Proyecto de fin de carrera

Diseño e implementación de un procesador específico para la resolución de Sudokus

Director: Javier Resano

Codirector: Carlos González

Autor: Javier Olivito del Ser

Departamento de informática e ingeniería de sistemas

gaZ

Grupo de Arquitectura de Computadores

Centro Politécnico Superior

Ingeniería Informática

Abril 2010

Índice

Índice de figuras

Índice de tablas

Diseño e implementación de un procesador específico para la resolución de Sudokus

Resumen

Este proyecto surge como propuesta de participación en el Field-Programmable Technology ’09 Design Competition, concurso de diseño hardware que propuso el desarrollo de un procesador específico para resolver Sudokus de diferentes tamaños sobre una FPGA.

Nuestro primer diseño consistió en la implementación de un algoritmo de ramificación y poda, utilizando como función de poda la eliminación de candidatos mediante las reglas del Sudoku. El diseño de la memoria y de la ruta de datos estuvo encaminado a explotar el paralelismo que presenta dicha función de poda. Los resultados de esta primera versión evidenciaron una necesidad de mejora, puesto que nuestro diseño era ineficiente en la resolución de Sudokus de gran tamaño o de alta complejidad.

La versión final de nuestro procesador mejora estos resultados incorporando una etapa de preprocesamiento que aplicaba un conjunto de heurísticas capaces de acotar el espacio de búsqueda.

Paralelamente, se desarrolló una versión software equivalente que se utilizó para depurar el diseño hardware y para evaluar la eficacia de las heurísticas existentes antes de implementarlas en el procesador hardware.

Las mejoras de esta versión permiten una resolución eficiente de Sudokus de baja-media complejidad y gran tamaño (hasta orden 11: 121x121 casillas), si bien aun se muestra ineficiente en la resolución de los Sudokus de alta complejidad.

Los resultados obtenidos con este diseño nos permitieron lograr el primer premio del FPT '09 Design Competition. Además el diseño fue elegido para su presentación en el congreso y una descripción del mismo fue publicada en sus actas, siendo accesible a toda la comunidad científica a través del IEEE Xplorer.

1. Introducción

El proyecto surgió como propuesta de participación en el FPT 2009 Design Competition (en adelante, concurso), concurso internacional de diseño hardware. El objetivo del proyecto consistía en el diseño e implementación de un procesador específico para la resolución de Sudokus.

Además, la realización del proyecto conlleva varios objetivos:

- Aprender diseño hardware avanzado
- Familiarización con un entorno de descripción hardware (Xilinx ISE) y el lenguaje de descripción hardware VHDL
- Comparación de soluciones Hardware / Software para un problema dado

El contenido de esta memoria sigue un orden cronológico del desarrollo del proyecto.

La sección 2 describe el desarrollo de la primera versión del diseño, basada en la técnica de ramificación y poda, así como los resultados que motivaron el desarrollo de una segunda versión.

La sección 3 describe dicha segunda versión, la cual añade un conjunto de heurísticas que acotan en espacio de búsqueda.

La sección 4 consta de las conclusiones, tanto a nivel personal como aquellas extraídas del desarrollo del proyecto y de sus resultados.

La sección 5 muestra la planificación inicial de las tareas del proyecto y las tareas y sus distribuciones temporales finales.

El anexo I contiene el artículo que se publicó en el concurso, describiendo el trabajo presentado.

El anexo II contiene el poster que se presentó en el concurso, describiendo más someramente el trabajo.

1.1 ¿Qué es un Sudoku?

Sudoku es un juego de lógica de origen japonés que consiste, en su versión estándar, en rellenar una caja de 9x9 casillas divida en nueve cajas de 3x3 casillas con los números del 1 al 9 cumpliendo las siguientes restricciones:

- En cada fila aparece cada número una vez
- En cada columna cada número aparece una vez
- En cada caja cada número aparece una vez

El problema del Sudoku es generalizable a orden N, resultando un Sudoku de N^2 x N^2 casillas dividido en N^2 cajas de tamaño NxN, que deben ser rellenadas con los

números del 1 al N^2 siguiendo las mismas restricciones que las anteriormente mencionadas para el tamaño estándar.

Un sudoku correctamente planteado posee solución única.

1.2 ¿Qué es una FPGA?

Una FPGA (*Field Programmable Gate Array*) es un circuito integrado que contiene bloques de lógica, elementos de memoria e interconexiones, todos ellos programables. La configuración de la FPGA mediante la interconexión de los bloques lógicos y la funcionalidad de los mismos, permite generar el sistema lógico deseado.

La descripción del sistema lógico que se desea diseñar se suele realizar mediante el uso de un lenguaje de descripción de hardware, siendo los más usados VHDL (acrónimo de VHSIC HDL, *Very High Speed Integrated Circuit Hardware Description Language*) y Verilog.

1.3 Especificaciones del concurso

El concurso está centrado en la computación de propósito general en FPGAs. Más específicamente, se propone diseñar un procesador que resuelva Sudokus de distintos tamaños.

Las especificaciones detalladas más relevantes son:

- Se pretende resolver Sudokus desde orden 3 hasta orden 15

- La puntuación final será calculada mediante la siguiente expresión:

$$\sum_{N=3}^{15} \frac{N^6}{t_N}$$

  Siendo t_N el tiempo medio en resolver un Sudoku de orden N.
  El tiempo máximo permitido para resolver un Sudoku de orden N está definido por: $t_{max} = 3 * 10^{-4} N^6$ segundos.
  Todo Sudoku no resuelto en tiempo igual o menor al definido por la anterior función tendrá puntuación cero.

- Además del Sudoku resuelto, se debe calcular y enviar el *checksum* del Sudoku resuelto según la siguiente expresión:

$$\sum_{r=0}^{N^2-1} \sum_{c=0}^{N^2-1} (-1)^{((r+c)\%2)} d[r,c]$$

Siendo *d[r,c]* el valor de la casilla situada en la fila *r* y la columna *c*.

- La entrada y salida se realizará mediante RS-232 ajustándose a el siguiente formato:

  Entrada de datos a la FPGA

0xA5	0x3C	0x5A	0xC3	N*N	d[0,0]	d[0,1]	…	d[N*N,N*N]	checksum

  Salida de datos desde la FPGA

0xA5	0x3C	0x5A	0xC3	checksum	N*N	d[0,0]	d[0,1]	…	d[N*N,N*N]

- La FPGA sobre la que se implementará el diseño debe ser una de las siguientes:

  - DE2 Development and Education Board
  - XUP Virtex-II Pro Development System
  - Xtreme DSP Starter Platform – Spartan-3A DSP 1800A Edition
  - DE2-70 Development and Education Board
  - Altium NanoBoard 3000

  Nuestro diseño ha sido implementado sobre la XUP Virtex-II Pro Development System (en adelante, FPGA).

2. Versión inicial: Procesador basado en la técnica de ramificación y poda

2.1 Motivación

Como primera aproximación para la resolución de un Sudoku mediante un procesador dedicado, planteamos un algoritmo de ramificación y poda (en adelante, *backtracking*) cuya función de poda sea la eliminación como candidato para una casilla de todo número que esté en la fila, columna o caja correspondientes a dicha casilla.

La implementación hardware de este algoritmo llevada a cabo permite explotar el paralelismo que posee la función de poda, puesto que las comparaciones que precisa (comparar un posible candidato con los números fijados de su correspondiente fila, columna y caja) se pueden realizar en paralelo, permitiendo acelerar la búsqueda del siguiente candidato válido para cada casilla.

2.2 Diseño del procesador

El diseño del procesador se debe ajustar tanto a las especificaciones del concurso, como a las limitaciones que presenta la FPGA.

Son destacables dos decisiones de diseño:

a) Diseño de la memoria:

Decidimos almacenar en memoria el Sudoku de tres formas distintas. Esto triplica el espacio necesario en memoria con respecto a almacenarlo de un único modo, pero simplifica enormemente el acceso a la información necesaria en cada instante. Además diseñamos una memoria capaz de proporcionar tanto una fila, como una columna y una caja en único ciclo de reloj (ver figura 1).

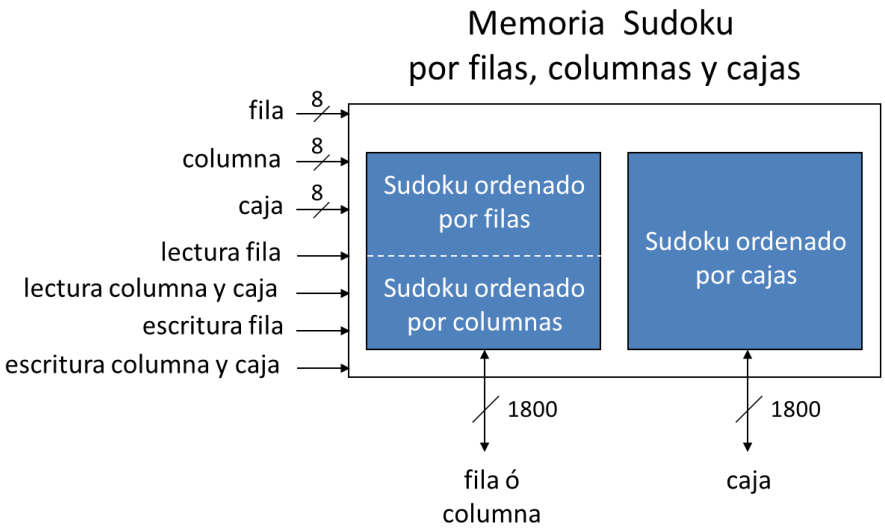

Fig. 1. Memoria capaz de suministrar una columna y una caja en un solo ciclo de reloj, y una fila, columna y caja en dos ciclos de reloj.

b) Diseño del evaluador de la función de poda:

Para explotar el paralelismo existente en la función de poda de esta versión, debemos proveer a la arquitectura de capacidad de comparación para evaluar la validez de más de un candidato simultáneamente. En nuestro diseño hemos conseguido evaluar 8 candidatos en paralelo (ver figura 2).

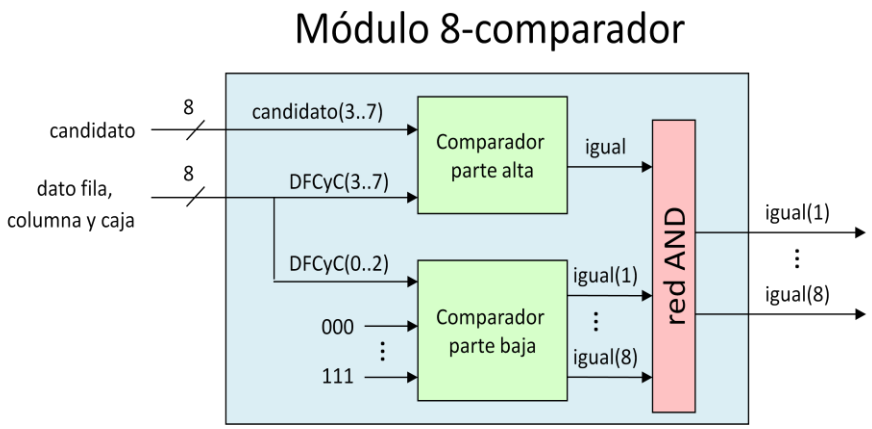

Fig. 2. Módulo 8-comparador. Sea $c_7c_6c_5c_4c_3c_2c_1c_0$ la codificación en binario de la entrada *candidato,* se compara la entrada *DFyC* con los candidatos $\{c_7c_6c_5c_4c_3000 \ldots c_7c_6c_5c_4c_3111\}$.

2.3 Detalles de implementación

La implementación del diseño se lleva a cabo mediante el lenguaje VHDL dentro de entorno Xilinx ISE 10.1.3, haciendo uso de un diseño modular con uso de genéricos, lo cual permite dimensionar la arquitectura en cualquier momento para diferentes tamaños máximos del problema.

2.3.1 Requisitos de memoria

Un Sudoku de orden 15 consiste en una matriz cuadrada de orden 225 cuyos elementos son los enteros en el rango {0..225}(1). Para cada casilla es necesario además indicar si se trata de una casilla inicialmente fija o no. Así pues, almacenar un Sudoku de orden 15 requiere 225^2 * (8 + 1) bits = 455.625 bits.

Puesto que deseamos almacenar el Sudoku ordenado por filas, por columnas y por cajas, el coste de almacenamiento del Sudoku será: 225^2 * 8 * 3 + 225^2* 1 = 1.265.625 bits.

2.3.2 Implementación de la memoria

La operación de memoria más frecuente en nuestro algoritmo de *backtracking* es la lectura de los elementos de la columna y la caja correspondientes a la casilla en procesamiento (ver figura 4), de manera que diseñamos la memoria de manera que suministre los elementos de una columna y una caja en un solo ciclo de reloj.

El algoritmo trabajará sobre un registro que almacena los elementos de la fila correspondiente a la casilla que se está procesando, y cada vez que haya un salto

(1) El '0' indica que la casilla está libre

de fila (incremento o decremento) se actualizarán las tres memorias con los datos procedentes de dicho registro (ver figura 3).

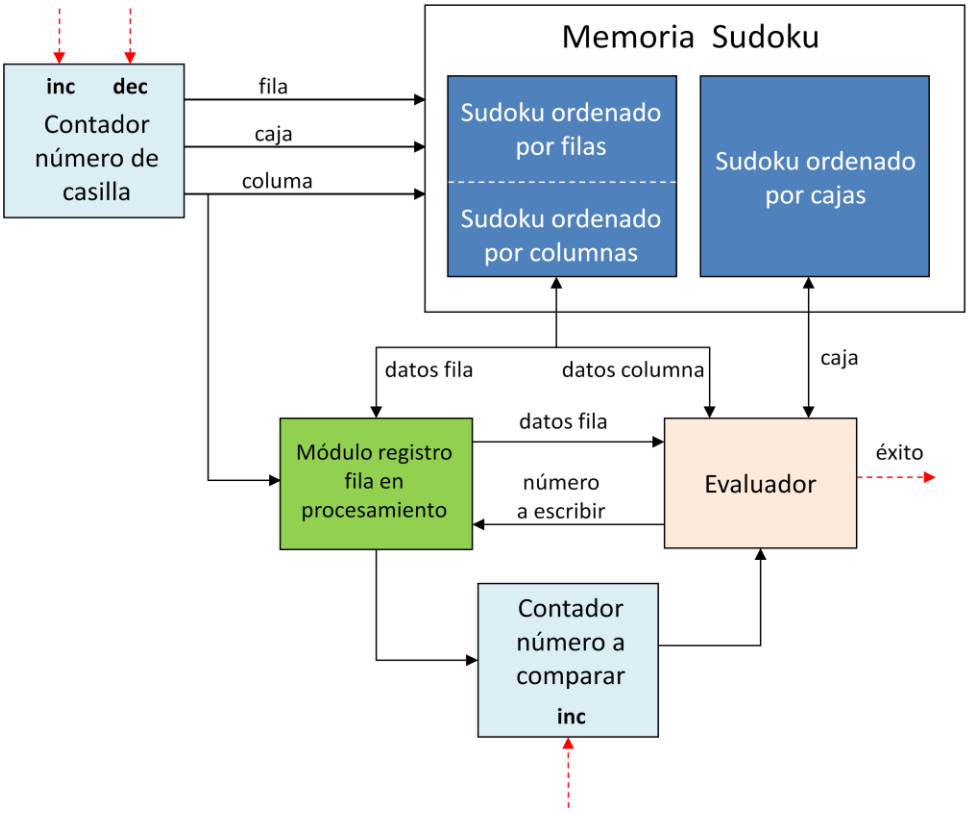

Fig. 3. Ruta de datos simplificada de la implementación del algoritmo de ramificación y poda. Las flechas discontinuas en rojo indican señales hacia/desde la unidad de control.

En la ejecución del algoritmo de *backtracking* en nuestro diseño se distinguen dos situaciones con respecto a las necesidades de la memoria: Salto de fila, y procesamiento dentro de una fila.

En la primera situación, nuestra memoria requiere de dos ciclos para suministrar la información necesaria.

En el primer ciclo se lee de memoria la fila correspondiente y se carga en un registro. En el segundo ciclo se suministra al evaluador la columna y la caja correspondientes a la casilla en procesamiento.

En la segunda situación, tan solo es necesario leer los elementos de la columna y la caja de la casilla en procesamiento, siendo así el coste de un solo ciclo.

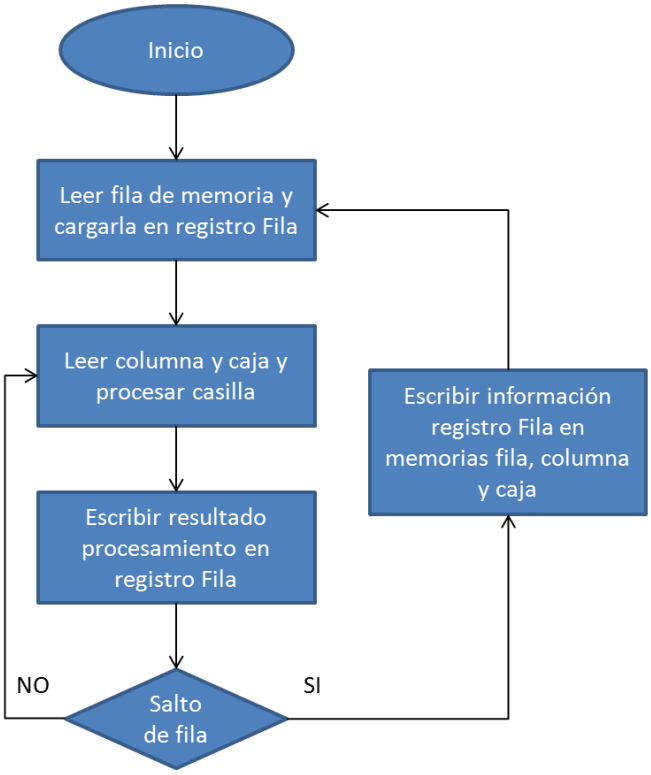

Fig. 4. Diagrama de flujo de las operaciones de memoria necesarias en la ejecución del algoritmo de *backtracking* sobre el Sudoku. La bifurcación "NO" se toma con mayor frecuencia.

2.3.3 Implementación del evaluador de la función de poda

i. Primera aproximación: Evaluación de un candidato

Comprobar la validez de un candidato para una casilla en un solo ciclo de reloj implica comparar en paralelo el candidato en cuestión con todos los elementos de su fila, columna y caja. El candidato será válido para dicha casilla cuando el resultado de todas las comparaciones sea negativo, esto es, el candidato no está en ninguna casilla de su fila, columna y caja.

Se precisan 225 * 3 = 675 comparadores de 8 bits y una red ORs de 675 entradas de 1 bit.

ii. Diseño final: Evaluación de varios candidatos en paralelo

Diseñamos un evaluador que consigue evaluar 8 candidatos en paralelo por medio de 675 módulos de comparación compuestos por un

comparador de 5 bits para los 5 bits más significativos, 8 comparadores de 3 bits para los 3 bits menos significativos, y un codificador con prioridad que selecciona el candidato que proceda en caso de varios éxitos en las comparaciones (ver figura 5).

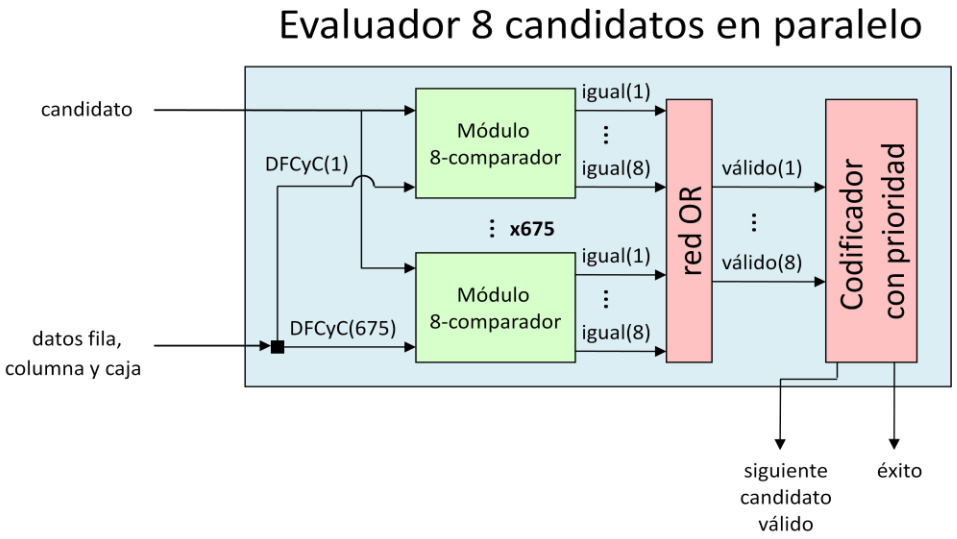

Fig. 5. Evaluador de 8 candidatos en paralelo basado en el módulo 8-comparador anteriormente descrito.

2.4 Depuración

Una vez implementado el diseño, se pudo comprobar que solo era capaz de resolver a tiempo los Sudokus 3a, 3b, 4a, 6a, 7a y 8a.

En este momento del proyecto desconocíamos las diferencias entre los Sudokus de tipo a y de tipo b de los *benchmarks*, por lo que el hecho de que nuestro diseño fuese capaz de resolver un Sudoku de orden 8 (8a) de manera prácticamente instantánea y no fuese capaz de resolver uno de orden 4 (4b) en varias horas, nos hizo pensar que nos encontrábamos ante un fallo en el diseño.

Para localizar el hipotético fallo decidimos desarrollar una versión SW equivalente que generase una traza de la ejecución del algoritmo de *backtracking*, y modificar la versión HW para que generase del mismo modo dicha traza.

La comparación de ambas nos permitiría hallar dónde fallaba nuestro diseño.

2.4.1 Resultado de la fase de depuración

Las conclusiones tras comparar las trazas de las versiones SW y HW, que resultaron ser iguales, fueron:

- Estábamos depurando algo que funcionaba correctamente
- La complejidad de un Sudoku depende fuertemente, no solo de su tamaño, sino también de la cantidad de casillas fijadas inicialmente y de su disposición.

Llegamos a la conclusión que la diferencia entre los *benchmarks* de tipo A y los de tipo B era la dificultad de los mismos. Los *benchmarks* de tipo A son de dificultad baja y los de tipo B son de dificultad alta. Dado que el concurso no facilitaba esta información, para corroborar nuestra suposición realizamos un análisis de los distintos sudokus. Los resultados se muestran en la figura 6.

Los *benchmarks* de tipo B se caracterizan por poseer un mayor número de casillas inicialmente libres que los de tipo A, además de una disposición de las mismas tal que el número de candidatos promedio de cada casilla es mucho más elevado en los de tipo B que en los de tipo A. Estos dos factores conllevan un incremento exponencial del espacio de búsqueda.

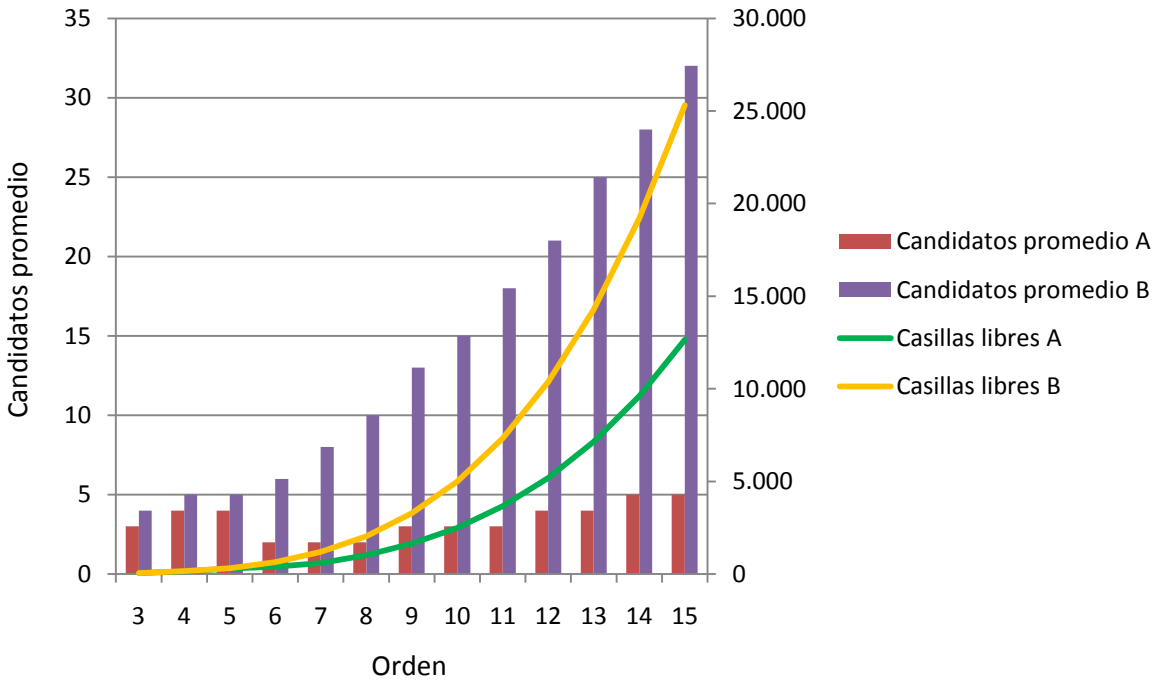

Fig. 6. Comparativa de la dificultad entre Sudokus de tipo A y B basada en el número de casillas libres y candidatos promedio iniciales.

2.5 Resultados

Los resultados de esta primera versión demuestran una alta ineficacia resolviendo Sudokus de tipo B – tan sólo es capaz de resolver a tiempo el de orden 3 -. Para los de tipo A los resultados son mejores, siendo capaz de resolver a tiempo 5 de ellos (ver tabla 1). El descenso de candidatos promedio para los Sudokus 6a, 7a y 8a (ver figura 6) explica que este diseño sea capaz de resolver los *benchmarks* 6a, 7a y 8a, pero no el 5a. En general son unos resultados pobres que manifiestan una necesidad de mejora del diseño que permita resolver un mayor número de Sudokus eficientemente.

	Orden	**t_{max}**	**Versión HW**	**Versión SW**	**Speedup HW-SW**
Dificultad Baja	3	0,2187	0,011862	0,030	2,53
	4	1,2288	0,334415	1,202	3,59
	5	4,6875	∞	∞	-
	6	13,9968	0,022935	0,040	1,74
	7	35,2947	0,041170	0,060	1,46
	8	78,6432	0,282009	3,596	12,75
	9-15	159-3417	∞	∞	-
Alta	3	0,2187	0,00769	0,020	2,60
	6-15	1,2-3147	∞	∞	-

Tabla. 1. Tiempos de resolución (en segundos) de los distintos *benchmarks*. Se muestran los tiempos del diseño hardware inicial y de la versión software equivalente.
∞ indica que el Sudoku no fue resuelto en tiempo menor o igual a su t_{max}

Versión	BRAMs	Slices
Evaluación 1 candidato/ciclo	107 (78%)	9.471 (69%)
Evaluación 2 candidatos/ciclo	107 (78%)	9.716 (70%)
Evaluación 4 candidatos/ciclo	107 (78%)	10.699 (78%)
Evaluación 8 candidatos/ciclo	107 (78%)	12.227 (89%)

Tabla 2. Recursos de la FPGA utilizados en función del número de candidatos evaluados por ciclo

3. Versión final: Procesador basado en la técnica de ramificación y poda con funciones heurísticas para acotar el espacio de búsqueda

3.1 Motivación

Anteriormente hemos probado que nos enfrentamos a un problema de búsqueda de solución en un espacio de búsqueda extremadamente amplio. La literatura sobre Sudokus ofrece una colección de técnicas de eliminación de candidatos que permiten reducir el espacio de búsqueda. Es nuestro objetivo pues conocer dichas técnicas y decidir qué conjunto de ellas implementar y cómo hacerlo para poder tratar con Sudokus no abordables hasta el momento.

3.2 Descripción de las heurísticas

Candidato único: Si existe una casilla con un solo candidato, podemos fijar dicho candidato como número de dicha casilla.

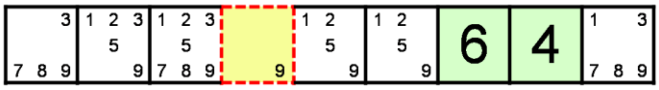

Fig. 7. Candidato único. El 9 es candidato único de la casilla 4. Podemos fijarlo y eliminar el 9 como candidato del resto de casillas.

Candidato único oculto: Dada una fila, columna o caja (en adelante, región), si un candidato dado aparece en una única casilla, podemos fijar dicho candidato como número de dicha casilla.

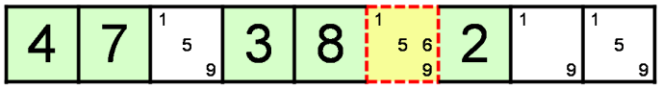

Fig. 8. Candidato único oculto. El 6 es candidato único oculto de la casilla 6. Podemos fijarlo.

Pares escondidos: Dada una región, si existe una dupla de candidatos que solo pueden ir en dos casillas de dicha región, y dichas casillas son las mismas para ambos candidatos, podemos asegurar que esas dos casillas estarán ocupadas por la dupla de candidatos en cuestión, y por lo tanto eliminar el resto de candidatos de ambas casillas.

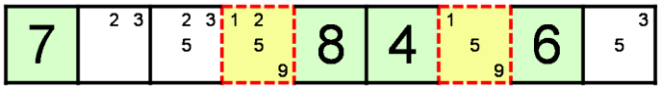

Fig. 9. Pares escondidos. El 1 y el 9 forman un par escondido en las casillas 4 y 7. Podemos eliminar el resto de candidatos de dichas casillas.

Tríos, cuartetos y quintetos escondidos: Extensión de la técnica "Pares escondidos" a tres casillas y tres candidatos, cuatro casillas y cuatro candidatos, y cinco casillas y cinco candidatos respectivamente.

Pares pelados: Si existen dos casillas en una región ambas con solo dos candidatos, y dichos candidatos son los mismos en las dos casillas, podemos eliminar ambos candidatos del resto de casillas de la región.

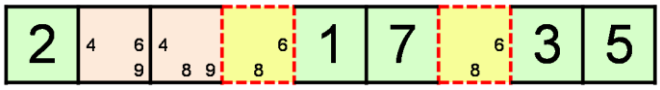

Fig. 10. Pares pelados. El 6 y el 8 forman un par pelado en las casillas 4 y 7. Podemos eliminar el 6 y el 8 como candidatos del resto de casillas.

Tríos, cuartetos y quintetos pelados: Extensión de la técnica "Pares pelados" a tres casillas y tres candidatos, cuatro casillas y cuatro candidatos, y cinco casillas y cinco candidatos respectivamente.

Líneas de candidatos: Si existe una caja en la cual algún candidato aparece únicamente en una sola fila/columna de dicha caja, podemos eliminar dicho candidato del resto de casillas de la fila/columna fuera de la caja.

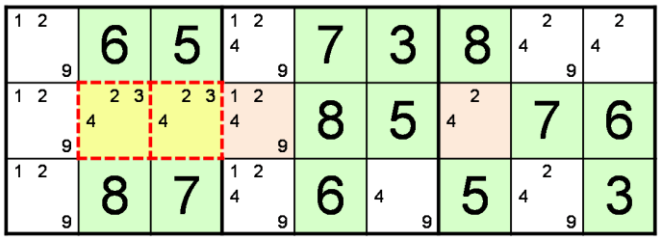

Fig. 11. Líneas de candidatos. El 4 aparece como candidato únicamente en la fila central para la caja de la izquierda. Podemos eliminar el 4 como candidato en las casillas 4 y 7 de la fila central.

Líneas dobles: Dadas dos cajas ubicadas en la misma “línea de cajas”, si existe algún candidato que solo puede ubicarse en dos filas/columnas en ambas cajas, y dichas filas/columnas son las mismas para las dos cajas, entonces podemos eliminar dicho candidato del resto de casillas de tales filas/columnas.

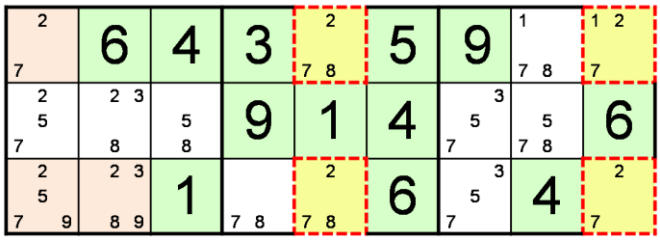

Fig. 12. Líneas dobles. En las cajas central y derecha el 2 aparece como candidato únicamente en las filas superior e inferior. Podemos eliminar el 2 como candidato en la casilla 1 de la fila superior y las casillas 1 y 2 de la fila inferior.

Líneas triples: Extensión de la técnica “Líneas dobles” a tres cajas y tres filas/columnas.

Casillas forzadas: Elegimos aquellas casillas con dos candidatos posibles. Inicialmente, asumimos que el número definitivo de dicha casilla es uno de los dos candidatos, y en función de ello, aplicamos el resto de técnicas al Sudoku resultante de aplicar dicha hipótesis, almacenando qué números se fijan y qué candidatos se eliminan como consecuencia de la asunción que hemos realizado.

A continuación repetimos el proceso eligiendo esta vez como número definitivo de la casilla el otro candidato.

Finalmente, comparamos los números fijados y los candidatos eliminados en cada asunción: aquellos números fijados que coincidan en ambas asunciones pueden ser fijados de manera segura en la solución del Sudoku, y aquellos candidatos eliminados que coincidan en ambas asunciones pueden ser eliminados de manera segura en el proceso de resolución del Sudoku.

Esta técnica es extensible a casillas con más de 2 candidatos siguiendo los mismos razonamientos.

3.3 Elección del conjunto de heurísticas a implementar

Las limitaciones de la FPGA, tanto en lógica programable como en memoria, no permiten implementar todas las heurísticas descritas.

Determinamos que la mejor manera de seleccionar el conjunto de heurísticas que serían finalmente implementadas en la versión HW, era implementar todas ellas

en la versión SW y determinar experimentalmente qué heurísticas consiguen mejores resultados sobre los Sudokus de los *benchmarks* y precisan de recursos, tanto a nivel de lógica como de memoria necesarias, que no excedan los disponibles en la FPGA.

La evaluación de las heurísticas sobre la versión SW reveló la extrema eficacia de las heurísticas "Candidatos únicos" y "Candidatos únicos ocultos". Además, estas heurísticas destacan por ser las que poseen una complejidad computacional más baja y por tener un coste de implementación bajo. Todo esto motivó su implementación en la versión HW.

La siguiente heurística elegida, siguiendo los mismos criterios, fue "Pares escondidos". Finalmente añadimos "Tríos escondidos" y "Cuartetos escondidos" por sus buenos resultados en Sudokus de gran tamaño y alta dificultad y, fundamentalmente, debido a que reutilizan hardware de la implementación de "Pares escondidos", siendo así su coste de implementación menor que otras heurísticas de similar eficacia.

Heurísticas en la versión HW	**Heurísticas en la versión SW**	
Candidatos únicos	Candidatos únicos	Tríos pelados
Candidatos únicos ocultos	Candidatos únicos ocultos	Cuartetos pelados
Pares escondidos	Pares escondidos	Quintetos pelados
Tríos escondidos	Tríos escondidos	Líneas de candidatos
Cuartetos escondidos	Cuartetos escondidos	Líneas dobles
	Quintetos escondidos	Líneas triples
	Pares pelados	Casillas forzadas (para casillas con 2 y 3 candidatos)

Tabla 3. Enumeración de las heurísticas implementadas en las versiones SW y HW finales

3.4 Diseño del procesador

El objetivo de las heurísticas es reducir el espacio de búsqueda hasta dejarlo asequible para ser resuelto mediante *backtracking*, o bien resolver el Sudoku mediante las mismas exclusivamente.

3.4.1 Estrategia global

El conjunto de heurísticas seleccionadas no garantiza la resolución de cualquier Sudoku. Necesitamos pues una estrategia híbrida consistente en la aplicación iterativa del conjunto de heurísticas hasta que no consigan reducir el espacio de búsqueda y posteriormente, si es necesario, aplicar *backtracking* sobre el Sudoku resultante de la aplicación de las heurísticas.

El algoritmo propuesto en la figura 13 garantiza encontrar solución a cualquier Sudoku(2), pero no garantiza hacerlo en un tiempo igual o menor que el máximo indicado en las especificaciones del concurso.

Este algoritmo aplica secuencialmente las heurísticas implementadas, ordenándolas según su coste computacional. En caso de éxito en la aplicación de una heurística, se vuelve a la heurística más simple que proceda(3). De este modo se aplican las heurísticas más complejas solo cuando las de menor complejidad dejan de conseguir resultados por sí mismas. Finalmente, en caso de que la heurísticas más compleja implementada no consiga resultados se procede a la búsqueda de la solución en el espacio de búsqueda resultante mediante *backtracking*.

(2) El Sudoku propuesto debe ser correcto

(3) Para las técnicas N-escondidos se vuelve a "Candidato único oculto" ya que la aplicación de N-escondidos nunca tendrá como resultado inmediato nuevos candidatos únicos

1. Aplicar Candidato único
   Si se ha fijado algún número **entonces**
   Ir a 1**;**
   Si no
   Ir a 2**;**
2. Aplicar Candidato único oculto
   Si se ha fijado algún número **entonces**
   Ir a 1**;**
   Si no
   Ir a 3**;**
3. Aplicar Pares escondidos
   Si se ha eliminado algún candidato **entonces**
   Ir a 2**;**
   Si no
   Ir a 4**;**
4. Aplicar Tríos escondidos
   Si se ha eliminado algún candidato **entonces**
   Ir a 2**;**
   Si no
   Ir a 5**;**
5. Aplicar Cuartetos escondidos
   Si se ha eliminado algún candidato **entonces**
   Ir a 2**;**
   Si no
   Ir a 6**;**
6. Aplicar *backtracking*

Fig. 13. Algoritmo de resolución mediante heurísticas y *backtracking* utilizado en la versión HW

3.5 Detalles de implementación

3.5.1 Requisitos de memoria

Todas las heurísticas se basan en la eliminación de candidatos, por lo tanto precisan conocer los candidatos de cada casilla.

Almacenar los candidatos de una casilla requiere N^2 bits, siendo N el orden del Sudoku. El coste de almacenamiento de los candidatos de cada casilla para un Sudoku será: N^2 (filas) * N^2 (columnas) * N^2 (tamaño vector candidatos casilla)

Esto representa una limitación a la hora de implementarlas, ya que, como se puede observar en la tabla 3, imposibilita alcanzar orden 12 y superiores.

Orden Sudoku	Coste almacenamiento candidatos	Coste almacenamiento Sudoku	Coste almacenamiento total
15	11.390.625	455.625	11.846.250
14	7.529.536	345.744	7.875.280
13	4.826.809	266.904	5.093.713
12	2.985.984	186.624	3.172.608
11	1.771.561	117.128	1.888.689

Tabla 4. Requisitos de memoria (en bits) para la implementación de heurísticas de eliminación de candidatos. La FPGA dispone de 2448 Kb de memoria, por lo que el diseño final queda limitado a orden 11.

3.5.2 Implementación de la memoria

La implementación de las heurísticas requiere añadir una nueva memoria para almacenar los candidatos de cada casilla.

Por cuestiones de eficiencia (obtener los candidatos de cada casilla en un ciclo de reloj, independientemente de la heurística en ejecución), añadimos una tercera memoria que almacena los números que faltan en cada fila, columna y caja.

Así pues, tenemos tres memorias en esta versión:

1) Memoria para el Sudoku
2) Memoria para los candidatos de cada casilla
3) Memoria para los candidatos de cada fila, columna y caja

La memoria 2 almacena la información resultante de la aplicación de las heurísticas N-escondidos .

La memoria 3 almacena los candidatos de cada fila, columna y caja en función de los números fijados, y permite obtener los candidatos para cada casilla según este criterio.

3.5.3 Implementación de las heurísticas seleccionadas

Candidato único:

Para cada casilla, el evaluador examina su lista de candidatos en busca de casillas con un solo candidato. El evaluador determina que existe solo un candidato en caso de que el primer y el último candidato para la casilla coincidan.

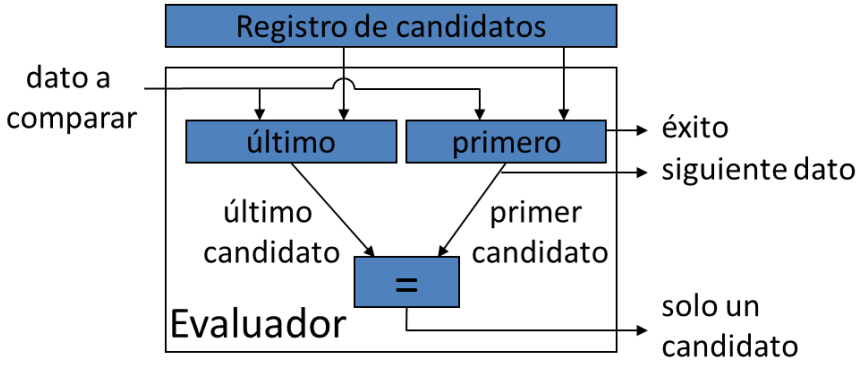

Fig. 14. Esquema hardware del evaluador que implementa la heurística "candidatos únicos".

Candidato único oculto

Para cada región del Sudoku, se examina el número de ocurrencias de cada candidato en busca de aquellos cuyo número de ocurrencias sea igual a uno.

Se tiene un contador de apariciones para cada candidato y tres registros para cada candidato que almacenan fila columna y caja respectivamente. Aquellos contadores que tras examinar las listas de candidatos de cada casilla de una región contengan un uno indicaran que el candidato que corresponde a dicho contador se puede fijar en la casilla almacenada en los registros de posición correspondientes a dicho candidato.

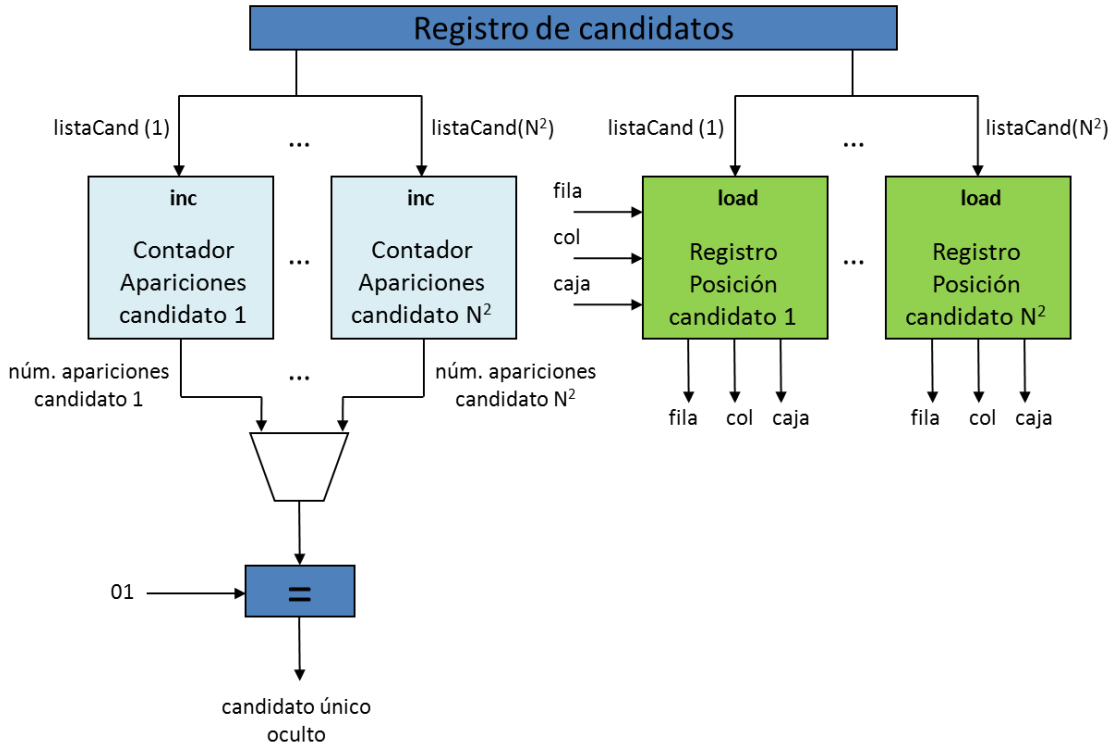

Fig. 15. Esquema de la implementación hardware de la heurística "candidato único oculto".

Par escondido:

Para cada región del Sudoku, se examina el número de ocurrencias de cada candidato almacenándolas en contadores dispuestos para cada candidato. Se almacenan también las dos últimas posiciones en las cuales aparece cada candidato en los registros de posición.

Posteriormente se comparan todas las duplas de candidatos en busca de aquellas cuyo número de apariciones para los dos candidatos sea dos y las posiciones de los dos candidatos sean las mismas. Aquellas casillas que cumplan estas condiciones contienen un par escondido.

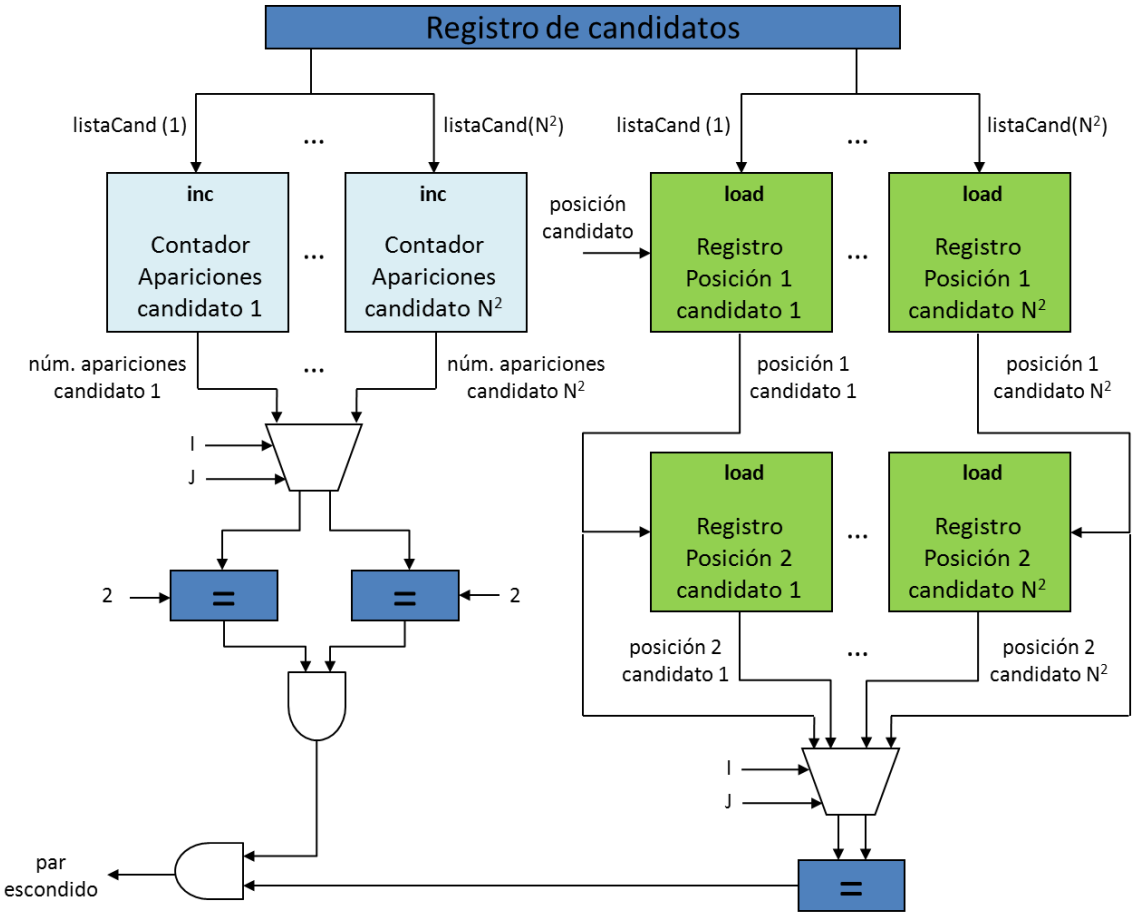

Fig. 16. Esquema de la implementación hardware de la heurística "Pares escondidos".

Trío y cuarteto escondido:

Dada una región de un Sudoku, una G-tupla de candidatos forma un G-escondido si se cumple:

i. Todo candidato de la G-tupla aparece {2,…,G} veces
ii. El número de posiciones distintas de los candidatos de la G-tupla es exactamente G

Para descubrir un G-escondido, con G ≥ 3, es necesario añadir los siguientes elementos a la ruta de datos:

1) G registros de posición de los candidatos a examen de la G-tupla.
   Almacena las G posiciones de las casillas donde se ubica el G-escondido.

2) Un contador de número de posiciones de $\lfloor \log_2 G \rfloor + 1$ bits.
   Verifica la condición ii de un G-escondido.

Además, se precisa modificar los siguientes elementos de la ruta de datos para pares escondidos:

1) Memoria de candidatos para establecer casillas con 2..G candidatos.
2) G * N^2 registros de posición
3) N^2 contadores de apariciones de $\lfloor \log_2 G \rfloor + 1$bits
4) G contadores índice de $\lceil \log_2 N \rceil$ bits

Debido a las limitaciones de la FPGA, solo se implementó esta técnica para G = 2, 3, 4.

3.6 Resultados

La tabla 5 muestra los recursos (memoria y lógica) utilizados por el diseño final en función de qué heurísticas se implementen.

La tabla 6 muestra los tiempos de resolución de los *benchmarks* de la versión presentada al concurso.

Esta versión es muy eficiente resolviendo Sudokus de tipo A, si bien aun no es capaz de tratar satisfactoriamente con los de tipo B.

Versión	BRAMs	Slices
Candidato único	29 (21,1%)	2.183 (16%)
Candidato único oculto	134 (97,8%)	7.675 (56%)
Pares escondidos	134 (97,8%)	9.187 (67%)
Tríos escondidos	134 (97,8%)	10.080 (78%)
Cuartetos escondidos	134 (97,8%)	12.265 (90%)

Tabla 5. Recursos de la FPGA utilizados en función de las heurísticas implementadas. Cada fila representa una versión que implementa la heurística indicada y las de sus filas superiores.

	Orden	t_{max}	Versión HW	Versión SW 1	Speedup HW-SW1	Versión SW 2	Speedup HW-SW2
Dificultad Baja	3	0,2187	0,008600	<0,001	0,116	0,03	3,49
	4	1,2288	0,024761	0,01	0,404	0,010	0,40
	5	4,6875	0,060281	0,03	0,498	0,038	0,63
	6	13,9968	0,117894	0,01	0,085	0,024	0,20
	7	35,2947	0,224523	0,02	0,089	0,522	2,30
	8	78,6432	0,058375	0,04	0,685	3,210	55,00
	9	159,4323	0,300835	0,1	0,332	10,402	34,58
	10	300,0000	0,322603	0,234	0,725	48,520	150,40
	11	531,4683	0,381182	0,464	1,217	0,652	1,71
	12	895,7952	-	1,53	-	∞	-
	13	1.448,0427	-	3,137	-	∞	-
	14	2.258,8608	-	6,898	-	∞	-
	15	3.417,1875	-	151,561	-	∞	-
Dificultad alta	3	0,2187	0,009762	0,01	1,02	0,009	0,92
	4	1,2288	∞	∞	-	1,810	-
	5	4,6875	∞	∞	-	56,133	-
	6-15	14-3.417	∞	∞	-	∞	-

Tabla 6. Tiempos de resolución (en segundos) de los distintos *benchmarks*.
La versión SW 1 utiliza las mismas heurísticas que la versión HW. La versión SW 2 utiliza todas las heurísticas descritas en la sección 3.2 y se caracteriza por resolver los Sudokus exclusivamente mediante heurísticas.

4. Conclusiones

Los distintos resultados obtenidos y la experiencia adquirida durante el desarrollo de los diseños expuestos en este trabajo permiten extraer las siguientes conclusiones:

Factores que intervienen en la complejidad del Sudoku:

En un principio establecíamos la complejidad de un Sudoku en función de su tamaño. Esto nos condujo a interpretar incorrectamente los resultados del diseño inicial: La resolución inmediata de un Sudoku de orden 8 y la no resolución en horas de uno de orden 4 nos hizo dedicar un mes en busca de un fallo que no existía. Finalmente, comprendimos que las diferencias entre *benchmarks* de tipo A y B, expuestas en la sección 2.4.1, influyen en gran medida en la complejidad de resolución.

Paradójicamente, dicho mes buscando el fallo que no existía acabó siendo de gran utilidad debido a que condujo al desarrollo de la versión SW, que acabó resultando crucial para el diseño de la versión HW final.

Naturaleza exponencial del problema:

El crecimiento exponencial del espacio de búsqueda exige incorporar funciones de poda de tiempo polinómico que acoten el espacio de búsqueda.

Distintas pruebas llevadas a cabo nos permitieron comprobar que la aplicación iterativa de las heurísticas “candidato único” y “candidato único oculto” en los *benchmarks* de tipo A de hasta orden 14, acotan el espacio de búsqueda en tiempos del orden de milisegundos de manera que el algoritmo de *backtracking* encuentra la solución en el árbol podado en tiempos del orden de microsegundos.

Cabe destacar que “candidato único” y “candidato único oculto” son además las heurísticas de menor complejidad.

La resolución de los *benchmarks* de tipo B requiere utilizar un conjunto más amplio de heurísticas, que además, a diferencia de los de tipo A, es creciente con el orden del Sudoku.

Comparativa Hardware – Software:

El diseño hardware permite obtener procesadores muy eficientes en la resolución de un problema dado. Los resultados (ver tabla 6) muestran cómo nuestro diseño HW implementado sobre tecnología de 2002 y con un reloj a 32 MHz. es capaz de obtener tiempos similares a una versión software equivalente corriendo sobre un computador de 2007 con un procesador trabajando a 1,66 GHz.

La principal desventaja que presenta el diseño hardware en comparación con el desarrollo software es la mayor complejidad del primero y la necesidad de un mayor tiempo de depuración. La versatilidad y facilidad de uso de recursos de los desarrollos software es también un punto a favor de los mismos.

Es importante también considerar que el mayor beneficio que obtiene un diseño hardware específico reside en la explotación del paralelismo del problema que se trate. Cabe destacar que el problema al que nos hemos enfrentado en este trabajo no destaca precisamente por su alto grado de paralelismo, dado que el valor de una casilla influencia sobre el resto, debiendo tratarse de manera secuencial.

Estrategia híbrida:

En determinados casos, aplicar tantas heurísticas como sean necesarias para resolver el Sudoku sin necesidad de *backtracking* resulta perjudicial en términos de tiempo de resolución con respecto a una estrategia híbrida consistente en la aplicación de un menor número de heurísticas y *backtracking* (ver tabla 6, columnas SW1 y SW2).

Mejoras futuras:

El objetivo ideal sería un diseño capaz de resolver a tiempo todo el abanico de Sudokus de los *benchmarks*. Para ello son precisas una serie de mejoras encaminadas a dos objetivos:

1) Tratar con Sudokus de hasta orden 15

   Se contemplan dos posibilidades para tratar con Sudokus de hasta orden 15. La primera sería utilizar la DDR SDRAM de la FPGA sobre la que hemos implementado el diseño, cuyo módulo de memoria dispone de suficiente memoria para cumplir con los requisitos detallados en la tabla 4. Llevar esto a cabo implica implementar un controlador de memoria para dicha placa.

   La segunda posibilidad sería utilizar una placa con mayores prestaciones, en concreto, con suficiente memoria SRAM para albergar las estructuras necesarias para Sudokus de orden 15 (ver tabla 4), si bien esta opción no hubiese sido válida para el concurso, ya que el conjunto de placas sobre las que se podía implementar el diseño (ver sección 1.3) se caracterizaban por ser de similares características. Ninguna de ellas poseía suficiente memoria para tratar con Sudokus de orden 12 o superiores mediante las estrategias discutidas en la versión final.

2) Resolver eficientemente Sudokus de dificultad alta

   Las pruebas realizadas sobre la versión SW demostraron que la aplicación de un mayor conjunto heurísticas consigue resolver eficientemente Sudokus de tipo B. Sin embargo, al contrario que los Sudokus de tipo A, los de tipo B requieren un conjunto de heurísticas creciente con el tamaño del Sudoku. Por lo tanto, se propone como mejora

la adición de las heurísticas descritas en la sección 3.2, así como otras adicionales, previo estudio de la eficacia de las mismas en la versión SW.

Además, la implementación de una función (obtenida experimentalmente), que determine cuando es más conveniente aplicar *backtracking* en detrimento de la aplicación de sucesivas heurísticas, conduciría a un diseño mucho más eficiente en la resolución de Sudokus, tanto de dificultad alta como baja. Para ello sería necesario realizar un análisis similar al explicado en la figura 6 en tiempo de ejecución.

Experiencia personal:

El desarrollo de ese PFC me ha permitido iniciarme en el diseño hardware (herramientas, lenguajes, metodología, etc...). Dentro de mi formación como ingeniero en informática he adquirido especial interés por el área de la arquitectura de computadores, y dentro de esta, tras haber realizado este proyecto, el diseño hardware me ha motivado especialmente. Espero que en el futuro pueda seguir aplicando los conocimientos adquiridos.

El desconocimiento inicial de la naturaleza del problema que hemos tratado y de los métodos de resolución me han permitido adquirir experiencia en cómo abordar un problema complejo y desconocido, y a interpretar resultados para una mejor comprensión del problema y como fuente de posibles mejoras. Esto me ha permitido aprender a aplicar técnicas de depuración que permitan encontrar más rápidamente los errores, y de estimación y análisis que ayuden a tomar decisiones de diseño.

5. Planificación

En un principio tan solo planteamos llevar a cabo la implementación del diseño hardware, como se aprecia en la figura 15.

El desarrollo final del proyecto incluyó el desarrollo en paralelo de una versión software y una fase de depuración del diseño hardware más prolongada de lo esperado (figura 16).

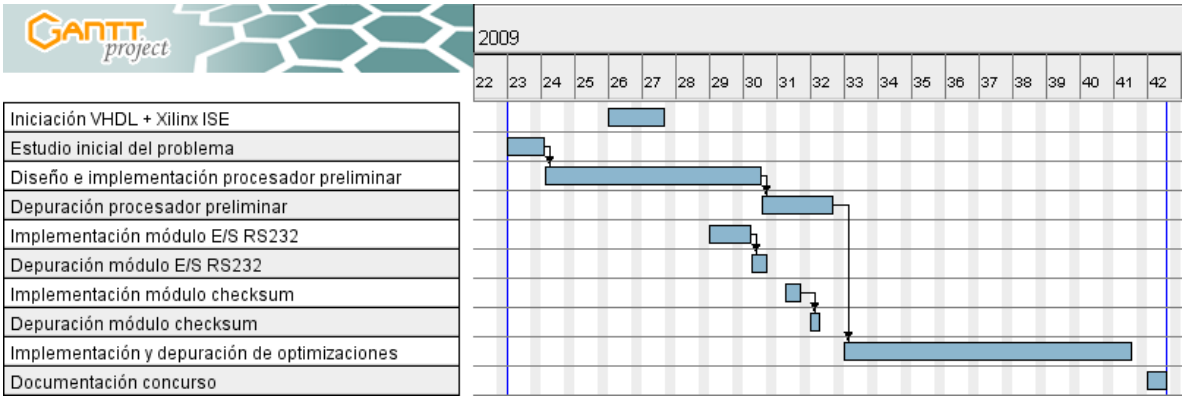

Figura 17. Diagrama de Gantt de la planificación inicial del proyecto.

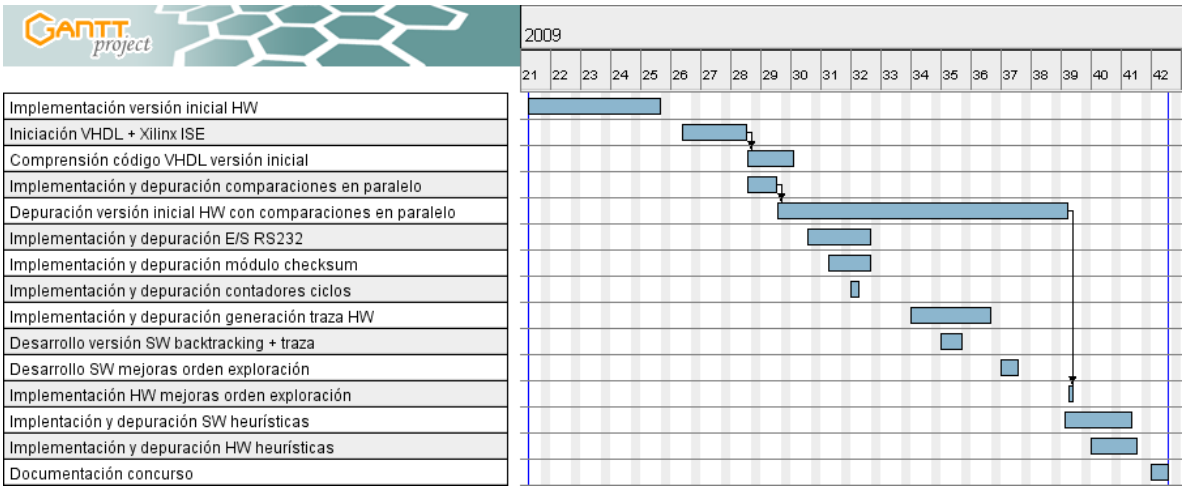

Figura 18. Diagrama de Gantt del desarrollo real del proyecto

La dedicación en este periodo fue prácticamente completa con un total de 760 horas dedicadas al desarrollo de este proyecto.

Como se puede ver en las figuras 17 y 18, pudimos cumplir el plazo final marcado en la planificación, el cual era imprescindible para participar en el concurso. También se puede observar como una de las tareas que inicialmente debería durar 2 semanas, acabó durando 10, debido a los problemas que encontramos durante la etapa de diseño, que eran imprevisibles a la hora de planificar.