3 vidas para adivinar

VICENTE BARRIOS[1], JOSÉ ALAMOS[1]
[1]Pontificia Universidad Católica de Chile (e-mail: vicente.barrios@uc., josealamos@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 Para la implementación del proyecto se programó la FPGA Zybo Z7-10 en VHDL. El funcionamiento se basa en un juego de dos jugadores, donde el jugador 2 debe adivinar el número ingresado por el jugador 1. Para esto los jugadores deben ingresar sus números mediante los cuatro switches, por lo tanto se puede elegir del 0 al 15. El jugador 2 tiene tres intentos para adivinar y además se le dan pistas, como por ejemplo si el número que ingresó es mayor o menor al número a adivinar y también puede pedir mostrar el primer y último bit del número.
Se programó una máquina de estados donde los distintos estados son las jugadas a ingresar o la lectura de datos guardados, se incluyó comunicación AXI Lite en test modo y AXI Full en advance mode para guardar las jugadas en una memoria RAM, se utilizan los switches para el ingreso de información como lo son los números que deben escribir los jugadores y también para ver las jugadas guardadas en la RAM. Los Leds se implementaron para mostrar cuando el switch se encuentra en '1', para así visualizar de mejor manera el número ingresado y además indican si la jugada fue exitosa o errónea, usando RGB con un color distintivo.
Los botones se usan para avanzar o retroceder de estado, para reset y poner el modo información.
La implementación fue exitosa y se cumplieron las actividades obligatorias y complementarias siguiendo la línea del desarrollo del juego.

INDEX TERMS FPGA, Zybo Z7-10, VHDL, AXI Lite, AXI Full, máquina de estados, memoria ram.

I. ARQUITECTURA DE HARDWARE (1 PUNTO)

Observando el diagrama general expuesto en la figura 1, comenzando de abajo hacia arriba tenemos la unión de los bloques PrimeraJugada y atr que lo que hacen es entregarnos en que posición estamos del juego, según las entradas de los botones nxt, atrboton y rst.
Así, esta señal de estado va al bloque intento 1 que nos entrega la jugada del J1 y los 3 intentos del J2, según sean los switches escogidos.
Por otro lado, abajo tenemos el bloque EasyMode el cual se activa con el botón btn1, el que cuando lo apretamos podemos revelar el bit de más a la derecha y después el bit de más a la izquierda del J1. Este bloque entrega una señal al bloque general MuestroLeds para revelar los bits mencionados del J1, según sea el estado del juego.
Por la parte de arriba, tenemos la comunicación AXI Full, en la que podemos mandar 4 ráfagas de 4 números cada una, correspondientes a las 16 posibles jugadas del J1 que puede elegir con los 4 switches. Mientras que abajo tenemos la comunicación AXI Lite en la que escribimos los 16 registros respectivos con los que puede jugar el J2 en sus intentos. De esta forma, estos registros tanto del J1 como del J2 se guardan en la memoria ram especificada en el bloque memoria.
Finalmente, el bloque MuestroLeds, es el bloque central que nos entrega los 4 leds según sea el estado del juego en que nos encontramos, así se puede mostrar la jugada del J1 o algún intento del J2, o los bits revelados del J1. También entrega colores rgb para indicar si el J2 acertó o no el número del J1.

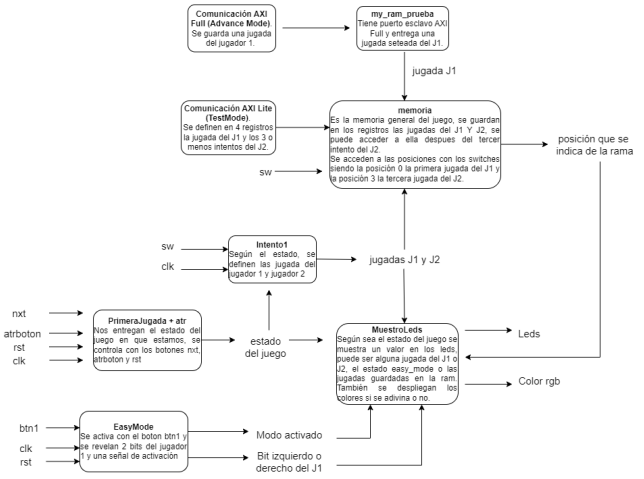

FIGURE 1: Diagrama general.

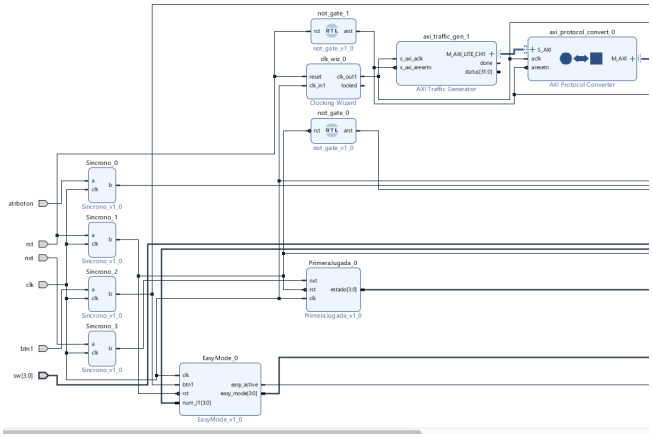

FIGURE 2: Primera parte block-design.

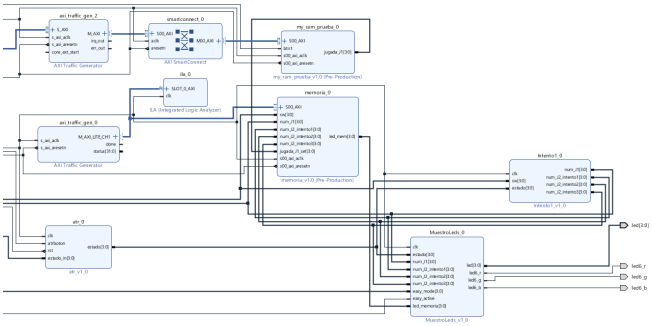

FIGURE 3: Segunda parte block-design.

II. ACTIVIDADES REALIZADAS (1.5 PUNTOS)

AO1 (100%): *Descripción:* El código cumple con la integración de cinco components dentro del IPCore MuestroLeds, los cuales cumplen las funciones de comparar las jugadas del jugador 1 y del jugador 2 (en sus respectivos 3 intentos), indicar cuando es mayor o menor respecto a la jugada del jugador 1 e informar si se adivino o no el numero del jugador 1.

Los componentes utilizados se llaman clk_divider, parpadeo, Compara, EsMayorOmenor y Color_rgb. Todos se ubican dentro del IPCore MuestroLeds.

Nivel de Logro: 100%. Completamente logrado, el código implementado ha compilado, se logra simular luego de la implementación, se logra generar bitstram y cargarlo en la ZYBOZ7.

AO2 (100%): El juego cuenta con más de 3 packages o IPCores creados por nosotros mismos, específicamente se utilizan los packages PrimeraJugada, EasyMode, atr, MuestroLeds, Intento1, memoria, sincrono, my_ram_prueba creados 100% por nosotros. El resto son bloques configurados por nosotros, como los ATG, smartconnect, axi_protocol_convert entre otros.

De esta forma el package PrimeraJugada y atr cumplen la función de entregarnos el estado del juego en que estamos según los botones de avanzar, retroceder y rst que apretamos en el juego. Mientras que el package Intento1 recibe el estado y le asigna valores a los intentos de los jugadores (juegan con los switches) según el estado en que se encuentre el juego. Posteriormente, el package MuestroLeds define que mostrar en los leds según el estado del juego y asigna colores si se adivina o no el número del jugador 1, y se realizan las comparaciones entre las jugadas. Después, en el package memoria se pueden guardan las jugadas tanto del jugador 1 como de jugador 2 mediante protocolo AXI. También se añado un package Sincrono para poder sincronizar respecto a un reloj los botones que se utilizan en el juego, para evitar problemas de glitches.

Nivel de Logro: 100%. Completamente logrado, el código implementado ha compilado, se logra simular luego de la implementación, se logra generar bitstram y cargarlo en la ZYBOZ7.

AO3: (100%): El código cumple con la implementación de dos ATG configurados en modo test mode para la implementación de comunicación AXI Lite y un ATG configurado en modo advance para la implementación de la comunicación AXI full.

En primer lugar, usamos la comunicación AXI Full para escribir cuatro ráfagas correspondientes a los posibles 16 registros con los que puede jugar el jugador 1, para poder realizar esto usamos un bloque AXI protocol convert para poder comunicar el puerto maestro axi lite con el esclavo AXI Full y después mandamos esto a un IPCore creado por nosotros con puerto axi full el que se denomina my_ram_prueba, aca usamos un smartconnect para poder comunicar el puerto maestro axi full con el esclavo axi full.

En segundo lugar, mediante un ATG AXI lite en modo test mode cargamos las posibles 16 jugadas con las que podía jugar el J1 y los 3 intentos del J2, entonces así después cargábamos los registros a la memoria principal denominada memoria. De esta forma, cuando el jugador 2 completa su tercer intento, y presionamos el botón nxt se muestra en los leds lo que guardamos en la ram, siendo la posición 0 la

primera jugada del J1, la posición 1 la primera jugada del J2, la posición 2 la segunda jugada del J2, y la posición 3 la tercera jugada del J2.
Nivel de Logro: 100%. Completamente logrado, el código implementado ha compilado, se logra simular luego de la implementación, se logra generar bitstram y cargarlo en la ZYBOZ7.
AC1: (100%): En el proyecto se utilizó una máquina de estados para revelar el bit de más a la izquierda y de más a la derecha de la jugada del J1, esto se hace cuando apretamos el botón de información el cual es el btn2 (el tercero de derecha a izquierda) de la tarjeta ZYBO. Esto se realiza específicamente en el IPCore EasyMode, ya que cuando se presiona el botón se va avanzando de estado, cuando estamos en el estado 1 se revela el bit de más a la izquierda del jugador 1 (bit en la posición 3) y cuando estamos en el estado 2 se revela el bit de más a la derecha (bit en la posición 0). Vale recalcar que esta implementación se muestra en los leds solamente cuando estamos en las secciones del juego en que estamos comparando la jugada del jugador 1 con el intento X del jugador 2 (marcado también por luces rgb)
Nivel de Logro: 100%. Completamente logrado, el código implementado ha compilado, se logra simular luego de la implementación, se logra generar bitstram y cargarlo en la ZYBOZ7. A continuación se muestra el desarrollo en donde la estructura de la máquina de estados fue obtenida de la clase máquina de estados del curso.

```
----- upper section State Machine ----------
process (estado_actual, btn1)
begin
   if rising_edge(btn1) then              -- Sólo si apretamos el botón btn1 avanzamos al estado siguiente.
   case estado_actual is
       when estado0 =>
               estado_next <= estado1;    --Avanzamos al estado siguiente.
       when estado1 =>
               estado_next <= estado2;    --Avanzamos al estado siguiente.
       when estado2 =>
               estado_next <= estado1;    --Avanzamos al estado siguiente.
        when others =>
               estado_next <= estado0;
    end case;
    end if;
end process;
```

FIGURE 4: Seccion Upper de máquina de estados.

```
----- lower section State Machine ----------
process (clk, rst)
begin
    if (rst = '1') then
        estado_actual <= estado0;
    elsif (NOT clk'STABLE AND clk='1') then      --Uso atributo -> es como rising_edge(clk).
        estado_actual <= estado_next;
    end if;
end process;
-------------------------------
```

FIGURE 5: Seccion Lower de máquina de estados.

AC2: (100%): Tal como se muestra en el video para el juego utilizamos los 4 botones, 4 leds, y 4 switches.
El botón nxt (btn0 de tarjeta) lo usamos para avanzar de estado en el juego, el botón atrboton (btn1 de tarjeta) para retroceder de estado, el btn1, nombre en constrains (btn2 de tarjeta) para extraer información de la jugada del J1, y el botón rst para resetear el juego. También mediante los switches escogíamos las jugadas respectivas a cada jugador y mediante los leds podíamos ver estas.

```
process (estado_actual, num_J1)
begin
    case estado_actual is
        when estado1 =>                                          --En el estado1 revelamos el bit de más a la izquierda del J1.
            result_func_i <= revela_bit_izquierda(num_J1);
            case result_func_i is
                when '0' =>
                    easy_mode <= "0000";
                when '1' =>
                    easy_mode <= "1000";
                when others  =>
                    easy_mode <= "0000";
            end case;
        when estado2 =>                                          --En el estado1 revelamos el bit de más a la derecha del J1.
            result_func_d <= revela_bit_derecha(num_J1);
            case result_func_d is
                when '0' =>
                    easy_mode <= "0000";
                when '1' =>
                    easy_mode <= "0001";
                when others  =>
                    easy_mode <= "0000";
            end case;
        when others =>
            easy_mode <= "0000";
    end case;
end process;
```

FIGURE 6: Continuación de máquina de estados.

Nivel de Logro: 100%. Completamente logrado, el código implementado ha compilado, se logra simular luego de la implementación, se logra generar bitstram y cargarlo en la ZYBOZ7.
AC3: (100%): En nuestro proyecto logramos utilizar 5 operadores y atributos deferentes.
En el video se muestran los distintos proyectos en Vivado en que se usaron los códigos.

- Operadores usados:
  1) **NOT** línea 91 de IPCore EasyMode.
  2) **AND** línea 91 de IPCore EasyMode.
     Estos fueron usados con la intención de hacer un rising_edge para el clock.
  3) **=** ,en línea 100 de IPCore EasyMode.
     Para ver cuando la señal de rst era igual a 1.
  4) $>$ en línea 52 del component Compara del IPCore MuestraLeds.
     Con el propósito de ver si la la jugada del J1 es mayor.
  5) $<$ en línea 55 del módulo Compara del IPCore MuestraLeds.
     Con el propósito de ver si la la jugada del J1 es menor.
- Atributos usados:
  1) **STABLE** en línea 91 de IPCore EasyMode.
     Para poder hacer un rising_edge para el clock.
  2) **LEFT** en línea 54 del IPCore EasyMode.
     Para poder obtener la posición del bit de mas a la izquierda del J1 para así revelar su valor.
  3) **RIGHT** en línea 67 del IPCore EasyMode.
     Para poder obtener la posición del bit de mas a la derecha del J1 para así revelar su valor.
  4) **EVENT** en línea 68 del component Compara del IPCore MuestraLeds.
     Para poder detectar cuando cambia la señal de clock y así hacer un rising_edge.
  5) **LENGTH** en línea 60 del IPCore PrimeraJugada.
     Para poder obtener el largo del vector que queremos castear de entero a std_logic_vector.

AC4: (100%): Se utiliza una variable denominada "conta", en el proceso_contador del component Color_rgb (línea 63) del

IPCore MuestroLeds.

```
proceso_contador: process (clk)
variable conta: integer := 0;      --Utilizo una variable para poder implementar el contador de la pwm.
begin                              --Esta variable es interna al process y se actualiza intantáneamente
    if rising_edge(clk) then       --de esta forma no necesita que termine el process para cambiar su valor.
        conta := conta + 1;
        if (conta = 255) then
            conta := 0;
        end if;
    end if;
    contador <= conta;
end process proceso_contador;
```

FIGURE 7: Utilización de variable conta

La finalidad de usar esta variable interna al proceso_contador es para implementar el contador respectivo para generar la PWM a utilizar para generar los colores para indicarle al jugador 2 si adivino o no la jugada del jugador 1.
AC5: (100%): En este proyecto utilizamos código secuencial y código concurrente para poder desarrollar el proyecto.
De esta forma, usamos código secuencial en la mayoría de las veces, como por ejemplo en los diferentes process utilizados dentro del componente Color_rgb para definir la pwm como los valores máximos a los que se tenía que llegar según el color que queríamos indicar.

```
proceso_select: process (clk, winOlose)
begin
    if rising_edge(clk) then
        case winOlose is
            when '0' =>             --Color rojo
                max_r <= 255;
                max_g <= 0;
                max_b <= 0;
            when '1' =>             --Color azul
                max_r <= 0;
                max_g <= 0;
                max_b <= 255;
            when others =>            --Color verde
                max_r <= 0;
                max_g <= 255;
                max_b <= 0;
        end case;
    end if;
end process;
```

FIGURE 8: Utilización de código secuencial.

Mientras que código concurrente utilizamos en el componente EsMayorOmenor dentro del IPCore MuestroLeds, en el hicimos un multiplexor utilizando la clausula when - else, asignando la señal de salida un vector según si el juego había terminado o no, y si era mayor o menor, esto se evidencia en el juego posterior a que el jugador 2 juega su respectivo turno.
A continuación se señala su implementación.

```
entity EsMayorOmenor is
  Port (MayorOmenor : IN STD_LOGIC := '0';
        winOlose : IN STD_LOGIC := '0';
        led_mayor_o_menor : OUT STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000"
        );
end EsMayorOmenor;

architecture Behavioral of EsMayorOmenor is
--Acá implementamos un multiplexor ---> Código concurrente que no se gatilla por un clk.
-- La salida (led_mayor_o_menor) solamente depende de las entradas actuales (MayorOmenor y winOlose) y no
-- de sus estados previos.
begin
    led_mayor_o_menor <= "0000" WHEN winOlose='1' ELSE
                         "0010" WHEN (winOlose='0' AND MayorOmenor='1') ELSE
                         "0001" WHEN (winOlose='0' AND MayorOmenor='0') ELSE
                         "1111";

end Behavioral;
```

FIGURE 9: Utilización de código concurrente.

Al igual que se menciona dentro del código, es importante notar que la salida (led_mayor_o_menor) solamente depende de las entradas actuales (MayorOmenor y winOlose) y no de sus estados previos.
AC6: (100%): En el código se utilizan functions y procedures para el desarrollo del juego. Específicamente, se utilizan 2 funciones en el IPCore EasyMode, para revelar el bit de más a la izquierda y de más a la derecha del jugador 1. Estas se detallan a continuación:

```
architecture Behavioral of EasyMode is

----- Function------
--En la sig función nos entrega el valor del bit de más a la izquierda de una señal a.
FUNCTION revela_bit_izquierda(SIGNAL a: std_logic_vector(max_val downto 0))
    RETURN std_logic is
begin
    if (a(a'LEFT) = '1') then
        return '1';
    else
        return '0';
    end if;
end function;
--------------------
```

FIGURE 10: Función revela_bit_izquierda.

```
----- Function------
--En la sig función nos entrega el valor del bit de más a la derecha de una señal a.
FUNCTION revela_bit_derecha(SIGNAL a: std_logic_vector(max_val downto 0))
    RETURN std_logic is
begin
    if (a(a'RIGHT) = '1') then
        return '1';
    else
        return '0';
    end if;
end function;
--------------------
```

FIGURE 11: Función revela_bit_derecha.

Así, estas funciones las utilizamos cuando apretamos el botón de información (tercer botón de derecha a izquierda), en las situaciones del juego en donde comparábamos resultados.
Por otra parte, se utilizó un procedure dentro del component Compara del IPCore MuestraLeds.
La importancia de este código es que nos entrega una señal "winolose" que vale 1 solamente cuando los números a comparar son iguales mientras que nos entrega otra señal "MayorMenor" cuando a es mayor a b. En el fondo la importancia del uso de este procedure es que podemos retornar más de un valor y estas sirven para ingresar después al código concurrente anteriormente especificado.
Su detalle se muestra a continuación:

```
architecture Behavioral of Compara is

---
PROCEDURE CualEsMayorYganaOno( SIGNAL a, b : IN std_logic_vector(3 DOWNTO 0);
    SIGNAL MayorMenor, winlose: OUT std_logic) is
        begin
            if (a>b) then
                MayorMenor <= '1';
                winlose <= '0';
            elsif (a<b) then
                MayorMenor <= '0';
                winlose <= '0';
            else
                MayorMenor <= '0';
                winlose <= '1';
            end if;
end CualEsMayorYganaOno;
---
```

FIGURE 12: Procedure CualEsMayorYganaOno.

AC7: (100%): Como actividad extra, utilizamos el IPCore Integrated Logic Analyzer entre la conexión del Axi Traffic Generator con puerto maestro AXI Lite y el IP Core memoria con puerto axi esclavo axi Lite. Este sirve para implementar una especie de osciloscopio de las señales, ya que podemos ver en un diagrama como cambian en el tiempo.
A continuación se detalla su conexión dentro del diagrama de bloques:

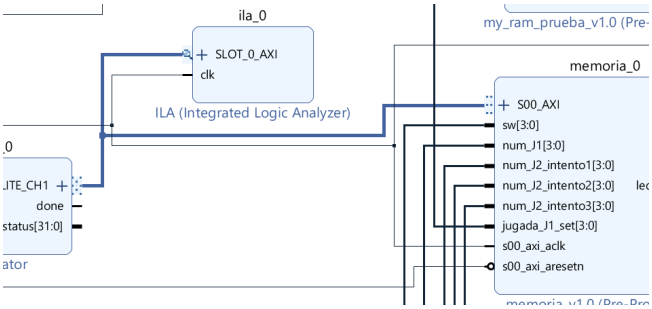

FIGURE 13: Conexión bloque ILA (Integrated Logic Analyzer).

III. RESULTADOS DE SIMULACIÓN (3 PUNTOS)

En la imagen a continuación se muestra la simulación para Intento1:

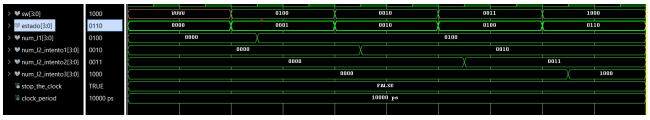

FIGURE 14: Simulación Intento1

En esta se puede notar que al cambiar de estado, los valores de switch coinciden con las distintas jugadas de los jugadores, por lo que la máquina de estados funciona correctamente. Respecto a esto se puede ver que al cambiar de estado e ingresar valores en los switch existe un pequeño delay al actualizar las variables de jugada, sin embargo esto no influye en el diseño, ya que el delay es muy pequeño y al ser un proceso manipulado por una persona no alcanzaría a cambiar de estado antes de actualizar correctamente las señales.

En la siguiente imagen se muestra la simulación de PrimeraJugada:

FIGURE 15: Simulación PrimeraJugada

Se puede ver que al hacer un cambio en el botón nxt se avanza de estado.

La imagen que aparece a continuación muestra la actualización de la señal contador dentro de MuestroLeds:

FIGURE 16: Simulación contador

En este caso contador es una señal por lo que se actualiza cada vez que termina el process, a la cual se le asigna el valor de conta, que es una variable, por lo tanto, esta se actualiza inmediatamente después de que se ejecuta la instrucción que la modifica.

A continuación se muestra la comunicación de AXI Full:

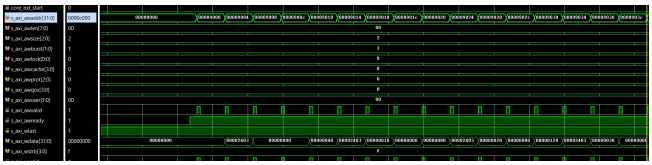

FIGURE 17: Simulación AXI Full 1

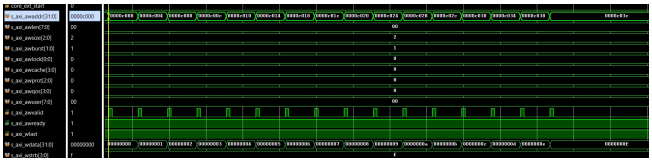

FIGURE 18: Simulación AXI Full 2

En la primera imagen se pueden ver los datos que se escriben en el DUT por parte del Driver, en este caso son los datos de instrucciones. En la segunda imagen se pueden ver los datos que se cargan en la RAM del DUT, los cuales son una secuencia de 0 a 15.

Por último se muestra la comunicación AXI Lite:

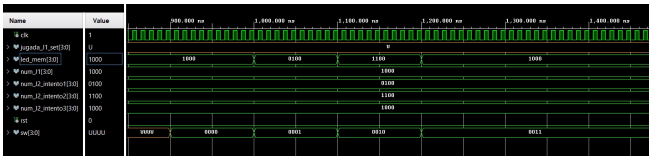

FIGURE 19: Simulación AXI Lite

En la comunicación AXI Lite, se guardan las jugadas hechas. En la figura se puede ver que al ingresar un 0 en los

switch se lee la posición 0, que es la que contiene la jugada del jugador 1, cuando los switch son 1 se lee el primer intento del jugador 2, cuando los switch son 2 se lee el segundo intento del jugador 2 y finalmente cuando los switch son 3 se lee el tercer intento del jugador 2. Se puede notar que la comunicación funciona, ya que la salida led_mem contiene correctamente las jugadas según el valor ingresado en los switch.

IV. RESULTADOS IMPLEMENTACIÓN (0.1 PUNTOS)

En aspectos generales, fue exitoso el resultado de la implementación del juego, lo más complejo fue organizar bien el bloque MuestroLeds, ya que es el que recibe la mayor cantidad de señales provenientes de otros bloque por lo que había que ser muy ordenado y simular el comportamiento de todas las señales.
También, resulto desafiante la implementación de muchos IPCores en carpetas diferentes, ya que optamos por separar las funciones en distintos bloques por lo que coordinarlos era una tarea no menor.

V. CONCLUSIONES(0.2)

- En el proyecto se logró el objetivo de la utilización de más de tres packages creados completamente por nosotros para la implementación del juego como por ejemplo, PrimeraJugada, atr, MuestroLeds, EasyMode, Intento1, etc.
- Tambien, se logró la utilización de más de tres components creados por nosotros dentro de un package, como es el caso de los components clk_divider, parpadeo, Compara, EsMayorOmenor, Color_rgb, etc.
- Además, se consiguió la comunicación de la jugada del jugador 1 y de las tres jugadas del jugador 2 mediante ATGs en modo test mode y advance mode los que se comunicaron con dos IPCores creados por nosotros, y los datos convergen a la ram general del juego, en la cual podemos ingresar posterior al tercer intento del jugador 2 y así tener un historial de las jugadas de la partida.

VI. TRABAJOS FUTUROS (0.2)

Es importante notar que para acceder al bloque de memoria ram el jugador 2 debe jugar si o si 3 intentos, independientemente si este adivinó al primer intento o al último (tercer intento), o si no pudo adivinar. Por lo que un trabajo futuro que se puede realizar para este proyecto es poder acceder a la memoria ram en cualquier momento del juego sin estar condicionado al estado en que se encuentre.

REFERENCES

[1] Ayudantías y Laboratorios del curso Sistemas Electrónicos Programables. IEE2463. Profesor Félix Rojas. Pontificia Universidad Católica de Chile.
[2] Diapositivas curso Sistemas Electrónicos Programables. IEE2463. Pontificia Universidad Católica de Chile.