E.T.S. de Ingeniería Industrial, Informática y de Telecomunicación

TRANSMISOR ARINC 429

Grado en Ingeniería en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Yelko Vázquez Suárez

Directores de proyecto

- Orbital: Carlos Zubieta Arricivita
- UPNA: Santiago J. Led Ramos

PAMPLONA, JUNIO DE 2015

Abstract

The aim of this project is to design and implement an ARINC 429 transmitter with an AXI communication interface using a hardware description language (HDL). This includes the design and execution of all kinds of required simulations to implement a pre-certifiable IP core, as well as programming and testing it on a FPGA. This is made by using Xilinx tools as Vivado and SDK for programming and using Xsim, Modelsim and Octave for executing and testing the simulations.

The design is approached in a modular way, breaking down the project into six blocks and four phases of development. The blocks are the AXI slave, the FIFO, the control logic, the gap controller, the shift register and parity check and the waveform shaper. The development phases are designing and implementing the blocks, designing and implementing the simulations, synthesizing and implementing, and programming and testing the FPGA.

Resumen

El objeto de este proyecto es diseñar e implementar un transmisor ARINC 429 con un interfaz de comunicación AXI usando un lenguaje de descripción hardware. También abarca el diseño y ejecución de todos los tipos de simulación requeridos para hacer un IP Core pre-certificable, así como la programación y testeo en la FPGA. Se realiza con las herramientas de Xilinx Vivado y SDK para la programación y con Xsim, Modelsim y Octave para las simulaciones.

El diseño se aborda de forma modular, descomponiendo el proyecto en seis bloques y cuatro fases de desarrollo. Los bloques son el esclavo AXI, la FIFO, la lógica de control, el controlador de gap, el registro de desplazamiento y comprobación de paridad y el codificador. Las fases de desarrollo son el diseño e implementación de los bloques, el diseño e implementación de las simulaciones, la síntesis e implementación, y la programación en la FPGA.

Palabras Clave

AMBA, ARM, ARINC 429, Aviónica, AXI, FPGA, IP Core, Modelsim, Octave, Script, Transmisor, VHDL, Vivado, Xilinx, Xsim, Zedboard, Zynq.

Índice de Contenido

1 Introducción 6
2 Objetivos 7
3 Metodología 8
3.1 Implementación de Módulos y Testbenches 8
3.2 Agilismo Aplicado al Proyecto 11
4 Desarrollo e Implementación 12
4.1 Introducción 12
4.2 Definición Técnica de un Transmisor ARINC 429 12
4.3 Definición Concreta del Interfaz Esclavo AXI 15
4.4 Uso del IP Core 17
4.5 Módulos e Implementación 17
4.5.1 Esclavo AXI 17
4.5.2 FIFO 20
4.5.3 Control Logic 21
4.5.4 Gap Control 23
4.5.5 Shift Register and Parity Check 24
4.5.6 Waveform Shaper 26
4.6 Diseño e Implementación de Testbenches 28
4.6.1 Control Logic 28
4.6.2 Gap Control 30
4.6.3 Shift Register and Parity Check 31
4.6.4 Waveform Shaper 32
4.6.5 Transmitter 33
4.6.5.1 Transmitter Automatic Testbench 33
4.6.5.2 Transmitter Autotestbench Status 36
4.6.5.3 Transmitter Frequencytest 37
4.7 Síntesis e Implementación 37
4.8 Programación de la FPGA 38
5 Análisis de Resultados y Conclusiones 41
6 Líneas Futuras 42
7 Bibliografía 45

Índice de ilustraciones y tablas

Ilustraciones

Ilustración 1. Formato de palabra en ARINC 429 6
Ilustración 2. Diseño preliminar del transmisor A429 con los módulos a implementar 7
Ilustración 3. Estructura tipo del método de diseño de dos procesos 9
Ilustración 4. Estructura tipo del método empleado para realizar testbenches automáticos ... 10
Ilustración 5. Definición gráfica de la distribución del tiempo en un proyecto llevado con Scrum (4) 11
Ilustración 6. Diseño preliminar con interconexiones entre registros del AXI y los módulos del transmisor 13
Ilustración 7. GUI de configuración del módulo FIFO Generator 13
Ilustración 8. Gráfico de pulso para definición de las tolerancias de la línea de salida (5) 14
Ilustración 9. GUI de generación de esclavo AXI básico 15
Ilustración 10. Selección del nombre y path del AXI esclavo 18
Ilustración 11. Definición del tipo de interfaz AXI a generar 18
Ilustración 12. Módulo esclavo AXI personalizado en formato caja negra 19
Ilustración 13. Interconexión interna entre registros y salidas adicionales 20
Ilustración 14. Visualización del funcionamiento de la FIFO (6) 20
Ilustración 15. Módulo FIFO en formato caja negra 21
Ilustración 16. Módulo Control Logic en formato caja negra 23
Ilustración 17. Módulo Gap Control en formato caja negra 24
Ilustración 18. Formato de transferencia de palabras ARINC429 25
Ilustración 19. Módulo Shif Register and Parity Check formato caja negra 26
Ilustración 20. Entradas/Salidas módulo Waveform Shaper 27
Ilustración 21. Módulo Waveform Shaper formato caja negra 27
Ilustración 22. Diseño preliminar final 28
Ilustración 23. Simulación de comportamiento del módulo Control Logic 29
Ilustración 24. Muestra de la salida generada por la ejecución del script de comprobación 29
Ilustración 25. Simulación de comportamiento del módulo Gap Control 30
Ilustración 26. Fragmento de la salida provocada por la ejecución del script de comprobación 31
Ilustración 27. Simulación de comportamiento del módulo Shift Register and Parity Check 32
Ilustración 28. Fragmento de la salida producida por el script de comprobación 32
Ilustración 29. Simulación de comportamiento del módulo Waveform Shaper 33
Ilustración 30. Salida producida por el script de comprobación 33
Ilustración 31. Simulación a gran escala del transmisor 34
Ilustración 32. . Simulación a pequeña escala del transmisor 35
Ilustración 33. Fragmento de la salida del script de comprobación del transmisor 35
Ilustración 34. Simulación del registro de estatus, comienzo y vista global 36
Ilustración 35. Salida del script de comprobación del estado del transmisor 37
Ilustración 36. Salida del script de comprobación de la frecuencia 37
Ilustración 37. Esquemático de interconexión del módulo AXI y el Transmitter 38
Ilustración 38. Interconexionado interno del módulo Transmitter 38

Ilustración 39. Diagrama de bloques del sistema completo 39
Ilustración 40. Zedboard con transmisor y receptor 40
Ilustración 41. Interfaz del SDK de Xilinx 40
Ilustración 42. ILA integrado en Vivado en ejecución 41
Ilustración 43. IP Cores aislados (7) 43

Tablas

Tabla 1. Tolerancias definidas en la especificación ARINC 429 (5) 14
Tabla 2. Bits y funcionalidad desempeñada por el registro de control del esclavo AXI 16
Tabla 3. Bits y funcionalidad desempeñada por el registro de estado del esclavo AXI 16

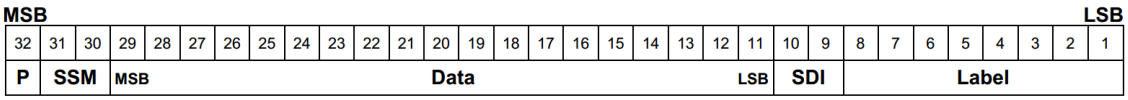
1 Introducción

Este proyecto es una pequeña parte de un proyecto mucho mayor que está realizando la empresa Orbital Aerospace y que consiste en diseñar y construir una plataforma hardware pre-certificable en la que poder hacer testeo de aplicaciones de un sistema de aviación o incluso para ir montada en un avión. Al tratarse de hardware para un sistema de aviónica y querer que sea pre-certificable hay que seguir las normas establecidas en la RTCA/DO-254 (1).

Esta plataforma abarataría mucho los costes de desarrollo de software crítico ya que se podrían probar y correr tests directamente sobre esta plataforma sin tener que depender de un banco de pruebas externo. Esta dependencia es una causa muy importante de la demora y coste de desarrollo en este campo ya que los grupos de desarrollo y testeo pierden un porcentaje muy elevado de tiempo del proyecto en conseguir acceso a un banco de pruebas.

En esta memoria se describe el diseño y la implementación de un transmisor ARINC-429 además de las múltiples simulaciones y procesos aplicados sobre él. El diseño de este transmisor se decidió implementar en una FPGA y programarlo en VHDL descomponiendo el IP Core en varios módulos. Estos módulos deberán pasar una serie de testbenches que garanticen el correcto funcionamiento de cada módulo por separado y del conjunto que forman para asegurarnos de que una vez forme parte de la plataforma hardware, ésta esté preparada para ser certificada.

ARINC 429 es un bus de datos muy común en aviónica. Se trata de un bus simplex sobre un par de cobre con un solo transmisor y hasta 20 receptores cuya codificación es RZ bipolar. Este bus tiene dos velocidades de funcionamiento: baja velocidad entre 12 - 14.5 Kbps, y alta velocidad alcanzando 100 Kbps. El formato de las palabras enviadas por este protocolo, como se puede ver en la Ilustración 1, es de palabras de 32 bits de los cuales los 8 primeros son de etiqueta, los bits 9 y 10 corresponden al Source/Destination Identifier (SDI), los bits comprendidos entre el 11 y el 29 son los datos, los bits 30 y 31 corresponden al Sign/Status Matrix (SSM) y el bit 32 es el de comprobación de paridad. De todos estos campos solo son obligatorios dos que son el bit de paridad y la etiqueta. El protocolo establece que tiene que haber un espacio de tiempo entre cada palabra enviada al que se denomina GAP.



ARINC 429 32 bit Word Format

Ilustración 1. Formato de palabra en ARINC 429

Este transmisor se implementará en una FPGA (Field Programmable Gate Array). Una FPGA es un circuito integrado que contiene bloques de lógica configurable lo que la convierte en una herramienta muy potente que nos permite diseñar e implementar hardware complejo de una manera relativamente sencilla. Este diseño se realiza gracias a un lenguaje de descripción hardware o HDL por sus siglas en ingles. Estos lenguajes permiten un alto nivel de abstracción a la hora de realizar un diseño hardware ya que son similares a los lenguajes de programación pero con la diferencia de que gracias al programa sintetizador podemos convertir esa programación en su circuito equivalente. En este caso vamos a emplear VHDL por su robustez y flexibilidad.

Ya existen en el mercado COTS de IP Cores transmisores A429 pero su precio es muy elevado. Dos ejemplos de COTS nos los dan las empresas LogiCircuit y DEVELeast que tienen a la venta cada una de ellas un IP Core transmisor ARINC 429 por 15000$ y 12000€ respectivamente. Como podemos ver es un precio difícil de asumir por pequeñas empresas y merece la pena embarcarse en este proyecto, no solo por el importante ahorro económico si no que pese al gasto de tiempo, los 4 meses de duración estimada podrían compensarse con creces con una sola venta de este IP Core al margen de las posibles ganancias del proyecto global.

2 Objetivos

Los objetivos marcados para este proyecto son implementar cuatro módulos VHDL junto con otros módulos del catálogo de Xilinx que permitan establecer una conexión entre una aplicación corriendo en el procesador y otro módulo ARINC 429 receptor. La conexión entre el procesador y el IP Core transmisor ARINC 429 se realizará a través de un bus AXI que también deberá ser implementado. Se empleará la herramienta proporcionada por Xilinx Vivado para el diseño del IP Core y para las simulaciones se usarán tanto Modelsim como Xsim, esta última integrada en el propio Vivado.

El IP Core Transmisor ARINC 429 estará compuesto por un módulo AXI esclavo, un módulo que contenga la lógica de control al que llamaremos Control Logic, una FIFO de 32 bits y 64 palabras, un módulo que sea un registro de desplazamiento y que además se encargue de calcular el bit de paridad de cada palabra, otro módulo que será encargado de controlar la duración de los gaps entre palabras enviadas y un último módulo que se encargue de codificar la salida del transmisor.

Una vez realizados estos módulos el Control Logic, la FIFO, el registro de desplazamiento, el controlador de gap y el codificador serán englobados en un mismo módulo al que llamaremos módulo transmisor. El esclavo AXI estará interconectado con este modulo transmisor y ambos a su vez formarán un módulo llamado AXI_A429_TX_v1_0.

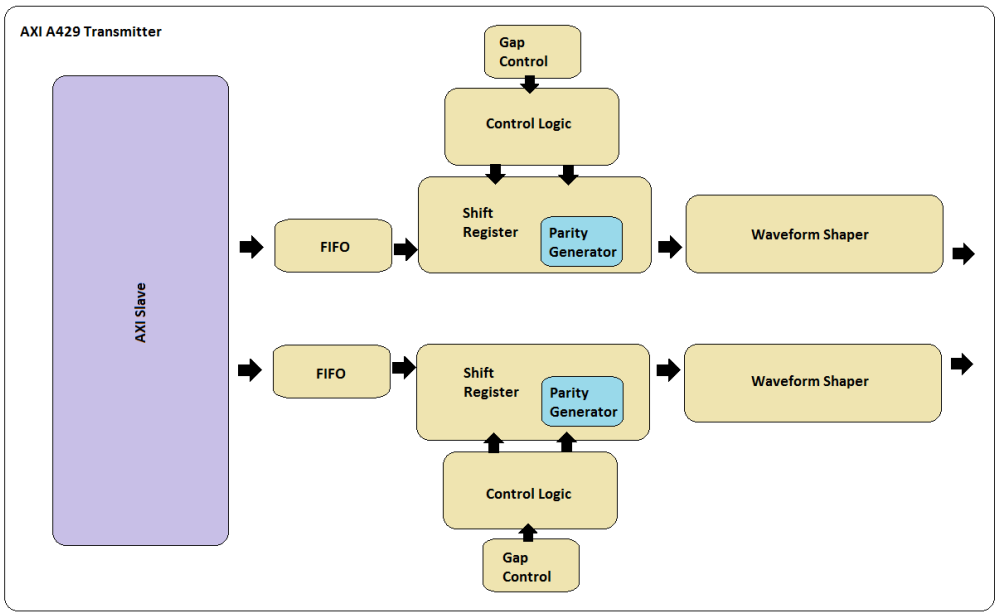


Ilustración 2. Diseño preliminar del transmisor A429 con los módulos a implementar

En la Ilustración 2 vemos el diseño preliminar del transmisor a implementar con sus módulos y su posible disposición. En la FIFO se guardarán los datos provenientes del esclavo AXI y el registro de desplazamiento será el encargado de recogerlos y enviarlos.

De forma paralela a la implementación de estos módulos en VHDL se irán implementando y ejecutando unas simulaciones (Testbenches) básicas que comprueben las funcionalidades de cada módulo a medida que se generen.

Una vez implementados los módulos y comprobadas sus funcionalidades básicas se procederá a la implementación de testbenches automáticos que comprueben la funcionalidad completa de cada módulo así como el conjunto de todos los módulos llamado módulo Transmisor.

Tras haber realizado satisfactoriamente las simulaciones se programará en la FPGA y será depurada (debug) mediante una herramienta incluida en Vivado. Se probará su funcionamiento y se mejorará el diseño hasta que la depuración funcione correctamente.

El objetivo último de este proyecto es enlazarlo con otro que consiste en la realización de un receptor ARINC 429, es decir, implementar ambos proyectos en la FPGA y conseguir mediante una aplicación corriendo en el procesador una comunicación entre el módulo transmisor y el receptor.

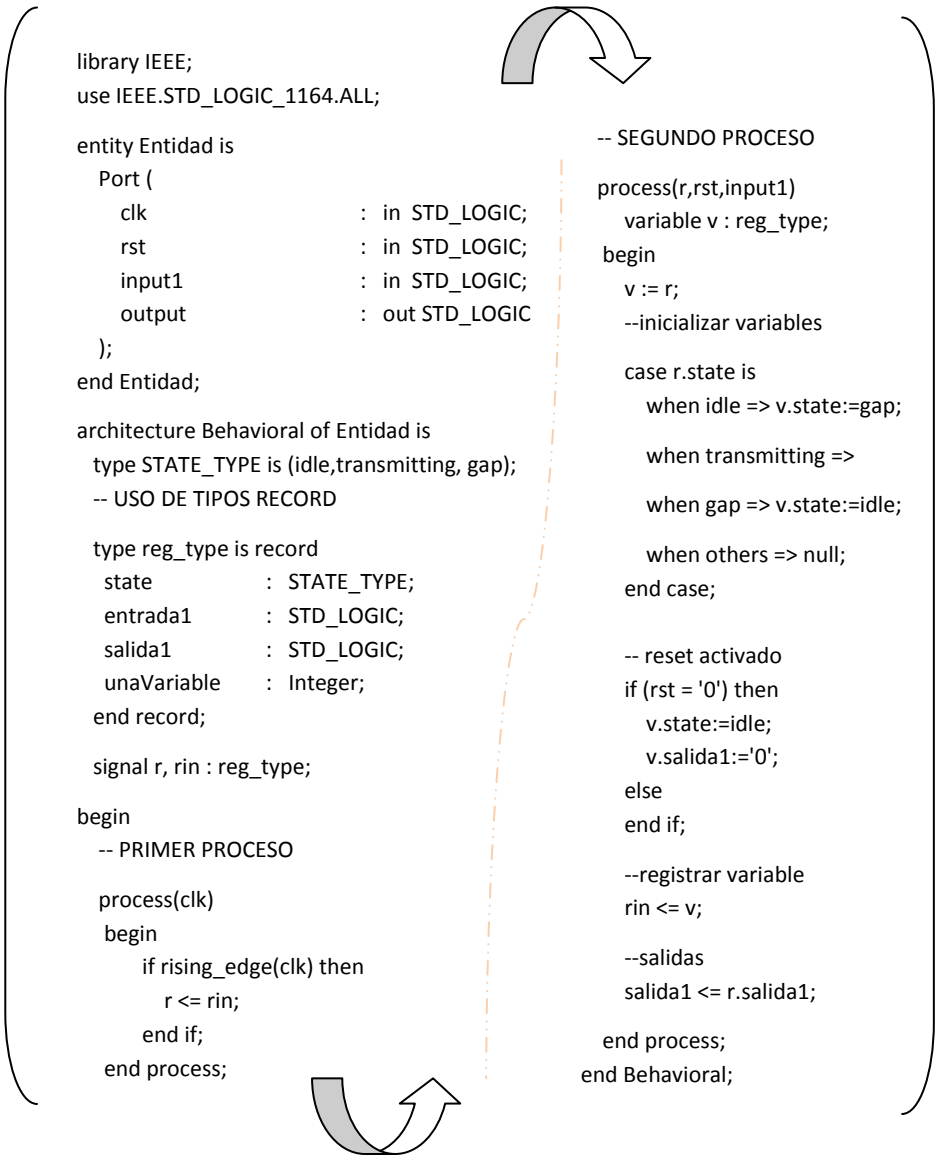
3 Metodología

3.1 Implementación de Módulos y Testbenches

Para la implementación del IP Core se generará código VHDL de los módulos diseñados empleando una metodología de diseño de dos procesos. Este método permite reducir tiempo de diseño y evita los típicos problemas inherentes a los sistemas diseñados con el método comúnmente empleado más parecido a la programación estructurada. Lo que se pretende conseguir con el uso de este método es un mecanismo uniforme de codificación de algoritmos, aumentar el nivel de abstracción, facilitar la comprensión del código, simplificar su depuración y conseguir ver de una manera más clara las conexiones en el análisis RTL. Es muy sencillo entender el funcionamiento de este método, solo hay que cumplir dos condiciones y una estructura. Las condiciones son emplear tipos record (record types) en todos los puertos y declaraciones de señal y solo utilizar dos procesos por módulo. Y la estructura a seguir se muestra en la Ilustración 3.

Las simulaciones se ejecutarán en modo comportamiento (behavioral) y una vez funcionen al 100% se pasará a ejecutar en modo funcional (functional) y temporal (timing). El método que se empleará para evaluar estas simulaciones será mediante archivos. Estos serán generados mediante un script Octave y contendrán los estímulos de entrada para cada módulo. El testbench se encargará de escribir un archivo de salida con las respuestas provocadas por cada módulo y se terminarán automáticamente al probar todos los estímulos posibles. Por medio de otro script Octave se comprobará que la respuesta del módulo a cada uno de los estímulos de entrada ha sido la que se esperaba y en base a esto el script decidirá si el módulo funciona o no correctamente.

Para introducir estos estímulos en cada módulo los testbench tendrán que ser capaces de leer los archivos de entrada y de escribir un nuevo archivo de salida. Esto lo haremos en todos los testbench siguiendo el modelo de la Ilustración 4.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Entidad is
  Port (
    clk                 :  in  STD_LOGIC;
    rst                 :  in  STD_LOGIC;
    input1              :  in  STD_LOGIC;
    output              :  out STD_LOGIC
  );
end Entidad;

architecture Behavioral of Entidad is
  type STATE_TYPE is (idle,transmitting, gap);
  -- USO DE TIPOS RECORD

  type reg_type is record
    state             :  STATE_TYPE;
    entrada1          :  STD_LOGIC;
    salida1           :  STD_LOGIC;
    unaVariable       :  Integer;
  end record;

  signal r, rin : reg_type;

begin
  -- PRIMER PROCESO

  process(clk)
   begin
       if rising_edge(clk) then
          r <= rin;
       end if;
   end process;
```

```
-- SEGUNDO PROCESO

process(r,rst,input1)
   variable v : reg_type;
begin
   v := r;
   --inicializar variables

   case r.state is
      when idle => v.state:=gap;

      when transmitting =>

      when gap => v.state:=idle;

      when others => null;
   end case;


   -- reset activado
   if (rst = '0') then
      v.state:=idle;
      v.salida1:='0';
   else
   end if;

   --registrar variable
   rin <= v;

   --salidas
   salida1 <= r.salida1;

 end process;
end Behavioral;
```

Ilustración 3. Estructura tipo del método de diseño de dos procesos

En la Ilustración 3 vemos la estructura de un módulo diseñado con el método de dos procesos. El primer proceso solo contiene en su lista de sensibilidad el reloj y dentro simplemente en el flanco de subida actualiza las variables. En el segundo proceso, como vemos, su lista de sensibilidad contiene la variable r de tipo record, el reset y las entradas del módulo. Entrando en el proceso vemos que dependiendo de en qué estado se encuentre el módulo podremos programar un comportamiento u otro. En este caso simplemente del estado de espera pasará al estado de gap y viceversa. Al final de este proceso si el reset esta activado se inicializa todo y por último se actualizan las variables y las salidas. Como vemos con este método solo asignamos valores registrados a las salidas y no variables (salida1 <= r.salida1 en vez de salida1 <= v.salida1) lo que evita muchos errores de implementación.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
use std.textio.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Entidad_autoTestbench is
end Entidad_autoTestbench;

architecture Behavioral of Entidad_autoTestbench is
  component Componente is
    Port (
          ACLK      : in std_logic;
          ARESETN : in std_logic;
          input1    : in STD_LOGIC_VECTOR(31 downto 0);
          output1   : out std_logic
    );
  end component;

  signal clk                  : STD_LOGIC;
  signal resetn               : STD_LOGIC;
  signal input1_signal        : STD_LOGIC_VECTOR(31 downto 0);
  signal output1_signal       : STD_LOGIC;
  signal I_want_to_finish_autoTestbench :STD_LOGIC:='0';
begin

Componente_inst : Componente
  PORT MAP (
    ACLK          => clk,
    ARESETN       => resetn,
    input1        => input1_signal,
    output1       => output1_signal
  );

generate_reset : process
  begin
    resetn <= '0';
    wait for 100 ns;
    resetn <= '1';
    wait for 1ms;
    I_want_to_finish_autoTestbench<='1';
    wait;
  end process;

generate_clock : process
  begin
    clk <= '1';
    wait for 1 ns;
    clk <= '0';
    wait for 1 ns;
  end process;
writing_in_file : process
    function value_to_bin(c : character) return std_logic is
    begin
      if(c='1') then return ('1');
      else return ('0');
      end if;
    end;
    file DataInput: TEXT open read_mode is "Path\documento.txt";
    variable DataLine : line;
    variable v : std_logic;
    variable n_columnas_archivo:Integer:=32;
    variable n_filas_archivo:Integer:=32;
  begin
   for i in 1 to n_filas_archivo loop
    readline (DataInput,DataLine);
    for j in 1 to n_columnas_archivo loop
      v:= value_to_bin(DataLine(n_columnas_archivo+1-j));
      input1_signal(j-1) <= v;
    end loop;
    wait for 20 ns;
   end loop;
   wait;
  end process;

reading_from_file : process
  file OutputFile : TEXT open write_mode is "Path\documento2.txt'
  variable outLine : line;
  begin
    wait until output1'EVENT;
    write(outLine,output1);
    writeline(OutputFile,outLine);
    if I_want_to_finish_autoTestbench='1' then
      file_close(OutputFile);
      assert false report "end of simulation" severity failure;
    else
    end if;
  end process;
end Behavioral;
```

Ilustración 4. Estructura tipo del método empleado para realizar testbenches automáticos

En la Ilustración 4 podemos ver la estructura seguida en todos los testbenches automáticos realizados en este proyecto. Al comienzo del testbench hay que añadir las librerías necesarias para la escritura que son IEEE.STD_LOGIC_TEXTIO, y std.textio y en el caso de querer escribir alguna variable de tipo Integer antes deberá ser pasada a STD_LOGIC_VECTOR para lo que necesitaremos la librería IEEE.NUMERIC_STD. El siguiente paso es implementar los códigos que desempeñan las funciones de reloj y reset. Por último

leemos los datos escritos en un archivo línea a línea y almacenamos los caracteres, predeciblemente ceros y unos, en la variable de entrada de nuestro componente. Mantenemos ese valor en la variable durante, en este ejemplo, 20 nanosegundos y la cambiamos a la siguiente cadena escrita en nuestro archivo. Y este proceso se repite hasta el número de líneas especificado. Suponemos que nuestro componente reacciona a los estímulos y lo que hacemos es escribir la variable de salida en un segundo archivo. Esto ocurrirá hasta que cambie la salida y haya pasado 1 milisegundo ya que entonces el testbench parará al saltar la alerta de fallo grave.

La RTCA/DO-254 establece un proceso de elaboración de hardware de tal manera que nos aseguremos de que el resultado será hardware fiable. Primero hay que realizar una captura de requisitos del hardware y en base a estos se realiza un diseño conceptual y posteriormente un diseño detallado. Una vez atravesadas estas tres fases se pasa a la implementación y posterior verificación o testeo y por último, una vez testeado con éxito pasa a producción.

3.2 Agilismo Aplicado al Proyecto

Para la realización del proyecto también se ha puesto en práctica una metodología de desarrollo ágil (2) derivada de la denominada Scrum (3). Se trata de un conjunto de buenas prácticas diseñadas para agilizar y optimizar el trabajo. En Scrum se definen unos roles que son el gestor principal (Scrum Master), cliente (Product Owner) y el equipo (Team). También se dividen las tareas en sprints que duran entre 3 y 4 semanas durante los cuales el equipo no puede ser molestado con actualizaciones de requisitos sobre el producto. Se realizan reuniones todos los días de 15 minutos para que cada miembro del equipo sepa lo que ha hecho cada uno de sus compañeros y lo que tiene pensado hacer ese mismo día. Por esto último se realizan al comienzo de la jornada. Este procedimiento se muestra de manera gráfica en la Ilustración 5. Además de esta reunión diaria, se realiza otra de mayor duración cada semana para hablar de los problemas y aciertos de la semana y otra reunión al comienzo de cada sprint para planear y definir las tareas del siguiente.

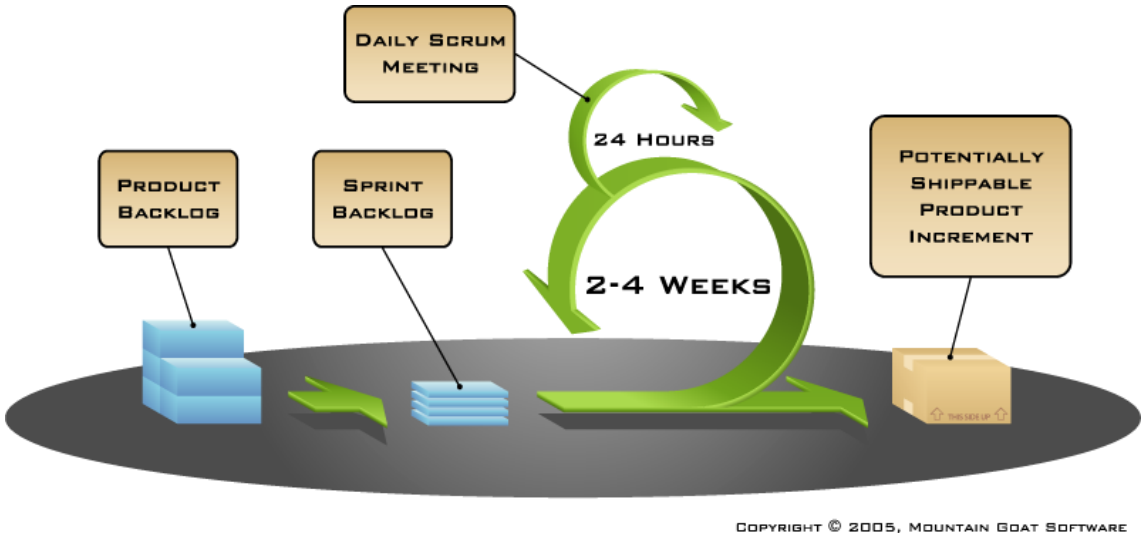


Ilustración 5. Definición gráfica de la distribución del tiempo en un proyecto llevado con Scrum (4)

La metodología de trabajo aplicada a este proyecto ha sido realizada siguiendo un patrón parecido al definido por Scrum, sin reuniones diarias pero sí semanales. Al tratarse de un proyecto de I+D interno de la empresa, el rol de cliente fue asumido por el director de I+D. El rol de gestor principal fue adjudicado al jefe del proyecto al contar con una visión global del proyecto y mayor conocimiento. Como en todo proyecto llevado mediante Scrum, al comienzo se hizo un listado con las tareas a realizar a lo largo de 4 meses y se estableció una duración aproximada a cada tarea. Se realizó un Gant y se fijaron 4 fases.

Una primera fase de familiarización con las herramientas de uso y aprendizaje de protocolos de aviónica y lenguaje de descripción hardware elegido (VHDL). Además esta primera fase incluía el aprendizaje de procesos de calidad realizados por la empresa así como de metodologías de implementación y trabajo.

La segunda fase consistió en el diseño e implementación de los módulos que componen el transmisor A429 empezando por el registro de desplazamiento pasando por el controlador de gap y el codificador de señal y terminando por el bloque con la lógica de control. A esto hay que añadir los testbenches de comprobación básica.

A continuación la tercera fase y la más larga, engloba la realización de los testbenches automáticos así como los scripts Octave de comprobación de resultados. Esta fase lleva implícita la corrección de errores en módulos además de la ejecución de todos los métodos de simulación necesarios para evaluar el funcionamiento del sistema completo.

La cuarta y última fase consiste en la programación del IP Core en una FPGA y probar su funcionamiento contra un receptor A429 implementado en la misma. También se comprobarán el flujo de datos en tiempo real en el transmisor implementado. En esta última fase adquirirá un papel protagonista la redacción de la documentación necesaria.

4 Desarrollo e Implementación

4.1 Introducción

A la hora de comenzar este proyecto se decidió por su robustez usar el lenguaje de descripción hardware VHDL frente a la alternativa Verilog. Al no tener experiencia con esta clase de lenguajes, la primera tarea fue la de familiarizarse y aprender las reglas y metodologías básicas de implementación como las ya descritas en el apartado 3.1. A esta primera tarea hay que añadirle la de documentación sobre el protocolo ARINC 429 y las características técnicas necesarias de un transmisor y receptor en este protocolo.

4.2 Definición Técnica de un Transmisor ARINC 429

El transmisor ARINC 429 contará con dos líneas de transmisión que se ajusten a la especificación y podrán ser usadas como caminos redundados o independientes. Cada línea de transmisión tendrá dos posibles velocidades de transmisión:

- Alta velocidad → 100 Kbps
- Baja velocidad → 12.5 Kbps

El interfaz con el procesador será un esclavo AXI, y los interfaces externos del transmisor podrán conectarse directamente a un controlador de línea comercial como por ejemplo HOLTIC HI-318x, HI-8597 o Intersil HS3182. En los Objetivos ya hemos podido ver un diseño preliminar al que le faltaba contenido al esclavo AXI. En cada transmisor habrá un módulo FIFO con capacidad para 64 palabras de 32 bits.

Para implementar este transmisor con dos líneas de transmisión, se implementará una línea con un esclavo AXI y una vez terminada se duplicará el diseño y se englobarán ambos en un único IP Core.

En la Ilustración 6 vemos que el módulo esclavo AXI contiene tres registros que están directamente relacionados con el resto de módulos del transmisor. El módulo que contiene la lógica de control necesita conocer la información escrita en el registro de control del AXI para poder actuar en consecuencia y a su vez el registro de estatus deberá reflejar el estado actual del transmisor y esa información deberá recibirla del mismo módulo con la lógica de control. En este diseño de definición técnica no entramos en mayor profundidad a la hora de especificar la interconexión entre módulos.

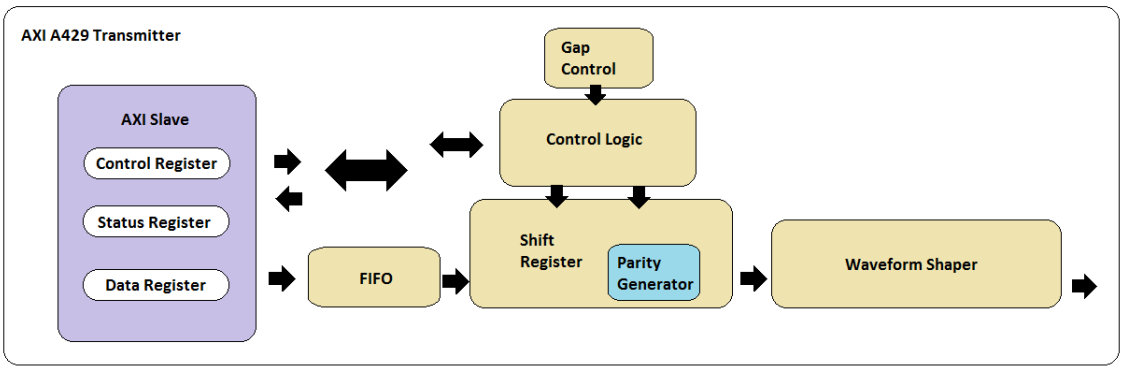


Ilustración 6. Diseño preliminar con interconexiones entre registros del AXI y los módulos del transmisor

Para la FIFO usaremos un IP Core del catalogo de Xilinx llamado FIFO Generator. Con este módulo podemos generar la FIFO que necesitamos fácilmente a través de una GUI como vemos en la Ilustración 7.

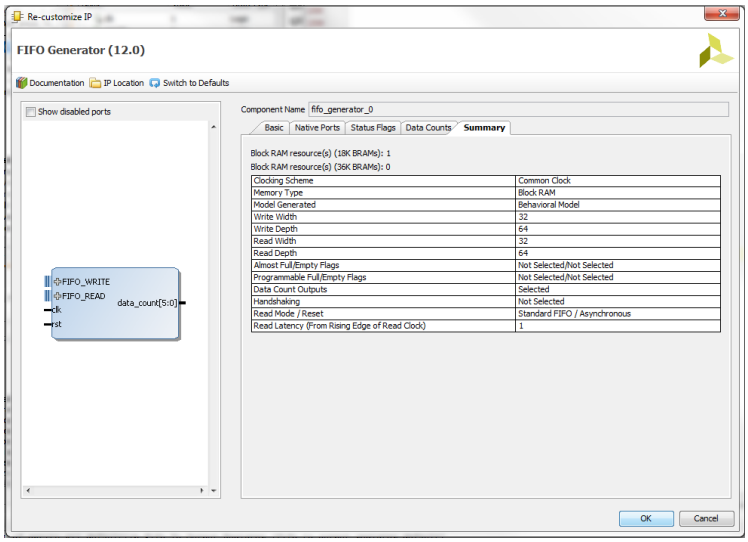


Ilustración 7. GUI de configuración del módulo FIFO Generator

En dicha ilustración, la pestaña seleccionada es la de resumen y vemos que las especificaciones de nuestra FIFO coinciden con las especificadas anteriormente. Capacidad de albergar 64 palabras de 32 bits.

Las especificación del protocolo ARIN 429 define que las señales de salida estarán codificadas con el código RZ Bipolar con unos parámetros de tolerancia indicados en la Tabla 1 y Ilustración 8.

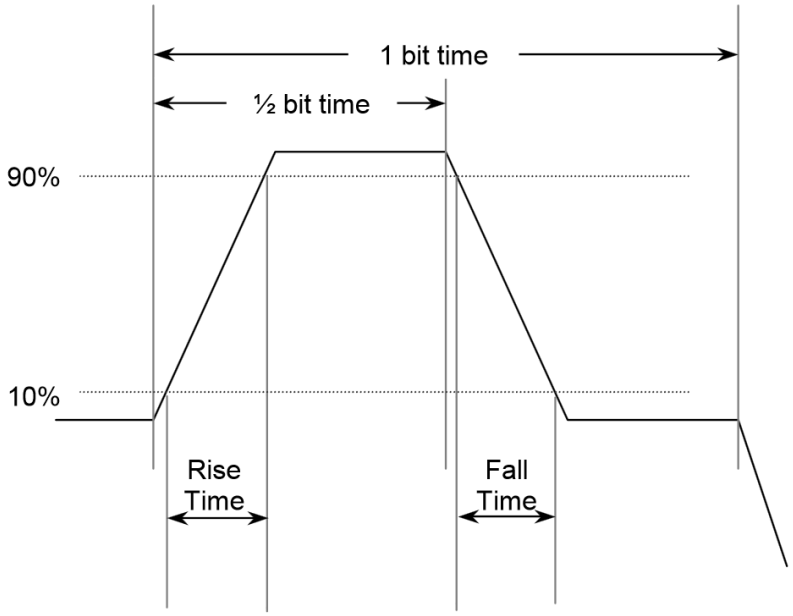


Ilustración 8. Gráfico de pulso para definición de las tolerancias de la línea de salida (5)

La Ilustración 8 puede servirnos de apoyo gráfico para entender los rangos de tolerancia admitidos según la especificación ARINC 429. Como vemos el tiempo de flanco de subida se considera a partir del 10% del valor de amplitud y hasta el 90% del mismo al igual que pasa con el periodo de flanco bajada.

En la Tabla 1 podemos ver los datos concretos de tolerancia especificados por la norma. En este proyecto hemos elegido una frecuencia de operación a baja velocidad de 12.5 Kbps y por tanto el periodo de bit será de 80 µseg con una tolerancia de 2 µseg.

Parámetro	Operando a alta velocidad	Operando a baja velocidad
Tasa de bits	100 Kbps ± 1%	12 - 14.5 Kbps
Tiempo 1 bit	10 µseg ± 2.5%	(1/Tasa de bit) µseg ± 2.5%
Tiempo ½ bit	5 µseg ± 5%	(1 bit time/2) ± 5%
Tiempo de flanco de subida	1.5 µseg ± 0.5 µseg	10 µseg ± 5 µseg
Tiempo de flanco de bajada	1.5 µseg ± 0.5 µseg	10 µseg ± 5 µseg

Tabla 1. Tolerancias definidas en la especificación ARINC 429 (5)

Entre envío de palabras, el transmisor tiene que dejar un tiempo de espera manteniendo su salida a nivel lógico bajo. A este tiempo de espera le llamamos gap y deberá

ser de por lo menos cuatro veces el periodo de bit. De esta manera nos aseguramos el no necesitar el envío de una señal de reloj adicional para sincronizar transmisor y receptor.

4.3 Definición Concreta del Interfaz Esclavo AXI

El interfaz esclavo AXI es el módulo que permite la comunicación entre el transmisor ARINC 429 y el software ejecutado en el procesador al otro lado del AXI. Este módulo contiene tres registros y como vimos en la Ilustración 6 son un registro de control, un registro de estado y un registro de datos.

Vivado nos permite generar un esclavo AXI personalizado así que nos valdremos de la GUI para crear el modelo básico y ampliar lo que sea necesario.

En la Ilustración 9 vemos el primer paso para crear un periférico AXI. Si seguimos la GUI podremos especificar el tamaño, la profundidad y la cantidad de registros que queremos incluir en nuestro periférico AXI. Además nos da la posibilidad de establecer nuestro módulo como maestro o esclavo. Para este proyecto necesitamos un periférico AXI esclavo de cuatro registros de 32 bits y de una sola palabra de profundidad. Establecemos cuatro registros al ser el mínimo permitido por la GUI de Vivado aunque solo usaremos tres. Un registro de control, un registro de estado y por último un registro de datos.

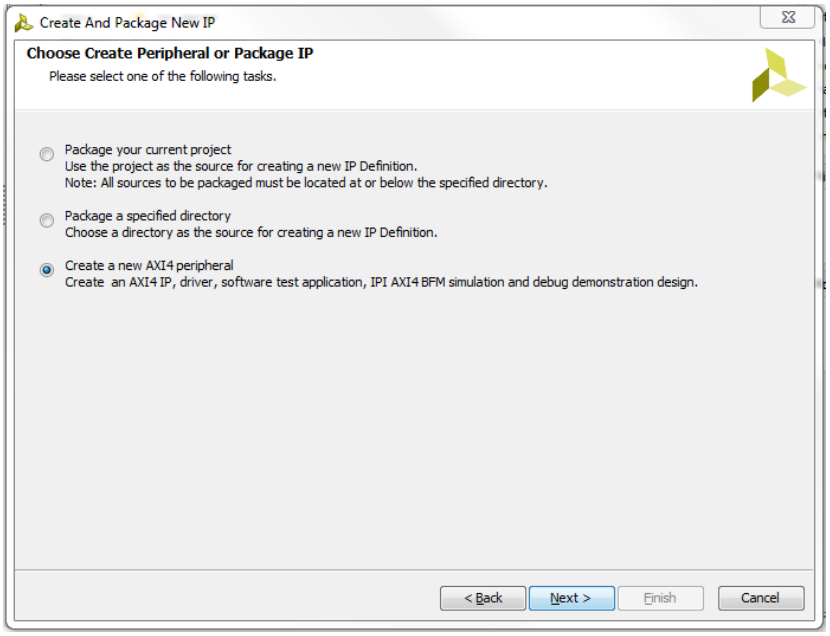


Ilustración 9. GUI de generación de esclavo AXI básico

En la Tabla 2 tenemos descrita la funcionalidad de cada bit del registro de control. El bit 0 activa o desactiva el transmisor, es decir actúa como si estuviera reseteado. Con el bit 1 a 0 y el bit 0 a 1 lo que conseguimos es que los datos provenientes del esclavo AXI se almacenen en la FIFO pero sin transmitir. Los bits de paridad determinan la paridad a construir al igual que el bit de tasa de bit determina la frecuencia de transmisión. Los gap se controlan con los bits del 4 al 7. Si el bit 4 está a 1, la duración del gap vendrá determinada por el número binario formado por los bits del 5 al 7 siendo el 5 el LSB, obteniendo un rango del 0 a 7 periodos de bit posibles para la duración del gap.

Bit	Función	Estado	Descripción
CR0	Estado del transmisor	0	Desactivar
		1	Activar
CR1	Estado de transmisión	0	Detener transmisión
		1	Activar transmisión
CR2	Activar calculo de paridad	0	Desactivar
		1	Activar
CR3	Tipo de paridad	0	Paridad impar
		1	Paridad par
CR4	Controlador de Gap	0	4 veces el periodo de bit (Por defecto)
		1	Duración especificada en bits CR5 - CR7
CR5 - CR7	Periodo de Gap	0 - 7	GAP = Valor * Periodo de bit
CR8	Tasa de bit	0	Baja velocidad 12.5 Kbps
		1	Alta velocidad 100 Kbps

Tabla 2. Bits y funcionalidad desempeñada por el registro de control del esclavo AXI

En la Tabla 3 podemos ver como este registro refleja en cada momento el estado del transmisor. Si esta activado, transmitiendo datos o esperando que le lleguen del procesador y el estado de la FIFO en cada momento.

Bit	Función	Estado	Descripción
ST0	Estado del transmisor	0	Desactivado
		1	Activado
ST1	Acción del transmisor	0	Esperando datos
		1	Transmisión en curso
ST2	FIFO medio llena	0	No está medio llena
		1	Está medio llena
ST3	FIFO llena	0	FIFO sin llenar
		1	FIFO llena

Tabla 3. Bits y funcionalidad desempeñada por el registro de estado del esclavo AXI

Por último contaremos con un registro de datos en el cuál se almacenarán 31 bits provenientes del bus AXI y en última instancia del software. Estos datos vendrán dados según el formato establecido en ARINC-429 que ya vimos en la Ilustración 1.

4.4 Uso del IP Core

Tras desactivar el reset poniéndolo a 1, el transmisor A429 estará listo para recibir la configuración y los datos para transmitir.

El primer paso es activar el transmisor a través del bit del registro de control correspondiente (bit CR0, Tabla 2) y lo mismo con el resto de parámetros permitidos por dicho registro que son el control de paridad, la tasa de bit y el periodo de gap.

Mientras el bit CR0 esté activo el transmisor almacenará datos en la FIFO y hasta que el bit CR1 no se active, no se transmitirá ningún dato. Esta funcionalidad permite por ejemplo redundar los datos con varios transmisores controlando el desfase entre los diferentes caminos y adaptándolo.

El transmisor podrá cambiar de velocidad al finalizar la transmisión de una palabra pero no durante. De esta manera nos aseguramos que la velocidad no produce errores en la transmisión aunque la aplicación no debería cambiar de velocidad una vez instalada en un bus ARINC-429.

4.5 Módulos e Implementación

Como ya hemos dicho en los Objetivos el transmisor ARINC-429 finalmente estará compuesto por 5 módulos: Un esclavo AXI, una FIFO, un Control Logic, un Gap Control, un Shift Register and Parity Check y un Waveform Shaper. En este apartado serán descritos así como su implementación.

4.5.1 Esclavo AXI

Este módulo es el encargado de leer los datos del bus AXI que se dirigen al módulo transmisor y actualizar sus registros de control y datos en base a estos mensajes originados en el microprocesador. Cada vez que actualiza su registro de datos este envía una señal de un ciclo de reloj de duración al Control Logic para hacerle saber que hay un dato nuevo y que tiene que escribirlo en la FIFO.

Para la realización de este módulo nos hemos ayudado de la GUI de Vivado ya que nos proporciona una herramienta para generar automáticamente un módulo muy semejante al que necesitamos para este proyecto. Como ya vimos en la Ilustración 9 así comienza el proceso de creación del esclavo.

En la Ilustración 10 se muestra el segundo paso de la GUI para la creación de un esclavo AXI. Simplemente tenemos que añadir el nombre del nuevo IP y el sitio en el que queremos albergarlo.

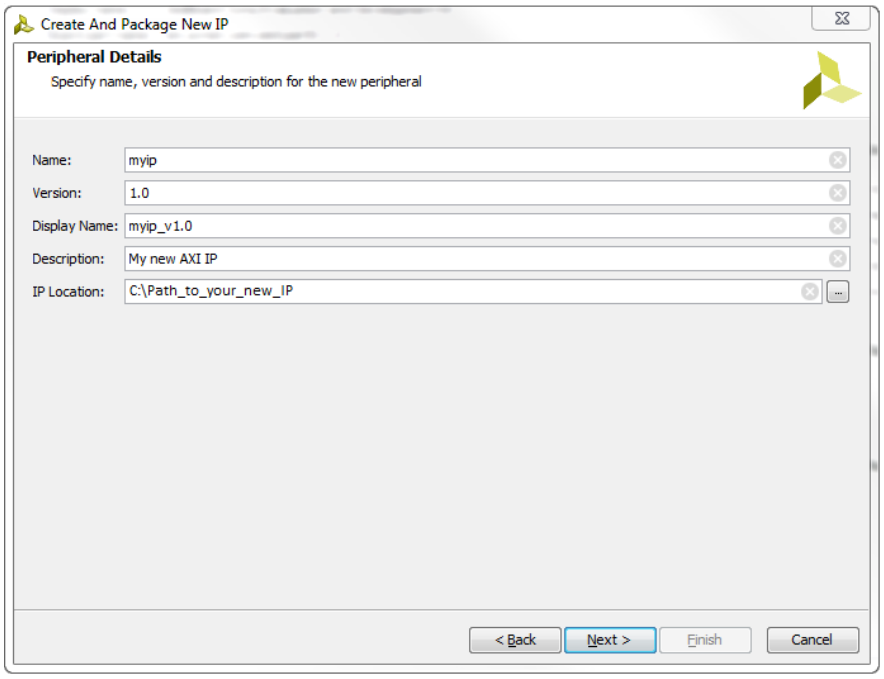


Ilustración 10. Selección del nombre y path del AXI esclavo

En la Ilustración 11 seleccionamos el tipo de interfaz AXI que necesitamos. En nuestro caso hemos seleccionado un interfaz AXI Lite esclavo con cuatro registros de 32 bits y con capacidad para una sola palabra.

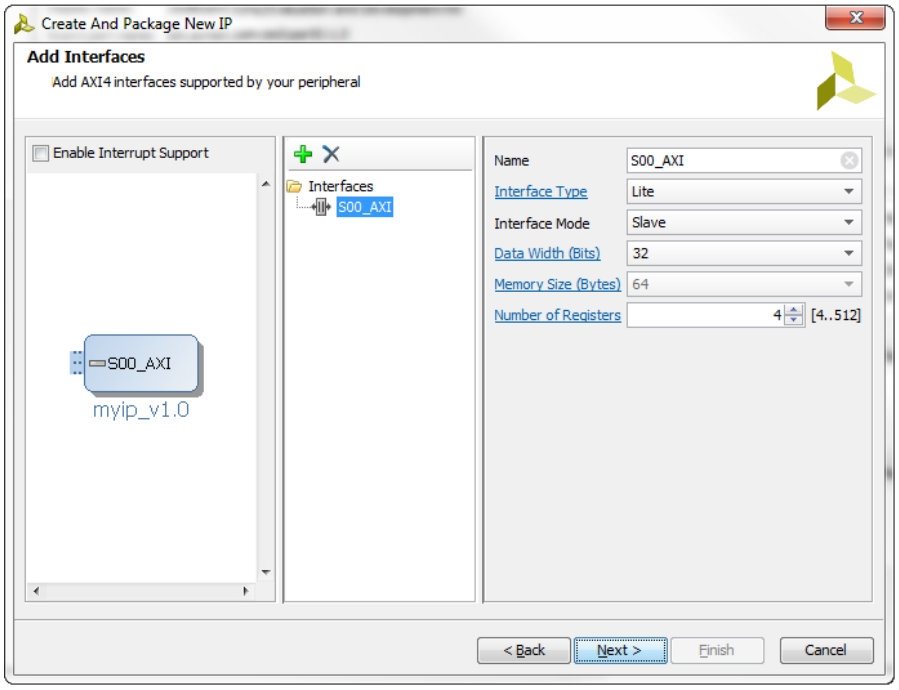


Ilustración 11. Definición del tipo de interfaz AXI a generar

En la siguiente ventana no tenemos más que seleccionar la opción de editar y ahí es cuando se nos creará un proyecto de edición del interfaz AXI con el bloque generado en código VHDL el cual podremos editar.

A este bloque le añadimos una entrada y tres salidas. La entrada estará directamente asociada al registro de estado de este módulo. Y dos de las tres salidas se corresponderán con los registros de control y datos internos del módulo. La tercera salida, como ya hemos nombrado antes, será la encargada de indicar cuándo se han escrito nuevos datos en el registro de datos del módulo, provenientes del microprocesador a través del bus AXI.

En la Ilustración 12 podemos ver como el módulo AXI se ha creado correctamente y mediante unas pequeñas modificaciones en el VHDL para añadir las entradas y salidas extras, resaltadas en azul, además de la lógica para que la salida to_control_logic_data_update funcione correctamente nos queda el bloque lógico de la ilustración.

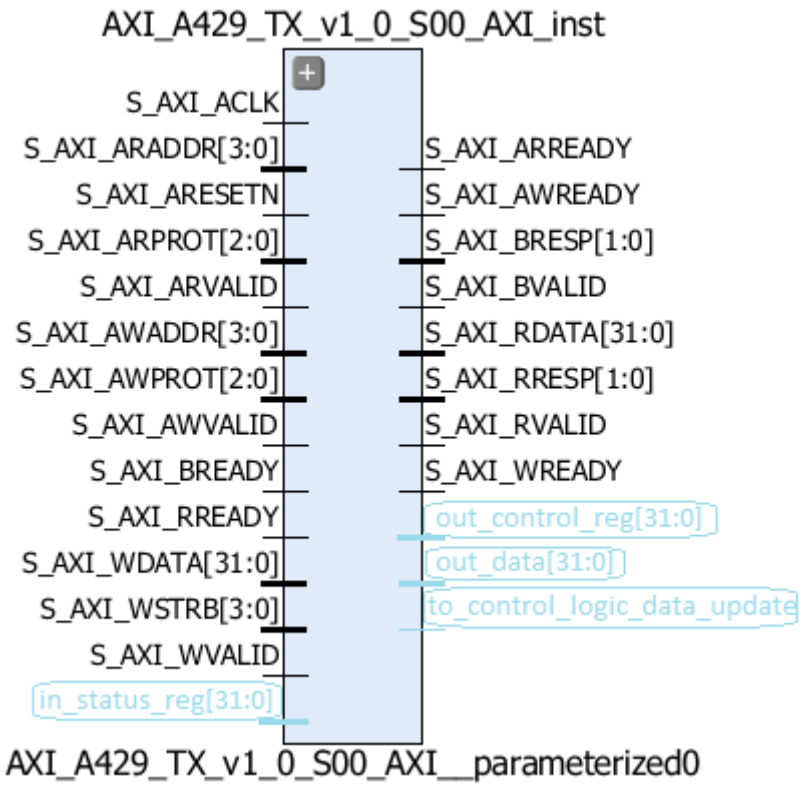


Ilustración 12. Módulo esclavo AXI personalizado en formato caja negra

En la Ilustración 13 vemos el esquemático de interconexión entre los registros del AXI y las salidas personalizadas añadidas a posteriori. La entrada del registro de estatus no va a parar a un registro si no que va directamente a un multiplexor. Cuando el microprocesador pida los datos del estado del transmisor se seleccionará esa entrada del multiplexor y los datos estarán listos para ser leídos directamente en la salida S_AXI_RDATA.

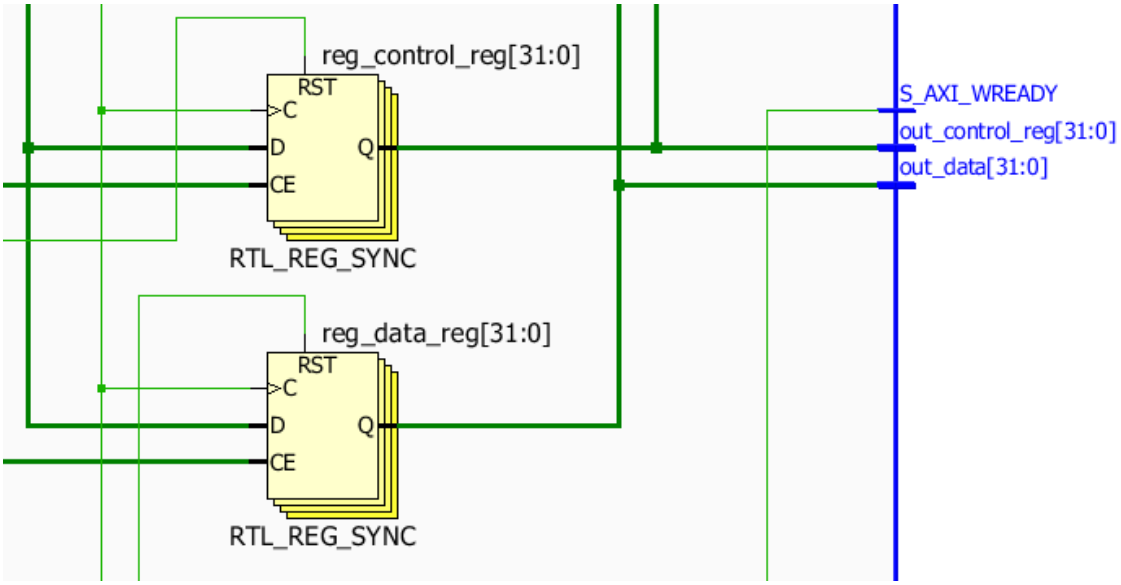


Ilustración 13. Interconexión interna entre registros y salidas adicionales

4.5.2 FIFO

La FIFO utilizada en este proyecto es creada mediante la GUI citada anteriormente que puede ver en la Ilustración 7. Siguiendo los pasos del módulo FIFO Generator del catálogo de Xilinx obtendremos la FIFO que necesitamos.

El módulo FIFO es el módulo encargado de almacenar los datos provenientes del AXI para su posterior envío. Necesitaremos una FIFO con capacidad para 64 palabras de 32 bits cada una.

Las características de esta FIFO son muy concretas. El reset es activo en flanco de subida y permanece en estado de reset tres ciclos de reloj. Para escribir en ella cuenta con una entrada wr_en en la que al poner un 1 los datos que se encuentren en la entrada de datos de la FIFO serán almacenados en su registro interno durante ese mismo ciclo de reloj.

En la Ilustración 14 vemos que una vez activado el reset tienen que pasar tres flancos de subida de reloj hasta que la FIFO vuelve a aceptar señales de lectura y escritura.

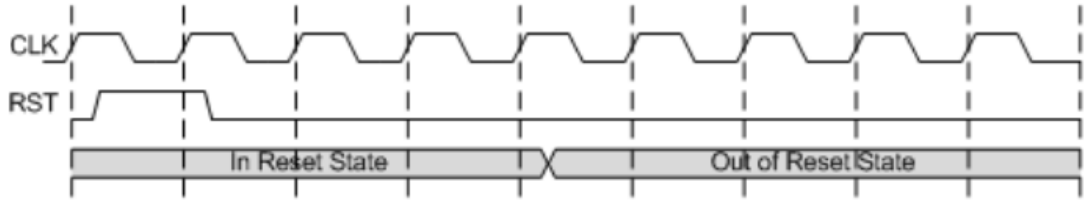


Ilustración 14. Visualización del funcionamiento de la FIFO (6)

Para leer datos, tiene otra entrada rd_en que puesta a 1 actualizará con el siguiente dato almacenado en su registro interno la salida de datos en el siguiente ciclo de reloj.

Además cuenta con salidas adicionales que proporcionan información sobre su estado. Una salida indicando si está llena y otra que indica si está vacía, full y empty. También cuenta con una salida que indica el número de datos que tiene almacenados. De esta última salida solo utilizamos el bit MSB ya que lo único que nos interesa es saber cuándo se encuentra la FIFO con más de 31 palabras, es decir la mitad o más de la mitad de su capacidad para actualizar el bit del registro de estatus correspondiente.

La Ilustración 15 muestra el módulo FIFO con todas sus entradas y salidas. Como vemos la entrada y salida de datos es de 32 bits, correspondientes al formato ARINC 429, y la señal data_count es de 6 bits ya que como mucho puede haber 65 palabras. Cuando la FIFO se llena, el bit de FIFO full se pone activo y la señal data_count marca "000000".

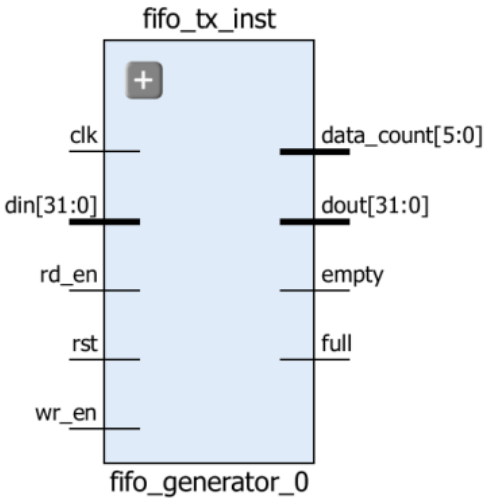


Ilustración 15. Módulo FIFO en formato caja negra

4.5.3 Control Logic

Este bloque es el que traduce el registro de control del AXI en órdenes para cada módulo del transmisor. Siguiendo el modelo de dos procesos de la Ilustración 3, se incluyen en este módulo dos estados, el estado de espera (idle) y el estado activado (enabled). Primero con la señal de reset se inicializa al estado de espera y se queda en ese estado hasta que el primer bit del registro de control (CR0 de la Tabla 2) se active.

Si este bit está activo pasará al estado de activado y comprobará si la señal del AXI que indica que se han actualizado datos está activa y si la FIFO está vacía. En este caso escribirá en la FIFO dicho dato. Mientras permanezca en este estado comprobará estas dos condiciones cada ciclo de reloj y repetirá el mismo proceso.

Para escribir este dato en la FIFO lo que hace es coger la salida de datos del AXI y pasarla a la entrada de la FIFO además de activar su bit fifo_wr.

Todo esto se llevará a cabo si no se ha producido un reset desde hace menos de 3 ciclos de reloj, porque en tal caso tendremos que esperar los ciclos correspondientes para poder escribir en la FIFO como hemos visto en la Ilustración 14. Lo que hará en el caso de que se haya producido el reset hace menos de 3 ciclos de reloj es poner en la entrada de datos a la FIFO los datos y esperar los ciclos restantes para activar la entrada a la FIFO wr_en. Esto lo hacemos así para prevenir que en el improbable caso de que lleguen dos datos en ese rango de tres ciclos de reloj poder escribir ambos en la FIFO.

Además de controlar la escritura de la FIFO hace de intermediario entre el AXI y el resto de módulos distribuyendo la información del registro de control entre el módulo controlador de Gap y el módulo que contiene un registro de desplazamiento y control de paridad. También recibe de este último una señal indicando cuando empieza la transmisión de bits al exterior del transmisor.

En el registro de estado (descrito en la Tabla 3) escribe en los primeros cuatro bits de la siguiente manera "3210", siendo el bit 3 el MSB. En la posición 3 la señal proveniente de la FIFO que indica si está llena (fifo_full). En la 2 según si el sexto bit de la señal data_count de la FIFO está activado o la FIFO está llena se escribirá un 1 o si no se da ninguna de las condiciones se escribirá un 0. Y en las posiciones 1 y 0 se escribirán la señal proveniente del módulo registro de paso indicando si el módulo se encuentra transmitiendo y la señal interna de este módulo indicando si se encuentra en estado de espera (0) o activado (1) correspondientemente.

Al módulo registro de desplazamiento y comprobación de paridad le envía un vector de 4 bits con el siguiente formato: "Parar transmisión, Tasa de bit, paridad, tipo de paridad".

El primer bit se desactivará siempre que en el registro de control los bits CR0 y CR1 estén activados. Y los otros tres bits son directamente interconectados con los bits CR8, CR2 y CR3 correspondientemente. Además de este vector de 4 bits se envía una señal adicional que indica si se ha desactivado el Gap indicando en el registro de control que tenga duración 0. Ya que en este caso la lógica de envío y carga de datos del módulo cambia.

También se envía un vector de 5 bits al módulo controlador de Gap con el siguiente formato: "Tasa de bit, periodo de bit (3 bits), Gap especificado". Este vector está directamente conectado a los bits CR8 - CR4 correspondientemente.

En la Ilustración 16 vemos las entradas y salidas del bloque de control con nombres que representativos. Las salidas comienzan por to_ seguidas del módulo correspondiente y las entradas por from_ y el módulo al que se dirigen excepto fifo_count, fifo_full y fifo_wr que son suficientemente representativas sin necesidad de añadidos.

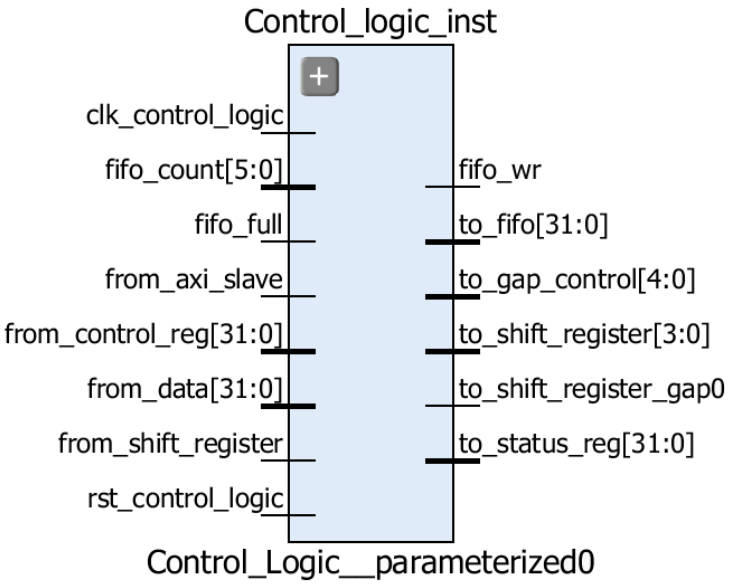


Ilustración 16. Módulo Control Logic en formato caja negra

4.5.4 Gap Control

Este bloque es el encargado de tomar el control al finalizar la transmisión de cada palabra y mantener un tiempo determinado un silencio en la línea de transmisión.

Llamamos silencio o estado nulo a que en la línea de transmisión de niveles lógicos altos y la línea de bajos permanezcan a 0V durante un período de tiempo múltiplo del periodo de bit.

La duración del periodo nulo viene determinada por los bits CR8 - CR4 de los cuales el CR4 es el que determina si la duración es de cuatro veces el periodo de bit o viene especificada en los 3 bits siguientes (véase Tabla 2 para conocer la configuración concreta de los bits).

El bit restante es el que define la frecuencia de trabajo del conjunto de módulos y por tanto es necesario conocerla para poder conocer el número de ciclos de reloj equivalente a un periodo de bit. Para calcular el número de ciclos de reloj total que tiene que durar el Gap se multiplica el número de ciclos de reloj de un periodo de bit por el número de periodos de bit que se han especificado y en el siguiente ciclo de reloj este resultado se divide por el periodo de reloj en nanosegundos. El hecho de separar la multiplicación y división en dos ciclos de reloj no es trivial ya que si lo intentamos hacer en un solo ciclo el retardo introducido por la división es muy grande. En este caso era tan grande que el retraso introducido era mayor que el periodo de reloj y por tanto impediría que el módulo funcionara correctamente.

La Ilustración 17 muestra las señales pertenecientes al Gap Control. Las señales from_shift_register y to_shift_register son las señales que emplean los módulos de registro de paso y controlador de gap para indicar quién toma el control de la salida del transmisor. Es decir cuando el registro de paso termina de transmitir una palabra, y el gap esta activado, el registro de paso pone la salida from_shift_register activa. En ese momento el Controlador de gap calcula los ciclos que tiene que cumplir y empieza a contar ciclos de reloj hasta que se

termina el gap. Una vez llegado al número de ciclos especificado activa la señal to_shift_register durante un solo ciclo de reloj y pasa de nuevo al estado de espera (idle).

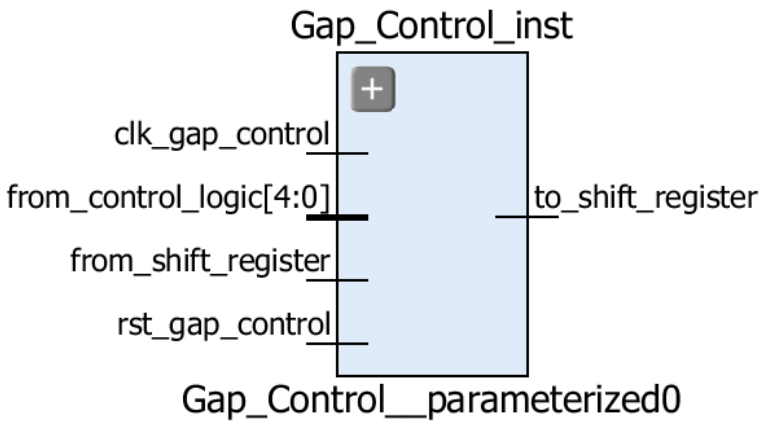


Ilustración 17. Módulo Gap Control en formato caja negra

4.5.5 Shift Register and Parity Check

El registro de desplazamiento con comprobación de paridad es el encargado de controlar y realizar la lectura de datos de la FIFO, de transmitir los bits con la periodicidad especificada, de gestionar el Gap entre palabras y de calcular la paridad de cada palabra.

Para la implementación de toda esta funcionalidad se ha hecho uso de cuatro estados según el método de dos procesos comentado anteriormente en la Ilustración 3. Estos estados son idle, transmitting, gap y load.

Los datos de control de este módulo vienen del módulo de lógica de control con el siguiente formato: "Parar transmisión, Tasa de bit, paridad, tipo de paridad" a la señal from_control_logic. Además, de la lógica de control viene una señal adicional indicando si el gap esta activado o desactivado.

Al activar el reset el módulo pasa al estado de espera (idle) y permanece en ese estado transmitiendo una señal al codificador (Waveform Shaper) indicando que no hay transmisión para que no haya voltaje en el bus. Mientras está en este estado de espera se queda monitorizando la señal FIFO empty hasta que esa señal indique que la FIFO no está vacía. En caso de que la FIFO tenga datos pasará al estado de carga de datos (load).

En el estado de carga de datos se activa la señal rd_en de la FIFO y espera un ciclo de reloj a los resultados. Una vez recogidos los datos los formatea como establece la norma ARINC-429 dándole la vuelta a la etiqueta de tal modo que queda el MSB de la etiqueta en el bit 0 y se calcula el número de ciclos de reloj total por cada periodo de bit.

En la Ilustración 18 se muestra el formato en que el software almacena las palabras ARINC 429 en el registro de datos del esclavo AXI y el formato al que este módulo transforma dicha palabra para su posterior transmisión. Como se ha comentado anteriormente se invierte la etiqueta (label) quedando de tal manera que su MSB sea el primer bit de la palabra en ser transmitido.

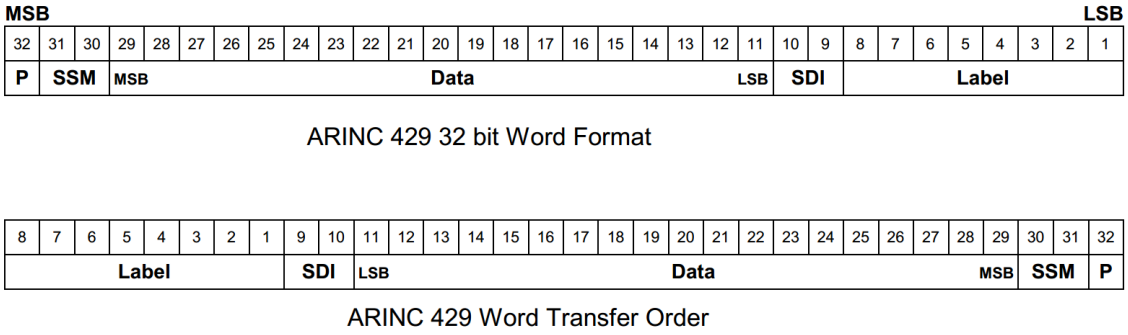


Ilustración 18. Formato de transferencia de palabras ARINC429

Tras el proceso descrito en el párrafo anterior pasará de nuevo al estado idle, en el que esta vez se quedará esperando a que se cumplan los ciclos de reloj por bit establecidos. Un ciclo antes de concluir el periodo de bit se pasa al estado transmitting para que al cambiar de estado el periodo de bit se haya concluido perfectamente.

En el estado transmitting se selecciona el siguiente bit de la palabra y se actualiza la salida del bit al codificador (Waveform Shaper). En este estado es donde se calcula la paridad. Cada vez que se cambia de bit se entra en este estado así que se van sumando los bits de manera binaria. Cuando llega el bit 30, siendo el 0 el primero, se establece el bit 31 o bit de paridad según el tipo de paridad seleccionada por el registro de control y el valor de la suma.

Una vez cambiado el bit se repite el proceso hasta que el bit de paridad de la palabra es enviado. En la transmisión de dicho bit mientras está en el estado de gap el comportamiento de este módulo varía un poco dependiendo de si el gap esta activado o desactivado. Si el gap esta desactivado dos ciclos antes de finalizar la transmisión del bit dependiendo de si hay datos en la FIFO o no pasará al estado de carga de datos o al de espera respectivamente. En cambio si el gap esta activado, dos ciclos antes de terminar la transmisión de este último bit se activa la señal to_gap_control que indica al gap control que entre en juego.

Estando el gap activado y una vez concluidos los ciclos de reloj de la transmisión del bit de paridad, el módulo pasará al estado de gap. En este momento el módulo envía constantemente la señal indicando al codificador que transmita silencio (gap) y se queda esperando a la señal proveniente del controlador de gap indicando que el periodo se ha finalizado. Una vez recibida la señal del controlador de gap comprobará si la FIFO tiene datos; si es así se pasará al estado de carga de datos y si por el contrario está vacía pasará al estado de espera (idle). En este último caso permanecerá enviando la señal de silencio al codificador mientras se repite todo el proceso.

Para que el registro de estado refleje en tiempo real si el transmisor se encuentra esperando a recibir datos del software o por el contrario está transmitiendo tenemos que tener en cuenta que desde que sale la señal del módulo de registro de paso hasta que sale del codificador pasan dos ciclos de reloj o dicho de otra manera hay un desfase de dos ciclos de reloj. La señal que indica en qué estado se encuentra no puede provenir del codificador ya que éste no tiene información sobre el controlador de gap y mientras se envía dicho gap y hay

datos en la FIFO el transmisor aun está transmitiendo por lo que se decidió que esa señal se originase en este módulo con un desfase de dos ciclos de reloj.

Este desfase de dos ciclos de reloj se ha implementado con una variable en el estado idle que indica si el receptor estaba activo en el estado anterior ya que cuando se activa ya habrán pasado dos ciclos justos. Más fácil ha sido a la hora de desactivar dicha señal ya que en el estado de gap si no hay datos en la FIFO se considera que ya no está transmitiendo por tanto al ciclo siguiente de llegar al estado si no hay datos se desactiva cumpliendo con el desfase necesario.

En la Ilustración 19 podemos ver las entradas y salidas de este módulo definidas anteriormente excepto las que se dirigen hacia el codificador o Waveform Shaper. La señal to_waveform_bitrate como su propio nombre indica es la que indica la tasa de bit a la que está trabajando el módulo. Esto es necesario ya que la tasa de bit no se actualiza directamente del registro de control si no que solo se actualiza al comienzo de transmisión de una palabra y por tanto solo la puede conocer si le informa el registro de desplazamiento. La segunda señal dirigida al codificador está formada por dos bits. el MSB es el bit que indica si se está transmitiendo un silencio o gap o un bit valido. Y el segundo bit es el dato a transmitir. Si el bit MSB es un 1 el codificador enviará un silencio y si es un 0 leerá el LSB y lo transmitir.

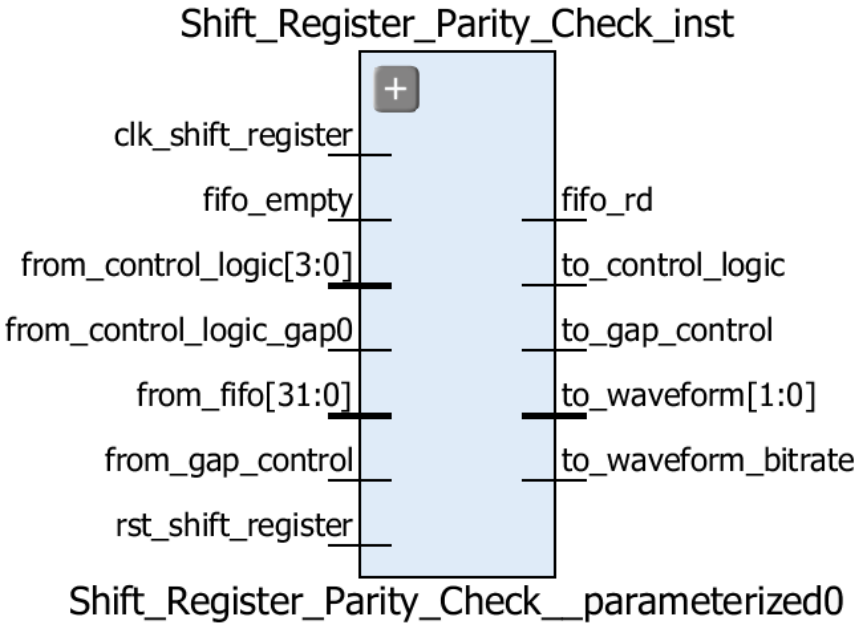


Ilustración 19. Módulo Shif Register and Parity Check formato caja negra

4.5.6 Waveform Shaper

Este bloque es el encargado de codificar la salida del transmisor. ARINC 429 establece una codificación RZ Bipolar y lo que se implementa es la transmisión de los unos y los ceros en diferentes salidas siempre retornando a cero.

En la Ilustración 20 se muestran el vector de entrada from_shift_register descompuesto en bits y las salidas que provocará en el módulo. El bit 1 es el MSB y el 0 el LSB.

El bit 1 indica si se está transmitiendo un gap o no y en caso de no estar transmitiendo un gap el bit 0 indica el bit a transmitir. Como vemos en la ilustración cuando el bit 1 está activo en la salida de ambas líneas habrá un valor de tierra a lo que llamamos silencio o null. Y por el contrario cuando el bit 1 tiene un valor de tierra se activa la salida seleccionada por el bit 0. Es decir si el bit 0 está activo la salida tx_hi emitirá un pulso de periodo igual a medio periodo de bit y si por el contrario tiene un valor de 0 voltios se emitirá ese pulso en la salida tx_lo.

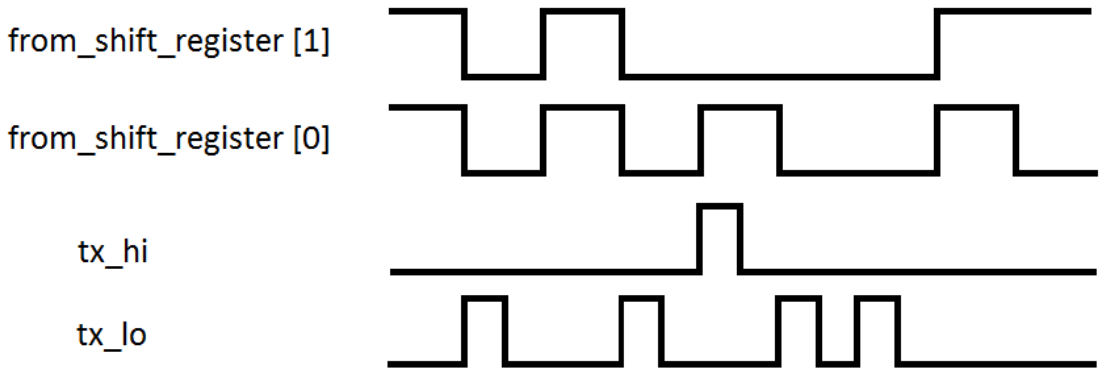


Ilustración 20. Entradas/Salidas módulo Waveform Shaper

Este módulo controla que el pulso y el periodo de null entre periodos se cumple correctamente con cierto margen de error especificado en la norma. Esta duración depende de la tasa de bit seleccionada por el registro de desplazamiento y el Waveform Shaper solo recoge y actualiza este valor cada vez que finaliza un periodo completo de bit y gap o de manera constante si está en estado de espera, es decir, cuando el bit MSB de from_shift_register esta activo y no se está transmitiendo ningún bit.

La Ilustración 21 muestra de manera gráfica las salidas y entradas del módulo definidas anteriormente. Dos entradas provenientes del Registro de desplazamiento y generador de paridad y dos salidas a la línea de transmisión además de la señal de reloj y la de reset.

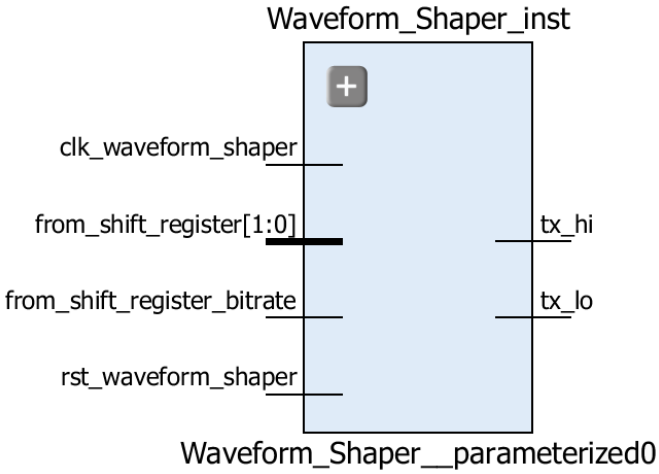


Ilustración 21. Módulo Waveform Shaper formato caja negra

4.6 Diseño e Implementación de Testbenches

Durante el proceso de implementación de los módulos se han realizado varios testbenches básicos para comprobar que los módulos funcionaban correctamente. Una vez terminado el módulo se procedía al diseño e implementación de los testbench automáticos que se encargan de comprobar la funcionalidad completa de cada módulo.

Como podemos ver en la Ilustración 22 todos los módulos menos el esclavo AXI han sido agrupados en un solo módulo al que hemos llamado transmitter.

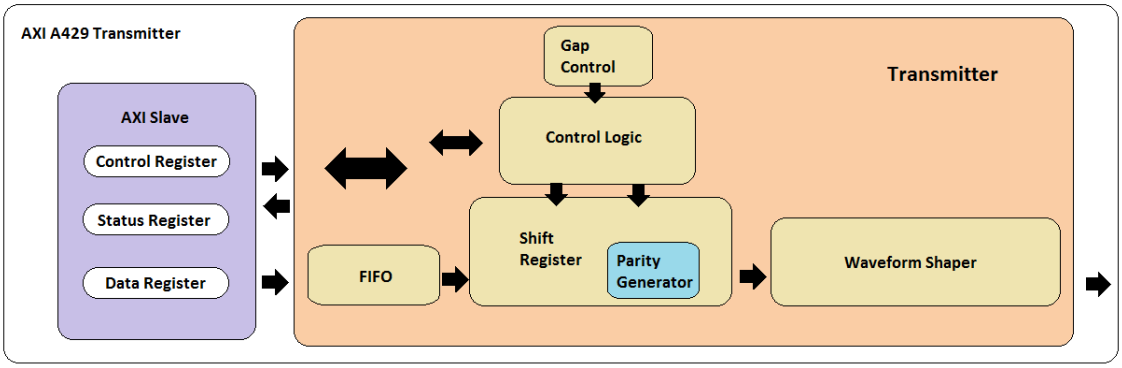


Ilustración 22. Diseño preliminar final

Esta agrupación de módulos ha sido realizada para la implementación de tres testbenches adicionales que se encarguen de comprobar que el conjunto de módulos se comporta como debería.

Como hemos visto en la Ilustración 4 llamamos testbench automático a una simulación que recoge un archivo de entrada generado con un script Matlab/Octave, que contiene todas las posibles entradas al módulo o el mayor número posible de entradas. Esta simulación produce un archivo de salida que contiene la información necesaria para que otro script pueda evaluar mediante el archivo de entrada y el de salida si el funcionamiento del modulo en cuestión es el debido.

4.6.1 Control Logic

Para probar este módulo lo que se ha hecho es cambiar la entrada de datos y de control cada dos ciclos de reloj ya que las reacciones ya serán visibles en el segundo ciclo. Se leerá un archivo de entrada llamado "Control_Register_Input_File.txt" que contiene todas las posibles entradas al Registro de control (Ilustración 16). Cada vez que cambia la entrada de datos y de control se activa la entrada del AXI que indica que hay un nuevo dato al módulo.

Las entradas provenientes de la FIFO se simulan a 0 es decir como si la FIFO no almacenara ningún dato ya que lo que nos interesa es ver que la señal de escritura y datos hacia la FIFO son correctas.

En el archivo de salida de esta simulación se escriben todas las salidas durante el segundo periodo de reloj que dura la entrada de datos. El nombre de este archivo es "Output_Gap_Control_File.txt" y el script que comprueba la validez de estos archivos de

entrada y salida se llama "check_GC.m" y se encarga de comprobar la activación de escritura en la FIFO, que los datos de entrada a la FIFO son correctos, y que según los bits del registro de control se envíe la información correcta al resto de módulos así como al registro de estatus (exceptuando los valores de la FIFO).

Una vez ejecutado el script se mostrará por pantalla una tabla con todos los casos probados con información sobre los fallos encontrados en cada caso o si no ha habido fallos.

En la Ilustración 23 vemos que las señales de entrada obtienen la respuesta de salida un ciclo de reloj más adelante excepto las señales de control que van al resto de módulos ya que éstas están directamente conectadas o solo atraviesan comparaciones simples que deberían ser sintetizadas como puertas lógicas que introducirían un retraso despreciable.

El formato de tabla que vemos en la Ilustración 24 muestra las últimas líneas de una simulación correcta. Como vemos los datos de entrada a la FIFO se van alternando entre dos valores en este caso esto es correcto al estar en el registro de control el bit LSB activo.

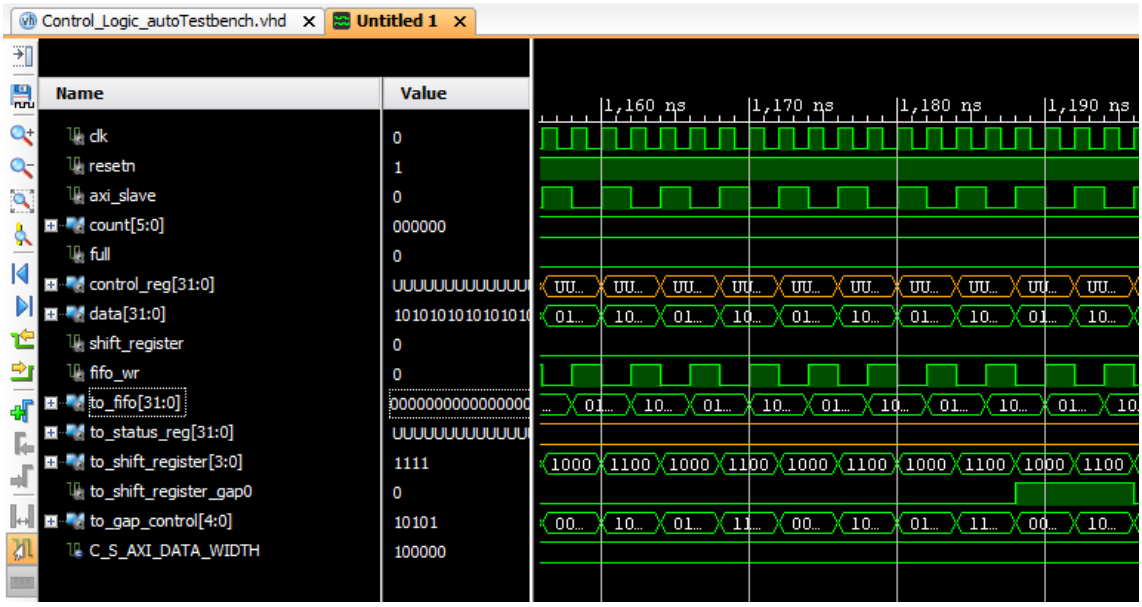


Ilustración 23. Simulación de comportamiento del módulo Control Logic

```
    110111111          1   10101010101010101010101010101010    1       UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU0001    0111        0       11011
    001111111          1   01010101010101010101010101010101    1       UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU0001    0011        0       00111
    101111111          1   10101010101010101010101010101010    1       UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU0001    0111        0       10111
    011111111          1   01010101010101010101010101010101    1       UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU0001    0011        0       01111
    111111111          1   10101010101010101010101010101010    1       UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU0001    0111        0       11111
Control Register    fifo wr             to fifo             Reset fifo          Status Register         Shift Register Gap0 to Gap Control

    Control Logic testbench succesfully completed without any errors
```

Ilustración 24. Muestra de la salida generada por la ejecución del script de comprobación

4.6.2 Gap Control

Lo que se ha hecho para comprobar este módulo es comprobar con determinada señal de control y la señal procedente del registro de desplazamiento indicadora del comienzo de gap si el módulo permanecía en dicho estado de gap el número de ciclos de reloj correcto. Todos los posibles casos de entrada (Ilustración 17) de gap han sido contemplados en el archivo de entrada con nombre "Control_GC_Input_File.txt". Se contemplan todos los casos excepto que el gap sea 0 ya que en ese caso no le llegaría ninguna señal desde el registro de desplazamiento

La simulación actualiza el valor proveniente del control logic y a los dos ciclos de reloj activa la señal de entrada indicadora de gap. Desde el comienzo hay una señal que acumula el número de ciclos de reloj que pasan en la simulación, así que al llegar la señal de registro de desplazamiento almacenamos el valor de ciclos y esperamos a que el gap de por concluido el gap. Es en ese momento cuando restamos el valor actual de ciclos al anteriormente almacenado obteniendo así el número de ciclos total. Este número es almacenado en el archivo "Output_Gap_Control_File.txt".

El script encargado de contrastar los archivos de entradas y salidas es "check_GC.m" y lo único que hace es comprobar que el número de ciclos de salida se corresponde con lo que indica la señal de control.

En la Ilustración 25 fijándonos en los dos primeros duran cuatro periodos de bit y el tercero dura solamente uno. Poniendo el cursor encima vemos que el valor de control esta especificando que el gap debe durar solo un periodo de bit y por tanto está correcto.

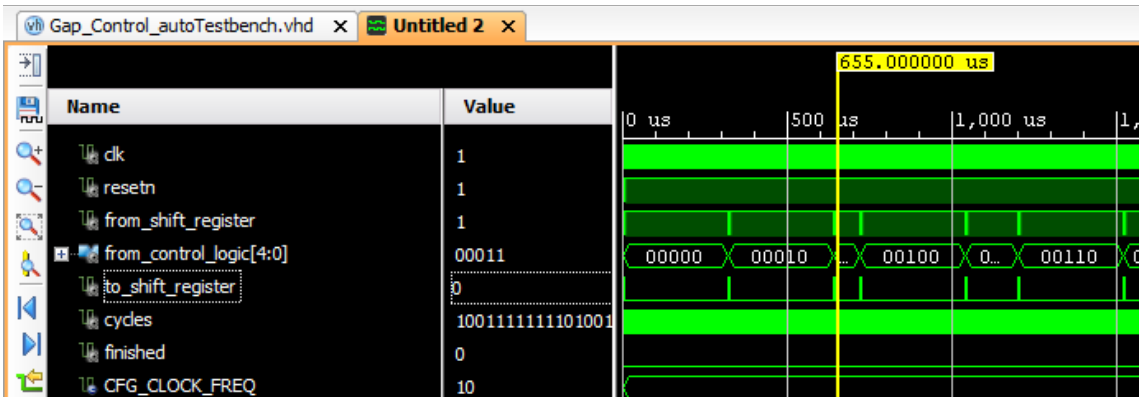


Ilustración 25. Simulación de comportamiento del módulo Gap Control

Como vemos en la Ilustración 26 el script tiene en cuenta la tasa de bit a la que está funcionando el módulo y la aplica a la ecuación de comprobación. Si el módulo funciona correctamente saldrá el mensaje de la ilustración y si no, vendrá indicado en el lateral derecho de la ecuación fallida.

```
160000  ==  4 * 40000   =   160000
160000  ==  4 * 40000   =   160000
40000   ==  1 * 40000   =   40000
30000   ==  6 * 5000    =   30000
20000   ==  4 * 5000    =   20000
35000   ==  7 * 5000    =   35000

Succesfully completed Gap Controller comprobation
```

Ilustración 26. Fragmento de la salida provocada por la ejecución del script de comprobación

4.6.3 Shift Register and Parity Check

Para comprobar este módulo leemos dos archivos de entrada que contienen por un lado las variables de control y por otro datos aleatorios y se llaman respectivamente "Control_SR_Input_File.txt" y "Data_Register_Input_File.txt".

Se comprueban todas las entradas posibles de control y el mismo número de entradas de datos aleatorios. Se comprueba que si el vector de control indica que el transmisor estaba activado se han transmitido los datos con los parámetros correspondientes de tasa de bit y paridad. Por tanto comprueba si la transmisión de cada palabra a durado lo que debería además de comprobar que el bit de paridad ha sido generado de manera coherente con el registro de control y que la etiqueta ha sido volteada correctamente.

Para realizar todas estas comprobaciones, tenemos como en la anterior simulación una variable que cuenta los ciclos de reloj y un contador de bits. Desde el primer envío de datos se van muestreando los bits cada periodo de bit, y cada vez que se muestrea aumentamos el contador de bit. Una vez ha llegado el contador a 32 y habiendo cogido en el primer bit los ciclos de reloj restamos el valor actual entre este consiguiendo los ciclos de reloj que ha durado la transmisión. Escribimos estos datos además de la tasa de bit en cada línea del archivo "Output_Shift_Register_File.txt" y con esto ya podemos comprobar la funcionalidad del módulo.

El script de comprobación no tiene más que ir leyendo los tres archivos y según lo que aparezca en la señal de control comprobar si el tiempo de transmisión está bien y, si aplicando lo indicado en la señal de control sobre la señal de datos obtenemos como resultado la palabra producida por nuestro módulo.

En la Ilustración 27 vemos que la transmisión de datos en la señal to_waveform coincide con los datos en from_fifo. Además podemos ver la señal que desencadenaría la lectura una vez conectado a una FIFO.

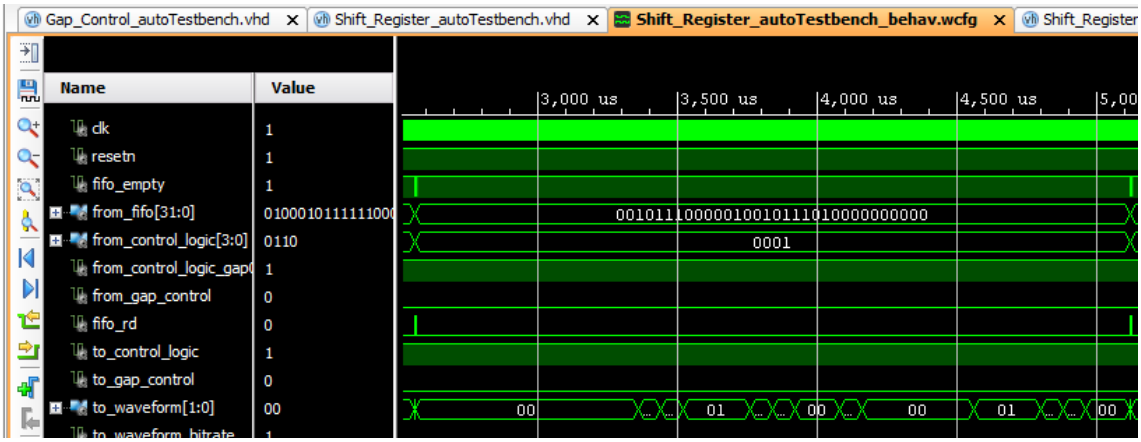


Ilustración 27. Simulación de comportamiento del módulo Shift Register and Parity Check

Como se puede ver en la Ilustración 28 el formato de salida del script es muy parecido a los anteriores. Si está correcto pondrá la frase concreta y si no saltarán los errores para poder solucionarlos. La primera columna es la señal de control, la segunda es la señal de datos modificada por el script y la tercera es la salida producida por el módulo.

```
0101    00111000110010111000000010000000 == 00111000110010111000000010000000
0110    01000101111110000011000010000000 == 01000101111110000011000010000000
0111    01101000001110000001110010000000 == 01101000001110000001110010000000

Comprobation succesfully completed
```

Ilustración 28. Fragmento de la salida producida por el script de comprobación

4.6.4 Waveform Shaper

Este módulo se comprueba mediante un archivo de entrada que se llama "Waveform_Shaper_Input_File.txt" y contiene todas las posibles entradas procedentes del registro de desplazamiento. La simulación genera otro archivo llamado "Waveform_Shaper_Output_File.txt" y el script de comprobación se llama "check_WS.m".

Como en la anterior simulación tenemos una variable que se encarga de contar los ciclos de reloj durante esta. El proceso de recolección de datos es muy similar al anterior. Cuando llega un bit a una de las líneas se almacena el numero de ciclos, y cuando termina de transmitir la parte alta del bit se escriben: el estado de las salidas en el momento de llegada del bit, los ciclos de reloj que ha permanecido en estado activo y, si se han transmitido dos datos seguidos, el número de ciclos completo del periodo de bit.

Como vemos en la Ilustración 29 este módulo es el más fácil de comprobar ya que el número de posibles entradas es muy bajo. En la ilustración no se aprecia pero entre la señal de entrada y la de salida hay dos ciclos de reloj. Un retraso necesario para proteger la señal frente a retrasos entre módulos. También podemos ver que para calcular la duración del periodo de bit tiene en cuenta la señal que proviene del registro de desplazamiento. A partir de los 320 microsegundos se ve como los pulsos se transmiten a una frecuencia mucho mayor (al principio a 12.5 Kbps y al final 100Kbps).

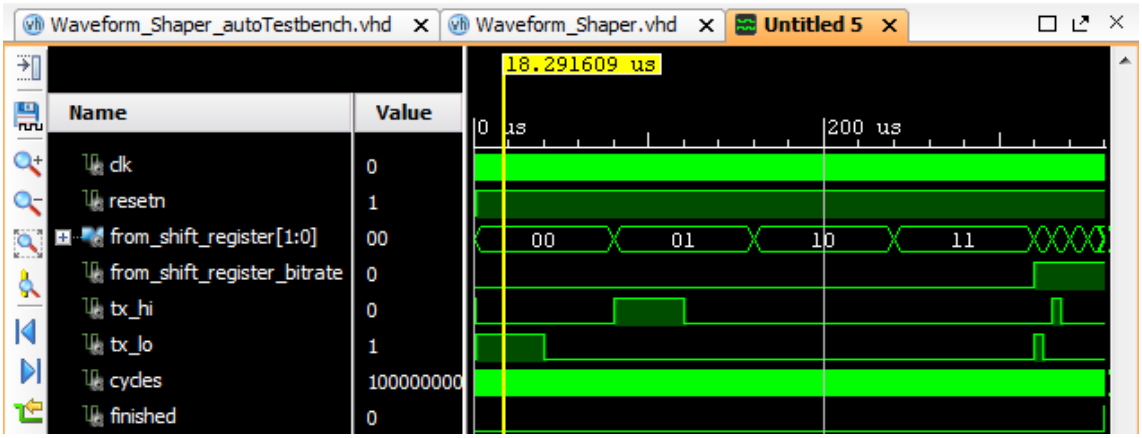


Ilustración 29. Simulación de comportamiento del módulo Waveform Shaper

En la Ilustración 30 el primer valor indica la tasa de bit a la que estaba configurado el codificador. El segundo indica el numero de ciclos de reloj estimado a partir de los ciclos que ha durado el pulso activo. Y el último dato refleja la cuenta total de ciclos de reloj del anterior bit, la cual solo puede ser calculada sí hay dos bits seguidos. Los null mostrados por pantalla son los que debería haber según las entradas de control y si los comparamos con la Ilustración 29 vemos que el módulo funciona correctamente.

```
Bitrate : 12.5 kbps cycles count : 40000
Bitrate : 12.5 kbps cycles count : 40000 ExactBitPeriod : 40000
null
null
Bitrate : 100 kbps cycles count : 5000
Bitrate : 100 kbps cycles count : 5000 ExactBitPeriod : 5000
null
null

Succesfully completed Waveform Shaper comprobation
```

Ilustración 30. Salida producida por el script de comprobación

4.6.5 Transmitter

Para evaluar el conjunto de módulos se han englobado en uno al que hemos llamado transmitter. Y para ser capaz de probar las funcionalidades del transmisor completo se han diseñado tres simulaciones, dos de ellas muy similares. Las tres simulaciones las hemos llamado Transmitter_automatic_testbench, Transmitter_autoTestbench_status y Transmitter_frequencyTest.

4.6.5.1 Transmitter Automatic Testbench

Este test es el encargado de comprobar la funcionalidad de transmisión. Es decir el test simula el comportamiento del esclavo AXI y escribe los resultados de salida en el archivo "Data_Register_Output.txt".

Para simular el esclavo AXI recoge las posibles entradas de datos y de control de dos archivos llamados "Control_Register_Input_File" y "Data_Register_Input_File.txt". El primer archivo contiene todas las posibles combinaciones del registro de control $2^9 = 512$ y el segundo archivo tiene todas las combinaciones de SDI, SSM y etiquetas $2^{12} = 4096$ al no poder probar todos los datos, ya que es innecesario y sería una pérdida de tiempo.

El procedimiento que sigue la simulación es sencillo. Por cada palabra de control se prueban 8 de datos. El testbench está diseñado para que se envíen todas las palabras que entran en la FIFO. Por ejemplo si el transmisor está activado pero la transmisión está parada esos datos que se almacenan en la FIFO se enviarán la próxima vez que el transmisor este activo y este transmitiendo. Debido a esto en la simulación estos registros de control que permiten la transmisión duran más que el resto para permitir a la FIFO que se vacíe.

Gracias a que la FIFO no se llena nunca, podemos estar seguros de que no se van a perder datos a no ser que haya algún fallo. El script de comprobación lo único que tiene que hacer es a medida que lee las palabras de control y de datos comprobar que cuando la palabra de control especifica que el transmisor está activado la palabra se ha escrito en el orden correcto en el archivo de salida.

Este script tiene en cuenta de manera similar a en la simulación del módulo Shift Register and Parity Check que en la palabra escrita por el módulo deberán haberse aplicado los cambios indicados por el registro de control como son la generación de la paridad y darle la vuelta a la etiqueta.

En la Ilustración 31 se puede observar el funcionamiento de la simulación descrito en los anteriores párrafos. En el segundo cambio el transmisor está activo pero la transmisión esta parada y podemos ver que los datos se escriben en la FIFO si nos fijamos en las señales de wr_en y empty pero no transmite ningún dato. En cambio en el cuarto cambio del registro de control vemos que el transmisor comienza a vaciar la FIFO sin dejar de escribir en ella y que en efecto el tiempo que permanece con los mismos datos se duplica.

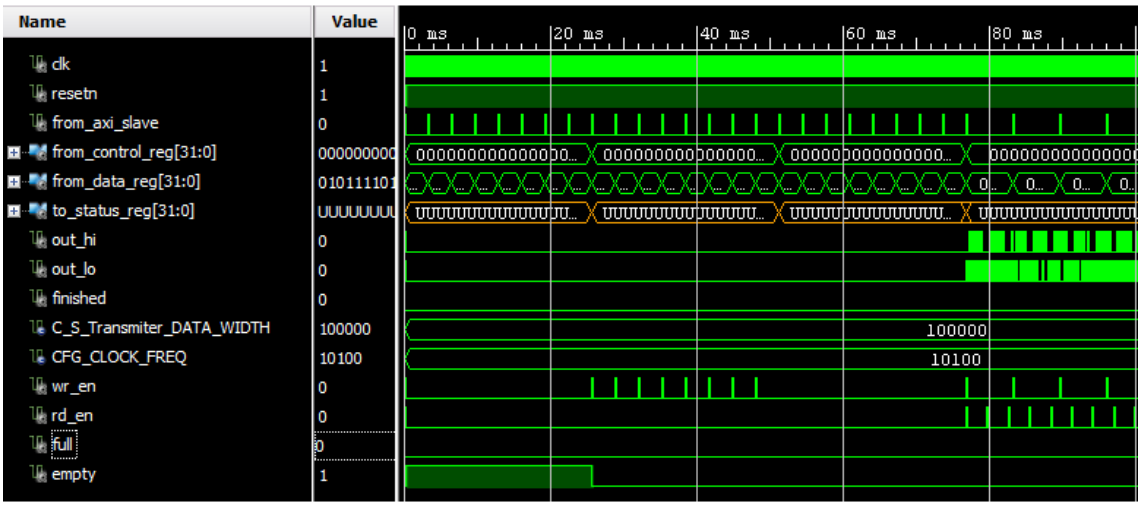


Ilustración 31. Simulación a gran escala del transmisor

En la Ilustración 32 se muestra la misma simulación que en la Ilustración 31 con un pequeño zoom que nos permite ver los datos y como se transmiten finalmente. Vemos también como hay cuatro periodos de bit entre cada palabra ya que es el Gap por defecto establecido por el registro de control.

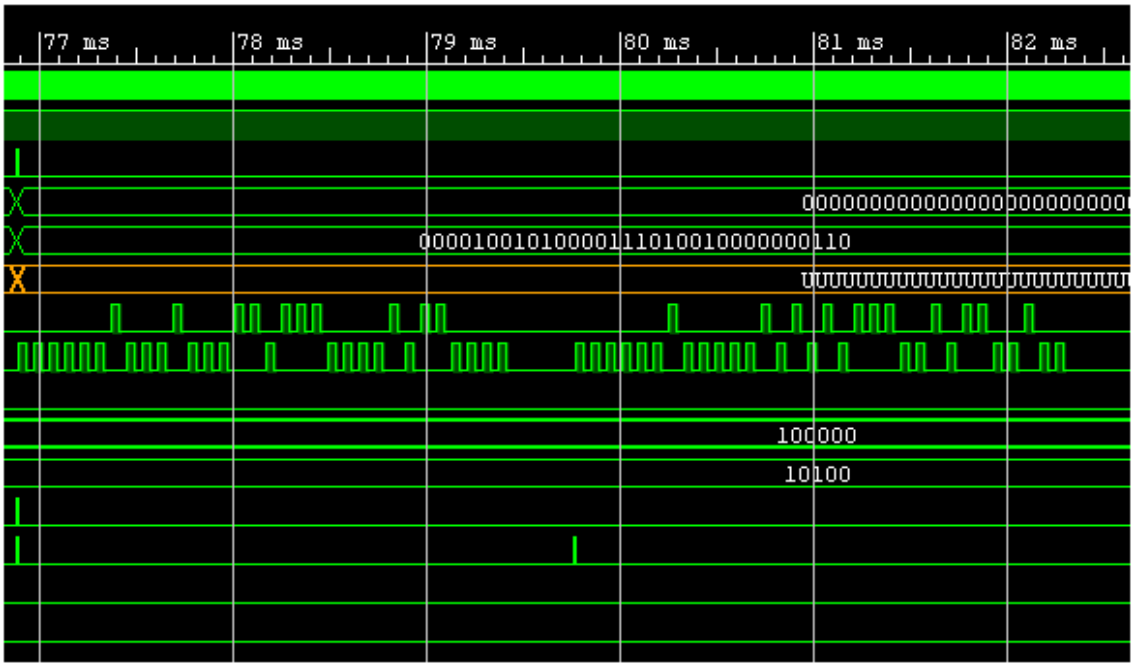


Ilustración 32. . Simulación a pequeña escala del transmisor

El formato de salida que vemos en la Ilustración 33 nos muestra arriba a la izquierda los bits de control para cada ristra de 8 palabras y debajo se muestran las palabras, a la izquierda las de entrada añadiéndoles el bit de paridad y a la derecha las de salida reconstruida.

```
111111110 :
111111111 :
10001101000011101111111111111110 => 10001101000011101111111111111110
00111001000000100110001111111110 => 00111001000000100110001111111110
01000110111000010100101111111110 => 01000110111000010100101111111110
01111011110111101111101111111110 => 01111011110111101111101111111110
10010001010010001001101111111111 => 10010001010010001001101111111111
10101101110100000010011111111111 => 10101101110100000010011111111111
11000011011101011010101111111111 => 11000011011101011010101111111111
11110100111010010000111111111111 => 11110100111010010000111111111111


    Transmitter functionality fully working properly
```

Ilustración 33. Fragmento de la salida del script de comprobación del transmisor

4.6.5.2 Transmitter Autotestbench Status

Este test solo se encarga de comprobar que el transmisor cambia correctamente y a tiempo el registro de estado. Para realizar esta comprobación se ha diseñado una simulación que se encarga de saturar la FIFO en un tiempo determinado menor a la transmisión de la primera palabra. De esta manera tendremos el registro de estado con 4 unos antes de que termine la transmisión de la primera palabra.

El registro con 4 unos significa que la FIFO está llena y que el transmisor está activado y transmitiendo. A partir de ahí conociendo la frecuencia de operación del reloj podemos calcular el número de ciclos exacto en el que debería cambiar el estado. Por último al terminar de transmitir el test desactiva el transmisor finalizando así la simulación con el registro de estado con 4 ceros.

Al controlar la escritura en la FIFO y escribir de manera equiespaciada también podemos calcular al cabo de cuantos ciclos de reloj debería empezar a transmitir y por tanto reflejarlo en el estatus así como cuando debería activarse el flag de FIFO medio llena y llena.

Esto mismo es lo que comprueba el script a partir de unas cifras de tiempo dadas calculadas a priori de una manera muy sencilla. Para calcularlas solo hay que ir sumando los retrasos producidos por procesado de datos y el desfase entre salida y entrada, entendiendo entrada como el módulo de lógica de control y salida como la salida del codificador.

Una vez llenada la FIFO calcular cuando tiene que cambiar al siguiente valor es directo ya que solo hay que contar cuantas palabras deben vaciarse y multiplicar ese número por el tiempo que tarda en transmitirse una palabra.

En la primera imagen de la Ilustración 34 vemos como saturamos la FIFO enviando 65 palabras en menos de 1'5 microsegundos y si no fijamos en el comienzo de la transmisión vemos que el registro de estatus es actualizado en el mismo ciclo de reloj que la salida de nivel bajo es activada. En la segunda imagen podemos ver con una visión global de la simulación como el registro de estado va cambiando a medida que se envían palabras. A simple vista podemos decir que tiene un buen comportamiento ya que el bit del registro de estado que indica si la FIFO esta medio llena permanece el mismo tiempo activado como desactivado.

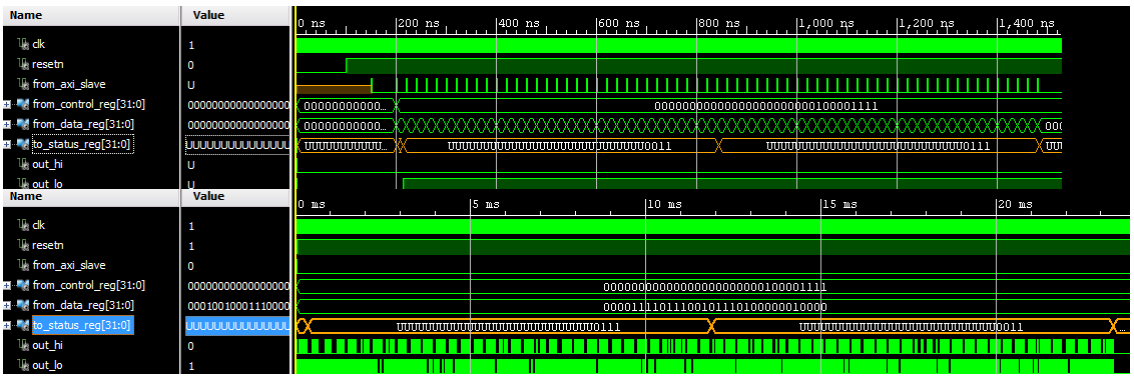


Ilustración 34. Simulación del registro de estatus, comienzo y vista global

En este caso la Ilustración 35 muestra el tiempo en el que el módulo ha cambiado de estado, a que estado ha cambiado y si el resultado es el esperado.

```
check_status
     Tiempo                    Status  Result
       202 ns            ->     0001  success
       214 ns            ->     0011  success
       844 ns            ->     0111  success
       1484 ns           ->     1111  success
       360208 ns         ->     0111  success
       11880208 ns       ->     0011  success
       23360214 ns       ->     0001  success
```

Ilustración 35. Salida del script de comprobación del estado del transmisor

4.6.5.3 Transmitter Frequencytest

Esta simulación es muy parecida a la anterior, lo único que cambia es que en este caso se comprueba si la actualización del estado y el transmisor funciona correctamente a una frecuencia de reloj tan baja como 400KHz.

En la Ilustración 36 se muestra la salida del script que comprueba el funcionamiento del transmisor a una frecuencia baja. Como vemos la diferencia entre la activación del transmisor y el comienzo de la transmisión pasa de 12 nanosegundos a 15 microsegundos. Para entender estos datos de salida hay que saber que para pasar por todos los estados en este caso hemos tenido que almacenar y transmitir 68 palabras en cambio en el anterior solo 65 por eso hay tanta diferencia cuando comparamos ambas tablas.

```
check_freq
     Tiempo                    Status  Result
       202.5 us          ->     0001  success
       217.5 us          ->     0011  success
       880 us            ->     0111  success
       1577.5 us         ->     1111  success
       1650 us           ->     0111  success
       13170 us          ->     0011  success
       24657.5 us        ->     0001  success
```

Ilustración 36. Salida del script de comprobación de la frecuencia

4.7 Síntesis e Implementación

Una vez ejecutadas todas las simulaciones del apartado anterior se ha sintetizado el código VHDL y se han realizado de nuevo todas las pruebas. Sintetizar es transformar el código VHDL en circuitos digitales específicos para una placa en concreto. Tras solucionar los fallos mostrados al ejecutar las simulaciones post-sintetizado del tipo funcional y de tiempo se ha procedido a la implementación del diseño y nuevamente a la realización de las simulaciones y solucionar los fallos mostrados. La implementación distribuye la lógica en la placa especificada.

La Ilustración 37 muestra el IP Core completo ya sintetizado y la interconexión entre el esclavo AXI y el transmitter.

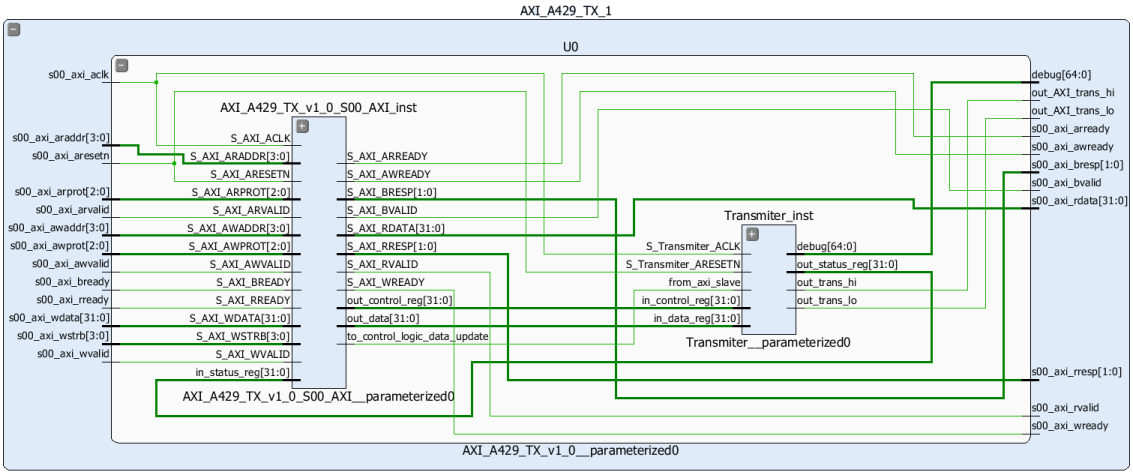


Ilustración 37. Esquemático de interconexión del módulo AXI y el Transmitter

En la Ilustración 38 vemos los módulos que forman el Transmitter y la interconexión entre sí. Como se puede ver todas las entradas al módulo pasan por el bloque de lógica de control y finalmente las salidas del módulo se corresponden con las del codificador. En este caso hemos añadido unas salidas extra de depuración.

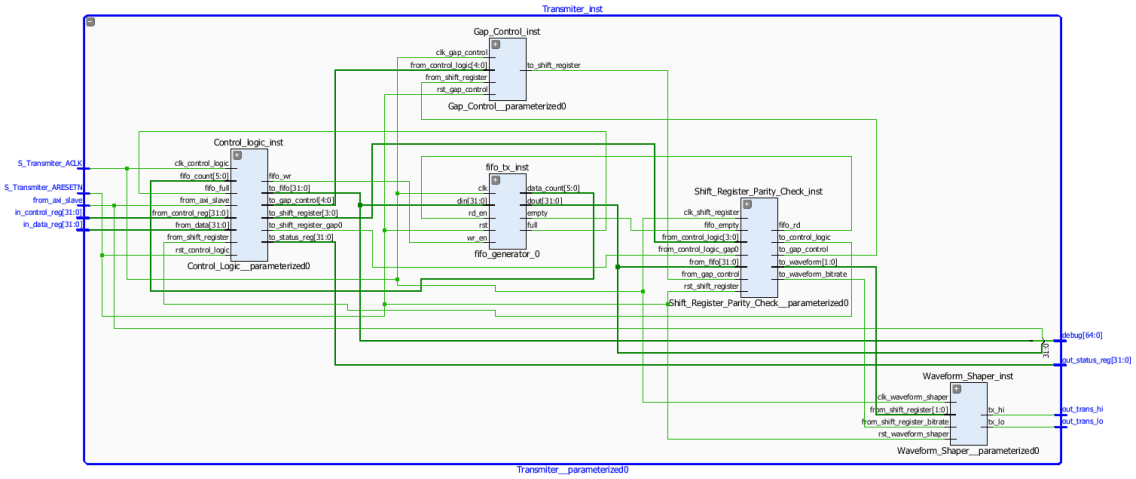


Ilustración 38. Interconexionado interno del módulo Transmitter

4.8 Programación de la FPGA

Para esta tarea hemos decidido usar una Zedboard debido a su disponibilidad y a contar con un chip (Zynq 7000) que contiene una parte programable (PL o Programmable Logic) y otra parte de procesador (PS o Processing System). A este tipo de chips se los denomina SoC (System on chip) programable y nos va a permitir probar nuestro hardware corriendo una aplicación de prueba que interactúe con él.

Mediante el SDK de Xilinx es posible introducir un software simple que probara el módulo transmisor junto con un módulo receptor ARINC 429 cuya implementación se realizó de manera paralela.

El proceso de programación es sencillo, solo necesitamos unos drivers para poder interactuar con el transmisor y receptor y un pequeño programa de prueba. Este software se ha realizado en C y lo que hace es escribir palabras de 4 bytes en el registro de datos del esclavo AXI del transmisor. A su vez el software recoge las palabras que llegan al receptor y las muestra en un terminal conectado al bus serie de la FPGA. Esta transmisión se realiza manteniendo el registro de control fijo a un valor que nos permite transmitir.

Para llegar a este punto de programación software hemos tenido que añadir unos módulos adicionales en nuestro diseño. Como vemos en la Ilustración 39 estos módulos son los que hacen referencia al PS. Hemos añadido los bloques correspondientes al bus AXI, al procesador (procesador y resets del sistema) y a los dos puertos UART para monitorizar la FPGA y continuar con el proyecto, además se ha añadido un último módulo para depurar la aplicación mientras esté corriendo. Los dos módulos que quedan por nombrar son el transmisor y receptor ARINC 429.

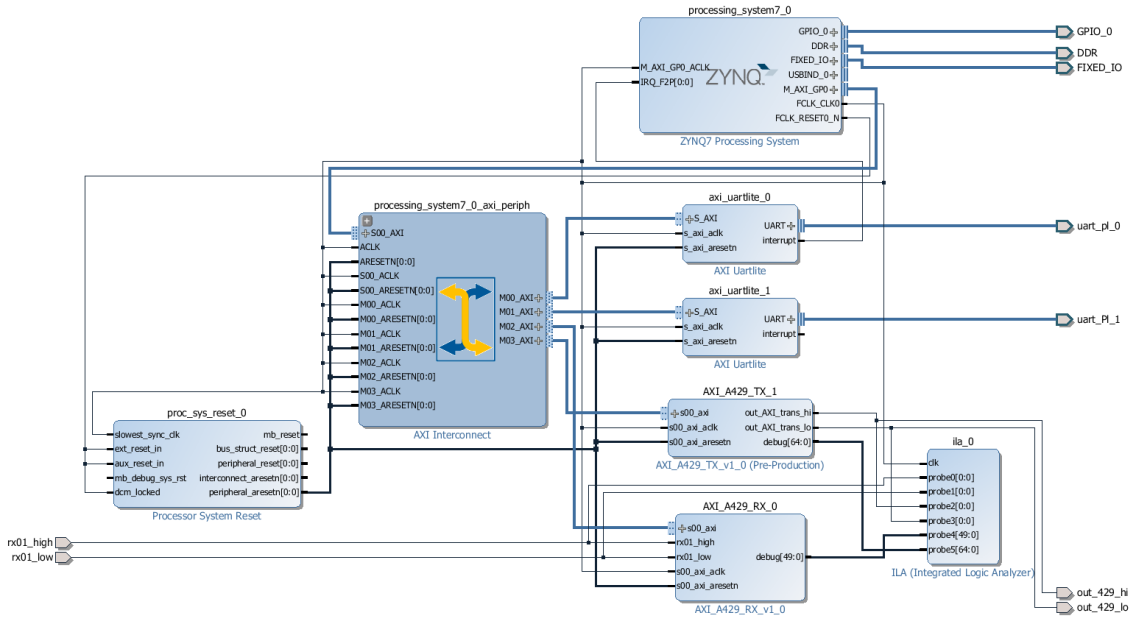


Ilustración 39. Diagrama de bloques del sistema completo

Para interconectar el transmisor y receptor en la FPGA les hemos asignado unos puertos de salida y entrada respectivamente. Como vemos en la Ilustración 40 cortocircuitamos los puertos del transmisor y receptor mediante el cable marrón y el cable verde. Con la FPGA conectada como muestra la Ilustración 40 y el diseño de bloques como en la Ilustración 39 ya está todo listo para realizar la programación.

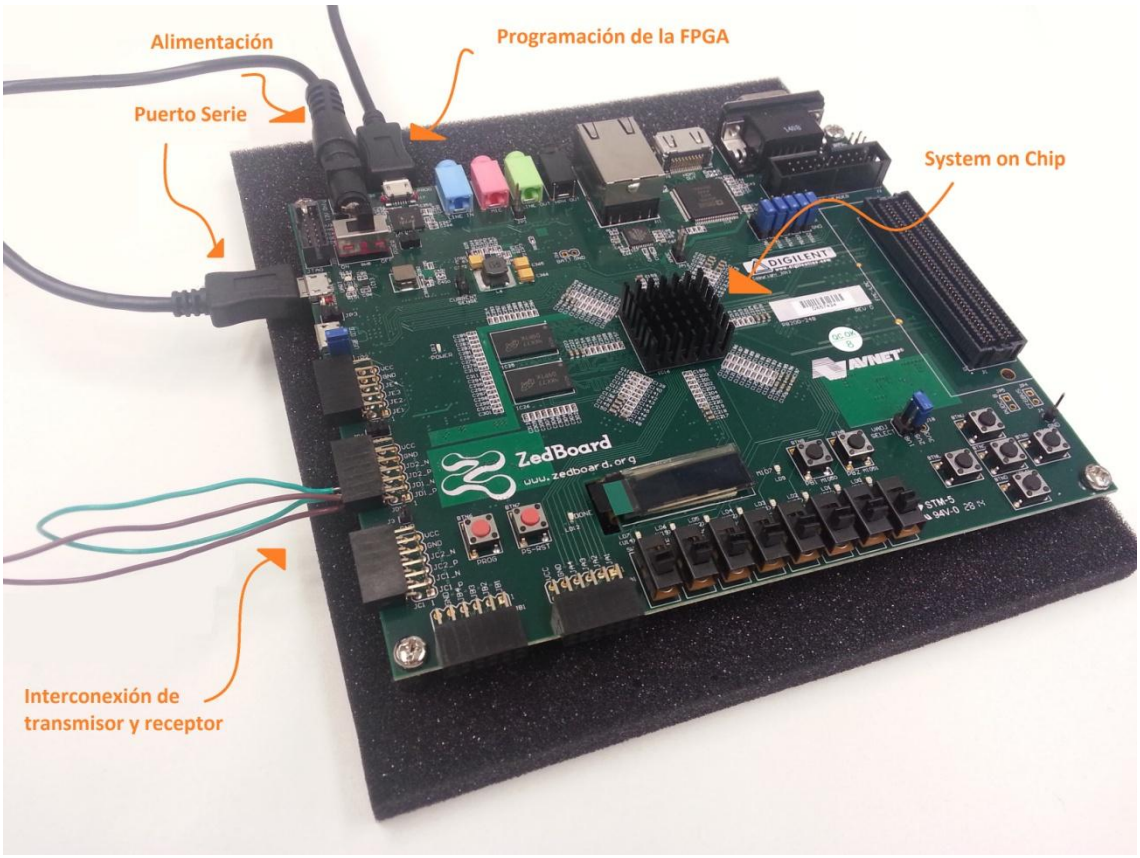


Ilustración 40. Zedboard con transmisor y receptor

La Ilustración 41 nos muestra el interfaz del SDK de Xilinx con el que una vez exportado el bitstream desde Vivado nos permite programar la FPGA y correr una aplicación en el PS. También vemos indicados el botón con el que programar la FPGA y el terminal en el que veremos los datos del bus serie. Para poder hacer esto el terminal deberá estar configurado a una tasa de baudios de 115200.

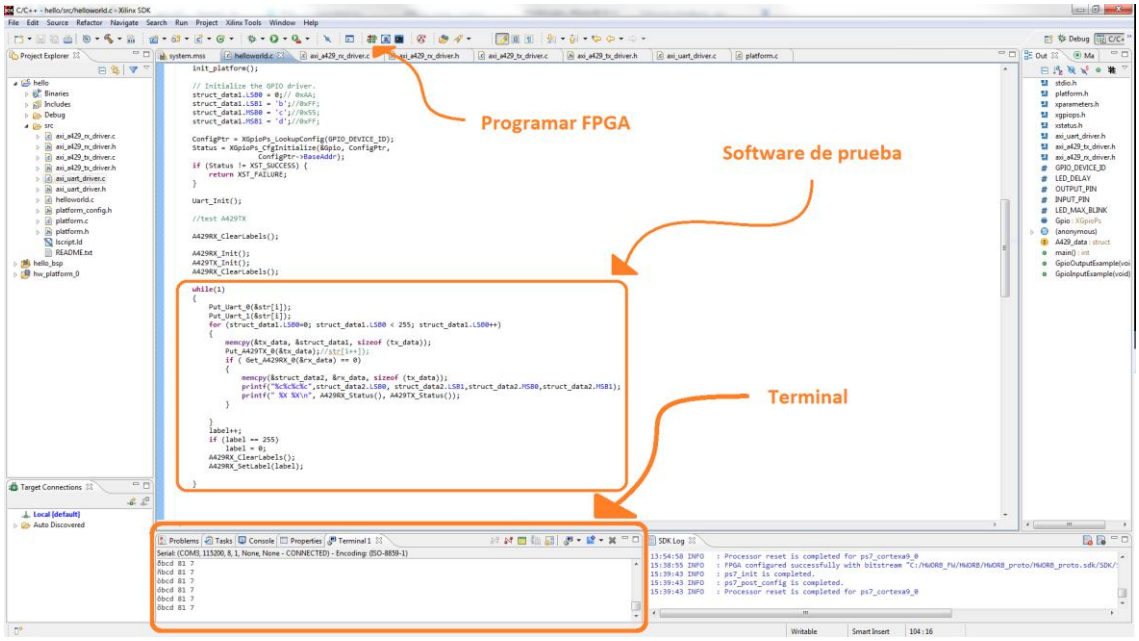


Ilustración 41. Interfaz del SDK de Xilinx

En la parte destacada del software podemos ver cómo lo que hacemos es escribir en el registro de datos del transmisor y, si han llegado datos al receptor, los mostramos por el bus junto con los registro de estatus del receptor y transmisor en ese orden. Los 8 bits LSB de los datos escritos van cambiando en cada iteración recorriendo de esta manera todas las posibles etiquetas, que es por lo que vemos en el terminal como va cambiando la primera letra de la palabra recorriendo todo el alfabeto ASCII.

La Ilustración 42 muestra como se depura mediante el ILA (Integrated Logic Analyzer). Esta herramienta nos permite ver las señales internas de la placa en tiempo real para en última instancia ver porque están fallando los módulos, en caso de que fallen.

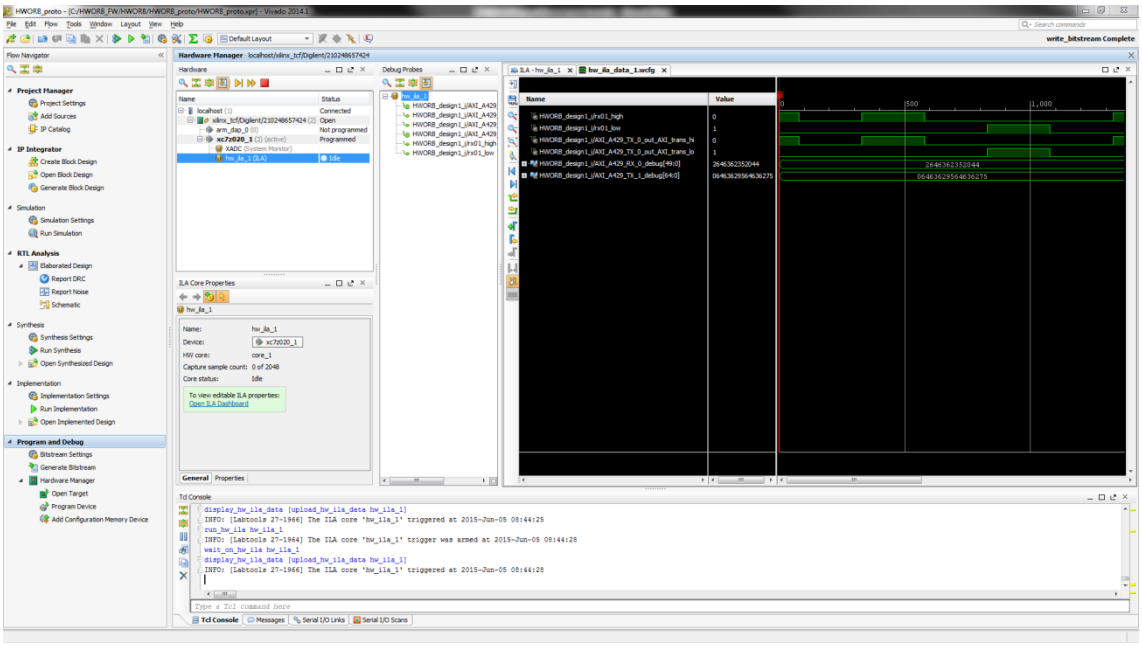


Ilustración 42. ILA integrado en Vivado en ejecución

5 Análisis de Resultados y Conclusiones

Como primer y principal resultado del trabajo destacar que se han cumplido los objetivos y se ha diseñado, desarrollado y testeado un transmisor ARINC 429. Se ha llegado a implementar y probar en una FPGA frente a un receptor ARINC 429 consiguiendo la comunicación deseada, y además se ha hecho de manera que una vez se implemente en una aplicación, el conjunto pueda ser certificable. Por todo esto concluimos que los objetivos de este proyecto han estado bien definidos y estructurados ya que se ha terminado con éxito cumpliendo todos ellos y en el rango de tiempo estipulado.

De este proyecto se puede extraer que realizar cualquier implementación de hardware crítico requiere una cantidad de simulaciones, pruebas y documentación muy alta, lo que se traduce en un gasto importante de tiempo. A pesar de ello el resultado final es de una calidad que compensa con creces ese gasto.

También podemos ver como la tecnología y protocolos utilizados en aviación son muy concretos y la información sobre ellos se limita a una visión generalista. Por tanto la búsqueda de información se traduce en una larga tarea que te hace recurrir a los documentos de definición del protocolo.

VHDL es un lenguaje muy amplio del cual es sintetizable un porcentaje bastante bajo y aun utilizando ese pequeño porcentaje, al realizar las simulaciones funcionales y temporales el módulo puede llegar a fallar por diversas causas. Esto produce una continua búsqueda del código que mejor encaje con el comportamiento que necesitamos implementar, así como un análisis de las diferentes sintetizaciones posibles.

Pero hay que destacar que VHDL a pesar de crear quebraderos de cabeza también hace que el diseño de hardware complejo sea mucho más sencillo. Permitiendo una gran abstracción de la electrónica durante la mayor parte del proceso de desarrollo.

Las herramientas que proporciona Xilinx te permiten una visualización de los componentes y las conexiones además de proporcionarte ayuda a la hora de configurar los puertos de la FPGA. También cuenta con un sintetizador muy potente capaz de traducir tu módulo VHDL en una gran cantidad de FPGAs del mercado.

Estas herramientas necesitan de licencia para ser usadas y, aunque la licencia gratuita te permite hacer gran cantidad de cosas solo puedes usarla para un ordenador. Una de las faltas de este programa es precisamente la necesidad de una licencia de pago para poder usar el ILA, herramienta muy útil a la hora de depurar las señales digitales una vez implementado el módulo en la FPGA. Además Vivado experimenta múltiples fallos repentinos que te obligan a volver a ejecutarlo con la implícita pérdida irremediable de datos y tiempo.

A la hora de ejecutar una simulación podemos usar el simulador Xsim integrado con Vivado o el simulador Modelsim que requiere de licencia. El simulador Xsim es funcional y permite ejecutar cualquier simulación pero cuando hablamos como en este proyecto de amplias simulaciones que llegan a durar días el Xsim puede llegar a ser excesivamente lento además de llegar a llenar la RAM impidiendo finalizar la simulación y provocando fallos en Vivado. Por eso se hace necesario emplear un simulador como Modelsim que no sobrecarga la RAM de nuestro ordenador y es ligeramente más rápido.

Durante el transcurso del proyecto se hizo claro que la Zedboard fue un gran acierto al incorporar un System on Chip que nos ha permitido probar el funcionamiento del hardware una vez implementado en su parte de lógica programable (PL) mediante una pequeña aplicación corriendo en su procesador basado en ARM (PS).

6 Líneas Futuras

La continuación lógica de este proyecto es acoplar un controlador de línea como uno de los mencionados en la Definición Técnica de un Transmisor ARINC 429 además de, una vez hecho esto, probar la comunicación de nuestro módulo con un dispositivo real que emplee esta tecnología de comunicación.

Una vez realizado el apartado hardware este proyecto tiene un gran rango de ampliación en su software, ya que la aplicación de prueba realizada para este es muy básica. Se quiere integrar un sistema linux en la Zedboard que nos permita correr aplicaciones propias sin necesidad de conectarla a un ordenador. Pudiendo así conectar directamente un monitor y un teclado a la placa.

Además como hemos hecho referencia en la Introducción esta es una pequeña parte de un proyecto mucho mayor. Por tanto el siguiente paso en este proyecto será la implementación de módulos transmisor y receptor AFDX y 1553.

Y una vez elaborados todos los protocolos necesarios para un sistema de comunicación de aviónica, el siguiente paso sería el aislamiento de los IP Cores. Es decir asignar a cada IP Core implementado un espacio delimitado por lógica sin utilizar dentro de la FPGA. Estos bloques aislados cuentan con entradas y salidas para interactuar con el resto de IP Cores.

En la Ilustración 43 podemos ver el significado de aislar un IP Core de manera gráfica. Como vemos quedan espacios libres en la FPGA pero este paso no es más que un medio para llegar a la reconfiguración parcial.

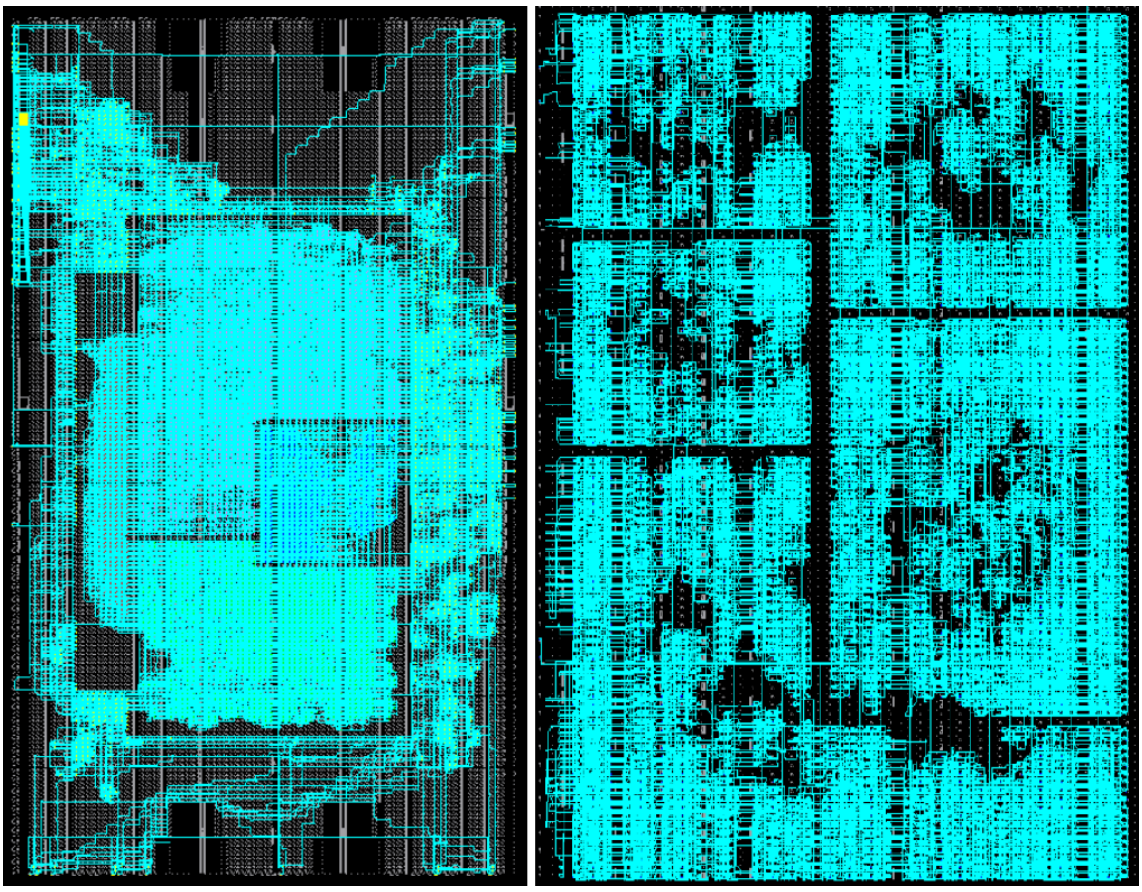


Ilustración 43. IP Cores aislados (7)

Lo que se quiere conseguir con este aislamiento de módulos es poder aplicar una reconfiguración parcial durante la ejecución del sistema. Consiguiendo reservar un espacio en la FPGA lo bastante grande para albergar cualquiera de los IP Cores de comunicación nombrados anteriormente y teniendo disponibles los bitstream de cada uno de ellos en una

memoria externa, se puede conseguir que el sistema operativo reprograme ese espacio de la FPGA para usar uno de los interfaces de comunicación cada vez que sea necesario sin tenerlos todos implementados a la vez. Esto implica un ahorro de espacio en la FPGA además de una optimización del System on Chip (SoC) ya que se podría llegar a crear un sistema genérico que se auto-reprogramara en uno específico cuando el mismo sistema operativo lo requiera.

7 Bibliografía

1. Wikipedia RTCA DO254. [En línea] https://en.wikipedia.org/wiki/DO-254.

2. Wikipedia desarrollo ágil. [En línea] http://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software.

3. Wikipedia Scrum. [En línea] http://es.wikipedia.org/wiki/Scrum.

4. Imagen Scrum. [En línea] http://epf.eclipse.org/wikis/scrum/Scrum/guidances/supportingmaterials/resources/ScrumLargeLabelled.png.

5. AIM online página 12. [En línea] http://www.aim-online.com/pdf/OVIEW429.PDF.

6. Xilinx. [En línea] http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v10_0/pg057-fifo-generator.pdf.

7. **Xilinx.** *r2-flow-guidance.*