Fecha de entrega 11 de 12 de 2023.

Disclaimer This latex template has been obtained from IEEE ACCESS for educational purpose.

Jack The Duck

JUAN IGNACIO LORCA[1], KEVIN JUSTINIANO[1]
[1]Pontificia Universidad Católica de Chile (e-mail: juan.lorca@uc.cl, kevin.justiniano@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 Se desarrolló un juego tipo $Arcade$ el cual consiste en el control de la posición vertical del personaje $Duck$ con tal de evadir los $logs$ que se arrastran por el suelo. El presente proyecto es demostrado en el perisférico TFT Graphic Display Module (LCD) acopañado de una melodía que es emitida por el perisférico BUZZER en paralelo a la ejecución del código. La tarjeta de desarrollo Zybo Z7-10 en conjunto a la tarjeta Booster de Texas Instruments (TI) fueron utilizadas para la creación de este $Arcade$. Esta última fue controlada por medio del procesador ARM Cortex-A9 incorporado como $Zynq7\ Processing\ System$ en la tarjeta Zybo Z7-10. Se lograron implementar todas las actividades pedidas por enunciado de proyecto e incluso se le otorgó una funcionalidad al almacenamiento ejecutado en la tarjeta SD externa.

INDEX TERMS ARM, Cortex A-9, programación en C, Vitis, Vivado, máquina de estados, PWM, memoria, VHDL, Booster, Zybo Z7-10, Arcade, sensor.

I. ARQUITECTURA DE HARDWARE Y SOFTWARE (1 PUNTO)

EN el marco de este proyecto, se llevaron a cabo la implementación y desarrollo de los atributos fundamentales que caracterizan a un juego de arcade. Se crearon códigos tanto para el hardware como para el software esencial destinado al diseño de "Jack The Duck". Este juego se centra en el control de la posición vertical de un personaje con el objetivo de esquivar troncos o registros que se desplazan hacia su ubicación. El juego incorpora un sistema de dificultad que consta de tres niveles: Fácil (Easy), Medio (Medium) y Difícil (Hard), cada uno con una duración predeterminada de 160 segundos. Al concluir este tiempo, el juego se detiene automáticamente, dirigiendo el flujo del programa hacia un menú final. En este menú, los usuarios tienen la opción de reiniciar el juego, salir del mismo y finalizar el programa, o guardar los datos del registro de puntajes en caso de haber jugado una o más veces. Esta funcionalidad proporciona una experiencia interactiva y versátil para los jugadores, permitiéndoles tomar decisiones sobre el curso del juego y la gestión de sus puntajes.

Se utilizaron distintos protocolos de comunicación, entre ellos I2C y SPI, para comunicar los distintos perisféricos de la Booster con el procesador Zynq ubicado en la tarjeta Zybo Z7-10. Cada uno de estos protocolos fue inicializado y configurado en Vitis por medio de programación de software. La lógica del juego también fue programada en el software y cargada en un set de instrucciones al procesador.

Para comunicar el procesador con el Hardware de la Zybo se utilizó el protocolo de comunicación AXI, en particular AXI Lite, que nos permitió acceder a los 4 registros asociados a los puertos esclavo y así poder escribir datos e información en ellos que nos fueron de gran utilidad a la hora de manejar módulos de la Programmable Logic (PL) con el Processing System (PS).

La arquitectura del Hardware se detalla en el siguente diagrama de bloques 1:

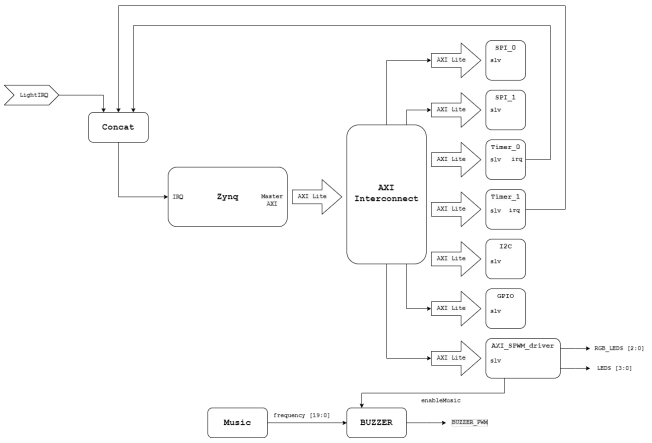

FIGURE 1: Diagrama del Hardware

Se pueden observar los módulos asociados a los distintos protocolos de comunicación cuyos pines se encuentran rutea-

dos a los conectores JC, JD y JE de la tarjeta Zybo Z7-10. De esta manera se logran enviar señales entre la Booster y la Zybo.

En cuanto a la arquitectura de Software se utilizaron loops *while* para el secuenciamiento de instrucciones. Además, se definieron variables globales y dentro de la inicialización del `main()`. La lógica utilizada para la programación del juego en Software puede ser modelada de acuerdo a la figura **??**.

II. ACTIVIDADES REALIZADAS (1.5 PUNTOS)

AO1 (100%): La pantalla simula el juego *Jumping Jack* el cual, para efectos de este proyecto, será llamado *Jumping Duck*. El usuario puede controlar la posición `y` de *Duck* con tal de lograr sobrepasar los *logs* o *troncos* sin que este sea tocado por uno, por lo tanto se actualiza dinámicamente el contenido enseñado por el LCD. La dificultad del juego radica en que persférico de control es utilizado. Para este juego se definieron los perisféricos *Joystic*, *Potencimetro* y *Acelermetro* y la dificultad radica según el orden anterior. Además se utilizan los perisféricos *BUZZER* y el *sensor de luz*, cuya funcionalidad será explicada mas adelante. El código utilizado se encuentra comentado y dividido en secciones de acuerdo al perisférico utilizado. *Nivel de Logro:* 100%. Completamente logrado, el código implementado ha compilado, se logra simular por medio de debugging en Vitis luego de la implementación y generación del bitstream.

AO2 (90%): Se utilizan *AXI Timers* de 32 bits cada uno, cuyos tiempos de acción son de 10 segundos y fueron definidos en Vitis por medio del comando `TMR_LOAD0` y `TMR_LOAD1`. Estos timers fueron conectados al procesador Zynq por medio de un bloque *Concat*. Para ello fue necesario habilitar las interrupciones `IRG_F2P` del procesador y conectar la salida del módulo *Concat* a esta entrada. Es importante mencionar que el Zynq posee de un controlador de interrupciones *General Interrupt Controller* (GIC) en su interior. La inicialización y posterior configuración de los timers y sus respectivas interrupciones fueron realizadas en Vitis por medio de comandos encontrados las librerías `"xtmrctr.h"`, `"xscugic.h"` y `"xparameters.h"` y que, por ejemplo, nos permitieron asignar la periocidad para cada interrupción realizada por un timer. Ambas interrupciones están asociadas a una representación en el cambio de nivel durante el juego. La interrupción del timer 0 cambia el el display del nivel en la esquina superior izquierda del LCD y la interrupción del timer 1 se encarga de mostrar el nivel por medio de los 4 leds presentes en la tarjeta Zybo Z7-10.

Se utilizó la interrupción dada por el sensor de luz OPT3001 [1]. Este sensor posee de un pin de interrupción (INT Pin) que se encuentra ruteado al pin U15 de los conectores Pmod JD de la tarjeta Zybo z7-10. Luego, en los constraints de Vivado, este pin fue descomentado y utilizado en el block design del proyecto por medio del módulo *Concat* que va al manejo de interrupciones del procesador para que posteriormente la interrupción fuese configurada en Vitis. Para ello, fue necesario buscar el ID asignado a esta interrupción en la librería `xparameters_ps.h` y luego utilizarlo en la inicialización de las conexiónes que realiza el GIC. Con tal de levantar esta interrupción cuando la medición por parte del sensor de luz superara un umbral o threshold, fue necesario acceder a los registros del chip OPT3001 Ambient Light Sensor (ALS) para configurar esta opción por medio de buses I2C, los cuales son enviados a los 7-bits de *address* del dispositivo en cuestión. Estos 7-bits vienen dados por `OPT_ADDR = 0x44` y la función utilizada para el envío de los datos de configuración fue `XIic_Send()`. Esta función recibe los argumentos `iic.BaseAddress`, `OPT_ADDR`, `byteCount`, `XIIC_STOP`, y el puntero `(u8 *)&config` que apunta a los datos de configuración para el sensor. Se definieron 3 arreglos, de 3 datos de 8 bytes cada uno, para configurar la interrupción. Los primeros 8 bytes indican el *address* del registro de configuración y los restantes 16-bits (offset) son parámetros de configuración:

1) `u8 config[3] = {0x01, 0xC4, 0x18}`: se accede al registro de configuración (0x01) y se indica por medio del bit 4 el *latch field* que controla la funcionalidad del mecanismo de reporte de interrupciones que posee el sensor.
2) `u8 configLow = {0x02, 0x5A, 0x9A}`: se accede al registro *Low Limit* (0x02) que establece el límite de comparación inferior para el mecanismo de interrupción. Los 16-bits restantes se dividen en 4 bits para un exponente $LE[15:12]$ y 12 bits para un valor de threshold $LT[11:0]$. Estos bits son utilizados en la siguiente ecuación para el cómputo del valor de umbral asociado para la comparación inferior de la luz medida [1]:

$$lux_{low} = 0.01 \cdot (2^{LE[3:0]}) \cdot TL[11:0] \qquad (1)$$

   Así, el valor definido para el threshold inferior fue de: 868.48.
3) `u8 configHigh = {0x03, 0xBF, 0xBB}`: se accede al registro *High Limit* (0x03) que establece el límite de comparación superior para el mecanismo de interrupción. Al igual que el registro *Low Limit*, el valor para el umbral superior de comparación se calcula usando la ecuación 1, lo que resulta $lux_{high} = 68157.44$

Lamentablemente esta actividad no pudo ser llevada a cabo por completo dado el mal estado del sensor. Se realizaron múltiples experimentos en Vitis donde se hizo debuggin y una serie de *prints* y se llegó a la conclusión de que el sensor de luz no lograba realizar una medición correcta par cierta configuración de umbrales y mecanismo de interrupción. Posteriormente, se ajustó la configuración de manera diferente, y las interrupciones se activaban con una polaridad opuesta a la establecida inicialmente. En este nuevo escenario, cuando los valores se encontraban por debajo de los umbrales inferiores o por encima de los umbrales superiores, la interrupción no se generaba. Sin embargo, en cualquier otra condición, la interrupción se producía según lo previsto.

Esto nos llevó a cambiar el bit número 3 de confiruración al acceder al registro 0x01 del sensor, pero nada funcionó. De todas formas, la interrupción se encuentra activa en el código y conectada al GIC del Zynq. *Nivel de Logro:* 90%. Casi-completamente logrado, el código implementado ha compilado, se logra simular por medio de debugging en Vitis luego de la implementación y generación del bitstream, pero no puede ser utilizada la interrupción por problemas en el sensor.

AO3 (100%): Se diseñó un módulo en Vivado con puerto AXI. Este módulo fue definido como AXI_SPWM_driver el cual recepciona por medio de AXI y sus registros lo siguiente:

1) `slv_reg0`: recibe la señal `enable` (0x00000000 o 0xFFFFFFFF) la cual permite la modulación de una sinusoide de 1 [Hz] por medio de PWM (Pulse Width MOdulation) y esta modulación va a parar a el led RGB. La señal `enable` es enviada cada vez que la posición `y` de $Duck$ supera cierto umbral, con tal de encender en el led RGB una combinación de sus canales cada vez que se acciona cualquiera de los perisféricos utilizados para el control de $Duck$.
2) `slv_reg1`: actúa como buffer que se dirige hacia los 4 leds de la tarjeta Zybo Z7-10. Los leds son escritos cada vez que la interrupción asociada al timer1 se levanta.
3) `slv_reg2`: (UART) Por medio del computador se escribe en el terminal un número de 1 a 15 para determinar el color asociado al led RGB de la tarjeta Zybo Z7-10, el cual a través del protocolo de comunicación UART es enviado al procesador Zynq el cual se encarga de escribir en este registro el comando asociado a la combinación de canales para el led RGB. Logramos escribir comandos en el terminal gracias a la función `scanf()` que recibe la dirección de la variable dónde se quiere almacenar el comando enviado.
4) `slv_reg3`: (UART) por medio del computador se escribe en el terminal el número 1 el cual a través del protocolo de comunicación UART es enviado al procesador Zynq el cual se encarga de escribir en este registro el valor 0xFFFFFFFF para habilitar al perisférico $BUZZER$ y que este pueda tocar la música implementada.

Esta comunicación Progracessing System - Programmable Logic (PS - PL) la logramos gracias al comando `Xil_Out32(.)` de la librería `xil_io` en Vitis, el cual recibe las direcciones de los registros asociadas al puerto AXI en las cuales se quiere esribir un dato y el dato a escribir. La dirección principal del módulo con puerto AXI viene dada por el parámetro `ENABLE_SPWM_BUFFER_ID` y le sumamos 4 para ir accediendo a cada uno de sus registros esclavos.

Por medio de lo escrito en los registros `slv_reg2` y `slv_reg3` logramos configurar nuestro hardware utilizando el protocolo UART. Se define un BaudRate de 115200 (tasa de transferncia de bits), 1 bit de inicio, uno de término y 8 bits de datos para establecer la comunicación computador-procesados. En particular, los parámetros configurados fueron el color del led RGB cuya intensidad lumínica es modulada por una SPWM (Sinusoidal Pulse Width Modulation) y se habilitó la música emitida por el módulo $BUZZER$. La lógica asociada a la modulación SPWM se encuentra dentro del módulo AXI_SPWM_driver. *Nivel de Logro:* 100%. Completamente logrado, el código implementado ha compilado, se logra simular por medio de debugging en Vitis luego de la implementación y generación del bitstream.

AC1 (100%): Se diseñaron 2 máquinas de estado en Vitis. Dado que se desarrolló un juego tipo $Arcade$ fue necesario implementar lógica para un menú de inicio, para el flujo del juego, para un menú final, para un mensaje de término y para una sección extra que guarda datos. Para la implementación de cada una de estas lógicas, y sus respectivos $while\ loops$ por separado, se utilizó la variable `playing` como un estado que toma valores de 0 a 4.

AC2 (100%): Se utilizaron estructuras, arreglos, punteros, funciones y macros en el desarrollo del juego. La estructura utilizada fue definida como `Player` y recibe los atributos `name` y `playerScore`. Se utilizan macros para definir las funciones asociadas a la dinámica de $Duck$. Se definieron las siguientes ecuaciones dinámicas:

$$v(t) = a \cdot t + v_o \tag{2}$$

definida en el macro `VEL(A,V,Ts)` y la ecuación

$$y(t) = \frac{a}{2}t^2 + v_y \cdot t + y_o \tag{3}$$

definida por el macro `POS(A,V,X,Ts)`. También, se utilizaron funciones para ir actualizando los gráficos y/o pixeles asociados al display LCD. Por útltimo se utilizan los puteros auxiliares `aux_int` y `aux_float` que son usados constantemente para no tener que hacer variables nuevas que nunca se volverán a usar, de esta forma se ahorra memoria.

AC3 (100%): Se utilizan los los perisféricos sensor de temperatura (TMP006) y micrófono incorporados en la tarjeta de desarrollo Booster. El primero es utilizado para detener la reproducción de la música emitida por el perisférico $BUZZER$ si es que se supera cierto umbral de temperatura definido en el código. El segundo es utilizado para accionar la reproducción de música en el $BUZZER$ si es que la medición de este supera cierto umbral. Ambos thresholds o umbrales fueron determinados de manera experimental.

AC4 (100%): (Video) Se logró bootear la tarjeta Zybo Z7-10 al almacenar el código en su memoria Flash. Para ello se creó el archivo BOOT.bin, luego se añadieron los archivos necesarios para crear la imagen y programar la memoria flash de la Zybo Z7-10. Por último, se cortocircuitó el jumper $qspi$ para poder energizar la tarjeta y correr el código e intrucciones cargadas. La imagen asociada al archivo `.bin` fue creada de manera correcta y la operación de programar la memoria Flash resultó exitosa.

AC5 (100%): Se hace uso de los módulos VIO e ILA de Vivado para medir las señales dentro de nuestro PL. Se utiliza el módulo Vio para visualizar los valores de las frecuencias de salida en de un bloque diseñado en Vivado que contiene los valores máximos de un contador asociado a la frecuencia de un tono en específico. El módulo ILA es utilizado para medir la señal PWM que va hacia el $BUZZER$, como también la $SPWM$ que va hacia el led RGB.

AC6 (100%): Se utiliza el bloque $BUZZER$ para emitir una música tipo $Arcade$ de 5 tonos y en loop. Para ello se diseñó el bloque $Music$ que almacena valores máximos de un contador los cuales se relacionan a la frecuencia del tono emitido por medio de la siguiente ecuación:

$$MAX_{count} = \frac{f_{clk}}{f_{tone}} \tag{4}$$

dónde la frecuencia de reloj utilizada fue la del clock maestro de la Zybo Z7-10 (125 [MHz]). De esta manera, fijando el valor de MAX_{count} para un contador se puede obtener una señal diente de sierra cuya frecuencia viene dada por f_{tone}. Las frecuencias utilizadas y el su MAX_{count} asociado vienen dados por la siguiente tabla 1:

Frecuencia [Hz]	MAX_COUNT [Hex]
220	8AB76
247	7B8D9
277	6E2BF
330	5C7A4
415	49895

TABLE 1: Mapeo de frecuencias

Para que el $BUZZER$ emita los tonos anteriormente descritos es necesario que a la salida genere una señal PWM con ciclo de trabajo del 50% y de la frecuencia dada por los valores entregados en la tabla 1. De esta forma, el bloque $BUZZER$ entregado en el template de Vivado para el proyecto fue modificado.

AC7 (100%): Se utiliza la memoria SD externa para almacenar puntajes almacenados en el arreglo `scoreRegister` y valores de temperatura en archivos `.csv` por separado. Para poder crear el archivo a escribir y escribir en tal archivo se usó la librería `xiffs` la cual fue agregada al proyecto de plataforma en Vits. Esta librería provee del *headers* `ff.h` el cuál fue utilizado en los archivos de código `.c` y el *header* `.h` llamados `sdCard.c` y `sdCard.h`. Estos archivos fueron extraídos de [2].

III. RESULTADOS (3 PUNTOS)

A. IP CORE CREADOS

Se realizaron dos simulaciones post síntesis. La primera muestra los resultados obtenidos para el módulo $SPWM$ que se encuentra dentro del bloque AXI_SPWM_driver. Este módulo posee de las entradas `clk`, `rst`, `freq` y `enable`, y la salida `pwm_out`. A continuación se simula considerando que la señal `enable`, que proviene desde el procesador al accionar cualquier perisférico asociado al control de $Duck$, siempre estará activa. Además, se utiliza una frecuencia de 100 [Hz] para el contador que recorre el arreglo de tonos.

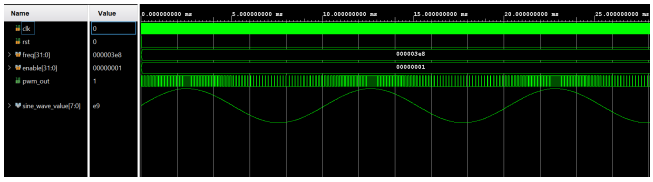

FIGURE 2: Test Bench módulo $SPWM$.

Se puede apreciar de manera correcta la modulación de la sinusoide por medio de la señal diente de sierra.

La segunda simulación viene dada por el módulo $Speaker$ el cuál fue implementado sólo para fines de simulación, ya que en el diseño real para el proyecto, los dos módulos que componen a $Speaker$, $Music$ y $BUZZER$, se encuentran por separado.

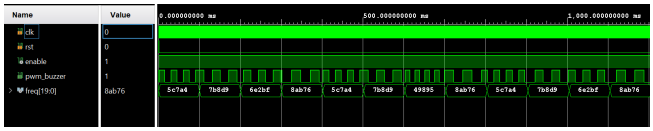

FIGURE 3: Test Bench módulos $Music$ & $BUZZER$

Se puede apreciar de manera correcta el cambio en la frecuencia de la señal dada por la PWM de cyclo de trabajo 50%.

B. DEBUGGING: CÓDIGOS EN C

Con tal de debuggear nuestro código en C se utilizaron 2 métodos, el primero por medio de constantes *prints* utilizando la función `xil_printf()` de la librería `xil_printf.h`, y el segundo realizando debugging en Vitis agregando las variables más influyente a la sección $Expressions$. A continuación se muestran imágenes con el debugueo implementado para corregir nuestro código. Se observaran las variables de tipo `int`: `playing` (representa los estados del juego), `difficulty` (representa la dificultad escogida en el menú de inicio) y `score`. Con tal de capturar una interrupción se creó la variable `int intrCounter` la cual representa un contador que se incrementa cada vez que la **interrupción Timer0** es levantada.

Se inicializan todas las variables a observar al pasar a esta línea de código:

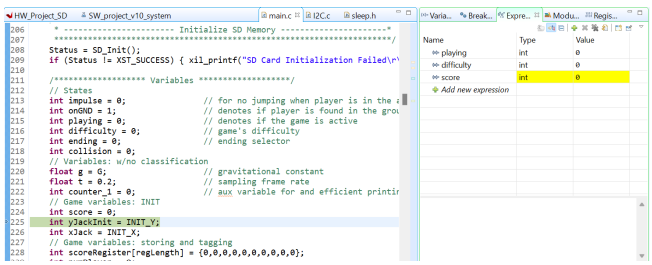

FIGURE 4: Inicialización variables

Podemos ver que a todas se les asigna el entero 0, como esperábamos a que pasara. Luego se entra al menú principal ya que `palying=0` y se mueve la posición vertical del persférico *Joystic* para ver un cambio en la variable `difficulty`. Este cambio se puede ver a continuación:

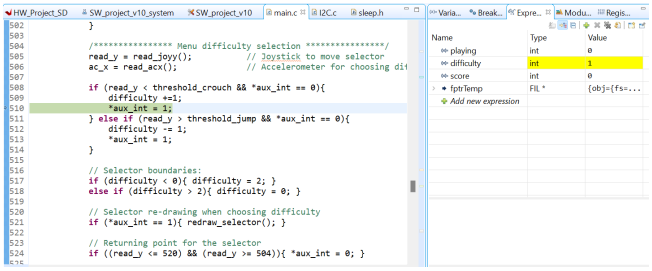

FIGURE 5: Cambio en una sóla variable de observación

Con esto último podemos validar la operación del *Joystic* como selector de dificultad para el juego.

Para capturar la interrupción, debemos esperar a que se inicialicen y configuren todas las interrupciones, por lo que una vez que pasemos todo eso utilizando *StepOver*, cada 10 segundos deberíamos ver un incremento en la variable `intrCounter`.

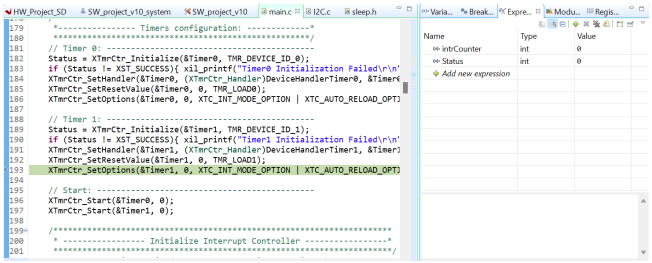

FIGURE 6: Estado interrupción 1

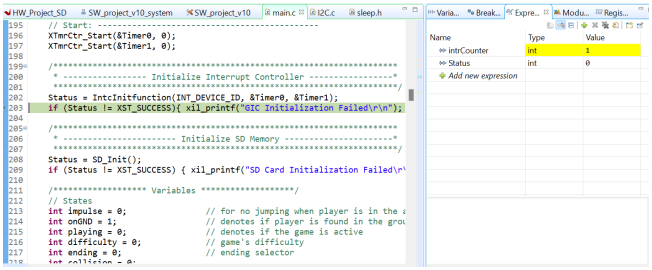

FIGURE 7: Estado interrupción 2

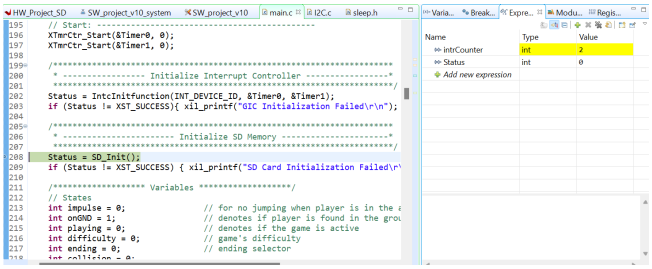

FIGURE 8: Estado interrupción 3

No se observa un incremento en la variable `intrCounter` cada 10 segundos, sino que esta incrementa cada vez que pasan 10 segundos y nos movemos a la siguiente línea de código, en otras palabras el Program Counter (PC) apunta a la siguiente instrucción.

C. DEBUGGING: VIO

Para realizar un debugging de los comandos enviados por AXI desde el procesador al PL, se utilizó VIO que nos permite ver el dato enviado en su representación digital y numérica. En esta prueba se quizo ver si era factible activar la música del *BUZZER* por medio de un comando enviado desde el terminal del computador al PS y luego al PL. El resultado se muestra a continuación:

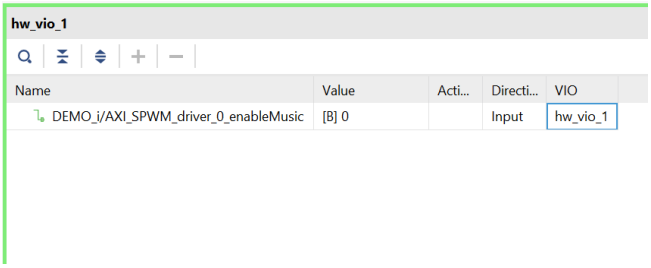

FIGURE 9: Visualización inputs 1

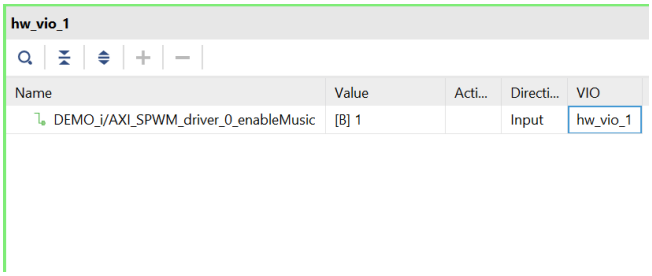

FIGURE 10: Visualización inputs 1

Se puede observar que para este caso la variable `enableMusic` de 1-bit cambia de 0 a 1. Este cambio es producto de: (1) Se envía un 1 desde el computador hacia el PS por medio de UART. (2) El procesador recepciona este 1 y lo envía por AXI Lite al módulo *AXI_SPWM_driver*. Esto último lo hace cuando el PS corre la instrucción dada por la función `Xil_Out32(.)`. (3) El módulo AXI recepciona

este valor, lo guarda en uno de sus 4 registros y luego este registro actúa como buffer hacia la salida `enableMusic`.

IV. RESULTADOS IMPLEMENTACIÓN (0.1 PUNTOS)

La implementación fue correcta, el flujo del programa funciona bien, las interrupciones asociadas a los timers se levantan cada 10 segundos por lo que funcionan de manera correcta (esto simulando ya el juego en la tarjeta Booster y utilizando $prints$), lo único que no pudo ser validado por los resultados anteriores fue la interrupción asociada al sensor de luz.

Debugging por VIO nos fue de gran utilidad para identificar el estado de la variable `enableMusic` y así validar el camino descrito que recorre un dato desde el computador hacia el PL.

V. CONCLUSIONES(0.2)

Se evidencia un flujo coherente del programa, tanto en su componente de hardware como en el software implementado. Se logra una paralelización efectiva de los contadores vinculados a los temporizadores con la ejecución del código y el bucle principal, permitiendo la incorporación eficiente de interrupciones. Dado su carácter de juego tipo "Arcade", las variables y periféricos bajo control son actualizados en tiempo real, consolidando la dinámica esencial que demanda un juego digital dirigido por un usuario. Este enfoque asegura una experiencia de juego fluida y receptiva, contribuyendo al éxito global de la implementación.

VI. TRABAJOS FUTUROS (0.2)

Como trabajo futuro quedaría mejorar considrablemente la lógica asociada al juego y con ello optimizar el flujo del programa. Se podrían agregar objetos “voladores" para los cuales $Duck$ tendría que agacharse. Esto último requerie de un amplio conocimiento de lo que es programar en C. Aprender a utilizar de manera eficiente punteros y estructuras sería un aspecto clave a la hora de mejorar el programa.

Además, se piensa hacer una implementación mas eficiente y ordenada. Esto último se debe a que por ejemplo, para entrar a jugar en uno de los modos de dificultad enseñados en el menú de inicio, hay que mover la trajeta Booster para activar un threshold asociado a la posición x leída por el acelerometro, lo que se podría hacer de una manera más fácil utilizando los botones de la tarjeta Booster (que se encuentran ruteados a la Zybo) y módulos GPIO en Vivado para hacer la comunicación con el procesador.

REFERENCES

[1] Texas Instruments (2014). POPT3001 Ambient Light Sensor (ALS) Datasheet (Rev. A, November 2017). Texas Instruments Incorporated.
[2] Peralta, I. (2022, November 16). Video Resumen Ayudantía 10 - SD. YouTube. https://www.youtube.com/watch?v=n5zFwmG9dhw&list=PLD_NS6WnGnVCsTGb5ug_y1m0h0Z-GIsbS&index=11