Fecha de entrega 11 de Mayo de 2024.

Implementación de Conway's game of life en FPGA

CRISTOBAL ROSALES[1], JOAQUÍN LÓPEZ[1]
[1]Pontificia Universidad Católica de Chile (e-mail: crosalessoto1@uc.cl, jlpez@uc.cl)

Este proyecto ha sido desarrollado bajo el curso IEE2463: Sistemas Electrónicos Programables.

ABSTRACT El juego de la vida de Conway es un juego sin jugadores, que se puede representar como una matriz de leds que pueden estar encendidos o apagados, esto permite representar células que pueden estar "vivas" o "muertas", y cuya "muerte" o "nacimiento" se basa en reglas que permiten evaluar el mapa y realizar una iteración del juego para llegar a un estado futuro. La implementación realizada en FPGA se basa en el lenguaje VHDL, y utiliza el diseño digital mediante lógica secuencial y combinacional para hacer el cálculo del estado futuro en base al estado actual. Se hace uso del bloque *AXI Traffic Generator* para la inicialización de parámetros dentro del diseño, y para la carga del mapa inicial para la ejecución del programa, se hace uso del bloque *Integrated Logic Analizer* y *Virtual Input Output* para poder enviar y visualizar señales de los bloques realizados para el proyecto. Al final de la implementación se logró configurar dos bloques mediante el uso de ATGs, y se logró simular una secuencia completa de el juego sin errores en el calculo de cada estado.

INDEX TERMS Conway's game of life, FPGA, VHDL, Diseño digital.

I. ARQUITECTURA DE HARDWARE (1 PUNTO)

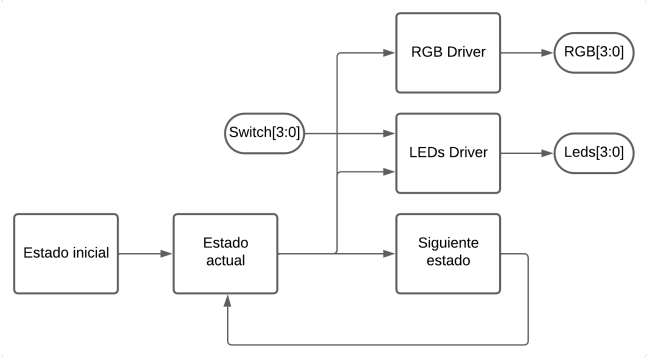

FIGURE 1: Diagrama de bloques de la implementación en FPGA

EL juego de la vida de Conway (Gardner, 1970) es un juego sin jugadores que se basa en una grilla de células donde cada una puede estar "viva" o "muerta" en un determinado momento. El juego corre en tiempo discreto donde cada vez que se ejecuta una iteración se calcula si cada célula va a "nacer", "morir" o "sobrevivir". Este cálculo se realiza en todas las células al mismo tiempo y se obtiene a partir de la suma de células vivas adyacentes. Las reglas que rigen el juego son: 1) Si una célula está viva y tiene dos o tres vecinas vivas, sobrevive. 2) Si una célula está muerta y tiene tres vecinas vivas, nace. 3) Si una célula está viva y tiene más de tres o menos de dos vecinas vivas, muere.

La implementación realizada consta de un mapa de 8x8 celdas, y tiene la particularidad de que los bordes se tocan, es decir, los costados derecho e izquierdo, y arriba y abajo limitan entre sí. Se realizo la implementación en una FPGA, se hizo uso de lógica secuencial y combinacional para la ejecución del programa, y se hizo uso de los LEDs integrados en la tarjeta para la visualización de este. El diseño funcional por bloques se puede ver en la figura 1, esta muestra los siguientes bloques:

- Estado inicial: Fue implementado utilizando un *ATG Test Mode* que inicializa un *ATG Advanced Mode*, este último manda por AXI el mapa inicial que se usara para ejecutar. Y finalmente un IP-Core esclavo que recibe por AXI y muestra como salida de 64 bits el estado inicial.
- Estado actual: Fue implementado mediante un IP-Core que incluye una RAM que almacena síncronamente lo que recibe del bloque "Siguiente estado", y un multiplexor que permite elegir si la salida corresponde al valor almacenado en la RAM o al estado inicial.

- Siguiente estado: Corresponde a un IP-Core que recibe el estado actual y envía el siguiente estado calculado según las reglas que rigen el juego.
- LEDs Driver: Debido a la limitación de utilizar únicamente cuatro LEDs para la implementación, solo se pueden mostrar 4 de las 64 celdas totales del juego, para esto se usa un IP-Core que recibe como entrada cuatro switches que se usan para elegir la región del estado actual a mostrar.
- RGB Driver: Este bloque tiene por objetivo mostrar una "barra de vida" al suman la cantidad de celdas que se encuentran vivas en cada momento. Utiliza dos PWMs para variar la intensidad del LED RGB, de esta manera para por un continuo de colores entre verde y rojo. Los valores de comparación son instanciados a los registros del bloque mediante el uso del bloque ATG en test mode.

II. ACTIVIDADES REALIZADAS (1.5 PUNTOS)

AO1 (100%): Los bloques *clc_select*, *Next_State_Proc* y *Led_Display* contienen utilizando *component* los bloques *Clock_Div*, *adjacente* y *Led_Display* respectivamente.

AO2 (100%): Dentro de la arquitectura presentada anteriormente, que también se puede ver en la figura 6, se pueden ver los bloques *RAM_PL_AXI*, *LEd_display*, *RGB_driver*, *MAP* y

AO3 (100%): El ATG en Test Mode se usa para configurar las PWM del *RGB_driver*, de tal manera de aconfigurar los colores que se mostrarán en el led RGB para distintos estados del juego. Por otro lado el ATG en Full Mode es utilizado para cargar el estado inicial en la RAM que guarda el mapa del juego. Para lograr esto se envían dos mensajes usando *burst* que representan los 64 bits del mapa.

AC1 (100%): En el bloque *RAM_PL_AXI* existen dos entradas relacionadas con el mapa, una proviene del *AXI FULL* y la otra del PL. Dentro de este bloque se utiliza una maquina de estados para determinar cual de estas dos entradas mostrar como mapa actual, siendo el estado inicial (S_0) cuando se muestra el mapa inicial recibido por AXI, y un segundo estado (S_1) el que muestra el estado que va recibiendo desde el PL.

AC2 (100%): ILA y VIO son ambos utilizados dentro del proyecto. Algunos de los resultados de ejecuciones se pueden visualizar mediante ILA en la sección IV.

AC3 (100%): Se hace uso de múltiples operadores como concatenación (&), suma (+), mayor o igual (>=). Se hace uso de dos atributos que son *range*, que devuelve el rango de valores posibles para una variable, dentro de la funcion *count_ones*, la cual se usa en los bloques *next_state* y *RGB_driver*. Y el otro es dentro del mismo bloque *next_state*, en donde se utiliza el atributo *LOW* de la señal *address* para encontrar las posiciones de las celdas adyacentes a la que nos encontramos calculando.

AC4 (100%): Se utilizó variables en 3 partes del diseño. La primera de estas es en la función anteriormente mencionada *count_ones*, en donde la señal a la que se le va sumando la cantidad de 1's en el vector es una variable llamada *temp*. La segunda de estas fue en los dos casos en los que se llamó a esta función, en donde se creó una variable '*sum*'. El último caso es en el componente *adjacente*, dentro de *next_state*, en donde el valor de la siguiente celda es actualizado mediante el uso de una variable llamada *buff*.

AC5 (100%): Se usó código secuencial y concurrente en distintas partes del diseño. Un ejemplo de código secuencial es en el bloque *next_state*, en donde toda la lógica de este ocurre dentro de un *process(clk)*. Un ejemplo de código concurrente se encuentra en el bloque *led_mux*, en donde lo que se envía a los leds se escoge utilizando un *with/select* de la entrada de los switches.

AC6 (100%): Se hace uso de algunas funciones como *rising_edge*, además de la función *count_ones* que se hizó en los bloques *next_state* y *RGB_driver*, la cual cuenta la cantidad de 1's en una señal, la cual fue utilizada para contar la cantidad de celdas vivas en alguna sección de la grilla. Con respecto a los procedures, en el component *adjacente*, se creo la procedure *update_next*, el cual recibe la cantidad de vecinos vivos, y el estado actual de la celda, y te devuelve su siguiente estado.

AC7 (100%): Se utiliza el bloque AXI SmartConnect de Vivado. Este bloque actúa como intermediario entre maestros y esclavos en el bus AXI. Dentro de las propiedades de este bloque se puede elegir la cantidad de conexiones maestro y esclavo, lo que significa que un solo bloque es suficiente para interconectar todas las conexiones del proyecto que sean de tipo AXI.

III. RESULTADOS DE SIMULACIÓN (3 PUNTOS)

Se hizo una simulación post implementación del bloque *Next_State_Proc*, con los resultados que se pueden ver en la figura 2, en donde se separaron los vectores de 64 bits de entrada y salida en 8 vectores representando las filas de la grilla. En este, se puede ver como el bloque es capaz de calcular el siguiente estado de manera correcta. Si hacemos zoom a uno de los cambios, como se muestra en la figura 3, se puede apreciar como existe un pequeño delay en el cambio del estado de cada una de las filas, con respecto al reloj. También se puede observar como el valor de *cell_next* varia bastante durante el *rising_edge* del *clk*. Esto es debido a que los valores que almacenan el estado de los vecinos de una celda se almacena en la señal *neighbors*, por lo que tiene problemas para actualizar todos sus valores simultáneamente. Al contrario, el valor de *cell_next* es actualizado mediate el uso de una variable, por lo que es mucho más capaz de actualizar su valor frente a los cambios rápidos generados por *neighbors*.

Luego, en la implementación final, se agregaron momentáneamente dos ILAs para observar las transacciones generadas por los ATGs. En el caso del ATG en advanced mode, se pueden observar las señales relacionadas con la escritura en la figura 4. En este se puede observar el proceso completo de escribir en dos bursts el estado inicial del mapa, donde la primera mitad se escribe como el hexadecimal 0C8Cf000, y

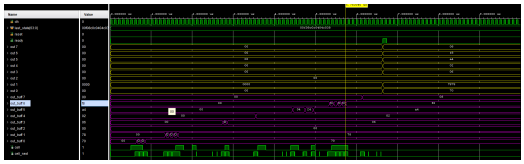

FIGURE 2: Resultados de la simulación post implementación del bloque *Next_State_Proc*

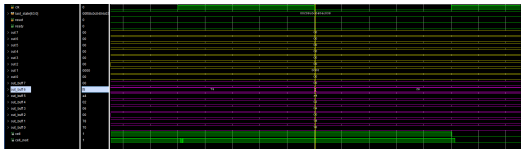

FIGURE 3: Resultados de la simulación post implementación del bloque *Next_State_Proc*, haciendo zoom a un cambio

la segunda mitad como 38C00404. También se puede observar el handshake hecho por las señales *WVALID* y *WREADY*.

Por otro lado, se puede ver la transacción generada por el ATG en test mode en la figura 5. En esta se puede ver como va cambiando la dirección de escritura, el dato a escribir, y el handshake hecho por las señales *WVALID* y *WREADY*.

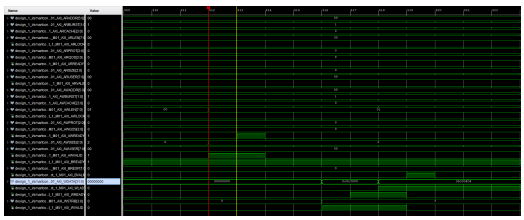

FIGURE 4: ILA leyendo el ATG en advanced mode

FIGURE 5: ILA leyendo el ATG en test mode

IV. RESULTADOS IMPLEMENTACIÓN (0.1 PUNTOS)

Para la implementación final se armó el *Block Design* de la figura 6. En este se puede ver como, con el objetivo de debuguear el sistema, se integró un *ILA* para observar las señales de *last_state*, *next_state* y *ready*, para confirmar que el sistema se encuentra calculando de manera correcta cada nuevo estado de la grilla. Adicionalmente, tenemos un *VIO*, el cual genera dos señales para debuguear, una de ellas determina si usamos un reloj de baja frecuencia para el juego, o si usamos un botón. La otra señal se utiliza para determinar si en los leds mostramos el estado anterior, o el proceso de calculo del siguiente estado. Para confirmar el correcto funcionamiento, es necesario saber cual es el estado inicial del juego, el cual fue configurado con el ATG en Full Mode, el cual configura un estado inicial en hexadecimal de (00f08c0c0404c038). Con este estado inicial, el juego seguirá la secuencia:

1) 00F08C0C0404C038
2) 08F8A40206007870
3) 88ACAD03063C4840
4) 9C82888811324CC0
5) 8CD0C4993936DCF0
6) 09251007C1838701
7) 811A0D0344040404
8) 0F9A1989040E0E02
9) 1880029F01000500
10) 00008A8C8D000018
11) 000009D08F000000
12) 00008850CF070000
13) 0000006E68080200
14) 0000046C68140000
15) 00000C6C40380000
16) 00001C7C44301000
17) 0008044244383000
18) 000004065C482800
19) 0000060232401000
20) 0000060320100000
21) 0000070700000000
22) 0002050502000000

Luego de llegar al estado 22, el siguiente estado será igual al anterior, por lo que se podría decir que la simulación terminó.

Si implementamos esto en la Zybo, podemos conseguir los resultados que se pueden observar en las figuras 7, 8, y 9. En estas se puede apreciar que el cambio de la grilla inicial (1) a la grilla (2) se ejecuta correctamente, también el cambio de (2) a (3), y que, una vez ha transcurrido el proceso por un tiempo, el proceso llográ llegar al cambio final desde el estado (21) al (22).

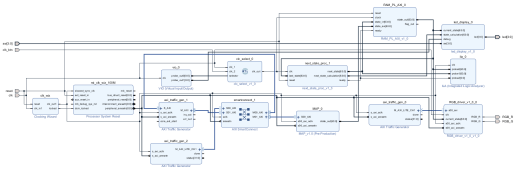

FIGURE 6: *Block Design* final del proyecto

FIGURE 7: Señales en el ILA durante al final del primer ciclo del juego

FIGURE 8: Señales en el ILA durante al final del segundo ciclo del juego

FIGURE 9: Señales en el ILA durante al final del ultimó ciclo del juego

V. CONCLUSIONES (0.2)

- Se esperaba que el ATG en advanced mode fuera capaz de cargar el estado inicial de '0F08C0C0404C038' en 2 bursts. Luego, en la figura 4, se puede ver como se cargaron los datos en el tiempo esperado.
- Se esperaba que la implementación fuese capaz de simular la secuencia de estados que se mostró en la sección IV. Luego, como se observó en las figuras 7, 8 y 9, se consiguió obtener los vectores de datos correctos en todas los ciclos del juego.

VI. TRABAJOS FUTUROS (0.2)

Existen dos grandes avances que se pueden realizar para continuar con este proyecto, estos son:

- Haciendo uso de un procesador: Implementar un procesador permite reemplazar el ATG en Full Mode, facilitando la carga del mapa inicial sin la necesidad de regenerar el bitstream. Además, con este procesador se pueden realizar diversos ajustes en el sistema, como cambiar el tamaño del mapa o provocar un reinicio. También se puede considerar la implementación de un canal UART para establecer una comunicación con el usuario a través de una consola.
- Hacer uso del puerto HDMI: Lograr generar video permitiría visualizar el desarrollo del juego de una forma más completa, ya que se tendría la opción de ver el mapa completo a la vez en lugar de utilizar solo 4 LEDs a la vez.

REFERENCES

[1] Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game "life" by Martin Gardner. Scientific American, 223, 120–123.