

DEPARTMENT OF ELECTRONICS
AND TELECOMMUNICATIONS

SERIAL DATA TRANSMISSION AND PROCESSING OF AUDIO SIGNALS USING FPGA'S

AUTHOR:

Verónica Gabriela Ordóñez León

In order to obtain the degree of Electronics and Telecommunications Engineering

DIRECTOR:

Dr Jorge E. Parra

ALBUQUERQUE – UNITED STATES OF AMERICA

2008

DECLARACIÓN CESIÓN DE DERECHOS

Yo, Verónica Gabriela Ordóñez León, declaro conocer y aceptar la disposición del Art. 67 del Estatuto Orgánico de la Universidad Técnica Particular de Loja que en su parte pertinente textualmente dice: “Forman parte del patrimonio de la Universidad la propiedad intelectual de investigaciones, trabajos científicos o técnicos y tesis de grado que se realicen a través, o con el apoyo financiero, académico o institucional (operativo) de la Universidad.

F……
Verónica Gabriela Ordóñez León

DECLARACIÓN DE AUTORÍA DEL TEMA DE TESIS

rpt

Los conceptos, interpretaciones e ideas en el presente proyecto son plasmados bajo responsabilidad exclusiva de la autora.

Autora

F: ……
Verónica Gabriela Ordóñez León

INTRODUCTION

Nowadays, FPGA's (field-programmable gate array) are gaining importance in both, the commercial and research area. Since this technology is far more affordable than custom-manufactured ASICs, smaller companies and universities can take advantage of the capabilities of large-scale integration (1). Thus, implementations on hardware might have interesting and cheaper applications for both, educational and research purposes.

FPGA's are an extremely valuable tool in teaching VLSI (Very-large-scale integration) design, and working on a real hardware environment can motivated the students. Programming a configurable device like a FPGA has the benefit of exposing students to a sophisticated hardware and software when designing interactive programs and circuits (1). Moreover, diverse and innovative applications can be achieved, since boards like (the one is used in this implementation) the XUP Virtex II Pro, allow to plug in or connect a microphone, speaker, mouse, serial cable, screen, etc.

Audio Processing and Serial Transmission are two applications of the FPGA's field. The present implementation joins the two mentioned applications to construct a system that is able to record voice and playback it. These two actions are controlled by HyperTerminal, which requires setting up a serial connection. A design tool, the ISE 9.2i, is used in order to describe pieces of hardware on VHDL. Since this system is constructed by modules and components, they can be used separately to achieve similar or different applications that required more complexity.

The first chapter is a brief introduction of the general approach to the architecture and the components of the system. Chapter II is about the protocols, interfaces and hardware to be described to achieve a properly serial communication between the computer and the FPGA. Chapters III and IV show the procedure to open an audio path using the audio codec of the board; and the implementation of a finite state machine that makes defined tasks to be performed orderly, respectively. Finally, some potential applications, an illustrative description of the implementation and results are shown.

This report is a documentation of the knowledge acquired about hardware description in the "Hardware Design with VHDL" class. It was developed during class and winter break

under the direction of Dr Jorge Parra, researcher of The University of New Mexico at the Electrical Engineering Department.

GENERAL OBJECTIVE AND SPECIFIC OBJECTIVES

General Objective

This implementation is aimed to contribute to extend the spectrum of applications to be developed on the line of description of hardware locally and nationally, and to be taken as a teaching and learning tool. From the following described system, more complex descriptions and applications can be developed in serial communications and audio processing fields, thus, a better understanding about the hardware is being used is achieved.

Specific Objectives

- ✓ To record a few seconds of audio into a memory, hooking up a microphone to the audio line port of the XUP Virtex II Pro.
- ✓ To make the XUP board to play the audio signal back over the speaker connected to its amp-out port when a button is pushed.
- ✓ To use an UART open core to achieve RS232 interface using Wishbone protocol.
- ✓ To transmit serially characters from a computer to an FPGA.
- ✓ To control the audio recording and playback through HyperTerminal, using a standard computer (laptop/desktop).

DEDICATION

A mis padres,
por su sacrificio, apoyo, trabajo,
ejemplo, sueños y decisiones que
hoy me hacen la persona que soy.

A mis hermanas,
por todas las alegrías
compartidas, la confianza y apoyo.

A mis compañeros y amigos,
Johana, Maria de los Ángeles, Axel y Lenin
por toda su ayuda y apoyo en la cercanía y a la distancia.

A Josh,
por permitirme ver el mundo a través de sus ojos.

ACKNOWLEDGMENT

Quiero agradecer al Dr Jorge E. Parra, por cada valiosa hora de su tiempo frente al computador, y por las lecciones de perseverancia y optimismo. De la misma manera, agradezco a Craig Kief, por facilitarme el software y hardware necesario en el laboratorio de UNM.

De manera especial quiero expresar mi agradecimiento al Ing. Jorge Jaramillo Pacheco, por todo el apoyo recibido antes, durante y después del intercambio. Gracias por el tiempo invertido en gestiones y papeleos. Ante todo, gracias por la motivación y por creer en mí.

A todos los Ingenieros e Ingenieras de la Escuela de Electrónica y Telecomunicaciones, por todo lo aprendido en los cinco años de carrera, y por su generosidad en cuanto a transmitir conocimiento.

CONTENT

Declaración de Cesión de Derechos ..... **I**

Declaración de Autoría del Tema de Tesis ..... **II**

Introduction ..... **III**

General Objective and Specific Objectives ..... **IV**

Dedication ..... **V**

Acknowledges ..... **VI**

Content ..... **VII**

List of Figures ..... **IX**

List of Tables ..... **X**

Chapter I: General Aspects **1**

1.1 Introduction 1
1.2 Hardware Description using VHDL 1
1.3 Approaches to the Problem 2
1.3.1 Voice Recorder 2
1.3.2 Controlling Audio Recording and Playback through HyperTerminal 6
1.4 Architecture **7**

Chapter II: Serial Communication **9**

2.1 Serial Communication 9
2.1.1 Asynchronous Communication 9
2.1.2 Operation Speed 10
2.1.3. RS-232 Serial Interface 10
2.2 WISHBONE Protocol 11
2.3 UART 12

Chapter III: Audio Processing **23**

3.1 Introduction 13
3.2 AC'97 Audio Codec 13

3.2.1 Block Diagram 14

Chapter IV: Finite State Machine 15

4.1 Description 15

Chapter V: Potential Implementations 18

5.1 Using the xupv2pro_wrapper module 18
5.2 Using Interface and wrapper modules 19

Chapter VI: Demo 18

6.1 Description 21
6.2 ChipScope shots 23

Chapter VII: Conclusions y Recommendations 22

7.1 Conclusions 25
7.2 Recommendations 26

References 27
Annexes 28

LIST OF FIGURES

Figure 1.1 xupv2pro_wrapper architecture 2

Figure 1.2 Core Schematic Symbol 3

Figure 1.3 Counter and Block Memories..... 5

Figure 1.4 Counter Scheme 5

Figure 1.5 General System State Machine 6

Figure 1.6 System Architecture 7

Figure 2.7 Serial Communication Parameters 9

Figure 2.8 Male DB-9 connector 9

Figure 2.9 WISHBONE Protocol Signal Exchange 11

Figure 2.10 System Block Diagram..... 12

Figure 3.11 Block Diagram LM4550..... 14

Figure 4.12 audio_cw.vhd state machine..... 15

Figure 4.13 Selector Description..... 16

Figure 4.14 Reception and comparison of characters of control, Init_all file..... 16

Figure 4.15 Design Structure 17

Figure 5.16 Adding a filter 18

Figure 5.17 Sine generator 19

Figure 5.18 Serial Transmission of audio signals 20

Figure 5.19 Serial Transmission of coded audio signals..... 20

Figure 6.20 USB-to-Serial Converter 21

Figure 6.21 Microphone and Speaker connected 18

Figure 6.22 Com Port selection..... 19

Figure 6.23 HyperTerminal Settings 19

Figure 6.24 Screen of HyperTerminal 20

Figure 6.25 Hello 20

Figure 6.26 Baby 21

LIST OF TABLES

Table 1.1 Core Signal Pinout 4

Table 1.2 RAM blocks available 4

Table 1.3 Pushbutton Distribution 6

Table 1.4 Characters of control 7

Table 2.5 Pin Description (male connector) 10

CHAPTER 1

GENERAL ASPECTS

1.1 Introduction

This chapter is divided in two parts. The first one describes the construction of a voice recorder controlled using 2 pushbuttons of the XUP board Virtex II Pro. The voice recorder makes use of the AC'97 Codec that allows converting analog to digital signals and acquiring and playback audio signals. Additionally, an AC'97 controller is described, in order to manage inputs and outputs and convert from serial to parallel bit stream. The second part is about serial transmission from a computer to a FPGA and vice versa. Using the voice recorder of the first part, recording and playback now are controlled through HyperTerminal. An UART open core is used, and wishbone protocol is implemented to provide handshaking to the serial transmission, both are about to be described in next chapters.

1.2 Hardware Description using VHDL

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a language that allows describing and modeling digital systems, that is, it makes possible to analyze, and evaluate the behavior of a digital electronic system.
It is particularly suited as a language to describe the structure and behavior of digital electronic hardware designs, such as ASICs and FPGA's (2).

On VHDL a hierarchy design is necessary to handle large hardware implementations; subsequently, components are ordered from the top level to the bottom level. The complete description of hardware of the system is made on VHDL. ISE 9.2i (Integrated Software Environment) allows the design of Xilinx FPGA and CPLD devices in this language. As well, Chipscope 9.2i is used as verification software.

1.3 Approaches to the Problem

The system is built in two parts. The acquisition of audio signals problem is solved first by building a voice recorder. In the second part, serial communication is added to the first part.

1.3.1 Voice Recorder

In the construction of a voice recorder, components, cores, and VHDL modules are used. The implementation is described in the following steps.

Step 1._ The main VHDL module is the xupv2pro_wrapper. It is used to interface a control VHDL file to external components, and to provide synchronization with AC97 codec. The architecture is the following:

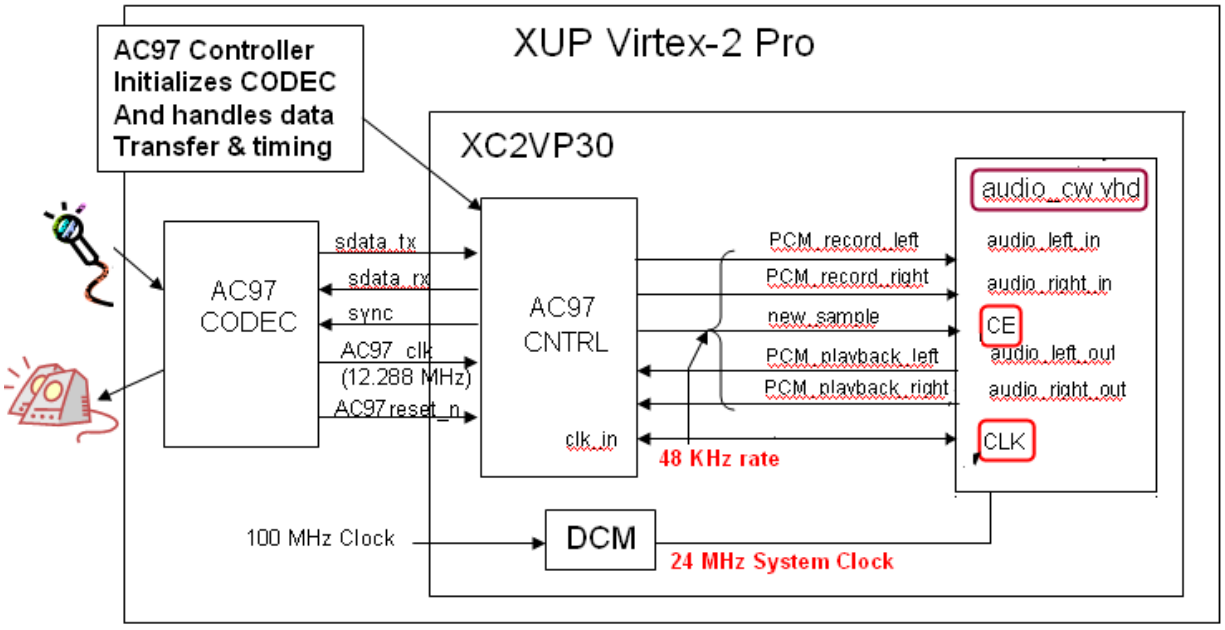


Figure 1.1 xupv2pro_wrapper architecture (3)

The AC97 is a serial interface, so data is transmitted to and from the codec one bit at a time. On every cycle of the AC97 bit clock, one bit of data is transferred from the AC97 controller (the FPGA) to the codec over the *sdata_out* wire, and one bit of data is transferred from the codec to the FPGA over the *sdata_in* wire. The AC97 CNTRL is

used to initialize the CODEC and perform a serial/parallel conversion. A 12.288 MHz bit clock is generated by the codec (3).

After serial/parallel conversion at the AC97 CNTRL: PCM_record right and left, and PCM_playback right and left as well, are now input/output of the next component *audio_cw* (24 MHz master clock). *new_sample* wire enables the CE signal; the sample rate of this wire is 48 kHz, which is the professional digital audio sample rate. Every time *new_sample* is equals to 1, *audio_cw* knows there are new data. The function of *audio_cw* will be described in step number 4.
Data passing from the codec to the FPGA are divided into frames. The start of each frame is indicated by a rising edge of the *SYNC* signal. The wrapper module waits for a pulse (sync = 1) and transmits the frame: tags and channel configuration and data.

Step 2._ In order to record voice, a Single-Port Block Memory core is used.
The Single-Port Block Memory module is generated based on the user-specified width and depth, 71424 and 16 respectively (these values are gotten afterward). During a write operation (WE asserted), the data presented at the port's data input are stored in memory at the location selected by the port's address input (4). The scheme of this core is shown below. Specifically, only 5 pins of 10 available are used.

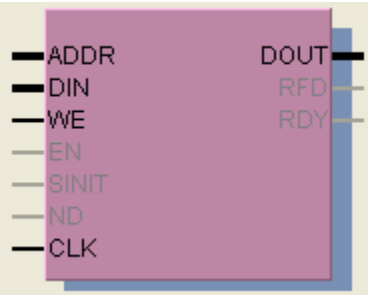


Figure 1.2 Core Schematic Symbol (4)

Signal DIN is connected to one of the input channels, right or left. Thus, for a stereo input, two cores are needed since there is only one input port in this core. The addresses are provided for a counter (figure 1.3), which is to be described in the next step.

Table 1.1 Core Signal Pinout (4)

Signal	Direction	Description
DIN[n:0] (optional)	Input	**Data Input**: Data written into memory.
ADDR [m:0]	Input	**Address**: Memory location for data written to/read from.
WE (Optional)	Input	**Write Enable**: Allows data transfer into memory.
CLK	Input	**Clock**: All memory operations synchronous with rising or falling edge of clock input, depending on user configuration of the clock pin polarity. When memory is enabled, all control signals, input/output data are registered on the rising or falling edge of clock.
DOUT[n:0]	Output	**Data Output**: Synchronous output of the memory.

According to the specifications of the XUP board Virtex II Pro, the available number of blocks for the RAM memory is:

Table 1.2 RAM blocks available (5)

Device(1)	Block SelectRAM+		DCMs	Maximum User I/O Pads
	18 Kb Blocks	Max Block RAM (Kb)		
XC2VP30	136	2,448	8	644

In chapter 3, it will be mentioned that it is necessary to use another memory (smaller), so in this case only 127 blocks are used to store data.

$$127 \times 18\ Kb = 2285568\ bits = 285694\ bytes$$

Taking into account a stereo input is wanted, two channels are needed.

$$285696 / 2\ channels = 142848$$

Since every word has 16 bits, there are 71424 words per channel. So this memory is able to store data from 0 to 71424 -1 addresses (116FF). For a stereo input the recording time is $142848 \times 1 / 48x10^3 = 1.488$ s.

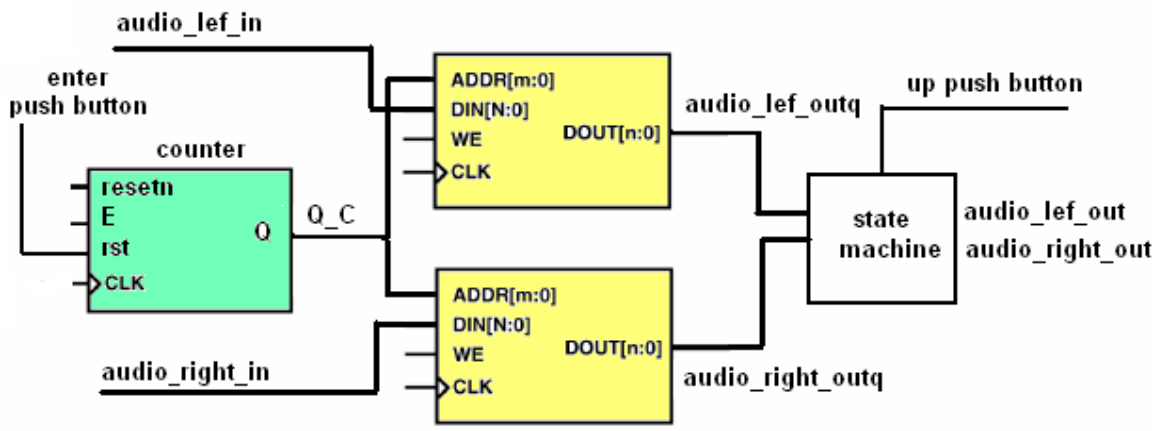


Figure 1.3 Counter and Memories

Step 3._ It is required to create a component that provides to the memory the addresses where the data are allocated. A counter is the answer to this requirement. As it was seen, for a stereo input, it is needed to count from 0 to 116FF (see previous step). So, a 17 bits counter should be implemented. In the following scheme signals of the counter are shown. The clock frequency is 48 KHz, since codec's ADCs and DCAs operate at a 48 KHz sample rate. When signal Start_1 equal to 0 at the counter, the count is set to 0. Every time E (enable) is equal to 1 the counter starts to count.

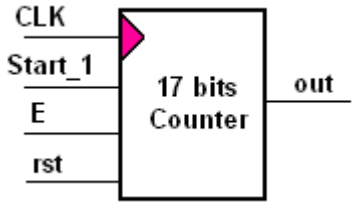


Figure 1.4 Counter Scheme

Step 4._ audio_cw file is the one that receives inputs from the AC'97 CNTRL, audio_left_in and audio_right_in; and transmits as well two outputs, audio_left_out and audio_right_out. The described memory and counter are contained and connected on this file.

Some mean is required to activate the counter, memory, recording, playback and other actions orderly. A finite state machine is implemented in this file in order to do it. Two pushbuttons of the board are the inputs of this state machine: record (center button), playback (up button).

Table 1.3 Pushbutton Distribution

N	Pushbutton location	Function
1	Center	Record
2	Up	Playback

After recording and playback buttons are pushed, a sequence of code instructions is executed. The basic scheme of states in this machine is the following.

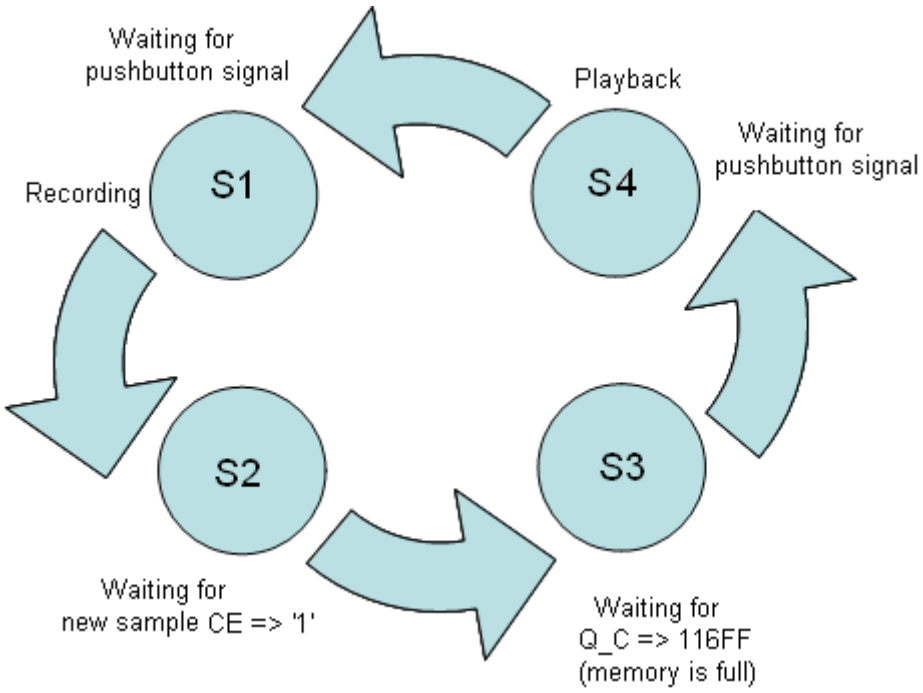


Figure 1.5 General System State Machine

1.3.2 Controlling Audio Recording and Playback through HyperTerminal

For the voice recorder described previously, recording and playback are controlled using pushbuttons. Some changes are needed to be made on it, in order to control the system using a standard computer.

The objective is to record the signal played through the microphone in a memory system and replay the same when desired; but now when r (record) and p (playback) characters are transmitted through the RS232 interface.

Table 1.4 Characters of control

N	Character	ASCII	Function
1	R	72	Record
2	P	70	Playback

Another VHDL module is added to the structure; in this case, *MiniUART* and *Init_all* (WISHBONE handshaking) components are joined in *Interface*. The UART is designed to make possible the communication between the computer and the FPGA. A HyperTerminal connection needs to be set up in order to transmit characters of control. A serial cable is used to connect the interfaces RS-232 on the computer and the FPGA. Characters of control have to be specified in this VHDL module in ASCII format (Table 1.4). By comparing the received character with the specified one, recording and playback are or are not executed. It is important to mention that since two VHDL modules are needed to be connected, a top level VHDL Module is created to be on first place in the hierarchy of the whole system: *Launion.* It can be seen that individual modules become components when joined in a top level vhdl file.

1.4 Architecture

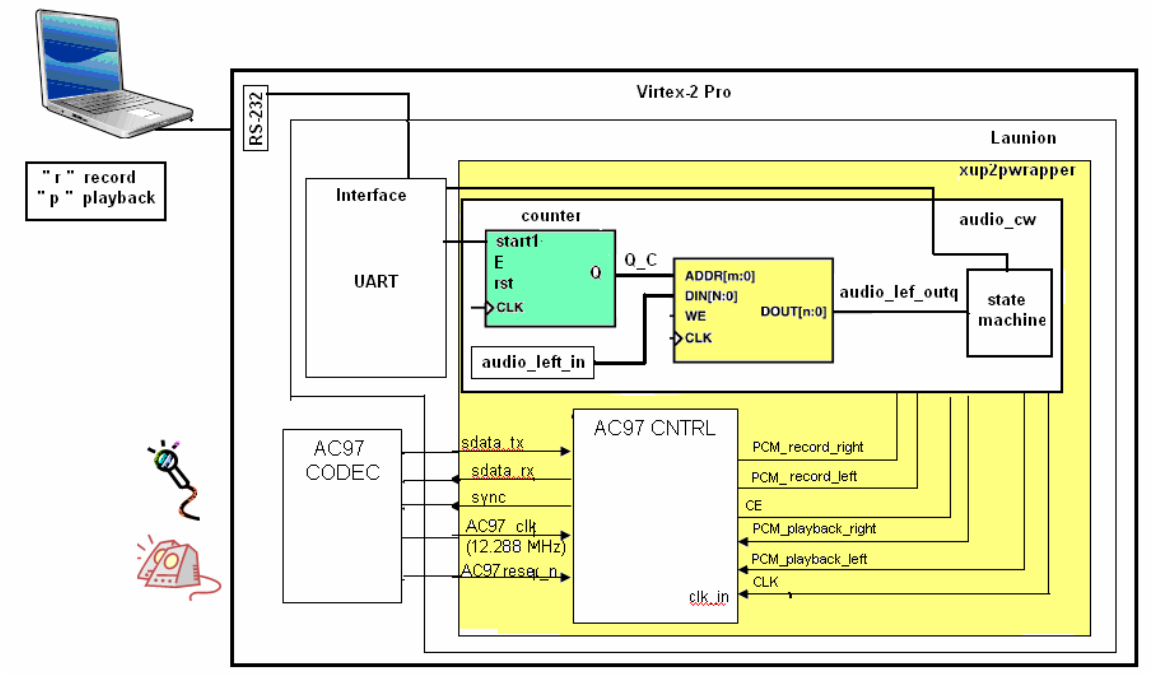


Figure 1.6 System Architecture

Figure 1.6 shows the final architecture of the system. As it can be seen, only one block memory is used, so a mono input is acquired. Components and modules that were already described are also seen in this figure.

CHAPTER II

Serial Communication

2.1 Serial Communication

In a serial communication, data are sent one bit at a time. One wire is used for each direction. Since computers usually need at least several bits of data, the data is serialized before being sent. Data is commonly sent by chunks of 8 bits. The LSB (data bit 0) is sent first, the MSB (bit 7) last (6).

2.1.1 Asynchronous Communication

The RS232 interface uses an asynchronous protocol, which means that no clock signal is transmitted along with data. The receiver has to have a way to time itself to the incoming data bits (6). In this interface both sides of the cable agree in advance on the communication parameters (see Figure 2.7). These parameters are set manually before communication starts.

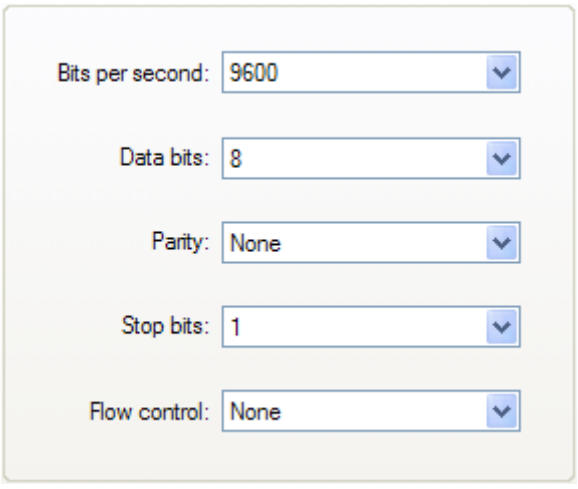


Figure 2.7 Serial Communication Parameters

Once the communications starts, the transmitter sends a "1" when the line is idle, and a "start" (“0”) before each byte is transmitted, so that the receiver can understand that data is coming. After the "start", data comes in with the agreed speed and format, so the receiver can interpret it. After each data byte the transmitter sends a "stop" ("1") (6).

2.1.2 Operation Speed

The speed is specified in bauds (bits-per-seconds that can be sent). Commonly RS-232 implementations do not allow just any speed to be used. Some standard speed values are: 1200, 9600, 38400, and 115200 bauds. The operation speed of the system is 9600 bauds (6).

2.1.3 RS-232 Serial Interface

The RS-232 Serial Interface uses a 9 pin connector DB-9 (newer laptops do not have serial port anymore, so a USB to serial converter is required). It allows bidirectional full-duplex communication (the PC can send and receive data at the same time).

A DB-9 male connector and its pin description are shown in the figures below. It has 9 pins, and among them the important ones are: pin 2, RD (receive data); pin 3, TD (transmit data); and pin 5, GND (ground). Using just 3 wires, data can be sent and received.



Figure 2.8 Male DB-9 connector (6)

Table 2.5 Pin Description, male connector (6)

Pin	Name	Dir	Notes/Description
1	DCD	IN	Data Carrier Detect. Raised by DCE when modem synchronized.
2	RD	IN	Receive Data (RD, Rx). Arriving data from DCE.
3	TD	OUT	Transmit Data (TD, Tx). Sending data from DTE.
4	DTR	OUT	Data Terminal Ready. Raised by DTE when powered on. In auto-answer mode raised only when RI arrives from DCE.
5	GND	-	Ground
6	DSR	IN	Data Set Ready. Raised by DCE to indicate ready.
7	RTS	OUT	Request To Send. Raised by DTE when it wishes to send. Expects CTS from DCE.
8	CTS	IN	Clear To Send. Raised by DCE in response to RTS from DTE.
9	RI	IN	Ring Indicator. Set when incoming ring detected - used for auto-answer application. DTE raised DTR to answer.

2.2 WISHBONE Protocol

Basically, the WISHBONE protocol is a set of handshake signals carried out in a systematic exchange fashion in order to maintain synchronization between two systems. WISHBONE System-on-Chip Interconnect protocol can be used as an interface to cores that require interfacing to other cores inside a chip (FPGA, ASIC, etc.). So the wishbone protocol allows a synchronized communication that uses a UART (an asynchronous module) to perform a serial transmission.

The WISHBONE standard is not copyrighted, and is in the public domain. It may be freely copied and distributed by any means. The handshake signaling is shown as follows.

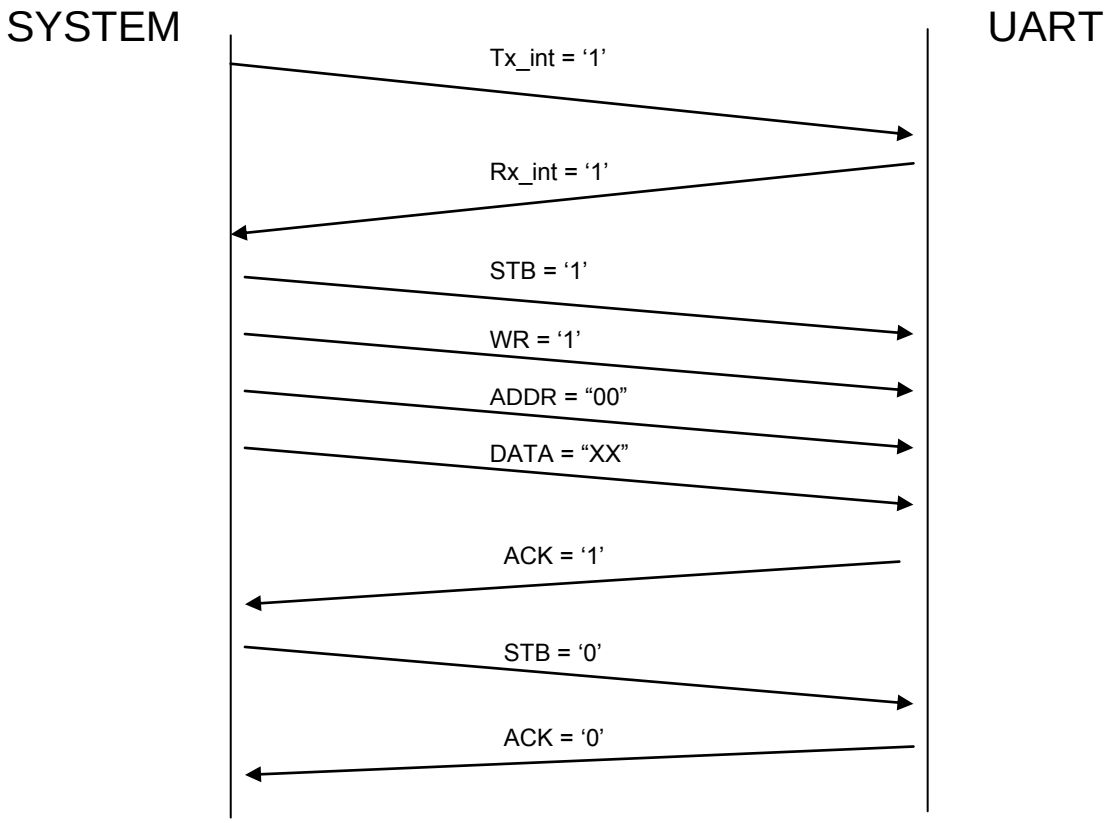


Figure 2.9 WISHBONE Protocol Signal Exchange (6)

The WISHBONE architecture sits between the PC and UART interface as shown below.

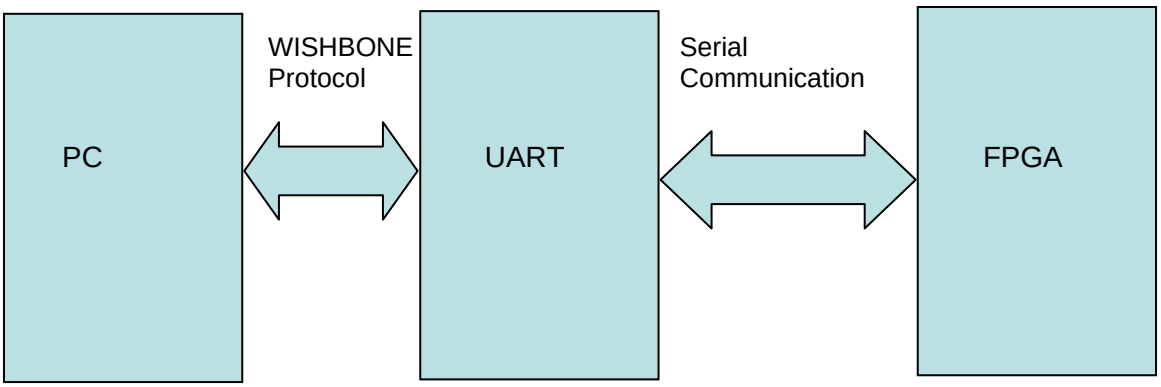


Figure 2.10 System Block Diagram (6)

2.3 UART

UART is the acronym for Universal Asynchronous Receiver/Transmitter. A UART is an integrated circuit that transforms the parallel data stream within the computer to the serial, one-after-the-other data stream used in asynchronous communications. UART's are commonly used in conjunction with other communication standards such as EIA RS-232. This device takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes (7).

The used UART is an Open Core: MiniUART IP Core. It is designed to make an interface between a RS232 line and a Wishbone bus. There are full functionality UARTs available on opencores.org but the full functionality is not required for this project.

The MiniUART is not suited to interface a modem, because there is not handshaking. It integrates two separate clocks, one for wishbone bus, and the other one for bitstream generation. It allows the user to set a desired frequency for the baudrate. Supports: WISHBONE interface in 8-bit data bus, two clock: one for wishbone. For this system, a 9600 Hz clock (baudrate) is generated (8).

CHAPTER III

Audio Processing

3.1 Introduction

Audio processing covers diverse fields, but all of them are involved in presenting sound to human listeners. Some processing methods and application areas of audio processing include: storage, level compression, data compression, transmission, enhancement (equalization, filtering, noise cancellation, echo or reverb removal or addition) (9).

3.2 AC'97 Audio Codec

The LM4550 is an integrate that performs the function of the AC'97 architecture. It is located next to the left pushbutton on the Virtex II Pro board. The LM4550 is a multi-channel audio codec for PC systems with stereo headphone amplifier, and 4 stereo and 4 mono inputs. The codec contains a number of registers which are used to control parameters like output volume, or to select the input source to be sampled by the ADCs (10). These registers allow opening the audio path in the AC97 audio codec. At this point, a study of the LM4550 datasheet is required to figure out what the inputs to the registers should be. Figure 3.11 shows the opened path from the "Mic 1" input to the FPGA and back from the FPGA through the AC97 to the "Line Out" output.

This application writes to the following registers (10).

- Master Volume Register (02h)

  It controls the volume level on the line-level output jacks. Volume is specified with separate 5-bit values for each of the left and right channels. 00000 is the loudest setting and outputs are muted on reset.
- Record Gain register (1Ch)

  It sets the amount by which the incoming audio is amplified before being sampled by the ADC's.
- PCM Out Volume register (18h)

  It sets the gain applied to the DAC outputs before feeding them into the output mixer.
- Mic Volume register (0Eh)

  It controls an additional 20dB boost for the selected microphone input by setting the 20dB bit (bit D6).

The registers settings are loaded using a FIFO memory located in the xupv2pro_wrapper module.

3.2.1 Block Diagram

In the following block diagram, the registers used to open the audio path are shown.

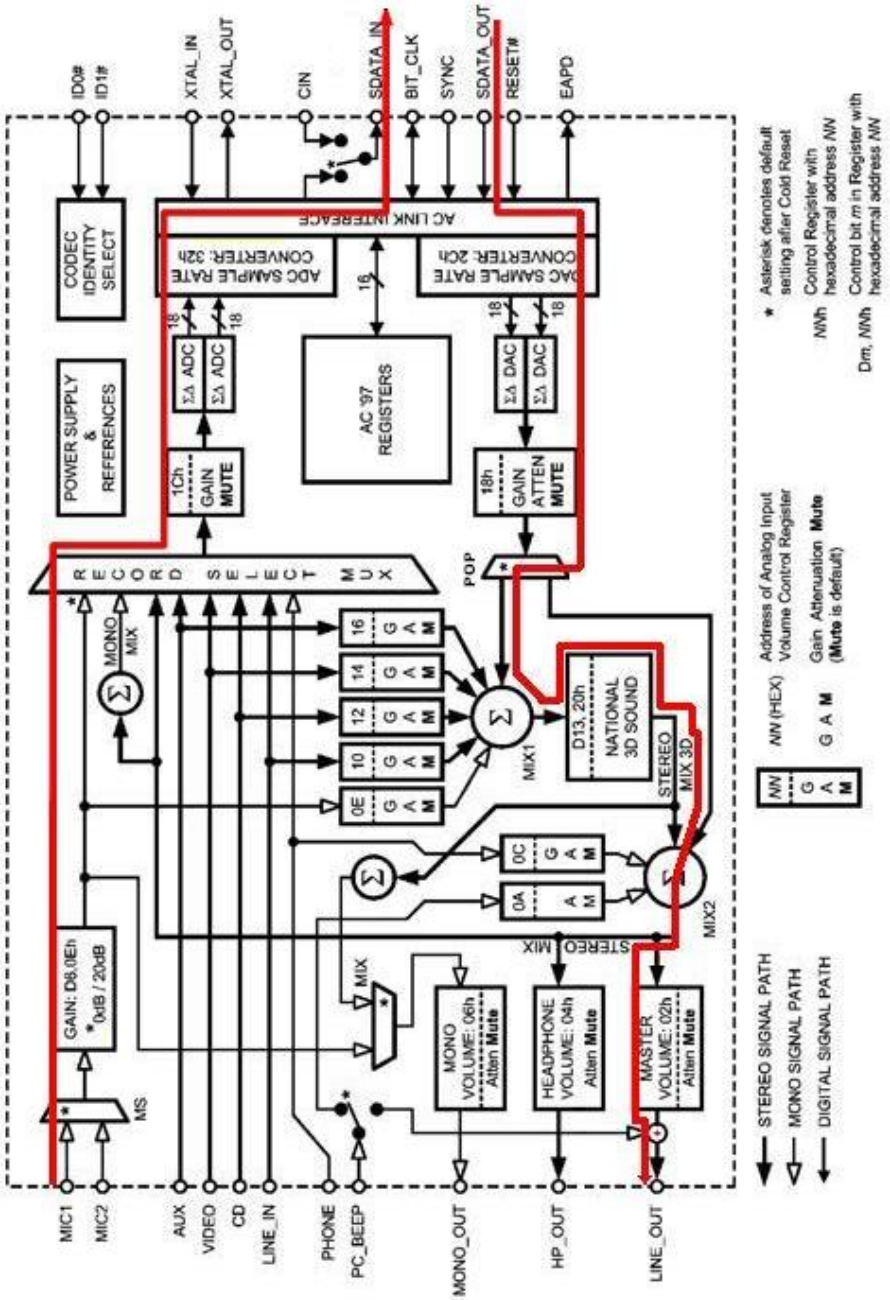


Figure 3.11 Block Diagram LM4550 (10)

CHAPTER IV

Finite State Machine

4.1 Description

A Finite State Machine is a model of computation that consists in a set of states. In the system, a finite state machine is used to perform tasks orderly such us: recording and play back audio signals (11). It has 4 states. Every time AC'97 CNTRL sends the signal CE equals to '1' there is a new sample o data, and the first state is activated. The counter, which provides the addresses to the memory, is activated as well. The memory starts to record, so audio_left_in and right are stored in their respective address. Figure 4.12 shows the finite state machine described in the audio_cw.vhd file.

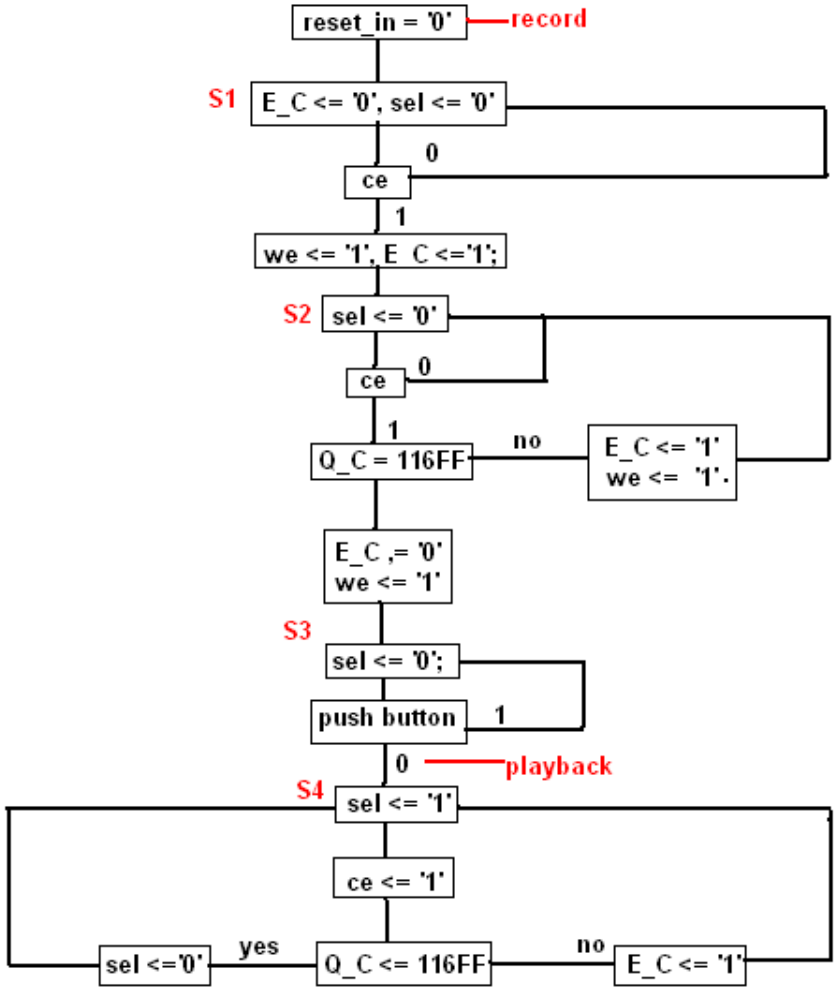


Figure 4.12 audio_cw.vhd state machine

A selector is implemented in order to control when the data stored into the memory have to be assigned to the outputs audio_left and right_out (Figure 4.13). For the second state (sel <= 0) the recorded voice is kept into the memory. Again CE is equal to '1', but the memory can only stored until the address number 116FF, so if the count is equal to that value, the counter stops (E_C <= '0'). The selector keeps its value of '0' (state 3) until the up pushbutton is pushed. When sel <='1' (state 4), the recorded voice is playback over the speaker. Every time the center pushbutton is pressed the state machine goes to State 1.

```
with sel select
     audio_left_out <= audio_left_outq when '1',
                              x"0000" when others;
with sel select
     audio_right_out <= audio_right_outq when '1',
                               x"0000" when others;
```

Figure 4.13 Selector Description

To control recording and playback through HyperTerminal, the pushbuttons signals of the audio_cw are changed to *record_audio*, and *playback_audio*. In *Init_all*, after the UART receives an ack = '1', a comparison has place. Depending on what the UART receives: "72" (record), or "70" (playback); the signals: record, and playback audio are set to '0' or '1'. Right after, they are sent to the *audio_cw* file where the memory, counter, selector and state machine are located. For example, when record_ audio signal is set to '1' the recording process starts in audio_cw file. Since that signal becomes an input of this file, the counter is activated and the memory stars to store data. After every character is pressed, the UART is configured to transmit back the character "a" that can be seen on the screen of HyperTerminal.

```
if UART_ack = '1' then  --Wait for ack
       if  rx_data = x"72" then -- Comparing for rx an "r" (record)
           record_audio <= '1'
           state <= s0;
       elsif
           rx_data = x"70" then -- Comparing for rx an "p" (playback)
           playback_audio <= '1';
           state <= s0;
       else
           state <= sa;
       end if;
             UART_tr_cycle <= '0';
      else
            state <= sd;
end if;
```

Figure 4.14 Reception and comparison of characters of control, *Interface* file

As it was mentioned before, *Launion* (VHDL top level module) makes possible the interaction among inputs and outputs of the *xuv2p_wrapper* and *Interface* modules. These modules become components after they are joined in the top level module. In addition, external inputs (pushbutton's signals) and outputs (led's) are declared in the entity of this file. The files located at the bottom are made up of more components as well.

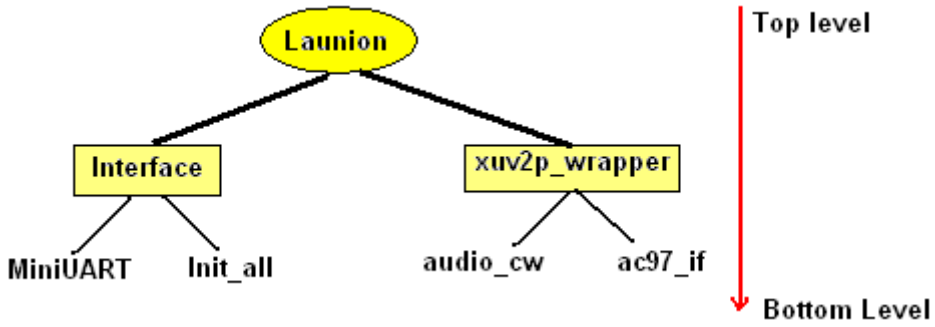


Figure 4.15 Design Structure

.

CHAPTER V

Potential Implementations

As it has been mentioned, the voice recorder and serial interface modules can be used to construct applications that required more complexity by adding some components or lines of code. Some of the potential applications of these two modules are briefly addressed above. These implementations can have educational and research purpose.

5.1 Using xupv2pro_wrapper module

- ✓ In the voice recorder, a filter can be located after the memory. This filter could be a Low Pass, High Pass, or another type of filter. Also there are some cores of filters available on Internet that allow noise reduction and improving voice quality.

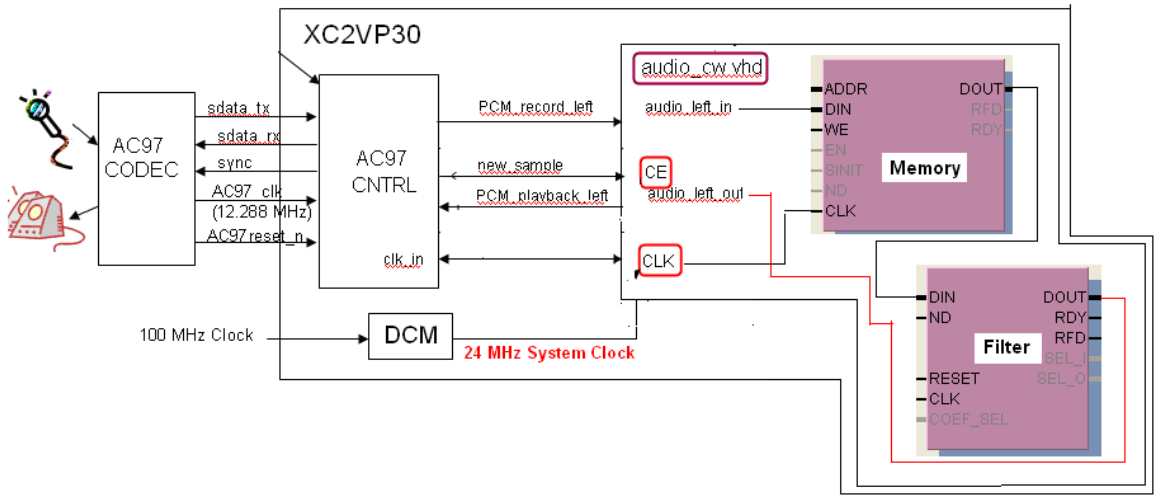


Figure 5.16 Adding a filter

- ✓ Since it is already possible to playback a signal over the speaker, a signal generator can be implemented. In this case, without using the microphone and with a sine wave generator as an input (it can be a core or a new description). The sine wave can be seen on an oscilloscope and playback over the speaker. A square generator can be creating using a counter. So it is not required to use the microphone to provide an input to the module.

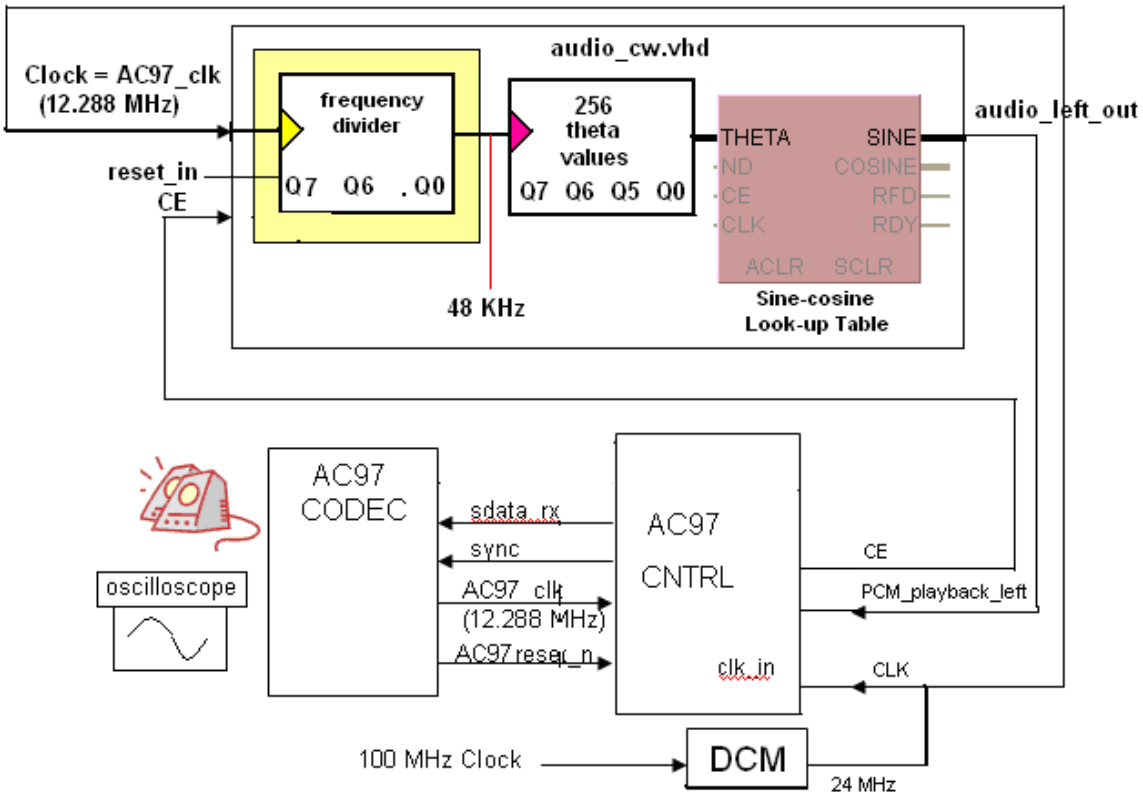


Figure 5.17 Sine generator

✓ Effects like echo and reverberation can be implemented in a different component and connected to the module. Echo effect for example is simply the sum of the original input and a delayed sample. So an algorithm (to be written on VHDL) for this effect would be to delay the sound, repeat the sound and output the modified sound (12).

5.2 Using Interface and wrapper modules

✓ With some changes in the Interface module, transmission from one FPGA and reception in another one can be achieved. Audio signal is acquiring in the first board and then serially transmitted (using a crossover cable) to the other board where voice is stored in a block memory and play it back over the speaker when desired.

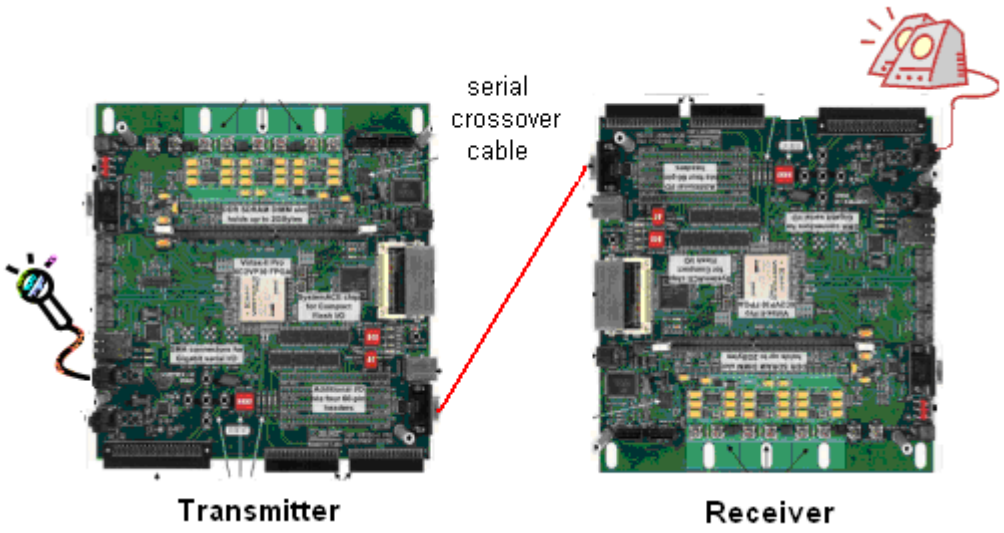


Figure 5.18 Serial Transmission of audio signals

- ✓ Having a transmitter and receiver already set up, serial transmission of coded audio signals is also possible. For example, when an audio signal is acquired is immediately converted from analog to digital, so only by using the unique chip the digital signal was encoded with at the transmitter; it is possible to get the same signal at the receiver.

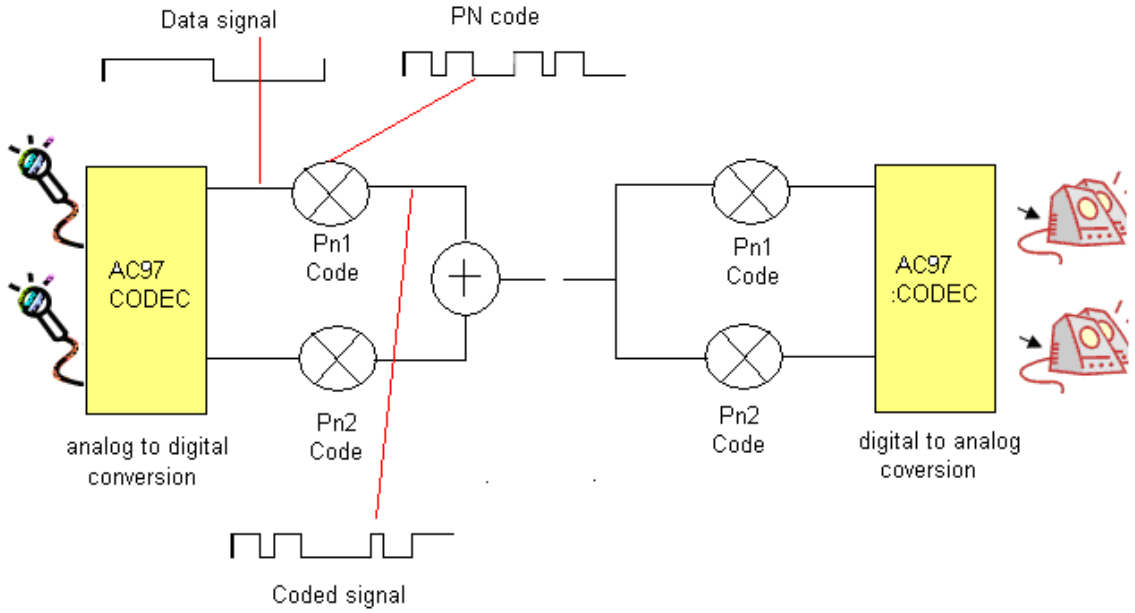


Figure 5.19 Serial Transmission of coded audio signals

CHAPTER VI

Demo

6.1 Description

1. First, it is needed to connect the Digilent XUP Virtex II Pro to a laptop/PC.

2. The serial port (DCE terminal) available on the FPGA board has to be connected to the USB port available on the laptop/PC using a USB-to-Serial converter as shown below.



Figure 6.20 USB-to-Serial Converter

3. A microphone and a speaker are plugged into the audio line and amp of the board.



Figure 6.21 Microphone and Speaker connected

4. After switching on the board, it is needed to program the board using the Xilinx ISE 9.2i tool.

5. The HyperTerminal tool is open, and after giving a name to the connection, the following window will show up. Select the appropriate com port and press OK.



Figure 6.22 Com Port selection

6. Set the baudrate to 9600 baud, since in the present design it is configured for that speed.

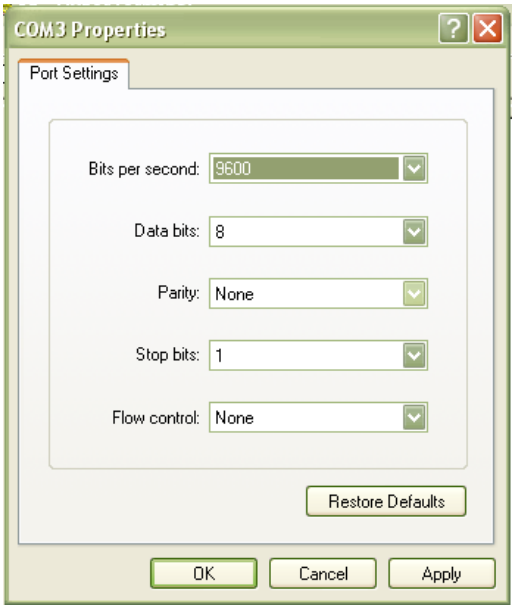


Figure 6.23 Hyper Terminal Settings

7. On the HyperTerminal screen the characters of control have to be typed using the keyboard.

8. Recording is made by typing ‘r’. When desired, in order to playback the audio signal, ‘p’ can be typed. The recording has to be heard over the speaker. These characters have to be typed in the earlier order. “31” character has to be present on the screen; it can be changed in the *Init_all* file.

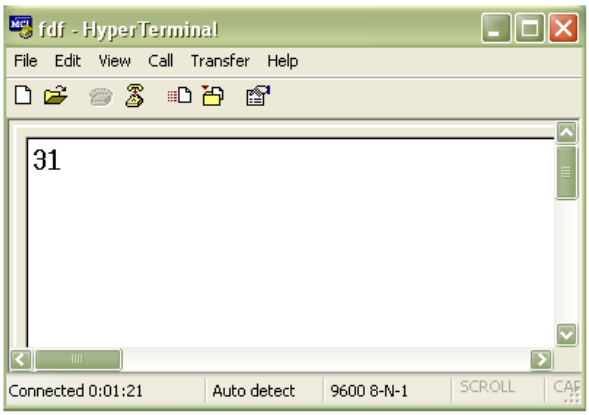


Figure 6.24 Screen of HyperTerminal

5.2 ChipScope shots

Since the present is a written documentation, ChipScope Pro is used as a verifying tool. Two testing words were to get the following shoots.

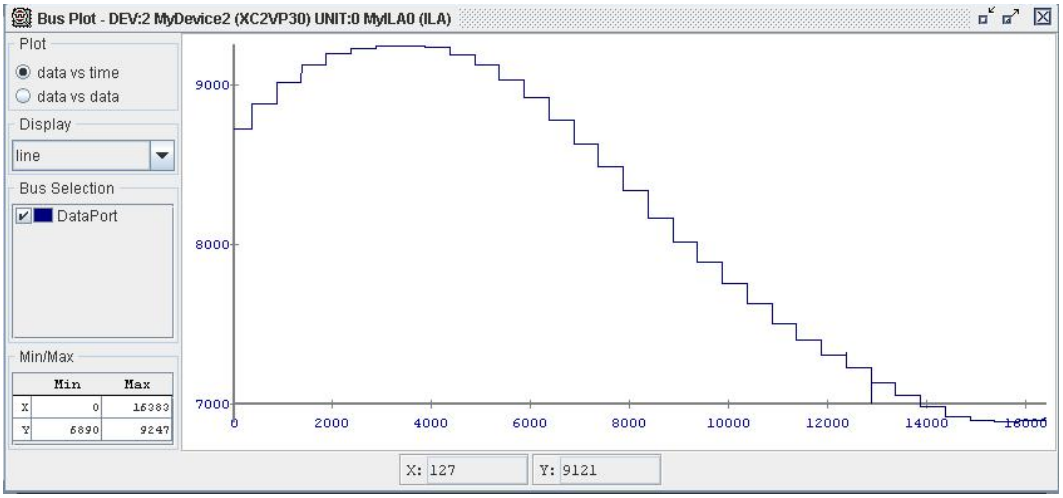


Figure 6.25 Hello

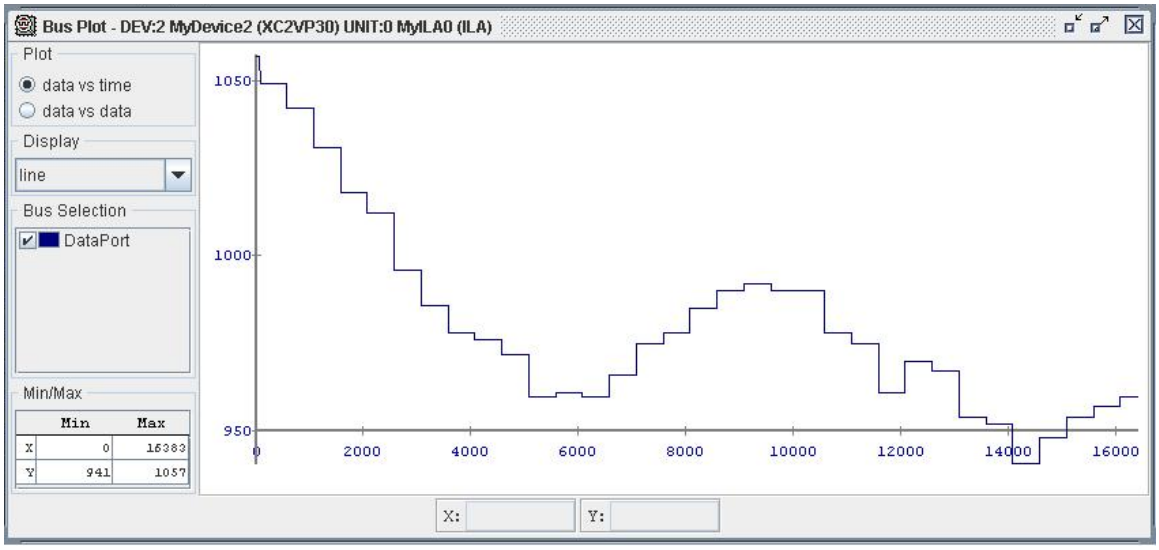


Figure 6.26 Baby

CHAPTER VII

Conclusions and Recommendations

7.1 Conclusions

- Two modules made up of some components were used and joined in a top level module: the xupv2pro_wrapper and Interface. The first one permitted the acquisition of an audio signal and playback it; and the second one allow a serial communication between a computer and a FPGA. Using a hierarchical design made easier building a system like the one that was described, so either modules or components can be used individually.

- The open cores used, MiniUART and xupv2pro_wrapper, made faster this implementation. A complete description of the system would have taken more time to implement and also it would have required more experience in architecture of computers, VHDL programming skills, and hardware description. It was necessary to make some changes to them like adding handshake signals for transmission and reception, implementing components (counter, memory), and constructing states machines.

- Once it is possible to acquire a voice signal using a FPGA, applications and processing methods can be extended too. Different filters can be located before the audio signal is playback over the speaker and get an enhancement of the signal like noise cancellation, and also some effect can be added, echo or reverberation, for example.

- Using a synchronous communication allows sending and getting data correctly at the transmitter and receiver, respectively. Since a UART is used in asynchronous communications, using the wishbone protocol solved the synchronization issue.

- Creating a finite state machine is the best solution to the problem of executing a sequence of commands or actions orderly. Depending on the design, it demands more complexity.

7.2 Recommendations

- In an educational and research environment, the modular design allows creating a common repository of components and modules. Thus, students are able to re-used one or more depending on the application.

- In order to achieve the previous item, it is necessary to describe generic components. For example, using “generic” a VHDL keyword, the number of bits of a counter and the limit of the count, have to be specified at the entity of the component. So, the same component can be use for applications that required a different number of bits, and big modifications aren’t needed.

- For beginners in the field of hardware description, the use of open cores, code from other sources like Internet, books and older projects should be encouraged. According to Dr Steve Suddarth, member of the faculty of UNM, the rule for beginners in the area is ‘copy like a mad, but fairly’. This means students can take projects and implementations from different sources, but it has to be required a contribution, modification or improvement to the design.

- The state machine created for the audio_cw file is a little complex, taking into account the application. It is not the only method to create one; students should be encouraged to use other methods. More experience in the construction of a machine using different techniques, would reduce the number of lines of code and the complexity.

REFERENCES

[1] Andreas Koch, Ulrich Golze (Technical University Braunschweig), FPGA Applications in Education and Research, Germany

[2] What is VHDL?
http://www.doulos.com/knowhow/vhdl_designers_guide/what_is_vhdl/

[3] Mike Wirthlin, HDL Wrapper XUP Virtex-II Pro

[4] Xilinx Logic core Single-Port Block Memory Core v6.2, April 28, 2005
http://www.xilinx.com/support/documentation/ipmeminterfacestorelement_ramrom_singleportblockmem.htm

[5] XUP Virtex II Pro, Datasheet, http://www.xilinx.com/support/documentation/virtex-ii_pro_data_sheets.htm

[6] Vallabh Srikanth Devarapalli (University of New Mexico) Tutorial Serial RS-232, ISE 9.2 and ModelsimXE on the Digilent Spartan-3E board, 2008

[7] Frank Durda, Serial and UART Tutorial, http://www.freebsd.org/doc/en/articles/serial-uart/

[8] Philippe Carton, MiniUART IP Core Specification Rev. 0.1, September 10, 2002

[9]Steven W. Smith, Ph.D, Audio Processing, The Scientist and Engineer's Guide to Digital Signal Processing, http://www.dspguide.com/ch22.htm

[10] National Semiconductor, LM4550: AC '97 Rev 2.1 Multi-Channel Audio Codec with Stereo Headphone Amplifier, Sample Rate Conversion and National 3D Sound, May 2004

[11] Paul E. Black, Finite State Machine,
http://www.nist.gov/dads/HTML/finiteStateMachine.html

[12] Dixie Xue, Enhanced Audio Project, March 22, 2004

ANEX A. Launion, Top level Module in the Hierarchy

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

entity Launion is
  port(
    clk_in1 : in std_logic;
    reset_in : in std_logic;
    -- CODEC signals
    AC97Reset_n : out std_logic;
    AC97Clk   : in  std_logic;-- master clock for design
    Sync      : out std_logic;
    SData_Out : out std_logic;
    SData_In  : in  std_logic;
            rx_in :in std_logic;
    tx_out : out std_logic;
            LED : out std_logic;
            LED2 : out std_logic;
            LED3 : out std_logic;
            LED4 : out std_logic);
end Launion;

architecture Behavioral of Launion is

signal record_audio: std_logic;
signal playback_audio: std_logic;
signal stop_audio: std_logic;


component Interface
      port(
          CLK : in std_logic;
          RST : in std_logic;
                                    rx_in :in std_logic;
                          tx_out : out std_logic;
                                    record_audio: out std_logic;   -- (r) record
                          playback_audio: out std_logic; -- (p) playback
                  stop_audio: out std_logic
                );
end component;

component xupv2pro_wrapper
                        port(
                                    clk_in : in std_logic;
                                    reset_in : in std_logic;
                                    -- CODEC signals
                                    AC97Reset_n : out std_logic;
                                    AC97Clk   : in  std_logic;          -- master clock for design
                                    Sync      : out std_logic;
                                    SData_Out : out std_logic;
                                    SData_In  : in  std_logic;
                                    record_audio: in std_logic;
                                    playback_audio: in std_logic;
                              stop_audio: in std_logic;
                                    LED : out std_logic;
                                    LED2 : out std_logic;
                                    LED3 : out std_logic;
                                    LED4 : out std_logic
```

```
      );
end component;


begin

comp1: interface
  PORT MAP(
                        CLK => clk_in1,
         RST => reset_in,
                              rx_in => rx_in,
                        tx_out => tx_out,
                              record_audio => record_audio,   -- (r) record
                        playback_audio => playback_audio,
                              stop_audio => stop_audio
                              );

comp2: xupv2pro_wrapper
  PORT MAP(
        LED => LED,
                              LED2 => LED2,
                              LED3 => LED3,
                              LED4 => LED4,
                        clk_in => clk_in1,
                              reset_in => reset_in,
                              -- CODEC signals
                              AC97Reset_n => AC97Reset_n,
                              AC97Clk => AC97Clk,           -- master clock for design
                              Sync => Sync,
                              SData_Out => SData_Out,
                              SData_In  => SData_In,
                              record_audio => record_audio,
                              playback_audio => playback_audio,
                              stop_audio => stop_audio
                  );

end Behavioral;
```

ANEX B. Interface

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

entity Interface is
    Port (
        CLK : in std_logic;
        RST : in std_logic;
                        rx_in :in std_logic;
                tx_out : out std_logic;
                        record_audio: out std_logic;
                        playback_audio: out std_logic;
                        stop_audio: out std_logic
                );
end Interface;

architecture Structural of Interface is

component Init_all is
    Port ( rst : in std_logic;
        ck : in std_logic;
                                --Signals for the UART
                UART_addr_bus: out std_logic_vector(1 downto 0);
                UART_data_bus: out std_logic_vector(7 downto 0);
                UART_data_bus_rx: in std_logic_vector(7 downto 0);
                UART_wr_rd: out std_logic;
                UART_tr_cycle: out std_logic;
                UART_tx_int: in std_logic;
                UART_rx_int: in std_logic;
                UART_ack: in std_logic;
                record_audio: out std_logic;   -- (r) record
                playback_audio: out std_logic;  -- (p) playback
                stop_audio: out std_logic
            );
end component;

component UART is
  port (
-- Wishbone signals
    WB_CLK_I : in  std_logic;  -- clock
    WB_RST_I : in  std_logic;  -- Reset input
    WB_ADR_I : in  std_logic_vector(1 downto 0); -- Adress bus
    WB_DAT_I : in  std_logic_vector(7 downto 0); -- DataIn Bus
    WB_DAT_O : out std_logic_vector(7 downto 0); -- DataOut Bus
    WB_WE_I  : in  std_logic;  -- Write Enable
    WB_STB_I : in  std_logic;  -- Strobe
    WB_ACK_O : out std_logic;        -- Acknowledge
-- process signals
    IntTx_O  : out std_logic;  -- Transmit interrupt: indicate waiting for Byte
    IntRx_O  : out std_logic;  -- Receive interrupt: indicate Byte received
    BR_Clk_I : in  std_logic;  -- Clock used for Transmit/Receive
    TxD_PAD_O: out std_logic;  -- Tx RS232 Line
    RxD_PAD_I: in  std_logic);  -- Rx RS232 Line
end component;

Signal S_UART_addr_bus: std_logic_vector(1 downto 0);
Signal S_UART_data_bus: std_logic_vector(7 downto 0);
Signal S_UART_wr_rd: std_logic;
Signal S_UART_tr_cycle: std_logic;
```

```
Signal S_UART_tx_int: std_logic;
Signal S_UART_rx_int: std_logic;
Signal S_UART_ack: std_logic;
Signal Bus_data_out: std_logic_vector(7 downto 0);
Signal int_rx: std_logic;
Signal rx_out: std_logic;
Signal RST_H: std_logic;

begin

RST_H <= not (RST); -- FOR COMPATIBILITY WITH XUP BOARD!!!!!

 Mach: Init_all
   port map( rst => RST_H,
           ck => CLK,
                        -- UART Signals
                        UART_addr_bus => S_UART_addr_bus,
                        UART_data_bus => S_UART_data_bus,
                                UART_data_bus_rx => Bus_data_out,
                        UART_wr_rd    => S_UART_wr_rd,
                        UART_tr_cycle => S_UART_tr_cycle,
                        UART_tx_int => S_UART_tx_int,
                                UART_rx_int => int_rx,
                        UART_ack => S_UART_ack,
                                record_audio => record_audio,   -- (r) record
                        playback_audio => playback_audio,
                                stop_audio => stop_audio
                       );

mini_u: UART
  port map(
                     -- Wishbone signals
              WB_CLK_I => CLK,
              WB_RST_I => RST_H,
              WB_ADR_I => S_UART_addr_bus,
              WB_DAT_I => S_UART_data_bus,
              WB_DAT_O => Bus_data_out, --taked out
              WB_WE_I => S_UART_wr_rd,
              WB_STB_I => S_UART_tr_cycle,
              WB_ACK_O => S_UART_ack, --taked out
                        -- process signals
              IntTx_O => S_UART_tx_int,
              IntRx_O => int_rx,
              BR_Clk_I => CLK,
              TxD_PAD_O => tx_out,
              RxD_PAD_I =>  rx_in
                      );

end Structural;
```

ANEX C. MiniUART

```
-- File        : MiniUart.vhd
-- Author      : Philippe CARTON
--             (philippe.carton2@libertysurf.fr)
-- Organization:
-- Created     : 15/12/2001
-- Last update : 8/1/2003
-- Platform    : Foundation 3.1i
-- Simulators  : ModelSim 5.5b
-- Synthesizers: Xilinx Synthesis
-- Targets     : Xilinx Spartan
-- Dependency  : IEEE std_logic_1164, Rxunit.vhd, Txunit.vhd, utils.vhd
-------------------------------------------------------------------------------
-- Description: Uart (Universal Asynchronous Receiver Transmitter) for SoC.
--    Wishbone compatable.
-------------------------------------------------------------------------------
-- Copyright (c) notice
--    This core adheres to the GNU public license
--
-------------------------------------------------------------------------------
library ieee;
   use ieee.std_logic_1164.all;

entity UART is
  generic(BRDIVISOR: INTEGER range 0 to 65535 := 2604);
                                                                        -- 2604
for 100MHz @9600
                                                                        -- 521
Baud rate divisor  for 40MHz
  port (                                                        -- 325 for 25 MHZ
                                                                        -- 651
for 50 MHZ
                                                                        -- 430
for 33 MHz @ 19200
                                                                        -- 65 for
5 MHZ
                                                                        -- 10 for
781250Hz
-- Wishbone signals
     WB_CLK_I : in  std_logic;  -- clock
     WB_RST_I : in  std_logic;  -- Reset input
     WB_ADR_I : in  std_logic_vector(1 downto 0); -- Adress bus
     WB_DAT_I : in  std_logic_vector(7 downto 0); -- DataIn Bus
     WB_DAT_O : out std_logic_vector(7 downto 0); -- DataOut Bus
     WB_WE_I  : in  std_logic;  -- Write Enable
     WB_STB_I : in  std_logic;  -- Strobe
     WB_ACK_O : out std_logic;        -- Acknowledge
-- process signals
     IntTx_O  : out std_logic;  -- Transmit interrupt: indicate waiting for Byte
     IntRx_O  : out std_logic;  -- Receive interrupt: indicate Byte received
     BR_Clk_I : in  std_logic;  -- Clock used for Transmit/Receive
     TxD_PAD_O: out std_logic;  -- Tx RS232 Line
     RxD_PAD_I: in  std_logic);  -- Rx RS232 Line
end UART;

-- Architecture for UART for synthesis
architecture Behaviour of UART is

  component Counter
  generic(COUNT: INTEGER range 0 to 65535); -- Count revolution
```

```
port (
   Clk      : in  std_logic;  -- Clock
   Reset    : in  std_logic;  -- Reset input
   CE       : in  std_logic;  -- Chip Enable
   O        : out std_logic); -- Output
end component;

component RxUnit
port (
   Clk    : in  std_logic;  -- system clock signal
   Reset  : in  std_logic;  -- Reset input
   Enable : in  std_logic;  -- Enable input
   ReadA  : in  Std_logic;  -- Async Read Received Byte
   RxD    : in  std_logic;  -- RS-232 data input
   RxAv   : out std_logic;  -- Byte available
   DataO  : out std_logic_vector(7 downto 0)); -- Byte received
end component;

component TxUnit
port (
   Clk    : in  std_logic;  -- Clock signal
   Reset  : in  std_logic;  -- Reset input
   Enable : in  std_logic;  -- Enable input
   LoadA  : in  std_logic;  -- Asynchronous Load
   TxD    : out std_logic;  -- RS-232 data output
   Busy   : out std_logic;  -- Tx Busy
   DataI  : in  std_logic_vector(7 downto 0)); -- Byte to transmit
end component;

signal RxData : std_logic_vector(7 downto 0); -- Last Byte received
signal TxData : std_logic_vector(7 downto 0); -- Last bytes transmitted
signal SReg   : std_logic_vector(7 downto 0); -- Status register
signal EnabRx : std_logic;  -- Enable RX unit
signal EnabTx : std_logic;  -- Enable TX unit
signal RxAv   : std_logic;  -- Data Received
signal TxBusy : std_logic;  -- Transmiter Busy
signal ReadA  : std_logic;  -- Async Read receive buffer
signal LoadA  : std_logic;  -- Async Load transmit buffer
signal Sig0   : std_logic;  -- gnd signal
signal Sig1   : std_logic;  -- vcc signal

begin
sig0 <= '0';
sig1 <= '1';
Uart_Rxrate : Counter -- Baud Rate adjust
   generic map (COUNT => BRDIVISOR)
   port map (BR_CLK_I, sig0, sig1, EnabRx);
Uart_Txrate : Counter -- 4 Divider for Tx
   generic map (COUNT => 4)
   port map (BR_CLK_I, Sig0, EnabRx, EnabTx);
Uart_TxUnit : TxUnit port map (BR_CLK_I, WB_RST_I, EnabTX, LoadA, TxD_PAD_O, TxBusy, TxData);
Uart_RxUnit : RxUnit port map (BR_CLK_I, WB_RST_I, EnabRX, ReadA, RxD_PAD_I, RxAv, RxData);
IntTx_O <= not TxBusy;
IntRx_O <= RxAv;
SReg(0) <= not TxBusy;
SReg(1) <= RxAv;
SReg(7 downto 2) <= "000000";

-- Implements WishBone data exchange.
-- Clocked on rising edge. Synchronous Reset RST_I
WBctrl : process(WB_CLK_I, WB_RST_I, WB_STB_I, WB_WE_I, WB_ADR_I)
variable StatM : std_logic_vector(4 downto 0);
```

```
  begin
    if Rising_Edge(WB_CLK_I) then
      if (WB_RST_I = '1') then
            ReadA <= '0';
            LoadA <= '0';
          else
            if (WB_STB_I = '1' and WB_WE_I = '1' and WB_ADR_I = "00") then -- Write Byte to Tx
          TxData <= WB_DAT_I;
              LoadA <= '1';   -- Load signal
        else LoadA <= '0';
        end if;
            if (WB_STB_I = '1' and WB_WE_I = '0' and WB_ADR_I = "00") then -- Read Byte from Rx
              ReadA <= '1';   -- Read signal
        else ReadA <= '0';
        end if;
      end if;
    end if;
  end process;
  WB_ACK_O <= WB_STB_I;
  WB_DAT_O <=
    RxData when WB_ADR_I = "00" else  -- Read Byte from Rx
    SReg when WB_ADR_I = "01" else    -- Read Status Reg
    "00000000";
end Behaviour;
```

ANEX D. Init_all

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

entity Init_all is
    Port ( rst : in std_logic;
           ck : in std_logic;
                    --Signals for the UART
                    UART_addr_bus: out std_logic_vector(1 downto 0);
                    UART_data_bus: out std_logic_vector(7 downto 0);
              UART_data_bus_rx: in std_logic_vector(7 downto 0);
                    UART_wr_rd: out std_logic;
                    UART_tr_cycle: out std_logic;
                    UART_tx_int: in std_logic;
                    UART_rx_int: in std_logic;
                    UART_ack: in std_logic;
                    record_audio: out std_logic;   -- (r) record
                    playback_audio: out std_logic; -- (p) playback
                    stop_audio: out std_logic
                );
end Init_all;

architecture Behavioral of Init_all is

              type state_type is (sa, sb, sc, sd,
s0,s1,s2,s3,s3delay,s3nitx,s4,s5,s5delay,s5nitx,s8,s8LFTx,s8LFAk,s8delay,s8CRTx,s8CRAk,s8delay2,s9);
              signal state: state_type ;
              --Signals for Nibble Tx
              Signal Nib_Tx: std_logic_vector(7 downto 0);
              signal rx_data : std_logic_vector(7 downto 0);

begin

  process (ck,rst)

  variable ASCIItot: std_logic_vector(7 downto 0);

  begin
    if (rst ='1') then
      --Initialization of state
      state <= sa;
              -- Initialization of UART
              UART_addr_bus <= "00";
              UART_data_bus <= "00000000";
              UART_wr_rd <= '1';
              UART_tr_cycle <='0';
              -- Initalization of Nibble
              Nib_Tx <= "00000000";
              record_audio <= '0';
              playback_audio <= '0';

    elsif (ck='1' and ck'Event) then
      case state is

               when sa =>
                                 if UART_rx_int = '1' then
                                          state <= sb;
                                 else
                                          state <= sa;          -- Waiting until the UART is ready to RX
```

```
                        end if;

            when sb => state <= sc;   -- Delay to allow pin int up.

            when sc => if UART_rx_int = '1' then          -- Rx
                                                    state <= sd;
                                                    UART_addr_bus <= "00";
                                                    UART_wr_rd <= '0';
                                                    rx_data  <=  UART_data_bus_rx;
                                                    UART_tr_cycle <= '1';
            else
                                                    state <= sc;
            end if;

      when sd => if UART_ack = '1' then  --Wait for ack
                        if  rx_data = x"72" then -- Comparing for rx an "r" (record)
                                                        record_audio <= '1';
                                                                stop_audio <= '1';
                                                        state <= s0;
                                                    elsif
                                                            rx_data = x"70" then --
Comparing for rx an "p" (playback)
                                                        playback_audio <= '1';
                                                                stop_audio <= '1';
                                                        state <= s0;
                                                    elsif
                                                            rx_data = x"73" then --
Comparing for rx an "s" (stop)
                                                        playback_audio <= '0';
                                                        record_audio <= '0';
                                                                stop_audio <= '0';
                                                                state <= s0;
                                                          else
                                                        state <= sa;
                                                    end if;
                                                            UART_tr_cycle <= '0';
             else
                                                    state <= sd;
             end if;
     --parte de grabacion
                    when s0 => state <= s1;     -- Delay to allow pin int UART down

            when s1 => -- Starting the machine. Checking for End of Conversion.

                                        state <= s2;
                                        Nib_Tx <= x"31"; -- Adquiring new data 31      8 bits
                                    -- Keeping UART IDLE
                                    UART_addr_bus <= "00";
                                    UART_data_bus <= "00000000";
                                    UART_wr_rd <= '1';
                                    UART_tr_cycle <='0';

            when s2 => state <= s3;  --Tx first nibble
                                  UART_addr_bus <= "00";
                                    ASCIItot:= Nib_Tx(7 downto 4) + X"30";  --lee solo 8 bits
                                    UART_data_bus <= ASCIItot(7 downto 0);-- Tx first nibble
                                    UART_wr_rd <= '1';
                                    UART_tr_cycle <='1';  -- For WishBone protocol

            when s3 => if UART_ack = '1' then  -- Waiting for UART ack
                                        state <= s3delay;
                                  UART_addr_bus <= "00";
```

```
                                UART_data_bus <= ASCIItot(7 downto 0); -- Tx first nibble
                                UART_wr_rd <= '1';
                                UART_tr_cycle <='0'; -- For WishBone protocol
                               else
                                state <= s3; -- Waiting until for UART confirmation of WishBone
          end if;

when s3delay =>  --Delay
                state <= s3NiTx; --Delay to allow pin int UART down.

        when s3NiTx => if UART_tx_int = '1' then
                                state <= s4;
                               else
                                state <= s3NiTx;          -- Waiting until the UART is ready to TX
          end if;

        when s4 => state <= s5;  --Tx second nibble
                               UART_addr_bus <= "00";
                               ASCIItot:= Nib_Tx(3 downto 0) + X"30"; --lee los otro 8 bits
                               UART_data_bus <= ASCIItot(7 downto 0);-- Tx first nibble
                               UART_wr_rd <= '1';
                               UART_tr_cycle <='1';  -- For WishBone protocol

        when s5 => if UART_ack = '1' then  -- Waiting for UART ack
                                state <= s5delay;
                                UART_addr_bus <= "00";
                                UART_data_bus <= ASCIItot(7 downto 0); -- Tx first nibble
                                UART_wr_rd <= '1';
                                UART_tr_cycle <='0'; -- For WishBone protocol
                               else
                                state <= s5; -- Waiting until for UART confirmation of WishBone
          end if;

  when s5delay =>  --Delay
                state <= s5NiTx; --Delay to allow pin int UART down.

        when s5NiTx => if UART_tx_int = '1' then
                                state <= s8;
                               else
                                state <= s5NiTx;          -- Waiting until the UART is ready to TX
          end if;

        when s8 => state <= s8LFTx;   -- Delay to allow pin int UART down.

        when s8LFTx => if UART_tx_int = '1' then  -- Tx a LF
                        state <= s8LFAK;
                                        UART_addr_bus <= "00";
                                        UART_wr_rd <= '1';
                                        UART_data_bus <= X"0A"; --0x0A Transmitting a LF
                                        UART_tr_cycle <= '1';
            else
                                    state <= s8LFTx;
            end if;

  when s8LFAK => if UART_ack = '1' then  --Wait for ack
                        state <= s8delay;
                                        UART_addr_bus <= "00";
                                        UART_wr_rd <= '1';
                                        UART_data_bus <= X"0A";
                                        UART_tr_cycle <= '0';
            else
                                    state <= s8LFAK;
```

```
                end if;

            when s8delay => state <= s8CRTx;            -- Delay to allow pin int UART down

            when s8CRTx => if UART_tx_int = '1' then  -- Tx a CR
                        state <= s8CRAK;
                                        UART_addr_bus <= "00";
                                        UART_wr_rd <= '1';
                                        UART_data_bus <= X"0D"; --0x0D Transmitting a CR
                                        UART_tr_cycle <= '1';
                else
                                    state <= s8CRTx;
                end if;

        when s8CRAK => if UART_ack = '1' then  --Wait for ack
                        state <= s8delay2;
                                        UART_addr_bus <= "00";
                                        UART_wr_rd <= '1';
                                        UART_data_bus <= X"0D";
                                        UART_tr_cycle <= '0';
                else
                                    state <= s8CRAK;
                end if;

            when s8delay2 =>   -- Delay to allow pin int UART down
                                                state <= s9;

        when s9 => if UART_tx_int = '1' then
                      state <= sa;
                   else
                                  state <= s9;
                               end if;

---

     end case;
   end if;

  end process;
end Behavioral;
```

ANEX E. xupv2pro_Wrapper

```
-- Structure:
--              ac97_core
--                ac97_timing
--              srl_fifo
--
----------------------------------------------------------------------------
-- Author:          Mike Wirthlin
-- Revision:        $$
-- Date:            $$
--library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity xupv2pro_wrapper is
  port (
    clk_in : in std_logic;
    reset_in : in std_logic;
    -- CODEC signals
    AC97Reset_n : out std_logic;
    AC97Clk   : in  std_logic;          -- master clock for design
    Sync      : out std_logic;
    SData_Out : out std_logic;
    SData_In  : in  std_logic;
              record_audio: in std_logic;
    playback_audio: in std_logic;
              stop_audio: in std_logic;
              LED : out std_logic;
              LED2 : out std_logic;
              LED3 : out std_logic;
              LED4: out std_logic
    );

end xupv2pro_wrapper;

architecture imp of xupv2pro_wrapper is


 COMPONENT ClockGen
    PORT
              (
                CLKIN_IN        : IN std_logic;
                        RST_IN          : IN std_logic;
                        CLKFX_OUT       : OUT std_logic;
                        CLK0_OUT1           : OUT std_logic;
                        CLK0_OUT        : OUT std_logic;
                        LOCKED_OUT      : OUT std_logic
                         );
              END COMPONENT;

              signal ClkDiv        : std_logic;
    signal dcm_lock      : std_logic;


  COMPONENT audio_cw
            PORT(
                                clk: in std_logic;
                                reset_in: in std_logic;
                                ce                      : in std_logic;
                                audio_left_out: out std_logic_vector(15 downto 0);
```

```
                                audio_left_in: in std_logic_vector(15 downto 0);
                                stop_audio: in std_logic;
        record_audio: in std_logic;
                                playback_audio: in std_logic;
                                LED : out std_logic;
                                LED2 : out std_logic;
                                LED3 : out std_logic;
                           LED4 : out std_logic
                        );
            END COMPONENT;


    COMPONENT ac97_if is
    port (
            clk_in : in std_logic;
            --Reset : in std_logic;
            -- All signals synchronous to clk_in
            PCM_Playback_Left: in std_logic_vector(15 downto 0);
            PCM_Playback_Right: in std_logic_vector(15 downto 0);

            PCM_Record_Left: out std_logic_vector(15 downto 0);
            PCM_Record_Right: out std_logic_vector(15 downto 0);
                                        PCM_Record_Valid : out std_logic;

            AC97Reset_n : out std_logic;        -- AC97Clk

            -- CODEC signals (synchronized to AC97Clk)
            AC97Clk   : in  std_logic;
            Sync      : out std_logic;
            SData_Out : out std_logic;
            SData_In  : in  std_logic
        );
    end component ac97_if;

        signal new_sample : std_logic;
        signal audio_left_in : std_logic_Vector(15 downto 0) := "0000000000000000";
                                signal audio_right_in : std_logic_Vector(15 downto 0) := "0000000000000000";
        signal audio_left_out : std_logic_Vector(15 downto 0) := "0000000000000000";
                                signal audio_right_out : std_logic_Vector(15 downto 0) := "0000000000000000";
        signal audio_left_wire : std_logic_Vector(15 downto 0) := "0000000000000000";
                                signal audio_right_wire : std_logic_Vector(15 downto 0) := "0000000000000000";

        signal debug_i : std_logic_vector(3 downto 0);
        signal reset_i : std_logic;
                                signal AC97Reset_n_i: std_logic;
                                signal enable_i : std_logic;

        signal sync_i,sdata_out_i : std_logic;

begin

    Inst_ClockGen: ClockGen PORT MAP(
                        CLKIN_IN        => clk_in,
                        RST_IN          => not reset_in,
                        CLKFX_OUT       => ClkDiv,
                        CLK0_OUT1       => open,
                        CLK0_OUT        => open,
                        LOCKED_OUT      => dcm_lock
            );

    AC97Reset_n <= AC97Reset_n_i;
    Sync <= sync_i;
```

```
SData_Out <= SData_Out_i;
enable_i <= dcm_lock;

ac97_if_I : ac97_if
port map (
   clk_in => ClkDiv,

   PCM_Playback_Left   => audio_left_out,
   PCM_Playback_Right  => audio_right_out,

   PCM_Record_Left     => audio_left_in,
   PCM_Record_Right    => audio_right_in,
                  PCM_Record_Valid    => new_sample,

   AC97Reset_n         => AC97Reset_n_i,
   AC97Clk             => AC97Clk,
   Sync                => sync_i,
   SData_Out           => SData_Out_i,
   SData_In            => SData_In
 );

Inst_audio_cw: audio_cw
PORT MAP(
                  clk               => ClkDiv,
                  reset_in          => reset_in,
                  ce                                                    => new_sample,
                  audio_left_out    => audio_left_out,
                  audio_left_in     => audio_left_in,
   record_audio      => record_audio,
                  playback_audio    => playback_audio,
                  stop_audio        => stop_audio,
                  LED               => LED,
                  LED2              => LED2,
                  LED3              => LED3,
                  LED4              => LED4
                  );

end architecture imp;
```

ANEX F. audio_cw

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

entity audio_cw is
        port ( clk: in std_logic;
             reset_in: in std_logic;   -- enter push button
                        ce : in std_logic;
                        record_audio: in std_logic;   -- '1' record
                        playback_audio: in std_logic; -- '1' playback
                        stop_audio: in std_logic;
                        audio_left_in: in std_logic_vector(15 downto 0);
      audio_left_out: out std_logic_vector(15 downto 0);
      LED : out std_logic;
      LED2 : out std_logic;
      LED3 : out std_logic;
      LED4 : out std_logic
               );
end audio_cw;

architecture Behavioral of audio_cw is
 signal audio_left_outq, audio_right_outq: std_logic_vector(15 downto 0);
 signal Q_C: std_logic_vector(19 downto 0);
 signal we, E_C, rst_C, sel: std_logic;
 signal start: std_logic;
 type state is (S0,S1, S2, S3, S4);
 signal y: state;

        component cont
                port (
                    clk, start1, E, rst: in std_logic;
                            Q : out std_logic_vector (19 downto 0));
        end component;

        component mem_reco
                port (
                addr: IN std_logic_VECTOR(16 downto 0);
                clk: IN std_logic;
                din: IN std_logic_VECTOR(15 downto 0);
                dout: OUT std_logic_VECTOR(15 downto 0);
                we: IN std_logic);
        END component;

BEGIN

LED4 <= stop_audio;

with sel select
    audio_left_out <= audio_left_outq when '1',
                      x"0000" when others;

 transitions: process (clk,reset_in) -- states change
 begin
        if reset_in = '0' then
                y <= S0;
                start <= '0';
                LED <= '0';
                LED2 <= '1';
                LED3 <= '1';
```

```
        elsif (clk'event and clk = '1') then
                LED <= '1';

                case y is
                        when S0 =>
                            if record_audio='1' then
                                                LED2 <= '0';
                                                LED3 <= '1';
                                    y <= S1;
                                        start <= '1';
                                 else
          y <= S0;
        end if;

                        when S1 =>
                                if ce = '1' then y <= S2;
                                else y <= S1; end if;

                        when S2 =>
                                if ce = '1' then
                                        if Q_C = x"116FF" then y <= S3;
                                          else y <= S2; end if;
                                else
                                          y <= S2;
                                end if;

                        when S3 =>
                          if playback_audio = '1' then
                                                LED2 <= '1';
                                                LED3 <= '0';
                                        y <= S4;

                                else
                                        y <= S3;
                                end if;

                        when S4 =>

                                end case;
        end if;

end process;

Outputs: process (y)   -- signals change
        begin
                rst_C <= '0'; sel <= '0'; E_C <= '0'; we <= '0';

                case y is

                        when S0 =>

                        when S1 =>
                                        rst_C <= '1'; E_C <= '1';      -- C <= 0
                                        sel <= '0';
                                if ce = '1' then
                                        we <= '1';
                                        rst_C <= '0';
                                        E_C <= '1'; -- C <= C + 1
                                end if;

                        when S2 =>
```

```
                                        sel <= '0';
                                        if ce = '1' then
                                                if Q_C = x"116FF" then
                                                        rst_C <= '1'; E_C <= '1';   -- C <= 0
                                                        we <= '1';
                                                else
                                                        E_C <= '1';    -- C <= C + 1
                                                        we <= '1';
                                                end if;
                                        end if;

                                when S3 =>
                                        sel <= '0';

                                when S4 =>
                                        sel <= '1';
                                        if ce = '1' then
                                                if Q_C = x"116FF" then
                                                        sel <= '0';
                                                else
                                                        E_C <= '1'; -- C <= C + 1
                                                end if;
                                        end if;

                        end case;
                end process;

m1: mem_reco port map (
    clk => clk,
                addr => Q_C(16 downto 0),
                din => audio_left_in,
                we => we,
                dout => audio_left_outq);

c1: cont port map (
    clk => clk,
                start1 => start,
                E => E_C,
                rst => '0',
                Q => Q_C);

end Behavioral;
```