Fecha de entrega 09 de Mayo de 2025.

Zybomon

RODRIGO ELGUETA[1], JOSÉ QUIMI [1]
[1]Pontificia Universidad Católica de Chile (e-mail: primerintegrante@uc., jquimi@uc.cl)

SI Autorizo que mi proyecto (tal como ha sido entregado, sin nota ni comentarios de evaluación) sea publicado en un repositorio para pueda servir de guía y ser mejorado en proyectos de futuros estudiantes.

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

ABSTRACT El proyecto consiste en una simulación de combate por turnos inspirada en el popular videojuego Pokémon®. Para el desarrollo del juego se utilizó la FPGA Zybo Z7, donde, mediante módulos lógicos, se crean de manera pseudo-aleatoria los distintos Zybomones y se almacenan en un bloque de memoria RAM. El jugador puede interactuar a través de diferentes menús creados con máquinas de estados para definir su estrategia y responder a su oponente. El sistema proporciona información sobre las distintas estadísticas y atributos de los Zybomones a dispocición de cada jugador mediante retroalimentación visual utilizando los LEDs integrados en la placa. Por último se utilizaron "AXI traffic generators" para cargar los datos de color y luminosidad de los LEDs y los datos de la traducción de las estadísticas de los Zybomones en la BRAM a los números que se utilizan para el cálculo del daño que reciben, ahorrando espacio en la memoria. Se logró implementar correctamente el manejo de menús, la representación en LEDs y el control de lectura y escritura de la BRAM. Además se corroboró que la generación aleatoria produjera solo datos válidos y que el cálculo de daño fuera el correcto según la ecuación que determina este parámetro en base a las estadísticas de los Zybomones involucrados.

INDEX TERMS Pokemon-inspired, FPGA, VHDL, maquina de estados, block RAM.

I. ARQUITECTURA DE HARDWARE (1 PUNTO)

ZYbomon es un juego por turnos basado en el videojuego Pokémon®en el cual 2 jugadores pelean entre si comandando Zybomones para atacar. La implementación de este juego en la tarjeta Zybo-Z7 se compone de un generador de datos pseudo-aleatorios que son almacenados en una block RAM, luego estos datos se muestran al jugador utilizando los leds como medio de visualización y por medio de un conjunto de menús ambos jugadores eligen que acción tomar ese turno, otro módulo procesa estos inputs y carga a la memoria el nuevo estado del juego, este termina cuando todos los Zybomones de un jugador se quedan sin vida. El generador de datos consiste en un Linear Feedback Shift Register (LFSR) que genera una secuencia de bits pseudo-aleatoria que es leída por el modulo *mon_generator* el cual genera 8 secuencias de bits que corresponden a los datos de los Zybomones de cada jugador siendo la primera mitad para el jugador 1 y la segunda mitad para el jugador 2. Cada Zybomon se compone de distintos atributos:

- Vida actual: cuanto daño le falta recibir para quedar fuera de combate.
- Activo/Inactivo: solo un Zybomon por jugador puede combatir a la vez y este puede ser cambiado por alguno que este inactivo.
- Tipo elemental del Zybomon: puede ser fuego, planta, agua o normal y modifica el daño que recibe dependiendo del tipo elemental del ataque que reciba.
- Vida máxima: Vida inicial del Zybomon.
- Ataque: Cuanto daño hace con sus ataques.
- Defensa: Cuando daño recibe de ataques.
- Velocidad: Determina el orden de ataque, si es más veloz ataca primero, si es igual a su oponente se deja al azar.
- 4 ataques donde cada uno posee:
  - Tipo elemental del ataque: Altera el daño dependiendo del tipo del objetivo.
  - Poder: Cuanto daño hace el ataque en especifico.
  - Usos restantes: Cuantas veces se puede utilizar el ataque.

Cuando se terminan de generar y cargar los datos en la memoria se manda una señal al módulo *main_menu*. Este se compone de una maquina de estados que lee los datos y que genera una arquitectura de menús donde cada uno es representado por un estado distinto de la maquina. Los jugadores podrán ver la vida y el tipo del Zybomon enemigo activo por medio del LED rgb, luego podrán acceder a un

menú para seleccionar un ataque de su pokemon activo o un menú para cambiar su Zybomon activo. Además en estos menús tendrán la opción de revisar el poder y usos restantes de sus ataques o revisar las estadísticas de sus Zybomones respectivamente.

Una vez elegidas las acciones de los jugadores estas son enviadas al módulo *game_math*, el cual lee los datos de los Zybomones involucrados. Puede cambiar los Zybomones activos de los jugadores y/o determinar el orden de combate y calcular el daño infligido a cada Zybomon actualizando sus datos en la memoria y mandando una señal a *main_menu* cuando los eventos del turno se hayan terminado de procesar. Un ATG en AXI full carga las "look-up tables" que traducen los atributos de los Zybomones desde su representación en la BRAM a su valor matemático real utilizado en los cálculos.

Como varios bloques distintos se comunican con la misma block RAM el módulo *ram_controller* administra el acceso dependiendo de la señal de control que emite *main_menu*.

Por último el modulo *LED_handler* se encarga de procesar la información de los menús y mostrarla a los jugadores a través de los 4 LEDs corrientes y el LED RGB. Un ATG en AXI lite le entrega a este módulo los valores RGB que representan cada tipo elemental de Zybomon y los valores que utiliza el PWM proporcional para representar atributos como la vida y el poder de ataque con la intensidad de brillo.

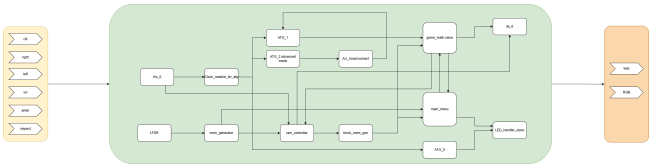

FIGURE 1: Diagrama general de la arquitectura lograda.

II. ACTIVIDADES REALIZADAS (1.5 PUNTOS)

AO1 (100%): El diseño cumple con los requerimientos de la actividad, la cual consiste en implementar al menos 3 componentes dentro de una misma entidad, se puede apreciar en los módulos de *mon_generator* para la creación pseudo-aleatorio de la información instancia *data_generator*, *led_handler* para el control de PWM de los RGB LEDs instancia 3 veces *proportional_pwm* y *main_menu* para la implementación del menó de selecciones instancia *select_menu*. Todos los módulos fueron correctamente implementados y funcionales.

AO2 (100%): El diseño cumple con los requerimientos de la actividad, la cual consiste en generar al menos 3 ip-cores propios en *block design* para incorporar al código. Esta actividad se cumple en los siguientes módulos: *LFSR*, para la generación de números pseudo-aleatorios, *mon_generator*, es un modulo propio que genera la información del juego y *ram_controler* que coordina el acceso de varios módulos a la block RAM.

AO3 (100%): El diseño cumple con la actividad, que consiste en implementar comunicación AXI con: i) al menos un IP esclavo propio con un ATG maestro en *test mode* y ii) al menos un IP esclavo propio con un ATG maestro en *advanced mode*, esta actividad se cumple en los módulos *game_math*, el cual recibe información de un ATG en *advanced mode* que es inicializado por un ATG en *test mode* y el móudlo *led_handler* que recibe información directamente de un ATG en *test mode*

AC1 (100%): El diseño cumple con la actividad, que consiste en generar una maquina de estados que sea central en el proyecto, esta actividad se cumple en el módulo *main_menu*, donde se utiliza una maquina de estados para determinar el cambio entre menús.

AC2 (100%): El diseño cumple con la actividad, que consiste en utilizar Vio e ILA para monitorear y modificar señales para le proyecto, esta actividad se cumple por medio de un módulo ILA que permite ver la salida de la BRAM, su señal de escritura y el daño calculado por *game_math* y el modulo Vio, conectado a *RAM_controler* para cambiar los datos escritos. Esto permite cargar cualquier estado de la partida particular que se desee examinar y probar.

AC3 (100%): El diseño cumple con la actividad, que consiste en utilizar 2 operadores y 2 atributos diferentes. Esta actividad se cumple en el módulo *LFSR* que emplea los atributos HIGH y LOW para identificar los bits más y menos significativos independiente del largo del vector. En *LED_handler* se utilizan los operadores lógicos XOR y AND para determinar que LED debe parpadear en el menú de selección.

AC4 (100%): El diseño cumple con la actividad, que consiste en utilizar variables para actualizar valores instantáneamente, esta actividad se cumple en el módulo *game_math*, donde se emplea una variable para actualizar el valor de un índice de un vector varias veces en un solo ciclo de reloj.

AC5 (100%): El diseño cumple con la actividad, que consiste en utilizar código secuencial y concurrente. Esto se cumple en el modulo *mon_generator* con código concurrente donde el uso de *generate* crea una secuencia binaria que solo depende de las entradas del bloque. Mientras que en el mismo módulo se emplea código secuencial en forma de una máquina de estados que va escribiendo la BRAM, donde la dirección de escritura de salida y los datos que se cargan dependen de la dirección de escritura del estado anterior.

AC6 (100%): En el módulo *mon_generator* se utiliza la función propia *concat_data* para contanenar los datos de un Zybomon generado en la secuencia de bits que lo representa en la BRAM. Mientras que en módulo *main_menu* el procedure *unravel_data* realiza la operación inversa entregando los datos del Zybomon en múltiples variables. Además el procedimiento *first_available* recibe una variable de entrada lo que no es posible hacer con funciones. Los procedimientos y funciones propias están definidas en un "package" llamado *zybomon_data*.

AC7 (100%): El diseño cumple con la actividad, que consiste en implementar algún ip-core que no se haya presentado en el curso. Esta actividad se cumple en el modulo *blk_mem_gen_0*, que consiste en un bloque de memoria

generado a través del ip-core Block memory generator. De esta forma se aprovecha el "hardware" dedicado de la ZYBO Z7 para generar memorias RAM.

III. RESULTADOS DE SIMULACIÓN (3 PUNTOS)

Se realizó una simulación de comportamiento y post-implementación de la escritura de la RAM por parte del módulo *game_math* con el objetivo de comparar la teoría y la realidad del circuito diseñado.

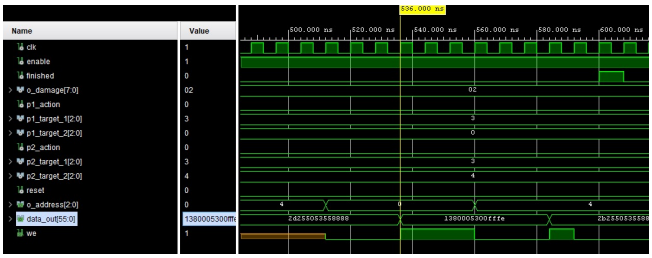

FIGURE 2: Simulación de comportamiento de lectura y escritura de la BRAM en *game_math*.

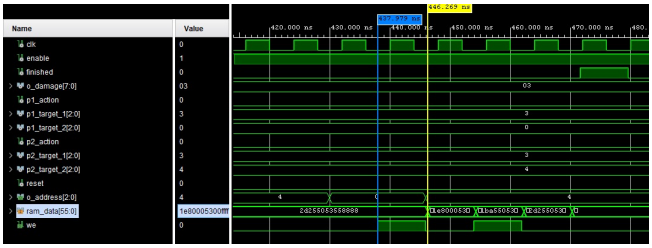

FIGURE 3: Simulación post-implementación de lectura y escritura de la BRAM en *game_math*.

En primer lugar se puede apreciar que en ambas simulaciones existe un retardo entre que se cambia la dirección de lectura de la BRAM y que cambie su salida de datos. Por otro lado, se puede apreciar en la figura 2 que en la simulación de comportamiento el "write enable" tiene un efecto simultáneo en los datos de la memoria, mientras que en la figura 3 esta operación tiene un retardo de 8.29ns, ligeramente mayor a un ciclo de reloj. Esto implica que la dirección de escritura cambia antes de que esta se realice dejando la posición 0 sin escribir su dato correspondiente, arruinando el funcionamiento del juego.

En cuanto a la diferencia entre señales y variables en el mismo módulo *game_math* se utilizó una variable denominada *index* que sirve para indexar un vector de estados. Este funciona como un "stack" donde se colocan en orden los estados que va a seguir la máquina el cual se construye en un ciclo de reloj cambiando varias veces el valor de *index* para ir rellenando el vector de estados. Con el objetivo de visualizar la variable en la simulación se creo una señal *var_index* que se actualiza cada ciclo de reloj con el valor de *index*. Además se creó otra señal *sig_index* con el objetivo de comparar como se comportaría "*index* si fuera una señal.

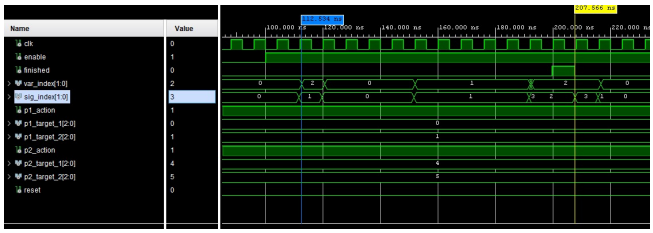

FIGURE 4: Carga de datos al módulo *led_handler* por AXI lite.

De la figura 4 se aprecia que en el marcador azul cuando se construye el "stack" *var_index* es 2 ya que en este caso las acciones de los jugadores producen que se carguen 3 estados en la pila. Sin embargo *sig_index* es 1 indicando que su valor solo se actualizó una vez provocando una construcción erronea de la pila. Además esta señal alcanza valores de 3 en ciertos momentos mientras que *var_index* nunca so sobrepasa.

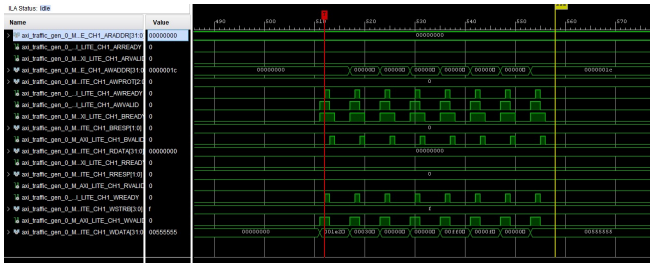

FIGURE 5: Carga de datos al módulo *led_handler* por AXI lite.

La figura 5 muestra las señales involucradas de AXI lite para cargar los datos de color y luminosidad. La señal *AWADDR* indica la dirección de escritura en el esclavo y la señal *WDATA* los datos que el maestro esta transfiriendo. El "handshake" se realiza por medio del levantamiento de las señales *AWVALID*, *BREADY* y *WVALID* que indican que la dirección es valida, que el esclavo esta listo para recibir y que los datos son validos respectivamente. Luego se levanta *AWREADY* indicando el comienzo de la transacción y por ultimo se levanta *BVALID* indicando que el esclavo recibió la información comenzando la transferencia de otro dato. Para la transmisión de datos por AXI full se utilizó un ATG en AXI lite que primero carga las instrucciones en la *command RAM*, luego carga datos en el registro maestro y por último activa el control maestro. Esta transacción se aprecia en la figura 6 y se emplean las mismas señales que las mencionadas de la figura 5. Aunque las direcciones y los datos transmitidos son distintos y en mayor cantidad.

La transacción en AXI full se muestra en la figura 7. Se puede apreciar que se realiza toda en una sola ráfaga incremental como indica la señal *AWBURST* que esta en 1. Además la cantidad de datos escritos en ráfaga es 24 como indica la señal *AWLEN* (18 en hexadecimal es 24). En este modo la dirección de escritura representada por *AWADDR*

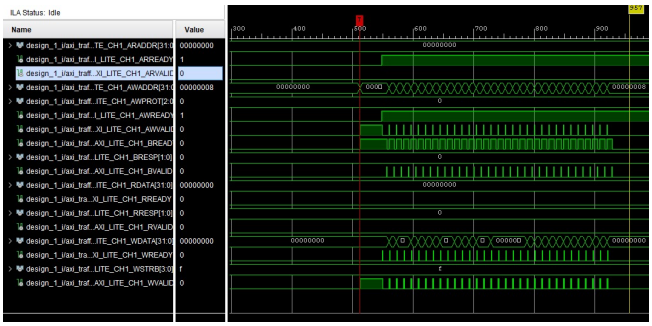

FIGURE 6: Inicialización del ATG AXI full por medio de AXI lite.

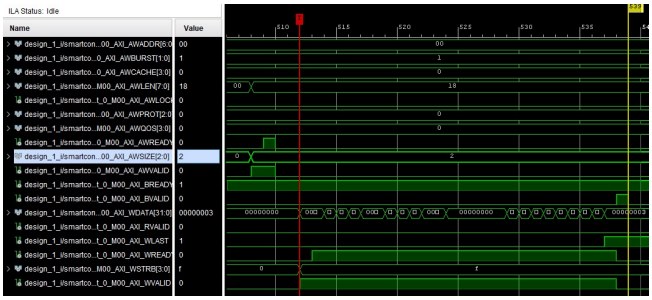

FIGURE 7: Carga de datos al módulo *game_math* por medio de AXI full

se mantiene constatne en el primer valor y los datos de la señal *WDATA* son transmitidos rápidamente. Con respecto al "handshake" este empieza con el maestro levantando la señal *AWVALID*, si *BREADY* esta activa también entonces se levanta *AWREADY*. Luego de unos instantes se levanta *WVALID* y comienza la transmicsión hasta que el maestro envia la señal *WLAST* indicando el último dato a escribir. Una vez terminada el esclavo envía *WDATA* confirmando la recepción.

IV. RESULTADOS IMPLEMENTACIÓN (0.1 PUNTOS)

Se pudo implementar Zybomon correctamente en la tajeta Zybo Z7. Se generaron datos aleatorios válidos, el flujo por los menús funciona y muestra la información apropiada por los LEDs y la lectura y escritura de la BRAM es adecuada de acuerdo a los comandos de los jugadores. Esto a pesar de que hubo complicaciones en el desarrollo algunos módulos.

Como se explicó anteriormente existe un retardo entre la activación del "write enable" de la BRAM y la actualización de sus datos, además hay otro retardo de 2 ciclos de reloj entre que se cambia la dirección y se entrega la información correspondiente. Esto se resolvió implementando un tiempo de espera de 2 ciclos para cada operación de lectura y escritura. Además en el módulo *game_math* el cálculo del daño involucra multiplicar y dividir, este corresponde a la ecuación (1). Sin embargo en simulaciones post-implementación el resultado dio un valor completamente distinto a lo esperado. Se teorizó en su momento que, dada la complejidad de la operación, podría tratarse de un problema de timing que requería de pipelining. Esto se pudo solucionar aplicando la sugerencia encontrada en [1] que propone usar registros para guardar las entradas de la ecuación.

$$dmg = poder * \left(1 + \frac{ataque - 1}{defensa}\right) * mod\_por\_tipo \quad (1)$$

V. CONCLUSIONES(0.2)

- Al generar datos los datos de manera pseudo-aleatoria se obtienen valores de vida máxima, ataque, defensa, velocidad y poder de ataque entre 0 y 2. Este es el comportamiento esperado del módulo *mon_generator* ya que representan los 3 posibles valores que debieran tomar estos atributos: bajo, medio y alto.
- El cálculo de daño del módulo *game_math* para un combate con ataque de 10, defensa de 3, poder de ataque de 3 y modificador por tipo de 2 resultó en 24. Siguiendo la ecuación (1) aplicando los mismos datos se obtiene que el resultado esperado es el mismo por lo que el módulo funciona correctamente.

VI. TRABAJOS FUTUROS (0.2)

Una mejora del diseño actual para trabajar a futuro es implementar animaciones al final del turno, ya que en su estado actual es difícil captar que sucede cada turno sin un indicador claro que muestre que acción elige cada jugador y sus consecuencias. Esto se podría lograr con un nuevo módulo que reciba instrucciones desde el menú principal y utilice un "axi timer" o contadores propios para apagar y encender los LEDs de manera sincronizada y en momentos precisos generando patrones fácimente interpretables. Otra opción es utilizar una pantalla con "sprites" de los Zybomones y que estos se muevan de acuerdo a lo que sucede en el juego.

Otra mejora posible es utilizar un procesador para realizar los cálculos de daño estableciendo una comunicación por AXI entre el procesador ARM y el módulo *game_math* añadiendo un esclavo AXI adicional. Esto permitiría usar números más grandes y complejizar la ecuación añadiendo variables adicionales como un parámetro que represente la relación entre el tipo elemental del ataque y del Zybomon atacante, a diferencia del ya implementado que relaciona el tipo del defensor con el del ataque recibido.

REFERENCES

[1] How to pipeline DSP48 input? - Page 1 (2023, September). Eevblog.com. https://www.eevblog.com/forum/fpga/how-to-pipeline-dsp48-input/.