Universidad Zaragoza

Trabajo Fin de Grado

Codiseño hardware software para la implementación eficiente una aplicación del juego Trax

Autor

José Ángel Castán Sánchez

Director

Jesús Javier Resano Ezcaray

Escuela de Ingenieria y Arquitectura

Curso 2015-2016

Codiseño hardware software para la implementación eficiente una aplicación del juego Trax

Resumen

Recientemente ha aparecido en el mercado un nuevo tipo de SoCs (System on chip) que incluyen un sistema basado en procesadores ARM, como los que se pueden encontrar en los teléfonos móviles o tabletas, junto con una FPGA (Field Programable Gate Array), que es un dispositivo hardware que se puede programar para que realice la funcionalidad deseada.
Este tipo de plataformas permite diseñar SoCs a medida para una aplicación dada en los que la mayor parte de la funcionalidad se ejecuta de forma convencional en el procesador, pero algunas funciones críticas se ejecutan en aceleradores hardware implementados en la FPGA.
Este proyecto pretende analizar las posibilidades que ofrecen este nuevo tipo de SoCs utilizando como aplicación de prueba un juego de mesa llamado TRAX. En este juego dos jugadores tratan de construir un camino cerrado antes que el rival. Los juegos de mesa son aplicaciones con gran demanda computacional y pueden beneficiarse ampliamente de aceleradores hardware que permitan procesar el tablero de forma más eficiente. Por esa razón el juego TRAX ha sido elegido para la competición de diseño en FPGA en el *International Conference on Field-Programable Technology* del año 2015.
A la hora de diseñar la aplicación, se ha optado por utilizar una inteligencia artificial muy sencilla y centrarnos en implementar de la forma más eficiente posible las funciones que procesan el tablero tanto en hardware como en software. Ambas versiones son funcionalmente equivalentes y pueden utilizarse indistintamente. También se ha tratado de optimizar las comunicaciones entre el procesador que ejecuta el código y los aceleradores.
Tras verificar el correcto funcionamiento de las funciones desarrolladas hemos comparado el rendimiento obtenido en cada caso. Los resultados muestran que la versión que utiliza los aceleradores es 11.5 veces más rápida de media y consume 12.5 veces menos de energía.

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. José Ángel Castán Sánchez,

con nº de DNI 73001340F en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado en Ingenieria Informática, (Título del Trabajo) Codiseñor hardware software para la implementación eficiente una aplicación del juego TRAX,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 2 de febrero de 2016

Fdo: José Ángel Castán Sánchez

Índice

1. Introducción

Desde hace ya unos años han aparecido en el mercado como plataformas de desarrollo los System-on-Chip (SoC). Estos SoC, como su nombre indica, se componen de un sistema con distintos recursos computacionales que se agrupa en un solo chip. Frecuentemente estos sistemas son heterogéneos, incluyendo tanto procesadores de propósito general, como procesadores para gráficos, para procesamiento digital de señales o aceleradores hardware. De esta forma, pueden ejecutar parte de la carga de trabajo en un procesador convencional, pero recurrir para las partes especialmente críticas a los procesadores especializados.

El agrupamiento de estos distintos recursos proporciona un alto rendimiento en un pequeño espacio, optimizando las comunicaciones y reduciendo el consumo de potencia y energía, lo que ha producido que sean muy utilizados en los dispositivos que utilizan baterías cómo teléfonos móviles o tabletas. Dos ejemplos representativos son los Exynos de Samsung[1] o el A8 de Apple[2] (Fig.1), que incluyen varios procesadores ARM, recursos de memoria, diversos controladores, aceleradores hardware y procesadores para gráficos.

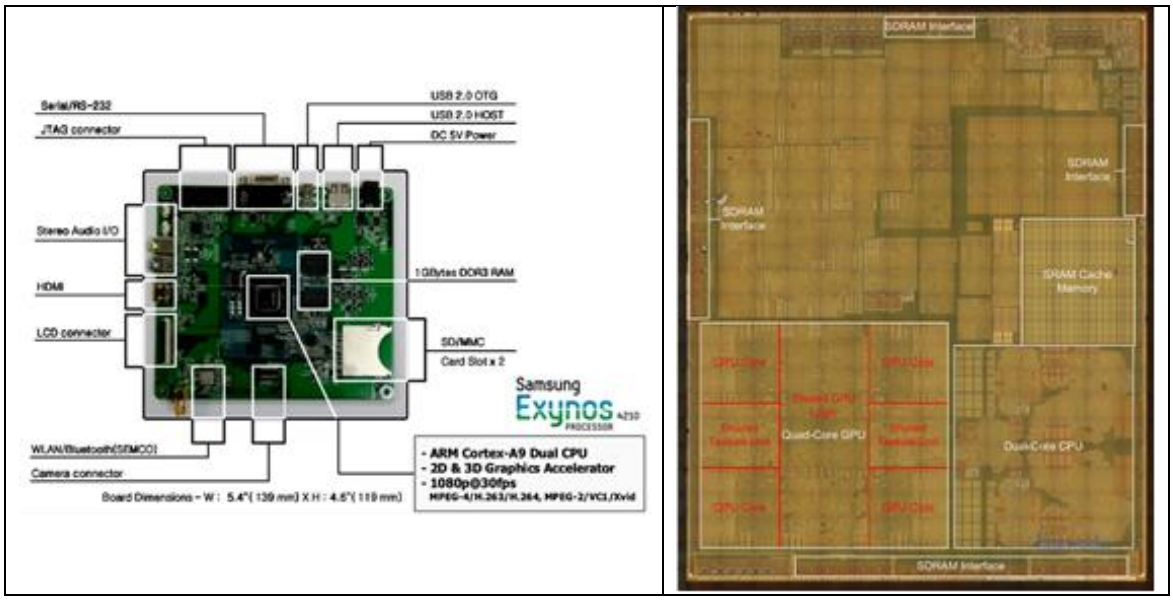


Figura 1. SoCs Exynos 4210 y Apple A8

Este tipo de SoCs no se utilizan únicamente en dispositivos móviles, sino que también se utilizan para placas de desarrollo, y para sistemas en los que se demande alto rendimiento, pero también eficiencia energética, siendo el caso de la Tegra X1 de Nvidia[3] o la OMAP 5 de Texas Instruments[4]. Con la Tegra X1 se puede procesar imágenes en tiempo real en resolución Ultra HD con el Nvidia SHIELD, y las OMAP se han utilizado para el dispositivo Google Glass

[1] http://www.anandtech.com/show/4900/samsung-talks-about-32nm-15ghz-exynos-soc
[2] http://www.anandtech.com/show/8562/chipworks-a8
[3] http://www.nvidia.com/object/tegra-x1-processor.html
[4] http://www.ti.com/product/OMAP5432

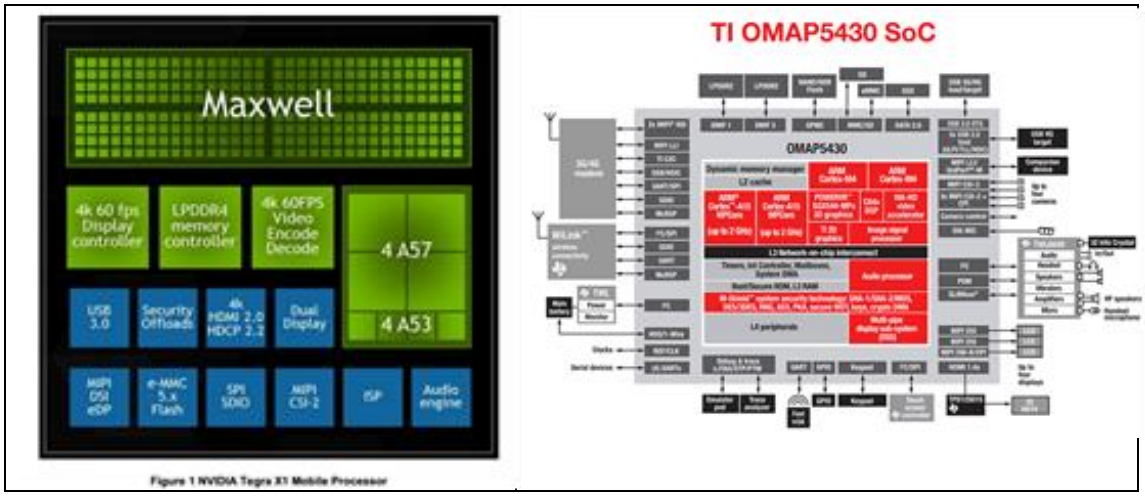


Figura 2. SoCs Tetra X1 y OMAP 5

En este tipo de SoCs una forma de conseguir a la vez rendimiento y eficiencia energética es incluir aceleradores hardware, existiendo principalmente dos tipos. El primero de estos tipos son los Circuitos Integrados para Aplicaciones Específicas (o *ASICs*, por sus siglas en inglés). Estos aceleradores consiguen una gran mejora en rendimiento y consumo, pero a costa de que la funcionalidad sea fija. Es importante que esta funcionalidad a acelerar se utilice mucho, como puede ser algoritmos de encriptado o la descompresión de audio, ya que serán aceleradores específicos para esa aplicación y no se podrán utilizar para nada más.

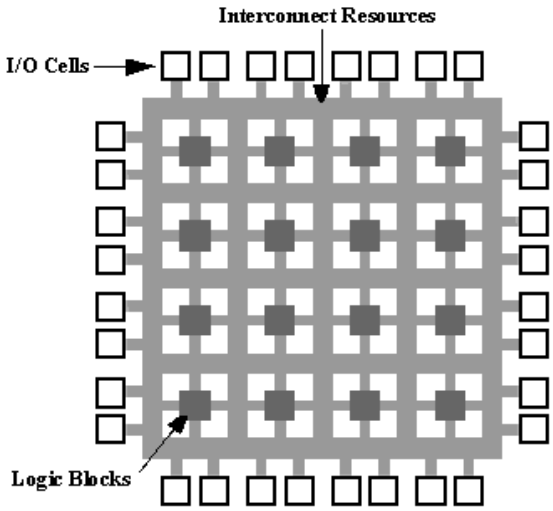


Figura 3. Arquitectura interna de una FPGA

Frente a esta opción ha surgido recientemente la posibilidad de incluir Field Programmable Gate Arrays (FPGA) en los SoCs. Estas FPGAs son también circuitos integrados como los ASIC, pero la principal diferencia es que se pueden configurar tantas veces como sea necesario. La arquitectura interna específica es una malla de bloques lógicos programables que se pueden utilizar para implementar la funcionalidad deseada tanto secuencial como combinacional. Según las necesidades, se utilizaran más o menos bloques conectándolos entre sí. Estas conexiones de nuevo pueden configurarse a la mediada de un diseño dado. Finalmente el diseño puede conectarse con los

distintos puertos de entrada/salida, y utilizar los recursos de procesamiento interno (bloques especiales para procesamiento digital de señal) y los recursos de memoria RAM empotrados en la FPGA.
Las FPGAs se pueden usar como plataforma definitiva de un diseño dado, o como paso previo para probar prototipos de funcionalidad para ASICs, ya que utilizando la misma descripción de alto nivel de un circuito digital se puede implementar tanto en FPGA como en ASIC. Si bien el proceso de diseño de un ASIC es mucho más complejo y caro. Por el contrario las ASICs proporcionan mejor rendimiento y menos consumo de energía dado que la flexibilidad de las FPGAs tiene como contrapartida una merma en su eficiencia. Esto no significa que no sean suficientemente potentes para la mayor parte de las aplicaciones. Las FPGAs actuales proporcionan gran cantidad de recursos que se pueden utilizar en paralelo, proporcionando muy buenos números tanto en rendimiento, como en eficiencia (rendimiento por vatio consumido). Un ejemplo interesante es un proyecto de conducción automática que está desarrollando Audi[5]. En este proyecto la computación se realizaba en un servidor con varios procesadores que ocupaba el maletero de un coche, pero en la siguiente versión, ya más cerca de producción, el servidor ha sido sustituido por una FPGA que se ocupa de todo.
Parece que la industria ha acogido con gran interés los SOCs que incluyen procesadores y FPGAs. Por ese motivo, y también porque son plataformas de desarrollo baratas y versátiles, hemos decidido estudiar en este proyecto las ventajas de utilizar este tipo de SOCs para una aplicación dada. A continuación se presenta la plataforma de desarrollo y la aplicación seleccionada.

1.1 Plataforma

La plataforma seleccionada es la ZedBoard[6] (Fig.4) de la empresa Xilinx que es actualmente el mayor fabricante de FPGAs. Esta plataforma está basada en el chip Xilinx Zynq-7000 Extensible Processing Platform.

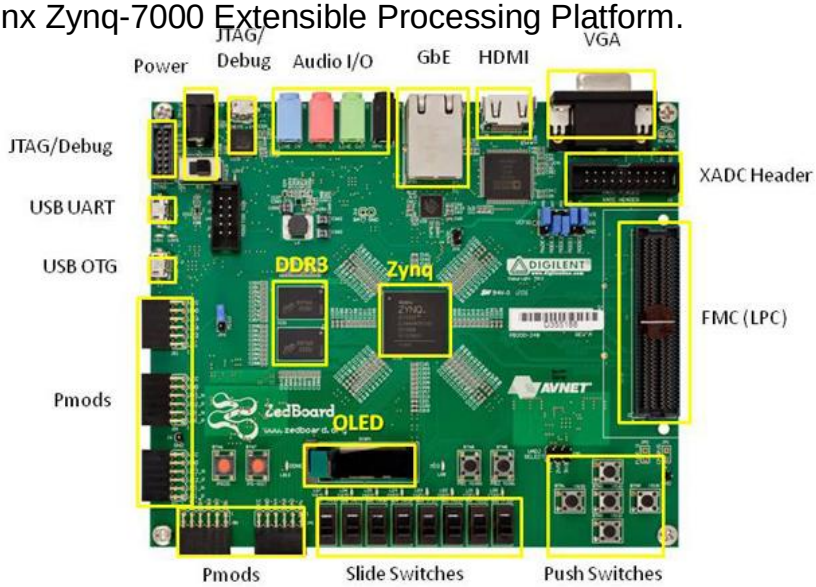


Figura 4. Placa de desarrollo ZedBoard

[5] http://newsroom.altera.com/press-releases/nr-altera-audi-adas-cyclone-v.htm
[6] http://zedboard.org/product/zedboard

La Zynq[7] (Fig. 5) está compuesta por un ARM Cortex A9 de dos núcleos, una FPGA, y diversos controladores y recursos de memoria. El reloj del ARM en nuestro chip es de666 MHz, aunque puede ser más elevado en otros chips. Para las comunicaciones entre el Cortex A9 y la FPGA, se utiliza el interfaz Advanced eXtensible Interface (AXI) en su versión 4, que es una

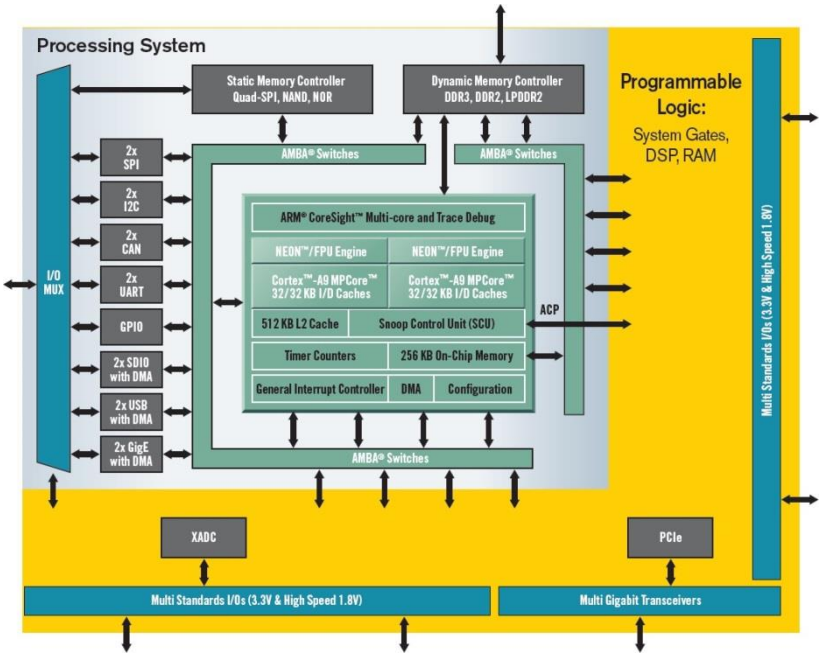


Figura 5. Diagrama de Zynq 7000

especificación del estándar Advanced Microcontroller Bus Architecture (AMBA) para ARM[8]. Tiene tres tipos de conexiones:

1 AXI4: orientada a una gran utilización del canal en ambos sentidos

2 AXI4-Lite: orientada a comunicación en forma de registros mapeados en memoria que son compartidos por el procesador y el periférico.

3 AXI-Stream: orientada a gran transferencia de datos en un sentido

Lo mejor de usar este estándar, es que una vez diseñado el acelerador para la FPGA y su interfaz, el procesador se podrá comunicar con el acelerador como si fuera cualquier otro periférico, con llamadas a funciones que escriben y leen datos a través de direcciones mapeadas en memoria.

1.2 Aplicación de prueba: TRAX

La aplicación que se va a utilizar para analizar las posibilidades que ofrece usar FPGAs como aceleradores es un juego de mesa llamado TRAX.

En este juego dos jugadores, uno blanco otro negro, tratan de construir un camino cerrado antes que el rival, usando una serie de fichas. En la Figura 6 se puede ver una partida ganada por el jugador blanco al conseguir cerrar un camino.

[7] http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
[8] http://www.xilinx.com/ipcenter/axi4.htm

Figura 6. Partida ganada por el jugador blanco

Los juegos de mesa se pueden usar como reto para evaluar una nueva tecnología dada su alta demanda computacional. En el caso de este juego, fue sido elegido para la competición de diseño en FPGA en el *International Conference on Field-programable Technology* del año 2015[9]. En este congreso, la placa ZedBoard, que hemos utilizado, era una de las posibles plataformas válidas para participar en el concurso. Además esta aplicación volverá a utilizarse como reto en la próxima edición de esta competición de diseño hardware en Diciembre del 2016[10] por lo que, si se obtiene la financiación necesaria, existe la posibilidad de presentar el diseño realizado en esa competición.

Pasaremos a explicar el juego en cuestión. Lo primero es presentar las fichas. Hay sólo dos tipos de ficha, que pueden rotarse para obtener las seis combinaciones de la figura 7.

Figura 7. Fichas del juego Trax

Ambos jugadores pueden poner cualquiera de estas fichas, Las fichas se pueden poner en los bordes del tablero que ya existe, junto a otra ficha, respetando que el color sea continuación de los adyacentes. En el caso de que algún borde tuviese colores cambiados, ese movimiento no se podría hacer ya que sería un movimiento inválido.

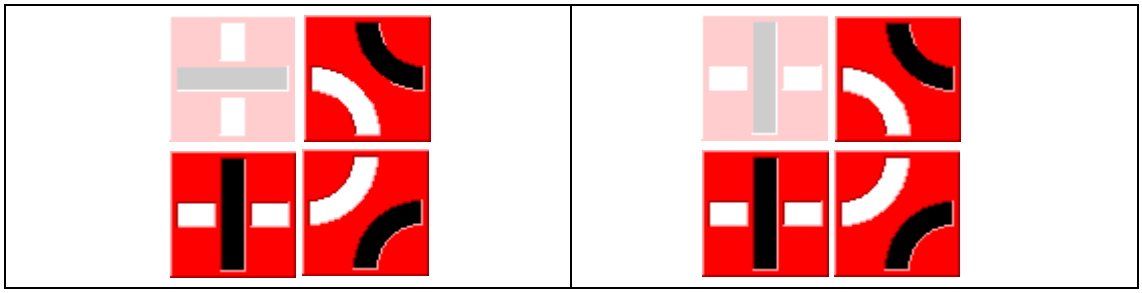


Figura 8. Movimiento inválido Movimiento válido

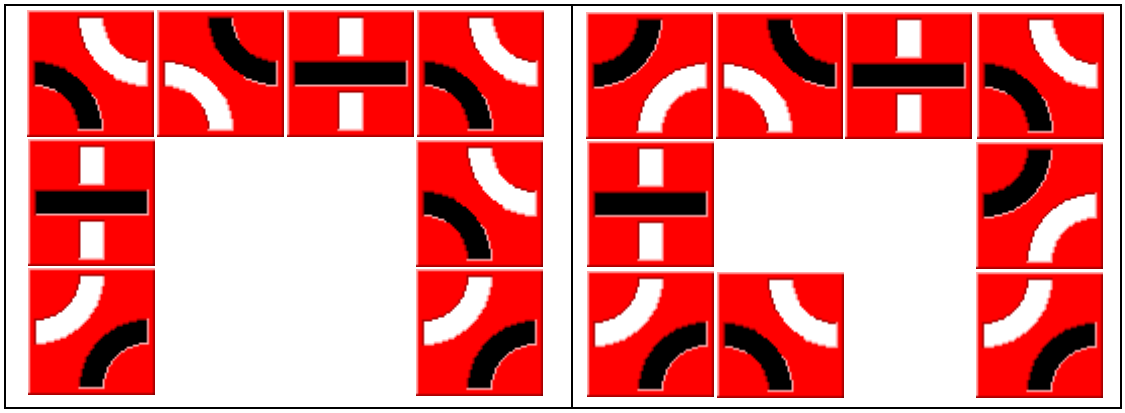
Las reglas se complican ya que tras colocar una ficha es posible que se deban colocar más de forma automática y obligatoria. Estos movimientos obligatorios se producen cuando en una casilla sólo existe la posibilidad de colocar una ficha en una determinada posición. Esto sucede cuando a una casilla vacía le llegan dos caminos del mismo color. En esa situación el único

[9] http://fpt.massey.ac.nz/design_competition_rules.asp
[10] http://www.icfpt2016.org/designcomp_details.jsp

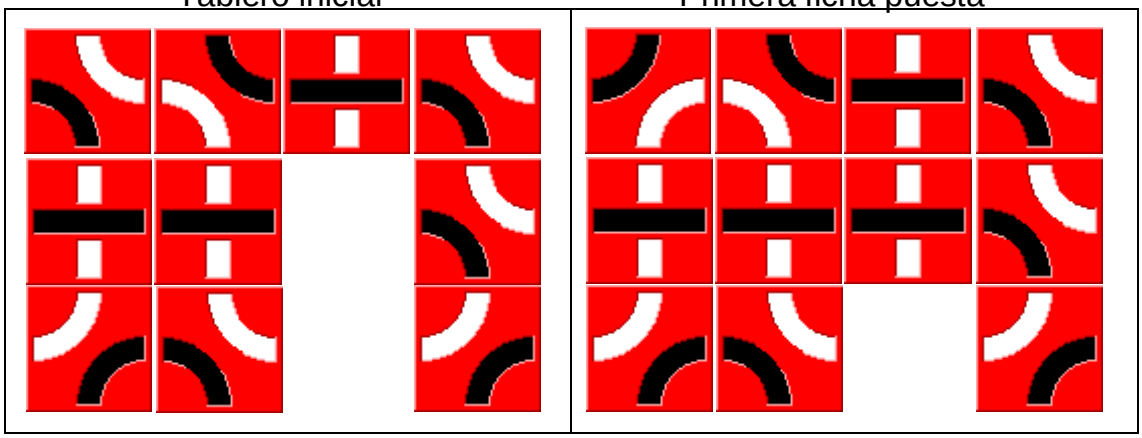
movimiento posible consiste en unir esos caminos. Y esa unión debe hacerse automáticamente. Estos movimientos pueden producir otros movimientos obligatorios, y también pueden llegar a producir movimientos inválidos. En caso de que se produzca en algún momento un movimiento inválido, se cancela toda la secuencia de movimientos, y se tiene que volver al tablero inicial que había antes de poner la ficha que empezó toda la cadena de movimientos automáticos.
Estas mecánicas no son fáciles de ver, así que pondremos dos ejemplos, uno para el caso de un movimiento obligatorio fácil, y otro para uno complicado, que además termina con un movimiento invalido

Tablero inicial Se pone ficha Movimientos obligatorios

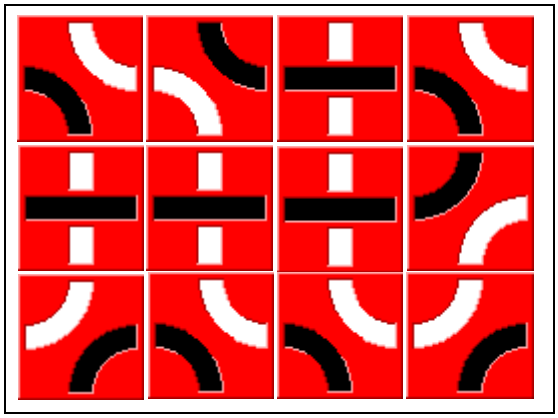
Figura 9. Movimiento obligatorio fácil



Tablero inicial Primera ficha puesta



Sucesivos movimientos obligatorios



Ultimo movimiento obligatorio con tablero final invalido
Figura 10. Movimiento obligatorio difícil

1.3 Objetivos del proyecto

Con las explicaciones que ya hemos dado, podemos concretar los objetivos que esperamos conseguir con este proyecto. La meta final es estudiar las posibilidades que ofrecen los System on Chip, en concreto esta nueva plataforma que une ARM y FPGA. Para probar las capacidades de esta tecnología, se ha decidido utilizar el juego de mesa TRAX, ya que ha sido seleccionado como base para un concurso de diseño en una conferencia internacional sobre FPGAs. Los objetivos concretos a seguir serán:

- Estudiar las necesidades computacionales del juego
- Realizar en software una aplicación capaz de jugar, incluyendo una inteligencia artificial sencilla
- Identificar las funcionalidades con mayor demanda computacional
- Diseñar aceleradores a medida para las funciones seleccionadas
- Estudiar las distintas posibilidades de comunicación
- Analizar los rendimientos, tanto computacionales como energéticos, que se obtienen con las distintas opciones

2. Análisis

El primer punto que debemos abordar es el de analizar las necesidades que plantea el desarrollo de la aplicación de prueba, es decir ¿qué hace falta para jugar al TRAX de manera correcta? Lo primero que se tendrá concretar será como se ha de representar el tablero de juego, seguido de las diferentes funciones para poder avanzar la partida. Por ultimo deberemos elegir que parte de esta funcionalidad puede ser mejor trasladar al hardware, teniendo en cuenta tanto la carga computacional como el coste de las comunicaciones. También habrá que discutir por cuál de las distintas opciones de comunicación nos decantaremos entre las posibilidades que nos brinda el interfaz AXI del que dispone la FPGA

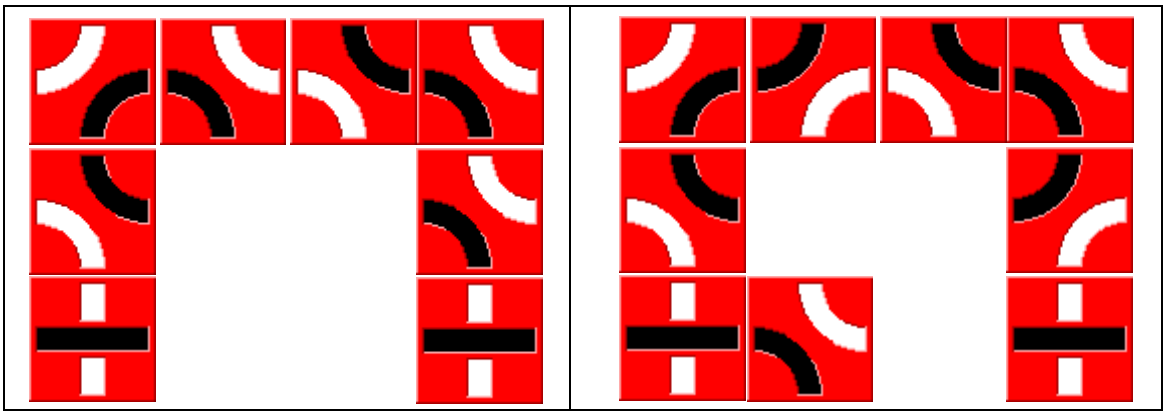
2.1 Representación

Existen varias modalidades del juego. En el juego de mesa convencional se utiliza un tablero de 8X8, pero en las aplicaciones para ordenador frecuentemente no hay límites en el tablero. Mientras se cumpla las reglas de colocación de fichas, el tablero puede crecer hasta que se termine la partida. Este es el caso que se tomó como referencia en la competición del *International Conference on Field-programable Technology* del año 2015. Sin embargo trabajar con un tablero virtualmente infinito no es buen punto de partida para diseñar aceleradores hardware eficientes.
La solución que hemos planteado en este proyecto es diseñar aceleradores versátiles que puedan funcionar con distintos tamaños de tablero. En función del tipo de partida que se quiera jugar, y de los recursos hardware disponibles, se instanciará un acelerador del tamaño deseado. En caso de que el tablero creciese por encima del tamaño del acelerador, o bien se instanciaría un nuevo acelerador, o bien se inhabilitaría, y se utilizaría una versión software equivalente. La primera opción es muy interesante porque las FPGAs permiten cambiar su funcionalidad en tiempo de ejecución, quitando y poniendo aceleradores según sea necesario, sin embargo, gestionar esos cambios es complejo y no ha sido tratado en este proyecto.
A la hora de las fichas simplemente habrá que poder representar los distintos tipos de fichas. Para esto se pueden ver como seis tipos distintos de fichas o tres tipos con dos rotaciones. En ambos casos resulta trivial representarlas tanto en hardware como en software.

2.2 Funciones

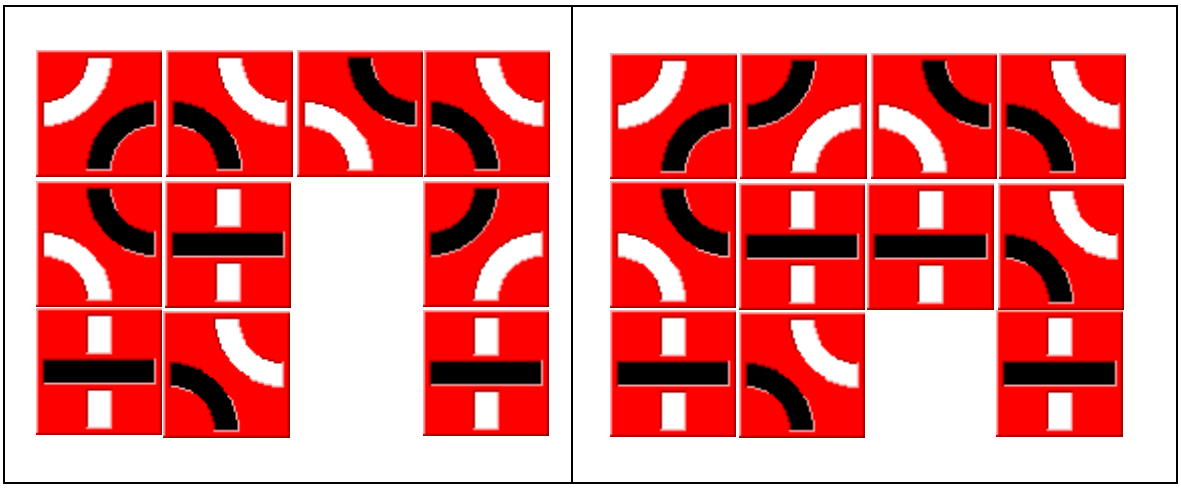
El primer punto que debemos tratar es la interacción con el tablero de juego. Lo primero que necesitaremos será poder determinar que movimientos son válidos. Para ello sabemos que solo se pueden poner fichas en posiciones adyacentes a las que ya se han puesto, respetando el color de dichas fichas. El siguiente paso será poner ficha sobre el tablero, que deberá detectar y realizar los sucesivos movimientos obligatorios. Esta sucesión puede llegar a desembocar en que sea un movimiento inválido, lo cual también deberemos

detectar. Por último deberemos saber si en algún momento se ha llegado al final de la partida, y cuál de los dos jugadores ha ganado.
El siguiente punto a tratar será la Inteligencia Artificial de la que dispondrá para que nuestra aplicación pueda jugar de manera autónoma. Para ello necesitaremos una función que nos permita poder evaluar los tableros que se generen con los movimientos válidos y otra para elegir el movimiento que obtenga la mejor evaluación. Para la función que evaluara tablero, nos basaremos en dos patrones básicos que se encuentran en el juego: las amenazas y las esquinas.
Por un lado las amenazas se producen cuando un jugador está a un turno de formar un ciclo y ganar. Si a nuestro jugador le toca mover y tiene una amenaza la utilizará para ganar la partida, si la tiene el rival, deberá elegir un movimiento que la elimine. Hay amenazas fáciles de ver, como es el caso de la Figura 9 que hemos usado antes para enseñar los movimientos obligatorios. Pero hay otras no tan fáciles de ver, como la de la Figura 11. Las amenazas son importantes ya que te permiten ganar en el siguiente turno, pero tienen el problema de que al crearlas, el rival puede cancelarlas. La forma de ganar es consiguiendo dos amenazas, ya que probablemente el rival sólo se podrá ocupar de una y ganaremos utilizando la otra.

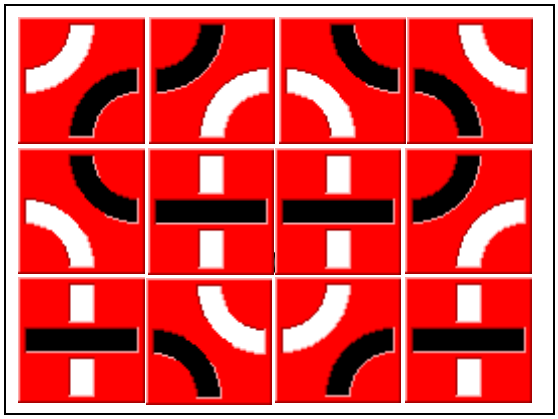


Tablero inicial

Primera ficha puesta



Sucesivos movimientos obligatorios



Ultimo movimiento obligatorio que produce que gane jugador blanco
Figura 11. Amenaza difícil

Por otro lado en este juego se define una esquina como un camino que se puede cerrar en dos turnos. La esquina más sencilla que se puede crear es simplemente con una ficha que no tenga forma de cruz, aunque también existen esquinas más difíciles de ver, como cualquier movimiento que llevase al tablero inicial de la Figura 11. Tener esquinas permite generar amenazas, y la forma habitual de ganar es siendo capaz de convertir dos esquinas en amenazas en un único movimiento. Por tanto es muy importante tratar de producir esquinas que puedas utilizar más adelante, y tratar de que tu rival no las tenga.
Resumiendo en este proyecto se va a desarrollar una aplicación para el juego de TRAX con la siguiente funcionalidad:

- Obtener movimientos legales.
- Poner una ficha en el tablero y saber si es un movimiento valido.
- Comprobar si se ha ganado partida.
- Obtener el número de amenazas.
- Obtener el número de esquinas.
- Valorar un tablero en función de las amenazas y esquinas existentes.
- Elegir un movimiento de entre todos los posibles.

2.3 Asignación de las funciones

Como la plataforma que utilizamos permite ejecutar la funcionalidad tanto en procesadores como en aceleradores hardware un aspecto clave de este proyecto es decidir dónde se va a ejecutar cada función. Para realizar este análisis partiremos de una versión completa en software, y deberemos elegir que funciones queremos que se procesen en la FPGA porque, aunque previsiblemente todas vayan a funcionar mejor en la FPGA, al ejecutarse en un hardware diseñado a su medida, la FPGA tiene una capacidad limitada, y el diseño de los aceleradores supone una complicación adicional del proceso de diseño. Además también hay que tener en cuenta que la comunicación del software al hardware puede tener un gran peso y puede no compensar el ejecutar una función en la FPGA si la ganancia es menor que la penalización generada por las comunicaciones. Por ejemplo si hay que enviar un tablero para hacer un procesamiento sencillo el coste de enviar el tablero puede ser mayor que lo que se gana en el procesamiento. Por lo tanto habrá que elegir

funciones que vayan a compensar el gasto de la comunicación. Al proceso de diseño mixto, tanto de software como de aceleradores hardware, se le llama codiseño, y es un campo en el que la comunidad científica está muy activa desde hace años con diversos congresos y revistas especializadas, lo que demuestra el interés por este tipo de enfoque.
Para analizar las posibilidades del codiseño en esta aplicación se han elegido las siguientes funciones susceptibles de aceleración en la FPGA:

- Obtener movimientos legales.
- Poner ficha en el tablero y saber si es un movimiento valido.
- Comprobar si se ha ganado la partida.
- Obtener el número de amenazas.
- Obtener el número de esquinas.
- Obtener el valor con función de evaluación para un tablero.

Por lo tanto la única función que dejaremos siempre en software es la elección del movimiento. El planteamiento seguido es procesar el tablero en hardware y tomar las decisiones en software. De esta forma obtendremos un diseño modular en el que se pueden probar distintas estrategias de juego sin tener que cambiar el hardware.

3. Diseño

Una vez tenemos analizados los requisitos de la aplicación, pasaremos a diseñar como se ha de realizar la funcionalidad especificada en el anterior apartado. Explicaremos el diseño que se ha realizado en cada función individualmente, y como va a ser en software y en hardware las que vayamos a realizar en la FPGA. En todos los casos se tratará de diseñar una función eficiente en software ya que uno de los objetivos de este trabajo es comparar el rendimiento entre ambas versiones.

3.1 Diseño del tablero

El primer desafío para la implementación de este juego se presenta aquí, ya en teoría el tablero es infinito. Para emular completamente esta característica lo más idóneo sería que el tablero fuese dinámico. Sin embargo, las estructuras dinámicas, aunque tienen la ventaja de que pueden crecer, son mucho más complejas de implementar y mantener, especialmente en hardware, por lo que se ha decidido que se realizara un tablero estático. Eso sí, el diseño será parametrizable para que pueda funcionar con cualquier tamaño. Es decir, el tamaño del tablero es un parámetro del juego que debe definirse al inicio del programa. Después, durante la partida no se podrá cambiar. En realidad, como hemos comentado en la sección anterior, sería posible cambiarlo, pero cambiar la funcionalidad de la FPGA en tiempo de ejecución es un proceso complejo que implica seguir un flujo de diseño distinto. Por lo que no se ha contemplado esa posibilidad en este proyecto.
La mayor ventaja que obtendremos al trabajar con un tablero fijo es que la posición absoluta de las fichas será siempre la misma, en contraste con una representación dinámica, en la que la posición de las fichas cambiaria según fuese creciendo el tablero.
La representación de las fichas es un tema mucho más sencillo, pero muy importante, ya que afectará a todas las funciones.
En primer lugar hay que decidir el número de bits utilizados. Las alternativas más lógicas son representar las fichas con tres o cuatro bits. En principio sólo hay que representar seis tipos de ficha, más la casilla vacía, y por tanto con las ocho posibilidades que ofrecen los tres bits sería suficiente. En lugar de codificar la casilla vacía como una opción más, se puede utilizar un cuarto bit para saber si la casilla está vacía u ocupada. Está opción acelera la gestión del tablero y fue nuestro planteamiento inicial.
Sin embargo, la decisión que se tomo fue una matización sobre la versión de cuatro bits, que consiste en representar no la ficha en sí, sino los caminos que salen de cada lado. Concretamente cada bit nos dice si en un determinado lado hay un camino negro o no. Si el bit vale 1 quiere decir que por ese lado hay camino negro, y si vale cero, no lo hay. En el segundo caso puede haber un camino blanco o no haber ningún camino. Esta codificación es más coherente con la filosofía del juego dónde lo importante no son las fichas sino los caminos.
Una pega de esta elección es que aparentemente no distingue entre caminos blancos y fichas vacías. Pero en realidad la distinción es trivial. Si una posición del tablero está ocupada, tendrá obligatoriamente dos lados con

caminos negros y otros dos con caminos blancos, ya que todas las fichas del juego siguen ese esquema, sino es así es que está vacía.

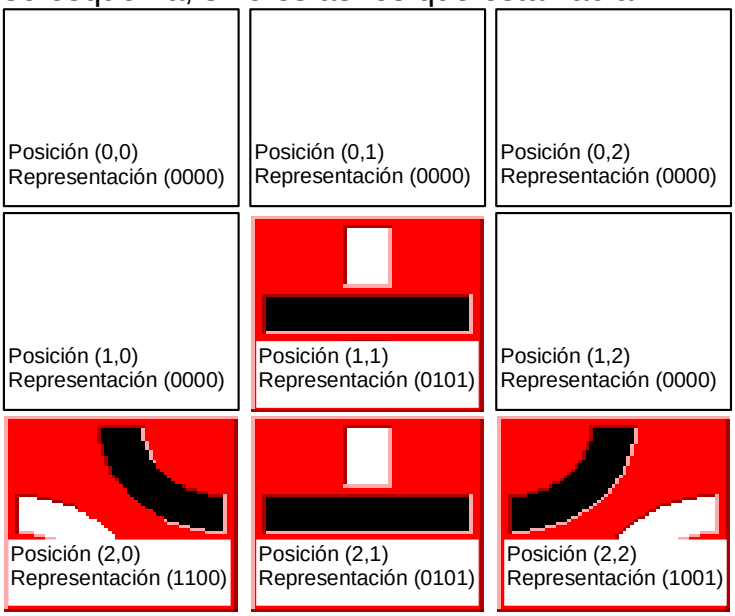


Figura 12. Representación del tablero propuesta

En software esta representación será llevada a cabo mediante una tabla de nodos como la que se muestra en la figura 12. Estos nodos estarán compuestos por tres valores: la fila, la columna y la ficha que contiene. La matriz de nodos será creada al principio con tantos posiciones como se especifiquen. Con esta representación, para saber si una casilla está vacía bastará con compararla con el patrón “0000”.

A la hora de diseñar la representación en hardware se ha creado una matriz, que puede almacenar un tablero del tamaño deseado. En un principio esta matriz contenía la misma información que en software, pero dado que los recursos hardware son limitados, y que según las reglas del juego a una arista del tablero nunca le puede llegar por un lado un camino blanco y por otro un camino negro, se decidió almacenar cada arista de forma única, en lugar de almacenarlas dos veces (en las dos fichas que linden con la arista en cuestión).De esta forma se elimina la información redundante, y se ahorra mucho espacio.

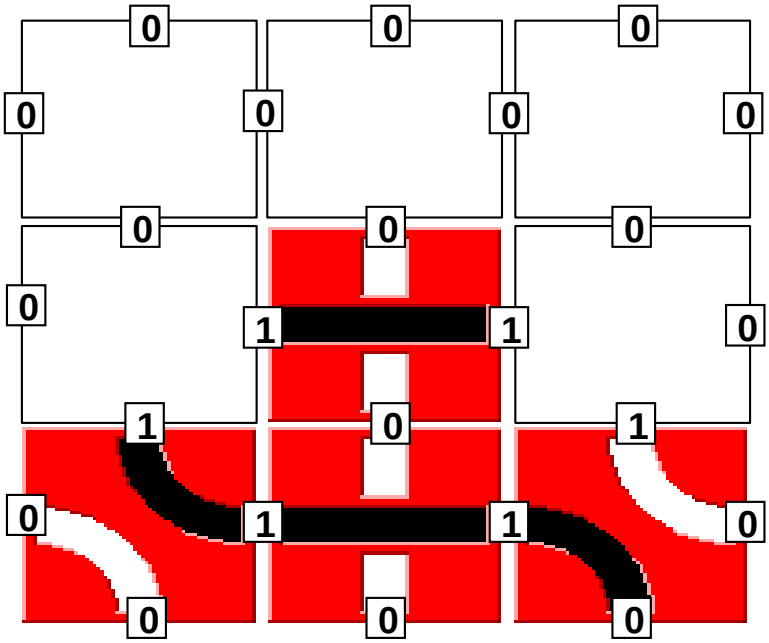


Figura 13. Mejora de representación para hardware

En la Figura 13 se puede ver cómo es la matriz que representa un pequeño tablero de 3x3. Cada una de las cajitas representa una arista que conecta dos casillas, y será un registro de un bit. En este caso, para detectar si una casilla está vacía hay que leer las cuatro aristas que la delimitan. Si hay dos unos y dos ceros estará ocupada, sino estará vacía.

3.2 Diseño de la función "encontrar movimientos legales"

A la hora de identificar los movimientos legales posibles en un tablero, hay que recordar dónde se pueden poner fichas dado un tablero, y cuando un movimiento es legal. Solo se pueden poner fichas cuando hay otra ficha ya puesta en una casilla adyacente. Por ejemplo, en el tablero de la Figura 13, no se podrían poner fichas en la posición (0,0) o la (0,2), pero en el resto se podrían poner fichas. La otra norma a tener en cuenta es que las fichas legales son las que no rompen un camino blanco con uno negro, y viceversa. Tomando como referencia también la figura 13, en la posición (1,0) solo se podría poner un tipo de ficha, la que crea un camino negro desde abajo a la derecha, siendo el resto de fichas invalidas.
En software para buscar estas fichas validas se recorre todo el tablero, y se tiene que buscar posiciones que están vacías y comprobar si alguna de sus casillas adyacentes está ocupada. En dicho caso, probara cuales de las posibles fichas pueden colocarse sin que haya colisión. Pero este proceso sería muy costoso en el caso de tener que recorrer todo el tablero, así que se guardaran las coordenadas donde está ocupado el tablero, así se limitara el espacio de búsqueda a las adyacentes.
En hardware el concepto será muy diferente, ya que se procesará todo el tablero en paralelo. En este caso cada casilla recibirá la siguiente información:

1. Los valores de las aristas que le rodean
2. Un bit indicando si las casillas adyacentes están libres u ocupadas

Procesando estos datos cada casilla puede identificar sus movimientos válidos. Previsiblemente esta función conseguirá un enorme rendimiento en hardware, porque podrá identificar todos los posibles movimientos en paralelo.

3.3 Diseño de la función "actualizar tablero"

El segundo gran desafío que presenta esta aplicación se encuentra a la hora de colocar una ficha. La razón es que al poner una ficha, esta puede desencadenar que se realicen movimientos obligatorios, los cuales pueden llegar a producir, tanto otros movimientos como movimientos inválidos. Estos movimientos obligatorios se dan si en algún momento a una casilla le llegan dos caminos del mismo color. En esos casos es obligatorio añadir la ficha que una esos dos caminos. Es importante remarcar que hay que guardar una lista con las fichas que se han puesto en un movimiento, tanto la que ponemos como las obligatorias, ya que, como veremos más adelante, puede ser muy útil para otras funciones.
En software se pondrá la primera ficha y se comprobara todas las posiciones vacías adyacentes para comprobar si alguna cumple los requisitos para que sea un movimiento obligatorio. Si en alguna posición se cumple, se añadirá

dicha ficha a la lista de fichas colocadas y se repetirá el mismo proceso hasta que no queden fichas por poner. Si en algún momento se intenta poner una ficha, pero resulta que tiene colisiones de caminos, se parara la sucesión de actualizar tablero y se marcara el movimiento como invalido.
La versión hardware también será muy distinta, como en el caso de encontrar movimientos válidos. De nuevo, se gestionará el tablero en paralelo, con un bloque hardware que procese cada posición.
Este bloque recibirá la misma información que el anterior (las aristas que le rodean, y si las casillas adyacentes están vacías u ocupadas). Si el bloque detecta que se debe hacer un movimiento obligatorio lo hará de forma automática.
Para actualizar un movimiento en el tablero en hardware bastará con poner la primera ficha cambiando el valor de sus aristas. A continuación, los bloques hardware realizarán los movimientos obligatorios.

3.4 Diseño de la función "encontrar ciclo"

La siguiente función es muy importante, ya que no sólo sirve para determinar cuándo se gana una partida, sino que sirve también para obtener las amenazas y las esquinas.
A la hora de diseñar esta función una primera aproximación sería recorrer todos los caminos del tablero para ver si alguno es un ciclo cerrado. Sin embargo en software esta solución es muy costosa y poco eficiente ya que se procesa todo el tablero a pesar de que habrá regiones en las que no ha habido ningún cambio. Para mejorar los resultados de la versión software utilizaremos la lista de fichas colocadas en el último movimiento.
El razonamiento que hemos usado es sencillo. Tenemos la seguridad de que en el tablero inicial no había ningún ciclo, ya que en caso contrario se habría terminado la partida. Por tanto, para saber si tras un movimiento se ha generado un ciclo solo hay que recorrer los caminos que se han creado o alargado con las últimas fichas que se han puesto. Esta aproximación reduce mucho los cálculos necesarios. Con este método solamente hay que recorrer dos caminos, uno blanco y otro negro por cada ficha puesta, y diremos que hay un ciclo en el caso de que al recorrer el camino se llegue a la casilla inicial.
Una pequeña posible optimización, sería dejar de buscar ciclos en el caso de haber encontrado ya uno. Pero hay que tener en cuenta que en el caso de que ambos jugadores tengan ciclo, gana el que haya realizado el último movimiento. Además, a la hora de evaluar un tablero, por ejemplo al hacer un árbol de búsqueda de movimientos futuros, es posible que nos interese marcar todos los ciclos, así que recorreremos siempre todos los caminos nuevos para proporcionar toda la información.

3.5 Diseño de la función "evaluar tablero"

A continuación nos tocara hablar sobre la función más grande. La hemos dejado al final, no solo por ser la más grande, sino porque usara todas las funciones que hemos creado hasta ahora. Incluimos aquí también el diseño de encontrar amenazas y esquinas porque en vez de tener una función especial para hallarlas, se contabilizaran al hacer la evaluación. La función

de evaluación recibirá un tablero, un movimiento, y a qué jugador le toca mover y devolverá:

- Si al realizar el movimiento ha ganado un jugador, y cuál ha sido
- Si con ese movimiento se llega a un tablero invalido
- Cuantas amenazas distintas se crean para cada jugador
- Cuantas esquinas distintas se crean para cada jugador

Para que la evaluación sea precisa es importante la puntualización que hacemos de que las amenazas y esquinas son todas distintas, ya que normalmente un camino se puede cerrar de dos maneras distintas, pero debe contarse sólo una vez. La figura 14 muestra un ejemplo en el que dos movimientos dan la victoria al jugador blanco. Sin embargo, los dos generan el mismo ciclo, y son en realidad la misma amenaza.

Figura 14. Dos movimientos para la misma victoria

En este caso no diferenciaremos entre el diseño entre software y hardware, ya que las diferencias se encuentran entre como hemos diseñado los módulos que usaran, que hemos descrito antes.

Para identificar las amenazas y esquinas, usamos su definición: una amenaza es un camino que se puede cerrar en un movimiento, y una esquina un camino que se puede cerrar en dos. Por tanto para detectarlas partimos del tablero inicial y, utilizando un tablero temporal, exploramos todas las posibilidades para los dos próximos movimientos. Si al explorar un posible primer movimiento encontramos un ciclo, habremos identificado una amenaza. Los movimientos que no hayan generado amenaza se exploran en un árbol de búsqueda de nivel dos, es decir para cada uno de ellos se exploran de nuevo todos los movimientos posibles. Si en ese árbol aparece un ciclo, se tratará de una esquina.

Para conseguir que los ciclos se contabilicen sólo una vez, haremos una máscara que se actualizará cada vez que encontremos un nuevo ciclo. A la hora de encontrar un nuevo ciclo, comprobaremos si este se encuentra en nuestra mascara. En el caso de que sea así, no lo contabilizaremos, y en caso contrario lo contabilizaremos y añadiremos a la máscara.

Es muy importante que primero probemos todos los movimientos de primer nivel para detectar las amenazas antes de empezar a buscar las esquinas ya que en caso contrario, por el método que usamos para no repetir ciclos, existe la posibilidad de contabilizar una amenaza como si fuese una esquina. Si dejásemos pasar una amenaza rival al contabilizarla como una esquina, esto nos llevaría a perder al siguiente turno.

Al explorar los movimientos de primer nivel, guardaremos en una lista todos los que deben explorarse en segundo nivel todos los que no han generado ciclos. A esa lista la llamamos lista de empates.
A continuación se presenta el esquema de los pasos que sigue esta función:

1. Actualizar tablero con el nuevo movimiento
2. Comprobar si se ha creado ciclo
    a. En el caso que se ha creado, devolver quién ha ganado y terminar
    b. En el caso contrario, continuar
3. Obtenernos movimientos legales en el nuevo tablero y ponemos la máscara de ciclos encontrados a cero.
4. Exploración nivel 1: Para cada movimiento válido:
    a. Actualizamos un tablero temporal con el movimiento valido
    b. Comprobamos si se ha creado ciclo
        i. Si se ha creado un nuevo ciclo, lo contabilizamos como amenaza y actualizamos la máscara.
        ii. En caso contrario, añadimos el movimiento a lista de empates.
5. Exploración nivel 2: Para cada movimiento en la lista de empate:
    a. Actualizamos el tablero temporal con el movimiento de la lista de empate y obtenernos movimientos legales en este nuevo tablero Para cada movimiento legal:
        i. Actualizamos con el siguiente movimiento legal
        ii. Comprobamos si hay algún ciclo
            1. Si hay ciclos nuevos los contabilizamos como esquinas y actualizamos la máscara.
            2. En caso contrario, no hacemos nada
6. Fin de la evaluación

Como puede verse, usamos todas las funciones explicadas antes. El coste computacional de esta función es muy alto, pero la información que proporciona es la clave para jugar bien a este juego.

3.6 Diseño de la función “selección de movimiento”

Lo último que queda será elegir el movimiento a realizar, función que solo desarrollaremos en software. Para ello, se obtendrán todos los movimientos validos dado un tablero, y se utilizara la función de evaluación para sacar el valor que se le da a cada tablero seleccionando el que mejor puntuación nos proporcione.
Los principales puntos a tener en cuenta son:

- Si un movimiento nos lleva a la victoria, se puede seleccionar el movimiento directamente y terminar la selección.
- Si un movimiento nos lleva a la derrota, no seleccionarlo nunca, ya que con ese perdemos directamente.

- Si un movimiento deja una o más amenazas rivales, se tiene que evitar todo lo que se pueda, ya que es el turno del rival, así que la podría utilizar para ganar
- Si un movimiento genera más de una amenaza nuestra y ninguna del rival lo seleccionaremos ya que en el próximo turno nos dará la victoria.
- En cualquier otro caso elegiremos el movimiento que maximice nuestras esquinas y minimice las del rival.

Sobre estas directrices, y para mejorar el nivel de nuestro jugador, se pueden añadir distintas opciones de exploración comunes en las aplicaciones de Inteligencia Artificial, cómo árboles de búsqueda Alpha/Beta, estrategias de podas, ordenamiento de los movimientos... Esta parte se podría realizar en software utilizando los aceleradores hardware para procesar los tableros generados. Pero no era parte de los objetivos del proyecto, y queda como posible trabajo futuro.

4. Implementación

El último paso que debemos dar es el de implementar todo lo anteriormente especificado, en ambas plataformas, hardware y software. La mayor diferencia entre ambos es que software tendrá que hacer todas las tareas de manera secuencial, mientras que en hardware se podrán hacer en paralelo lo cual permitirá obtener una mejora de prestaciones muy importante.

4.1Implementación del tablero

En primer lugar tendremos que hablar de cómo se ha implementado la representación de las fichas del tablero, ya que será necesario para poder implementar todas las funciones.
En software simplemente se trata de una tabla de dos dimensiones de tipo de dato *unsigned char*. Este tipo de dato contiene 8 bits, y necesitamos 4 para representar una ficha con la codificación que hemos especificado anteriormente. Se valoró usar representación con aristas, o usar cada *unsigned char* para almacenar dos posiciones, para aprovechar mejor el espacio, pero al probar la segunda opción, se observó que manipular los bits empeoraba el tiempo de ejecución, así que se decidió que cada *unsigned char* almacenara solo una ficha de 4 bits.
El tamaño de esta tabla de dos dimensiones se puede cambiar simplemente con un *define* para modificar todo la implementación.
Para hardware hemos creado un módulo registro, que permitirá almacenar un valor en el tiempo. La cantidad de bits que puede almacenar es configurable a la hora de crear un nuevo registro, ya que aunque para la implementación del tablero por aristas solo necesitaremos almacenar un bit, es un módulo que se va a poder utilizar en muchos lugares. Las entradas de este módulo será un bit que indicara cuando hay que actualizar el valor y otra que será un array de bits tan grande como le hayamos especificado, que indica el valor a actualizar, aparte de la entrada para la señal de reset y la del reloj, que serán típicas en los módulos hardware. La salida será otro array de bits que indicara el valor actual. A la hora de crear el tablero tendremos un doble bucle, para generar filas y columnas, que creara tantos registros como sean necesarios. El número de filas y columnas será definido de la misma forma que en software, para poder ser cambiado fácilmente.

Figura 15. Entradas y salidas del módulo hardware Registro

4.2Implementación de la función “encontrar movimientos legales”

En software la función para encontrar los movimientos legales es muy proceso iterativo sencillo. Se recorren todas las posiciones del tablero, hasta que encontremos una que no tenga una ficha puesta, pero que alguna de sus fichas adyacentes sí que esté ocupada. Una vez llegamos a esta situación, paramos a evaluar que fichas son válidas en dicha posición. Para ello tenemos una función que, dándole un tablero, una ficha y una posición, nos devuelve si produce colisión, por lo que no se podría poner. Usamos esta función para probar cuales de los seis tipos distintos de fichas se pueden poner y las anotamos en caso afirmativo en una lista estática de nodos, por lo cual anotaremos la fila, la columna y el tipo de ficha. Cuando terminamos, anotamos la última posición de la lista estática con una ficha inválida, para denotar que es la última posición. Por lo tanto la única entrada es el tablero del que sacar los movimientos legales, y la única salida la tabla con nodos de movimientos legales.
Para mejorar el rendimiento en lugar de evaluar todas las posiciones del tablero se incluyó una máscara que marcaba las posiciones que debían explorarse (las adyacentes a posiciones ocupadas).
En hardware no tendremos estos problemas de rendimiento, ya que todas las casillas se procesan en paralelo en un ciclo. Cada casilla incluye un sencillo módulo hardware que calcula qué fichas se pueden poner en esa posición. Como se ve en la Figura 16 estos módulos tienen dos entradas: las aristas que componen una determinada casilla (arriba_in, abajo_in, izquierda_in, derecha_in) y la información de si están ocupadas o no las fichas adyacentes (nul_{arriba, abajo, izq, der}_in). En cuanto a las salidas, los movimientos válidos se indican en una señal de seis bits. Además se informa a los módulos adyacentes de si la casilla está vacía (nul_out).
Con lo cual, pasado un ciclo, tendremos una matriz de salida en la que cada posición estará compuesta de un vector de seis bits, indicando que fichas se pueden poner. Para verificar el funcionamiento de este módulo se ha implementado en software una función equivalente.

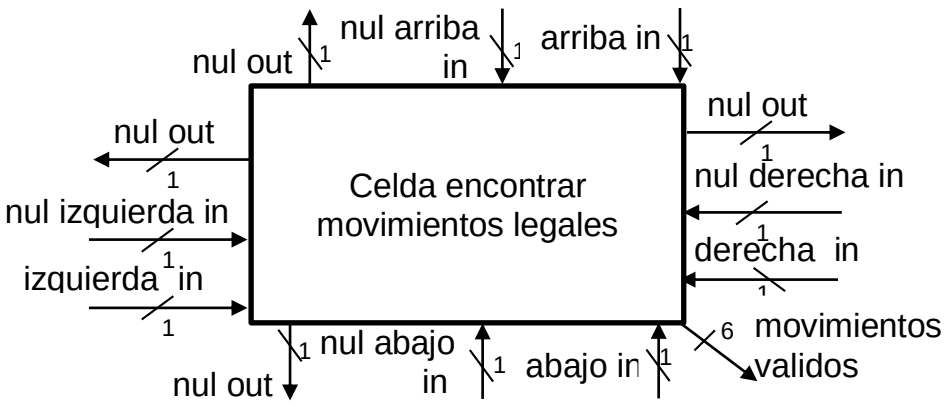


Figura 16. Entradas y salidas de las celdas de encontrar movimientos legales

La matriz de salida es una estructura muy grande y difícil de manejar. Para utilizar la información obtenida de forma eficiente se ha diseñado un módulo auxiliar que procesa la matriz y almacena r la información útil en una memoria. Para ello usaremos codificadores con prioridad. Estos codificadores tienen una entrada con un vector de 2^n bits y una salida de n bits, que corresponde a la representación binaria de la entrada activa con mayor prioridad. Esto nos servirá para poder reconocer de forma secuencial las casillas con información útil y almacenar esa información en una memoria en el menor tiempo posible. Este proceso se gestionara mediante una máquina de estados, cuyos estados serán:

- idle: será el estado de espera, hasta que le llegue la señal de empezar
- sacarParaRAM: en este estado comprobara si ya se ha almacenado la información de todas las casillas con movimientos válidos, en cuyo caso se volverá al estado idle. En caso contrario, identificará, por medio del codificador de prioridad, la siguiente casilla a procesar y pasaremos al estado siguiente.
- escribirenRAM: este estado se escribe en la memoria el nuevo valor, se incrementa la siguiente dirección a escribir en la memoria y se enmascarara el último valor insertado, para no volverlo a insertar.

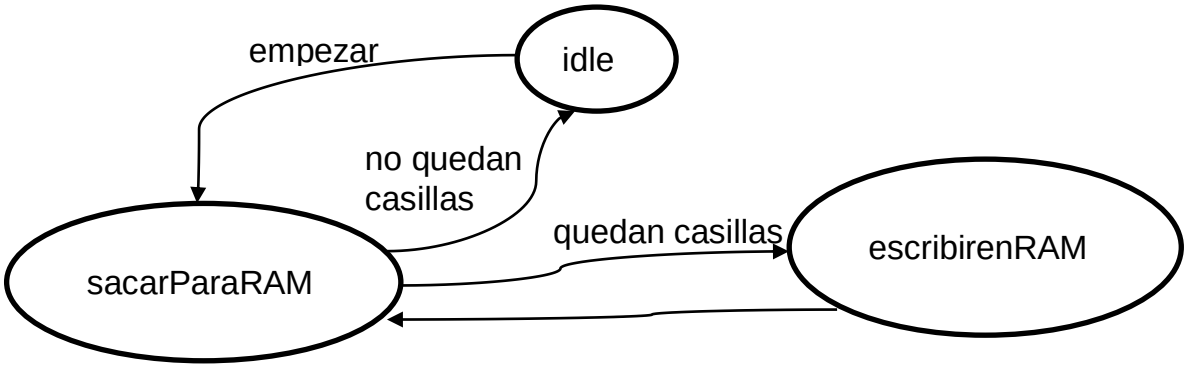


Figura 17. Máquina de estados para pasar de tablero a memoria

El método para averiguar la siguiente posición a meter en la memoria es el siguiente. Reducimos con una puerta or los movimientos válidos, ya que solo nos interesa saber si hay o no movimientos en una posición. Para cada fila, metemos estos valores en un codificador, que usaremos para saber en qué columna se encuentra el primer valor en cada una de las filas. También haremos una or por fila, y todas estas or llegan a un segundo codificador que nos indicará cuál es la primera fila que tiene valor. Con esta fila elegiremos cuál de los codificadores de columna tenemos que consultar, y teniendo ya la fila y columna, consultamos el tablero original para encontrar los valores, siendo estos tres valores los necesarios para la memoria.

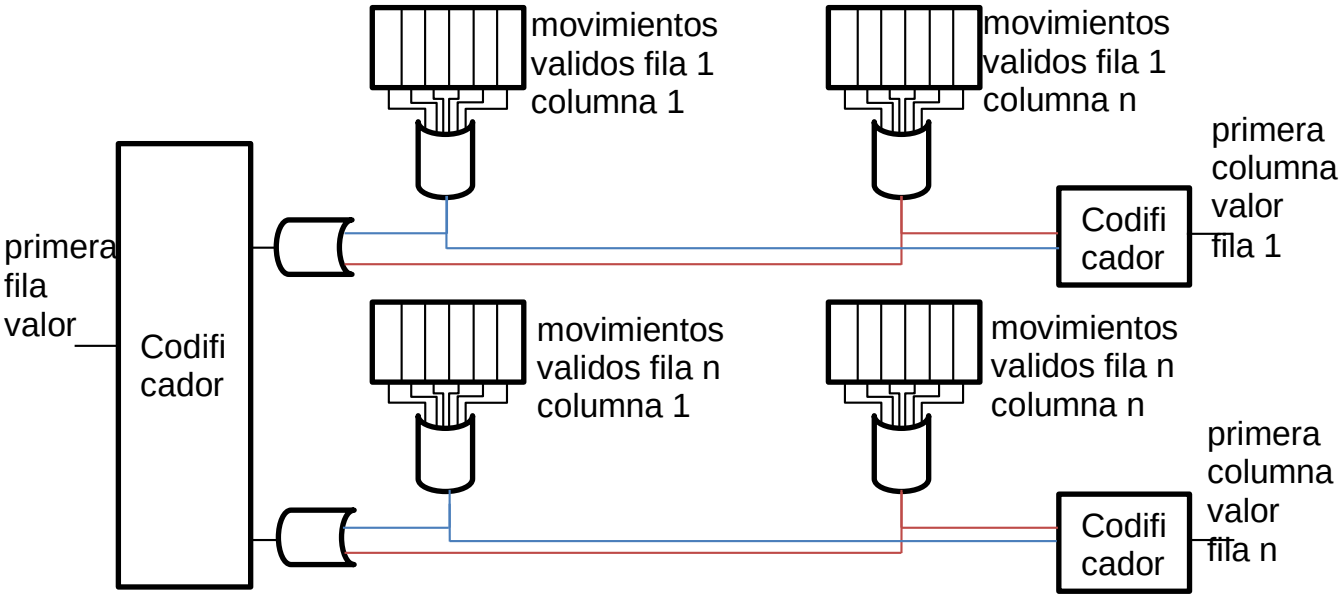


Figura 18. Funcionamiento para la búsqueda de fila y columna

4.3Implementación de la función "actualizar tablero"

El último paso que necesitamos antes de poder pasar a la función de evaluar tableros es el de poder saber cuándo se ha ganado una partida, ya que se necesitara para poder identificar las amenazas y las esquinas.
En software se han realizado varias versiones para esta función, la más eficiente utiliza una lista de fichas nuevas y sólo comprueba los caminos que se han creado o alargado con estas nuevas fichas Para cada camino se utiliza una función que recorre los caminos hasta llegar a una posición vacía, o a la posición inicial, en cuyo caso se marcará que se ha encontrado un ciclo. Una se vez encuentra un ciclo, es hora de comprobar que no lo tenemos almacenado en la máscara de ciclos ya recorridos. El módulo hardware fue desarrollado por un compañero de la asignatura de Laboratorio de Sistemas Empotrados (Alberto Millán) por lo que no se incluye su descripción. Eso sí, fue necesario modificar el módulo inicial para integrarlo con el resto de módulos añadiendo una salida para crear la máscara de ciclos, y adaptándolo a la representación final del tablero. Estos cambios se realizaron en la capa más externa, así que no hizo falta tocar el funcionamiento del módulo, solo la interacción con el exterior.

4.4Implementación de la función "evaluar tablero"

Esta es la función más completa del trabajo, ya que se nutre del resto de funciones. En este caso lo más sencillo va a ser explicar las máquinas de estados que guían esta función, sin distinguir entre software y hardware, ya que la mayor diferencia está en cómo se han implementado las funciones usadas. Para reducir la extensión de la memoria en este apartado sólo describiremos la máquina de estados de más alto nivel. Es una máquina de

estados jerárquica, en la que algunos de sus estados son en realidad otras máquinas de estado. La descripción de esas otras máquinas está en el anexo A.

La máquina de estados principal se compone de seis estados:

- Idle: estado de espera, avanza cuando recibe la señal de inicio
- ActualizarTablero: actualiza el tablero con la ficha que se quiere probar
- ComprobarGanador: Lo primero que se prueba es si este movimiento lleva directamente a final de la partida, en cuyo caso se devolvería simplemente que termina la partida, y quien ha ganado. En caso contrario, se continúa el proceso.
- IdentificarLegalMoves: se identifican los movimientos legales de este nuevo tablero para probarlos todos.
- ProbarMovLegal: la siguiente máquina de estados, en la que se contabilizaran las amenazas y esquinas.

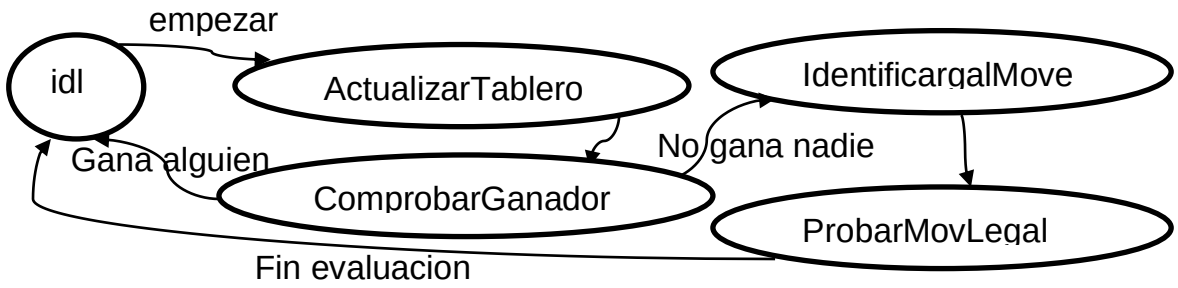


Figura 18. Maquina estados evaluar tablero

La implementación hardware inicialmente dio problemas al utilizar demasiados recursos. Sin embargo, la causa era que se utilizaban módulos idénticos en distintas funciones como se muestra en la Figura 19. En la Figura 19 se puede ver como estaban repetidos los módulos, y el porcentaje de ocupación en la FPGA por cada uno.

	ActualizarTablero (4,5%)	MovimientosValidos (9,5%)	EncontrarCiclos (8%)
EvaluarTablero	✓	✓	✓
Probar Movimientos Legales	✓		✓
EvaluarTablero Simplificado		✓	
Probar Movimientos Legales Simpificado	✓		✓

Figura 19. Replicación de modulos

Para arreglar este problema, se creó solo una instancia de cada módulo, que se reutilizaba en distintos componentes. Para ello se añadieron multiplexores para seleccionador las entradas. Esto produjo que se liberasen alrededor del 30% de los recursos de la FPGA.

4.5 Implementación de la función "selección de movimiento"

Toca analizar la función que usara la evaluación del tablero. Esta función es la única que solo se realizara en software, ya que lo que definimos con esta función es la estrategia que usamos a la hora de jugar, y es interesante poder explorar varias. Como prueba hemos diseñado una muy sencilla, la cual pretendemos que pueda cambiarse fácilmente.
La dinámica de la selección del movimiento será la siguiente:

1. Actualizar el tablero con la última ficha del rival.
2. Obtener los movimientos validos de este nuevo tablero
3. Evaluar estos movimientos con la función de evaluar tablero
4. Elegir el movimiento con mayor puntuación.

La puntuación es 0 si se pierde, 255 si se gana y una combinación lineal del número de amenazas y esquinas en el resto de casos. Es una heurística muy sencilla ya que no se hace árbol de búsqueda, pero es suficiente para comprobar que el diseño es capaz de jugar de forma autónoma que era nuestro objetivo.

4.6 Implementación de la comunicación

El último punto que queda es el de la comunicación entre el hardware y el software. Para la implementación del sistema de comunicación se han priorizado dos aspectos:

- Minimizar las comunicaciones para que afecten lo menos posible al rendimiento.
- Facilidad de uso para el programador de software.

Para minimizar las comunicaciones lo que se ha hecho es que hardware mantenga algunas estructuras actualizadas sin intervención del software. Por ejemplo el tablero del juego o a qué jugador le toca mover.
La facilidad de uso, se consigue haciendo funciones software que se ocupen de todos los detalles de bajo nivel de las comunicaciones con la FPGA.
Para comunicarnos con la FPGA se ha diseñado un sencillo interfaz con cuatro registros de 32 bits mapeados en memoria. El software puede leer y escribir en estos registros como si correspondiesen a una dirección cualquiera de memoria.
El primer registro se utiliza para mandar órdenes a la FPGA, los dos siguientes para recibir resultados y el último para consultar un contador interno de la FPGA que se utiliza para hacer medidas de rendimiento.

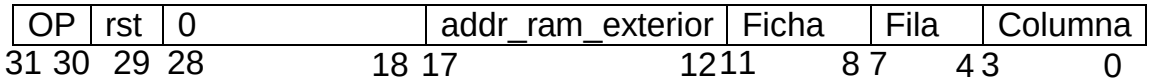


Figura 20. Registro orden (tamaño tablero 16, bits RAM 6)

El registro de orden (figura 20) tendrá dos bits que marcaran la operación. Hay tres tipos de operación con sus respectivos argumentos:

- OP=01 Actualiza el tablero con la ficha especificada, en la fila y columnas especificadas. También obtiene los nuevos movimientos válidos y los mete en una RAM.
- OP=10 Consulta la dirección de RAM especificada
- OP=11 Evalúa el tablero con la ficha especificada, en la fila y columnas especificadas.

La posición de los argumentos puede cambiar según los valores que se pongan en la aplicación de TRAX de la FPGA, aunque lo común sea un tamaño de tablero de dieciséis y seis bits para la dirección de la memoria RAM. La señal de reset sirve para resetear totalmente la aplicación de TRAX, borrando incluso el tablero.
Hay dos registros de resultado, para las distintas operaciones. En el registro uno (Fig 21) se escriben los resultados de la operación 01 y 10, mientras que en el registro dos (Fig 22) se escribe el resultado de la operación 11.

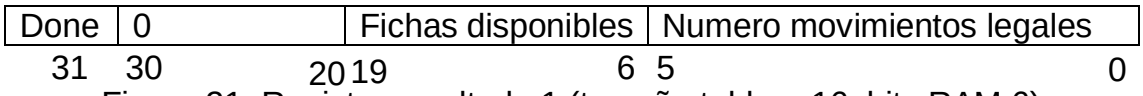


Figura 21. Registro resultado 1 (tamaño tablero 16, bits RAM 6)

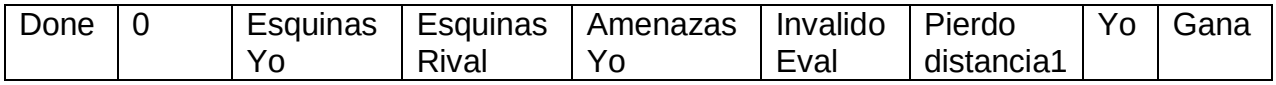


Figura 22. Registro resultado 2 (bits amenazas 3, bits esquinas 6)

En ambos registros el bit 31 a que la operación ordenada ha terminado. El resto de resultados es distinto. Para la operación 01, en el registro uno se indica cuantas casillas contienen movimientos legales generados por el movimiento que acabamos de hacer. Para la operación 10, lo que se obtiene son los tipos de fichas que son válidos, junto a en qué posición se encuentran. Por ultimo en el registro 2 obtenemos toda la información que nos proporciona el módulo de evaluar tablero.
Cabe mencionar que también se dispone de un cuarto registro, que está conectado a un contador que se incrementa cada ciclo, teniendo en cuenta que el reloj de la FPGA va a 100MHz. Este registro se ha utilizado principalmente para la evaluación de los resultados.
Para acceder a estos registros se utilizan las funciones Trax_helper-mWriteReg y mReadReg (Figura 23 y 24) que permiten escribir y leer en el registro deseado. .

```
TRAX_HELPER_mWriteReg(direccion, dato_scribir);
```

Figura 23. Función para escribir en un registro

```
valorVolcado=TRAX_HELPER_mReadReg(direccion);
```

Figura 24. Función para leer de un registro

5. Verificación

La verificación se realizó en varias fases.

En la primera fase se comprueba cada módulo del diseño por separado. Para ello se diseñan casos de prueba y en primer lugar se comprueban que las salidas son correctas en la versión software. Después se comprueba que el hardware devuelve exactamente el mismo resultado. Esto se comprueba en primer lugar utilizando un simulador, y posteriormente en la placa.

En la segunda fase se realizaron las pruebas de integración. Una vez que hemos terminado de desarrollar todos los modelos de la aplicación, de nuevo hay que comprobar que funcionan correctamente. Para ello se diseñaron bancos de prueba y se comprobó que tanto las versiones hardware como software identificaban los mismos movimientos posible y les asignaban la misma puntuación.

Finalmente se comprobó que el diseño podía jugar partidas completas de forma autónoma. El primer paso para esta verificación, fue probar que las funciones en software pudiesen jugar varias partidas contra un jugador humano. Después se realizó un banco de pruebas en el que nuestra aplicación jugaba contra sí misma. Una vez comprobado que la versión software funcionaba correctamente, se fueron incluyendo los módulos hardware verificando que la aplicación devolvía exactamente el mismo resultado.

6. Resultados

Para medir el rendimiento de las distintas versiones desarrolladas hemos usado un contador, que se encuentra dentro de la FPGA, que nos devuelve el número de ciclos transcurridos. Como medida del rendimiento utilizaremos el tiempo que cuesta elegir un movimiento, así que leeremos el valor del contador antes y después de ejecutar la función, y al restarlo obtendremos cuantos ciclos le ha costado ejecutarse, sabiendo que cada ciclo son diez nanosegundos.

La tabla 1 muestra el rendimiento obtenido tanto utilizando aceleradores HW como en nuestra mejor versión software. Es interesante explicar que los tiempos de ejecución varían mucho según la situación del tablero, y según cuanto tenga que recorrer en el árbol de posibilidades a la hora de evaluar. Por esa razón en la tabla mostramos todos los movimientos de una partida completa y en la última fila los resultados medios.

	HW	**SW**	
Color	**ms**	**ms**	**SW/HW**
Blanco	0.41	8.55	20.98
Negro	2.43	4.46	1.84
Blanco	2.43	21.94	9.03
Negro	3.19	53.82	16.86
Blanco	6.03	96.80	16.06
Negro	10.83	186.29	17.21
Blanco	3.15	33.16	10.53
Negro	30.12	321.84	10.69
Blanco	5.43	53.96	9.93
Negro	36.75	415.31	11.30
Blanco	2.34	6.36	2.71
Negro	3.95	48.49	12.29
Media	8.92	104.25	11.69

Tabla 1. Rendimiento hardware

Como se puede ver en la tabla la versión que utiliza los aceleradores hardware proporciona una ganancia muy importante, ejecutándose aproximadamente 11.5 veces más rápido.

Para valorar estos resultados es importante aclarar que la versión software ha sido objeto de diversas optimizaciones de cara a realizar una comparación lo más justa posible. De hecho la versión software final es diez veces más rápida que la versión inicial. Esta ganancia se consiguió mejorando los algoritmos de procesamiento y las estructuras de datos utilizadas. Los cambios principales, que se han mencionado ya en las secciones previas, se resumen a continuación:

- Explorar sólo los caminos creados o modificados en el último movimiento
- Gestionar una máscara que nos diga que parte del tablero se debe explorar
- Utilizar una representación del tablero y las fichas más fácil de procesar

- Utilizar estructuras estáticas en lugar de memoria dinámica

Además se han explorado distintas opciones de compilación. Al final el mejor resultado en rendimiento se obtuvo con –O2, ya que con ella conseguimos el doble de velocidad que sin mejoras. Con las opciones de –O1 y –O3 se aumentaba un poco más el rendimiento, aunque no tanto como en el caso previo.
Los resultados de la Tabla 1 se han obtenido para un tablero de 16x16. Es interesante explicar que tanto en los aceleradores como en el software el tiempo de ejecución no depende mucho del tamaño del tablero, sino sólo de los caminos existentes. La razón es que en los aceleradores el tablero se procesa totalmente en paralelo, mientras que en el software, gracias a las máscaras que utilizamos, sólo se procesa la parte del tablero que se está utilizando.
También se ha medido el consumo que tiene la plataforma cuando está ejecutando la versión puramente software y al usar la versión con parte software, y parte en la FPGA. La diferencia en potencia no es muy significativa, pero esto se debe a que sigue ejecutando gran parte en software. Esta diferencia se hace más notoria a la hora de calcular la energía en una partida como la que hemos analizado en la figura 25, lo que nos da que consume casi 12.5 veces menos energía en una partida estándar.

	potencia(W)
Codiseño	4.49
Software	4.57
SW/HW	1.0173

	energia(mJ)
Codiseño	287.45
Software	3566.83
SW/HW	12.4084

Tablas 2. Rendimiento energético

7. Conclusiones

A la vista de los resultados, es evidente que vale la pena tener en cuenta el codiseño software-hardware a la hora de optimizar una aplicación, ya que proporciona un gran abanico de mejoras. Aunque también es necesario mencionar que para aprovechar totalmente las ventajas que nos presentan estas plataformas hace falta cambiar el método de pensar a la hora de diseñar, ya que es necesario que sea lo más concurrente posible. Un aspecto a interesante de los resultados es que, aunque en un principio los resultados obtenidos al utiliza los aceleradores hardware eran dos órdenes de magnitud mejores que la versión equivalente en software, tras dedicar tiempo a la mejora del software se consiguió reducir bastante la diferencia de rendimiento. La conclusión que se extrae es que antes de plantear una solución basada en el codiseño, es importante asegurarse de que realmente es necesaria, invirtiendo tiempo en mejorar el software. Sólo en el caso de que no seamos capaces de que nuestro software proporcione el rendimiento o la eficiencia energética deseada es cuando habría que estudiar las posibilidades del codiseño.

Otro punto a tratar es que era la primera vez que trabajaba en diseño hardware, así que al principio me costó un poco manejarme con las herramientas que había que usar, la forma de programar o depurar. Esto se puede observar en el diagrama de Gantt presentado en el anexo B, en el que la primera función desarrollada, que era relativamente sencilla, me llevó casi tanto tiempo como la última, que es mucho más compleja. La razón es que en esa primera función está incluido el tiempo de familiarizarme con el sistema.

8. Trabajo a futuro

Existen muchas posibilidades para continuar este trabajo. El principal objetivo sería el de añadir distintas mejoras para que pudiese participar en el *International Conference on Field-Programable Technology* del año 2016. El mayor punto a mejorar es el de la Inteligencia artificial (IA), ya que la actual está hecha con el único propósito de conseguir un jugador autónomo con el que probar rendimientos al jugar una partida completa. Para hacer una IA potente, habría que explorar distintas estrategias de exploración de los movimientos futuros. Esta parte se podría hacer inicialmente en software utilizando los aceleradores para evaluar los distintos tableros que se desee analizar

Otro tema a mejorar es el tamaño del tablero que manejamos, ya que aunque la versión software puede ampliarse más allá del 16x16, en hardware no hay suficientes recursos para implementar un tablero de 32x32. Por lo tanto hay que explorar distintas estrategias para que, o bien se puedan reducir los recursos necesarios para implementar el tablero, o bien aprovechamos al máximo el tablero, por ejemplo haciendo ajustes automáticos para centrar las fichas. En todo caso el tamaño implementado parece suficiente para hacer la práctica totalidad de las partidas (el tablero original es de 8x8). Esta mejora tendría menos prioridad que la anterior, ya que por ahora no hemos conseguido encontrar situaciones reales que demanden un tablero mayor.

9. Anexos

A. Explicación detallada de máquina de estados Evaluar tablero

Como se ha descrito antes la máquina de estados principal se compone de seis estados:

- Idle: estado de espera, avanza cuando recibe la señal de inicio
- ActualizarTablero: actualiza el tablero con la ficha que se quiere probar
- ComprobarGanador: Lo primero que se prueba es si este movimiento lleva directamente a final de la partida, en cuyo caso se devolvería simplemente que termina la partida, y quien ha ganado. En caso contrario, se continúa el proceso.
- IdentificarLegalMoves: se identifican los movimientos legales de este nuevo tablero para probarlos todos.
- ProbarMovLegal: la siguiente máquina de estados, en la que se contabilizaran las amenazas y esquinas.

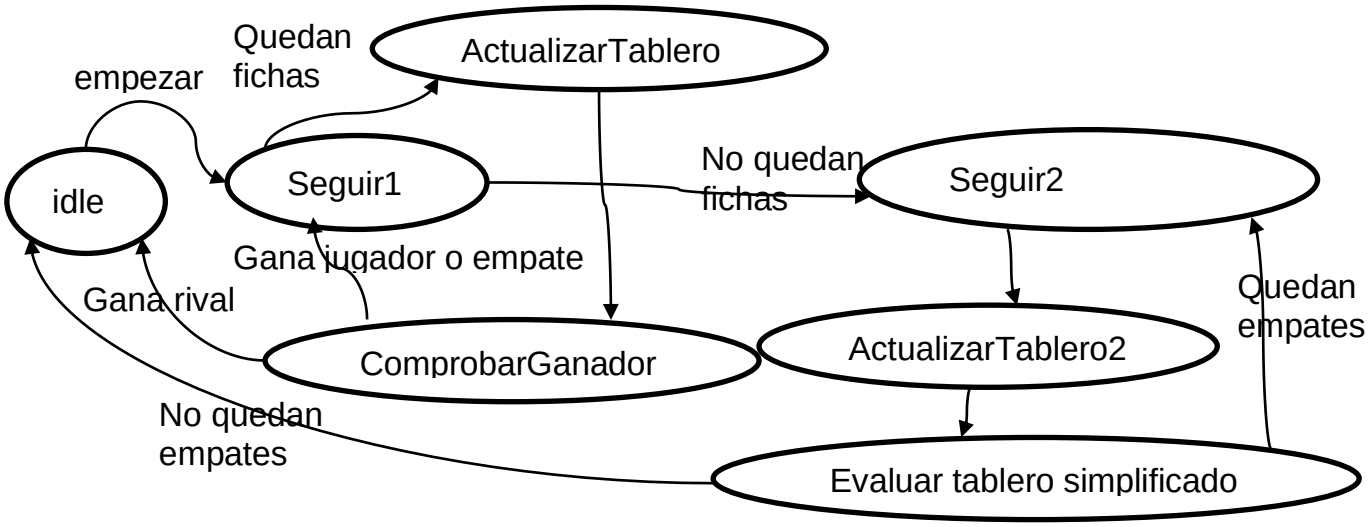


Figura 1. Maquina estados probar movimientos legales

Algunos de los nodos presentados en esta figura son en a su vez máquinas de estado. A continuación las describiremos en detalle.

Ahora que hemos pasado la máquina de estados más complicada, toca una de las más fáciles. Lo único que precisa es sacar los movimientos legales, ya que la función anterior se ocupa de actualizar el tablero, y al estar evaluando empates no hace falta comprobar quien ha ganado, porque ya sabemos que es un empate. Por lo tanto, los estados son:

- Idle: estado de espera, avanza cuando recibe la señal de inicio
- SacarLegalMoves: se sacan los movimientos legales de este nuevo tablero para probarlos todos.
- ProbarMovLegalSimplificado: la siguiente máquina de estados, en la que se contabilizaran las amenazas y esquinas

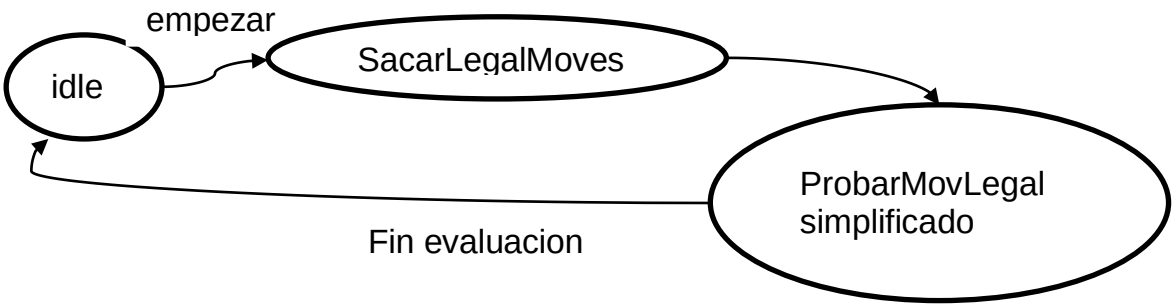


Figura 2. Maquina estados evaluar tablero simplificado

El estado Evaluar tablero simplificado es de nuevo otra máquina de estado. Esta máquina es más sencilla ya que lo único que precisa es sacar los movimientos legales, ya que la función anterior se ocupa de actualizar el tablero, y al estar evaluando empates no hace falta comprobar quien ha ganado, porque ya sabemos que es un empate. Por lo tanto, los estados son:

- Idle: estado de espera, avanza cuando recibe la señal de inicio
- Seguir: este estado comprueba si hay más movimientos a evaluar, en caso afirmativo, selecciona el siguiente movimiento, y en caso contrario termina ya la evaluación.
- ActualizarTablero: actualiza el tablero auxiliar con el nuevo movimiento
- ComprobarGanador: comprueba si alguno de los dos jugadores gana la partida con este movimiento. En caso de que se gane, se aumenta el número de esquinas rivales o del jugador, según quien haya ganado. En caso de empate, no hace nada

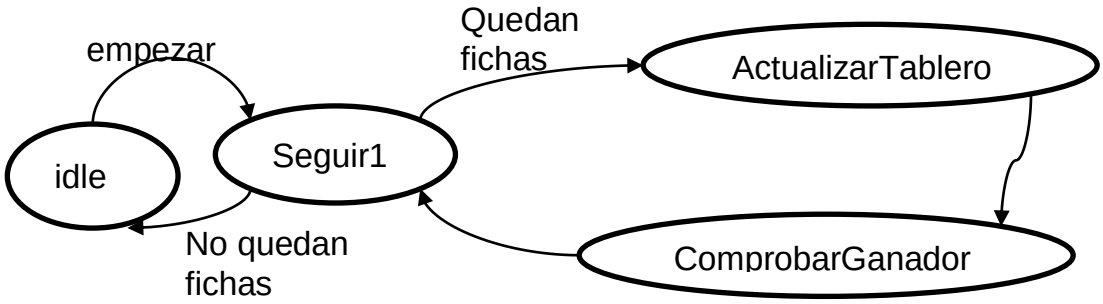


Figura 3. Maquina estados probar movimientos legales simplificado

B. Diagrama de Gantt

	Task Name	Work	Start	End
	Trabajo fin de grado	**544 hrs**	**04/03/2015**	**28/01/2016**
	Analisis sistema	48 hrs	04/03/2015	08/04/2015
	Movimientos legales	**112 hrs**	**10/04/2015**	**08/06/2015**
	Representación aristas	**48 hrs**	**09/06/2015**	**07/07/2015**
	Actualizar tablero	**88 hrs**	**08/07/2015**	**20/08/2015**
	Encontrar ciclos	**16 hrs**	**21/08/2015**	**28/08/2015**
	Evaluar tablero	**128 hrs**	**31/08/2015**	**09/11/2015**
	Memoria	104 hrs	10/11/2015	28/01/2016