un ser humano.

5 Aplicación y resultados.

Para aplicar la metodología anteriormente descrita, se obtuvo un primer diseño funcional en Verilog que implementaba memoria RAM interna. En este primer diseño no se realizó ningún tipo de optimización, implementando del modo más claro y modular posible el funcionamiento descrito por las especificaciones del dispositivo. Sobre este diseño se fueron introduciendo mejoras a lo largo de doce iteraciones. Entre las mejoras introducidas se aplicaron técnicas como compartición de recursos, unificación de máquinas de estados, valores de salida en función de los bits del registro de estado, etc... Este proceso de optimización produjo un ahorro de recursos de en torno al 50%, tanto de LUTs como de FFs con respecto al diseño inicial; pasando de 435 a 212 LUTs y de 232 a 118 FFs. Un elemento sobre el que no hubo ningún ahorro durante este proceso fue el uso de un único bloque BRAM, inferido por XST para implementar la memoria interna de caracteres.

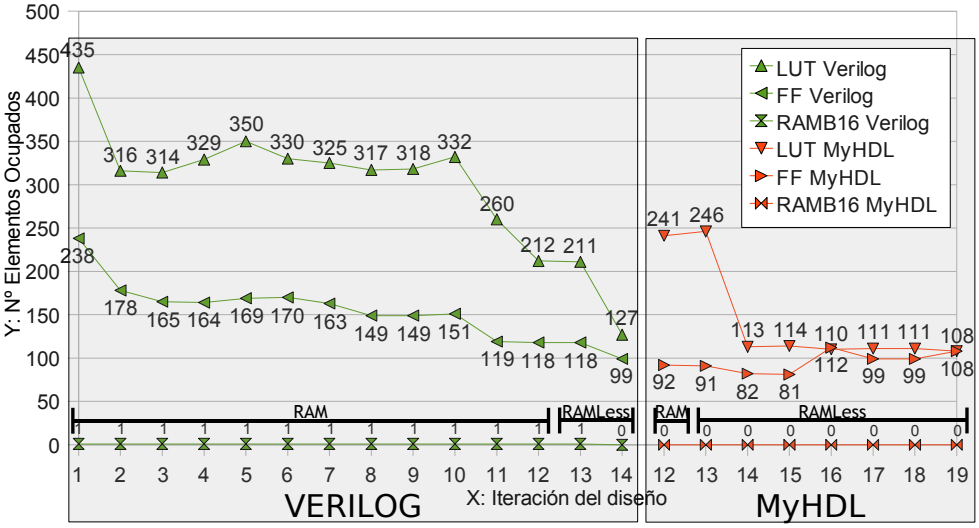

Fig 4. Gráfica de ocupación para los flujos basados en Verilog y MyHDL en las sucesivas iteraciones

Para comprobar si toda la batería de optimizaciones que se había aplicado utilizando Verilog era también aplicable a un diseño realizado con Python, se desarrolló una primera versión utilizando las técnicas de modelado de MyHDL. Esta primera versión fue una traducción a partir del core en Verilog, en la que se trató de respetar en la medida de lo posible su estructura y filosofía.

Los resultados obtenidos por la herramienta de síntesis a partir de esta primera versión en Python resultaron sorprendentes, pues no solo no fueron peores que los obtenidos mediante la aplicación concienzuda de técnicas de optimización de un modo manual, sino que generó unos resultados que debían ser estudiados con detalle, debido en su mayor parte a que la herramienta de síntesis no infirió los elementos del diseño del mismo modo. La versión en Python redujo en un 27% el uso de FFs y aumentó en un 13% el uso de LUTs con respecto al core óptimo en Verilog. Este aumento de LUTs se debe a que éstas fueron utilizadas para implementar la memoria RAM interna del core en lugar de utilizar un bloque BRAM, con lo que el ahorro en bloques BRAM en este caso fue del 100%. Sobre este primer diseño en Python no se consiguió aplicar ninguna optimización conducente a una mejora de la ocupación global.

Llegados a este punto, (iteración 12),no se encontraron mejoras en ninguno de los dos flujos que no alterasen de algún modo la filosofía de funcionamiento. Para continuar el proceso de optimización, se modificaron ambos cores para suprimir la necesidad de la memoria interna; ésta sería sustituida por la que implementa el chip externo al coste una una mayor latencia. Este cambio dio lugar a mejoras drásticas en la ocupación. En el caso del diseño en Verilog se dejó de ocupar el bloque BRAM y la reducción en el uso de LUTs y FFs fue respectivamente del 40% y del 16%. Para el caso del diseño en Python, en la iteración 15 se consiguió una reducción del 53% las LUTs y del 12% de los FFs.

Sobre estos diseños no se consiguieron optimizaciones globales, pero dado que las FPGAs suelen implementar en la misma proporción elementos de lógica y de almacenamiento, como es el caso de la Spartan 3E [22] sobre la que se ha realizado la comparativa, se trató de equilibrar el uso de LUTs y de FFs para así evitar que el uso excesivo de alguno de

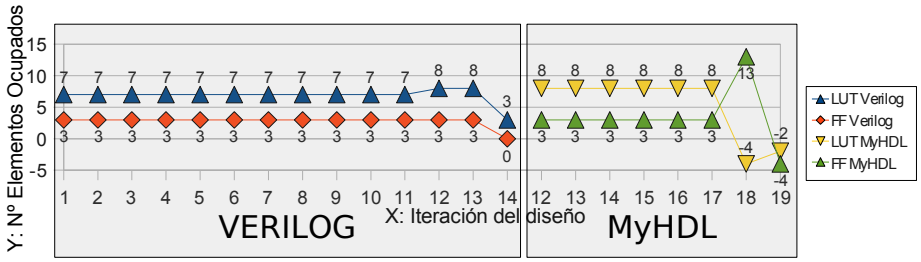

Fig 5. Impacto producido por la inclusión de un interfaz Wishbone durante las sucesivas iteraciones

ellos supusiera un incremento de la ocupación global tomando slices o CLBs como unidad de medida. Partiendo de los diseños óptimos conseguidos en la etapa anterior, en el caso de Verilog, no se consiguió una mejora en el equilibrio de de los dos tipos de elementos que no supusiese un incremento sustancial de la ocupación global. En el caso del diseño en Python, se consiguió el equilibrio óptimo con una cifra de 108 LUTs y 108 FFs, cuya suma, 216 elementos, resultó inferior que el mejor resultado obtenido con Verilog cuya suma fue de 226 (127 LUTs y 99 FFs).

Otro apartado evaluado ha sido el impacto de la adición de un módulo, que a modo de envoltura, dotase al core de un interfaz diferente. En este caso se desarrolló un interfaz basado en el estándar Wishbone, debido principalmente a su extendido uso en el mundo de los System-on-Chip [23]. Este interfaz mapeaba en memoria, además de la memoria interna del core, un registro de estado y uno de control. En este apartado, el característico modo que MyHDL utiliza para generar código, deshaciendo la jerarquía e instanciando todos los generadores en un mismo módulo, aporta ciertas ventajas a la vista de los resultados obtenidos. A lo largo de todas las iteraciones se han tomado medidas de ocupación de los diseños tanto con la interfaz nativa mapeada en memoria como añadiéndole el módulo conversor. En todos los casos con el ciclo basado en Verilog, la envoltura ha añadido cierta ocupación, llegando a ser mínima en la última iteración con tan solo 3 LUTs de sobrecarga. En el caso del flujo basado en MyHDL, la adición de este interfaz no solo no ha supuesto un coste extra, sino que se ha conseguido un ahorro en el mejor casos de 4 LUTs y 3 FFs. Intuimos que este resultado se debe a que al estar toda la jerarquía instanciada dentro de un solo módulo, la herramienta de síntesis ha sido capaz de eliminar aquellos elementos que dan generalidad al interfaz pero que dejan de ser útiles cuando todo se percibe como un único sistema, pudiendo suprimir aquellos elementos introducen una complejidad innecesaria en un caso concreto de aplicación.

Con esta prueba de concepto no debemos olvidar que lo que tratamos de evaluar no es la optimalidad del código autogenerado frente al código Verilog, sino la expresividad de Python mediante la capacidad de generar resultados equivalentes funcionalmente y en el mismo rango de prestaciones que los obtenidos manualmente. Optimizaciones a nivel de herramienta de síntesis, como el aplanamiento de la jerarquía, con gran seguridad habrían hecho casi indistinguibles los resultados obtenidos al añadir el interfaz Wishbone.

6 Conclusiones

Con esta experiencia hemos sido capaces de obtener una primera valoración de MyHDL como posible alternativa al flujo de desarrollo tradicional utilizando para ello su aplicación a un desarrollo real. Si bien, los resultados han sido obtenidos a partir de una muestra muy

reducida, son lo suficientemente positivos (superiores en algunos casos a los obtenidos con Verilog) como para justificar el inicio de una línea de trabajo para la obtención de nuevos resultados en este sentido.
Por otro lado, este tipo de herramientas abre toda una serie de nuevos horizontes en cuanto a metodologías y técnicas de modelado de alto nivel se refiere. El hecho de que tanto Python como MyHDL sean Software Libre [24] los convierte no solo en la potente herramienta que hemos podido comprobar, sino que también, debido a la posibilidad de modificarlos y adaptarlos a nuevos escenarios, los conforman como un potente laboratorio de investigación sobre HDLs en el que el mundo académico tiene la oportunidad de aplicar y poner en práctica sobre diseños reales todo tipo nuevas ideas dentro del campo de HDLs y nuevas metodologías.

Referencias

1. DeMichelli, G.: Modeling Languages and Abstract Models, Stanford University. 2006.
2. Damaševičius R.: A Subset-Based Comparison of Main Design Languages. Informacinės technologijos ir valdymas. - ISSN 1392-124X. - 1997. - Nr. 3(6). - pp. 20-29
3. IEEE Standards Board: IEEE standard VHDL language reference manual (integrated with VHDL-AMS changes), IEEE std 1076.1, 1997.
4. IEEE Standards Board: IEEE standard for verilog hardware description language, 2001. IEEE std. 1364-2005.
5. IEEE Standards Board: IEEE Standard VHDL Language Reference Manual Amendment 1: Procedural Language Application Interface, IEEE std 1076c-2007, 2007.
6. Guo S. Luk W.: Compiling Ruby into FPGAs. Will Moore and Wayne Luk, editors, Field-Programmable Logic and Applications, pages 188–197. Springer-Verlag, Berlin, / 1995.
7. Bellows P, Hutchings B.: JHDL - an HDL for reconfigurable systems. In: in Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines; 1998. p. 175-184.
8. Decaluwe J.: MyHDL Manual. Release 0.6. http://www.myhdl.org, 2009.
9. Van Rossum G. The Python Language Reference Manual. Network Theory Ltd.; 2003.
10. Sobel J, Friedman D.: An Introduction to Reflection-Oriented Programming, University of Indiana, 1996.
11. Python Enhancement Proposals: PEP 289: Generator Expressions, 2002.
12. Python Enhancement Proposals: PEP 318: Decorators for Functions and Methods, 2003.
13. Mertz D.: Charming Python: Decorators make magic easy; A look at the newest Python facility for metaprogramming. *IBM developerWorks*. 2006.
14. Liskov B.: “A History of CLU”, 1992.
15. Ur S., Ziv A.: Cross-Fertilization between Hardware Verification and Software Testing, in 6th IASTED International Conference on Software Engineering and Applications (SEA2002). Cambridge, USA, 2002.
16. Cockburn A.: Agile Software Development: The Cooperative Game (2nd Edition) (The Agile Software Development Series). Addison-Wesley Professional. 2006.
17. Runeson, P.: A survey of unit testing practices,IEEE Software Magazine, V23n4(Jul/Aug 2006) pp22-29
18. SciPy Project: Scipy 0.7 Reference Guide, http://docs.scipy.org/doc/. 2009.
19. Sitronix: 16Cx40S Dot Matrix LCDController/Driver Specification Sheet. 2008
20. Herveille, R. “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores”, Opencores, 2002.
21. Xilinx, Xilinx Synthesis Technology User Guide 9.2, 2007.
22. Xilinx, Spartan-3 Generation FPGA User Guide, 2009.
23. Kamath , U., Kaundin , R., System-on-Chip Designs , Strategy for Success, Wipro Technologies.
24. Free Software Foundation, GPL y LGPL License v2.1 http://www.gnu.org/licenses/.1999