Facultad de Ingeniería de la
Universidad de la República

Proyecto final de la asignatura
Diseño Lógico 2

CAN Opener

Sniffer de una red CAN implementado en FPGA

Autores:
José Bentancour
Damián Vallejo

Tutor:
Leonardo Etcheverry


27 de noviembre de 2017

ÍNDICE GENERAL

CAPÍTULO 1

RESUMEN

Este proyecto plantea la realización un "sniffer" de una red CAN de un auto para encontrar mayor informacion que la brindada por un escáner ODBII estándar.

OBDII es un sistema de diagnóstico de vehículos, sucesor de OBDI [1]. Permite detectar fallos eléctricos, químicos y mecánicos que pueden afectar al nivel de emisiones del vehículo. El sistema verifica el estado de todos los sensores involucrados en las emisiones, como por ejemplo la inyección o la entrada de aire al motor. Además es capaz de bridar información en tiempo real como RPM, velocidad o temperarura del líquido refrigerante. ODBII es independiente del protocolo de comunicación pero desde hace ya varias generaciones de vehículos contienen buses de comunicación CAN. Si bien la información que se puede obteber a partir de este estándar está definida, existen otras partes de los sistemas electrónicos que se encuentran en los vehículos modernos que utilizan una comunicación CAN para su operación. Es este tipo de datos que no se encuentran accesibles por el puerto ODBII los que es de interés para este proyecto obtener a través de un mecanismo de ingeniería inversa.

Un escáner popular es el ELM327 [2], basado en un microprocesador PIC, con el cual se pudo identificar el bus que se desea investigar: ISO 15765-4 (CAN 11/500) [3].

Se implementó un sistema conectando el bus CAN (CAN_H y CAN_L), con ayuda de un chip SN65HVD230 [4] que permite obtener a partir de las líneas diferenciales una señal binaria de 3.3 V que se conectó a la placa DE0.

Dados la complejidad del protocolo a la hora de transmitir y el hecho de que solo se necesita recibir datos, solo se implementó la recepción de mensajes.

Finalmete, se realizó una etapa de recepción compuesta por un buffer de lectura que se encarga de leer los bits transmitidos la red CAN, identifica el comienzo y fin de cada mensaje, sus diferentes partes y guarda la información de interés en un registro FIFO. Por otro lado, una etapa de procesamiento de datos y despliegue de información comienza en la lectura del FIFO de la etapa anterior a través de una interfaz Avalon y un programa que se ejecuta en el procesador NIOS de la placa DE0 releva estos datos y los despliega en una consola en una computadora.

CAPÍTULO 2

INTRODUCCIÓN

2.1. Objetivo

El objetivo de este proyecto es la implementación de un receptor CAN conformado por dos etapas. Una de ellas es la etapa de recepción y la otra es la etapa de procesamiento de datos y despliegue de información.

La etapa de recepción está compuesta por un buffer de lectura que se encarga de leer los bits transmitidos desde una red CAN, identificar el comienzo y fin de cada mensaje, sus diferentes partes o frames, guardar la información de interés en un registro FIFO, identificar e ignorar los bits de stuffing y verificar el CRC. La etapa de procesamiento de datos y despliegue de información comienza en la lectura del FIFO de la etapa anterior, donde está guardada la información de identifier, data length code, data y correctitud del CRC y a través de una interfaz Avalon y un programa que se ejecuta en el procesador NIOS de la placa DE0, se relevan estos datos y se despliegan en una consola en una computadora.

2.2. Alcance

Este proyecto implementa solo la etapa de recepción, no es capaz de transmitir datos hacia el bus de datos.

Las funcionalidades del receptor no son configurables en tiempo de ejecución, el caso más notorio es el baudrate, es elegido en tiempo de complilación de los bloques VHDL cambiando la relación de un divisor de frecuencia de reloj.

La implementación del software y del SoC se centra en obtener una salida de datos básica, no realiza análisis alguno sobre los datos recibidos, se encarga solamente de imprimir los datos en estado crudo por la consola.

2.3. Casos de uso

- Monitoreo y debug de redes CAN.
- Esclavos CAN que no necesiten responder con datos al bus CAN.

CAPÍTULO 3

DESCRIPCIÓN DEL SISTEMA IMPLEMENTADO

3.1. Descripción general

Visto desde fuera (Figura 3.1) el sistema desarrollado tiene las interfaces:

- **Interfaz CAN**
  - **can_rx:** señal de entrada del bus de datos.
- **Interfaz Avalon-MM**
  - **slave_read**
  - **slave_write**
  - **slave_address**
  - **slave_writedata**
  - **slave_byteenable**
  - **slave_readdata**
- **Interfaz Avalon Clock-Reset**
  - **clk:** reloj principal del sistema.
  - **reset:** reset principal del sistema.

El archivo *can_opener.vhd* se encarga de mapear las señales de *buffer_fifo.vhd* (Figura 3.2) en las direcciones de memoria de la interfaz Avalon, permitiendo la lectura de los datos del FIFO. Además genera la señal **clk_can** (velocidad de datos en el bus CAN) a partir de la entrada **clk**. La señal **clk_nios** se encuentra conectada directamente a **clk**.

Figura 3.1: *can_opener.vhd*

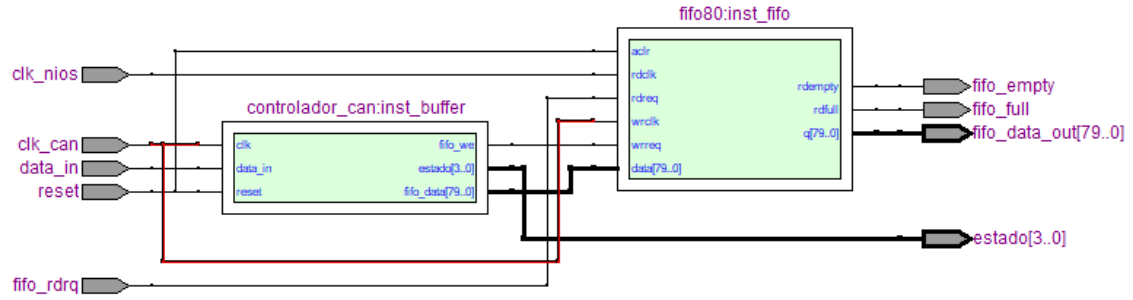

Figura 3.2: *buffer_fifo.vhd*

El archivo *controlador_can.vhd* es el encargado de decodificar los mensajes del bus y guardarlos en el FIFO *fifo80.vhd*. El FIFO fué creado con el MegaWizard y posee entradas de reloj distintas ya que la escritura se hace con el reloj del controlador **clk_can** y la lectura con el reloj de la interfaz Avalon **clk_nios**.

3.2. Implementación de la etapa de de recepción

Para implementar la etapa de recepción se diseñó un circuito en VHDL que implementa una máquina de estados para la decodificación de un mensaje CAN del tipo data frame y un registro FIFO donde se guarda cada nuevo mensaje. Para esta implementación se utilizó como refencia un proyecto libre publicado en Opencores.org escrito en Verilog [5] aunque no se utilizó ni tradujo a VHDL el código allí publicado sino que se tomaron como referencia los bloques allí implementados a nivel conceptual.

3.2.1. Mensaje CAN

El protocolo CAN [6] fue creado por la empresa alemana Bosch en 1986 y su última revisión es de 1991. Este procotolo fue creado como respuesta a la introducción de comunicación serial a un número creciente de aplicaciones y al requerimiento de estas de asignar identificadores a los mensajes para estandarizar funciones de comunicación. Existen dos tipos de especificaciones CAN, el receptor que se implementa en este proyecto utiliza la especificación 1.2 pero también existe otra llamada CAN extendido también conocido como CAN 2.0B. La principal diferencia entre ambos es el tamaño del campo identificador (11 bits en el caso tradicional y 29 en el extendido). El protocolo CAN es utilizado principalmente en el ámbito automotriz

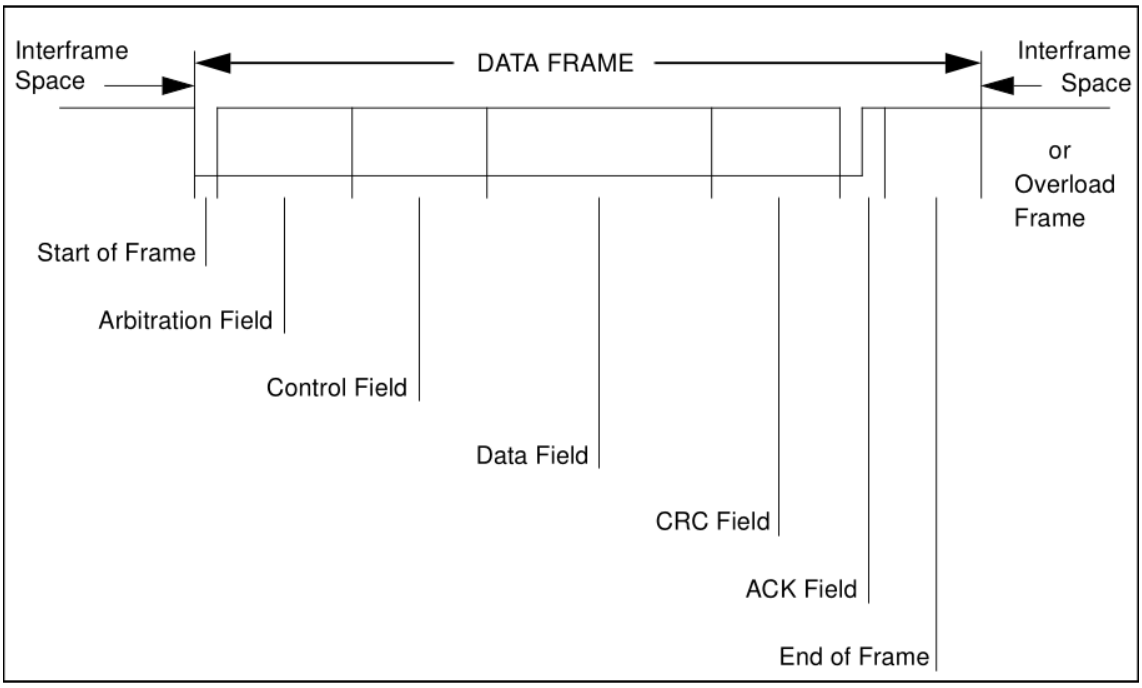

Figura 3.3: Composición del tipo de mensaje data frame según especificación 2.0.

debido a su característica de transmisión en entornos distribuidos y capacidad de comunicación entre varias CPUs.

Existen cuatro tipos de mensajes CAN:

- **Data frame:** lleva información del transmisor a los receptores.
- **Remote frame:** es transmitido por una unidad del bus para solicitar la transmisión de un data frame con el mismo identifier.
- **Error frame:** es transmitido por cualquier unidad que detecta un error.
- **Overload frame:** es utilizado para proveer un delay entre dos mensajes consecutivos.

Un mensaje CAN del tipo data frame tiene una estructura definida por diferentes campos según muestra la Figura 3.3. CAN es un protocolo donde un bit alto o "1", también llamado recesivo en la nomenclatura CAN, indica la falta de actividad en el bus y un bit bajo o "0", también llamado dominante, indica el comienzo de la actividad.

Los bits que componen este tipo de mensaje y cada uno de sus campos pueden observarse en la Tabla 3.1

A continuación una descripción más detallada de alguno de los bits o secuencias de bits nombrados en la Tabla 3.1:

- **Identifier:** Arreglo de bits que se utiliza para establecer una prioridad entre distintos dispositivos que intentan acceder a la escritura en el bus. Cualquier dispositivo que intente escribir en el bus debe verificar, primeramente que el

Campo	Largo en bits	Descripción
Start of frame	1	Dominante, indica el inicio de un nuevo mensaje
Arbitration Field	12	11 bits de indentifier y el RTR bit
Control field	6	2 bits reservados que deben ser dominantes y 4 de data length code
Data field	0-64	El payload del mensaje a ser transmitido, mensaje de largo dado por data length code
CRC Field	16	15 bits de verificación de redundancia cíclica para detección de errores y 1 bit de CRC delimiter
ACK field	2	ACK slot y ACK delimiter
End of frame	7	7 bits recesivos que indican la finalización del mensaje

Tabla 3.1: Detalle de cada uno de los campos que componen un mensaje CAN del tipo data frame.

bus no está siendo utilizado y luego en caso de que dos o más dispositivos intenten iniciar una transferencia al mismo tiempo, el dispositivo con el identifier más bajo tendrá prioridad y los demás deberán interrumpir la escritura y volver a intentarla una vez que el dispositivo con mayor prioridad finalice.

- **RTR bit:** Remote Transition Request bit, debe ser dominante en los mensajes del tipo data frame y recesivo en los del tipo remote frame.
- **2 bits reservados del control field:** deben ser dominantes en la escritura pero los receptores deben aceptar tanto recesivos como dominantes.
- **Data length code:** Cuatro bits que definen el largo en bytes del data field codificados en binario, debe ser menor a 8 y en caso de ser mayor a 8 el receptor debe interpetarlos como un 8.
- **CRC:** Verificación de redundancia cíclica, consiste en la división polinomial entre el mensaje enviado hasta el CRC field (Start of frame, Arbitration field, Control field, Data field) y 15 "0s" y el polinomio:

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$$

- **CRC delimiter:** Indica la finalización del CRC y debe ser recesivo.
- **ACK slot:** El transmisor del mensaje debe enviar este bit recesivo y los receptores deben sobreescribirlo en dominante en caso de verificar el CRC.
- **ACK delimiter:** Indica la finalización del ACK y debe ser recesivo.

Bit stuffing

Como medida complementaria al CRC, también se utiliza la herramienta del bit stuffing o relleno de bits en el protoclo CAN. Esto consiste en la prohibición dentro de un mensaje de varios bits consecutivos con la misma polaridad, en el caso de CAN dentro de un mensaje no puede haber más de 5 bits consecutivos iguales. Para esto el tranmisor debe insertar un bit de polaridad opuesta cada vez que envía 5 bits consecutivos de igual polaridad y el receptor debe detectar e ignorar el bit de relleno. Todos los frames se rellenan salvo el CRC delimiter, ACK field y End of frame. Los bits de stuffing no son considerados para el cálculo del CRC y el stuffing es recursivo (i.e. si por insertar un bit de stuffing se genera una secuencia de 5 bits iguales consecutivos contando al bit de stuffing, se deberá insertar otro de polaridad opuesta al final de la secuencia).

3.2.2. Estados de controlador_can.vhd

La primera entidad que se diseñó fue controlador_can. Esta entidad cuenta con entradas de reset y reloj, entrada de datos CAN y salidas al FIFO fifo_we y fifo_data. Los estados y la función de cada uno de ellos son detallados a continuación:

- estado_inicial: espera diez "1s" consecutivos para empezar a esperar el inicio de un mensaje
- espera_start_of_frame: espera un "0" que indica el inicio de un mensaje
- lee_arbitraje: lee los 12 bits de arbitraje (11 de dirección y 1 de transmisión de petición remota)

- lee_control: lee los 6 bits de control (extensión de identificador, reservado y 4 de longitud de datos)
- lee_data: lee los 8*longitud de datos bits de payload
- lee_crc: lee los 15 bits de CRC
- lee_end_of_frame lee el bit de delimitación de CRC y el hueco de ack
- guardar_mensaje: durante el delimitador de ack guarda los datos en un FIFO

Offset	Byte3	Byte2	Byte2	Byte0
0x00	Status	DataLen	AddresH	AddresL
0x01	Data3	Data2	Data1	Data0
0x02	Data7	Data6	Data5	Data4

Tabla 3.2: Mapa de memoria.

3.3. Interfaz Avalon

Las interfaces Avalon permiten interconectar componentes en un FPGA Intel. La familia de interfaces Avalon define interfaces apropiadas para transmitir datos de alta velocidad, leer y escribir registros y memoria, y controlar dispositivos fuera de chip.

Los tipos de interfaces definidos son:

- **Avalon Streaming Interface (Avalon-ST):** una interfaz que admite flujo unidireccional de datos, incluidos flujos multiplexados, paquetes y datos DSP.
- **Avalon Memory Mapped Interface (Avalon-MM):** una interfaz que permite lectura y escritura basada en direcciones, es una interfaz típica de una arquitectura maestro-esclavo.
- **Avalon Conduit Interface:** un tipo de interfaz que acomoda señales individuales o grupos de señales que no encajan en ninguno de los otros tipos de Avalon. Permite hacer conexiones con otros bloques que no sean Avalon o conectar pines externos del FPGA.
- **Avalon Tri-State Conduit Interface (Avalon-TC):** una interfaz que permite conexiones a periféricos fuera del FPGA. Múltiples periféricos pueden compartir pines a través de la multiplexación con compuertas tri-estado.
- **Avalon Interrupt Interface:** una interfaz que permite que los componentes señalicen eventos a otros componentes. Por ejemplo un periférico que requiera atención por parte del microprocesador.
- **Avalon Clock Interface:** una interfaz que permite recibir y generar señales de reloj.
- **Avalon Reset Interface:** una interfaz que permite la conexión de señales de reset.

Para la transferencia de los datos hacia el microprocesador se utilizó una interfaz Avalon Memory Mapped Interface (Avalon-MM), la cual mapea los datos recibidos y los estados del FIFO y decodificador CAN a direcciones de memoria. El bus de datos del microprocesador Nios II seleccionado es de 32 bits, por lo que se utilizaron tres direcciones de memoria para transmitir la totalidad de los datos. En la tabla 3.2 se observa la distribución de los mismos.

b7	b6	b5	b4	b3	b2	b1	b0
E3	E2	E1	E0	0	CRC	FUL	EMP

Tabla 3.3: Detalle bits de estado.

Los campos representan los siguientes datos:

- **Status:** Según el detalle de la tabla 3.3:
  - **E3-0:** Codifican el estado del *controlador_can.vhd*. Utilizado para la depuración de bugs en la máquina de estados.
  - **CRC:** 1 si el mensaje es correcto, 0 si el mensaje contiene errores. Utilizado para comprobar la integridad del mensaje.
  - **FUL:** 1 si el el FIFO se encuentra completo, 0 en caso contrario. Utilizado para detectar un overflow de datos.
  - **EMP:** 1 si el el FIFO se encuentra vacío, 0 en caso contrario. Utilizado para detectar la presencia de nuevos datos para leer.
- **DataLen:** contiene los 4 bits del campo Control field que indica el largo en bytes del mensaje.
- **AddressH y AddressL:** juntos contienen los 11 bits del campo Arbitration field que identifican el destinatario del mensaje.
- **DataN-0:** contienen los N+1 bytes transmitidos en el campo Data field.

Los datos que se leen son los presentes en la salida del buffer (a excepción de las señales FUL y EMP), para indicar la lectura de todos los datos al FIFO y que el mismo presente el siguiente mensaje recibido se debe realizar una escritura a la dirección 0x00 (los datos que se intenten escribir son ignorados). Las direcciones de memoria indicadas son de offset, es decir, son sumadas a una dirección base que es asignada por el SOPC Builder de Quartus.

Se utilizó una interfaz Avalon Conduit Interface para leer el estado del bus de datos a través de un pin del FPGA.

Finalmente las interfaces Avalon Clock Interface y Avalon Reset Interface proveen conexión para las señales de reloj y reset respectivamente.

3.3.1. Diagramas de tiempo

En la figura 3.4 se ven las formas de onda para una lectura y escritura de la interfaz Avalon-MM.

Lectura: El maestro levanta la señal **read** y presenta la dirección **address** y la máscara de bytes **byteenable**. Luego de que se dan estas condiciones en el flanco de subida el esclavo presenta los datos **readdata** por un único período de reloj.

En este caso el contenido de la señal **byteenable** es ignorado y se presentan todos los valores correspondientes **D0** a la dirección **A0**.

Escritura: El maestro levanta la señal de **write** y presenta la dirección **address**, la máscara de bytes **byteenable** y los datos **writedata** a escribir. Luego de que se dan estas condiciones en el flanco de subida el esclavo debe procesar los datos.

En este caso el contenido de las señales **address**, **byteenable** y **writedata** son ignorados y se realiza la read request al FIFO.

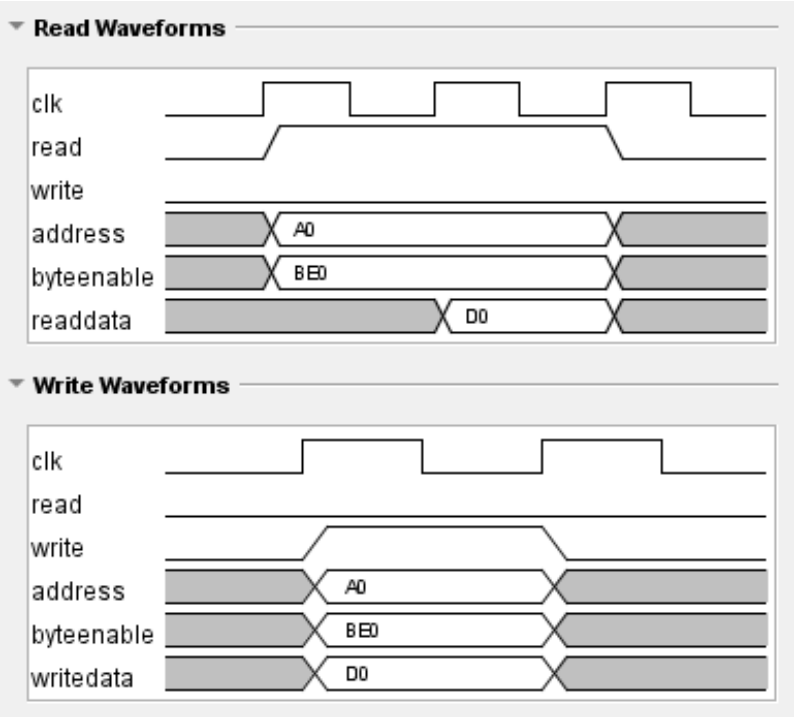

Figura 3.4: Diagrama de tiempos Avalon

CAPÍTULO 4

SISTEMA DE PRUEBA

4.1. Hardware utilizado

El sistema de prueba consiste en la interconexión de:

- Escáner OBDII (ELM327 Bluetooth)
- Arduino con placa trasnmisora CAN (MCP2515) [7]
- Sistema bajo prueba

4.2. Capa física

La comunicaión CAN se implementa en un par trenzado con señales CAN_H (CAN high) y CAN_L (CAN low). Estas señales componen a su vez una señal diferencial que va entre 0 V y 2 V nominales según se muestra en la Figura 4.1.

Dado que el FPGA maneja entradas no diferenciales de hasta 3.3 V se utilizó un chip trasnductor (SN65HVD230), el cual transforma las señales diferenciales del bus en una señal binaria de 3.3 V.

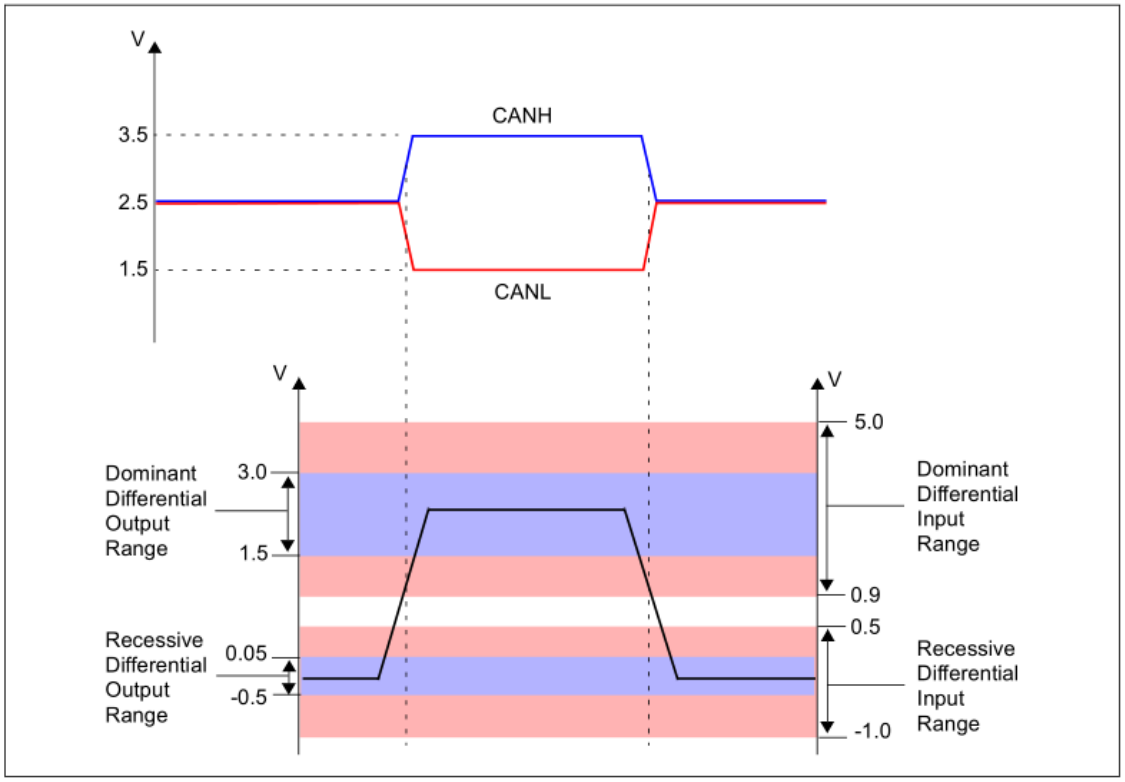

Figura 4.1: Niveles de tensión de las señales del bus CAN. Imagen obtenida de [8].

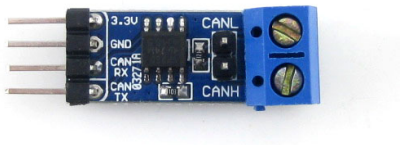

Figura 4.2: Placa con el chip SN65HVD230

En la figura 4.2 se ve la placa que se utilizó para conectar el FPGA al bus CAN y en la figura 4.3 se ve la constitución interna del mismo.

La función de los pines es la sigiente:

- **Vcc:** Fuente de alimenación de 3.3 V
- **Vref:** Salida de voltaje de referencia de $V_{cc}/2$ (no usado).
- **GND:** Conexón de masa, 0 V.
- **CANL CANH:** Conexiones al bus CAN.
- **D:** Entrada de datos para trasnmisión (no usado, solo se reciben datos).
- **R:** Salida de datos recibidos, LOW para el estado dominante y HIGH para estado recesivo (entrada de datos al FPGA).
- **Rs:** Selección del modo de funcionamiento del chip, GND = high speed mode y VCC = low power mode.

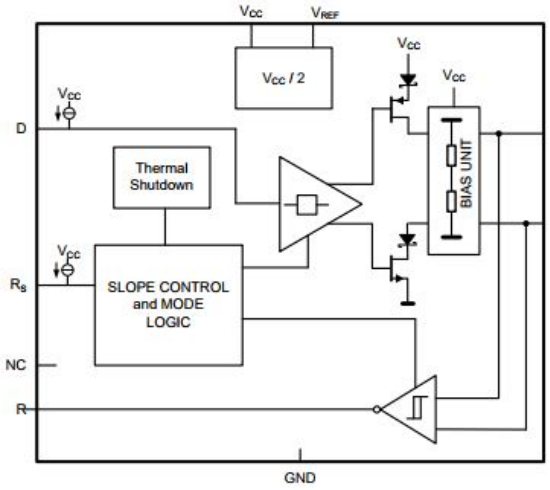

Figura 4.3: Esquemático interno SN65HVD230

La alimentanción es realizada desde la placa DE0 y se conecta el pin **R** a uno de los pines de banco GPIO0.

4.3. Diagrama de bloques

En la figura 4.4 se muestra un diagrama de la interconexión del hardware utilizado para realizar las pruebas.

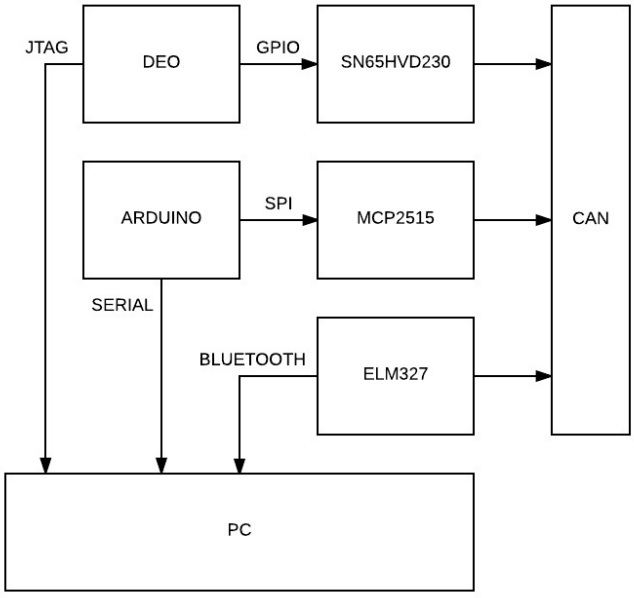

Figura 4.4: Esquemático del sistema de prueba

4.4. Arquitectura del SoC

En conjunto con la interfaz desarrollada se encuentran otros componenetes que hacen a una aplicación funcional, los mismos son:

- **Nios II:** microprocesador que corre la aplicación.
- **OnChip memory:** memoria donde se almacena (ROM) y corre la aplicación (RAM), la misma está embebida dentro del FPGA.
- **JTAG UART interface:** interfaz serial sobre JTAG, permite la cominicación de la consola a través de USB Blaster.
- **Parallel IO interface:** interfaz que permite manejar los leds incluidos en la placa con propósitos de debugueo.

⊟ **cpu**	Nios II Processor				
instruction_master	Avalon Memory Mapped Master	**clk_0**			
data_master	Avalon Memory Mapped Master			IRQ 0	IRQ 31
jtag_debug_module	Avalon Memory Mapped Slave			**0x00004800**	0x00004fff
⊟ **jtag_uart**	JTAG UART				
avalon_jtag_slave	Avalon Memory Mapped Slave	**clk_0**		**0x00005020**	0x00005027
⊟ **onchip_memory2**	On-Chip Memory (RAM or ROM)				
s1	Avalon Memory Mapped Slave	**clk_0**		**0x00002000**	0x00003fff
⊟ **pio_led**	PIO (Parallel I/O)				
s1	Avalon Memory Mapped Slave	**clk_0**		**0x00005000**	0x0000500f
⊟ **can_opener_inst**	can_opener				
slave	Avalon Memory Mapped Slave	**clk_0**		**0x00005010**	0x0000501f

Figura 4.5: SOPC

Visto desde fuera el sistema tiene las interfaces mostradas en la figura 4.6.

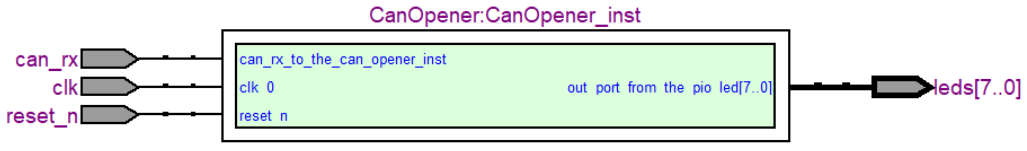

Figura 4.6: *CanOpener_inst.png*

4.5. Software

El software fue desarrollado en el IDE de Nios II en lenguaje C++. Se crearon funciones auxiliares para extraer los datos de la interfaz Avalon:

- **isEmpty()** devuelve el estado de la flag EMP.
- **isFull()** devuelve el estado de la flag FUL.
- **checkCRC()** devuelve el estado de la flag CRC.
- **available()** devuelve el true si hay nuevos datos para leer, false en caso contrario.
- **getAddr()** devuelve la dirección del destinatario del mensaje.
- **getDataLen()** devuelve el largo en bytes del mensaje.
- **getData(int i)** devuelve el contenido de byte i del mensaje.
- **next()** confirma la lectura de los datos y avanza el FIFO.

El software comprueba constantemente la presencia de nuevos datos recibidos e imprime en consola su contenido.

CAPÍTULO 5

VALIDACIÓN DEL PROYECTO

5.1. Pruebas realizadas

Durante el desarrollo del decodifigador se utilizaron pruebas de simulación y con hardware dentro del mismo FPGA: **Prueba 1:** dentro del software Quartus se simuló el comportamiento del mimso ante una froma de onda conocida.

Prueba 2: se creó un bloque en VHDL que inyectaba señales predefinidas en el pin de **can_rx**. También se probó con este mecanismo el software que corría en el microprocesador.

Con el sistema terminado y el hardware conectado se relizaron dos pruebas:

Prueba 3: el escáner ELM327 envía mensajes al bus, se pone MCP2515 en modo de recepción y se verifica que el sistema bajo pruba reciba el mismo mensaje. Con esta prueba encontramos los errores:

- Necesidad de operar el FIFO con diferentes fuentes de reloj para lectura y escritura.

Prueba 4: el escáner ELM327 se desconecta, se pone MCP2515 en modo de transmisión constante y se verifica que el sistema bajo pruba reciba los mensajes correctamente. Con esta prueba encontramos los errores:

- Se encontraron bucles infinitos en la máquina de estados del decodificador que se daban para determinadas condiciones de error en el bus de datos.

Dirección	Eventos que generan cambio en los datos
0x5D0	Posición de la llave
0x470	Luces, estado de las puertas
0x50	Estado de los cinturones de seguridad
0x280	Estado del acelerador y embrague (accionado) y posición del acelerador

Tabla 5.1: Direcciones y eventos que generan cambio en los datos

5.2. Utilización para ingeniería inversa

Una vez implementado y probado el sistema, se probó conectándolo a un bus CAN en un auto. De manera de obtener en qué direcciones y de qué manera se mapean los datos; se observaron las direcciones una por una, se realizaron diferentes eventos (e.g. abrir una puerta, dar contacto al motor, accionar el acelerador, etc.) y se observó si había algún cambio en el payload de los mensajes recibidos. Se logró deducir que en el punto de la red observado pasan los mensajes con las direcciones según se muestra en la Tabla tabla 5.1.

CAPÍTULO 6

CONCLUSIONES

6.1. Cumplimiento de los objetivos

Se puede decir que el objetivo fue cumplido, se creó un decodificador de mensajes CAN con interfaz Avalon. El mismo fue probado en un entorno real y fue utilizado con el propósito inicial de relizar ingeniería inversa en un bus de datos de un vehículo.

6.2. Para mejorar

- La generación de la señal de reloj para la máquina de estados es completamente asincónica con el bus, para obtener menores errores a la hora de decodificar mensajes sería mejor tener un mecanismo de sinconización con el inico de un nuevo frame de manera de obtener un uinstante de muestreo óptimo.
- Ninguno de los parámetros del decodificador es modificable en tiempo de ejecución, por ejemplo, el baudrate se define antes de la compilación y depende de la frecuencia de reloj del sistema.

REFERENCIAS

[1] P. Baltusis, *OBD SAE Communication Standards Update*. 2009.

[2] *ELM327 - OBD to RS232 Interpreter*. ELM Electronics, 2014.

[3] *Road vehicles. Diagnostics on Controller Area Networks (CAN)*. ISO, 2005.

[4] *SN65HVD23x 3.3-V CAN Bus Transceivers*. Texas Instruments, 2015.

[5] I. Mohor, "Can protocol controller." `https://opencores.org/project,can`, 2009.

[6] Robert Bosch GmbH, *CAN Specification 2.0*. 1991.

[7] *MCP2515 Stand-Alone CAN Contoller with SPI Interface*. Microchip Technology Inc., 2005.

[8] P. Richards, *A CAN Physical Layer Discussion*. 2002.

CAPÍTULO 7

ANEXO

7.1. Hoja de datos

7.1.1. Descripción general

Decodificador de mensajes CAN con interfaz Avalon-MM.

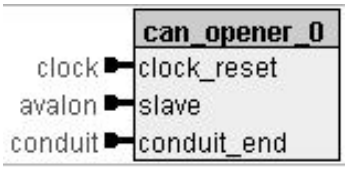

Figura 7.1: IP core

7.1.2. Características

- Los mensajes son guardados en un FIFO.
- La información de FIFO lleno o vacío está disponibles para la detección de nuevos mensajes y overflow del FIFO.
- Chequeo de CRC para asegurar la integridad del mensaje recibido.

Offset	Byte3	Byte2	Byte2	Byte0
0x00	Status	DataLen	AddresH	AddresL
0x01	Data3	Data2	Data1	Data0
0x02	Data7	Data6	Data5	Data4

Tabla 7.1: Mapa de memoria.

b7	b6	b5	b4	b3	b2	b1	b0
E3	E2	E1	E0	0	CRC	FUL	EMP

Tabla 7.2: Detalle bits de estado.

7.1.3. Descripción funcional

En la tabla 7.1 se observa el mapeo de los datos recibidos en memoria de la interfaz Avalon-MM.

Los campos representan los siguientes datos:

- **Status:** Según el detalle de la tabla 7.2:
  - **E3-0:** Codifican el estado del *controlador_can.vhd*. Utilizado para la depuración de bugs en la máquina de estados.
  - **CRC:** 1 si el mensaje es correcto, 0 si el mensaje contiene errores. Utilizado para comprobar la integridad del mensaje.
  - **FUL:** 1 si el el FIFO se encuentra completo, 0 en caso contrario. Utilizado para detectar un overflow de datos.
  - **EMP:** 1 si el el FIFO se encuentra vacío, 0 en caso contrario. Utilizado para detectar la presencia de nuevos datos para leer.
- **DataLen:** contiene los 4 bits del campo Control field que indica el largo en bytes del mensaje.
- **AddressH y AddressL:** juntos contienen los 11 bits del campo Arbitration field que identifican el destinatario del mensaje.
- **DataN-0:** contienen los N+1 bytes transmitidos en el campo Data field.

Los estados y la función de cada uno de ellos son detallados a continuación:

- **E=0000** estado_inicial: espera diez "1s" consecutivos para empezar a esperar el inicio de un mensaje
- **E=0001** espera_start_of_frame: espera un "0" que indica el inicio de un mensaje
- **E=0010** lee_arbitraje: lee los 12 bits de arbitraje (11 de dirección y 1 de transmisión de petición remota)
- **E=0011** lee_control: lee los 6 bits de control (extensión de identificador, reservado y 4 de longitud de datos)
- **E=0100** lee_data: lee los 8*longitud de datos bits de payload

- **E=0101** lee_crc: lee los 15 bits de CRC
- **E=0110** lee_end_of_frame lee el bit de delimitación de CRC y el hueco de ack
- **E=0111** guardar_mensaje: durante el delimitador de ack guarda los datos en un FIFO

7.1.4. Conexiones y diagramas de tiempo

Figura 7.2: *can_opener.vhd*

- **Interfaz CAN**
  - **can_rx:** señal de entrada del bus de datos.
- **Interfaz Avalon-MM**
  - **slave_read**
  - **slave_write**
  - **slave_address**
  - **slave_writedata**
  - **slave_byteenable**
  - **slave_readdata**
- **Interfaz Avalon Clock-Reset**
  - **clk:** reloj principal del sistema.
  - **reset:** reset principal del sistema.

En la figura 7.3 se ven las formas de onda para una lectura y escritura de la interfaz Avalon-MM.

Lectura: El maestro levanta la señal **read** y presenta la dirección **address** y la máscara de bytes **byteenable**. Luego de que se dan estas condiciones en el flanco de subida el esclavo presenta los datos **readdata** por un único período de reloj.

Escritura: El maestro levanta la señal de **write** y presenta la dirección **address**, la máscara de bytes **byteenable** y los datos **writedata** a escribir. Luego de que se dan estas condiciones en el flanco de subida el esclavo debe procesar los datos.

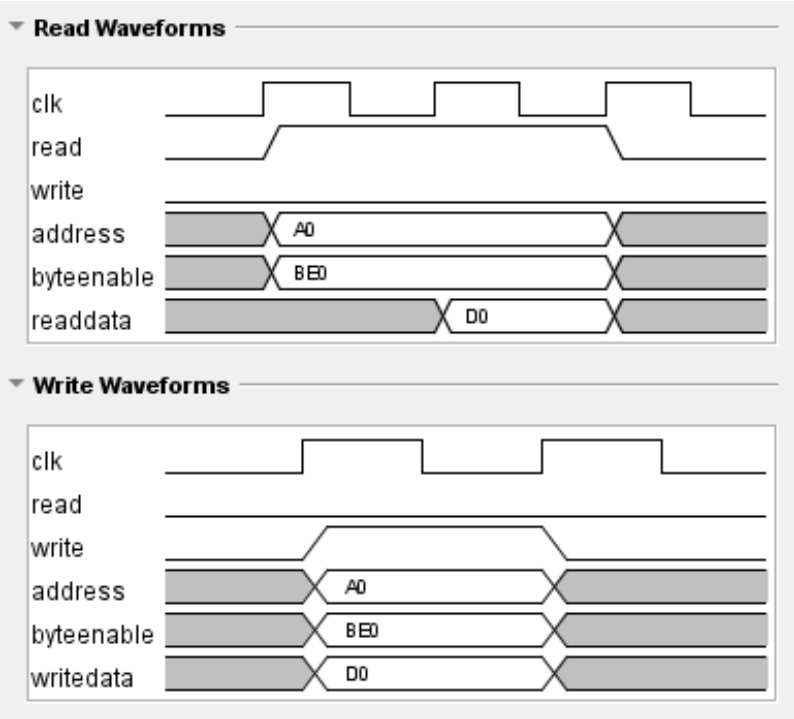

Figura 7.3: Diagrama de tiempos Avalon

7.1.5. Estructura interna

El archivo *can_opener.vhd* se encarga de mapear las señales de *buffer_fifo.vhd* (figura 7.4) en las direcciones de memoria de la interfaz Avalon, permitiendo la lectura de los datos del FIFO. Además genera la señal **clk_can** (velocidad de datos en el bus CAN) a partir de la entrada **clk**. La señal **clk_nios** se encuentra conectada directamente a **clk**.

El archivo *controlador_can.vhd* es el encargado de decodificar los mensajes del bus y guardarlos en el FIFO *fifo80.vhd*. El FIFO posee entradas de reloj distintas ya que la escritura deben hacerse con el reloj del controlador **clk_can** y la lectura con el reloj de la interfaz Avalon **clk_nios**.

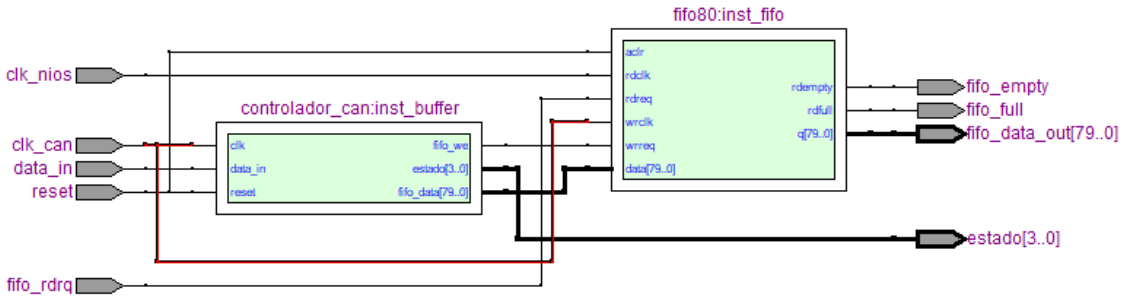

Figura 7.4: *buffer_fifo.vhd*