elektron ámbar

Original Article

Revista elektron, Vol. 6, No. 1, pp. 20-28 (2022)
https://doi.org/10.37537/rev.elektron.6.1.148.2022

Implementación eficiente de controladores difusos en FPGA basados en síntesis de alto nivel

Efficient Fuzzy Controllers for FPGA using High Level Synthesis

Luca Sarramone [#1], Martín Vázquez [*2], Lucas Leiva [*3]

[#] *Departamento de computación y sistemas, UNICEN*
Tandil, Bs. As., Argentina

[1] lsarramone@intia.exa.unicen.edu.ar

[*] *Laboratorio Sistemas Embebidos, INTIA, UNICEN*
Tandil, Bs. As., Argentina

Facultad de Ingeniería - Universidad Nacional de Tres de Febrero
Caseros, Buenos Aires, Argentina

[2] mvazquez@labset.exa.unicen edu.ar

[3] lleiva@labset.exa.unicen edu.ar

Recibido: 01/03/22; Aceptado: 04/05/22

***Resumen*— Los sistemas de control basado en lógica difusa (FLC, *Fuzzy Logic Controller*) poseen ventajas, ya que no requieren modelado matemático y además son útiles cuando se necesita del conocimiento de un experto para el manejo de datos imprecisos. Resulta interesante para la implementación de un FLCs la utilización de la tecnología FPGA. Esta tecnología presenta ventajas respecto a la velocidad de procesamiento, consumo de potencia, flexibilidad de diseño y reconfiguración. Este trabajo presenta una herramienta basada en HLS para generar FLCs sobre FPGA. Durante su desarrollo también se analizaron las directivas de síntesis con mayor impacto sobre la performance de los algoritmos. Además se desarrollaron tres problemas de lógica difusa para verificar el funcionamiento de la herramienta.**

Palabras clave: FLC; FPGA; HLS.

***Abstract*— Fuzzy Logic Controllers (FLC) are control systems commonly used on problems where data is not accurate or its domain is not well-known. This is because instead of using complex mathematical models to work, they use a set of rules to evaluate data. To implement this kind of controllers one interesting option is FPGA. This technology has advantages based on reconfigurability, performance, energy usage and design flexibility. This work presents a tool based on HLS and FPGA that allows users to generate Fuzzy Logic Controllers from abstract descriptions. Also the most impactful synthesis directives for optimizing the different stages of a FLC are detailed. Finally, three case studies are presented to evaluate the tool.**

Keywords: FLC; FPGA; HLS.

I. Introduction

La lógica difusa (FL, *Fuzzy Logic*) es una lógica alternativa a la lógica clásica que pretende introducir un grado de vaguedad a las funciones que evalúa. Su objetivo es incluir en el resultado el grado de incertidumbre que la mayoría de los fenómenos presentan en la descripción de su naturaleza [1,2]. Las funciones de pertenencia dentro de la teoría de conjuntos difusos pueden adquirir valores en el rango 0 a 1, a diferencia de la teoría de conjuntos clásicos donde únicamente se devuelven valores binarios.

Un sistema de inferencia difusa [3] es un controlador basado en lógica difusa (FLC). A diferencia de los sistemas de control clásicos, los FLC no requieren de modelos de procesos físicos ya sean analíticos como experimentales. Los FLC son especialmente adecuados para procesos complejos y mal definidos, para los cuales es complejo conseguir un modelo analítico que los describa. Los FLCs consisten básicamente en cuatro unidades: *i*) *fusificador*, encargado de combinar valores actuales, o *crisp*, con datos, mediante funciones de pertenencia para producir valores de entradas difusos; *ii) base de reglas difusas*, almacena las reglas difusas que describen el funcionamiento del controlador difuso; *iii) motor de inferencia difusa*, realiza el cálculo mediante la asociación de variables de entrada con reglas difusas; *iv) defusificador*, cuya funcionalidad es obtener un valor numérico representativo de todas las salidas (Fig. 1).

Para la implementación de sistemas de control en general, en los últimos años tuvo gran aceptación el uso de microcontroladores, DSPs (*Digital Signal Processors*) y FPGAs [4]. Los FPGAs se han usado de manera exitosa como soluciones de diseño de hardware en control industrial. El diseño en FPGA posee mucha flexibilidad y potencia, en donde en este último tiempo, la tendencia es el co-diseño de plataformas Hardware/Software basado en descripciones y síntesis de alto nivel [5,6]. Los avances sobre esta tecnología posibilitan una gran potencia de cálculo con un uso eficiente de energía. La utilización de FPGAs para la implementación de FLCs es una alternativa auspiciosa, no solo en lo que respecta a la velocidad de procesamiento, sino principalmente a la flexibilidad de diseño y reconfigurabilidad. Esta capacidad de reconfiguración permite a los usuarios evaluar diferentes diseños directamente sobre el dispositivo, y realizar ajustes de manera ágil hasta que el funcionamiento sea el esperado.

De esta manera, las imprecisiones propias de los problemas de lógica difusa pueden ser resueltas de manera sencilla mediante prueba y error.

En este contexto surge la síntesis de alto nivel (HLS), que pretende generar implementaciones RTL a partir de lenguajes de alto nivel tipo C, lo que permite a los programadores concentrarse en el comportamiento de los algoritmos, evitar errores y disminuir el tiempo de implementación [7, 8]. Esta nueva tendencia de desarrollo queda demostrada en las diferentes herramientas que hoy día se encuentran en el mercado, tales como *HLS Compiler* de Intel, *Synopsys Synphony C Compiler* de Microsemi o *Vivado HLS* de Xilinx.

Este trabajo realiza contribuciones en las áreas de desarrollo de controladores, tales como robótica y aceleración de procesos, donde la lógica difusa es utilizada con frecuencia. El objetivo es proveer una herramienta de software libre que permita a los desarrolladores agilizar los procesos de diseño, implementación y, especialmente, calibración de controladores difusos. Para ello se explotan las capacidades de reconfiguración de la tecnología FPGA y la facilidad de desarrollo que provee la síntesis en alto nivel.

Según el conocimiento de los autores no existen desarrollos de FLCs mediante el uso de HLS. Algunos trabajos presentan herramientas para desarrollar automáticamente implementaciones en HDL de FLC [9-12], pero ninguno analiza y explora la utilización eficiente de los recursos disponibles en los dispositivos FPGAs actuales. En [9] se presentan resultados sobre dispositivos FPGAs de Xilinx obsoletos. En [10] no se muestran detalles de implementación, solo se presenta el modelo VHDL generado. En lo que respecta a [11 y 12], si bien las herramientas generan FLCs modelados en VHDL que podrían sintetizarse en FPGA, las implementaciones se efectuaron en ASICs.

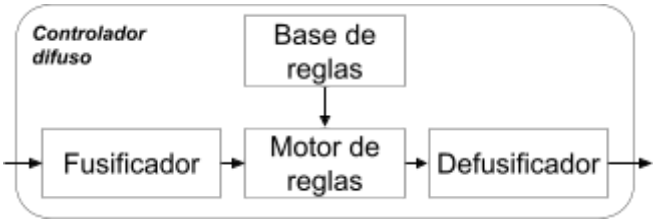

Fig 1. Diagrama de componentes de un FLC.

II. Síntesis de alto nivel de un FLC

El propósito de este trabajo es hallar la mejor combinación de directivas y optimizaciones que permitan generar FLCs eficientes, mediante el uso de Vivado HLS. Las directivas son instrucciones que se aplican sobre un fragmento de código específico, indicando el tipo de optimización que se debe utilizar.

Para alcanzar esta meta, lo primero que se tuvo en cuenta son las tres etapas: fusificación, evaluación de reglas y defusificación. Cada una de ellas se implementó como métodos separados, lo que posibilita aplicar mejoras específicas en lugar de mejoras generales que pierden eficacia al abarcar todo el código. Habiendo establecido esto, se realizaron pruebas sobre todas las etapas de forma individual para determinar las directivas óptimas

A. Fusificación

Respecto al fusificador surgen dos estrategias de implementación: la orientada a memoria o la orientada a cálculo. La primera precalcula los resultados de cada posible entrada y los almacena en memoria, mientras que la segunda los calcula en tiempo de ejecución. Si bien ambas opciones son válidas y la elección de alguna de ellas depende del contexto del problema, implementar ambos enfoques resulta costoso. Por esta razón, y siendo que el fusificador orientado a memoria tiene una complejidad especial alta para el tipo de problemas que se busca atacar inicialmente, se optó por desarrollar controladores con fusificadores orientados a cálculo.

Dado que existen una gran variedad de algoritmos disponibles para la fusificación, el primer paso fue tomar un número reducido de casos para el análisis. Se seleccionaron los tipos de funciones más comunes: Triangular, Trapezoidal, *Singleton*, Forma S y Forma Z. Cada una de ellas fue reescrita de forma conveniente, ya que las fórmulas usadas en la literatura para calcular sus valores utilizan divisiones, lo que resulta costoso. Las fórmulas usadas en este trabajo tienen la forma:

$$f(x) = P_1 * (x - a) \quad (1)$$

$$f(x) = P_2 * (a - x) \quad (2)$$

donde P_1 y P_2 son las pendientes constantes de las rectas que forman a cada función.

A continuación se realizaron diferentes pruebas, donde se determinó que *Pipeline* es la directiva con mayor impacto, reduciendo la latencia y el intervalo entre respuestas sucesivas al mínimo posible. El problema reside en que el fusificador se compone de varias instancias de funciones fusificadoras, por lo que se generó un método extra donde se invocan de forma ordenada. Esto implica que cualquier *pragma* aplicado sobre el método general no se traslada a las invocaciones. Para solucionarlo, se decidió que cada llamado a función sea reemplazado por el cuerpo de la función invocada. De esta manera, la concurrencia se aplica sobre todas las instrucciones, indistintamente del método al que pertenezcan.

Por último, como las entradas deben ser convertidas previamente a valores digitales por conversores A/D, usar variables de tipo double o float para almacenar los valores fusificados es ineficiente. En su lugar, se utilizó un tipo de variable nuevo (llamado *fixed_int*) de tipo entero, y cuya cantidad de bits depende directamente de la cantidad de bits de los conversores usados. Esto significa que si se tiene un conversor de *n* bits, las variables utilizadas en la fusificación tendrán *n* bits de ancho.

B. Evaluación de reglas

La evaluación de reglas toma el resultado de la etapa anterior y calcula los valores de pertenencia para cada variables de salida, usando una base de reglas. Existen gran cantidad de algoritmos disponibles para esta etapa pero, al igual que con el caso anterior, se decidió analizar el más usado. El mismo es conocido como método Mamdani o MinMax, por la forma en la que calcula los resultados.

Nuevamente surgen dos estrategias de implementación. Una de ellas se basa en el uso de matrices, donde para cada

combinación de entradas se establece una única salida. La otra utiliza secuencias de *if-then*, donde la condición indica la combinación de entradas y la consecuencia indica los valores de salida. Si bien la primera opción es más eficiente, el uso de matrices obliga a que las reglas de la base aparezcan una única vez obligatoriamente. Esto resulta problemático para ciertos problemas donde existen reglas repetidas o ignoradas. Dado que el objetivo de este proyecto es conseguir un generador de controladores, la flexibilidad del lenguaje es un atributo importante y por ello se optó por utilizar la segunda estrategia.

El método de MinMax se basa en agrupar las reglas según su consecuente, tomar el valor de pertenencia mínimo de los antecedentes de cada regla y, del subconjunto resultante de esta operación, buscar los máximos para cada subgrupo consecuente (Fig. 2). Dado que las comparaciones de mínimo o máximo deben realizarse de a pares, las reglas que contengan el mismo consecuente deben ejecutarse de forma serial, ya que para obtener el máximo es necesario encontrar todos los valores anteriores. Bajo estas condiciones, se estableció que el proceso de evaluación de reglas consista en una variable auxiliar (donde se almacena el mínimo de cada regla) que se contrasta con el mayor valor encontrado hasta ese punto. Por su parte, los valores máximos se almacenan aparte, de manera tal que al terminar el procedimiento se encuentren todos dentro de la misma estructura.

Finalmente, con respecto a las optimizaciones, se aplicó *Pipeline* sobre el método principal para que las instrucciones pudieran ejecutarse de forma concurrente, disminuyendo la latencia y el intervalo entre resultados sucesivos. Para esta etapa se utilizan nuevamente variables de tipo *fixed_int*.

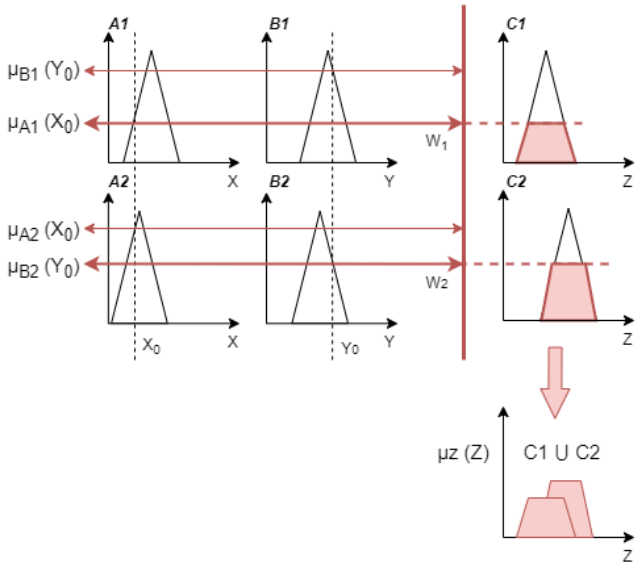

Fig 2. Funcionamiento del algoritmo Mandami.

C. Defusificación

La defusificación se encarga de reconvertir los valores *Fuzzy* a valores *Crisp*. De entre los algoritmos disponibles para esta etapa, se decidió implementar el conocido como Centroide, no solo por su popularidad, sino también por ser el más eficiente. Su funcionamiento consiste en una suma de resultados parciales, donde en el numerador se incluyen los productos entre el grado de pertenencia $(\mu_B(y_i))$ y el valor de salida correspondiente (y_i) y en el denominador únicamente los grados de pertenencia, como se muestra en la siguiente ecuación:

$$\frac{\sum_{i=1}^{n} y_i * \mu_B(y_i)}{\sum_{i=1}^{n} \mu_B(y_i)} = Y \qquad (3)$$

A partir de su análisis, se encontró que nuevamente la mejor directiva es *Pipeline*. Otros *pragmas*, tales como *Loop Unroll*, no consiguen mayor efecto ya que la sumatoria debe ejecutarse obligadamente de forma secuencial y como consecuencia solo se pueden paralelizar instrucciones muy específicas, lo que se consigue usando *Pipeline*.

Por otra parte, dado que el numerador y denominador se calculan mediante sumatoria de valores *fixed_int*, se debe hacer uso de un nuevo tipo de variable. El mismo fue llamado *fixed_aux*, y la cantidad de bits que ocupa está determinada tanto por la cantidad de variables de entrada, como por el tamaño de los conversores Este nuevo tipo tampoco cuenta con parte decimal.

D. Comunicación y ensamblado

Finalmente, cada uno de estos métodos individuales deben combinarse en un mismo programa para generar el controlador. Con este fin, se creó una nueva función *top-level* que invoca a cada etapa del FLC en orden. Además, se generaron las constantes usadas para la fusificación, y los tipos de variables especiales.

El problema ahora es establecer una comunicación entre las distintas etapas. La solución implementada utiliza arreglos intermedios que funcionan como buffers para los resultados de la etapa anterior. De esta manera, se puede segmentar el controlador en tres partes. Esto, sumado a la directiva *Dataflow* que se aplica sobre la función *top-level*, permite reducir el intervalo entre lecturas sucesivas del controlador a únicamente un ciclo. Además, se utilizó *Array Partition* sobre los arreglos para que los datos contenidos puedan ser accedidos de forma independiente al resto. De esta manera, se explota al máximo la concurrencia y se reduce aún más la latencia general del algoritmo.

Por último, dado que un problema de lógica difusa puede contar con más de una variable de salida, se implementó un nuevo tipo (llamado *fixed_out*) donde se concatenan los diferentes resultados en una misma variable. Por ejemplo, si se tiene un controlador con conversores de 8 bits y 3 variables de salida (A, B y C), *fixed_out* tendrá un ancho de 24 bits que se distribuye de la siguiente manera: bits [0;7] = Variable A, bits [8;15] = Variable B, bits [16;23] = Variable C.

E. Configuración general

En resumen, la mejor configuración de directivas y mejoras para optimizar un controlador difuso es la siguiente:

- Tres métodos separados (uno para cada etapa del controlador) con la directiva *Pipeline* aplicada.
- Una función *top-level* que invoque a cada etapa y con la directiva *Dataflow* aplicada.

- Arreglos intermedios que comunican cada etapa, particionados usando la directiva *Array Partition*.
- Tres tipos de variables predefinidas: *fixed_int*, *fixed_aux* y *fixed_out*.

F. Caso de uso: péndulo invertido

Para comprobar la eficiencia de las mejoras aplicadas, se tomó como caso de uso al problema del péndulo invertido [13]. El mismo es un sistema clásico, frecuentemente utilizado en este tipo de pruebas por ser un problema relativamente sencillo, cuyas soluciones pueden ser aplicadas al control de mecanismos más complejos, como transporte aéreo o robots bipedos. Consiste en una varilla (de masa *m* y longitud *L*) unida por un extremo al centro de un carro que se mueve horizontalmente. El prototipo está equipado con sensores para medir la velocidad, aceleración y ángulo a la que cae la varilla (Fig. 3). En base a estas variables se calcula la fuerza que debe ejercer el movimiento del carro para poder mantener la varilla en equilibrio.

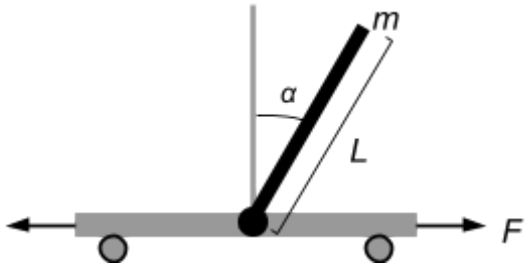

Fig 3. Esquema del problema de péndulo invertido.

Se generaron dos versiones del mismo problema. La primera de ellas no tiene ninguna mejora aplicada, mientras que la segunda utiliza todas las optimizaciones planteadas anteriormente. Con el objetivo de verificar que los resultados de ambos casos fueran correctos, las dos versiones fueron simuladas en alto nivel usando Vivado HLS versión 2020.1. Habiendo comprobado su funcionamiento, se sintetizaron sobre la misma plataforma usando la placa xc7z020clg400-1 [14, 15].

Los resultados presentados (Tabla I y II) demuestran que las optimizaciones permiten obtener una mejora de 10x en la latencia, respecto a la versión sin optimizaciones. Al mismo tiempo, el intervalo de inicio se reduce a 1, admitiendo una nueva entrada en cada ciclo de reloj. Con respecto al uso de recursos se puede observar que a pesar de emplear directivas, la versión con mejoras posee una menor ocupación de recursos. Específicamente, para la plataforma usada en la síntesis, la última versión ocupa un 35%, 16% y 50% menos en relación al total de DSP, FF y LUT respectivamente. El código resultante en VHDL también se sintetizó usando el software de Vivado 2018 y los valores obtenidos son equivalentes .

Por otra parte, debido a que el tiempo de respuesta obtenido es de 0.280 µs (con un período de *clk* de 10 ns) y considerando que los conversores A/D de 12 bits que se encuentran en los dispositivos modernos de Xilinx (XDAC) [16] operan a 1 MSPS (*millon samples per second*), se puede asegurar el correcto funcionamiento del FLC entre lecturas sucesivas de las entradas.

Finalmente, utilizar variables con dimensiones arbitrarias obliga a que los valores se trunquen, y como consecuencia las salidas calculadas por el controlador no serán iguales a las obtenidas en simulación con tipos de datos con representación en punto flotante. No obstante, el mayor grado de imprecisión se añade al convertir las entradas a señales digitales, por lo que la única solución posible desde software es ajustar los valores empíricamente para la configuración y calibración del sistema.

TABLA I
RENDIMIENTO DE PÉNDULO INVERTIDO

Versión	Parámetros de rendimiento				
	Clk (ns)	*Latencia (ciclos)*		*Intervalo (ciclos)*	
		Min	*Max*	*Min*	*Max*
Sin mejoras	8.621	283	283	283	283
Con mejoras	8.581	28	28	1	1

TABLA II
USO DE RECURSOS DE PÉNDULO INVERTIDO

Versión	Recursos utilizados			
	BRAM	*DSP*	*FF*	*LUT*
Sin mejoras	8	80	19036	30877
Con mejoras	0	3	1713	4721
Disponible	**280**	**220**	**106400**	**53200**

III. GENERADOR AUTOMÁTICO DE FLC

A partir de los resultados obtenidos luego de las pruebas, se desarrolló una herramienta para la generación automática de FLCs [17]. El propósito de esta herramienta es abstraer el nivel de programación para conseguir que el desarrollo, testeo y calibración de los controladores sea más ágil. Para ello se estableció que la información mínima necesaria para describir un controlador se compone de: una serie de variables, una base de reglas y los algoritmos específicos de fusificación, defusificación y evaluación de reglas, además de la cantidad de bits que usarán los conversores A/D.

Se determinó que la mejor forma de ingresar estos datos es mediante un pseudolenguaje, que permita a los usuarios generar una descripción del controlador de la manera que les resulte más cómoda. Se ponderó la posibilidad de utilizar una interfaz gráfica, pero la herramienta perdería flexibilidad, en especial a la hora de definir la base de reglas. Dicho lenguaje consta de cuatro secciones. *Declare*, donde se definen las variables que usará el controlador y su tipo (entrada o salida). *Fuzz*, donde se establecen los conjuntos difusos asociados a cada variable, además del algoritmo y parámetros que se usarán en la fusificación. *Rules*, donde se determina la base de reglas y el algoritmo para evaluarlas. Y finalmente *Defuzz*, donde se definen los algoritmos de defusificación para las variables de salida. Además de estas cuatro partes, un controlador debe indicar un nombre y la cantidad de bits que usarán los conversores del sistema.

Luego de establecer que la entrada de la herramienta es una descripción basada en un pseudolenguaje propio, y sabiendo que la salida consta de un programa escrito en

C++, se decidió que la arquitectura del generador sea similar a la de un compilador, donde se distinguen al menos tres componentes: Analizador Léxico, Analizador Sintactico y Generador de Código.

En este punto es necesario mencionar los atributos de calidad deseados para el generador de FLC. En primera instancia, dada la reducida cantidad de algoritmos disponibles en esta primera versión de la herramienta, y tomando en consideración que la incorporación de nuevas opciones son parte del trabajo futuro, la escalabilidad se convierte en un aspecto primordial. Se espera también que estos cambios puedan ser aplicados por cualquier persona, por lo que además de escalable debe ser modificable, y para ello usar herramientas estandarizadas es de gran utilidad. Otra propiedad deseada es la flexibilidad, para que se adapte de la mejor manera a la mayor cantidad de problemas. Por último, el lenguaje usado para describir los controladores debe ser cómodo de utilizar por el usuario, por lo tanto la legibilidad o usabilidad es también un atributo de calidad buscado.

La primera parte del generador desarrollada fue el analizador léxico, encargado de leer el código escrito por el usuario, reconociendo palabras reservadas, identificadores y constantes, y volcando sus características en la tabla de símbolos. Existen varias herramientas que permiten definir un analizador léxico en base a una descripción de alto nivel, sin embargo se optó por una implementación propia. Para ello se generó un autómata finito que guía el proceso. A partir del estado 0, se comienza a leer el código del usuario caracter a caracter, construyendo el lexema y cambiando de estado hasta llegar al final. En ese momento se identifica el tipo de lexema (identificador, constante, palabra reservada, etc.) y se envía el token correspondiente a la próxima etapa del generador. También puede suceder que se encuentren errores durante el proceso, en especial cuando se leen caracteres no reconocidos por el lenguaje. Para evitar que frente a estos fallos el proceso de traducción se interrumpa, se creó un estado extra, donde cualquier lexema construido hasta el momento es descartado, se notifica al usuario del error y se continúa con la lectura del código.

Luego se implementó el analizador sintáctico, para el cual se utilizó YACC (*Yet Another Compiler Compiler*), un programa que a partir de una gramática permite generar un parser ascendente [18]. Debido al formato que tiene el lenguaje de entrada, la gramática implementada se dividió en cinco conjuntos de reglas, uno para cada sección (*Declare, Fuzz, Rules* y *Defuzz*) y otro para las características generales del FLC. Esta organización, sumado al formato estandarizado de YACC, otorga gran modificabilidad y escalabilidad a la herramienta. Además, se implementaron una serie de reglas que permiten definir varios controladores desde un mismo código de entrada. Esta característica tiene como meta agilizar los tiempos de desarrollo, evitando que las descripciones de controladores distintos tengan que compilarse de forma separada.

Con respecto al tratamiento de errores sintácticos, se desarrollaron algunas reglas especiales con expresiones mal formadas para capturar fallos comunes. Entre ellos se incluye la falta de punto y coma al cerrar una sentencia, secciones del código faltantes o declaradas pero sin contenido. Los errores no contemplados con estas reglas se marcan simplemente como un error de sintaxis y, como con todos los fallos, se le notifica al usuario en que linea se produjo el problema.

Finalmente se encuentra la etapa de generación de código, que se divide en dos partes: la traducción a código intermedio y la traducción a lenguaje de salida. Su funcionamiento se basa en asociar fragmentos de código a la gramática anteriormente descrita, que se ejecuta cuando una regla es reducida. En esta etapa se destaca el uso de una representación propia que busca maximizar la escalabilidad del sistema. Es usual que en los compiladores para lenguajes de alto nivel se usen como código intermedio tercetos, polaca inversa o árboles sintácticos. No obstante, utilizar algunas de estas representaciones no tiene sentido en este trabajo, ya que el lenguaje no es tan flexible como otros de alto nivel y cuenta con una estructura bien definida fácilmente identificable gracias a la forma en que se secciona la gramática. La representación intermedia que se utilizó en su lugar cuenta con seis estructuras o clases importantes:

- **FuzzySet:** Una clase abstracta que abarca los atributos de un conjunto difuso (nombre y tipo), además de los métodos usados para la traducción a código de salida de la etapa de fusificación. Las clases que heredan de *FuzzySet* implementan los distintos tipos de funciones fusificadoras mencionadas en la sección 2.A.
- **Variable:** Clase formada por un conjunto de *FuzzySet's*. Incluye métodos utilizados en la traducción de todas las etapas del FLC.
- **IOVars:** Clase con atributos y métodos estáticos, para que solo exista una instancia de la misma durante ejecución. Almacena todas las variables y sus características reconocidas en el código. Está formada por dos vectores de tipo *Variable*, uno para las de entrada y otro para las de salida.
- **Fuzzifier:** Clase sin atributos, que utiliza la información de *IOVars* para compilar la etapa de fusificación.
- **RulesEval:** Clase abstracta que incluye una matriz, donde se almacenan las reglas declaradas por el usuario, y una serie de métodos para traducir a código de salida la etapa de evaluación de reglas. La única clase que hereda de *RulesEval* implementa el método de *MinMax*.
- **Defuzzifier:** Clase abstracta que incluye los métodos necesarios para traducir a código de salida la etapa de defusificación. La única clase que hereda de *Defuzzifier* implementa el método de *Centroide*.

Como se puede observar, en caso de querer agregar un nuevo algoritmo de fusificación, evaluación de reglas o defusificación, solo basta con crear una clase que herede de la correspondiente clase abstracta e implementar los métodos solicitados.

Durante esta etapa también se verifican errores semánticos, lo que incluye: redeclaración de variables, variables o conjuntos difusos no declarados, número de parámetros incorrectos para determinado tipo de fusificador, variables declaradas pero sin conjuntos asociados o variables de salida sin método defusificador. También se

realizan chequeos para encontrar reglas repetidas o sin declarar, pero no se muestra como un error, sino como un *warning* en caso de que el usuario se haya equivocado.

IV. Resultados

Para comprobar el funcionamiento del generador de FLC se implementaron tres problemas de lógica difusa conocidos. Los mismos cuentan con características especiales que ponen a prueba las capacidades del generador. Al igual que en la sección II, para todos ellos se implementaron dos versiones: una optimizada usando la herramienta, y otra sin optimizar de forma manual.

El primero de los problemas es el ya mostrado péndulo invertido. La única diferencia con respecto a la implementación anterior es el uso de la herramienta, cuyos resultados coinciden con los observados en la versión optimizada del algoritmo de las tablas I y II. También se podría agregar a dichos parámetros el tiempo empleado para desarrollar cada una de las soluciones, observándose que utilizar la herramienta reduce considerablemente el tiempo de desarrollo y de líneas de código necesarias para obtener un FLC optimizado.

A. Caso de uso: Estacionamiento de vehículo

El siguiente problema se basa en el estacionamiento de un vehículo, originalmente un camión, donde se asume que el movimiento es únicamente marcha atrás y a una distancia fija en cada posible estado [19]. La posición del vehículo se determina en base al ángulo (ϕ) del mismo con respecto a la zona de estacionamiento y a la posición de su parte trasera en el plano, calculada en base a las coordenadas (*x*, *y)* El objetivo es que el vehículo llegue a aparcar en el espacio delimitado (*xf* , *yf*), con ángulo nulo respecto a la pared o borde ($\phi f = 0°$). Para ello el controlador debe responder con el ángulo de dirección del vehículo (θ) para cualquier posición *(x*, *y*) o ángulo ϕ dado. En la figura 4 puede observarse la distribución de las variables aquí mencionadas.

La particularidad de este problema es que sus conjuntos de salida son de tipo Triangular y no *Singleton*. Como consecuencia, no es posible utilizar el defusificador *Centroide* sobre ellos y, al no existir otro método implementado, resultaría imposible usar la herramienta con este problema. Para revertir esta situación, se modificó el esquema de manera tal que todas las variables de salida estuvieran compuestas por *Fuzzy Sets* tipo *Singleton*, cuyo valor es la media del conjunto. Gracias a que se distribuyen de manera simétrica, el resultado obtenido para cada una de ellas se acerca al pico de la función Triangular , donde el valor de pertenencia es total ($f'(x)=1$). Si bien los resultados no serán exactamente iguales, con este pequeño cambio es posible utilizar la herramienta y luego calibrar el sistema de manera rápida hasta que el controlador responda adecuadamente. Incluso el usuario puede decidir otra forma de convertir estos valores que se adapte específicamente al problema.

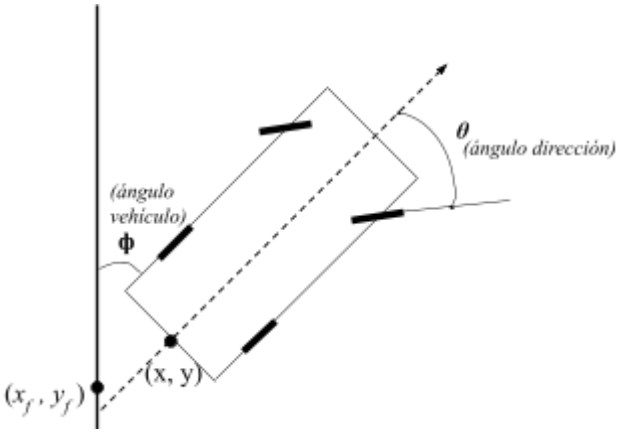

Fig 4. Esquema del problema de estacionamiento de vehículos.

TABLA III
Rendimiento de Estacionamiento de Vehículo

Versión	Parámetros de rendimiento				
	Clk (ns)	*Latencia (ciclos)*		*Intervalo (ciclos)*	
		Min	*Max*	*Min*	*Max*
Sin mejoras	9.403	202	202	202	202
Con mejoras	10.336	29	29	1	1

TABLA IV
Uso de Recursos de Estacionamiento de Vehículo

Versión	Recursos utilizados			
	BRAM	*DSP*	*FF*	*LUT*
Sin mejoras	12	79	17368	1575
Con mejoras	0	54	1575	4152
Disponible	**280**	**220**	**106400**	**53200**

Con los resultados obtenidos (Tablas III y IV), es posible afirmar que el uso de la herramienta no solo permite un desarrollo más ágil, sino además una mejora de casi 10x en la latencia y la reducción del intervalo a únicamente un ciclo. Esto implica que luego de 29 ciclos se consigue el primer resultado, y a partir de este punto por cada ciclo de 10 ns se obtiene el próximo valor. Si bien la periodo del reloj crece con respecto a la versión no mejorada, es posible reducirla a costa de aumentar la latencia. En el uso de recursos también existe una gran mejora, que se debe en gran parte a la implementación de tipos con ancho fijo. En particular, para la placa usada en simulación (xc7z020-clg400-1) del total disponible para BRAM, DSP, FF y LUT la ocupación se reduce desde 4%, 35%, 16% y 66% a 0%, 1%, 1% y 7% respectivamente.

B. Caso de uso: Autoenfoque

El último problema está basado en los sistemas automáticos de enfoque de las cámaras fotográficas, que miden la distancia con respecto al centro de la vista del localizador [20]. Si bien ofrece buenos resultados en la mayoría de los casos, en aquellas ocasiones donde el objeto de interés se encuentra fuera del centro de la vista surgen varios problemas de desenfoque. Una posible solución es medir más de una distancia, tal como podría ser la distancia a izquierda, centro y derecha, para luego utilizar lógica

difusa con el fin de encontrar el enfoque correcto (Fig. 5). El problema cuenta con tres variables de entrada: izquierda (*izq*), centro (*cen*) y derecha (*der*), además de tres posibles salidas correspondientes a la probabilidad de enfoque a izquierda (*prob_izq*), al centro (*prob_cen*) o a derecha (*prob_der*). Las variables de entrada pueden tomar valores en el rango [0;100], mientras que las de salida entre [0;1] ya que son probabilidades.

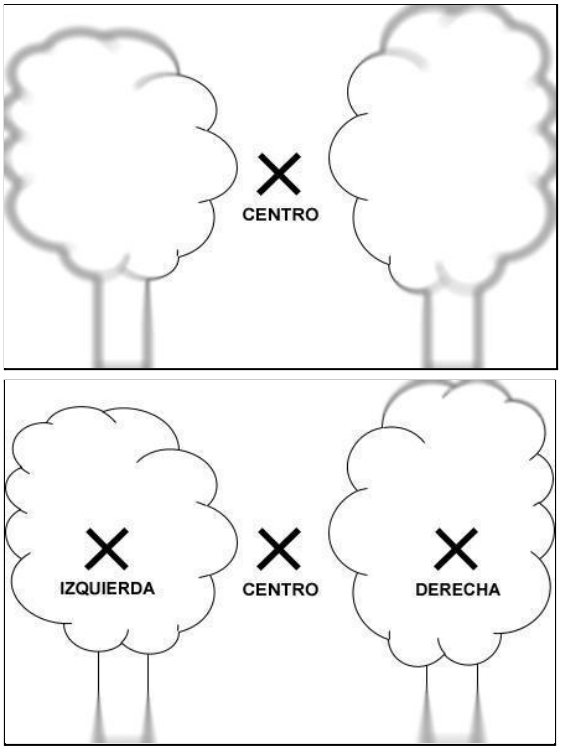

Fig 5. Distintas estrategias de enfoque. (Arriba) Enfoque con distancia al centro. (Abajo) Enfoque múltiple usando lógica difusa.

Este problema se diferencia de los anteriores en que cuenta con reglas repetidas e incompletas. Si bien la herramienta está preparada para estas situaciones, el inconveniente reside en la forma en que dichas reglas son escritas en el pseudolenguaje del generador. Por ejemplo, una de las reglas que puede encontrarse tiene la forma:

if (Izq is Cerca) then Prob_Izq is Media.

Como se puede observar, tanto su antecedente como consecuente están incompletos, especificando el valor de solo una de las tres variables de entrada y salida. Si se quisiera usar esta regla de forma directa en el código, resultaría imposible. En su defecto, se deben generar todas las posibles combinaciones para el resto de variables que no se especifican. Para asegurar que a partir de una lista de condiciones se obtenga siempre la misma base de reglas, se propone un procedimiento estándar que cuenta con seis pasos:

1. Dividir las reglas según la completitud del antecedente. Para este problema, se establece si son de grado 1, 2 o 3 si en la condición se mencionan 1, 2 o 3 variables respectivamente.
2. Se ordenan las reglas, priorizando aquellas con menor grado.
3. Se toma la primera de las reglas y se traduce al pseudolenguaje. Si la misma está incompleta se deben generar todas las posibles combinaciones para aquellas variables que no se mencionan en el antecedente. Las variables de salida no especificadas quedan en vacío.
4. Luego se toma la siguiente regla y se repite el proceso de traducción. No obstante, en este caso se debe verificar si alguna de las nuevas reglas coincide (mismo antecedente) con alguna de las ya generadas. Si eso sucede pueden pasar dos cosas:
   a. Los consecuentes pueden combinarse ya que no hay variables de salida que entren en conflicto. En estos casos, se modifica la antigua regla para que incluya al consecuente de la nueva.
   b. Los consecuentes no pueden combinarse ya que hay variables de salida que entren en conflicto. En estos casos, se genera una copia de la regla, donde la parte que entra en conflicto se modifica para que coincida con la de la nueva regla. Así se obtienen dos reglas con el mismo antecedente pero distinto consecuente.
5. El paso 4 se repite hasta traducir todas las reglas. Luego se verifica si existen aún reglas incompletas. De ser así, para aquellas variables de salida sin especificar se deben generar todas las posibles combinaciones.
6. Finalmente, se verifica que no existan reglas gemelas (mismo antecedente y consecuente). En caso de encontrarse alguna se deben eliminar las copias.

Con este procedimiento, la base de reglas obtenida cumple con todas las condiciones impuestas por el problema y siempre será la misma. Es importante mencionar que este método es solo una sugerencia y es decisión del usuario utilizarlo o no.

El controlador para este problema se sintetizó en dos versiones diferentes, tal como se hizo en problemas anteriores. Los resultados obtenidos pueden encontrarse en las tablas V y VI. Nuevamente se observa una mejora de latencia cercana a los 10x, y la reducción del intervalo a únicamente un ciclo. También se puede ver que esta optimización viene acompañada con un aumento en el periodo del *clock*, pero que se mantiene dentro del límite de 10 ns establecido en los parámetros de la síntesis. El uso de recursos también se ve enormemente reducido, pasando de un porcentaje de ocupación igual al 72%, 23% y 85%, con respecto al total de DSP, FF y LUT, a un mínimo de 2%, 3%, 12% respectivamente.

TABLA V
RENDIMIENTO DE AUTOENFOQUE

Versión	Parámetros de rendimiento				
	Clk (ns)	*Latencia (ciclos)*		*Intervalo (ciclos)*	
		Min	*Max*	*Min*	*Max*
Sin mejoras	8.960	371	371	371	371
Con mejoras	9.400	32	32	1	1

TABLA VI
USO DE RECURSOS DE AUTOENFOQUE

Versión	Recursos utilizados			
	BRAM	***DSP***	***FF***	***LUT***
Sin mejoras	0	159	24610	45531
Con mejoras	0	2	1797	3478
Disponible	**280**	**220**	**106400**	**53200**

C. Verificación de resultados

Con el fin de comprobar el correcto funcionamiento de los controladores aquí detallados, se realizaron una serie de chequeos usando distintas herramientas. Inicialmente, usando el IDE de Vivado HLS se verificaron errores sintácticos de forma rápida y sencilla.

Luego se simularon los algoritmos, comparando los resultados obtenidos en el *Test Bench* con los resultados obtenidos en una implementación manual, cuyas variables son de tipo primitivo con parte decimal. Durante esta verificación no se buscó que la coincidencia sea exacta, ya que la diferencia de tipos afecta a la precisión de los cálculos.

Finalmente, se verificó el correcto funcionamiento de la implementación RTL generada. Para ello se sintetizó el algoritmo y, mediante la co-simulación, se verificó que los resultados del *Test-Bench* coinciden tanto para la versión RTL como la versión en alto nivel.

V. CONCLUSIÓN

Se investigaron las mejores optimizaciones para los controladores desarrollados en Vivado HLS. Se encontró que la mejor configuración se basa en utilizar métodos separados para cada etapa del controlador y una serie de arreglos para comunicarlos. De esta manera, es posible aplicar directivas de síntesis específicas según los requerimientos de cada sección del FLC. Además, el uso de arreglos como buffers intermedios permite una comunicación más fluida, al habilitar un mayor *throughput* de datos y la posibilidad de aplicar otras directivas. La segunda conclusión que se extrajo es que utilizar *Pipeline* sobre cada etapa, *Dataflow* sobre la función principal y *Array Partition* sobre los buffers intermedios, permite maximizar la concurrencia y minimizar tanto la latencia como el intervalo. Todo estas optimizaciones se acompañan con el uso de tres tipos de datos con un ancho fijo de bits (*fixed_int, fixed_aux* y *fixed_out*), que no solo mejoran la performance, sino también reducen el uso de recursos considerablemente.

Finalmente, se desarrolló una herramienta que posibilita generar de manera automática descripciones sintetizables de FLCs sobre FPGAs basadas en HLS, que apliquen todas las mejoras descritas previamente. Si bien las versiones no optimizadas de los controladores (mostradas en las tablas de las secciones II y IV) operan por debajo de los tiempos de conversión A/D de las placas modernas de Xilinx (1 MSPS) [16], las métricas obtenidas luego de la síntesis de los distintos problemas muestran que, en todos los casos, utilizar el generador de FLC permite obtener en menos tiempo, una versión óptima del mismo controlador. Las mejoras consiguen reducciones cercanas a 10x para la latencia e intervalos de 1 ciclo, además de reducir el uso de recursos a menos del 15% en todos los casos.

Como trabajo futuro se propone actualizar la herramienta, agregando una interfaz gráfica, más algoritmos para fusificación, evaluación de reglas y defusificación, además de otras optimizaciones que se puedan aplicar sobre estas nuevas funciones. Además, se propone incluir a la herramienta descripciones de entrada basadas en la norma IEC 61131-7:2000, que define un lenguaje para la programación de aplicaciones de control difuso utilizado por los controladores programables.

AGRADECIMIENTOS

Este trabajo fue parcialmente financiado por la SeCAT de UNICEN (Código de Proyecto 03/C287).

REFERENCIAS

[1] Passino K.N. “Fuzzy Control”, Vol. 20. Addison Wesley, 1998.

[2] Steve M. “Fuzzy Logic Education Program”. Motorola Inc. 1992.

[3] Mandani E.H. “An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller”. International Journal of man-machine studies 7 (1). 1-13. 1975.

[4] Economakos C., Kiokes G. “Using Advanced FPGA SoC Technologies for Design of Industrial Control Applications”. International Conference on Information, Intelligence, Systems and Applications (IISA), 2015.

[5] Xilinx. “Introduction to FPGA Design with Vivado High-Level Synthesis”. 2019. https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

[6] Navarro D., Lucia O., Barragan A., Urriza I., Jimenez O. “High-level synthesis for accelerating the FPGA Implementation of computationally-demanding control algorithms for power converts”. IEEE Transactions on Industrial Informatics, vol. 9, no. 3, pp. 1371-1379, 2013.

[7] Fingeroff M. “High-Level Synthesis Blue Book”. XlibrisCorporation. 2010.

[8] Coussy P., Meredith M., Gasky D., Takach A. “An Introduction to High-Level Synthesis”. IEEE Design and Test of Computers 26(4):8 - 17. 2009.

[9] E. Lago, C.J. Jiménez, D.R. Lopez, S. Sánchez-Solano, A. Barriga, "Xfvhdl: A tool for the synthesis of fuzzy logic controllers." In Proceedings Design, Automation and Test in Europe, pp. 102-107. IEEE, 1998. <-9

[10] A. Barriga, S. Sánchez-Solano, C.J. Jiménez Fernández, "Automatic synthesis of fuzzy logic controllers." (1996).

[11] A. Costa, A., A. De Gloria, P. Faraboschi, A. Pagni. "A tool for automatic synthesis of fuzzy controllers." In Proceedings of 1994 IEEE 3rd International Fuzzy Systems Conference, pp. 1771-1775. IEEE, 1994.

[12] J.E.A. Cobo, W.A. Van Noije, L. Gualberto. "VHDL models for high level synthesis of fuzzy logic controllers." In Proceedings. XI Brazilian Symposium on Integrated Circuit Design (Cat. No. 98EX216), pp. 108-111. IEEE, 1998.

[13] Aracil J., Gordillo F. “El péndulo invertido: un desafío para el control no lineal”. Revista Iberoamericana de Automática e Ingeniería Industrial. Vol 2, No. 2, pp.8-19, 2005.

[14] Xilinx. “Vivado Design Suite User Guide: High-Level Synthesis”. 2018. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

[15] Xilinx. “Vivado Design Suite User Guide: Synthesis”. 2021. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug901-vivado-synthesis.pdf

[16] Xilinx. “7 Series FPGAs Data Sheet: Overview”. 2018. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[17] Luca Sarramone Github repository (2021), **Generador FLC**, [Online]. Available: https://github.com/LucaSarramone/ Generador-FLC.

[18] Levine, J. R., et al. “Lex & yacc”. O'Reilly Media, Inc (1992).

[19] Chen, G., & Pham, T. T. “Introduction to fuzzy sets, fuzzy logic, and fuzzy control systems”. CRC press, 2000.

[20] Mester, G. “Fuzzy Modeling of Automatic Focusing System for Compact Camera”. 2002