RIELAC, Vol. 42 2/2021 p. 34-54 Mayo-Agosto ISSN: 1815-5928

RIELAC
Revista de Ingeniería Electrónica, Automática y Comunicaciones

Módulo de inferencia difuso con base de conocimientos variable sobre hardware reconfigurable

Alejandro José Cabrera Sarmiento, Santiago Sánchez-Solano, Yasmani García Guirola

RESUMEN / *ABSTRACT*

En este artículo se presenta el desarrollo de un módulo de inferencia difuso (FIM, por sus siglas en inglés) implementado sobre hardware reconfigurable con capacidad de modificar dinámicamente su base de conocimientos. El FIM original se diseña utilizando el entorno de desarrollo de sistemas difusos Xfuzzy, el cual permite la generación de código en lenguaje de descripción de hardware VHDL para la arquitectura del FIM. Posteriormente se modifica el código VHDL para añadir sendos puertos con las señales de dirección, datos y control de lectura/escritura a las memorias de antecedentes y de reglas que contienen la base de conocimientos. El FIM modificado se encapsula en un módulo de propiedad intelectual siguiendo dos posibles opciones, realizando en cada caso las interconexiones correspondientes, de forma tal que desde un sistema de procesamiento empotrado en el mismo dispositivo se pueda acceder a estas memorias a través de los puertos añadidos, posibilitando la modificación de sus contenidos en tiempo de operación. Las implementaciones fueron realizadas sobre dos tipos de dispositivos de hardware reconfigurable: un FPGA Spartan-3E1600, utilizando un sistema de procesamiento basado en el *softcore* Microblaze y el entorno de desarrollo ISE/EDK; así como sobre un SoC-FPGA Zynq-7Z010, utilizando su sistema de procesamiento *hardcore* basado en ARM y el entorno de desarrollo Vivado, comprobándose la modificación dinámica de la base de conocimientos del FIM. Las modificaciones realizadas facilitan el ajuste de la base de conocimientos de un controlador difuso híbrido hardware/software durante su etapa de desarrollo así como la implementación de un controlador difuso adaptativo.

Palabras claves: Módulo de inferencia difuso, Xfuzzy, FPGA, SoC-FPGA

This paper presents the development of a fuzzy inference module (FIM) implemented on reconfigurable hardware with the ability to dynamically modify its knowledge base. The original FIM is designed using the fuzzy systems development environment Xfuzzy, which allows the generation of VHDL hardware description language code for the FIM architecture. Subsequently, the VHDL code is modified to add ports with the address, data and read / write control signals to the antecedents and rule memories that contain the knowledge base. The modified FIM is encapsulated in an intellectual property module following two possible options, making the corresponding interconnections in each case, so that these memories can be accessed through the added port from a processing system embedded in the same device, enabling the modification of its contents in operation time. The implementations were carried out on two types of reconfigurable devices: a Spartan-3E1600 FPGA, using a processing system based on the Microblaze softcore and the ISE/EDK development environment; as well as on a SoC-FPGA Zynq-7Z010, using its hardcore processing system based on ARM and the Vivado development environment, checking the dynamic modification of the FIM knowledge base. The carried out modifications make it easy to fine-tune the knowledge base during the development stage of a hybrid hardware/software fuzzy controller as well as the implementation of an adaptive fuzzy controller.

Keywords: Fuzzy inference module, Xfuzzy, FPGA, SoC-FPGA

Fuzzy inference module with variable knowledge base on reconfigurable hardware

Recibido: 8/6/2021 Aceptado: 29/8/2021

1.- INTRODUCCIÓN

Desde su surgimiento la lógica difusa ha demostrado ser una potente herramienta para el desarrollo de sistemas de control debido a su capacidad para describir la operación de sistemas complejos mediante reglas simples del tipo *IF* <antecedente$_i$> *and* <antecedente$_j$> *and… THEN* <consecuente$_k$> expresadas en lenguaje natural en lugar de recurrir a modelos matemáticos cuya solución puede requerir una elevada capacidad computacional [1].

Independientemente de la aplicación, el componente fundamental de un controlador basado en lógica difusa (en adelante, controlador difuso) es el módulo de inferencia difuso (FIM). De forma general, un módulo de inferencia difuso se compone de tres etapas lógicas (fusificación, inferencia y defusificación) que interactúan con una base de conocimientos que contiene la información relacionada con los conjuntos difusos de las entradas y las salidas así como con el conjunto de reglas.

La etapa de fusificación se encarga de establecer el grado de similitud entre las entradas del FIM y las funciones de pertenencia asociadas a los conjuntos difusos correspondientes, mientras que la etapa de inferencia realiza la evaluación de cada una de las diferentes reglas en base a los grados de similitud de sus antecedentes y de la interpretación del conectivo utilizado, determinando su aporte a la salida. Finalmente, los aportes de cada una de las reglas son combinados en la etapa de defusificación para obtener el valor de la salida del FIM, existiendo diversas formas de interpretar los antecedentes de las reglas (conectivos de antecedentes) y de combinar los aportes de las diferentes reglas (métodos de defusificación).

Existen dos alternativas generales para implementar un FIM: mediante una función software ejecutada sobre un sistema de procesamiento; o mediante su implementación hardware, bien sea sobre un ASIC (*Application Specific Integrated Circuit*) o sobre un dispositivo de hardware reconfigurable, tal como un FPGA o un SoC-FPGA (*System on Chip-FPGA*). La primera es la alternativa más generalizada, siendo ejecutada sobre múltiples sistemas de procesamiento que van desde computadoras personales (PC, por sus siglas en inglés) hasta microcontroladores. Sin embargo, cuando existen restricciones de velocidad, muy frecuentes en sistemas empotrados donde coexisten restricciones de volumen, peso y/o consumo de potencia, es necesario recurrir a la implementación hardware del FIM, sobre todo utilizando dispositivos de hardware reconfigurable por su capacidad de reutilización.

Adicionalmente, tanto la disponibilidad de módulos de propiedad intelectual (IP, por sus siglas en inglés), consistentes en descripciones software de componentes hardware complejas (también conocidos como *softcore*), de todos los componentes de un sistema de procesamiento (procesador, controladores de memoria, interfaces de comunicación, dispositivos de entrada/salida, temporizadores, etc.) que facilitan su implementación en un FPGA, como la disponibilidad en los dispositivos SoC-FPGA de todo un potente sistema de procesamiento ya empotrado (componentes denominados *hardcore*) hacen factible el desarrollo de controladores difusos híbridos hardware/software (HW/SW), en los que el FIM se implementa mediante hardware mientras que el sistema de procesamiento se encarga de otras tareas como pueden ser la adquisición y pre procesamiento de las señales de entrada al FIM, las tareas de comunicación, etc.

Existen múltiples reportes recientes de implementaciones hardware de módulos de inferencia difusos sobre FPGA y SoC-FPGA para diversas aplicaciones, sobre todo utilizando dispositivos de Xilinx [2–20] y de Intel (ex Altera) [21–27], los dos principales fabricantes de dispositivos de hardware reconfigurable.

En [2,3,21–23] se exponen implementaciones para aplicaciones de control de seguidores solares mientras que en [4–12,20] se presentan implementaciones relacionadas con control de motores (velocidad, torque, posición) y otros dispositivos electromecánicos. En [13,14,18,24] se describen implementaciones de FIM para aplicaciones relacionadas con la robótica o sistemas operados a distancia. Otras implementaciones corresponden al control de un intercambiador de calor [19], de un convertidor DC-DC [25] así como para aplicaciones diversas.

Del análisis de estas implementaciones hardware de módulos de inferencia difusos resalta que, con la excepción de la expuesta en [6], todas se caracterizan por el carácter estático de su base de conocimientos por lo que, una vez implementadas y en operación, no pueden ser modificadas dinámicamente. La posibilidad de modificación dinámica de la base de conocimientos del FIM durante su operación reviste particular interés ya que facilita el ajuste de sus parámetros durante su etapa de desarrollo sin tener que recurrir a múltiples re-implementaciones y, sobre todo, constituye una premisa para la implementación de un controlador difuso híbrido HW/SW adaptativo, en el cual el sistema de procesamiento puede ejecutar algoritmos de aprendizaje, recalcular la información de la base de conocimientos y modificarla dinámicamente.

Los autores de [6] proponen una estrategia basada en la opción de reconfiguración dinámica parcial, presente en muchos dispositivos FPGA y SoC-FPGA actuales. En particular, proponen la modificación dinámica de las implementaciones de las funciones de pertenencia de la etapa de fusificación mediante reconfiguración parcial en un dispositivo SoC-FPGA Zynq-7Z020. Si bien esta estrategia tiene la ventaja de que permite modificar dinámicamente no solo la base de conocimientos del FIM sino toda su estructura (incluyendo los operadores de los conectivos de los antecedentes, el método de defusificación, etc.), tiene la limitante de que requiere disponer previamente de los ficheros de reconfiguración, por lo que no permite el

autoajuste de forma independiente (por ejemplo, por un sistema de procesamiento empotrado) de la base de conocimientos, es decir, tiene que depender de una computadora en la cual se obtenga el nuevo fichero de configuración, mediante el entorno de desarrollo del dispositivo programable utilizado.

Otra característica común extraída del análisis de las implementaciones descritas es que poseen estructuras muy heterogéneas. A esta heterogeneidad contribuye la diversidad de herramientas de CAD (*Computer Aided Design*) utilizadas para la descripción, simulación y, sobre todo, implementación de módulos de inferencia difusos. Si bien la mayoría utiliza MATLAB/Simulink para la descripción, simulación y ajuste del FIM, muchas implementaciones se realizan de forma independiente en algún lenguaje de descripción de hardware (HDL, por sus siglas en inglés) con estructuras muy diversas en función de los criterios de los diseñadores [2,3,5,6,9,10,12,14,16,21,23–25,27]. Otras utilizan la herramienta de generación de código HDL (*HDL Coder*) disponible en MATLAB [13,22] o la herramienta *System Generator* de Xilinx [4,7,11], la cual añade a Simulink una biblioteca de bloques para el diseño e implementación de sistemas digitales para los dispositivos de este fabricante, mientras que otras realizan la simulación en MATLAB/Simulink y realizan la implementación utilizando LabView sobre placas de desarrollo específicas [18,19], todas también con estructuras muy diversas. Las restantes implementaciones analizadas se realizan directamente en HDL, con estructuras que añaden mayor heterogeneidad.

Esta heterogeneidad de estructuras reportadas en las implementaciones de módulos de inferencia difusos no facilita la implementación de una estrategia para la modificación dinámica de sus bases de conocimientos. Para ello es recomendable disponer de una arquitectura uniforme que permita su adaptabilidad a muy diversos escenarios y que sea eficiente en términos de velocidad de inferencia y consumo de recursos.

En este artículo se expone el desarrollo de un módulo de inferencia difuso implementado sobre hardware reconfigurable con capacidad de modificar dinámicamente su base de conocimientos. En la sección 2 se aborda la implementación hardware de módulos de inferencia difusos utilizando el entorno de desarrollo de sistemas difusos Xfuzzy y su peculiaridad de síntesis hardware del FIM basada en una arquitectura eficiente, configurable y parametrizable que puede ser utilizada en muy diversos escenarios. La sección 3 expone el procedimiento general para la obtención de un controlador difuso híbrido hardware/software con base de conocimientos variable empotrado en un dispositivo programable, mientras que la sección 4 detalla las modificaciones a realizar al FIM original generado por Xfuzzy y las variantes para su conversión en un módulo de propiedad intelectual. La sección 5 resume las implementaciones de controladores difusos híbridos HW/SW realizadas sobre un FPGA Spartan-3E1600 y un SoC-FPGA Zynq-7Z010 utilizados para comprobar experimentalmente la modificación dinámica de la base de conocimientos del FIM. Por último, se presentan las conclusiones del trabajo.

2.- ENTORNO XFUZZY

El entorno de desarrollo de sistemas difusos Xfuzzy [28], desarrollado por investigadores del Instituto de Microelectrónica de Sevilla (IMSE, CSIC/Universidad de Sevilla) desde 1992, consiste en una colección de herramientas de libre distribución que permiten, entre otras facilidades, describir (editar), simular, ajustar y realizar implementaciones tanto software (C, C++, Java) como hardware, de módulos de inferencia difusos. La Figura 1 ilustra algunas de las interfaces gráficas de herramientas disponibles en Xfuzzy, en particular relacionadas con la descripción (herramienta *xfedit*) de un FIM así como la obtención de la superficie de control correspondiente (herramienta *xfplot*).

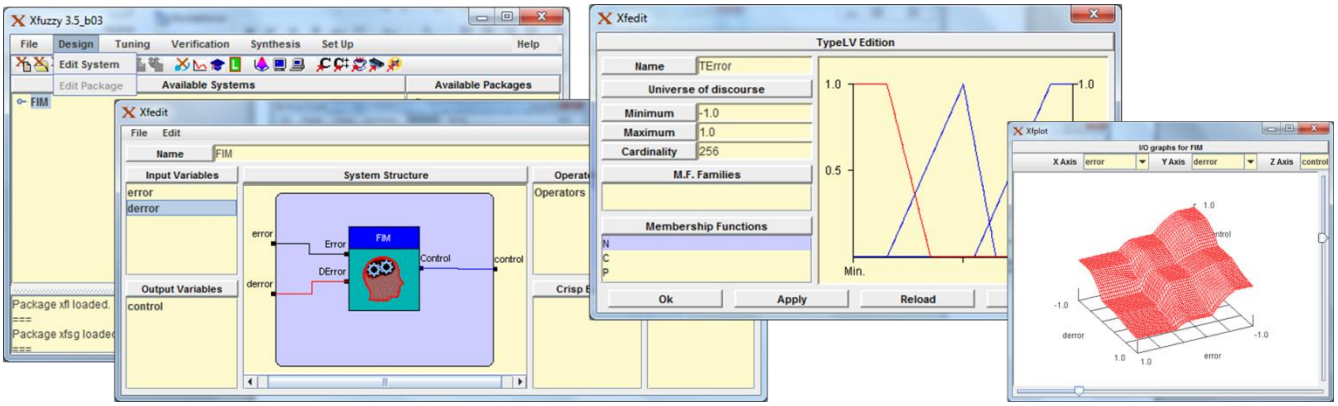

Figura 1

Breve ilustración del entorno de desarrollo Xfuzzy.

Una de las características más relevantes de Xfuzzy y que la distingue de otros entornos de desarrollo de sistemas de inferencia difusos, es su capacidad de síntesis hardware [29,30]. Esta implementación hardware puede hacerse de dos formas diferentes: 1) mediante la generación de código en lenguaje VHDL (*Very high speed integrated circuit HDL*) utilizando la herramienta *xfvhdl*; 2) mediante la generación de un modelo para Simulink, que puede ser implementado sobre los dispositivos de Xilinx mediante *System Generator*, utilizando la herramienta *xfsg*.

Existen diversos reportes de utilización de Xfuzzy en la descripción, simulación y/o implementación hardware de FIM. En [31] se expone la utilización de Xfuzzy para el desarrollo del módulo de inferencia difuso para el control de navegación de un vehículo autónomo, mientras que en [32] se describe la implementación de un controlador difuso PID supervisado, utilizando en ambos casos *xfvhdl* como herramienta de síntesis hardware. En [33] se utiliza Xfuzzy para la implementación de un controlador difuso jerárquico mediante la herramienta *xfsg*. En [34] se emplea Xfuzzy para describir, ajustar y simular un modelo de gestión de confianza basado en lógica difusa.

Otra característica relevante de Xfuzzy es que, independientemente de la herramienta de implementación hardware que se utilice, ambas están basadas en una arquitectura homogénea que puede ser aplicada a muy diversos escenarios, siendo además eficiente en términos de velocidad de inferencia y consumo de recursos. Esta característica de homogeneidad facilita la implementación de la estrategia para la obtención de un FIM con base de conocimientos variable.

2.1.- ARQUITECTURA DEL MÓDULO DE INFERENCIA DIFUSO

Las herramientas de síntesis hardware de módulos de inferencia difusos del entorno Xfuzzy están basadas en la arquitectura mostrada en la Figura 2, en la cual se ilustran los bloques fundamentales que intervienen en las etapas de fusificación, inferencia y defusificación, resaltando las memorias que almacenan la base de conocimientos del FIM: las memorias de antecedentes y la memoria de reglas. Todas las etapas son gobernadas por señales provenientes de una máquina de estados sincrónica con entrada de reloj *clk* que conforma el bloque de control [29–31].

Esta arquitectura está basada en tres aspectos que contribuyen a su elevada eficiencia, medida en términos del consumo de recursos del dispositivo programable y de velocidad de inferencia: 1) el procesamiento de reglas activas solamente, lo cual implica que no sean evaluadas aquellas reglas cuya contribución a la salida sea nula; 2) la limitación a dos del grado de solapamiento de los conjuntos difusos de las entradas, lo cual reduce de L^N a 2^N la cantidad de reglas a evaluar, siendo L la cantidad de conjuntos difusos de las entradas y N la cantidad de entradas del FIM; 3) la utilización de métodos de defusificación simplificados, que sustituyen la información de los consecuentes o conjuntos difusos de salida por parámetros que los representan.

El proceso de fusificación se realiza simultáneamente para cada una de las entradas mediante los circuitos de funciones de pertenencia (MFC, por sus siglas en ingles). Aunque existen dos formas para implementar los MFC, el componente fundamental en ambas es una memoria (memoria de antecedentes) que almacena la información que permite obtener a la salida dos pares de valores de etiquetas-grado de pertenencia (L_i, μ_i) correspondientes a los dos conjuntos difusos (debido al grado de solapamiento dos) del valor específico de la entrada. En el FIM de dos entradas ilustrado en la Figura 2 se obtendrán cuatro pares de valores L_i, μ_i, por ejemplo: L_{1a}, μ_{1a}; L_{1b}, μ_{1b}; L_{2m}, μ_{2m}; y L_{2n}, μ_{2n}, correspondientes a los conjuntos difusos L_a y L_b de un valor determinado de la entrada *Ent1*; y a los conjuntos difusos L_m y L_n de un valor determinado de la entrada *Ent2*.

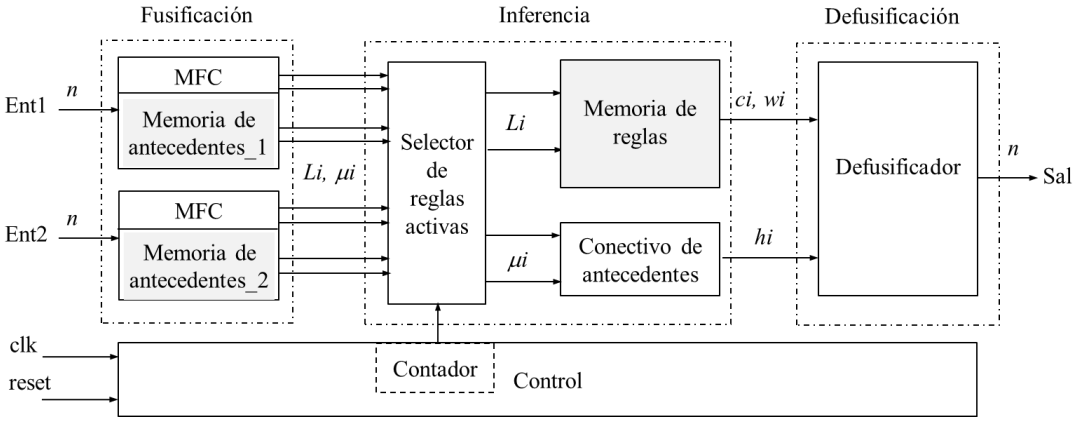

Figura 2

Arquitectura de un módulo de inferencia difuso de dos entradas y una salida.

Nótese que, para este caso, la cantidad de reglas a evaluar (reglas activas) será solamente de cuatro, independientemente de la cantidad de conjuntos difusos de cada una de las dos entradas. Estas cuatro reglas son las correspondientes a las cuatro posibles combinaciones de las etiquetas de salida de los MFC de la etapa de fusificación: L_{1a}, L_{2m}; L_{1a}, L_{2n}; L_{1b}, L_{2m}; y L_{1b}, L_{2n}.

A la entrada de la etapa de inferencia se encuentra el bloque selector de reglas activas, conformado por un arreglo de multiplexores gobernados por un contador presente en el bloque de control. Este bloque se encarga, por una parte, de combinar los valores de las etiquetas obtenidas de la fusificación de cada una de las entradas y, utilizando estas combinaciones como señales de dirección, ir accediendo secuencialmente a la memoria de reglas para obtener los valores que representan a los respectivos consecuentes. La información que se almacena en la memoria de reglas está en estrecha correspondencia con el modelo a seguir para el FIM (Mamdani o Takagi-Sugeno de orden cero) así como por el método de defusificación seleccionado. En la Figura 2 estos valores se representan por c_i, w_i, correspondientes a un modelo Mamdani utilizando la Media Difusa Ponderada como método de defusificación.

Simultáneamente a la obtención de los consecuentes de cada una de las reglas, los multiplexores del bloque selector de reglas activas van seleccionando las diferentes combinaciones de los grados de pertenencia μ_i de las entradas para obtener el grado de activación de la regla (h_i) mediante un operador de conectivo de antecedentes (operación mínimo u operación producto, según se seleccione).

A medida que se realiza la evaluación secuencial de cada una de las reglas, el bloque defusificador va procesando la información de salida de la etapa de inferencia, ya sea acumulando las contribuciones parciales de cada regla u obteniendo el consecuente de la regla con mayor grado de activación, en dependencia del método de defusificación seleccionado, de forma tal que, al finalizar el procesamiento de todas las reglas activas, solo puede ser necesario una operación de división para obtener el valor de la salida.

Nótese el elevado grado de paralelismo de esta arquitectura, tanto en la fusificación simultánea de todas las entradas como en la simultaneidad de los procesos de inferencia y defusificación como consecuencia de la utilización de métodos de defusificación simplificados. Para sacar provecho de este paralelismo las diferentes etapas se implementan de forma segmentada (*pipeline*) utilizando registros intermedios gobernados por el bloque de control, con lo cual se obtiene una elevada velocidad de inferencia.

Esta arquitectura es altamente configurable y parametrizable. La configurabilidad está dada por la cantidad de entradas que se pueden establecer, la existencia de diferentes opciones de implementación de los bloques MFC y de conectivo de antecedentes así como por la disponibilidad de diferentes métodos de defusificación simplificados, opciones que permiten al diseñador seleccionar aquellas que considere más pertinentes en función de la aplicación. La parametrización se establece mediante la cantidad de bits utilizados para codificar los diferentes parámetros, como los valores de las entradas y las salidas (*n* bits en la Figura 2), los grados de pertenencia μ_i, los valores de los consecuentes, etc. Debe tenerse muy presente que mientras mayor sea la cantidad de bits utilizados para representar los diferentes parámetros, mayor será el consumo de recursos del dispositivo programable y, en dependencia de que se requiera una operación de división en el bloque defusificador, menor podrá ser la velocidad de inferencia. En muchas aplicaciones pueden ser suficientes ocho bits para codificar los diferentes parámetros.

Dado que la estrategia para la obtención de un FIM con base de conocimientos variable seguida en este trabajo consiste en posibilitar la modificación de los contenidos de las memorias que almacenan dicha base, en el siguiente apartado se analizan las estructuras de las memorias de antecedentes y de la memoria de reglas.

2.2.- ESTRUCTURAS DE LAS MEMORIAS

Los bloques MFC pueden ser implementados utilizando tanto una tabla de búsqueda en memoria como mediante un circuito de cálculo aritmético, La utilización de esta última variante implica restricciones en cuanto a la forma de las funciones de pertenencia, limitadas a solo funciones triangulares y con valores de grado de pertenencia complementarios, por lo que en la memoria de antecedentes se almacenan los valores de la pendiente y del intersecto de cada uno de los segmentos de recta. En su operación, un circuito de cálculo aritmético va obteniendo los valores de la memoria de antecedentes y realizando el cálculo de cada segmento de recta hasta obtener el valor del grado de pertenencia correspondiente al valor de la entrada. Esta operación requiere de tantos ciclos de reloj como segmentos de recta existan.

Debido a las restricciones señaladas, esta variante es menos utilizada que la primera y no será objeto de análisis en el presente trabajo. No obstante, el procedimiento expuesto para modificar las memorias de antecedentes es perfectamente aplicable para esta variante.

La opción de implementar los bloques MFC mediante tablas de búsqueda en memoria para el almacenamiento de los antecedentes tiene las ventajas de no restringir las formas de las funciones de pertenencia de los conjuntos difusos de las entradas así como de requerir solo un ciclo de reloj para su operación.

La Figura 3 ilustra la organización de los contenidos de la memoria de antecedentes de una de las entradas utilizando esta variante de fusificación, en la cual los *n* bits con los que se codifican las entradas actúan como señales de dirección de la memoria.

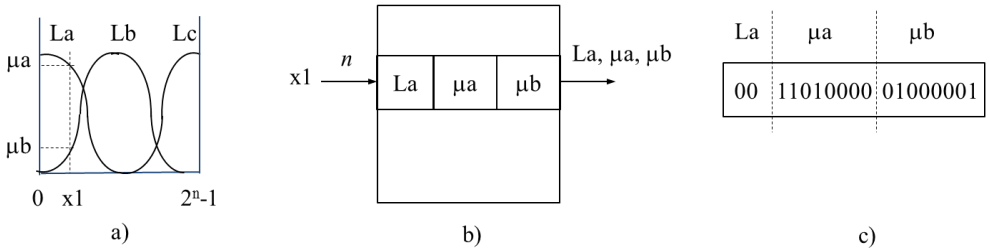

Figura 3

Ilustración de los contenidos de las memorias de antecedentes (a) Conjuntos difusos de una de las entradas (b) Contenido simbólico de la localización correspondiente al valor de entrada x1 de una memoria de antecedentes (c) Contenido binario de la localización x1.

Considerando que la entrada que es objeto de análisis tiene los tres conjuntos difusos de la Figura 3a se requiere una memoria de 2^n localizaciones, la cual almacena en cada localización el valor binario correspondiente a una de las etiquetas (el valor de la segunda etiqueta se obtiene mediante un simple incremento) así como los valores binarios de los grados de pertenencia μ_i de los dos conjuntos difusos correspondientes al valor de la entrada. Así, para el valor de entrada x_1 de la Figura 3a, la localización correspondiente de la memoria de antecedentes almacena la información mostrada de forma simbólica en la Figura 3b. Considerando que los grados de pertenencia han sido codificados con ocho bits, el contenido binario de la localización x_1 podría ser el mostrado en la Figura 3c. Note la utilización de dos bits para la codificación de una de las tres etiquetas.

De forma general, la capacidad total M_a de la memoria de antecedentes estará dada por la expresión (1), en donde P es la cantidad de bits utilizados para codificar el grado de pertenencia, L la cantidad de etiquetas y *n* el valor ya señalado.

$$M_a = 2^n . (2 . P + \lceil \log_2 L \rceil) \quad (1)$$

Note cómo la implementación de los bloques MFC mediante tablas de búsqueda en memoria puede sacar provecho de la disponibilidad de bloques de memoria RAM (BRAM) en todos los dispositivos programables actuales, tanto FPGA como SoC-FPGA. Dado que los BRAM actuales tienen capacidades que oscilan entre 18 y 36 kbits, en muchos casos basta con utilizar solo uno para cada entrada.

La estructura de la memoria de reglas es diferente a la de antecedentes, lo cual se ilustra en la Figura 4. En primer lugar la cantidad total de localizaciones que contienen los valores de los consecuentes de cada regla será de L^N (la cantidad total de reglas), siendo L la cantidad de etiquetas o conjuntos difusos de las entradas; y N la cantidad de entradas. Observe que este valor, en general, no es una potencia de dos ya que las cantidades típicas de etiquetas oscilan entre tres, cinco y siete. Sin embargo, dado que esta memoria es direccionada por las combinaciones de los bits de etiquetas de cada entrada, pueden existir combinaciones de estos bits (direcciones) que no sean válidas por lo que, de existir esas localizaciones, su contenido puede ser cualquiera ya que no corresponden a ninguna regla.

La Figura 4, correspondiente a la memoria de reglas del FIM de la Figura 2 con las funciones de pertenencia de las entradas de la Figura 3, ayuda a ilustrar esta situación. Al existir tres conjuntos difusos en cada entrada se utilizan dos bits para codificar las etiquetas de las funciones de pertenencia, por lo que las combinaciones de etiquetas de ambas entradas (L_1, L_2) de la Figura 4a serán de cuatro bits. Considerando que los consecuentes c_i y w_i se codifican ambos con ocho bits, el contenido binario de la localización 0100 podría ser el mostrado en la Figura 4b.

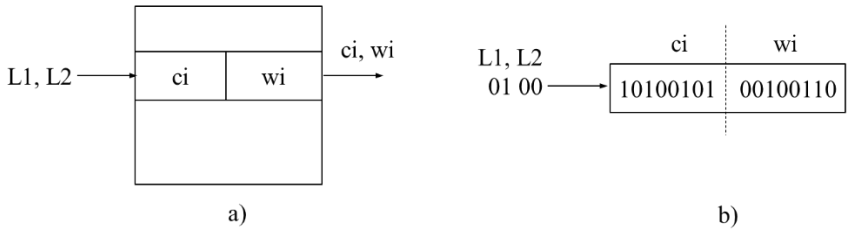

Figura 4

Ilustración de los contenidos de la memoria de reglas (a) Contenido simbólico de la dirección determinada por la combinación de las etiquetas L1 y L2 (b) Contenido binario de la localización 0100.

Note que si se utiliza una memoria convencional con decodificación total de sus entradas de dirección (como el caso de las BRAM), de las 16 localizaciones solo nueve contendrían información de los consecuentes; las siete restantes (correspondientes a las combinaciones 11xx y xx11 de los bits de etiquetas, donde x puede ser cualquier valor en ese bit) pueden contener cualquier valor.

Otra característica a considerar en la memoria de reglas es que, salvo FIM con muchas entradas y/o con muchos conjuntos difusos en cada entrada (que, por lo general, se descomponen en sistemas jerárquicos, cada uno con menor cantidad de entradas), la cantidad de localizaciones (cantidad de reglas) es un valor pequeño. Por ejemplo, apenas nueve para el FIM de la Figura 2; que se incrementaría a 49 en caso de tener siete conjuntos difusos en cada entrada.

Por otra parte, el contenido de cada una de las localizaciones dependerá del modelo del FIM a implementar y del método de defusificación a utilizar, pudiendo contener uno o dos parámetros para los modelos Mamdani y tres para los modelos Takagi-Sugeno.

Debido a estas características, en ocasiones puede ser preferible implementar la memoria de reglas de forma distribuida utilizando los recursos lógicos disponibles en el dispositivo programable en lugar de utilizar los bloques de BRAM.

2.3.- HERRAMIENTA XFVHDL

La herramienta *xfvhdl* es una de las disponibles en el entorno Xfuzzy para realizar el proceso de síntesis hardware, consistente en este caso en la obtención de código en lenguaje VHDL que describe el FIM previamente diseñado, basado en la arquitectura expuesta en la sección 2.1. Dada la existencia de múltiples opciones de configuración de los diferentes bloques del FIM, Xfuzzy incluye una biblioteca de componentes hardware parametrizables, de forma tal que con *xfvhdl* solo se genera el fichero de nivel superior del FIM que instancia los diferentes componentes de la biblioteca, así como un fichero de prueba (*testbench*) que permite simular su comportamiento.

La Figura 5 muestra una imagen de la interfaz gráfica de *xfvhdl*, en la cual se han resaltado algunos aspectos. Nótese en el recuadro izquierdo la parametrización de la cantidad de bits utilizados para representar las entradas, el grado de pertenencia, etc. Otros parámetros mostrados son extraídos del diseño del FIM, como la cantidad de funciones de pertenencia y la base de reglas.

El fichero del nivel superior del FIM generado por *xfvhdl* incluye un paquete de constantes, algunas de las cuales se corresponden con los diferentes parámetros seleccionados por el diseñador del FIM, mientras que otras son calculadas por la propia herramienta en función de la descripción del FIM realizada (por ejemplo, la cantidad de funciones de pertenencia; la cantidad de bits para codificar las etiquetas, etc.).

Un aspecto relevante para este trabajo es la forma en que se generen las descripciones de las memorias de antecedentes y de reglas, opciones que se muestran en el recuadro derecho de la Figura 5. Si se selecciona la opción de ROM, la descripción de la memoria se realiza mediante un arreglo de constantes que contiene todas las posibles direcciones con sus contenidos, la cual se incluye en el fichero del nivel superior del FIM. Si se selecciona la opción de RAM, en el fichero del nivel superior se instancia la componente de memoria correspondiente (`Antedecent_RAM.vhd` o `RulesMem_RAM.vhd`) disponibles en la biblioteca, las cuales corresponden a arreglos no inicializados con operaciones de lectura y escritura. En ambos casos la herramienta de síntesis del entorno de desarrollo del dispositivo programable utilizado puede inferir que estas memorias se implementen utilizando los bloques BRAM disponibles. Por último, la opción de bloques lógicos (*Logic Block*) genera la descripción de la memoria correspondiente en el fichero del nivel superior del FIM mediante una sentencia

CASE que establece las direcciones y sus correspondientes contenidos. En este caso la herramienta de síntesis del entorno de desarrollo infiere que estas memorias se implementen como memoria ROM distribuida utilizando los bloques lógicos del dispositivo programable, opción que puede ser deseable en ocasiones para implementar la memoria de reglas en base a lo planteado en la sección anterior.

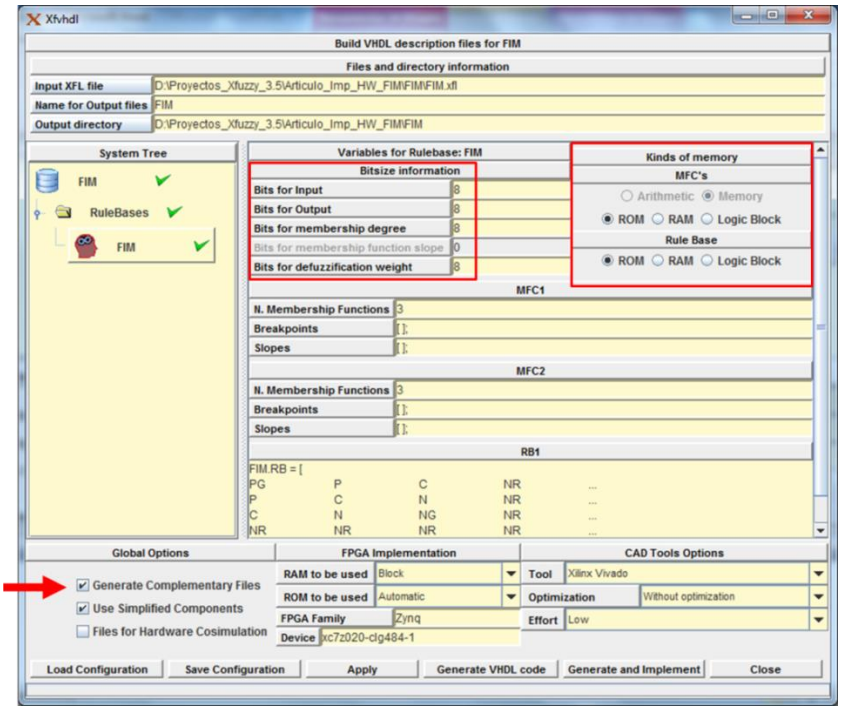

Figura 5

Ilustración de la herramienta xfvhdl.

Una característica importante de la herramienta *xfvhdl* para el presente trabajo es que permite obtener ficheros complementarios con los contenidos de las memorias de antecedentes y de reglas mostrados en las figuras 3c y 4b. Esta opción (resaltada con una flecha en la parte inferior izquierda de la Figura 5) permite la rápida obtención de la información correspondiente a diferentes bases de conocimientos que, una vez transformado el FIM, podrá ser modificada.

Por último, debe destacarse que aunque desde *xfvhdl* es posible invocar directamente la ejecución de las herramientas de síntesis de Xilinx (el principal fabricante de FPGA y SoC-FPGA), el código VHDL generado para el FIM así como los componentes de la biblioteca de Xfuzzy pueden ser utilizados en los entornos de desarrollo de dispositivos programables de cualquier otro fabricante.

3.- PROCEDIMIENTO GENERAL DE REALIZACIÓN

La Figura 6 ilustra el procedimiento general seguido en este trabajo para la obtención de un controlador difuso híbrido HW/SW con base de conocimientos variable empotrado sobre un dispositivo programable.

A partir de la descripción del módulo de inferencia difuso realizada en Xfuzzy, se lleva a cabo su implementación hardware mediante la herramienta *xfvhdl* obteniendo el código VHDL correspondiente a la arquitectura del FIM descrita en la sección 2.1. Posteriormente, mediante el entorno de desarrollo del dispositivo programable a utilizar, se modifica el código VHDL del FIM original para convertir en RAM doble puerto las memorias de antecedentes y de reglas que contienen la base de conocimientos. Seguidamente el FIM modificado se convierte en un módulo de propiedad intelectual de forma tal que pueda ser conectado a los buses del sistema de procesamiento empotrado en el mismo dispositivo, pudiendo acceder a los contenidos de las memorias de la base de conocimientos del FIM mediante los puertos añadidos. Por último, el programa de aplicación que se ejecute en el sistema de procesamiento del controlador difuso será el encargado, entre otras tareas, de cargar las memorias de antecedentes y de reglas del FIM con los contenidos de la base de conocimientos inicial , pudiendo modificar estos contenidos según se requiera y de forma dinámica durante la operación del controlador.

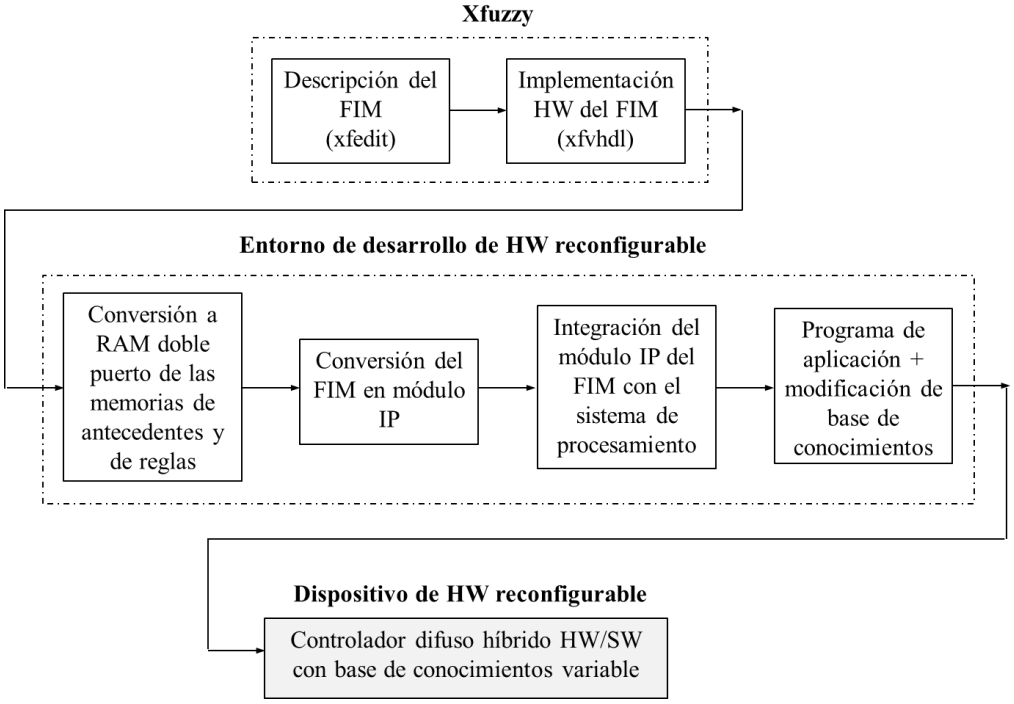

Figura 6

Procedimiento general de realización de un controlador difuso híbrido HW/SW con base de conocimientos variable.

Nótese que, además de la base de conocimientos inicial, las restantes pueden haber sido previamente calculadas (por ejemplo, con la ayuda de los ficheros complementarios generados con *xfvhdl* mencionados en la sección 2.3) y estar almacenadas en la memoria del sistema de procesamiento; o pueden ser recibidas a través de alguna interfaz de comunicación. Esta característica facilita el ajuste de la base de conocimientos de un controlador difuso durante su etapa de desarrollo ya que no requiere volver a implementar el hardware del FIM. Pero también las bases de conocimientos pueden ser calculadas dinámicamente, incluso por el propio sistema de procesamiento empotrado, sentando las bases para la implementación de un controlador difuso adaptativo.

Dado que la descripción del FIM y su implementación hardware con Xfuzzy ya han sido expuestas, en la próxima sección se detalla la obtención del FIM con base de conocimientos variable y su conversión en un módulo IP.

4.- FIM CON BASE DE CONOCIMIENTOS VARIABLE

La Figura 7 ilustra los elementos fundamentales para convertir el FIM generado por la herramienta de síntesis hardware *xfvhdl* de Xfuzzy en un FIM con base de conocimientos variable, en la cual se ha considerado que las entradas del FIM (Ent_i) se suministran desde un sistema de procesamiento empotrado en el mismo dispositivo programable, así como que la salida del FIM (Sal) se obtiene desde el sistema de procesamiento, conexión típica en un controlador difuso híbrido HW/SW.

Nótese que se incluye la arquitectura del FIM original mostrado en la Figura 2, en donde se convierten a memorias RAM doble puerto las memorias de antecedentes y de reglas, de forma tal que desde el sistema de procesamiento se generen las señales de dirección (Dir), datos de entrada (Din) y control de lectura/escritura (R/W) que permitan la escritura en la memoria a través del puerto añadido (Puerto B), así como la lectura de los datos de salida (Dout), por ejemplo, para verificar los valores de la nueva base de conocimientos escritos previamente. El puerto A de las memorias, sobre el cual solo se realizan operaciones de lectura, se utiliza para la operación normal del FIM.

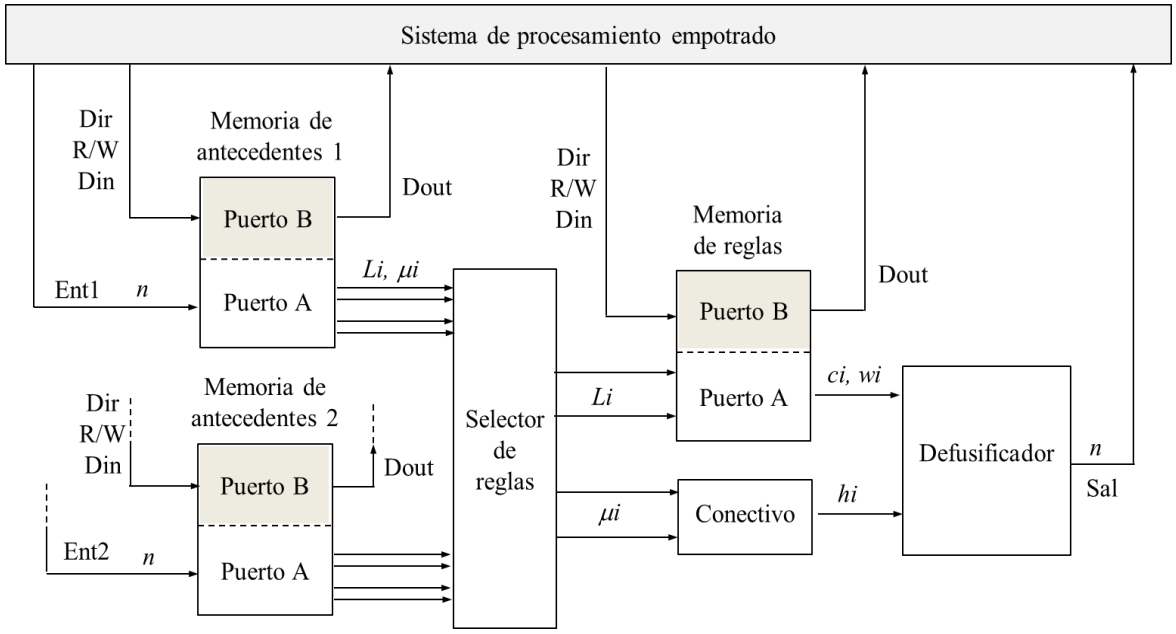

Figura 7

Ilustración del FIM con base de conocimientos variable.

Es importante resaltar que las modificaciones a realizar aprovechan la característica de todos los dispositivos programables actuales, tanto FPGA como SoC-FPGA, de incluir bloques de memoria RAM (BRAM) doble puerto, por lo que la conversión a RAM doble puerto de las memorias de la base de conocimientos del FIM no implicará un incremento apreciable en el consumo de recursos, siendo casi nulo para las memorias de antecedentes (normalmente ya implementadas en BRAM) y muy pequeño para la memoria de reglas en caso de ser implementada como memoria distribuida.

4.1.- MODIFICACIONES AL CÓDIGO VHDL DEL FIM

Tal como se expuso en la sección 2.3, para generar el código VHDL del fichero del nivel superior del FIM original se necesita seleccionar el tipo de memoria que será utilizado para generar las descripciones de las memorias de antecedentes y de reglas. Puede parecer obvio que, como finalmente ambos tipos de memorias serán RAM, esta sea la opción seleccionada para implementar el FIM. Sin embargo, dado que las descripciones de las memorias RAM de antecedentes y de reglas se encuentran en sendos ficheros de la biblioteca de Xfuzzy, esto implicaría modificar dichos ficheros (nada conveniente pues pueden ser utilizados por otras implementaciones de FIM) o añadir nuevos ficheros con memorias RAM doble puerto a la biblioteca de Xfuzzy, los cuales habría que instanciar también en el fichero del nivel superior del FIM en reemplazo de los componentes previamente generados. Es por ello que, para modificar solo el código VHDL del fichero del nivel superior del FIM y no modificar (ni añadir) ficheros en la biblioteca de Xfuzzy, se selecciona la opción de implementar todas las memorias como ROM en *xfvhdl*.

Las modificaciones a realizar consisten, en primer lugar, en agregar los puertos para las entradas de dirección, datos, control de lectura/escritura y reloj, así como para la salida de datos, en las entidades de las memorias de antecedentes y de reglas, modificando en correspondencia las respectivas arquitecturas. Todas estas modificaciones pueden ser realizadas y verificadas desde el entorno de desarrollo del dispositivo de hardware reconfigurable que se utilice.

El Código 1 muestra (en color rojo) las modificaciones realizadas al código VHDL de la entidad de una de las memorias de antecedentes mientras que el Código 2 muestra las modificaciones realizadas a la arquitectura de dicha memoria. Los parámetros `FIM_N`, `FIM_MFC1bits` y `FIM_grad` están incluidos en el paquete de constantes generado por *xfvhdl* en función del diseño del FIM (cantidad de bits para codificar las entradas, cantidad de bits para codificar las etiquetas y cantidad de bits para codificar el grado de pertenencia, respectivamente).

```
entity FIM_AntecedentMem1 is
port (
    -- Incorporacion del nuevo puerto a la memoria
    ----------------------------------------------------------------------------------------
     Addr_A1     : in std_logic_vector (FIM_N - 1 downto 0);
     Din_A1      : in std_logic_vector (FIM_MFC1_bits + 2*FIM_grad downto 1);
     We_A1       : in std_logic;
     Clk_A1      : in std_logic;
     Dout_A1     : out std_logic_vector (FIM_MFC1_bits + 2*FIM_grad downto 1);
    ----------------------------------------------------------------------------------------
    -- Puertos originales de la memoria de antecedentes
     pipe        : in std_logic;
     addr        : in std_logic_vector (FIM_N - 1 downto 0);
     alpha_1     : out std_logic_vector (FIM_grad downto 1);
     alpha_2     : out std_logic_vector (FIM_grad downto 1);
     L_1         : out std_logic_vector (FIM_MFC1_bits downto 1);
     L_2         : out std_logic_vector (FIM_MFC1_bits downto 1));
end FIM_AntecedentMem1;
```

Código 1

Modificaciones al código VHDL de la entidad de una de las memorias de antecedentes.

Nótese en la arquitectura la eliminación del arreglo de constantes que establecen los contenidos de la memoria de antecedentes (generada como ROM) del FIM original, así como el arreglo que define la memoria RAM (RAM_TYPE), sus localizaciones (RAM_WORD) y las conexiones para las operaciones de lectura y escritura a través de los puertos adicionados. Las modificaciones a realizar en las restantes memorias son similares.

Posteriormente, habrá que modificar la declaración de las componentes de las memorias de antecedentes y de reglas en correspondencia con los puertos añadidos; incorporar estos puertos en la entidad del fichero del nivel superior del FIM modificado (`FIM_M.vhd` en lo adelante) y, por último, instanciar las componentes de las memorias modificadas con las conexiones correspondientes a los puertos adicionados.

4.2.- MÓDULO IP DEL FIM CON BASE DE CONOCIMIENTOS VARIABLE

Siguiendo el procedimiento general expuesto en la sección 3, después de realizar las modificaciones al código VHDL del FIM para convertir las memorias de la base de conocimientos a doble puerto, se procede a convertir el FIM modificado en un módulo de propiedad intelectual compatible con el bus de expansión utilizado por el sistema de procesamiento empotrado en el dispositivo programable. Este proceso consiste en insertar el código VHDL del FIM modificado en el código VHDL de la interfaz del bus de expansión, añadiendo los recursos (registros, interfaces de interrupción, controladores de memoria, etc.) que permitan la interconexión entre ambos. De esta forma, el módulo IP del FIM podrá ser directamente conectado al bus de expansión del sistema de procesamiento, de forma similar a como se conectan los módulos IP de otros dispositivos, como temporizadores, interfaces de comunicación, etc., así como ser fácilmente accedido desde el programa que se ejecute en el procesador.

Los entornos de desarrollo de los diferentes fabricantes de FPGA y SoC-FPGA incluyen herramientas que facilitan la conversión a módulo IP de una descripción de hardware de usuario, las cuales liberan al diseñador de dominar los detalles de implementación de la interfaz con el bus de expansión utilizado por el sistema de procesamiento. Sin embargo, aunque similares en su propósito, no todas presentan las mismas características, por lo que el proceso de conversión del FIM a módulo IP será ilustrado con las herramientas disponibles en los entornos de desarrollo de Xilinx, tanto Vivado como ISE Design Suite, por ser los utilizados en las validaciones experimentales del presente trabajo.

Para la conversión de hardware de usuario a un módulo IP, las herramientas correspondientes de Vivado (*Create and Package New IP*) y de ISE/EDK (*Create or Import Peripheral*) generan dos ficheros de plantillas en lenguaje VHDL con estructura jerárquica, en donde el del nivel superior incluye las señales de interfaz con el bus mientras que el segundo incluye las funcionalidades seleccionadas por el usuario. Estas funcionalidades (o servicios) pueden ser, entre otras, el que funcione como maestro o como esclavo; la adición de una determinada cantidad de registros de lectura/escritura; incluir interfaces de interrupción, etc. Posteriormente, el diseñador debe incorporar a este fichero el hardware de usuario (en este caso, el del FIM modificado) e interconectarlo apropiadamente con los recursos asociados a los servicios seleccionados.

```
architecture FPGA of FIM_AntecedentMem1 is
signal s_addr : std_logic_vector (FIM_N - 1 downto 0);
signal s_data : std_logic_vector (2*FIM_grad + FIM_MFC1_bits downto 1);
-- Se modifican las lineas siguientes
-- subtype ROM_WORD is std_logic_vector (2*FIM_grad + FIM_MFC1_bits - 1 downto 0);
  subtype RAM_WORD is std_logic_vector (2*FIM_grad + FIM_MFC1_bits - 1 downto 0);
-- type ROM_TABLE is array (0 to 2**FIM_N - 1) of ROM_WORD;
   type RAM_TYPE is array (0 to 2**FIM_N - 1) of RAM_WORD;
-- Se eliminan todas las lineas del arreglo de constantes
-- constant ROM : ROM_TABLE := ROM_TABLE'(
-- Se agregan las señales siguientes:
signal RAM: RAM_TYPE;
signal s_addr_A1 : std_logic_vector (FIM_N - 1 downto 0);

begin
    s_addr     <= addr when rising_edge (pipe);
 -- s_data     <= ROM (conv_integer (s_addr));
    s_data     <= RAM (conv_integer (s_addr));
    L_1        <= s_data (2*FIM_grad + FIM_MFC1_bits downto (2*FIM_grad) +1);
    L_2        <= s_data (2*FIM_grad + FIM_MFC1_bits downto (2*FIM_grad) +1) + '1';
    alpha_1    <= s_data (2*FIM_grad downto FIM_grad+1);
    alpha_2    <= s_data (FIM_grad downto 1);
  -- Lectura del puerto incorporado.
    s_addr_A1  <= Addr_A1 when rising_edge (Clk_A1);
    Dout_A1    <= RAM (conv_integer (s_addr_A1));
  -- Escritura en el puerto incorporado
    RAM (conv_integer (Addr_A1)) <= Din_A1 when rising_edge (Clk_A1) and We_A1 = '1';
end FPGA;
```

Código 2

Modificaciones al código VHDL de la arquitectura de una de las memorias de antecedentes.

Adicionalmente, para facilitar el desarrollo de las aplicaciones software que utilicen el módulo IP de usuario, estas herramientas también crean funciones (*drivers*) en lenguaje C asociadas a los servicios incorporados, por ejemplo, funciones para escribir en un registro o habilitar la interrupción.

La opción más general para convertir el FIM modificado en un módulo IP, disponible tanto en Vivado como en ISE/EDK, es utilizar la funcionalidad de añadir registros de entrada/salida mapeados en el espacio de direcciones del procesador, a través de los cuales el programa que se ejecute en el sistema de procesamiento empotrado puede enviar a las memorias de antecedentes y de reglas las señales de dirección, datos de entrada y control de lectura/escritura, así como leer el valor de los datos de salida de las memorias. Nótese que el procesador no lee ni escribe directamente en las memorias sino que lo hace a través de los registros adicionados.

Esta opción ha sido la utilizada en Vivado con la herramienta *Create and Package New IP* para crear el módulo IP (denominado FIM versión 1.0) para la interfaz AXI-Lite del sistema de procesamiento *hardcore* PS7 empotrado en el SoC-FPGA Zynq-7Z010 ilustrada en la Figura 8, en donde se utilizan dos registros de 32 bits para acceder a cada uno de los puertos adicionados a las memorias de la base de conocimientos del FIM. Nótese que en un mismo registro se combinan las entradas de datos (Din), las entradas de dirección (Dir) y la de control de lectura/escritura, mientras que el segundo registro se utiliza para la lectura de la salida de datos (Dout) de la memoria. Además, se utilizan dos registros de escritura para proporcionar la información de entrada al FIM (solo se muestra Ent_1) y un tercero de lectura para obtener el valor de la salida (no mostrado en la figura).

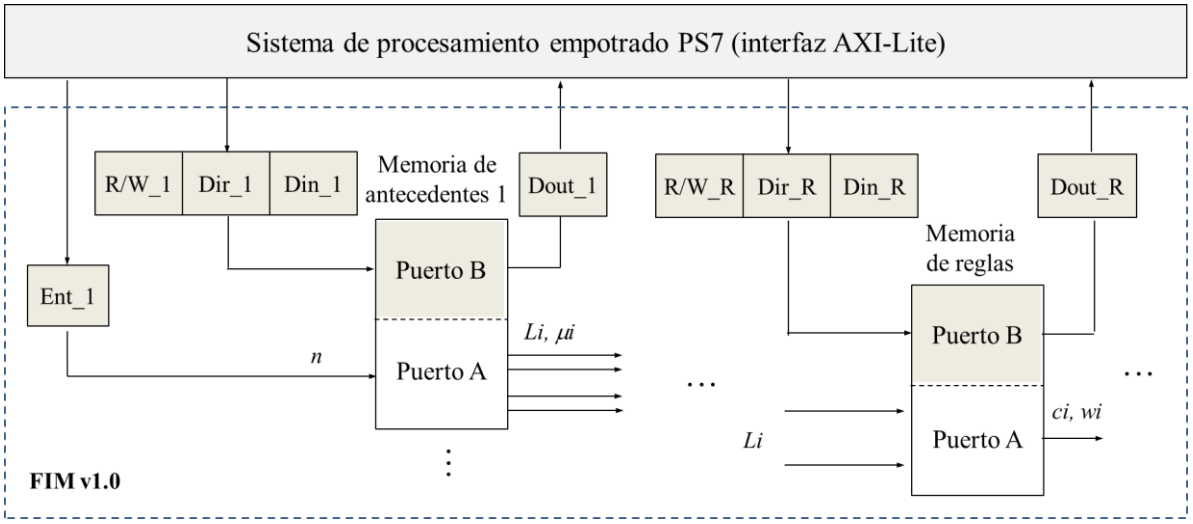

Figura 8

Ilustración del módulo IP del FIM con adición de registros.

Con esta opción, en el fichero de plantilla `FIM_v1_0_S00_AXI.vhd` generado por la herramienta *Create and Package New IP* se incluye la descripción y la lógica de selección y control de lectura y escritura de los nueve registros seleccionados por el diseñador. Posteriormente es necesario incluir la declaración de la componente del FIM modificado y realizar apropiadamente su interconexión con cada uno de los registros.

Aunque son varias las conexiones y/o modificaciones a realizar en el fichero `FIM_v1_0_S00_AXI.vhd`, en el Código 3 se muestra un fragmento con las conexiones del componente del FIM modificado, en donde se ha resaltado en rojo los puertos añadidos a las memorias de la base de conocimientos y en azul las señales de reloj y reset de la interfaz AXI.

Nótese la conexión de las señales de dirección, datos de entada y control de lectura/escritura de las memorias de antecedentes y de reglas a las señales **slv_reg**X (resaltadas en verde) que se conectan a los registros 0, 2 y 4 respectivamente, mientras que la salida de datos de estas memorias se conectan mediante las señales **mem_data_out_**X a los registros 1, 3 y 5 respectivamente. En la parte inferior del código aparecen las conexiones de las entradas del FIM a las señales **slv_reg**X que se conectan a los registros 7 y 8, mientras que la salida del FIM se conecta al registro 9 mediante la señal **reg_data_out**.

Una vez convertido el FIM en un módulo IP estará disponible para ser conectado a la interfaz AXI de forma similar a la conexión de cualquier otro módulo IP disponible en Vivado con esta interfaz.

El programa que se ejecute en el sistema de procesamiento PS7 será el encargado de cargar las memorias de antecedentes y de reglas con la información correspondiente a la base de conocimientos inicial del controlador difuso, utilizando para ello la función de escritura en registros (`FIM_mWriteReg`) incluida en el *driver* generado (también se puede verificar la escritura de estos valores mediante la función de lectura en registros `FIM_mReadReg`). Posteriormente, el programa podrá modificar los contenidos de las memorias en cualquier momento y tantas veces como sea necesario.

Otra opción, disponible en el entorno de desarrollo de sistemas de procesamiento empotrado EDK (*Embedded Development Kit*) de ISE utilizando la herramienta *Create or Import Peripheral* para el bus de expansión PLB, consiste en añadir, además de los registros de entrada/salida para la operación normal del FIM, la funcionalidad (o servicio) de mapeo de memorias de usuario en el espacio de direcciones del procesador. Al seleccionar esta opción, en los ficheros de plantilla generados (denominados ahora `FIM.vhd` y `user_logic.vhd`) se añade la descripción de circuitos de selección y control de un conjunto de memorias (tres, para el caso del FIM utilizado), mediante los cuales es posible accederlas directamente con las señales del bus de expansión PLB, es decir, directamente desde el procesador. Esto hará que, después de ser implementado el módulo IP del FIM, este contenga tres zonas de direcciones de memoria que deben ser asignadas dentro del espacio de direcciones del procesador. Dado que también se incluyen tres registros, sus direcciones también estarán mapeadas en el espacio de direcciones del procesador

```
--- Conexion del componente de la logica de usuario (top level de FIM_M)
FIM_1 : FIM_M
…
port map (
   Addr_A1  => slv_reg0 (25 downto 18),   -- Direccion (8 bits) para la mem Ant1 proveniente del registro 0
   Din_A1   => slv_reg0 (17 downto 0),    -- Dato de entrada (18 bit) para la mem Ant1 proveniente del registro 0
   We_A1    => slv_reg0 (31),             -- Control de L/E (bit MSB del registro 0)
   Clk_A1   => S_AXI_ACLK,                -- Señal de reloj de la interfaz AXI
   Dout_A1  => mem_data_out_A1,           -- Salida de datos (18 bts) de la mem de Ant1 (se conecta al registro 1)

   Addr_A2  => slv_reg2 (25 downto 18),   -- Direccion (8 bits) para la mem Ant2 proveniente del registro 2
   Din_A2   => slv_reg2 (17 downto 0),    -- Dato de entrada (18 bit) para la mem Ant2 proveniente del registro 2
   We_A2    => slv_reg2 (31),             -- Control de L/E (bit MSB del registro 2)
   Clk_A2   => S_AXI_ACLK,                -- Señal de reloj de la interfaz AXI
   Dout_A2  => mem_data_out_A2,           -- Salida de datos (18 bts) de la mem de Ant2 (se conecta al registro 3)

   Addr_R   => slv_reg4 (21 downto 18),   -- Direccion (4 bits) para la mem de reglas proveniente del registro 4
   Din_R    => slv_reg4 (15 downto 0),    -- Dato de entrada (16 bit) para la mem de reglas proveniente del registro 4
   We_R     => slv_reg4 (31),             -- Control de L/E (bit MSB del registro 4)
   Clk_R    => S_AXI_ACLK,                -- Señal de reloj de la interfaz AXI
   Dout_R   => mem_data_out_R,            -- Salida de datos (16 bits) de la mem de reglas (se conecta al registro 5)

-- Conexion del FIM a los registros de E/S adicionados
   clk   => S_AXI_ACLK,                   -- Señal de reloj de la interfaz AXI
   reset => NOT (S_AXI_ARESETN),          -- Señal de RESET de la interfaz AXI
   in1   => slv_reg7  (7 downto 0),       -- Entrada 1 de datos (8 bits) del FIM proveniente del registro 7
   in2   => slv_reg8  (7 downto 0),       -- Entrada 2 de datos (8 bits) del FIM proveniente del registro 8
   out1  => reg_data_out  (7 downto 0) ); -- Salida de datos (8 bits) del FIM (se conecta al registro 9)
```

Código 3

Fragmento de código VHDL con interconexiones realizadas en `FIM_v1_0_S00_AXI.vhd`.

Al incluir estos servicios, además de las funciones para leer y escribir en los registros adicionados, el *driver* generado incluye funciones para leer y escribir directamente en una dirección de memoria (`FIM_mReadMemory` y `FIM_mWriteMemory`).

Esta opción de utilización de los servicios de mapeo de memorias y de registros de lectura/escritura ha sido utilizada para crear el módulo IP (denominado FIM) para el bus PLB del sistema de procesamiento basado en el *softcore* del procesador Microblaze empotrado en un FPGA Spartan-3E1600 ilustrada en la Figura 9a, en la que se muestran los bloques de selección y control de una de las memorias de antecedentes y de la memoria de reglas (así como uno de los registros adicionados para proporcionar la información de entrada al FIM). La Figura 9b muestra el mapa de memoria resultado de la conexión del módulo IP del FIM al sistema de procesamiento, en donde se aprecian las tres zonas correspondientes a las memorias de antecedentes y de reglas, así como la zona correspondiente a los tres registros de entrada/salida.

Para incluir el hardware de usuario del FIM modificado se procede de forma similar a lo expuesto con la opción anterior. Utilizando ahora el fichero de plantilla `user_logic.vhd` generado por la herramienta *Create or Import Peripheral* (que incluye la descripción y la lógica de selección y control de lectura y escritura de las tres memorias y de los tres registros seleccionados por el diseñador), se incluye la declaración de la componente del FIM modificado y se realiza apropiadamente su interconexión con las señales de dirección, datos y control de lectura/escritura del puerto añadido a las memorias de antecedentes y de reglas, así como con cada uno de los tres registros para las entradas y salida del FIM.

Aunque son varias las conexiones y/o modificaciones a realizar en el fichero `user_logic.vhd`, en el Código 4 se muestra un fragmento con las conexiones del componente del FIM modificado, en donde se ha resaltado en rojo los puertos añadidos a las memorias de la base de conocimientos y en azul las señales del bus PLB.

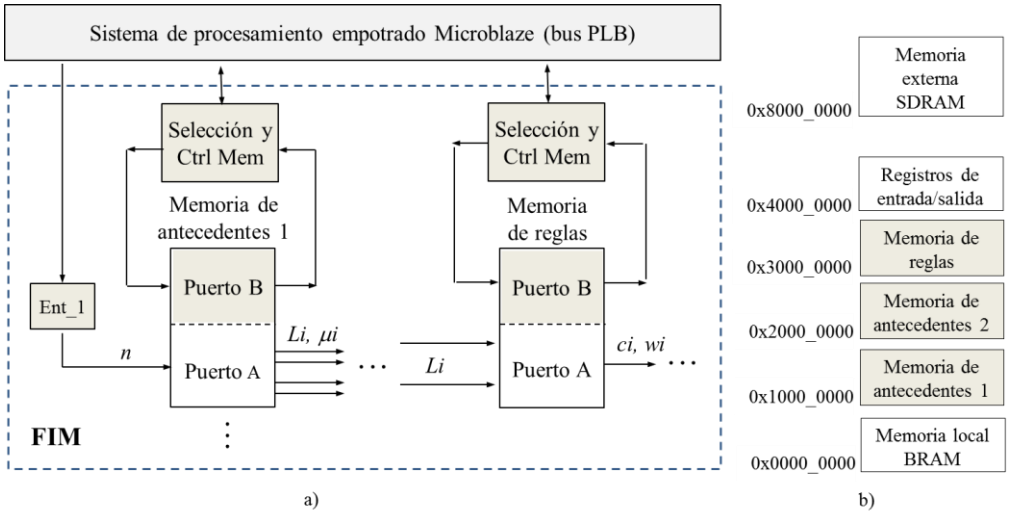

Figura 9

(a) Ilustración del módulo IP del FIM con mapeo de memorias (b) Mapa de memorias.

```
--- Conexion del componente de la logica de usuario (top level de FIM_M)
FIM_1 : FIM_M
...
port map (
   Addr_A1  => mem_address,          -- Señal de direccion (8 bits) para la memoria de antecedentes 1.
   Din_A1   => Bus2IP_Data (13 to 31), -- Dato de 18 bit proveniente del bus PLB
   We_A1    => memA1_write,          -- Control de L/E (incluye la seleccion de la memoria)
   Clk_A1   => Bus2IP_Clk,           -- Señal de reloj del bus PLB
   Dout_A1  => mem_data_out_A1,      -- Salida de datos (18 bts) de la memoria de antecedentes 1

   Addr_A2  => mem_address,          -- Señal de direccion (8 bits) para la memoria de antecedentes 2.
   Din_A2   => Bus2IP_Data (13 to 31), -- Dato de 18 bit proveniente del bus PLB
   We_A2    => memA2_write,          -- Control de L/E (incluye la seleccion de la memoria)
   Clk_A2   => Bus2IP_Clk,           -- Señal de reloj del bus PLB
   Dout_A2  => mem_data_out_A2,      -- Salida de datos (18 bts) de la memoria de antecedentes 2

   Addr_R   => mem_address (4 to 7), -- Señal de direccion (4 bits) para la memoria de reglas
   Din_R    => Bus2IP_Data (16 to 31), -- Dato de 16 bit proveniente del bus PLB
   We_R     => memR_write,           -- Control de L/E (incluye la seleccion de la memoria)
   Clk_R    => Bus2IP_Clk,           -- Señal de reloj del bus PLB
   Dout_R   => mem_data_out_R,       -- Salida de datos (16 bits) de la memoria de reglas
-- Conexion del FIM a los registros de E/S adicionados
   clk    => Bus2IP_Clk,             -- Señal de reloj del bus PLB
   reset  => Bus2IP_Reset,           -- Señal de RESET del bus PLB
   in1    => slv_reg0 (24 to 31),    -- Entrada 1 de datos (8 bits) del FIM proveniente del registro 0
   in2    => slv_reg1 (24 to 31),    -- Entrada 2 de datos (8 bits) del FIM proveniente del registro 1
   out1   => reg_data_out (24 to 31) ); -- Salida de datos (8 bits) del FIM (se conecta al registro 2)
```

Código 4

Fragmento de código VHDL con interconexiones realizadas en `user_logic.vhd`.

Debe tenerse presente que el procesador Microblaze y el bus PLB utilizan un formato de datos *big-endian*, en el cual el bit 0 es el más significativo. Las señales **mem_address** (de 8 bits), **mem**X**_write** y **mem_data_out_**X han sido previamente definidas y provienen de combinaciones de señales del bus PLB. Nótese también la escritura directa desde el bus de datos del PLB así como la conexión de las entradas y salidas del FIM a los tres registros adicionados mediante las señales **slv_reg**X (resaltadas en verde) y la señal **reg_data_out**.

Una vez convertido el FIM en un módulo IP, estará disponible para ser conectado al bus PLB de forma similar a la conexión de cualquier otro módulo IP disponible en EDK con interfaz para este bus.

Similar a la implementación sobre el SoC-FPGA, el programa que se ejecute en Microblaze será el encargado de escribir directamente en las memorias de antecedentes y de reglas la información correspondiente a la base de conocimientos inicial del controlador difuso, utilizando ahora la función `FIM_mWriteMemory` incluida en el *driver* generado (también se puede verificar directamente la escritura de estos valores mediante la función `FIM_mReadMemory`). Posteriormente, el programa podrá modificar directamente los contenidos de las memorias en cualquier momento y tantas veces como sea necesario.

5.- RESULTADOS DE IMPLEMENTACIONES

Los desarrollos expuestos fueron implementados y validados sobre dos tipos de dispositivos programables de Xilinx (un SoC-FPGA Zynq-7Z010, incluido en una placa de desarrollo Zybo Z7-10 y un FPGA Spartan-3E1600, incluido en una placa de desarrollo Spartan-3E1600 Development Kit), utilizando dos sistemas de procesamiento diferentes (el *hardcore* del sistema de procesamiento PS7, basado en procesadores ARM; y el sistema de procesamiento basado en el *softcore* del procesador Microblaze) y dos entornos de desarrollo diferentes (Vivado 2016.4 e ISE/EDK 14.7) con el objetivo de mostrar la versatilidad de la solución.

Aunque con Vivado pudo también realizarse la implementación de un sistema de procesamiento basado en Microblaze, este entorno de desarrollo solo permite trabajar con dispositivos programables de la serie-7 de Xilinx: FPGA de las familias Spartan-7, Artix-7, Kintex-7 y Virtex-7; así como SoC-FPGA y MPSoC-FPGA (*multiprocessor SoC-FPGA*) de la familia Zynq. Dada la amplia disponibilidad de FPGA (y placas de desarrollo) anteriores a la serie-7, que no pueden ser implementados mediante Vivado, se decidió incluir también el desarrollo utilizando ISE/EDK.

Para las comprobaciones experimentales de la modificación dinámica de la base de conocimientos del FIM se siguió el mismo procedimiento. Mediante Xfuzzy se describieron dos bases de conocimientos diferentes para el FIM de la Figura 2, modificando las funciones de pertenencia de las entradas y las salidas (también pudieron haberse modificado las reglas). Para ambas configuraciones se obtuvieron con *xfvhdl* los ficheros complementarios que facilitaron la codificación de los respectivos arreglos destinados a las memorias de antecedentes y de reglas en los programas de verificación desarrollados.

Como periféricos de los sistemas de procesamiento se utilizaron en ambos casos módulos IP *softcore* de un puerto de salida (para la conexión de LED de las respectivas placas como elementos de indicación visual) y de una interfaz serie UARTLite para la comunicación con una computadora personal. La Figura 10 muestra el diagrama de bloques de la implementación del controlador difuso híbrido HW/SW sobre el Zynq-7Z010 para la placa Zybo Z7-10, en donde se aprecia el *hardcore* del sistema de procesamiento PS7 así como los módulos IP del puerto de salida para los LED y de la interfaz de comunicación serie, resaltando el módulo IP del FIM con base de conocimientos variable, todos conectados a la interfaz AXI a través del módulo de interconexión (AXI Interconnect).

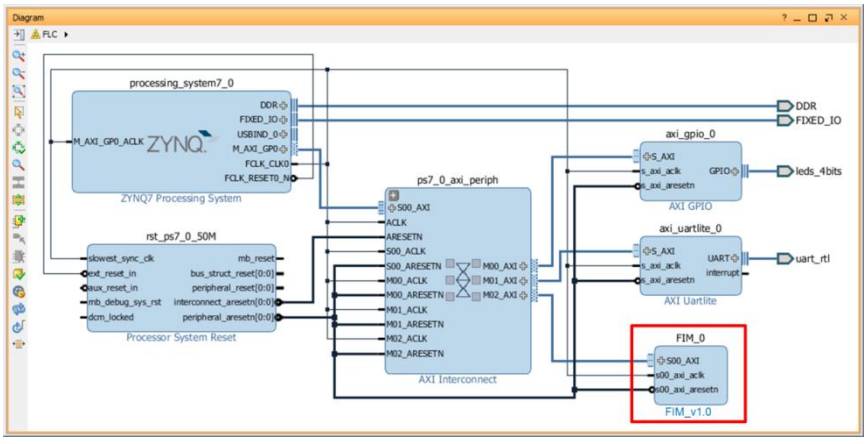

Figura 10

Implementación híbrida HW/SW de un controlador difuso utilizando el *hardcore* del sistema de procesamiento PS7 de un SoC-FPGA Zynq y el *softcore* del FIM modificado.

De forma equivalente, la Figura 11 muestra un fragmento del fichero de especificaciones de hardware (*Microprocessor Hardware Specifications*) `system.mhs`, quizá la forma más adecuada para describir el hardware de un sistema de procesamiento en EDK (su representación gráfica es pésima), en el que se aprecian las configuraciones e interconexiones de

los *softcore* del procesador Microblaze, el puerto para los LED, la interfaz serie UARTLite y el FIM modificado, todos conectados al bus PLB, resaltando las direcciones de las tres zonas de memoria (parámetros **C_MEMx_BASEADDR**) del FIM y de su zona de registros (parámetro **C_BASEADDR**) en correspondencia con la Figura 9b.

```
...
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER C_AREA_OPTIMIZED = 1
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER HW_VER = 8.50.c
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 BUS_INTERFACE DPLB = mb_plb
 BUS_INTERFACE IPLB = mb_plb
 BUS_INTERFACE DEBUG = microblaze_0_mdm_bus
 PORT MB_RESET = mb_reset
END
BEGIN xps_gpio
 PARAMETER INSTANCE = LEDs_8bit
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_GPIO_WIDTH = 8
 PARAMETER C_BASEADDR = 0x72000000
 PARAMETER C_HIGHADDR = 0x7200FFFF
 BUS_INTERFACE SPLB = mb_plb
END
...
```

```
...
BEGIN flc
  PARAMETER INSTANCE = FIM
  PARAMETER HW_VER = 1.00.a
  PARAMETER C_MEM0_BASEADDR = 0x10000000
  PARAMETER C_MEM0_HIGHADDR  = 0x1000FFFF
  PARAMETER C_MEM1_BASEADDR = 0x20000000
  PARAMETER C_MEM1_HIGHADDR  = 0x2000FFFF
  PARAMETER C_MEM2_BASEADDR = 0x30000000
  PARAMETER C_MEM2_HIGHADDR  = 0x3000FFFF
  PARAMETER C_BASEADDR = 0x40000000
  PARAMETER C_HIGHADDR  = 0x4000FFFF
  BUS_INTERFACE SPLB = mb_plb
END
BEGIN xps_uartlite
 PARAMETER INSTANCE = RS232
 PARAMETER HW_VER = 1.02.a
 PARAMETER C_USE_PARITY = 0
 PARAMETER C_BASEADDR = 0x71000000
 PARAMETER C_HIGHADDR = 0x7100FFFF
 BUS_INTERFACE SPLB = mb_plb
END
...
```

Figura 11

Fragmento del fichero `system.mhs` con las interconexiones de los módulos IP del procesador Microblaze, GPIO, UARTLite y el FIM modificado.

La Tabla 1 muestra un resumen del consumo de recursos de los módulos IP desarrollados para los FIM originales (solo tres registros para las entradas y la salida) y modificados, tanto con la opción del servicio de mapeo de memorias (sobre un FPGA Spartan-3E1600) como con la adición de otros seis registros (sobre un SoC-FPGA Zynq-7Z010) para acceder al puerto incorporado en cada memoria de la base de conocimientos. Nótese el incremento de 56 *Flip-Flop* (FF) y de 111 tablas de búsqueda (LUT) de cuatro entradas en el FIM modificado sobre el FPGA Spartan-3E1600 debido a la adición de los circuitos de selección y control de las memorias, mientras que los *Flip-Flop* se incrementan en 117 para el FIM modificado sobre el SoC-FPGA Zynq-7Z010 debido a los seis registros adicionados para el acceso indirecto a las memorias. La disminución en el consumo de las tablas de búsqueda en las implementaciones sobre el SoC-FPGA Zynq-7Z010 se debe a que son de seis entradas (equivalentes a cuatro tablas de búsqueda de un Spartan-3E). Todas las implementaciones consumen dos bloques de memora RAM y un bloque de multiplicación.

Tabla 1

Comparación del consumo de recursos de los FIM originales y los modificados.

Recursos	FPGA Spartan-3E1600			SoC- FPGA Zynq-7Z010		
	FIM original (3 registros)	**FIM modificado** (3 reg + memoria)	**Dif**	**FIM original** (3 registros)	**FIM modificado** (9 registros)	**Dif**
FF	183	239	+ 56	169	286	+ 117
LUT	238	349	+ 111	57	65	+ 8
BRAM	2	2	0	2	2	0
Mult	1	1	0	1	1	0

Dadas las enormes diferencias existentes entre los recursos hardware de un SoC-FPGA Zynq y un FPGA Spartan-3E, sobre todo debido a que el Zynq incluye la implementación hardware de todo el sistema de procesamiento PS7 (dos procesadores ARM con sus respectivas cache L1 y coprocesador NEON; memoria cache L2; memoria SRAM; controladores de memoria dinámica y de memoria Flash, etc.), así como las diferencias ya mencionadas en las dimensiones de las tablas de búsqueda de ambos dispositivos, no tiene sentido realizar comparaciones entre los consumos de recursos de las implementaciones de los controladores difusos híbridos HW/SW sobre ambas plataformas.

Los programas de verificación desarrollados para ambos sistemas de procesamiento utilizando las respectivas herramientas SDK (*Software Development Kit*) cargaron y verificaron la escritura de la primera base de conocimientos. Seguidamente, realizando ciclos de barrido anidados de los valores de cada una de las entradas que fueron suministrados al módulo IP del FIM, se obtuvieron los valores de las salidas correspondientes. En cada iteración fueron transmitidos los valores de las entradas y de la salida del FIM por la interfaz de comunicación serie hacia la computadora personal, con el objetivo de almacenarlas y obtener posteriormente el gráfico de la superficie de control. A continuación se repitió el mismo procedimiento cargando la segunda base de conocimientos en el FIM. La única diferencia entre los programas desarrollados para el PS7 y para Microblaze consistió en la forma de escribir y leer en las memorias de la base de conocimientos: de forma indirecta a través de los registros en el caso del PS7; o escribiendo y leyendo directamente de las memorias en el caso de Microblaze. En todos los casos se comprobó la correcta modificación dinámica de la base de conocimientos del FIM mediante la correspondencia de las superficies de control obtenidas de forma experimental con las obtenidas mediante Xfuzzy.

6.- CONCLUSIONES

Las implementaciones hardware de los módulos de inferencia difusos reportadas se caracterizan por el carácter estático de su base de conocimientos, no permitiendo su modificación de forma dinámica durante su operación. La gran mayoría se caracteriza por presentar estructuras muy diversas que impiden la implementación de una estrategia que facilite la modificación dinámica de la base de conocimientos.

Aunque es posible realizar la modificación de la base de conocimientos (y de todo el FIM en general) mediante la reconfiguración dinámica parcial disponible en los FPGA y SoC-FPGA actuales, esta estrategia requiere de la obtención previa de los ficheros de reconfiguración de las diferentes bases de conocimientos, lo cual limita su alcance.

Las herramientas de implementación hardware disponibles en el entorno de desarrollo de sistemas difusos Xfuzzy, basadas en una arquitectura uniforme, configurable y parametrizable, que puede ser utilizada en muy diversas aplicaciones, facilita la estrategia de realización seguida en este trabajo basada en convertir a memorias doble puerto las memorias de antecedentes y de reglas del FIM. De esta forma, mediante un sistema de procesamiento empotrado en el mismo dispositivo programable (típico de las realizaciones de controladores difusos híbridos HW/SW), se puede acceder a través de los puertos añadidos a los contenidos de estas memorias y modificarlos durante la operación del FIM.

Aunque la estrategia ha sido expuesta utilizando la herramienta de síntesis hardware *xfhvdl* de Xfuzzy por ser la más general e independiente del fabricante del dispositivo de hardware reconfigurable, también puede ser implementada con la herramienta *xfsg*, con la limitante de solo poder ser utilizada con dispositivos de Xilinx.

Aunque se ha expuesto una forma de realizar las modificaciones en el fichero de nivel superior del FIM generado por *xfvhdl* para la conversión a doble puerto de las memorias de antecedentes y de reglas, existen otras variantes para hacerlo.

La disponibilidad de herramientas similares en los diferentes entornos de desarrollo de dispositivos programables facilitan la conversión del FIM modificado en un módulo IP, siendo la opción más general la de adicionar registros de entrada/salida para el acceso indirecto desde el sistema de procesamiento a las memorias de antecedentes y de reglas.

Se han desarrollado y validado dos variantes de conversión del FIM modificado en módulos IP de usuario utilizando diferentes dispositivos (un FPGA Spartan-3E1600 y un SoC-FPGA Zynq-7Z010); dos entornos de desarrollo (ISE/EDK y Vivado); dos tipos de servicios para el acceso a las memorias (directamente mediante un controlador de memoria o indirectamente mediante puertos de entrada/salida); y dos sistemas de procesamiento (basados en el *softcore* de Microblaze con bus PLB y el *hardcore* del PS7 basado en procesadores ARM con interfaz AXI), lo cual evidencia la generalidad de la solución propuesta.

La solución expuesta para la modificación dinámica de la base de conocimientos del FIM durante su operación permite la implementación de controladores difusos híbridos HW/SW más flexibles, facilita el ajuste de los parámetros del FIM durante su etapa de desarrollo sin tener que recurrir a múltiples re-implementaciones y, considerando las potencialidades de los sistemas de procesamiento con capacidad para ejecutar algoritmos de aprendizaje, sienta las bases para la implementación de controladores difusos adaptativos.

AGRADECIMIENTOS

Esta investigación ha sido financiada parcialmente por el Consejo Superior de Investigaciones Científicas (CSIC) de España a través del proyecto COOPB 20420 correspondiente al Programa de Cooperación Científica para el Desarrollo i-COOP+.

REFERENCIAS

1. Jantzen J. Foundations of Fuzzy Control: a practical approach. 2nd. ed. Wiley; 2013.
2. Allani MY, Mezghani D, Tadeo F, Mami A. FPGA Implementation of a Robust MPPT of a Photovoltaic System Using a Fuzzy Logic Controller Based on Incremental and Conductance Algorithm. Engineering, Technology & Applied Science Research. 2019;9(4):4322–8.
3. Youssef A, El Telbany M, Zekry A. Reconfigurable generic FPGA implementation of fuzzy logic controller for MPPT of PV systems. Renewable and Sustainable Energy Reviews [Internet]. 2018;82:1313–9. Available from: http://dx.doi.org/10.1016/j.rser.2017.09.093
4. Boukadida S, Gdaim S, Mtibaa A. Hardware Implementation of a Neuro Fuzzy Based DTC-SVM of an Induction Motor on the FPGA. WSEAS Transactions on Power Systems. 2018;13:60–8.
5. Lotfy A, Kaveh M, Mosavi M, Rahmati A. An enhanced fuzzy controller based on improved genetic algorithm for speed control of DC motors. Analog Integrated Circuits and Signal Processing [Internet]. 2020;9. Available from: https://doi.org/10.1007/s10470-020-01599-9
6. Abdelkrim H, Othman S Ben, Saoud S Ben. FPGA Implementation of Self-Reconfigurable Fuzzy Logic Controller. In: 2018 International Conference on Advanced Systems and Electric Technologies (IC_ASET). Hammamet, Tunisia: IEEE; 2018. p. 151–6.
7. Mishra A, Dubey G, Joshi D, Agarwal P, Sriavstava SP. A Complete Fuzzy Logic Based Real-Time Simulation of Vector Controlled PMSM Drive. In: 2nd IEEE International Conference on Power Electronics, Intelligent Control and Energy Systems. Delhi, India: IEEE; 2018. p. 809–14.
8. Shah VS, Shah SA. Adaptive FPGA Based Three Phase Controller Inverter. In: 2018 International Conference on Research in Intelligent and Computing in Engineering (RICE). San Salvador, El Salvador: IEEE; 2018. p. 1–8.
9. Gdaim S, Mtibaa A, Mimouni MF. Design and Experimental Implementation of DTC of Induction Machine based on Fuzzy Logic Control on FPGA. IEEE Transactions on Fuzzy Systems. 2015;23(3):1–12.
10. Bishwokarma R, Khatiwoda S, Bhetwal B, Kumar S. FPGA based Fuzzy Logic Controller for Frequency Regulation of Synchronous Generator. In: 2020 International Conference on Electrical and Electronics Engineering (ICE3). Gorakhpur, India: IEEE; 2020. p. 699–704.
11. Moussa I, Khedher A. Fuzzy Logic Controller Hardware Implementation using XSG tools Applied to a Variable Speed Wind Turbine Emulator. In: 2019 International Conference on Control, Automation and Diagnosis (ICCAD). Grenoble, France: IEEE; 2019. p. 1–6.
12. Krishna VSS, Misra Y, Rao GA. FPGA Implementation of Variable Feed Rate Algorithm for a Three Input Fuzzy Controller to Maintain the Cane Level. International Journal of Innovative Technology and Exploring Engineering (IJITEE). 2019;8(10):3900–15.
13. Khati H, Mellah R, Talem H. Neuro-fuzzy Control of a Position-Teleoperation System Using FPGA. In: 2019 24th International Conference on Methods and Models in Automation and Robotics. Miedzyzdroje, Poland: IEEE; 2019. p. 64–9.
14. Boncalo O, Amaricai A, Lendek Z. Configurable Hardware Accelerator Architecture for a Takagi-Sugeno Fuzzy Controller. In: 2019 22nd Euromicro Conference on Digital System Design (DSD). Kallithea, Greece: IEEE; 2019. p. 96–101.
15. Mohammadi M, Shaout A. Reconfigurable Implementation of Fuzzy Inference System Using FPGA. In: 2017 International Conference on New Trends in Computing Sciences. Amman, Jordan: IEEE; 2017. p. 18–23.
16. Jokar E, Klidbary SH, Abolfathi H, Shouraki SB, Zand R, Ahmadi A. Hardware-Algorithm Co-Design of a Compressed Fuzzy Active Learning Method. IEEE Transactions on Circuits and Systems. 2020;67(12):1–14.
17. Economakos C, Kiokes G, Economakos G. Using Advanced FPGA SoC Technologies for the Design of Industrial Control Applications. In: 2015 6th International Conference on Information, Intelligence, Systems and Applications. Corfu, Greece: IEEE; 2015. p. 1–6.
18. Bicakci S. On the Implementation of Fuzzy VMC for an Under Actuated System. IEEE Access. 2019;7:163578–88.
19. Rajarshi P, Shreesha C. Design and Implementation of Fuzzy Logic Controller on MPSoC FPGA for Shell and Tube Heat Exchanger. In: Shreesha C, Ravindra D. Gudi, editors. Control Instrumentation Systems, Lecture Notes in Electrical Engineering, vol 581. Springer; 2018. p. 13–31.

20. Viajante GP, Chaves EN, Miranda LC, Freitas MAA, Queiroz CA, Santos JA. Design and Implementation of a Fuzzy Control System Applied to a 6x4 SRG. In: IEEE International Conference on Environment and Electrical Engineering. Genova, Italy: IEEE; 2019. p. 1–6.

21. Loukil K, Abbes H, Abid H, Abid M, Toumi A. Design and implementation of reconfigurable MPPT fuzzy controller for photovoltaic systems. Ain Shams Engineering Journal [Internet]. 2020;11(2):319–28. Available from: https://doi.org/10.1016/j.asej.2019.10.002

22. Al-Gizi A, Al-Rawe B, Al-Saadi M, Craciunescu A. Step by Step FPGA-Based Implementation of MPPT Fuzzy Controller for PV Systems. In: 2019 11th International Symposium on Advanced Topics in Electrical Engineering (ATEE). Bucharest, Romania: IEEE; 2019. p. 1–6.

23. Shehu Y, Irshaidat M, Soufian M. A FPGA Implementation of a Dual-Axis Solar Tracking System. In: 12th International Conference on Developments in eSystems Engineering (DeSE). Kazan, Russia: IEEE; 2019. p. 970–4.

24. Banjanovic L, Husejnovic A. FPGA based Hexapod Robot Navigation using Arbitration of Fuzzy Logic Controlled Behaviors. In: 2019 XXVII International Conference on Information, Communication and Automation Technologies (ICAT). Sarajevo, Bosnia and Herzegovina: IEEE; 2019. p. 1–6.

25. Chen C, Changyuan C, Han X. Design of Equivalent Single-Input Fuzzy PI Converter for Buck DC-DC Converters with Excellent Transient Performance. In: 2019 IEEE 3rd International Electrical and Energy Conference (CIEEC). Beijing, China: IEEE; 2019. p. 335–40.

26. Yang S, Deng B, Wang J, Liu C, Li H, Lin Q, et al. Design of Hidden-property-based Variable Universe Fuzzy Control for Movement Disorders and Its Efficient Reconfigurable Implementation. IEEE Transactions on Fuzzy Systems. 2019;27(2):304–18.

27. Ahmed HO. 10 . 52 GOPS Systolic Cores Fuzzy Logic System for Cognitive Dysfunction Self- Awareness U sing FPGA. In: 2020 3rd International Conference on Intelligent Robotic and Control Engineering (IRCE). Oxford, UK: IEEE; 2020. p. 11–7.

28. Xfuzzy: Fuzzy Logic Design Tools [Internet]. Available from: https://www2.imse-cnm.csic.es/Xfuzzy/

29. Brox M, Sánchez-Solano S, Toro E, Brox P, Moreno-Velo FJ. CAD Tools for Hardware Implementation of Embedded Fuzzy Systems on FPGAs. IEEE Transactions on Industrial Informatics. 2013;9(3):1635–44.

30. Cabrera A, Sanchez S, Jiménez C, Barriga Á, Baturone I. Arquitectura Eficiente para la Implementación Hardware de Sistemas de Inferencia Difusos. Revista de Ingeniería Electrónica, Automática y Comunicaciones. 2003;23(1):59–66.

31. Sánchez-Solano S, Cabrera AJ, Baturone I, Moreno-Velo FJ, Brox M. FPGA Implementation of Embedded Fuzzy Controllers for Robotic Applications. IEEE Transactions on Industrial Electronics. 2007;54(4):1937–45.

32. Boumazbar S, Bouall S, Hagg J. Co-simulation and Rapid Prototyping of Fuzzy Supervised PID controllers based on FPGA-Nexys2 Board. In: 7th International Conference on Modelling, Identification and Control (ICMIC 2015). Sousse, Tunisia: IEEE; 2015. p. 3–8.

33. Gersnoviez A, Brox M, Baturone I. Hierarchical Fuzzy Controllers for Explicit MPC Control Laws : Adaptive Cruise Control Example. In: 2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE). Glasgow, UK: IEEE; 2020. p. 1–7.

34. Reza S, Mahbub TN, Islam M, Arifeen M, Remu RH, Hossain DA. Assuring Cyber Security in Smart Grid Networks by Fuzzy-logic based Trust Management Model. In: IEEE International Conference on Robotics, Automation, Artificial-intelligence and Internet-of-Things. Dhaka, Bangladesh: IEEE; 2019. p. 5–8.

CONFLICTO DE INTERESES

Ninguno de los autores ha manifestado la existencia de posibles conflictos de intereses que debieran ser declarados en relación con este artículo.

CONTRIBUCIONES DE LOS AUTORES

Alejandro José Cabrera Sarmiento: conceptualización, investigación, metodología, curación de datos, software, validación – verificación, supervisión, redacción – revisión y edición.

Santiago Sánchez Solano: conceptualización, investigación, supervisión, redacción – revisión y edición.

Yasmani García Guirola: investigación, software, validación – verificación, revisión.

AUTORES

Alejandro José Cabrera Sarmiento: Ingeniero Electricista, Doctor en Ciencias Técnicas, Profesor Titular del Dpto. de Automática y Computación de la Universidad Tecnológica de La Habana (CUJAE), Cuba. Sus principales intereses de

investigación incluyen el desarrollo de sistemas empotrados sobre hardware reconfigurable para aplicaciones de control, de procesamiento de imágenes y de IoT. Email: alex@automatica.cujae.edu.cu. https://orcid.org/0000-0003-4129-911X

Santiago Sánchez Solano: Licenciado en Física, Doctor en Física, Investigador Científico del Instituto de Microelectrónica de Sevilla (CSIC/Universidad de Sevilla), España. Sus principales intereses de investigación incluyen la realización microelectrónica de sistemas neurodifusos, así como sus aplicaciones en robótica, procesamiento de imágenes, seguridad y redes de sensores inteligentes. Email: santiago@imse-cnm.csic.es. https://orcid.org/0000-0002-0700-0447

Yasmani García Guirola: Ingeniero en Automática por la Universidad Tecnológica de La Habana (CUJAE), Cuba, Especialista en Automática del Ministerio de la Agricultura. Sus principales intereses de investigación incluyen el desarrollo de sistemas de control y sistemas digitales sobre hardware reconfigurable. Email: yasmanigarciaguirola@gmail.com. https://orcid.org/0000-0002-4322-2645