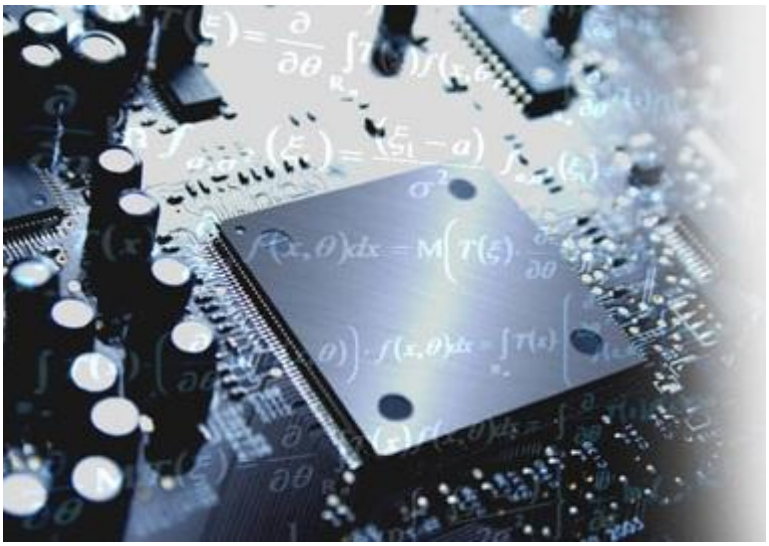
CHALMERS



Implementing an AC97Audio Controller IP

Master of Science Thesis in Integrated Electronic System Design

JOSÉ ROBERTO SÁNCHEZ MAYEN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

Implementing an AC97 Audio Controller IP

JOSÉ ROBERTO SANCHEZ MAYEN

Examiner: LARS SVENSSON
Supervisor: JIRI GAISLER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The cover picture shows a field programmable gate array attached to an electronic board.
Taken from: http://www.seasolve.com/wireless-ip-cores.html

Department of Computer Science and Engineering
Göteborg, Sweden June 2011

Implementing an AC97 Audio Controller IP

JOSÉ ROBERTO SÁNCHEZ MAYEN
Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering

Abstract

The purpose of this Master's project is to develop an AC97 IP core for Aeroflex Gaisler AB. The IP core's function is to control the playback of digital audio in an embedded system and it will be introduced in a LEON3-based system, which is programmed in a Virtex 5 FPGA in the Xilinx ML505 prototype board. The core is implemented in VHDL using the Two Process Methodology. The design of the AC97 core was divided in two stages. The first stage of the design aimed to achieve communication among the LEON3-based system, the AC97 core, and the AC97 CODEC (controlled by the AC97 IP core); in the second stage of the design, a DMA engine was introduced to transfer PCM data from memory to the CODEC. Through the medium of assessing the thesis objectives and the obtained outcomes, the design of the AC97 IP core was partially verified. Simulation in ModelSim was carried out for both stages. The second stage of the design still has to be thoroughly verified in hardware by playing real audio on the Xilinx prototype board.

Keywords: AC97, IP core, CODEC, LEON3, FPGA, VHDL, Two Process Methodology, DMA, PCM, ModelSim.

Acknowledgements

I would like to express my gratitude to my examiner Lars Svensson for accepting this responsibility, for taking the time to have meetings at the company, for following up my progress, for his detailed feedback on the report, and for helping me in any possible way he could.

Correspondingly, I am very grateful to my supervisor Jiri Gaisler for giving me the chance to work at Aeroflex Gaisler AB. I am deeply indebted with Jan Andersson in particular from Gaisler Research, because all along the development of my thesis he advised and aided me invaluably. Likewise, I found Magnus Hjorth's comments and advices very useful for the project. I appreciate the kindness of Gaisler's staff in general for their friendly behavior towards me.

Likewise, I would like to thank The Swedish Foundation for International Cooperation in Research and Higher Education for backing me up financially amid the two years the Master's degree lasted.

I am very thankful to Olivia Cintas Sánchez for giving me all her support and knowledge in the process of writing and presenting a Master's thesis.

Last but not least, I want to acknowledge my family: Aurelia Mayen Pereira, Roberto Sánchez Ramírez, Zyanya Sánchez Mayen, and Heriberta Mayen Pereira for encouraging me in my studies abroad and in every other aspect regarding my personal decisions in life.

Content

List of Figures

List of Tables

List of Excerpts

List of Acronyms

AC97:	audio coder-decoder 1997.
AHB:	advanced high performance bus.
AMBA:	advanced microcontroller bus architecture.
APB:	advanced peripheral bus.
ASB:	advanced system bus.
ASIC:	application specific integrated circuits.
CAD:	computer aided design.
CODEC:	coder-decoder.
CPU:	central processing unit.
DAC:	digital to analog converter.
DDR2:	dual data rate 2.
DMA:	direct memory access.
DMAC:	direct memory access controller.
EDA:	electronic design automation.
FIFO:	first in first out.
FPGA:	field programmable gate arrays.
GNU:	GNU's not unix.
GPIO:	general purpose input-output.
GPL:	general public license.
GRLIB:	Gaisler Research intellectual property library.
GRMON:	Gaisler Research debug monitor.
GUI:	graphical user interface.
HD:	high definition.
HDL:	hardware description language.
IC:	integrated circuit.
ID:	identification data.
IP:	intellectual property.
ISE:	intelligent synthesis environment.
JTAG:	joint test action group.
LFE:	low frequency effect.
LSB:	least significant bit.
MODEM:	modulator-demodulator.
MSB:	most significant bit.
PCM:	pulse code modulation.
PC:	personal computer.
SoC:	system on chip.
SPDIF:	Sony Phillips digital interface format.
SYNC:	synchronization.
TDM:	time division multiplexing.
USB:	universal serial bus.
VHDL:	very high speed integrated circuit hardware description language.

1. Introduction

Since more than forty years ago, the semiconductor industry has obeyed Moore's Law: the density of the transistors within an integrated circuit doubles approximately every two years [1]. It is possible to manufacture 22 nm working circuits with higher performance, lower power consumption, and lower costs than their predecessors, i.e. fulfilling Moore's Law [2]. This has made embedded systems and system on chips (SoC) a reality, starting with the manufacture of integrated circuits in the late 1950's [1].

Aeroflex Gaisler provides a full library of IP cores (i.e. GRLIB) as an evaluation version freely distributed in full source code under the GNU GPL open-source license so a corporate or an academic user can evaluate and utilize the cores. Nonetheless, for commercial applications, Gaisler offers low-cost commercial IP licenses for the whole or parts of the library. GRLIB and all the required supporting development tools for embedded processors are written in VHDL [3]. In spite of that, Aeroflex Gaisler does not have an audio coder-decoder 1997 (AC97) digital audio controller intellectual property (IP) core in its Gaisler Research IP Library (GRLIB). Implementing this audio controller IP will enrich its IP cores catalogue and herein lays the relevance of this Master's thesis.

The objective of the project is to design an AC97 IP core to control compatible sound devices (e.g. the AC97 CODEC [4]) and reproduce digital audio. The controller must read and write AC97 registers, read and write accesses to the sound channel FIFOs, include a direct memory access (DMA) engine to fill/empty the FIFOs without central processing unit (CPU) use, and generate interrupts on various events. The AC97 IP core should be implemented in VHDL using the Two Process Design Methodology [5].

The document is organized in eleven sections and eleven appendices. The section after this brief introduction (Section 2) gives an overview of the AC97 in a typical system; Section 3 deals with the background on the AC97 standard (including the AC97 specifications) and IP cores; subsequently, Section 4 gives a technical background on the used communication bus in the system; Section 5 explains the Two Process Design Method and compares it with the dataflow style; then, the design of the AC97 core is described in detail in Section 6 (the functionality of the AC97 IP core in a system-level is explained first and then the first stage of the design is described followed by an explanation of the second stage of the design); following the design description, Section 7 shows the verification process in both stages of the design; the discussions are presented in Section 8 and the conclusion is placed in Section 9; further work on this project is included in Section 10; finally, the references can be found in Section 11. In the appendices, all the VHDL and C code (all the entities of the design, stand-alone test benches, and test C programs) written by the author is included.

2. Overview of the AC97 IP Core in a Typical System

Herein, a brief top view is given of where the AC97 IP core can be used. The AC97 IP core maybe introduced in an embedded system which uses the LEON3 processor, as it was done in this project. Such a system has several IP cores besides the AC97. Gaisler Research supports and develops their own collection of IP cores (i.e. GRLIB) and it is intended to introduce the AC97 IP core in the GRLIB library. The system where the AC97 IP core is placed uses the AMBA bus as a backbone for communication among the cores and the processor itself. Besides the processor and the required bus, a typical system where the AC97 core could be utilized would need an AC97 CODEC to be controlled by the core. The CODEC is situated in the prototype board and has connection with the FPGA and audio output jacks.

2.1. LEON3

The embedded system, in which the AC97 core will be introduced, utilizes a LEON3 processor, which is a 32 bit processor that uses the SPARC V8 architecture that in turn supports multiprocessing configuration. The LEON3 multiprocessor solution has a better performance under lower frequencies than single processor configurations. This means that it reduces the overall costs, power, and time to market; while still keeping compatibility with standard EDA tools. Nonetheless, the AC97 IP core was is included in a single processor configuration system. Another important feature of this processor is that it takes advantage of the plug&play[1] capability of the GRLIB IP library; thus, the development time is reduced and its flexibility is increased [6].

2.2. GRLIB

The GRLIB is a set of reusable IP cores meant to be used in a SoC. These cores are set on a common bus (the AMBA bus) and are compatible with the current computer aided design (CAD) tools. Furthermore, the library has the plug&play configuration so the cores can be introduced easily into it without making any global changes [7].

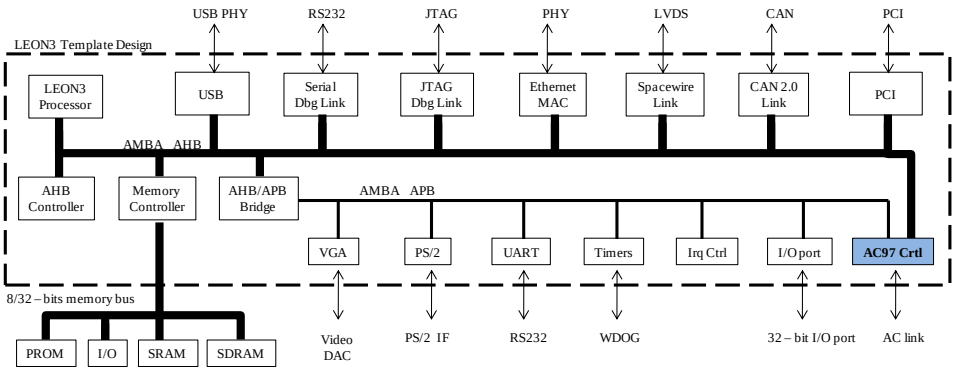


Figure 1: LEON3-based system designed with GRLIB.

[1] Plug&play is a capability developed by Microsoft for its Windows 95 and later operating systems that gives users the ability to plug a device into a computer and have the computer recognize that the device is there [7].

Figure 1 illustrates a LEON3-based system designed with GRLIB. Cores with higher bandwidth requirements are connected to the processor through the AHB bus and cores like the AC97 (highlighted in blue in Figure 1) are connected to the LEON3 through the APB bus. However, the AC97 IP core is connected also to the AHB bus through the DMA Engine (further explained in Section 6.3.3).

3. Background on the AC97 Standard and IP Cores

In this section, a brief background on the AC97 protocol and on IP cores will be given with the intention that the reader may get acquainted with these concepts, before getting into technical details about the design and the protocol itself.

The AC97 is a popular standard for computer audio and embedded systems, introduced by the Intel Corporation in 1997. It has been superseded by the high definition (HD) audio standard but it is still very much used in SoCs [8].

IP cores are components of integrated circuits (ICs) which have been designed and tested before-hand so they can be easily used in embedded applications. These kinds of cores are designed to be reusable hardware components [9]. They are part of the growing electronic design automation (EDA) industry trend towards the repeated use of previously designed and verified components [10]. This enables application specific integrated circuits (ASIC) or field programmable gate array (FPGA) designs to be built more quickly and at a lower cost. In this fashion, the opportunity to make it on time to the marketplace is ensured at a greater extent [9]. Ideally, an IP core should be entirely portable. This means that it should be able to easily be inserted into any vendor technology or design methodology [10].

3.1. AC97 Specification

This Section explains the AC97 specification in general terms. Then it goes into detail about the CODEC reset in Section 3.1.1, highlighting its importance. Subsequently, the communication between the controller and the CODEC is explained. The last Section gives a detailed description about the slots, which are part of the controller-CODEC communication protocol.

The AC97 consists of a digital controller (the core itself) and audio/MODEM CODECs. The CODEC sends the clock signal at a fixed frequency of 12.288 MHz (i.e. Bit Clock signal) to the controller. An external oscillator generates the clock (24.576 MHz) for the CODEC. It is divided by two inside the CODEC to obtain the required frequency for the core. The synchronization signal has to have a fixed frequency of 48 kHz (or 48 k frames are sent per second), which leads to 256 bits per frame (refer to Equation 1).

$$\frac{12.288\ M}{48\ K} = 256 \quad (1)$$

The communication between the core and the CODEC is called AC link digital serial interface protocol. This protocol consists of five signals (SYNC: the synchronization signal between the controller and the CODEC; BIT_CLK: the clock provided by the CODEC to the controller; SDATA_OUT: serial data going from the controller to the CODEC; SDATA_IN: serial data going from the CODEC to the controller; RESET#: reset signal provided by the controller) as shown in Figure 2:

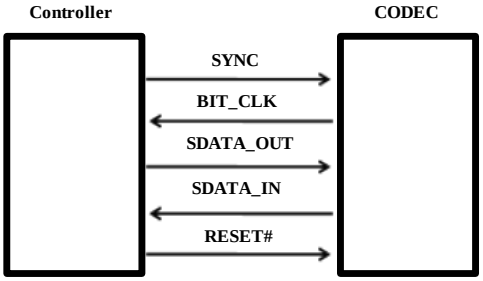


Figure 2: AC97 controller and AC97 CODEC.

3.1.1. CODEC Reset

The reset is vital for the correct communication between the core and the CODEC. When this signal is asserted, the codec will start to generate the Bit Clock. In the worst case scenario, if the reset signal is not properly asserted the CODEC will lose its clock and will lock up indefinitely [11].

The CODEC has three kinds of reset: cold reset, warm reset, and register reset. As seen in Figure 3, the cold reset is performed when the reset signal is pulled low for at least 1 µs. All internal circuitry and registers are set to their default values. This reset is done right after the system reset.

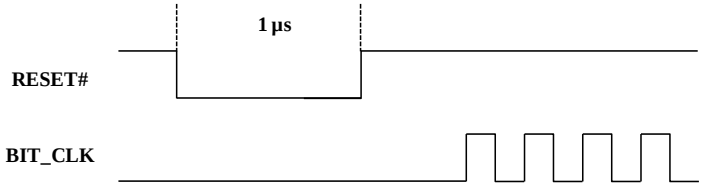


Figure 3: Cold reset.

Figure 4 shows how the warm reset happens. This reset occurs when the synchronization signal is held high for more than 1 µs when the Bit Clock is not present. It does not change the content of the internal circuits. The warm reset occurs when a restart is forced without powering down the circuitry. It is very important to realize that the synchronization signal has two purposes: warm reset when the Bit Clock is not present and proper synchronization between controller and CODEC when the Bit Clock is present. The issue of deciding which one is which is explained in more detail in Section 6.2.3.

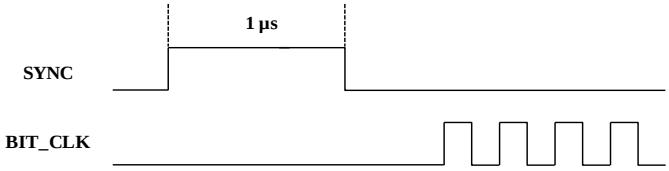


Figure 4: Warm reset.

The register reset occurs when any value is written into the reset register 0x0. This action will reset all the registers to their default values and will modify the configurations of circuitry accordingly, although it does not reset any other internal circuits [4] [12] [13].

3.1.2. AC Link Digital Serial Interface Protocol

The data flows from the CODEC to the controller and the other way around (in full duplex). The interface between the core and the CODEC is called AC link, as explained in Section 3.1. It consists of only 5 signals: clock, synchronization, reset, input data, and output data. The bidirectional data stream is divided into frames, with a frame rate of 48 kHz. Each one of these frames is subdivided into 13 slots of data employing a time division multiplexing (TDM) scheme. Slot 0, is known as the tag phase and it is the only one composed of 16 bits. Slots 1 to 12 are known as the data phase and they are composed of 20 bits each, summing up a total of 256 bits per frame (refer to Equation 1). For a better understanding of the protocol, Section 3.1.3 explains in detail the slot description.

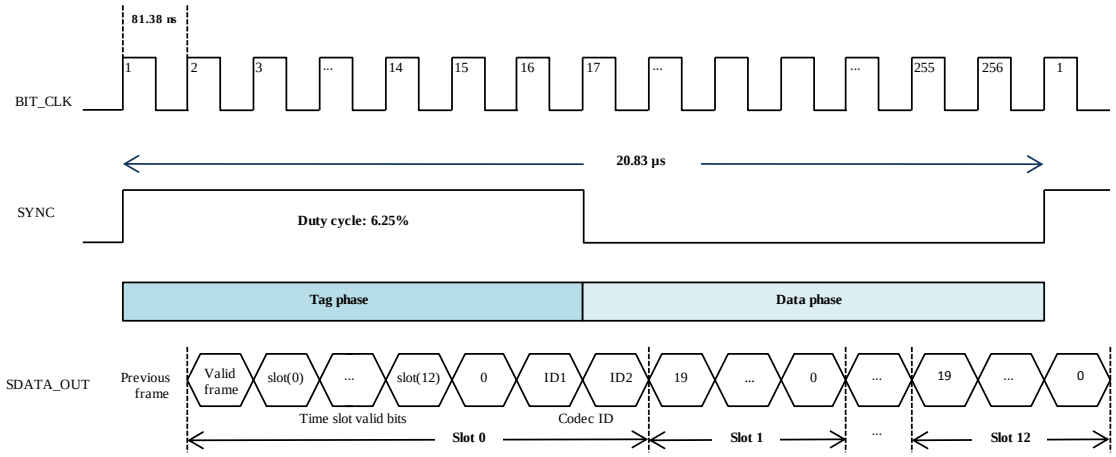


Figure 5: Detailed description of the AC Link.

The synchronization (SYNC) signal is sent by the controller to the CODEC to let it know that a new frame will arrive. It has a frequency of 48 kHz and a duty cycle of 6.25%. This signal remains high roughly the same time as the first phase of the data stream. It is very important to realize that the SYNC signal goes out one clock period before the data.

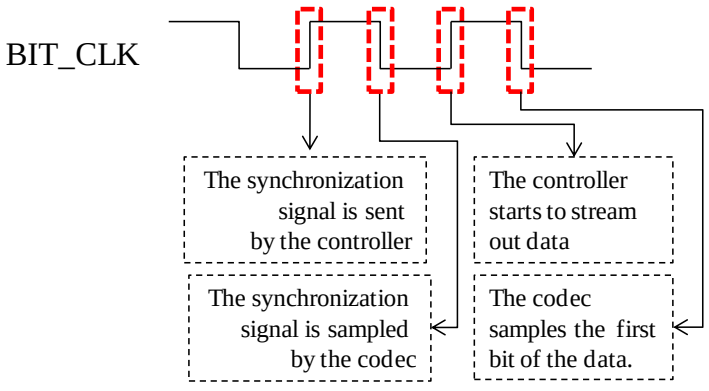


Figure 6: Different timing between the SYNC and SDATA_OUT signals.

The reset signal is given by the controller to the CODEC; it is an asynchronous cold reset, meaning that the CODEC registers are initialized to their default values. This is achieved by asserting it low for the minimum specified time (1 μs, around 13 Bit Clock cycles) and then de-asserting it high (please refers to Figure 3).

The data output signal goes from the controller to the CODEC. The audio streams use a pulse code modulation (PCM) scheme and a 16 bit resolution. The data going into the CODEC is sampled on the falling edges of the Bit Clock. The data going out of the controller is transitioned on the rising edges of the Bit Clock. This is illustrated in Figure 6 for the sake of clarity.

The data input signal goes from the CODEC to the controller. The data going into the controller is sampled in the falling edges of the Bit Clock. The data going out of the CODEC is transitioned on the rising edges of the Bit Clock.

3.1.3. Detailed Slot Description

As explained previously in Section 3.1.2, the audio frames in the AC97 communication protocol are divided into slots, and each slot has its own name and its content differs among each other. Table 1 summarizes how the slots are respectively named.

Table 1: Detailed slot description.

Slot number	Slot name
0	Tag phase
1	Control address
2	Control data
3	PCM playback left channel
4	PCM playback right channel
5	MODEM line 1 output channel
6	PCM playback center channel
7	PCM playback surround left channel
8	PCM playback surround right channel
9	PCM playback low frequency effect (LFE) channel
10	MODEM line 2 output channel
11	MODEM handset output channel
12	MODEM general purpose input output (GPIO) control channel.

The slots are in turn subdivided into bits. Slot 0 is called the tag phase and has only 16 bits. The description of the bits in slot 0 is represented in Table 2:

Table 2: Bit-wise representation of slot 0: tag phase.

Bit number	15	14:3	2	1:0
Description	Valid frame	Valid data in slots (1:12)	Reserved (0)	CODEC ID

Slot 1 is called the control address and it has 20 bits. The bits description in slot 1 is represented in Table 3:

Table 3: Bit-wise representation of slot 1: control address.

Bit number	19	18:12	11:0
Description	Read from (1)/write to (0) the CODEC's registers	Control register index. Specifies the register address of the operation.	Reserved (0's)

Slot 2 is called the control data and it has 20 bits. The bits description in slot 2 is represented in Table 4:

Table 4: Bit-wise representation of slot 2: control data.

Bit number	19:4	3:0
Description	If it is a read operation, they are filled with zeros. Otherwise, sends control data if it is a write operation.	Filled with 0's

Slot 3 is called the PCM playback left channel and it has 20 bits. The bits description in slot 3 is represented in Table 5:

Table 5: Bit-wise representation of slot 3: PCM playback left channel.

Bit number	19:4	3:0
Description	PCM audio data.	0's

Slot 4 is called the PCM playback right channel and it has 20 bits. The bits description in slot 4 is represented in Table 6:

Table 6: Bit-wise representation of slot 4: PCM playback right channel.

Bit number	19:4	3:0
Description	PCM audio data.	0's

It is important to emphasize that in slot 3, the sent out data are the 16 most significant bits (MSBs) (31:16) of the 32 bit PCM input. Similarly, in slot 4 the sent out data are the 16 least significant bits (LSBs) (15:0) of the 32 bit PCM input.

In implementation, only slots 0 to 4 are used. This is due to the fact that the prototype board has one single CODEC and it can drive only 1 stereo signal (i.e. PCM left & right channels). If the whole 6 audio channels (PCM left & right, PCM center channel, PCM surround left & right channels, PCM LFE channel) would have to be driven, 3 CODECs would be necessary. Furthermore, only playback will be implemented in the project as it was mentioned in the Abstract and Introduction.

4. Background on the Communication Bus

This Section explains the advanced microcontroller bus architecture (AMBA) that is used as the backbone for communication in the LEON3-based SoC in which the AC97 core will be introduced. After the AMBA description, the advanced high performance bus (AHB) and the advanced peripheral bus (APB) are explained along with their respective transfers.

4.1. AMBA

The AMBA specification defines an on-chip communication standard for designing high performance embedded microcontrollers. Three different buses are defined within the AMBA specification: the AHB, the advanced system bus (ASB), and the APB [14]. However, only the AHB and the APB buses were used in the design of the AC97 core.

The AMBA specification was designed: to satisfy four objectives. The first one is to ease the first-right-time development of embedded microcontroller products with one or more CPUs or signal processors. The second one is to be technology independent, along with the certainty that highly reusable peripheral and system macrocells can be migrated across a diverse range of IC processes; also, it has to be appropriate for full custom, standard cell and gate array technologies. The third objective is to encourage modular system design to improve processor independence, and provide a development road map for advanced cached CPU cores and the development of peripheral libraries. The last objective is to minimize the silicon infrastructure required to support efficient on-chip and off-chip communication for both operation and manufacturing test [8].

4.1.1. AHB

In order to attain the requirements of high performance synthesizable designs, the AHB bus has to be used. It supports multiple bus masters and provides high-bandwidth operation. This bus implements the required features for a high performance in high clock frequency systems, which include: burst transfers, split transactions, single cycle bus master handover, single clock edge operation, non tristate implementation, and wider data bus configurations [14].

A design that uses this bus may contain one or more bus masters. Typically, a system would contain at least the processor and the test interface as masters. However, it would be common for a DMA Controller to be included as a bus master. The external memory interface, an APB Bridge, and any internal memory are the most common AHB slaves. Any other peripheral in the system could also be included as an AHB slave. Nevertheless, low bandwidth peripherals normally reside on the APB [14].

4.1.2. AHB Transfers

The data is moved around the system by using data transfers. Just before an AMBA AHB transfer can start, the bus access has to be granted to a bus master. This process is started when the master asserts a request signal to the arbiter. Afterwards, the arbiter indicates when the master will be granted use of the bus. A granted bus master starts an AMBA AHB transfer by driving the address and control signals. These signals provide information on the address, direction, and width of the transfer, as well as an indication

if the transfer forms part of a burst (e.g. incrementing bursts, which do not wrap at address boundaries; wrapping bursts, which wrap at particular address boundaries) [14].

When the data has to be transferred from the master to the slave, a write data bus is used. Contrarily, whenever data is moved from a slave to the master, a read data bus is utilized. All the transfers consist of an address and control cycle, and one or more cycles for the data [14].

The slave can show the status of the transfer using the response signals in the *hresp*[1:0] vector. The possible responses are: *okay*, *error*, *retry*, and *split*. *Okay* is used to indicate that the transfer is progressing normally and the *hready*[2] signal goes high when the transfer has completed successfully. The *error* response indicates if a there was an error on the transfer and that it has been unsuccessful. The last two responses *retry* and *split* say that the transfer cannot complete immediately but the bus master should continue to attempt the transfer [14].

The AHB transfers can be classified into one out of four different types indicated by *htrans*[1:0]: *idle* (00_2), *busy* (01_2), *nonseq* (10_2), *seq* (11_2). The *idle* type indicates that no data transfer is required; *busy* indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately; *nonseq* indicates the first transfer of a burst or a single transfer; the remaining transfers in a burst are *seq* and the address is related to the previous transfer [14].

In the AMBA AHB protocol, burst operations are defined and the signal *hburst*[2:0] in the bus gives information about the burst: single transfers (*single* 000_2), incrementing bursts of unspecified length (*incr* 001_2), 4-beat wrapping burst (*wrap4* 010_2), 4-beat incrementing burst (*incr4* 011_2), 8-beat wrapping burst (*wrap8* 100_2), 8-beat incrementing burst (*incr8* 101_2), 16-beat wrapping burst (*wrap16* 110_2), 16-beat incrementing burst (*incr16* 111_2) [14].

4.1.3. APB

The APB is optimized for minimal power consumption and reduced interface complexity. It appears as a local secondary bus that is encapsulated as a single AHB slave device. The APB Bridge appears as a slave module which handles the bus handshake and control signal retiming on behalf of the local peripheral bus. To interface to any peripherals which are low bandwidth and do not require the high performance of a pipelined bus interface, the AMBA APB should be used. Usually, an AMBA APB implementation contains a single APB Bridge which is required to convert AHB transfers into a suitable format for the slave devices on the APB. The bridge provides latching of all address, data and control signals. The APB Bridge is the only bus master on the AMBA APB. Furthermore, the APB Bridge is also a slave on the higher level system bus [14].

4.1.4. APB Transfers

The APB bus might be used either for a read or a write transfer. During a write transfer, the address, write data signal, write signal, and the select signal change at the same time

[2] The data in a transfer can be extended using the *hready* signal. When it is low, this signal causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data [14].

after the rising edge of the clock. The first clock cycle of the transfer is called the *setup* cycle. After the following clock edge the enable signal *penable* is asserted (i.e. the *enable* cycle is taking place. The address, data, and control signals all remain valid throughout this cycle; the transfer completes at the end of the *enable* cycle). The *penable* enable signal, will be de-asserted at the end of the transfer. The select signal will be pulled down as well, unless the transfer is to be immediately followed by another transfer to the same peripheral. In order to reduce power consumption, the address and the write signals will not change after a transfer until the next access occurs [14].

For a read transfer, the timing of the address, write, select, and strobe signals are just the same as for a write transfer. Nonetheless, in the case of a read transfer, the APB slave should provide the data during the *enable* cycle and the data is sampled on the rising edge of the clock at the end of the *enable* cycle [14].

5. The Two Process Design Method

In this section, the Two Process Design Method is briefly explained. Afterwards, there is a comparison between the previously mentioned method and the traditional dataflow style.

The Two Process Design Method is used at Gaisler Research because it greatly simplifies the synthesis and design of complex embedded systems using a hardware description language (HDL) [15] [16]. The systems designed at Gaisler use this method, thus the AC97 core was designed using it as well. Basically, the method consists of a combinational process and a sequential process. In the former, all the combinational logic takes place whereas in the latter, all the registers are updated at the rising edge of the clock signal (i.e. the sole signal in the process sensitivity list). In this way, the output of the combinational process is the input of the sequential one, and also the output of the sequential process is the input of the combinational one. This idea is best expressed in the Figure 7.

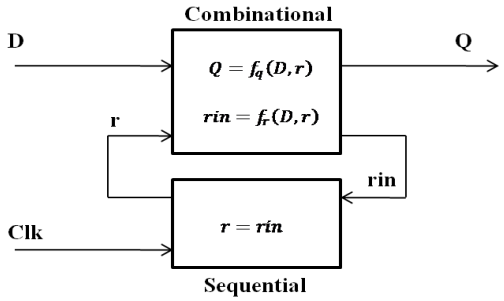


Figure 7: Generic two process circuit. Courtesy of Gaisler Research [15].

It is important to mention that this method is only applicable to synchronous and single clock designs. The goals of the method are: provide uniform algorithm encoding, increase abstraction level, improve readability, clearly identify sequential logic, simplify debugging, improve simulation speed, and provide one model for both synthesis and simulation. To achieve these goals, it is necessary to use three simple means: using record types in all port and signal declarations; using only two processes per entity; and using high level sequential statements to code the algorithm [15].

5.1. Comparison with the Dataflow Method

Not all the VHDL designs are done using the Two Process Design Method described before. The most commonly used design method for synthesizable VHDL models is called the dataflow style. The dataflow style arose from the old school hardware design where the engineers were used to a schematic entry as the design method. Later on, when VHDL started to be used as a design tool, the dataflow design style was similar to the schematics [17]. The low complexity of the designs at that time (e.g. less than 50 k gates in the early 1990's [15]), partly due to restrictions of the synthesis tools, made the dataflow style acceptable when coding VHDL [17].

This old method consists of many concurrent VHDL statement and small processes connected together through signals that in turn build components with a specific functionality. Designs that use the dataflow method become difficult to understand and analyze when the number of concurrent statements increases. The complexity in

understanding the code is due to the fact that concurrent statements and processes are not executed in the order they are written [17].

Table 7: Dataflow VS Two Process Comparison [15].

	Two Process Method	Dataflow Coding
Adding Ports	• Add field in interface record type.	• Add port in entity declaration. • Add port to sensitivity list (input). • Add port in component declaration. • Add signal to port map of component. • Add definition of signal in parent.
Adding Registers	• Add field in register record type.	• Add two signal declarations (d & q). • Add q-signal in sensitivity list. • Add driving signal in comb. process. • Add driving statement in seq. process.
Debugging	• Put a breakpoint on first line of combination process and step forward. • New signal values visible in local variable v.	• Analyze how the signal(s) of interest are generated. • Put a breakpoint on each process or concurrent statement in the path. • New signal value not immediately visible.
Tracing	• Trace the r-signal (state) • Automatic propagation of added or deleted record elements.	• Find all signals that are used to implement registers. • Trace all found signals • Re-iterate after each added or deleted signal

Table 7 illustrates the usefulness of the Two Process Method with the standard dataflow design style by comparing common development tasks. From Table 7, it can be noted that common development tasks are done with less editing or manual procedures, thereby improving efficiency and reducing coding errors [15].

6. The Design of the AC97 Core

This Section starts with the taken design choices; then, there is a brief explanation of the functionality of the AC97 IP core in a system level perspective; the next section describes the design of the AC Link, its correct timing module, and the interface between the APB Bridge and the previous two entities. The second stage of the design deals with the introduction of the DMA and FIFO to the core, and its correct interface with the AHB bus.

6.1. Design Choices

The whole design was made in VHDL using the ModelSim environment suite and following the Two Process Design Method.

The design process was divided in two phases. The first phase aimed to communicate the processor with the AC97 IP core through the APB bus, so the core could transmit o the CODEC commands given by the processor. The second phase aimed to communicate the processor with the AC97 IP core through the AHB bus (and still keep the communication with the APB bus) to fetch stored data in memory. The first stage was designed and simulated in ModelSim, and then verified in hardware before moving forward to the second stage.

Designing the core in two stages gave more fluency to the project, since it was easier to verify a simple communication between the core and the processor by first using the APB bus before adding more complexity to the design and start using the AHB bus (which in turn, the communication will require to design a DMA Engine). By verifying the first stage in hardware it also was easier to remove bugs from the design. However, taking the decision of designing the core in two stages has the implication of doing the verification twice, which requires more time. But in this way, it is certain that the first stage of the design is working properly.

If the project would have been carried out in a single stage, probably the design would have been finished before (verified in simulation), but it is likely that it would not have been verified completely in hardware. Meaning that perhaps simple communication between the core and the processor would not have been achieved.

Furthermore, the design was broken down into many different entities to have order in the code and also to perform specific tasks in the entities. For example, the Clocks entity was used for the CODEC reset and to get the rising and falling edges from the Bit Clock; in the AC Link entity the parallel data is being serialized and the synchronization signal is sent out with its correct timing; the Interface entity stores the commands in FIFOs and communicates with the APB bus; a main entity was included to make the design portable and compact within the entire system.

In the same train of thought, the entities have different inputs and outputs (record types for communication inside the core and with the system bus). It was thought that by having entities in charge of a specific collection of signals, the design could be visualized more clearly.

6.2. The Functionality of the AC97 IP Core in the LEON3-Based System

A brief overview of the functionality of the AC97 controller within the LEON3-based system is explained herein. The core stays idle until it is selected and enabled by the APB Bridge, either for a read or a write transfer. Whenever the LEON3 processor wants to write a command to the core (e.g. turn up the volume on the loudspeakers), the information is sent through the AHB bus, passing through the APB Bridge, and finally reaching the AC97 core (please refer to Figure 1). The command is then transmitted from the core to the CODEC via the AC link.

After a read or write completion, the AC97 core remains idle until the processor requests another task to be performed by the core. For instance, if an audio file has to be fetched from memory, this instruction is sent by the processor in the same way as if it were a regular command. Afterwards, the core realizes that it should use the DMA engine to get this data from memory.

The data transfer is done by 4-beat burst transfers through the AHB bus and stored temporarily in a core's FIFO. Immediately after the data is available in the before mentioned FIFO, it is streamed out of the core to the AC97 CODEC, so the audio data can be listened in the loudspeakers, which in turn are driven by the CODEC's DAC.

6.3. First Stage

Figure 8 clearly illustrates the Top Module interconnection of the AC97 IP core at its first stage. The thin lines are single bit signals whereas the thick lines are record types (*apbi* and *apbo* are further explained in Section 6.3.4; *inl_link* and *outl_link* are further explained in Section 6.3.2).

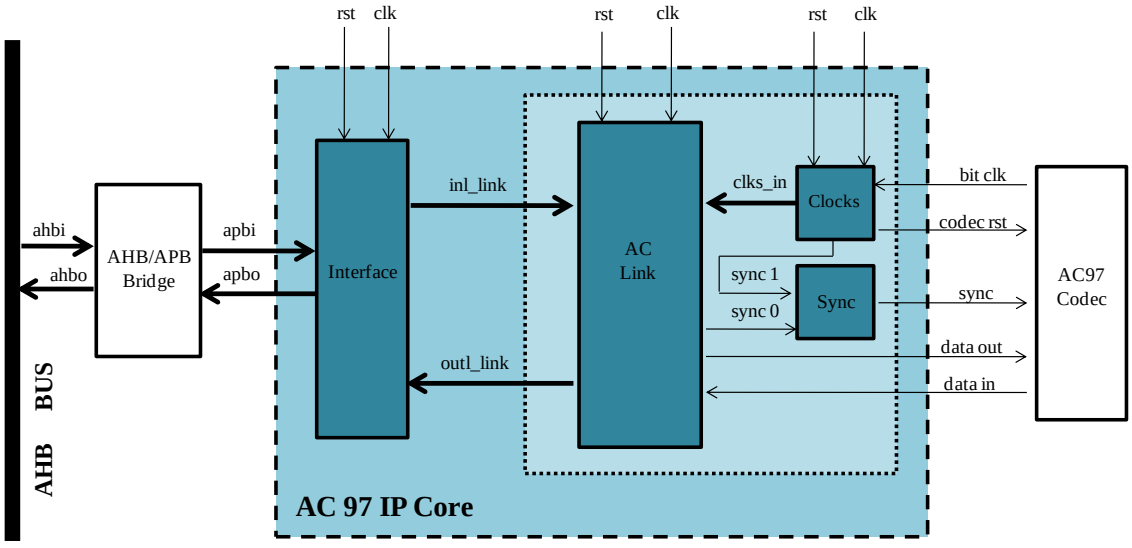


Figure 8: AC97 IP core at its first stage.

6.3.1. The Clocks Entity

The clock that drives the AC Link is coming from the CODEC, as it was previously explained in Section 3.1. This clock is slower than the clock that drives the entire system. Nonetheless, there must be synchronization between these two clocks;

otherwise communication will never be achieved. Furthermore, both the rising and falling edges are needed in the AC97 core. According to the AC97 specification, the output data (controller to CODEC) is sampled on the rising edges of the Bit Clock and the input data (CODEC to controller) is sampled on the falling edges of the Bit Clock. However, to have different edges on a design might result in not synthesizable code, since most synthesis tools do not support dual edge behavior (i.e. falling edge and rising edge) [18].

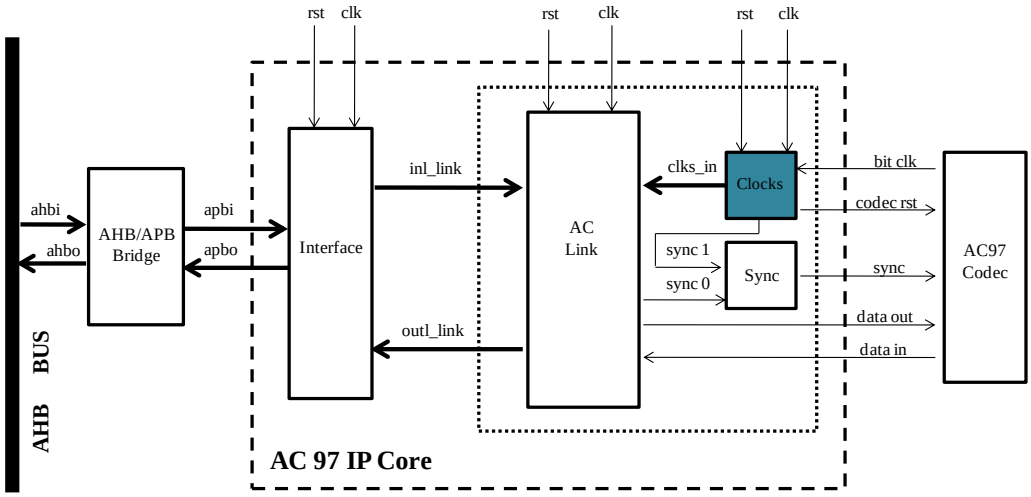


Figure 9: Clocks entity.

In order to obtain the rising and falling edges from the slower clock, an entity was designed (highlighted with dark blue in Figure 9 and named as Clocks). Obviously, the edges of the slower clock still have to be synchronized with the system clock. The entity has the Bit Clock as an input, and it is sampled with the system clock. From this entity, two signals are obtained and are used as the rising and falling edge clocks for the AC Link. The rising edge will be the trigger to update the outputs of the AC97 core, but the system clock will be the input signal in the sequential process. The falling edge was not actually used in the design of this core, since the purpose of this Master's project was to send only data from the controller to the CODEC and not the other way around.

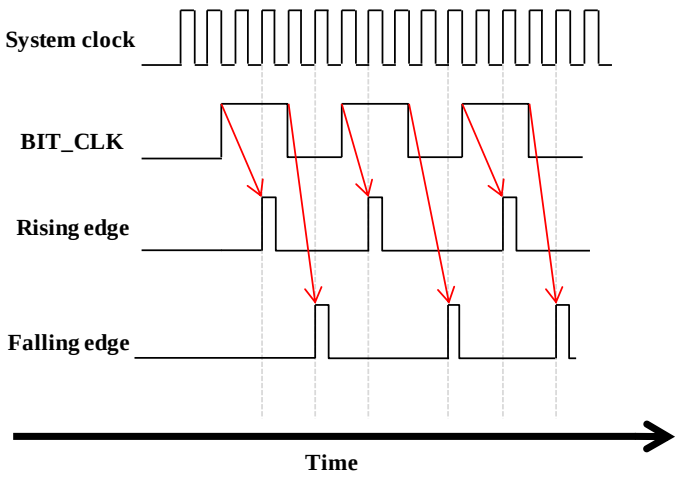


Figure 10: Rising and falling edges synchronized with the system clock.

As it can be observed in Figure 10, the system clock and Bit Clock are not synchronized. In the design, an entity is in charge of obtaining the rising and falling edges from the Bit Clock and have them synchronized with the system clock. It is important to realize that the falling edge clock is just the rising edge clock but 180° out of phase, which means that no real falling edge was used in the design.

The way the acquisition of the rising and falling edges was implemented, was serializing four flip-flops (refer to Figure 11). The input of them was the Bit Clock and they were sampled with the faster system clock. However, the Bit Clock had first to be passed through an anti-meta-stable stage. This also was done by serializing flip-flops (only two in this case). If data is sampled without any synchronization phase, the value of the sampled data would be uncertain (i.e. either 0 or 1). Thus, the output of the first flip-flop in the anti-meta-stable stage will go meta-stable, but the next flip-flop does not look at the data until a clock period later, which gives time to the previous flip-flop to stabilize the data. Now, the data passed through the next flip-flops will be stable data that is safe to sample.

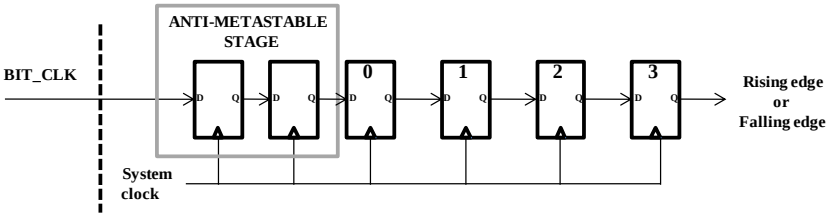


Figure 11: Implementing the rising and falling edges.

In this way, whenever the system clock samples two consecutive zeros and then two consecutive ones, the edge will be rising. Contrarily, in the case that the system clock samples two consecutive ones and then two consecutive zeroes, the edge will be falling. Table 8 resumes these combinations.

Table 8: Combinations for the rising and falling edges.

Q0	Q1	Q2	Q3	Edge
0	0	1	1	Rising
1	1	0	0	Falling

It is very possible that the system clock could not sample exactly two consecutive zeros and two consecutive ones (or vice versa). Thus, in any other case the edges will be set to zero and the counter for the warm reset will be reinitialized.

Additionally there will be a delay of four system clock cycles before either rising or falling edges are detected. However, this is not critical because it will only happen when the system is powered up or after a reset. Furthermore, the delay is in terms of the system clock and not with respect to the Bit Clock.

The case 0000_2 is a special one because if that happens, it means that there is not a present clock. If this is the case for at least 1μs (according the AC97 specification), a warm reset has to be asserted to the CODEC (please refer to Section 3.1.1 AC link for the details on the different kinds of CODEC reset). In different cases besides 0000_2 and the two previously explained in Table 8, the warm reset counter will be reinitialized and the states of the last edges will be set to zero.

zeferring to Figure 9, the output of the Clocks entity to goes into the AC Link is named *clks_in*. This port uses the record type *clks_in_type*. Table 9 explains such record type.

Table 9: *clks_in_type* signals.

Signal Name	Function
r_clk	This signal is the rising edge of Bit Clock and is sent by the Clocks entity to the AC Link. It is used in the AC Link to trigger the outputs to the CODEC.
f_clk	This signal is the falling edge of Bit Clock. However, it is not currently used in the design.
codec_rst	This signal contains the cold reset asserted by the Clocks entity. The cold reset is used in the AC Link to control the synchronization signal, sent to the CODEC.

6.3.2. The AC Link Entity

The connection between this entity and the Interface entity is done by using the input *inl_link* and the output *outl_link* (refer to Figure 12). These signals use the record types *aclink_inl_type* and *aclink_outl_type,* respectively. Table 10 explains the content of the input record type and Table 11 explains the content of the output record type.

Table 10: *aclink_inl_type* signals.

Signal Name	Function
valid_if	This signal tells the AC Link when a valid command has been written to the Interface.
ready_if	This signal tells the AC Link when to start streaming out the command to the CODEC.
equal	This signal tells the AC Link when the number of sent out commands is equal to the written commands in the Interface.
adres_if [31:0] [3]	The address in the command is sent from the Interface to the AC Link in this vector.
data_if [31:0] [4]	The data in the command is sent from the Interface to the AC Link in this vector.

Table 11: *aclink_outl_type* signals.

Signal Name	Function
cmd_rqst	This signal asks the Interface for a command.
Ack	This signal tells the Interface when the AC Link started to send data to the CODEC.
donecopy [7:0]	This signal tells the Interface how many frames have been sent out to the CODEC.

[3] Only the 7 LSBs of the address vector are used because the CODEC has 128 registers.
[4] Only the 16 LSBs of the data vector are used because only 16 bits of data can be written into the CODEC's registers.

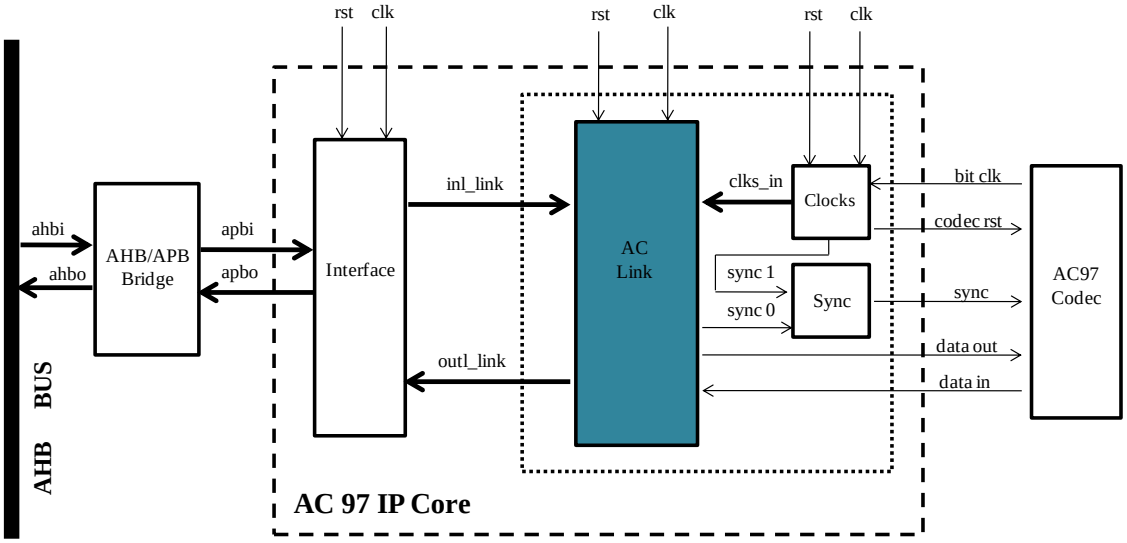


Figure 12: AC Link entity.

The AC Link (highlighted with dark blue in Figure 12 and named as AC Link) is a master of the Interface between the APB bus and the core. The main role of this entity is to convert the incoming parallel data to serial data, so the CODEC can receive these data according to the AC97 specifications. In addition it is also in charge of sending the synchronization signal to the CODEC. Figure 13 explains in a flow chart how the AC Link works.

The flow chart in Figure 13 starts in upper right corner, where the reset conditional is. If reset is equal to zero, the signals and outputs are set to their initial values. Right after the reset has been asserted the hardware is waiting for the rising edge of the Bit Clock. Until this edge is detected, the AC Link will not do anything.

Basically, there are six conditions that might be executed in parallel. However, some of them depend on the result of previous operations or input signals from the Interface entity. Whether they are mutually exclusive or not is further explained in this same section. In Figure 13, the left-most conditional asserts the request signal if there has not been written a valid command in the Interface and it keeps on sending the asserted request signal until a valid signal is sent back to it.

Working from left to right, the next condition in the flow chart of Figure 13 is used to prepare the slots when they contain a command, when they have PCM data, or when both a command and PCM data are present in the frame. The AC link prepares the frame's slots with the bits needed in the tag phase of the protocol, and includes the address and the data accordingly, as it will be further explained in Section 6.3.3.

The next condition is used to check if the equal signal is low and the CODEC reset is 1. The equal signal means that the number of written commands in the Interface is equal to the number of sent out frames. If the codec reset is 0 it means that a cold reset has been sent to the CODEC and the AC Link cannot send frames to it. If the equal signal is 0 and a cold reset has not been sent to the CODEC (e.g. CODEC reset signal is 1), it is possible that the AC Link can send the synchronization signal to the CODEC and send an acknowledgment signal to the Interface; else, the synchronization signal is not sent.

The next conditional in Figure 13 explains when to send the data to the CODEC. Whenever the ready signal comes asserted from the Interface and the synchronization has been sent to the CODEC, the AC Link will start to send out data to the CODEC. After the last bit (i.e. the 256th bit) or a complete frame has been sent out, a done signal is asserted in this same condition that will be further used to increase the counter that keeps on track of the sent frames.

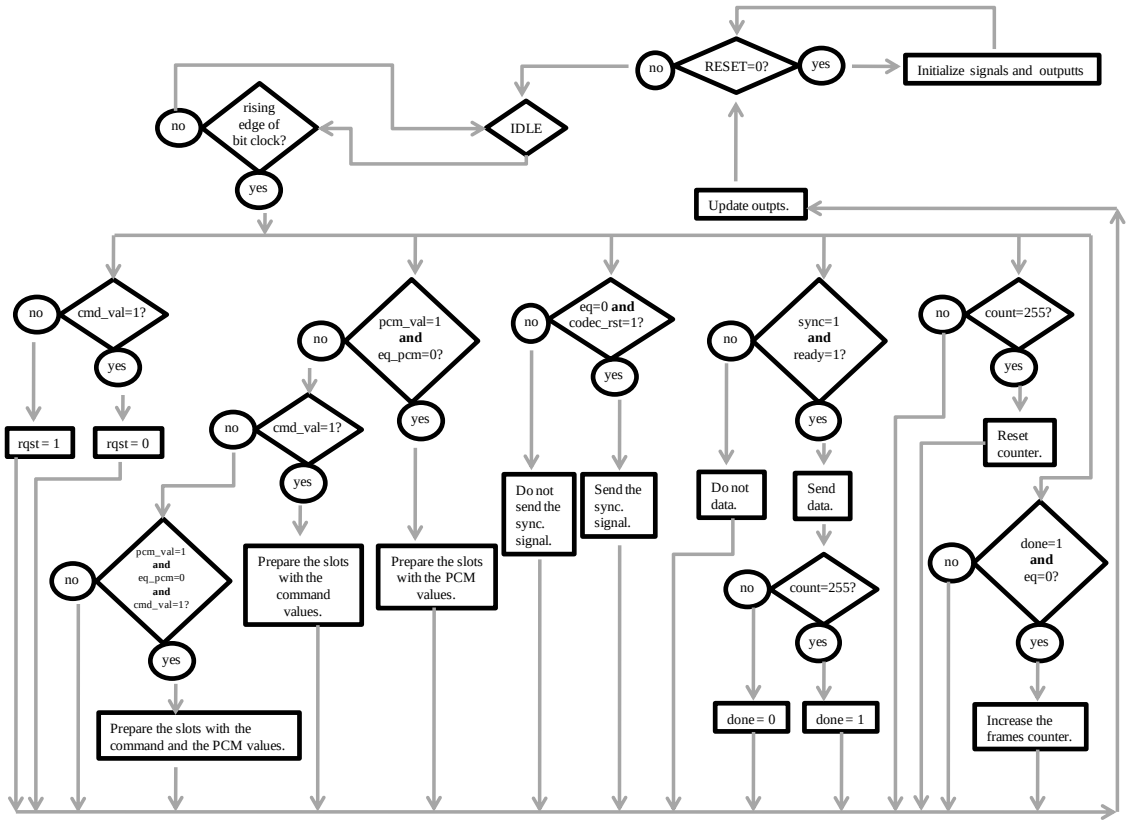


Figure 13: AC Link flow chart.

Outside the previous condition, the counter for the bits is checked against the maximum count (255). If the value has been reached, the counter has to be reset. It was chosen to reset the counter in this way instead of letting it wrap around itself after it had reached the maximum count to avoid not sending the last bit in the frame. For this purpose, the counter has 9 bits instead of 8 bits.

The last condition in Figure 13 is used to increase the frames counter. If the *done* signal is 1 (previously explained in this same flow chart) and the equal signal (checked in the Interface entity) is 0, the counter for the sent out frames has to be increased by one. This count is received and checked by the Interface. If the number of sent out frames is equal to the number of written commands, the equal signal is asserted; otherwise it will not be asserted.

6.3.3. Serializing the Parallel Data

The way the output data is transmitted is by calling a function that serializes the incoming 32-bits data vector. The function receives the vector which is used as a counter for the synchronization signal and returns a single bit for the data output to the CODEC. When the slot's count is between 0 and 15 (16 bits), the first slot is streamed

out; when the count is between 16 and 35 (20 bits), the second slot is sent out. This goes on until the count reaches 255, so the 13 slots have been transmitted. All the slots are transmitted in a MSB justified fashion (i.e. most significant bit goes first).

In this implementation, only the slots 0 up to 4 are used, as it was explained in Section 3.1.3. All slots are 20 bits wide, but in the first slot only the 16 MSBs are transmitted, in order to simplify and not have to deal with different vector sizes.

The transmission is done by logically shifting to the left (`sll` function in VHDL) the previously loaded slots. They are shifted depending on the slot's bit count. For instance, in the first slot, if the counter is 0, the vector will be shifted 0 times; if the counter is 12, the vector will be shifted 12 times. The slots are indexed from 0 to 19 and not 19 down to 0, in order to always shift the least significant bit and use the same operation for even different sized slots and keep the code simpler. In this way, the bit 0 in every vector slot is the bit that will be transmitted to the output.

Table 12: Values in slots while sending a command.

Slot 0																			
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*Not applicable*				1	1	1	*Zeros*												
Slot 1																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	0	*Address*							*Zeros*											
Slot 2																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*Data*																*Zeros*			

If the core has to send a command, the slots are loaded according to Table 12. Slot 0 is loaded with 0xE000 assuming the frame is valid (bit 15), slots 1 and 2 are valid (bits 14 and 13, respectively), and writing the correct CODEC ID in bits 1 and 0 (in this case 00 for the Analog Devices CODEC). Bits 12 to 2 are loaded with zeros since PCM data will not be sent yet, and bit 2 is reserved (e.g. filled in with a 0). Slot 1 is loaded as shown in Table 12 because a value will be written into the CODEC (bit 19 set to 0), the register address of the operation is specified in bits 18:12, and bits 11:0 are reserved (i.e. filled with zeros). Slot 2 is loaded as shown in Table 12 since it is a write operation (otherwise, the whole slot would be filled with zeros) and bits 3:4 should be filled in with zeros.

When the core sends PCM data to the CODEC, the slots will change according to Table 13. Slot 0 is loaded with 0x9800 assuming the frame is valid (bit 15), slot 1 (bit 14) and slot 2 (bit 13) are not valid since the CODEC's registers will not be targeted, PCM data on the left channel is valid (bit 12), PCM data on the right channel is valid (bit 11), and writing the correct CODEC ID in bits 1 and 0 (in this case 00 for the Analog Devices CODEC). Bits 10 to 2 are loaded with zeros and bit 2 is reserved (e.g. filled in with a 0). Correspondingly, slots 1 and 2 are loaded with different values if PCM data is sent instead of a command.

Referring to Table 13, slot 1 is loaded with 0x0 because nothing will be read from the CODEC (bit 19 set to 0), no register address will be targeted (bits 18:12), and bits 11:0 are reserved (i.e. filled with zeros). Referring to Table 13, slot 2 is loaded with 0x0

because no register address will be targeted. Thus, there is no need to send a value. Bits 3:4 should be filled in with zeros.

Since PCM data has to be sent to the CODEC, slots 3 and 4 are loaded in a different way compared to writing into the CODEC's registers. In Table 13, bits 19:4 from slot 3 are loaded with the 16 MSBs of the 32 bits PCM input and these data is sent to the left channel. In the same vein, bits 19:4 from slot 4 are loaded with the 16 LSBs of the 32 bits PCM input and these data is sent to the right channel.

Table 13: Values slots while sending PCM data.

Slot 0																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*Not applicable*				1	0	0	1	1	*Zeros*										
Slot 1																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*Zeros*																			
Slot 2																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*Zeros*																			
Slot 3																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*16 MSBs of the 32 bits PCM input.*																*Zeros*			
Slot 4																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*16 LSBs of the 32 bits PCM input.*																*Zeros*			

Nonetheless, it is possible to send both a command and PCM data in the same frame. If this has to be done, the slots are loaded differently. Table 14 explains how slot 0 is loaded in this case.

Table 14: Values slots while sending PCM data and a command.

Slot 0																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*Not applicable*				*Ones*					*Zeros*										
Slot 1																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	0	*Address*								*Zeros*										
Slot 2																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*Data*																*Zeros*			
Slot 3																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*16 MSBs of the 32 bits PCM input.*																*Zeros*			
Slot 4																				
Bit	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	*16 LSBs of the 32 bits PCM input.*																*Zeros*			

Referring to Table14, slot 0 is loaded with 0xF800 assuming the frame is valid (bit 15), control address in slot 1 (bit 14) and control data in slot 2 (bit 13) are valid, PCM data on the left channel is valid (bit 12), PCM data on the right channel is valid (bit 11), and

writing the correct CODEC ID in bits 1 and 0 (in this case 00 for the Analog Devices CODEC). Bits 10 to 2 are loaded with zeros and bit 2 is reserved (e.g. filled in with a 0). In this case slot 1 and slot 2 are charged in the same way as in Table 12; slot 3 and slot 4 are filled with the values used in Table 13.

6.3.4. The Link/Clocks Entity

There was a need to include the Clocks and the AC Link entities into a single higher level unit to make the design more conveyable and compact, because in this way, all the needed hardware to send the frames to the CODEC will be enclosed in one single entity (i.e. the AC Link entity to serialize the parallel input data and send the synchronization signal with its right timing; the Clocks entity to obtain the rising edge from the Bit Clock). This entity is named Link/Clocks entity. The highlighted square in light blue in Figure 8 indicates where in the core this takes place.

Moreover, in the Link/Clocks entity it will be decided when the synchronization signal will be treated as a warm reset or for synchronization purposes. This decision takes places in the dark blue square in Figure 8, which is a simple *OR* operation. On any occasion the Bit Clock is present, the synchronization signal will be used for synchronization purposes between the AC Link and the CODEC, and whenever the Bit Clock is not present during 2 µs the synchronization signal will be used as a warm reset to the CODEC.

Thus, this simple decision takes places in this entity instead of doing so in the Top Module, which is instantiated in the `leon3mp.vhd` file without doing any operations.

6.3.5. The AHB/APB Bridge

The AHB/APB Bridge entity was reused from the GRLIB library. It is an AMBA AHB slave interface which connects the AHB bus with any given peripheral. This bridge also acts as an APB master from the APB slave (i.e. the AC97 core).

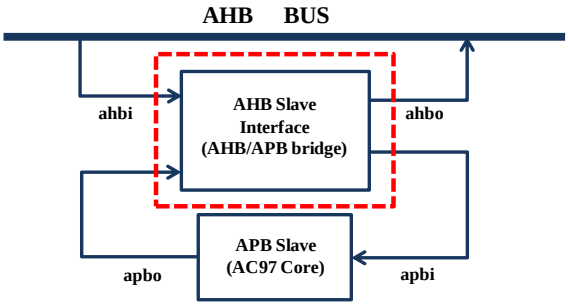


Figure 14: AHB/APB Bridge.

6.3.6. The Interface Entity

In Figure 14, the AHB/APB Bridge has incoming and outgoing signals. As a matter of fact, they are a collection of signals grouped into different buses. An input of the bridge coming from the AHB bus is a signal called *ahbi* (refer to Figure 14), which uses the record type *ahb_slv_in_type*. An output of the bridge is called *ahbo* in Figure 14 and it goes back to the AHB bus and uses the record type *ahb_slv_out_type.*

The bridge's output to the AC97 core is named *apbo* in Figure 8 and uses the record type *apb_slv_out_type*. On the other hand, the bridge's input called *apbi* in Figure 8 comes from the AC97 core and uses the record type *apb_slv_in_type*. The relevant signals for the AC97 core that are coming from the *apb_slv_in_type* are shown in Table 15:

Table 15: *apb_slv_in_type* signals for the AC97 core [14].

Signal Name	Function
Psel	Indicates that the slave device is selected and a data transfer is required.
penable	Used to indicate the second cycle of an APB transfer.
paddr [31:0]	APB address bus; driven by the peripheral bus bridge unit.
pwrite	When high, indicates an APB write access; when low, a read access.
pwdata [31:0]	The write data bus is driven by the peripheral bus bridge unit during write cycles.

The relevant signals for the AC97 core that are coming from the *apb_slv_out_type* are explained in Table 16:

Table 16: *apb_slv_out_type* signals for the AC97 core [7] [14].

Signal Name	Function
prdata [31:0]	The read data bus is driven by the selected slave during read cycles.
Pconfig	Contains information such as vendor ID, device ID, APB address, APB address mask.
pindex	Indicates the APB slave index.
pirq [31:0]	Interrupt line driven by the core.

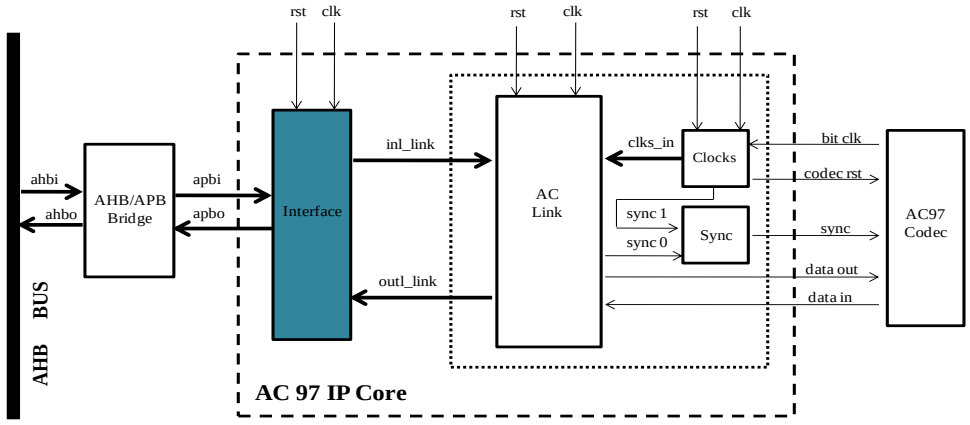


Figure 15: Interface entity.

The interface between the APB Bridge and the AC Link works as a slave from the latter. Thus, the master requests data from the slave all the time. If there is new data to be written into the Interface, the core has to be selected and enabled on the APB bus, and the write signal has to be asserted. In order to use the AHB/APB Bridge, the GRLIB library and AMBA package had to be added into the Interface entity (see Appendix A.6):

The flow chart in Figure 16 describes how the Interface entity works.

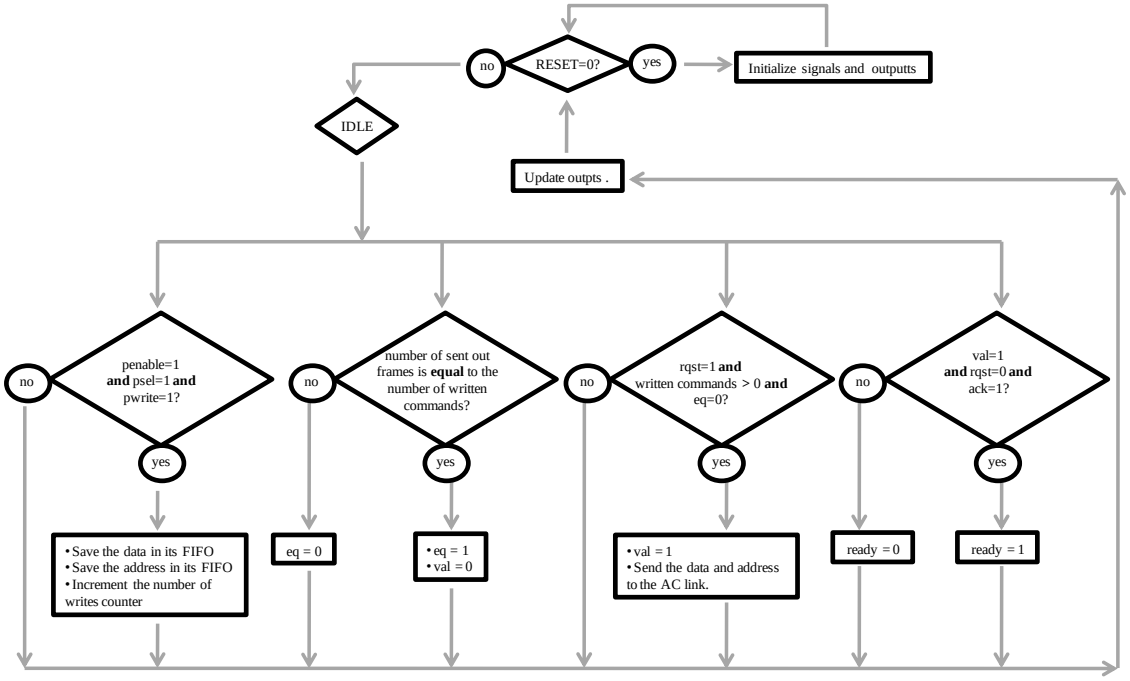


Figure 16: Interface entity flow chart.

The flow chart in Figure 16 starts from the top, where the reset conditional is. If reset is equal to zero, the signals and outputs are set to their initial values. After the reset has been asserted, the Interface entity stays idle until the core is selected and enabled.

These events are represented as conditionals in Figure 16. From left to right in this figure, the first conditional describes when a command has to be written into the core. In order for this to happen, the APB slave (i.e. the core) has to be selected, enabled, and the write signal has to be asserted. If so, the address and the data are stored in different FIFOs, and a *writes counter* is incremented by one (it counts how many writes have occurred). If this condition is not met, the Interface entity stays idle and none of the other conditionals can happen.

The next conditional from left to right in Figure 16, depicts when the number of sent out frames by the AC Link to the CODEC is equal to the number of written commands to the Interface. If this happens, the *eq* flag is asserted and the valid flag (*val* in the figure) is set to zero; if this does not happen, the *eq* flag is de-asserted and the next conditional cannot happen.

Following the previous conditional in Figure 16, there is another condition in the Interface. It checks when to assert the *val* flag and when to send the data and address to the AC Link. This occurs when the request input from the AC Link (*rqst* in Figure 16) is 1 and the *writes counter* is greater than zero and the *eq* flag is 0. If this does not happen, the Interface stays idle and the next conditional cannot happen.

The last conditional in Figure 16 checks when to send the ready signal to the AC Link, which indicates that the AC Link should stream out the command to the CODEC. The ready signal is asserted if the valid signal is 1 and the request from the AC Link is 0 and

if the incoming acknowledge signal (which indicates that a command has been received by the AC Link) is 1. Else, it will be set to zero.

At any time that the processor wants to read from the core, no extra logic is needed. The data will not collide with data coming from other slaves because it will not be outputted unless the core is selected by the AHB/APB Bridge. Hence, the read registers will be always available.

6.3.7. Plugging- in the Core into the System

To plug-in the core into the LEON3-based system in order to simulate the whole environment and verify the core's functionality, some files had to be modified. First of all, the core information was added to the `devices.vhd` and the `config.vhd` files. This modification is explained in Excerpts 1 and 2:

```
package devices is
constant GAISLER_AC97 : amba_device_type := 16#08C#;
constant gaisler_device_table :
device_table_type:=(GAISLER_AC97=>"AC97 Controller ");
```

Excerpt 1: Adding the AC97 core into Gaisler Research device ID's in the `devices.vhd` file.

```
package config is
constant CFG_AC97_ENABLE : integer := 1;
end;
```

Excerpt 2: Adding the AC97 core into the LEON3 design in the `config.vhd` file.

Secondly, the Interface entity includes the configuration information for this purpose, since this entity will interface the core with the APB bus. This is done in Excerpt 3:

```
entity apb_ac97_if is
generic (  pindex     :     integer := 10;
           paddr      :     integer := 10;
           pmask      :     integer := 16#FFF#;
           vendorid   : in  integer := 16#01#;
           deviceid   : in  integer := 16#08C#;
           version    : in  integer := 0);
end;

architecture apb_ac97_if_arch of apb_ac97_if is
constant pconfig:apb_config_type:=
(0=>ahb_device_reg(VENDOR_GAISLER,GAISLER_AC97,0,0,0),
 1=>apb_iobar(paddr, pmask));
end;
```

Excerpt 3: Generics declaration and configuration for the GRLIB.

6.3.8. AC97 Top Module

This is the highest level design's entity and where the whole of it was enclosed. In this way, all the lower level entities are bounded into this convenient unit.

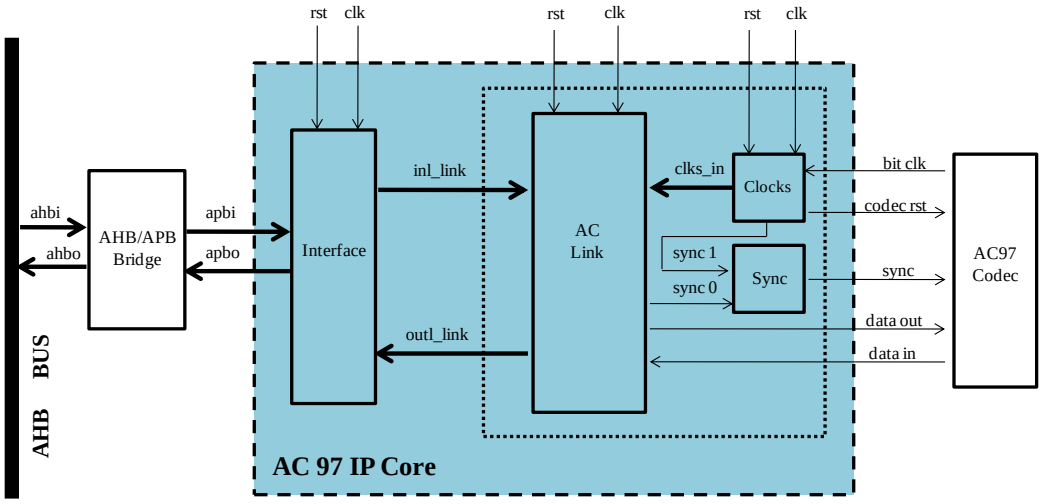


Figure 17: AC97 Top Module.

The Top Module design had to be instantiated in the `leon3mp.vhd` file so it could be encompassed into the whole embedded system. The way this was done is illustrated in the Excerpt 4.

```
ac97: if (CFG_AC97_ENABLE = 1) generate
ac97: ac97top generic map
          (pindex=>10,paddr=>10,pmask=>16#FFF#)
          port map (clk => clkm, rst => rstn,
          ac97top_inl              => apbi,
          ac97top_inr.data_in      => audio_sdata_in,
          ac97top_inr.bit_clk      => audio_bit_clk,
          ac97top_outl             => apbo(10),
          ac97top_outr.sync        => audio_sync,
          ac97top_outr.data_out    => audio_sdata_out,
          ac97top_outr.reset_codec => flash_audio_reset_b);
end generate ac97;
```

Excerpt 4: AC97 (1st stage of the design) component instantiation in the `leon3mp.vhd` file.

6.4. Second Stage

The second stage of the design is explained in here. First, the DMA to AHB in Figure 18 is explained, followed by a description of the DMA Controller (DMAC in Figure 18); there is a higher level entity which includes the previous two entities and it is named as DMA Engine (light blue square around DMAC and DMA to AHB in Figure 18), which is detailed after the DMAC. Some modifications had to be made to the AC Link entity, to the Interface entity, and to the Top Module. All of these are included in this section as well.

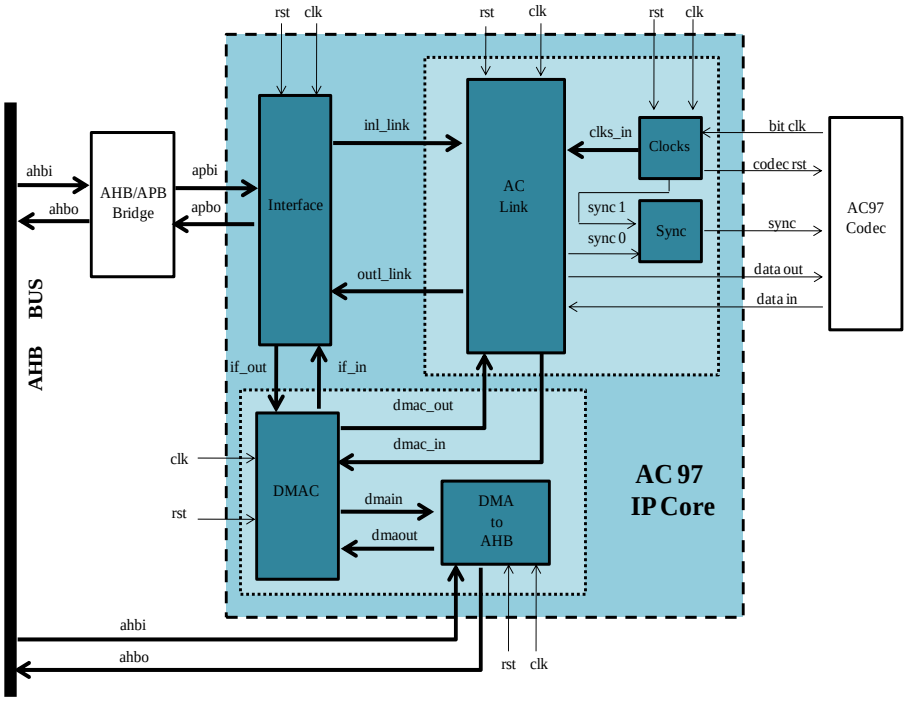


Figure 18: AC97 IP core at its second stage.

6.4.1. The DMA to AHB Entity

The DMA to AHB is an entity that was already designed at Gaisler Research. Hence, it was reused and included in the A97 core (in Figure 18, it is highlighted with dark blue and named as DMA to AHB). It gives communication between the AHB bus and the core itself, by acting as a master interface to the AHB bus. The DMA to AHB entity requests access to the bus. Subsequently, the bus gives the address to the entity (the entity starts fetching data from this address).

Figure 18 depicts the DMA to AHB in dark blue and how it is connected to the DMA Controller (DMAC in Figure 18) and to the AHB bus. It has four record types: *ahb_mst_in_type* and *ahb_mst_out_type* (*ahbi* and *ahbo* respectively in Figure 18), which are used to achieve communication with the AHB bus; *dma_in_type* and *dma_out_type* are used to connect with the DMAC (*dmain* and *dmaout* respectively in Figure 18).

Table 17 explains the signals in the record type *ahb_mst_in_type*, used to achieve communication between the AHB bus and the DMA to AHB entity.

Table 17: *ahb_mst_in_type* signals [14].

Signal Name	Function
hgrant	This signal indicates that the bus master is currently the highest priority master. A master gets access to the bus when both *hready* and *hgrant* are high.
hresp	The transfer response provides additional information about the transfer status, such as: *okay*, *error*, *retry*, and *split*.
hready	The data can be extended using this signal. When low, indicates that the transfer is to be extended and when high indicates that the transfer can complete. The slaves must only sample the address and control signals when this signal is high.
hrdata [31:0]	This is the read data bus, which is used to transfer data from bus slaves to the bus master during reading operations.

Table 18 explains the signals in the record type *ahb_mst_out_type*, used to achieve communication between the AHB bus and the DMA to AHB entity.

Table 18: *ahb_mst_out_type* signals [14].

Signal Name	Function
hbusreq	It is a signal from the bus master to the bus arbiter, which indicates that the bus master requires the bus.
hlock	When set to one, indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is low.
htrans [1:0]	Indicates the type of the current transfer (*nonsequential*, *sequential*, *idle*, or *busy*).
haddr [31:0]	The 32-bit system address bus.
hwrite	When high, it indicates a write transfer and when low a read transfer.
hsize [2:0]	Indicates the size of the transfer. It could a byte (8 bits), a half word (16 bits), or a word (32 bits).
hburst [2:0]	Indicates if the transfer forms part of a burst. Four, eight, and sixteen beat bursts are supported and the burst may be either incrementing or wrapping.
hindex	Indicates the master bus index.

Table 19 names the signals and their function in the record type *dma_in_type*, which is used to communicate between the DMA to AHB and the DMAC entities.

Table 19: *dma_in_type* signals for the AC97 core [14].

Signal Name	Function
reset	When asserted, makes the data transfer to be idle. When de-asserted, a data transfer can be started.
address [31:0]	Indicates from which address the data should be requested.
data [31:0]	Sent data to the bus.
request	Requests access to the AHB bus.
burst	Requests a burst transfer from the AHB bus.
beat [1:0]	Determines how many beats should be in the transfer.
size [1:0]	Indicates the size of the transfer. It could a byte (8 bits), a half word (16 bits), or a word (32 bits).
store	When set to zero is used for read operations. When set to one is set to write operations.
lock	When set to one, indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is low.

Table 20 names the signals and their function in the record type *dma_out_type*, which is used to communicate between the DMA to AHB and the DMAC entities.

Table 20: *dma_out_type* signals for the AC97 core [14].

Signal Name	Function
grant	This signal indicates that the bus master is currently the highest priority master.
okay	Indicates that the transfer is progressing normally.
ready	The data can be extended using this signal. When low, indicates that the transfer is to be extended and when high indicates that the transfer can complete. The slaves must only sample the address and control signals when this signal is high.
retry	Shows the transfer has not yet completed, so the bus master should retry the transfer.
fault	This signal shows an error has occurred.
data [31:0]	The data to be transferred from the AHB bus.

6.4.2. The DMA Controller

Figure 18 shows where the DMA Controller is placed in the design. It is highlighted with dark blue in the figure.

The DMA Controller (DMAC) uses the DMA to AHB entity, explained previously in Section 6.3.1. In order to use the latter entity, the GRLIB library and other packages had to be added in the `dmac.vhd` file (refer to Appendix A.4).

The DMAC manages the DMA to AHB entity with the state machine in Figure 19, and it also communicates with the AC Link and the Interface entities. The DMAC is necessary to start fetching data from memory, so these data could be passed to the AC Link through its output channel and then streamed from the core to the CODEC.

Right after the reset, the state machine stays in the state S0 (refer to Figure 19) until it receives the signal (from the Interface) which indicates when it should start fetching data from memory. When this start signal is asserted, it goes to S1.

The state machine stays in S1 if the grant signal (coming from the DMA to AHB entity) is 0. Otherwise, it will go to S2. In state S1, the base address from where the data should be fetched is obtained in the first burst transfer. If it is not the first burst transfer, the address is got from where the previous burst stopped.

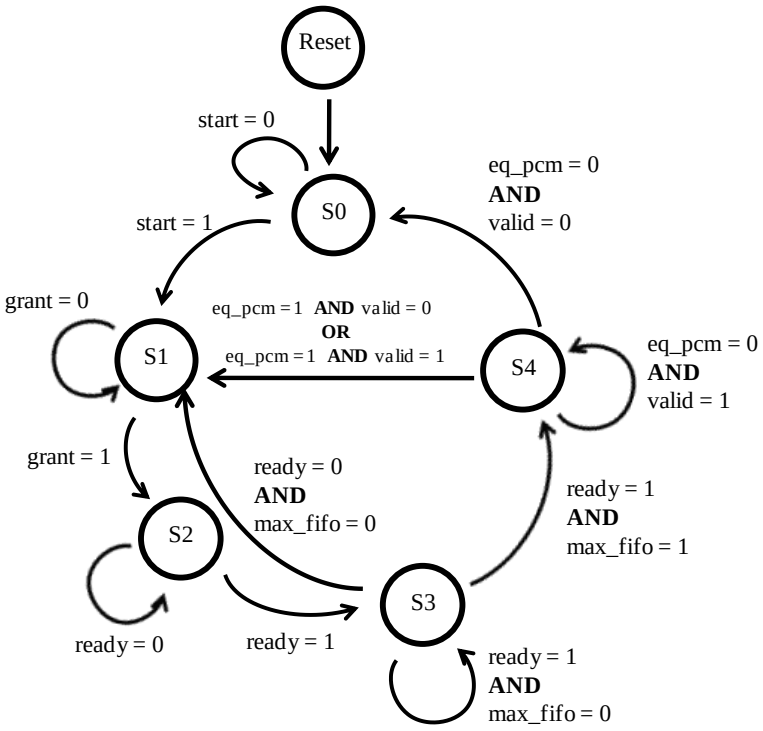


Figure 19: DMAC finite state machine.

Similarly, in S1 the *burst*, *request*, and *lock* signals (refer to Table 19) are asserted, *store* is set to zero, *size* is set to *hsize32* (word sized transfers), and *beat* is set to *hincr4* to request 4-beat incremental bursts. It was decided to request such sized bursts because the AC Link does not have a very high speed. Thus, the DMAC do not need very deep FIFO because the data will be transferred at slow rate (compared to the fetching rate). In this way, the FIFO size could be reduced and still the DMAC could get advantage of the burst transfers.

Likewise, S1 is in charge of asserting an acknowledgment signal that indicates that the start signal from the Interface has been received. Thus, this signal is sent by the DMAC and received by the Interface, causing the signal *start* to be de-asserted (preparing the state machine for the next initial burst transfer).

The state machine stays in S2 if the *ready* signal (refer to Table 20) is 0; else, it will advance to S3. If it remains in S2, the validity of the data is checked. This is done if the data is different than 0x0. If so, the *valid* signal (refer to Figure 19) is asserted; it will mean the memory has no more valid data in it. This verification is done only the first time the state machine is in S2, because it is possible that it goes back again to S1 and from there jump to S2. Hence, checking if the data is valid is not needed anymore.

Some other operations are performed in S2. One of those is to save the first incoming data into the FIFO. Otherwise, it will be lost if this is done in the next state. After storing the data, the counter that points to the FIFO index is incremented by one. In the same fashion, the *reset* signal (refer to Table 19) is de-asserted so the burst transfer can be initiated subsequently.

When the state machine is in S3, the rest of the data is stored in the FIFO. If the FIFO's counter has reached its maximum count (0x7), the *max_fifo* signal (refer to Figure 19) is asserted, data is saved, the base address is increased by 0x10 (since it is a 4-beat burst transfer) so the DMAC starts fetching data from the last accessed address instead of getting it from the initial address, and the FIFO's counter is no further increased. If the FIFO's counter has any other count besides its maximum, the data is still saved and the counter is increased by one.

Once in S3 there are three choices: to stay in S3 or to get out of this state with two possible directions (either S4 or S1). The state machine will be locked in S3 if *ready* is 1 and *max_fifo* is 0 (refer to Figure 19). To get out of this state and move on to S4, *ready* has to be one and *max_fifo* has to be one also. The last choice is to go to S1 if *ready* is 0 and *max_fifo* is 0.

The last state of the machine is S4. Just like S3, it has three possible combinations. Either to stay in S4, go back to S1, or complete the cycle by going back to S0. If it happens that the state machine stays in S4 is because *eq_pcm* is 0 and *valid* is 1 (refer to Figure 19). If the machine jumps from S4 to S1, *eq_pcm* is 1 and *valid* is 0, or *eq_pcm* is 1 and *valid* is 1. The loop is completed when *eq_pcm* is 0 and *valid* is 0. If this is the case, the state machine goes back to S0 and is ready to empty the memory again.

6.4.3. DMAC Status Flags

During the DMAC's state machine operation, status flags are sent to the Interface entity and these are transmitted to the APB bus through *pirq* (explained in Table 16) in *apbi* (refer to Figure 18) to generate interruptions.

Table 21 describes the DMAC status flags, when are they asserted/de-asserted, and their place in the *pirq* line.

Table 21: DMAC status flags description.

Status flag	**Asserted**	**De-asserted**	**Bit in *pirq***
busy_dma	S1, S2, S3	S0, S4	2
wait_dma	S0	S1	1
done_dma	S4	S0, S1, S2, S3	0

The first status flag (*busy_dma* in Table 21) is used to indicate when the DMAC's state machine is fetching data from memory. The status flag *wait_dma* announces that the state machine is in the idle state, waiting for an order from the processor to start fetching data. The last status flag in Table 21 is *done_dma*, which expresses that the state machine has finished the data burst transfer. The status flags were arbitrarily placed in bits 2 – 0 respectively in the *pirq* line, since these bits were not currently in use.

6.4.4. Data Burst Transfer Type

As explained in Section 6.4.2, 4-beat incrementing bursts are performed by the DMA to AHB master. In this way, the core can fetch data from memory in an efficient way. Figure 20 shows the timing diagram of this kind of transfer.

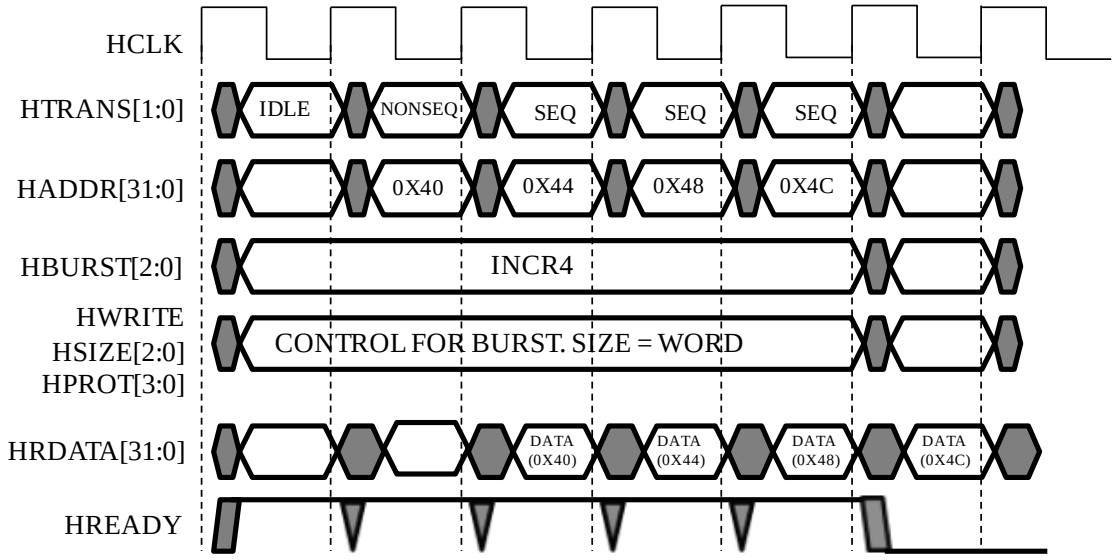


Figure 20: Four-beat incrementing burst.

The transfer starts with an idle state (refer to Figure 20). Afterwards, the transfer changes to a non-sequential type indicating the start of the burst and the remaining three transfers are of the sequential type. Following the last sequential transfer, the ready signal goes low indicating the termination of the burst. Since the burst uses word (32 bits) transfers, the addresses increase by four. It is important to note that the data will be available on the bus one clock cycle after the address has been requested.

6.4.5. The DMA Engine

The DMA engine is a higher level entity that contains the DMAC and the DMA to AHB, so all the DMA related operations are contained in one single entity and the design can be more portable and compact. The DMA Engine entity is highlighted in light blue and it is showed in Figure 18.

This entity has three input types and three output types, besides the clock and reset inputs. With these types, the engine can communicate with the other entities in the core and the AHB bus. The *ahb_mst_in_type* and *ahb_mst_out_type* are used for the AHB bus; the *pcm_in_type* and *pcm_out_type* are used with the AC Link entity; the *ac97if_in_type* and *ac97if_out_type* are used with the Interface entity. Additionally, to these types the engine has internal signals so the DMAC can be connected with the DMA to AHB entity, which are *dma_in_type* and *dma_out_type*.

Table 22 explains the function of the signals in the *pcm_in_type*, which goes out of the DMAC and goes into the AC Link entity.

Table 22: *pcm_in_type* signals.

Signal Name	Function
valid	The DMAC sends this signal to the AC Link when the read data from memory is valid. If so, the slots are loaded accordingly, the synchronization signal can be sent, and the data can be streamed out as well.
done_link	This signal is sent to the AC Link by the DMAC to reset the frame's counter (keeps track on how many frames have been sent) and sets low the signal that indicates when the number of sent out frames and written data are equal, if *done_link* is asserted.
waiting	The DMAC send this signal to the AC Link when the DMAC's state machine is in the state zero, waiting for the Interface to tell it to start fetching data.
pcm [31:0]	The PCM data is sent from the DMAC to the AC Link in this channel.

Table 23 explains the function of the signals in the *pcm_out_type*, which is an output of the AC Link entity and is an input of the DMAC entity.

Table 23: *pcm_out_type* signals.

Signal Name	Function
eq_pcm	This signal is asserted in the AC Link when the number of sent out frames to the CODEC has reached half the maximum depth of the FIFO so the DMAC can fetch more data.
frame [2:0]	This signal is sent by the AC Link and received by the DMAC to know which data in the FIFO should be sent to the AC Link.

Table 24 explains the function of the signals in the *ac97if_in_type*, which connects the outputs of the DMAC entity with the inputs of the Interface entity.

Table 24: *ac97if_in_type* signals.

Signal Name	Function
done_link	When this signal is asserted in the DMAC, the signal that indicates the start of the data fetch is de-asserted in the Interface entity.
busy_dma	This signal indicates when the DMAC's state machine is fetching data from memory.
wait_dma	This signal indicates that the state machine is in the idle state, waiting for an order from the processor to start fetching data.
done_dma	This signal indicates that the state machine has finished the data burst transfer.

Table 25 explains the function of the signals in the *ac97if_out_type*, which goes out of the Interface entity to the input of the DMAC entity.

Table 25: *ac97if_out_type* signals.

Signal Name	Function
start	This signal is asserted in the interface entity when the DMA engine should start fetching data from memory and it also de-asserted in the interface entity when the done_link signal (which goes from the DMAC to the interface) is asserted.
base [31:0]	This vector is sent by the interface to the DMAC and it is used as the base address, from where the data will start to be fetched from memory by the DMA engine.

6.4.6. The Modified AC Link Entity

The next step after completing the DMA Engine design was to modify accordingly the entities designed in the first stage, aiming to correctly interface all the entities in the design. For this purposes, new input and output ports were introduced to the AC Link and to the Interface entities.

Two new ports were included in the AC Link entity. The input port receiving the signals coming from the DMA Engine is called *dmac_in* and the output port sending the signals to the DMAC is called *dmac_out* (refer to Figure 18). The input uses the record type *pcm_in_type*, whilst the output uses the record type *pcm_out_type.*

Besides these changes in the ports, the hardware in the AC Link was modified. The changes are surrounded with dashed lines in Figure 21. All in all, seven changes were required for the correct functionality of the design. Going from left to right in Figure 21, the red dashed rectangles represent the first change. When the slots are charged with their corresponding bits (refer to Section 6.3.2.1), the signal called *pcm_ack* in Figure 21 is asserted if PCM data has to be flushed out of the core. When PCM data is not present in the slots, *pcm_ack* is set to zero.

The next modification in the design (dashed with a blue conditional in Figure 21), was done to decide when the synchronization signal could to be sent. The condition to do so, involves the commands, the PCM data, and the cold reset. Regarding the commands, the acknowledgment signal between the Interface and the AC Link has to be asserted; also, the number of written commands into the Interface and the number of sent out frames to the CODEC has to be different (i.e. there are still commands to be streamed out).

Concerning the PCM data, the signal that indicates if the data is valid has to be asserted and the DMAC's state machine has to be in a different state than the waiting state (i.e. already fetching data from memory). One or the other of those two previous conditions has to be true along with a third one, which requires that the cold reset to the CODEC has to be asserted so it can receive frames from the core.

A third change in the design is illustrated with a green dashed line in Figure 21. In there, it is decided when to send the data. The conditions from the first stage of the design remained, but one extra requirement was added for this action to be true: that the *pcm_ack* signal is asserted and the DMAC's state machine is not in the waiting state.

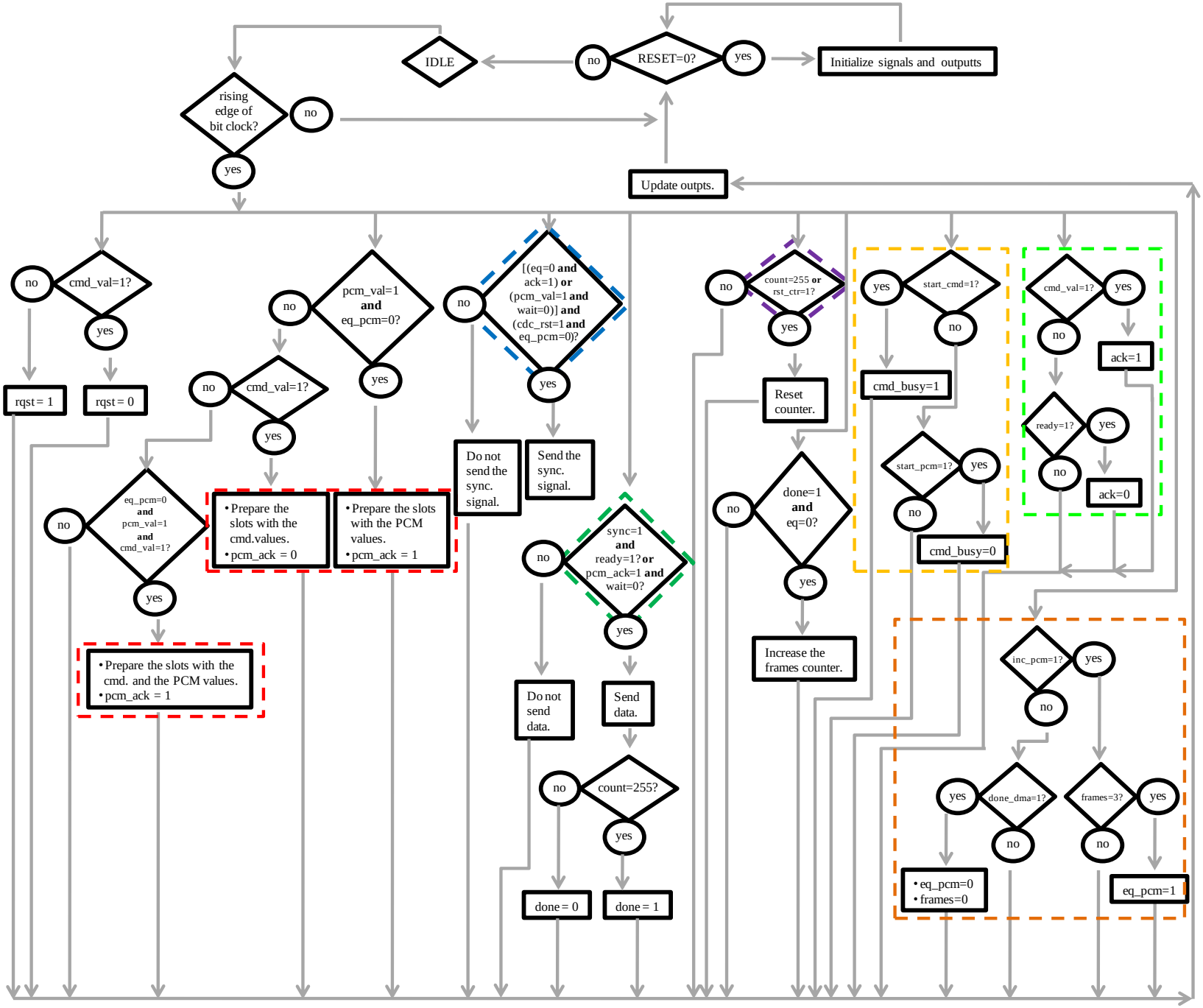


Figure 21: Modified AC Link entity (refer to Figure 13 for the original AC Link flow chart).

Referring to Figure 21, the purple conditional is a new feature in the design. In its first stage, it was decided to reset the counter of the bits in the frames when it had reached 255. However, in this second stage the counter was set back to zero if it count 255 or if *rst_ctr* is 1.

The *rst_ctr* signal is asserted by calling a function that checks when no commands have to send to the CODEC, the DMAC's state machine is the waiting state, and when the synchronization signal is 0. If these three conditions are not checked, the count could start erroneously and the frames could be sent incongruently.

In the modified AC Link entity, there was a need to include a signal (refer to *cmd_busy* inside the dashed yellow rectangle in Figure 21) that indicated when either commands or PCM data (sending PCM data also involves sending these data along with a command) had to be transmitted to the CODEC. If a command is being sent, then *start_cmd* (refer to Figure 21) should be one and *cmd_busy* is asserted. If PCM data is being sent, then *start_pcm* should be one and *cmd_busy* is set to zero.

To check if *start_cmd* was one or zero, a function was called to do so. The function was named *start_cmd* and it checked if the fifth bit of the frame is zero. If that is true, a command was being sent without PCM data and *start_cmd* was asserted; else, it was set to zero.

In the same fashion, to check if *start_pcm* was one or zero, a function was called to do so. The function was named *start_pcm* and it checked if the fifth bit of the frame was one. If that was true, the frame included PCM data and *start_pcm* was asserted; else, it was set to zero.

In Figure 21, there is a green dashed rectangle. It introduces a different way of asserting the acknowledgment signal in the AC Link. If there is a valid command coming from the Interface entity, *ack* is asserted. Whenever the command stops being valid and the ready signal is asserted in the Interface (meaning that a frame could be transmitted to the CODEC), *ack* will be set to zero.

The last modification to the AC Link entity can be visualized in the orange dashed rectangle on the right bottom corner in Figure 21. This new condition is used in the design to stop the DMAC's state machine by de-asserting the signal *eq_pcm*, which indicates that the number of streamed out frames equals the number of written data in the FIFO.

In order to de-assert the *eq_pcm* signal (please refer to Figure 21) and stop the DMAC's state machine, two conditions have to be fulfilled. The first one, is when the signal *inc_pcm* is not one, which is checked by calling the function *inc_pcm* that indicates when the counter of the frames containing PCM data should be increased by one. This function in turn, checks if the signal *cmd_busy* (which is inside the yellow dashed rectangle of Figure 21 and was already explained) is set to zero, indicating that a frame containing PCM data has been transmitted. If so, then it checks when the counter of bits in the frame is not 255 and thus, the signal *inc_pcm* will be set to zero.

The other condition to be met after the signal *inc_pcm* has been checked that is 0, so the DMAC's state machine can be halted, is that the signal *done_dma* is equal to one

(shown in Figure 21). This signal comes from the *done_link* output of the DMAC (refer to Table 24). When *done_dma* equals one, the signal *eq_pcm* can be set to zero and the DMAC's state machine can be driven to its waiting state, ready to receive another order to fetch more data. Along with this de-assertion, the counter of the frames containing PCM data should be set to zero, so it can be ready for another transmission.

Continuing with the orange rectangle, another case remains unexplained. In other words, when the signal *inc_pcm* is 1. Whenever this happens, the hardware checks if the PCM frames counter is equal to three (i.e. half the depth of the FIFO). If this is true, the signal *eq_pcm* will be asserted, pointing out that the DMAC's state machine should keep on working.

6.4.7. The Modified Interface Entity

Just like in the modified AC Link entity, two new ports were included in the Interface entity. The input port receiving the signals coming from the DMA Engine is called *if_in* and the output port sending the signals to the DMAC is called *if_out* (refer to Figure 18). The input uses the record type *ac97if_in_type*, but the output uses the record type *ac97if_out_type.*

The interface was slightly modified from the first stage of the design. One extra multiplexor was added to the hardware at the same level as the four previous ones. It is the upper most conditional marked with a dashed red line in Figure 22.

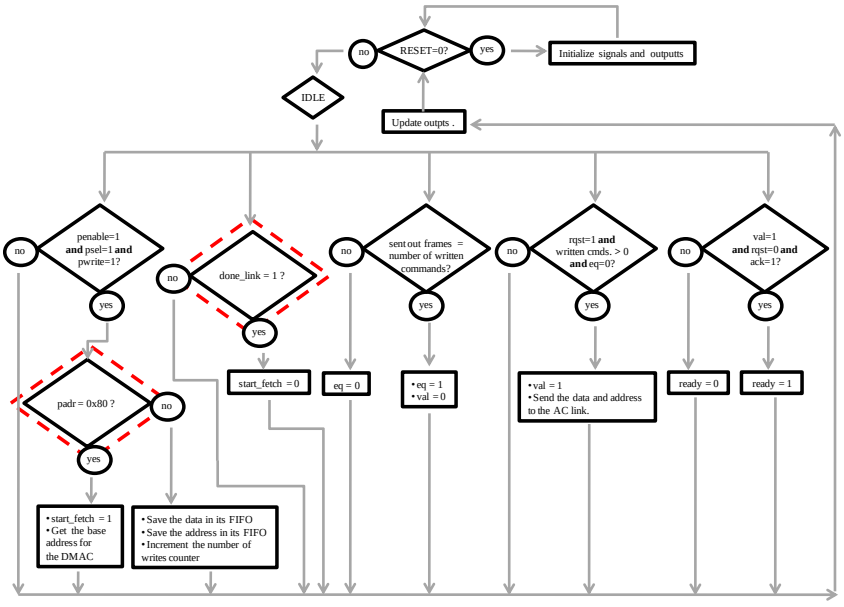


Figure 22: Modified Interface entity (refer to Figure 16 for the original Interface flow chart).

The aforementioned multiplexor checks if the signal that indicates the start of the data fetch should be de-asserted. This is performed by checking if the *done_link* signal (refer to Table 24) is 1. If this is not the case, the hardware does nothing.

The lower conditional marked with a dashed red line in Figure 22, representing another added multiplexor, was included inside the statement that checks if the core is selected, enabled, and if something can be written to it. What this new multiplexor performs is to

assert the *start_fetch* signal (which is then sent to the DMAC) and get the base address (which is also sent to the DMAC). This is done by checking if *padr* is 0x80. If so, *start_fetch* is asserted and the word in *pwdata* is used as the base address for the DMAC (further used as the start-fetching-command); else the data and address are saved in the corresponding FIFO.

Besides these modifications, the Interface entity also receives three status flags from the DMAC's state machine, which indicate when it is busy, when it is waiting, and when it is done. All these three inputs are then transmitted to the APB bus through the *pirq* line when the AC97 core is read. The rest of the hardware works in the same way as in the first stage of the design (refer to Section 6.3.5).

6.4.8. The Modified Top Module

The Top Module of the design had to be modified, since a new entity was introduced (i.e. the DMA Engine). However, since the DMA Engine is an entity that contains both the DMAC and the DMA to AHB, it was very easy to include the DMA Engine into the design's top module. Only two more ports had to be added to it. Namely, *ahbi* and *ahbo* (refer to Figure 18), which use the record types *ahb_mst_in_type* and *ahb_mst_out_type*, respectively.

The modified top module was instantiated in the LEON-3 based system as shown in Excerpt 5.

```
ac97: if (CFG_AC97_ENABLE = 1) generate
ac97: ac97top generic map (pindex=>10, paddr=>10, pmask=>16#FFF#)
port map
            (clk             => clkm,
             rst             => rstn,
             data_in         => AUDIO_SDATA_IN,
             bit_clk         => AUDIO_BIT_CLK,
             ac97top_inl     => apbi,
             ac97top_outl    => apbo(10),
             ac97dma_in      => ahbmi,
             ac97dma_out     => ahbmo(5),
             sync            => AUDIO_SYNC,
              data_out        => AUDIO_SDATA_OUT,
             codec_rst       => FLASH_AUDIO_RESET_B);
end generate ac97;
```

Excerpt 5: AC97 (2nd stage of the design) component instantiation in the `leon3mp.vhd` file.

In Excerpt 5, it can be observed that the *ac97dma_out* port core has assigned in the master bus index (i.e. *hindex*) the value 5. Thus, in the `leon3mp.vhd` file, the DDR2 memory's *hindex* was changed to 0x5, so there could be a communication between the AHB master (the AC97 core) and the AHB slave (the DDR2 memory).

This was done because the AC97 core uses the record type *ahb_mst_out_type* and the indexes 0x0 to 0x4 were already taken. Hence, the AC97 core had to use the next index (i.e. 0x5) to access the DDR2 memory.

7. Verification

Just like in the design process, the verification was divided in the same two parts: the first one is the AC Link, its correct timing module, and the Interface between the APB Bridge and the previous two entities; the second one, is the DMA, its FIFO, the correct interface with the AHB bus, and the Interface with the first stage of the design.

7.1. First Stage

This section describes the first stage of the verification process that basically consists of three steps: a stand-alone test bench, a system test bench, and post-synthesis verification. The Section is subdivided into four other Sections: first, there is an explanation of the stand-alone test bench (first step in the verification process), where the first stage of the design was simulated and verified with ModelSim; afterwards, there is a description of the system test bench (second step in the verification), in which the first stage of the design was simulated along with the whole LEON3-based system; following the system test bench is an explanation of the synthesis procedure of the first stage of the design; subsequently, there is presentation on how the first stage of the design was verified after being synthesized and programmed into the FPGA (third and last step in the verification process); at the end is a discussion of the encountered problems during post-synthesis verification and how they were solved.

7.1.1. Stand-Alone Test Bench in ModelSim

The first step in verifying the initial stage of the design, was to try out the AC97 core in a stand-alone test bench. Thus, on the transcript window was checked if the core was correctly displayed (see Excerpt 6).

```
# apbctrl: slv10: Gaisler Research AC97 Controller
# apbctrl: I/O ports at 0x80000A00, size 256 byte
```

Excerpt 6: System test transcript with the introduced AC97 controller (1st stage of the design).

To obtain the content of Excerpt 6, the AC97 core had to be correctly instantiated in the test bench. This is shown in Excerpt 7.

```
inst0 : ac97top
port map(clk              => clk,
         rst              => rstn,
         bit_clk          => bit_clk,
         ac97top_inl      => apbi,
         data_in          => data_in,
         codec_rst        => codec_rst,
         ac97top_outl     => apbo(pindex),
         sync             => sync,
         data_out         => data_out);
```

Excerpt 7: AC97 (1st stage of the design) component instantiation in the stand-alone test bench.

It is important to emphasize that the controller should appear as the APB slave 10 at the address 0x80000A00, so it will not overlap with any other slave in the system.

The stand-alone test bench was provided by the Gaisler staff. However, additional tests and functions (e.g. functions to read different size test vector files and to check whenever a test vector file is empty) were introduced to it to verify the core's basic functionality (refer to Appendix A.8 for the full VHDL code).

The test bench consists of seven tests. The first test wrote commands into the core and were stored in it. The address and data of these commands were increased one by one. In this way, the address 0x0 had 0x0, the address 0x1 had 0x1, etc. The second test read the values previously written in the core. The third test wrote a specific command and it was streamed out to the CODEC. The frame was checked to verify if the sent out frame was equal as the expected test vector. The next four tests did the same thing as the third test, although with different addresses and values. To run the simulation, a do file containing the commands in Excerpt 8 was executed:

```
restart -f
view signals wave
add wave *
force bit_clk 0
force bit_clk 1 17, 0 57 -repeat 80
run -all
```

Excerpt 8: Do file to execute the stand-alone test bench.

The Bit Clock starts in 17 ns just to make it asynchronous to the system clock and verify that synchronization is achieved between the two clock domains. It repeats itself every 80 ns to get 12.5 MHz, which is the closest to the required 12.288 MHz.

Figure 23 shows the wave form when the entire test was successfully passed:

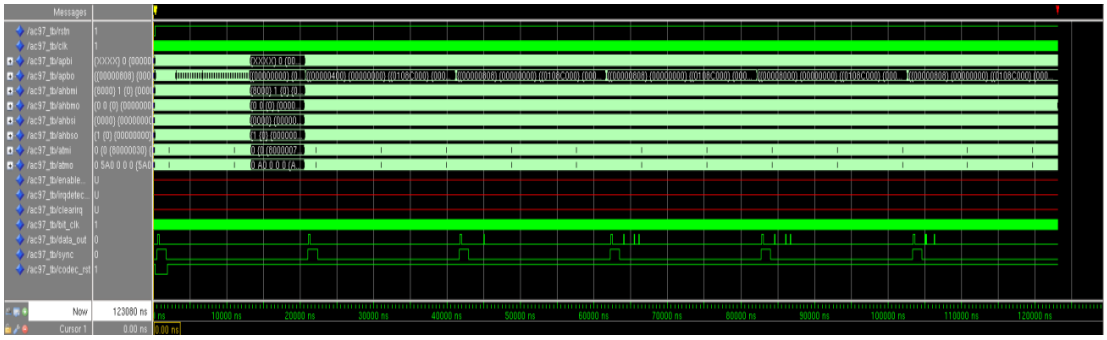


Figure 23: Successful stand-alone test bench simulation.

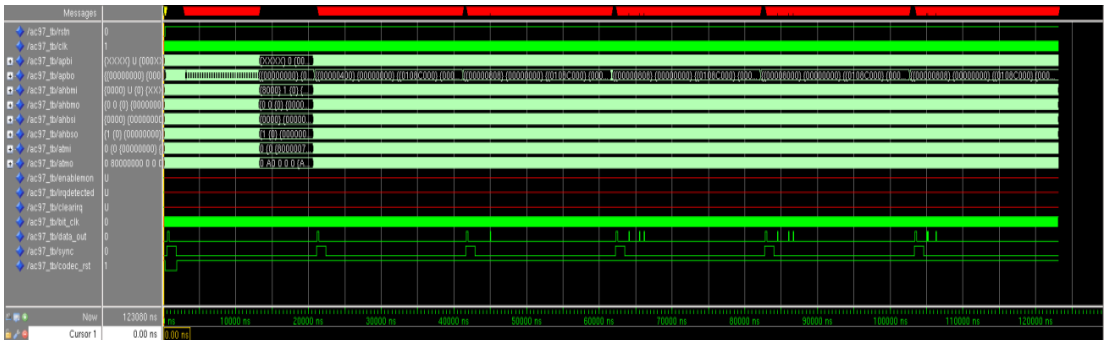


Figure 24: Intentionally unsuccessful stand-alone test bench simulation.

When the test bench was successfully verified, the tests were intentionally modified to make the test bench crash by loading wrong test vectors into it. In Figure 24, the red arrows on top of the wave forms indicate that the results are wrong; in Figure 23 there are no red arrows on top of the wave forms, which indicate that the results are correct.

7.1.2. System Test Bench in ModelSim.

The second step in verifying the initial stage of the design was to run a system test in ModelSim. Hence, the AC97 top module design was instantiated in the `testbench.vhd` file so it could be included in the whole system simulation.

```
cpu : entity work.leon3mp
port map (audio_bit_clk,audio_sdata_in,audio_sdata_out,audio_sync,
flash_audio_reset_b);
```

Excerpt 9: AC97 (1st stage of the design) component instantiation in the `testbench.vhd` file.

In order to perform the system test, a program was written in C language (please refer to Appendix A.10 for the program's code) to verify the core's functionality. The program wrote into the main volume register and the PC beep CODEC register to turn on the beep and modify the volume. The routines can be summarized in the following list:

- **a)** Turn on the main volume.
- **b)** Turn on the PC beep.
- **c)** Gradually turn down the volume in sixteen steps until the minimum volume is reached in the right side and in the left side at the same time.
- **d)** Gradually turn up the volume in sixteen steps until the maximum volume is reached in the right side and in the left side at the same time.
- **e)** Sweep the beep frequencies from 94 Hz to 12 kHz.
- **f)** Mute the main volume.

To run the simulation, a couple of commands (see Excerpt 10) had to be written in the ModelSim environment to get the Bit Clock that goes to the AC97 core and to watch the wave forms:

```
do wave_ac97_1st.do
force audio_bit_clk 1 17, 0 57 -repeat 80
run –all
```

Excerpt 10: Commands to execute the system test bench.

The do file in Excerpt 10 was slightly modified. The AC97 signals were added to it, so they could be seen during the simulation. Excerpt 11 shows how they were added to the do file.

```
add wave -noupdate -format Logic /testbench/audio_bit_clk
add wave -noupdate -format Logic /testbench/audio_sdata_in
add wave -noupdate -format Logic /testbench/audio_sdata_out
add wave -noupdate -format Logic /testbench/audio_sync
add wave -noupdate -format Logic /testbench/flash_audio_reset_b
```

Excerpt 11: Added AC97 signals to the system test bench do file.

With this test, the stereo capability of the CODEC should be verified and it should be controlled by the core. It was decided to use the PC beep to get sound out of the CODEC since the DMA was not introduced yet. The beep itself was generated inside the CODEC. The controller only wrote into the correct register but did not send audio data whatsoever.

7.1.3. Synthesis

Once the results were verified in simulation, synthesis was performed in batch mode using the Synplify tool and the `make synplify` command. Right after the netlist was generated, place&route was performed by issuing the `make ise-synp` command.

All the necessary VHDL files had to be added to the `Makefile` and the `leon3mp.ucf` file was modified as shown in Excerpt 12 to include the AC97 core signals during the synthesis.

```
NET AUDIO_BIT_CLK                LOC = "AF18";
NET AUDIO_SDATA_IN               LOC = "AE18";
NET AUDIO_SDATA_OUT              LOC = "AG16";
NET AUDIO_SYNC                   LOC = "AF19";
NET FLASH_AUDIO_RESET_B          LOC = "AG17";

NET AUDIO_BIT_CLK                LOC="AF18"|IOSTANDARD=LVCMOS33;
NET AUDIO_SDATA_IN               LOC="AE18"|IOSTANDARD=LVCMOS33;
NET AUDIO_SDATA_OUT              LOC="AG16"|IOSTANDARD=LVCMOS33;
NET AUDIO_SYNC                   LOC="AF19"|IOSTANDARD=LVCMOS33;
NET FLASH_AUDIO_RESET_B          LOC="AG17"|IOSTANDARD=LVCMOS33;
```

Excerpt 12: Modified `leon3mp.ucf` file.

7.1.4. Post-Synthesis Verification

The AC97 core design was downloaded to the Virtex 5 FPGA and verified in the Xilinx ML505 prototype board using the CODEC that is already integrated into it. Its vendor is Analog Devices and the model is AD1981 Audio CODEC, which supports stereo 16-bit audio with up to 48 kHz sampling frequency. The ML505 prototype board has several audio jacks: microphone, in and out analog lines, headphone, and Sony Phillips Digital Interface Format (SPDIF) [19]. For verification purposes, only the headphone output jack was used in this thesis (refer to Figure 25).

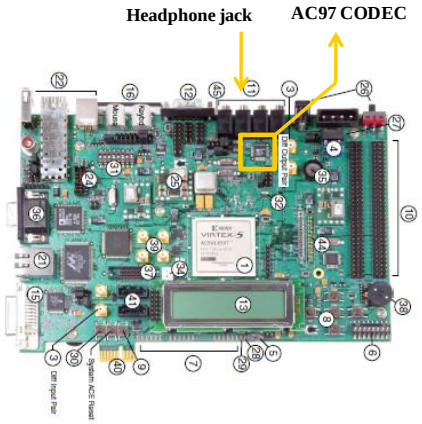


Figure 25: ML505 Xilinx prototype board. Courtesy of Xilinx [19].

Once the design was downloaded to the prototype board using the command `make ise-prog-fpga`, it was verified with the same C test program as a third step for the verification using GRMON[5]. To access such software tool, the command `grmon -xilusb -u` was issued in the batch mode and to download the C test program to the FPGA, the command `load systest.exe` was used, followed by `verify systest.exe` to check if the program was correctly downloaded.

The PC beep could actually be heard since a pair of loudspeakers was connected to the headphone audio jack as shown in Figure 25. Besides listening to the PC beep, the signal was also observed directly on the prototype board using the Agilent Technologies Mixed Signal Oscilloscope (MSO6054A).

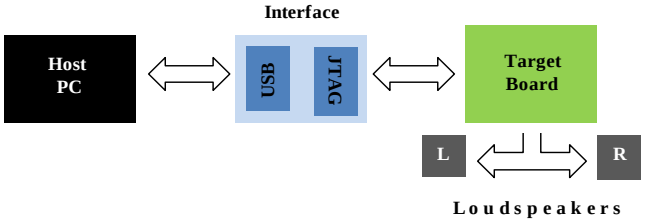


Figure 26: Connection between GRMON and the target board.

Figure 26 illustrates how the AC97 core was debugged and monitored using GRMON. From the host personal computer (PC), basic read and write memory commands were written and sent to the core. Similarly, C programs were written and downloaded into the board's DDR2 memory and then run with the purpose of a more thorough and automatic core verification. This tool was launched directly from the Linux terminal (i.e. in batch mode). No graphical user interface (GUI) was used.

7.1.5. Encountered and Solved Problems

A number of problems were found during the post-synthesis verification, which were not evident in simulation. The problems that caused troubles in hardware were two. The first obstacle was that in order to propagate the commands (given by the host PC via the software tool GRMON) from the core to the CODEC, they had to be typed twice in the batch mode. The second one was that while running programs in the processor that wrote into the core repetitively, the controller stalled all of a sudden while the program was still running. This problem was encountered later in time. Due to these facts, a revision of the design had to be made.

While reviewing the design with the aim to eliminate the first error, it was discovered that the signal (located in the Interface entity) which allowed the AC Link to stream out frames, was not controlled with the right timing. In other words, this enabling signal was set high after the synchronization signal had been sent out and set low after the data had been streamed over the link. So, on the first time only the synchronization signal was considered without taking into account the data; at the second time, once the enabling signal had been asserted, it was possible to send out the data along with the synchronization signal. This mistake was corrected by considering in the enabling

[5] GRMON is a software tool to debug and monitor LEON3 based embedded systems on real target hardware.

condition (in the Interface entity) an acknowledgment signal and the synchronization signal, both coming from the AC Link. After doing so, it was possible to execute the commands on the first trial.

Nonetheless, after solving this problem it was evident that there was a need to store the incoming commands somehow. Otherwise, they would be overwritten and some of them would be lost. To avoid losing information, FIFOs were introduced to save the incoming data and address. By using this storing mechanism, extra control between the AC Link and the Interface entities is implicit. Hence, this control logic was included in such a way that the incoming commands were stored into these FIFOs and when a frame was outputted, a counter that pointed to the FIFOs index was increased until the number of written commands would be equal to the number of streamed out frames. By doing so, it was certain that no commands would be missed or overwritten.

Heretofore, the first encountered problem after the synthesis was solved (i.e. having to type the commands twice so they could propagate all the way to the CODEC) by sending the output data at the right time and introducing FIFOs to store the commands. Notwithstanding, while running C programs on the FPGA, the core was not working properly.

After exhausting debugging, it was found that the second problem relied on the wrong synchronization between different clock domains (the core's clock and the CODEC's clock). Whilst executing a loaded program, the synchronization between them was lost forever and it could only be reestablished by re-programming the FPGA.

The way in which the synchronization was achieved in the first place was by sampling the Bit Clock with the system clock, passing it through two flip-flops, and detect the rising edges and falling edges of the sampled signal, so a new clock could be obtained. The rising edge of this newly created clock was used in the sequential process so the registers in the core could be updated. However, when verifying the design in hardware it turned out that this implementation caused the synchronization between the CODEC and the core to be lost.

After experiencing the problems that arose when using a different clock in the sequential process, it was considered to use another solution for a proper synchronization. It was observed that only the outputs to the CODEC have to be driven by the Bit Clock, while there was not a need for doing so in the sequential process. For this reason, as a second choice for the synchronization, it was decided to use the obtained rising edge by the Bit Clock (please refer to Section 6.3.1 to read through it in detail about how it was implemented) as a trigger for the AC Link outputs and use the system clock for the sequential process.

7.2. Second Stage

This section describes the second stage of the verification process that basically consists of two steps: a stand-alone test bench and a system test bench. Synthesis was completed but post-synthesis verification was not feasible (in same way it was done during the first stage).

This Section is divided in the following way: first there is a description of the stand-alone test bench, in which the second stage of the design was simulated and verified with ModelSim; the next Section explains the system test bench, where the second stage of the design was simulated along with the whole LEON3-based system; the following Section explains the synthesis procedure of the second stage of the design; the last Section introduces how the second stage of the design could be verified after being synthesized and programmed into the FPGA.

7.2.1. Stand-Alone Test Bench in ModelSim

For this purpose, the stand-alone test bench used during the first stage of the design was re-utilized but modified according to the new needs. For instance, an AHB memory was instantiated in the test bench file as it can be observed in Excerpt 13.

```
ahbslv0 : at_ahb_slv
generic map (hindex            => 1,
             bank0addr         => 16#400#,
             bank0mask         => 16#FFF#,
             bank0type         => AT_AHBSLV_MEM,
             bank0cache        => 1,
             bank0prefetch     => 1,
             bank0ws           => 1,
             bank0rws          => AT_AHBSLV_FIXED_WS,
             bank0dataload     => 0,
             bank0datafile     => "none")
port map (rstn=>rstn,clk=>clk,ahbsi=>ahbsi,ahbso=>ahbso(1),dbgi=>dbgi,
          dbgo=>dbgo);
```

Excerpt 13: AHB memory instantiation in the stand-alone test bench.

With the AHB memory in the stand-alone test bench, some values can be written into it and the AC97 core can start to fetch them using the DMA Engine. The AC97 controller was instantiated in the stand-alone test bench as shown in Excerpt 14.

```
inst0 : ac97top
port map(clk              => clk,
         rst              => rstn,
         bit_clk          => bit_clk,
         ac97top_inl      => apbi,
         ac97top_outl     => apbo(pindex),
         ac97dma_in       => ahbmi,
         ac97dma_out      => ahbmo(1),
         ac97dmac_inm     => atmo,
         debug            => deb,
         data_in          => data_in,
         codec_rst        => codec_rst,
         sync             => sync,
         data_out         => data_out);
```

Excerpt 14: AC97 (2nd stage of the design) component instantiation in the stand-alone test bench.

Once the memory and the core were included in the stand-alone test bench, it was possible to run tests. To do so, the same do file as in Excerpt 8 was run (i.e. same do file as in the first stage of the design).

It was decided that three tests should be run in this test bench. The first one wrote into some CODEC's registers. Afterwards, raw data was written into the memory, starting from address 0x40000000, since the AHB memory was placed in this area. To fetch the written data, the address 0x80 was written into the core with the value 0x40000000 in its data. This was done because, as explained in Section 6.4.5, this particular address is recognized in the AC97 core as a start-fetching-command, and takes the value in the read data as the base address to start fetching data from memory. The previous two tests were repeated one more time with the intention of verifying that the core could start all over again after it had completed his first task.

In Figure 27, the wave forms of these tests in the stand-alone test bench are shown.

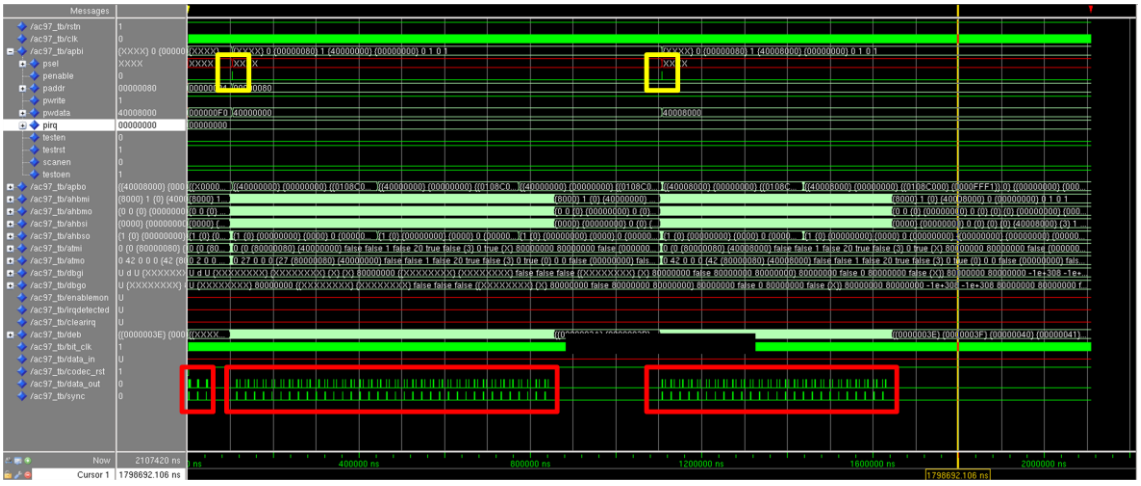


Figure 27: Wave form of the second stage stand-alone test bench.

The first red rectangle from left to right in Figure 27 shows the commands which are being sent out to the CODEC. The second red rectangle illustrates the first burst of PCM data which was first fetched from memory and then sent to the CODEC. The third red rectangle in the figure shows another burst of PCM data. On top of the figure, surrounded by yellow squares, it can be observed that the start-fetching-command is being issued to the AC97 core through the APB bus, just before the stream of PCM data was transmitted.

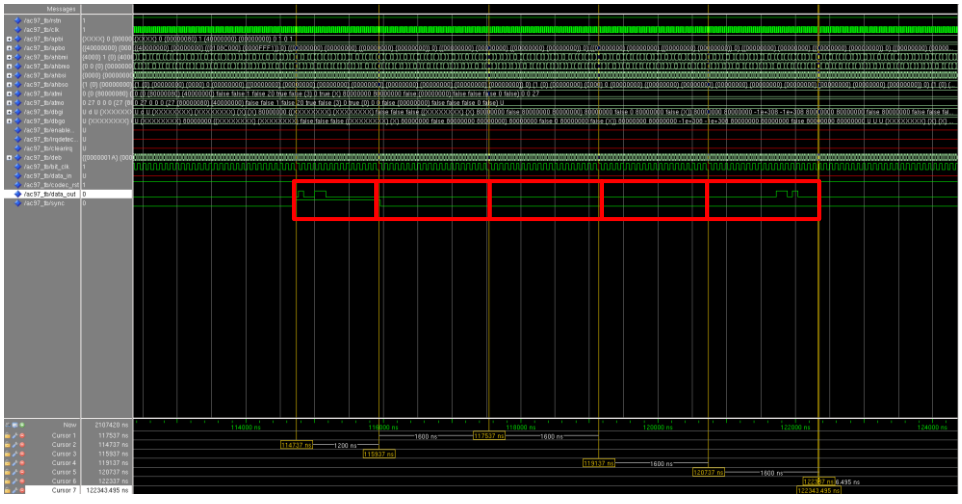


Figure 28: Zoom in to a specific frame of Figure 27.

Figure 28 shows a zoom in to a specific value in the stream of data in the previous tests.

Easily recognizable data was written into the memory, so it could be quickly observed in the wave form if it was correctly transmitted or not. As it can be seen in Figure 28, it has five red rectangles. Each one of them represent different slots of the frame (from slot 0 up to slot 4, from left to right). In the first rectangle (slot 0) it can be noted that has a logic one, followed by two logic zeros, and then a wider logic one. This means that it has two consecutive ones. Thus, bit 0 in slot 0 is 1 (meaning that the frame is valid) and bits 3 and 4 are valid (slots 3 and 4, where the PCM data is, are valid).

The second, third, and fourth red rectangles are full of zeros, but the fourth one is valid (according to bit 3 in slot 0). The last one contains logic ones in it. The value that was written in memory and that now is sent to the CODEC is the hexadecimal value 0x001A (11010_2), which is displayed on the last red rectangle of Figure 28. As a matter of fact, the slot does not end with 0x001A; instead, it does end with 0x001A0, which makes sense because the CODEC receives 20-bit sized slots (refer to Tables 5 and 6). If the slot is 16-bit, the rest of the bits are filled with zeros.

7.2.2. System Test Bench in ModelSim

Up to this point in the verification process, the design of the AC97 core was verified in a stand-alone test bench. Hereinafter, it was possible to move forward with the system test bench. The core was instantiated in the same way as it was done in Excerpt 11, but a new C code program was written for this test bench.

The C program basically did the same thing as in the stand-alone test bench. However, an array was filled in with raw data up to certain number of indexes in the array, and these values were transferred to the DDR2 memory in the system (please refer to Appendix A.11 for the complete C code). Subsequently, the start-fetching-command (refer to Section 6.4.5) was written into the core, by assigning the address 0x80 to a pointer and then getting the data from the array's starting address.

Similarly as in the stand-alone test bench, in the system test bench the wave form of the simulation was observed (Figure 29).

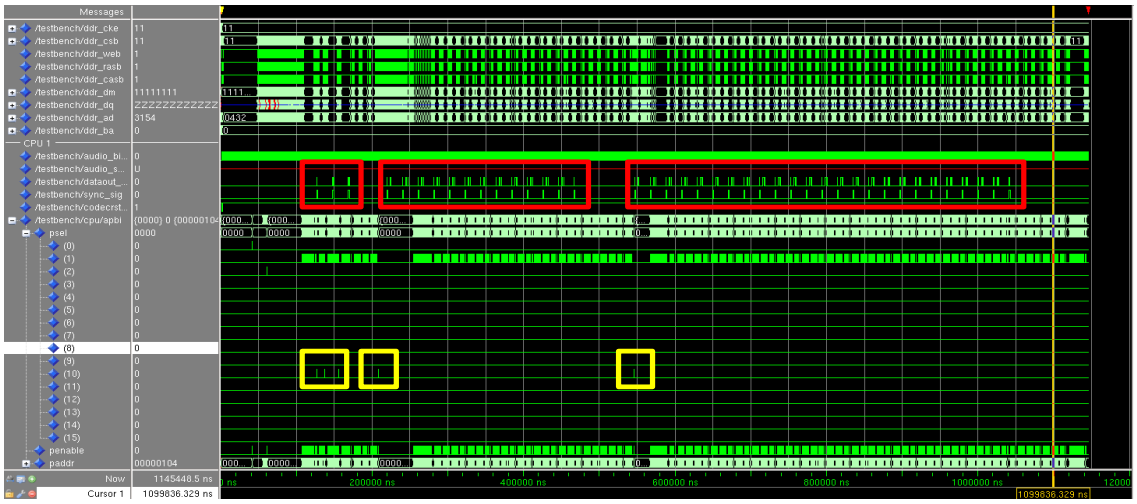


Figure 29: Wave form of the second stage system test bench.

The first red rectangle from left to right in Figure 29 shows the commands which are being sent out to the CODEC. The first yellow square in the same figure illustrates when these commands are being written into the core. The second red rectangle shows the first burst of PCM data which was first fetched from the DDR2 memory and then streamed out to the CODEC. The third red rectangle in the figure, shows another burst of PCM data. Furthermore, it can be observed below these red rectangles, three other yellow squares. The first one shows when the commands are being written into the Interface entity through the APB bus. The following two yellow squares show when the start-fetching-command is being issued to the AC97 core, just before the stream of PCM data was transmitted to the CODEC.

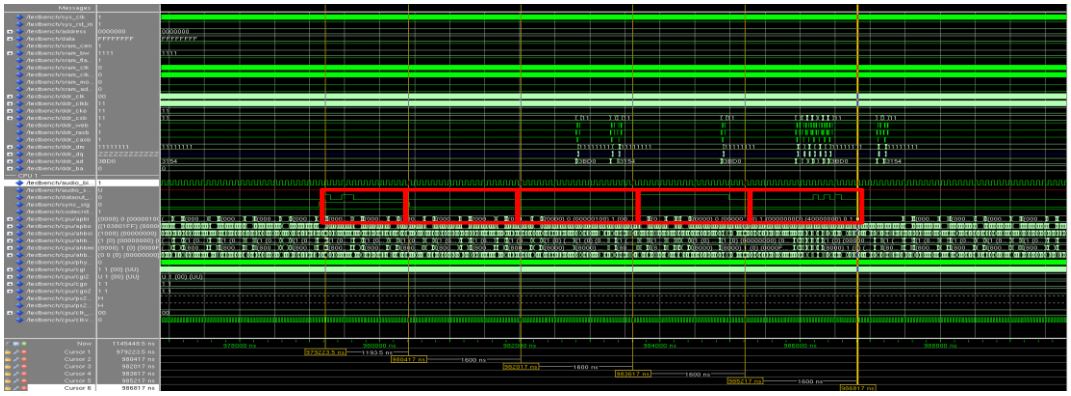


Figure 30: Zoom in to a specific frame of Figure 29.

Same as in the stand-alone test bench, easily recognizable data was written into the memory so it could be quickly observed in the wave form. By looking at Figure 30, it has five red rectangles and each one of them represent different slots of the frame.

The fourth rectangle in Figure 30 has a logic one much wider than the others. This is because the data that is being transmitted is 0xFFFF and it goes low at the end of the frame because it contains only 16 bits and not 20 bits (if this is the case, the last four bits are zero and the transmitted value is 0xFFFF0).

The last red rectangle in Figure 30, contains logic ones in it. The value that was written in memory and that now is sent to the CODEC is the hexadecimal value 0x0015 (10101_2), which is displayed on the last red rectangle of Figure 30. For the same reason with the slot 3, the red rectangle ends with 0x00150.

7.2.3. Synthesis

The second stage of the design was correctly synthesized with ISE. For this purpose, the command `make ise` was used in batch mode. Synplify was not used at this stage because it was being utilized by too many users at the same time or there was a license checkout.

7.2.4. Post-Synthesis Verification

Post-synthesis verification of the second stage of the design was not carried out, since the time to debug the design was not feasible. However, hereafter is a description of what has to be done to proceed with this verification.

Firstly, a file that contains PCM data has to be obtained. Certain types of *.wav* files do contain such kind of modulation scheme. Secondly, the .wav file has to be converted into an understandable representation for the memory in the prototype board (a *srecord*, for instance). After doing so, the *srecord* file has to be separated, in order to write the data into the memory. This is possible with a C program. The final step is to download the data to the memory using GRMON and then, fetch the data from memory with the AC97 core (using GRMON commands as well) and stream it out to the CODEC.

To convert *.wav* files containing PCM data into *srecord* files (i.e. the bytes of binary data are encoded as a 2-character hexadecimal number. The first character represents the high-order 4 bits and the second one represents the low order 4 bits of the byte [20]) that can be downloaded into the FPGA, the command showed in Excerpt 15 was used in batch mode.

```
sparc-elf-objcopy -I binary –O srec --change address 0x40000000
~/wav_file.wav test
```

Excerpt 15: Command to convert into *srecord* files.

The first part of the command (`sparc-elf-objcopy`) is used to create a *srecord* file; the second part of it (`-I binary`) means that it receives a binary file and it creates a *srecord* file as an output (`-O srec`). `--change address` literally means that the *srecord* file is downloaded into the specified address (`0x40000000`); the location of the file and its name, along with its extension, appears in `~/wav_file.wav`; at the end of the command (`test`) the name of the converted *srecord* file is given.

An example of a srecord file has the representation shown in Excerpt 16.

```
S3 15 40000000 524946465A5F030057415645666D7420 2D
```

Excerpt 16: Example of *srecord* file representation.

Excerpt 16 has been modified so the srecord can be illustrated clearly. After a file has been converted to a srecord file, the data has no blank spaces in between. It was done only for illustration purposes. The first two digits represent the type of record; the next two characters tell the record length (in this case the record is 15 bytes long); the following 8 characters represent the address (which is then increased by 0x10 to save new data); the data is placed in the next 32 characters; the last 2 address are the checksum, which is the least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the record length, address, and the code/data fields [20].

Once the *srecord* file was obtained, it had to be transferred to the memory in the prototype board. Since the *srecord* contains mixed data in each line (i.e. type, length, address, data, and checksum all mixed in one row), it had to be separated somehow so the memory can store the useful information. Hence, the data was compiled in a C program and then passed to the memory through GRMON. Nonetheless, the design could not be verified in the prototype board.

8. Discussion

In this section, a discussion on the changes to the original time plan is presented.

The original time plan was made with the idea that all the design should be verified first in simulation (i.e. in the ModelSim suite) before verifying it on hardware. At some point of the design, it was possible to download the core into the FPGA and achieve communication with the processor, since the Interface with the APB bus was already implemented. An engineer at Gaisler suggested me that it would be a good idea to try the design on hardware before adding more complexity to it.

It felt that the right choice was to not follow the original time plan, create a new one so it could fit into the new working context, and go for the hardware verification of the first stage of the design instead of doing everything first in simulation. In this way, the core's state could be verified and bugs were removed from it.

By taking this choice, more time was spent testing the first stage of the design in the prototype board even though the time budget was limited and less time would be spent verifying the second stage of the design. This choice caused delays in the new time plan but at the end, the core and the CODEC were correctly interfaced in the first stage of the design.

9. Conclusion

The main purpose of the project is to develop an AC97 IP core and implement it in VHDL using the Two Process Design Methodology. The core should read and write to the AC97 CODEC registers, read and write accesses to the sound channel FIFOs, include a DMA engine to fill and empty the FIFOs without CPU use, and generate interrupts on various events.

In the first stage of the project, it was possible to verify that the core was communicating with the rest of the LEON3-based system and the CODEC. Nonetheless, it was not possible to affirm it for the second stage of the design.

During the verification of the first stage of the design the AC97 core was able to write into the AC97 CODEC and through GRMON it was possible to read the core's registers. Furthermore, it was possible to listen to sounds coming out of the CODEC (i.e. sweeping different frequencies of the CODEC's PC beep) and to control the volume of the loudspeakers through the AC97 core.

In the second stage of the design, a DMA engine was included to fill and empty a FIFO without CPU use and the status of the DMAC's state machine was sent over the interrupt line. Unfortunately, the second stage of the design was not verified in hardware using the prototype board and no real audio could be played on it. This was due to the fact that the time budget for this thesis was limited. A lot of time was invested in correctly interfacing the core to the CODEC and getting rid of the problems explained in Section 7.1.5. It was not feasible to wait until the completion of the whole system's verification to write the thesis, since the remaining amount of time had to be spent either verifying the second stage of the design or writing the report. Certainly, the verification in hardware of the second stage of the design has to be left out as further work in the project.

By means of assessing the thesis objectives and the obtained outcomes, the design of the AC97 IP core was partially verified. The first and second stages of the design were verified in simulation but still the second stage of the design has to be thoroughly verified in hardware by playing real audio on the prototype board.

10. Further work

The continuation of the present project would include a thorough verification of the design. This means to download the second stage of the design into the prototype board's FPGA and follow the given recommendations in Section 7.2.4. The AC97 core also has recording capabilities. Therefore, it would be interesting to develop and include this functionality in the present AC97 core so it could be a full-duplex design, and be able to record sounds coming from the outside world.

11. References

[1] Brock, D. Understanding Moore's Law Four Decades of Innovation. *Chemical Heritage Foundation.* 2006.

[2] Bohr, M. Silicon Technology for 32 nm and Beyond System-on-Chip Products. *Intel Developer Forum.* 2009. Available at URL: http://download.intel.com/technology/architecture-silicon/32nm/IDF_Fall_09.pdf Accessed April 27, 2011.

[3] Gaisler.com − Available at URL: http://www.gaisler.com Accessed April 27, 2011.

[4] Audio Codec '97 Revision 2.3 Revision 1.0. *Intel*. April, 2002.

[5] Gaisler, J. Master's Thesis Proposal: Implementing an AC97 audio controller IP. *Aeroflex Gaisler AB*. December 5, 2010.

[6] LEON3 Multiprocessing CPU Core. *Aeroflex Gaisler*. 2010.

[7] GRLIB IP Library User's Manual. Version 1.1.0 B4100. *Gaisler Research*. October 1, 2010.

[8] Texas A&M Universtiy. The AC97 CODEC. *ECEN 449 - Microprocessor System Design.*

[9] Globalspec.com − About IP Cores. Available at URL: http://www.globalspec.com/LearnMore/Industrial_Computers_Embedded_Computer_Components/Industrial_Computing/IP_Cores Accessed April 23, 2011.

[10] Whatis.techtarget.com − IP core (intellectual property core). Available at URL: http://whatis.techtarget.com/definition/0,,sid9_gci759036,00.html Accessed April 23, 2011.

[11] Checkpoint 2 AC97 Audio. *University of California at Berkeley College of Engineering Department of Electrical Engineering and Computer Science.* EECS150 Spring 2005.

[12] LM4549A AC'97 Rev 2.1 Multi-Channel Audio Codec with Sample Rate Conversion and National 3D Sound. *National Semiconductor Corporation.* May 2004.

[13] LM4540 AC'97 Codec with National 3D Sound. *National Semiconductor Corporation*. February 1999.

[14] AMBA Specification revision 2.0, 1999. *ARM Limited.*

[15] Gaisler, J. Fault-tolerant Microprocessors for Space Applications. *Gaisler Research.*

[16] Gaisler.com − Management. Available at: http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=117&Itemid=38 Accessed May 21, 2011.

[17] Hellqvist, M. Implementation of a Linux Workstation Based on the LEON Processor. *Chalmers University of Technology.* 2005.

[18] Hildebrandt, R. The pseudo dual-edge d-flip-flop. Available at URL: http://www.ralf-hildebrandt.de/publication/pdf_dff/pde_dff.pdf Accessed April 26, 2011.

[19] ML505/ML506/ML507 Evaluation Platform User Guide UG347 v3.1.1. *Xilinx.* October 7, 2009.

[20] Appendix A S-Record Format. Application note: 68EVB912B32UM/D *Motorola*..

[21] GRMON Debug Monitor for Leon. *Aeroflex Gaisler*. 2010.

[22] GRMON User's Manual Version 1.1.49. *Aeroflex Gaisler AB.* April 2011.

[23] Ashenden P.J. & Lewis, J. The Designer's Guide to VHDL, Vol. 3, Third Ed. *Morgan Kauffmann.* 2008.

A. Appendix
A.1. AC Link VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;

-------------------------
-- package delcaration
-------------------------
package ac97link_pack is

-----------------------------
-- inputs from clks
-----------------------------
type clks_in_type is record
        r_clk     : std_logic;
        f_clk     : std_logic;
        codec_rst : std_logic;
end record;

-----------------------------------------------------
-- inputs from interface
-----------------------------------------------------
type aclink_inl_type is record
        ready_if        : std_logic;
        valid_if        : std_logic;
        equal           : std_logic;
        start_fetch     : std_logic;
        done_dma        : std_logic;
        adres_if        : std_logic_vector(31 downto 0);
        data_if         : std_logic_vector(31 downto 0);
end record;

-----------------------------------------------------
-- inputs from dmac
-----------------------------------------------------
type pcm_in_type is record
        valid     : std_logic;
        done_link : std_logic;
        waiting   : std_logic;
        pcm       : std_logic_vector(31 downto 0);
end record;

-----------------------------------------------
-- outputs to interface
-----------------------------------------------
type aclink_outl_type is record
        cmd_rqst : std_logic;
        start    : std_logic;
        ack      : std_logic;
        donecopy : std_logic_vector(7 downto 0);
end record;

-----------------------------------------------
-- outputs to dmac
-----------------------------------------------
type pcm_out_type is record
        eq_pcm  : std_logic;
        frame   : std_logic_vector(2 downto 0);
end record;
```

```
-----------------------------------------------------
-- component declaration
-----------------------------------------------------
component ac97link_comp
        port (clk       : in  std_logic;
              rst       : in  std_logic;
              clks_in   : in  clks_in_type;
              inl_link  : in  aclink_inl_type;
              data_in   : in  std_logic;
              in_pcm    : in  pcm_in_type;
              out_pcm   : out pcm_out_type;
              outl_link : out aclink_outl_type;
              data_out  : out std_logic;
              sync      : out std_logic);
end component;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.ac97link_pack.all;

-----------------------------------------
-- entity
-----------------------------------------
entity ac97link is

port (clk       : in  std_logic;
      rst       : in  std_logic;
      clks_in   : in  clks_in_type;
      inl_link  : in  aclink_inl_type;
      data_in   : in  std_logic;
      in_pcm    : in  pcm_in_type;
      out_pcm   : out pcm_out_type;
      outl_link : out aclink_outl_type;
      data_out  : out std_logic;
      sync      : out std_logic);
end;

-------------------------------------------------
-- architecture
-------------------------------------------------
architecture ac97link_arch of ac97link is

-------------------------------------------------
--records
-------------------------------------------------
type reg_type is record
        pcm                     : std_logic_vector(31 downto 0);
        data_sig                : std_logic_vector(31 downto 0);
        adres_sig               : std_logic_vector(31 downto 0);
        comb_sync, seq_sync     : std_logic_vector(8 downto 0);
        comb_done, seq_done     : std_logic_vector(7 downto 0);
        frames_c, frames_s      : std_logic_vector(2 downto 0);
        cmd_val_sig             : std_logic;
        pcm_val                 : std_logic;
        sync                    : std_logic;
        sync2                   : std_logic;
        latch2                  : std_logic;
        ready                   : std_logic;
```

```
        ack                     : std_logic;
        done                    : std_logic;
        equal                   : std_logic;
        trigger                 : std_logic;
        start_pcm               : std_logic;
        start_cmd               : std_logic;
        pcm_ack                 : std_logic;
        dataout                 : std_logic;
        cmd_busy                : std_logic;
        inc_pcm                 : std_logic;
        codec_rst               : std_logic;
        eq_pcm                  : std_logic;
        done_dma                : std_logic;
        start_fetch             : std_logic;
        waiting                 : std_logic;
        rst_ctr                 : std_logic;
end record;

type slots is record
        slot_0  : std_logic_vector(0 to 31);
        slot_1  : std_logic_vector(0 to 31);
        slot_2  : std_logic_vector(0 to 31);
        slot_3  : std_logic_vector(0 to 31);
        slot_4  : std_logic_vector(0 to 31);
        slot_5  : std_logic_vector(0 to 31);
        slot_6  : std_logic_vector(0 to 31);
        slot_7  : std_logic_vector(0 to 31);
        slot_8  : std_logic_vector(0 to 31);
        slot_9  : std_logic_vector(0 to 31);
        slot_10 : std_logic_vector(0 to 31);
        slot_11 : std_logic_vector(0 to 31);
        slot_12 : std_logic_vector(0 to 31);
end record;

signal signals  : reg_type;
signal slotsrec : slots;

---------------------------------
-- function to shift the slots
---------------------------------
function slot_shift(countfun:reg_type;
                    slotsfun:slots)return std_logic is
variable slotsvar   : slots := slotsfun;
variable shiftslot  : std_logic_vector(0 to 31);
variable outbit     : std_logic;
variable times:integer:=conv_integer(unsigned(countfun.comb_sync-2));
begin
if(countfun.comb_sync < 16)then
shiftslot:=to_stdlogicvector(to_bitvector(slotsvar.slot_0)sll(times));

elsif(countfun.comb_sync>=16 and countfun.comb_sync<=35)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_1)sll(times-16));

elsif(countfun.comb_sync>=36 and countfun.comb_sync<=55)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_2)sll(times-36));

elsif(countfun.comb_sync>=56 and countfun.comb_sync<=75)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_3)sll(times-56));
```

```
elsif(countfun.comb_sync>=76 and countfun.comb_sync<=95)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_4)sll(times-76));

elsif(countfun.comb_sync>=96 and countfun.comb_sync<=115)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_5)sll(times-96));

elsif(countfun.comb_sync>=116 and countfun.comb_sync<=135)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_6)sll(times-116));

elsif(countfun.comb_sync>=136 and countfun.comb_sync<=155)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_7)sll(times-136));

elsif(countfun.comb_sync>=156 and countfun.comb_sync<=175)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_8)sll(times-156));

elsif(countfun.comb_sync>=176 and countfun.comb_sync<=195)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_9)sll(times-176));

elsif(countfun.comb_sync>=196 and countfun.comb_sync<=215)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_10)sll(times-196));

elsif(countfun.comb_sync>=216 and countfun.comb_sync<=235)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_11)sll(times-216));

elsif(countfun.comb_sync>=236 and countfun.comb_sync<=255)then
shiftslot:=
to_stdlogicvector(to_bitvector(slotsvar.slot_12)sll(times-236));

elsif (countfun.comb_sync > 255)  then shiftslot(0):= '0';
end if;

outbit := shiftslot(0);
return outbit;
end slot_shift;

-----------------------------------------------------
-- ack function
-----------------------------------------------------
function start (countfun : reg_type) return std_logic is
        variable strt : std_logic;
        begin
          if   (countfun.comb_sync = 0) then strt := '1';
          else                               strt := '0';
          end if;
        return strt;
end start;
```

```
-------------------------------------------------------
-- function to indicate when is done
-------------------------------------------------------
function done (countfun : reg_type) return std_logic is
        variable fin : std_logic;
        begin
        if(countfun.cmd_busy='1' and
           countfun.eq_pcm='0' and
           countfun.equal='0') then
           if (countfun.comb_sync = 255) then fin := '1';
           else fin := '0';
           end if;
        else fin := '0';
        end if;
        return fin;
end done;

-------------------------------------------------------
-- function to indicate when a pcm has been sent
-------------------------------------------------------
function start_pcm (countfun : reg_type) return std_logic is
        variable flag : std_logic;
        begin
          case countfun.seq_sync is
           when "000000110" =>
             if (countfun.dataout = '1') then flag := '1';
             else flag := '0';
             end if;
           when others => flag := '0';
          end case;
        return flag;
end start_pcm;

-------------------------------------------------------------
-- function to indicate when to increase the pcm counter
-------------------------------------------------------------
function inc_pcm (countfun : reg_type) return std_logic is
        variable flag : std_logic;
        begin
          if (countfun.cmd_busy = '0') then
            if (countfun.seq_sync = "011111111") then flag := '1';
            else flag := '0';
            end if;
          else flag := '0';
          end if;
        return flag;
end inc_pcm;

-------------------------------------------------------
-- function to indicate when a cmd has been sent
-------------------------------------------------------
function start_cmd (countfun : reg_type) return std_logic is
        variable flag : std_logic;
        begin
          case countfun.seq_sync is
           when "000000110"=>if(countfun.dataout='0')then flag:='1';
                             else flag := '0'; end if;
           when others => flag := '0';
          end case;
        return flag;
end start_cmd;
```

```
-------------------------------------------------------
-- function to increase the pcm counter
-------------------------------------------------------
function frame_count (countfun : reg_type) return std_logic_vector is
variable frames : std_logic_vector(2 downto 0);
variable var    : reg_type := countfun;
begin
 if(countfun.inc_pcm='1')then var.frames_c:=countfun.frames_c+1;
                                frames:= var.frames_c;
 elsif (countfun.done_dma = '1') then frames := (others => '0');
 else frames := countfun.frames_c;
 end if;
return frames;
end frame_count;

-------------------------------------------------------
-- function to increase the cmd counter
-------------------------------------------------------
function cmd_count (countfun : reg_type) return std_logic_vector is
        variable cmds : std_logic_vector(7 downto 0);
        variable var  : reg_type := countfun;
        begin
        if(countfun.done='1')then var.comb_done:=countfun.comb_done+1;
                                  cmds:=var.comb_done;
        else cmds := countfun.comb_done;
        end if;
        return cmds;
end cmd_count;

-------------------------------------------------------
-- function to latch data out
-------------------------------------------------------
function latch2 (countfun : reg_type) return std_logic is
        variable latchfun : std_logic;
        begin
        if  (countfun.seq_sync <= 255) then latchfun := '1';
        else                               latchfun := '0';
        end if;
        return latchfun;
end latch2;

-------------------------------------------------------
-- function to latch  sync signal
-------------------------------------------------------
function latch1 (countfun : reg_type) return std_logic is
        variable latchfun : std_logic;
        variable fin      : std_logic := '1';
        begin
        if  (countfun.seq_sync < 16) then latchfun := '1';
        else                              latchfun := '0';
        end if;
        return latchfun;
end latch1;
```

```
-------------------------------------------------------
--  function to reset the counter
-------------------------------------------------------
function rst_ctr (countfun : reg_type) return std_logic is
        variable flag : std_logic;
        begin
 if(countfun.cmd_val_sig='0' and countfun.waiting='1' and
   countfun.sync='0')then flag := '1';
 else flag := '0';
 end if;
 return flag;
end rst_ctr;

-------------------------------------------------------
-- notx function
-------------------------------------------------------
function notx(d : std_logic_vector) return boolean is
        variable res : boolean;
        begin
        res := true;
        -- pragma translate_off
        res := not is_x(d);
        -- pragma translate_on
        return (res);
end;

begin
-------------------------------------------------------------------
-- combinational process
-------------------------------------------------------------------
combinational : process(rst, signals, slotsrec)
 variable v   : reg_type;
 begin
  if (rising_edge(signals.trigger)) then

  ---------------------------------------------------------------------
  -- slot preparation
  ---------------------------------------------------------------------
  if (signals.pcm_val = '1' and signals.eq_pcm = '0') then
          slotsrec.slot_0   <= x"98000000";
          slotsrec.slot_1   <= (others => '0');
          slotsrec.slot_2   <= (others => '0');
          slotsrec.slot_3   <= signals.pcm(31 downto 16) & x"0000";
          slotsrec.slot_4   <= signals.pcm(15 downto 0)  & x"0000";
          slotsrec.slot_5   <= (others => '0');
          slotsrec.slot_6   <= (others => '0');
          slotsrec.slot_7   <= (others => '0');
          slotsrec.slot_8   <= (others => '0');
          slotsrec.slot_9   <= (others => '0');
          slotsrec.slot_10  <= (others => '0');
          slotsrec.slot_11  <= (others => '0');
          slotsrec.slot_12  <= (others => '0');
          signals.pcm_ack   <= '1';

  elsif (signals.cmd_val_sig = '1') then
        slotsrec.slot_0<= x"E0000000";
        slotsrec.slot_1<='0'& signals.adres_sig(6 downto 0)&x"000000";
        slotsrec.slot_2<= signals.data_sig(15 downto 0) & x"0000";
        slotsrec.slot_3<= (others => '0');
        slotsrec.slot_4<= (others => '0');
        slotsrec.slot_5<= (others => '0');
```

```
        slotsrec.slot_6   <= (others => '0');
        slotsrec.slot_7   <= (others => '0');
        slotsrec.slot_8   <= (others => '0');
        slotsrec.slot_9   <= (others => '0');
        slotsrec.slot_10  <= (others => '0');
        slotsrec.slot_11  <= (others => '0');
        slotsrec.slot_12  <= (others => '0');
        signals.pcm_ack   <= '0';

elsif(signals.pcm_val='1' and signals.eq_pcm='0') and
(signals.cmd_val_sig='1')then
      slotsrec.slot_0 <= x"F8000000";
      slotsrec.slot_1 <='0'& signals.adres_sig(6 downto 0)& x"000000";
      slotsrec.slot_2 <= signals.data_sig(15 downto 0) & x"0000";
      slotsrec.slot_3 <= signals.pcm(31 downto 16) & x"0000";
      slotsrec.slot_4 <= signals.pcm(15 downto 0)  & x"0000";
      slotsrec.slot_5 <= (others => '0');
      slotsrec.slot_6 <= (others => '0');
      slotsrec.slot_7 <= (others => '0');
      slotsrec.slot_8 <= (others => '0');
      slotsrec.slot_9 <= (others => '0');
      slotsrec.slot_10<= (others => '0');
      slotsrec.slot_11<= (others => '0');
      slotsrec.slot_12<= (others => '0');
      signals.pcm_ack <= '1';

else signals.pcm_ack <= '0';
end if;

-----------------------------------------------------------------
--         send request signals
-----------------------------------------------------------------
if (signals.cmd_val_sig = '1') then outl_link.cmd_rqst <= '0';
elsif (signals.cmd_val_sig = '0') then outl_link.cmd_rqst <= '1';
else                                   outl_link.cmd_rqst <= '1';
end if;

-------------------------------------------------------------
--            synchronization signal
-------------------------------------------------------------
if(((signals.ack='1' and signals.equal='0')or
    (signals.pcm_val='1' and signals.waiting='0'))and
signals.codec_rst='1' and signals.eq_pcm='0')or(signals.sync='1')then

 if notx(signals.comb_sync) then
  v.comb_sync      := signals.comb_sync + 1;
  signals.seq_sync <= v.comb_sync;
  signals.sync     <= latch1(signals);
 end if;

elsif(signals.equal='1' or signals.waiting='1')then signals.sync<='0';
else signals.sync <= '0';
end if;

-----------------------------------------------------------
--    start streaming out
-----------------------------------------------------------
if ((signals.ready = '1' and signals.equal = '0') or
    (signals.pcm_ack = '1' and signals.waiting = '0')) or
    (signals.latch2 = '1') then
 data_out <= slot_shift(signals, slotsrec);
```

```
 signals.dataout        <= slot_shift(signals, slotsrec);
 signals.latch2         <= latch2(signals);
 signals.done           <= done(signals);

elsif (signals.sync2 = '0') then
 signals.latch2  <= '0';
 data_out        <= '0';
 signals.dataout <= '0';
else
end if;

-------------------------------------------------------
-- to reset the counter
-------------------------------------------------------
if notx(signals.seq_sync) then
 if (signals.seq_sync = "011111111" or signals.rst_ctr = '1') then
     signals.seq_sync <= (others => '1');
 else
 end if;
end if;

-----------------------------------------------------------
--increase the frame command counter
-------------------------------------------------------------
if notx(signals.comb_done) then
 if (signals.equal = '0' and signals.done = '1') then
  v.comb_done      := signals.comb_done + 1;
  signals.seq_done <= v.comb_done;
 else v.comb_done      := signals.comb_done;
      signals.seq_done <= v.comb_done;
 end if;
end if;

-------------------------------------------------------------
-- busy cmd
---------------------------------------------------------------
if (signals.start_cmd = '1') then signals.cmd_busy <= '1';
elsif (signals.start_pcm = '1') then signals.cmd_busy <= '0';
else
end if;

-------------------------------------------------------
-- acknowledgment signal
-------------------------------------------------------
if (signals.cmd_val_sig = '1') then signals.ack <= '1';
 elsif (signals.ready = '1') then signals.ack <= '0';
else
end if;

-------------------------------------------------------
-- to know when to stop sending pcm
-------------------------------------------------------
if (signals.inc_pcm = '1') then
 if (signals.frames_c = "011") then signals.eq_pcm <= '1';
 else
 end if;
else if (signals.done_dma = '1') then signals.eq_pcm<='0';
     signals.frames_s<=(others=>'0');
     else
     end if;
end if;
```

```
 sync               <= signals.sync;
 signals.start_cmd  <= start_cmd(signals);
 signals.done       <= done(signals);
 signals.inc_pcm    <= inc_pcm(signals);
 signals.start_pcm  <= start_pcm(signals);
 signals.frames_s   <= frame_count(signals);
 signals.rst_ctr    <= rst_ctr(signals);

 outl_link.start    <= start(signals);
 outl_link.donecopy <= signals.seq_done;
 outl_link.ack      <= signals.ack;

 out_pcm.frame      <= frame_count(signals);
 out_pcm.eq_pcm     <= signals.eq_pcm;

else
end if;

-----------------------------------------------
--             reset
-----------------------------------------------
if (rst = '0') then
 outl_link.cmd_rqst       <= '1';
 data_out                 <= '0';
 signals.dataout          <= '0';
 sync                     <= '0';
 signals.sync             <= '0';
 signals.pcm_ack          <= '0';
 signals.cmd_busy         <= '0';
 signals.ack              <= '0';
 signals.eq_pcm           <= '0';

 signals.seq_sync         <= (others => '0');
 signals.seq_done         <= (others => '0');
 signals.frames_s         <= (others => '0');


 slotsrec.slot_0          <= (others => '0');
 slotsrec.slot_1          <= (others => '0');
 slotsrec.slot_2          <= (others => '0');
 slotsrec.slot_3          <= (others => '0');
 slotsrec.slot_4          <= (others => '0');
 slotsrec.slot_5          <= (others => '0');
 slotsrec.slot_6          <= (others => '0');
 slotsrec.slot_7          <= (others => '0');
 slotsrec.slot_8          <= (others => '0');
 slotsrec.slot_9          <= (others => '0');
 slotsrec.slot_10         <= (others => '0');
 slotsrec.slot_11         <= (others => '0');
 slotsrec.slot_12         <= (others => '0');
else
end if;

end process;
```

```
-------------------------------------------------------------
-- sequential process
-------------------------------------------------------------

sequential : process(clk)

 begin

  if (rising_edge(clk)) then

      signals.comb_sync   <= signals.seq_sync;

      signals.cmd_val_sig <= inl_link.valid_if;
      signals.adres_sig   <= inl_link.adres_if;
      signals.data_sig    <= inl_link.data_if;
      signals.ready       <= inl_link.ready_if;
      signals.equal       <= inl_link.equal;
      signals.start_fetch <= inl_link.start_fetch;

      signals.comb_done   <= signals.seq_done;
      signals.sync2       <= signals.sync;
      signals.frames_c    <= signals.frames_s;

      signals.trigger     <= clks_in.r_clk;
      signals.codec_rst   <= clks_in.codec_rst;

      signals.pcm         <= in_pcm.pcm;
      signals.pcm_val     <= in_pcm.valid;
      signals.done_dma    <= in_pcm.done_link;
      signals.waiting     <= in_pcm.waiting;

      end if;

end process;

end;
```

A.2. AC97 Top Module VHDL Code

```
library ieee;
library grlib;
use ieee.std_logic_1164.all;
use work.ac97link_pack.all;
use grlib.amba.all;
use grlib.dma2ahb_package.all;
use grlib.at_ahb_mst_pkg.all;
use work.apbac97if_pack.all;


--------------------------------------
-- package declaration
--------------------------------------
package ac97top_pack is

-----------------------------------------
-- component declaration
-----------------------------------------
component ac97top
 generic (pindex    :     integer := 10;
          paddr     :     integer := 10;
          pmask     :     integer := 16#FFF#;
          vendorid  : in  integer := 0;
          deviceid  : in  integer := 0;
          version   : in  integer := 0);

 port (rst               : in  std_logic;
       clk               : in  std_logic;
       bit_clk           : in  std_logic;
       data_in           : in  std_logic;
       ac97top_inl       : in  apb_slv_in_type;
       ac97top_outl      : out apb_slv_out_type;
       ac97dma_in        : in  ahb_mst_in_type;
       ac97dma_out       : out ahb_mst_out_type;
       codec_rst         : out std_logic;
       data_out          : out std_logic;
       sync              : out std_logic);
end component;
end package;

library ieee;
use ieee.std_logic_1164.all;
use work.ac97link_pack.all;
use work.ac97top_pack.all;
use work.apbac97if_pack.all;

library grlib;
use grlib.amba.all;
use grlib.devices.all;
use grlib.dma2ahb_package.all;
use grlib.at_ahb_mst_pkg.all;
```

```
----------------------------------------
-- entity
----------------------------------------
entity ac97top is
 generic(pindex    :     integer := 10;
         paddr     :     integer := 10;
         pmask     :     integer := 16#FFF#;
         vendorid  : in  integer := 0;
         deviceid  : in  integer := 0;
         version   : in  integer := 0);

port (rst           : in  std_logic;
      clk           : in  std_logic;
      bit_clk       : in  std_logic;
      data_in       : in  std_logic;
      ac97top_inl   : in  apb_slv_in_type;
      ac97top_outl  : out apb_slv_out_type;
      ac97dma_in    : in  ahb_mst_in_type;
      ac97dma_out   : out ahb_mst_out_type;
      codec_rst     : out std_logic;
      data_out      : out std_logic;
      sync          : out std_logic);
end;

------------------------------------------------------------------------
-- architecture
------------------------------------------------------------------------
architecture ac97top_arch of ac97top is
 constant pconfig: apb_config_type:=
                (0=>ahb_device_reg(VENDOR_GAISLER,GAISLER_AC97,0,0,0),
                 1=>apb_iobar(paddr, pmask));
----------------------------------------------------
-- interface
----------------------------------------------------
component apb_ac97_if
        port(rst             : in  std_logic;
             clk             : in  std_logic;
             dmac_in         : in  ac97if_in_type;
             dmac_out        : out ac97if_out_type;
             apbac97if_inl   : in  apb_slv_in_type;
             apbac97if_inr   : in  aclink_outl_type;
             apbac97if_outl  : out apb_slv_out_type;
             apbac97if_outr  : out aclink_inl_type);
end component;
-----------------------------------------------
-- link clks
-----------------------------------------------
component linkclks
        port (rst            : in  std_logic;
              clk            : in  std_logic;
              bit_clk        : in  std_logic;
              data_in        : in  std_logic;
              inl            : in  aclink_inl_type;
              in_pcm         : in  pcm_in_type;
              out_pcm        : out pcm_out_type;
              codec_rst      : out std_logic;
              outl           : out aclink_outl_type;
              data_out       : out std_logic;
              sync           : out std_logic);
end component;
```

```
----------------------------------------
-- dma engine
----------------------------------------
component dma_engine
port (rst           : in  std_logic;
      clk           : in  std_logic;
      ahbin         : in  ahb_mst_in_type;
      dmacinr       : in  pcm_out_type;
      if_in         : in  ac97if_out_type;
      ahbout        : out ahb_mst_out_type;
      dmacoutr      : out pcm_in_type;
      if_out        : out ac97if_in_type);
end component;

signal dmac_in      : pcm_in_type;
signal dmac_out     : pcm_out_type;
signal inl_link     : aclink_inl_type;
signal outl_link    : aclink_outl_type;
signal dma2ahb_in   : dma_in_type;
signal dma2ahb_out  : dma_out_type;
signal if_in        : ac97if_in_type;
signal if_out       : ac97if_out_type;

begin
        ----------------------------------------
        --port map
        ----------------------------------------
        apbac97if0 : apb_ac97_if
        port map(clk             => clk,
                 rst             => rst,
                 dmac_in         => if_in,
                 dmac_out        => if_out,
                 apbac97if_inl   => ac97top_inl,
                 apbac97if_inr   => outl_link,
                 apbac97if_outl  => ac97top_outl,
                 apbac97if_outr  => inl_link);

        linkclks0 : linkclks
        port map(clk           => clk,
                 rst           => rst,
                 bit_clk       => bit_clk,
                 data_in       => data_in,
                 inl           => inl_link,
                 outl          => outl_link,
                 in_pcm        => dmac_in,
                 out_pcm       => dmac_out,
                 codec_rst     => codec_rst,
                 data_out      => data_out,
                 sync          => sync);

        dma_engine0 : dma_engine
        port map(rst           => rst,
                 clk           => clk,
                 ahbin         => ac97dma_in,
                 dmacinr       => dmac_out,
                 if_in         => if_out,
                 ahbout        => ac97dma_out,
                 dmacoutr      => dmac_in,
                 if_out        => if_in);
end;
```

A.3. Clocks VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.ac97link_pack.all;

----------------------------------------
-- entity
----------------------------------------
entity clks is

port (clk        : in  std_logic;
      rst        : in  std_logic;
      bit_clk    : in  std_logic;
      clks_out   : out clks_in_type;
      codec_rst  : out std_logic;
      warm_rst   : out std_logic);
end;

--------------------------------------------------
-- architecture
--------------------------------------------------
architecture clks_arch of clks is
 constant min : integer := 81;

 ------------------------------------------------------
 -- records
 ------------------------------------------------------
 type slow_clk is record
        r_sig   : std_logic;
        f_sig   : std_logic;
        meta0   : std_logic;
        meta1   : std_logic;
        clk0    : std_logic;
        clk1    : std_logic;
        clk2    : std_logic;
        clk3    : std_logic;
        cold    : std_logic;
        warm    : std_logic;
        states  : std_logic_vector(3 downto 0);
        c_cold  : std_logic_vector(6 downto 0);
        s_cold  : std_logic_vector(6 downto 0);
        c_warm  : std_logic_vector(7 downto 0);
        s_warm  : std_logic_vector(7 downto 0);
 end record;

signal signals_clk  : slow_clk;

-------------------------------------------------
--              notx function
-------------------------------------------------
function notx(d : std_logic_vector) return boolean is
variable res : boolean;
begin
 res := true;
-- pragma translate_off
 res := not is_x(d);
-- pragma translate_on
 return (res);
```

```
end;

begin

-------------------------------------------------------------
-- combinational process
-------------------------------------------------------------
combinational : process(rst, signals_clk)
   variable v : slow_clk;
   begin

   ------------------------------------------------
   -- rising or falling, and warm reset
   ------------------------------------------------
   if notx(signals_clk.states) then
    if  (signals_clk.states = "0000") then

    ------------------------------------------------------------
    --                       warm rst
    ------------------------------------------------------------
    if notx(signals_clk.c_warm) then
       v.c_warm            := signals_clk.c_warm + 1;
       signals_clk.s_warm  <= v.c_warm;
       if(signals_clk.c_warm>=min*2 and signals_clk.c_warm<min*3)then
            warm_rst <= '1';
       else warm_rst <= '0';
       end if;
    end if;

    elsif  (signals_clk.states = "0011") then
              signals_clk.r_sig   <= '1';
              signals_clk.f_sig   <= '0';
              signals_clk.s_warm  <= (others => '1');

    elsif  (signals_clk.states = "1100") then
              signals_clk.r_sig   <= '0';
              signals_clk.f_sig   <= '1';
              signals_clk.s_warm  <= (others => '1');

    else
              signals_clk.r_sig   <= '0';
              signals_clk.f_sig   <= '0';
              signals_clk.s_warm  <= (others => '1');
    end if;
   end if;

   clks_out.r_clk  <= signals_clk.r_sig;
   clks_out.f_clk  <= signals_clk.f_sig;

   ------------------------------------------------
   --               cold rst
   ------------------------------------------------
   if notx(signals_clk.c_cold) then
    if (signals_clk.cold = '1') then
        v.c_cold            := signals_clk.c_cold + 1;
        signals_clk.s_cold  <= v.c_cold;
        if (signals_clk.s_cold <= min) then
            signals_clk.cold   <= '1';
            codec_rst          <= '0';
            clks_out.codec_rst <= '0';
```

```
        else
            signals_clk.cold   <= '0';
            codec_rst          <= '1';
            clks_out.codec_rst <= '1';
        end if;
     else
     end if;
   end if;

   ---------------------------------------------
   --  reset
   ---------------------------------------------
   if (rst = '0') then

         signals_clk.r_sig   <= '0';
         signals_clk.f_sig   <= '0';
         signals_clk.cold    <= '1';
         signals_clk.warm    <= '0';
         warm_rst            <= '0';
         codec_rst           <= '1';
         clks_out.codec_rst  <= '1';
         signals_clk.s_cold  <= (others => '0');
         signals_clk.s_warm  <= (others => '1');

   else
   end if;

end process;

-------------------------
-- sequential process
-------------------------------------
sequential : process(clk)
begin
        if(rising_edge(clk)) then

           signals_clk.meta0  <= bit_clk;
           signals_clk.meta1  <= signals_clk.meta0;
           signals_clk.clk0   <= signals_clk.meta1;
           signals_clk.clk1   <= signals_clk.clk0;
           signals_clk.clk2   <= signals_clk.clk1;
           signals_clk.clk3   <= signals_clk.clk2;
           signals_clk.states <= signals_clk.clk0 & signals_clk.clk1 &
                                 signals_clk.clk2 & signals_clk.clk3;
           signals_clk.c_cold <= signals_clk.s_cold;
           signals_clk.c_warm <= signals_clk.s_warm;

        end if;

end process;

end;
```

A.4. DMAC VHDL Code

```
library grlib;
use grlib.dma2ahb_package.all;
use grlib.at_ahb_mst_pkg.all;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.ac97link_pack.all;
use work.apbac97if_pack.all;


----------------------------------------
-- entity
----------------------------------------
entity dmac is

port (hclk      : in  std_logic;
      hrstn     : in  std_logic;
      if_in     : in  ac97if_out_type;
      if_out    : out ac97if_in_type;
      dmacinr   : in  pcm_out_type;
      dmacoutr  : out pcm_in_type;
      dmacinl   : in  dma_out_type;
      dmacoutl  : out dma_in_type);
end;

----------------------------------------------------
-- architecture
----------------------------------------------------
architecture dmac_arch of dmac is

-----------------------------------------------------
-- records and signals
-----------------------------------------------------
type reg_type is record
 grant          : std_logic;
 ready          : std_logic;
 valid          : std_logic;
 maxfifo        : std_logic;
 eq_pcm         : std_logic;
 done_link      : std_logic;
 first          : std_logic;
 waiting        : std_logic;
 start          : std_logic;
 checked        : std_logic;
 base           : std_logic_vector(31 downto 0);
 data           : std_logic_vector(31 downto 0);
 adres_c        : std_logic_vector(31 downto 0);
 adres_s        : std_logic_vector(31 downto 0);
 fifo_c         : std_logic_vector(2 downto 0);
 fifo_s         : std_logic_vector(2 downto 0);
 frames         : std_logic_vector(2 downto 0);
end record;

type state_type is (s0, s1, s2, s3, s4);
type fifo_type  is array (0 to 7) of std_logic_vector(31 downto 0);
```

```
signal signals    : reg_type;
signal next_state : state_type;
signal curr_state : state_type;
signal fifo       : fifo_type;

-----------------------------------------------------
--         notx function
-----------------------------------------------------
function notx(d : std_logic_vector) return boolean is
        variable res : boolean;
        begin
        res := true;
        -- pragma translate_off
        res := not is_x(d);
        -- pragma translate_on
        return (res);
end;

-------------------------------------------------------------
-- valid data
-------------------------------------------------------------
function valid (countfun : reg_type) return std_logic is
        variable flag : std_logic;
        begin
        if (countfun.data = x"00000000") then flag := '0';
        else flag := '1';
        end if;
        return flag;
end valid;

begin

---------------------------------------------------------------------
-- combinational process
---------------------------------------------------------------------
combinational : process(hrstn, signals, curr_state)
 variable v    : reg_type;
 begin

 ----------------------------------------
 -- FSM
 ----------------------------------------
 case curr_state is
 when s0 => if (signals.start = '1') then next_state <= s1;
                signals.done_link <= '0';
                signals.first     <= '0';
            else next_state <= s0;
            end if;
            signals.maxfifo  <= '0';
            signals.waiting  <= '1';
            signals.fifo_s   <= (others => '0');

            dmacoutl.reset   <= '1';
            dmacoutr.waiting <= '1';

            if_out.wait_dma  <= '1';
            if_out.busy_dma  <= '0';
            if_out.done_dma  <= '0';
            if_out.done_link <= '0';
```

```
when s1 => if (signals.grant = '0') then next_state <= s1;
           elsif (signals.grant = '1') then next_state <= s2;

            ----------------------------------------
            -- getting the correct address
            ----------------------------------------
            if (signals.first = '0') then
                 dmacoutl.address <= signals.base;
                 signals.adres_s  <= signals.base;
            else dmacoutl.address <= signals.adres_c;
            end if;
           else
           end if;

           dmacoutl.burst   <= '1';
           dmacoutl.request <= '1';
           dmacoutl.lock    <= '1';
           dmacoutl.store   <= '0';
           dmacoutl.beat    <= hincr4;
           dmacoutl.size    <= hsize32;
           dmacoutr.waiting <= '0';

           signals.waiting  <= '0';

           if_out.wait_dma  <= '0';
           if_out.busy_dma  <= '1';
           if_out.done_dma  <= '0';
           if_out.done_link <= '1';

when s2 => if (signals.ready = '1') then next_state <= s3;
               signals.done_link <= '0';

               ------------------------------------------------
               -- saving the first data
               ------------------------------------------------
               fifo(conv_integer(signals.fifo_c)) <= signals.data;
               v.fifo_c := signals.fifo_c + 1;
               signals.fifo_s <= v.fifo_c;

               ----------------------------------------
               -- check if it is valid only the first time
               ----------------------------------------
               if (signals.checked = '0') then
                   dmacoutr.valid   <= valid(signals);
                   signals.valid    <= valid(signals);
                   signals.checked  <= '1';
               else
               end if;

           else next_state <= s2;
           end if;

           if_out.busy_dma  <= '1';
           if_out.done_dma  <= '0';

           dmacoutl.reset   <= '0';
```

```
when s3 => if (signals.fifo_c = "111") then
               signals.maxfifo <= '1';

               ------------------------------------------
               -- getting the last address
               ------------------------------------------
               v.adres_c        := signals.adres_c + x"10";
               signals.adres_s  <= v.adres_c;

               ------------------------------------------------
               -- saving the last data
               ------------------------------------------------
               fifo(conv_integer(signals.fifo_c)) <= signals.data;

           elsif (signals.fifo_c /= "111") then
               signals.maxfifo  <= '0';

               ------------------------------------------------
               -- saving data from 1 up to 6
               ------------------------------------------------
               fifo(conv_integer(signals.fifo_c)) <= signals.data;
               v.fifo_c           := signals.fifo_c + 1;
               signals.fifo_s  <= v.fifo_c;

           else
           end if;

  if(signals.ready='0' and signals.maxfifo='0')then next_state<=s1;
  elsif(signals.ready='0' and signals.maxfifo='1')then
  elsif(signals.ready='1' and signals.maxfifo='0')then next_state<=s3;
  elsif(signals.ready='1' and signals.maxfifo='1')then next_state<=s4;
  else
  end if;

  if_out.busy_dma  <= '1';
  if_out.done_dma  <= '0';

when s4=> if(signals.eq_pcm='0' and signals.valid='0')then
               next_state<=s0;
          elsif(signals.eq_pcm='0' and signals.valid='1')then
               next_state<= s4;
          else next_state<=s1;
               signals.done_link <= '1';
          end if;

          dmacoutl.request <= '0';
          dmacoutl.lock    <= '0';
          dmacoutl.burst   <= '0';
          dmacoutl.reset   <= '1';

          if_out.busy_dma  <= '0';
          if_out.done_dma  <= '1';

          signals.first        <= '1';
          signals.checked  <= '0';
          signals.fifo_s   <= (others => '0');

end case;

dmacoutr.done_link <= signals.done_link;
dmacoutr.pcm       <= fifo(conv_integer(signals.frames));
```

```
  ---------------------------------------------
  -- reset
  ---------------------------------------------
  if (hrstn = '0') then

    next_state      <= s0;

    signals.fifo_s  <= (others => '0');
    signals.adres_s <= (others => '0');
    signals.maxfifo <= '0';
    signals.first   <= '0';
    signals.valid   <= '0';
    signals.checked <= '0';

  else
  end if;

end process;

  ---------------------------------------------------------------
  -- sequential process
  ---------------------------------------------------------------

  sequential : process(hclk)
   begin
      if (rising_edge(hclk)) then

           curr_state            <= next_state;

           signals.grant         <= dmacinl.grant;
           signals.ready         <= dmacinl.ready;
           signals.data          <= dmacinl.data;

           signals.fifo_c        <= signals.fifo_s;
           signals.adres_c   <= signals.adres_s;

           signals.frames        <= dmacinr.frame;
           signals.eq_pcm        <= dmacinr.eq_pcm;

           signals.base          <= if_in.base;
           signals.start         <= if_in.start;

      end if;
  end process;
end;
```

A.5. DMA Engine VHDL Code

```
library ieee;
library grlib;
use ieee.std_logic_1164.all;
use work.ac97link_pack.all;
use grlib.amba.all;
use grlib.dma2ahb_package.all;
use grlib.at_pkg.all;
use grlib.at_ahb_mst_pkg.all;
use work.apbac97if_pack.all;

---------------------------------------
-- entity
---------------------------------------
entity dma_engine is

port (rst           : in  std_logic;
      clk           : in  std_logic;
      ahbin         : in  ahb_mst_in_type;
      dmacinr       : in  pcm_out_type;
      if_in         : in  ac97if_out_type;
      ahbout        : out ahb_mst_out_type;
      dmacoutr      : out pcm_in_type;
      if_out        : out ac97if_in_type);

end;

-----------------------------------------------
-- architecture
-----------------------------------------------
architecture dma_engine_arch of dma_engine is

-------------------------------------------------
-- dma 2 ahb
-------------------------------------------------
component dma2ahb

        generic(hindex     : in integer := 5;
                vendorid   : in integer := 0;
                deviceid   : in integer := 0;
                version    : in integer := 0;
                syncrst    : in integer := 1;
                boundary   : in integer := 1);

        port(hclk    : in  std_ulogic;
             hresetn : in  std_ulogic;
             dmain   : in  dma_in_type;
             dmaout  : out dma_out_type;
             ahbin   : in  ahb_mst_in_type;
             ahbout  : out ahb_mst_out_type);

end component;
```

```
------------------------------------------------
-- dmac
------------------------------------------------
component dmac
        port (hclk      : in  std_logic;
              hrstn     : in  std_logic;
              if_in     : in  ac97if_out_type;
              if_out    : out ac97if_in_type;
              dmacinr   : in  pcm_out_type;
              dmacoutr  : out pcm_in_type;
              dmacinl   : in  dma_out_type;
              dmacoutl  : out dma_in_type);
end component;

--------------------------------
-- intermediate signals
--------------------------------
signal dmac_in  : dma_out_type;
signal dmac_out : dma_in_type;

begin

        ---------------------------------------
        -- port map
        ---------------------------------------
        dma2ahb0 : dma2ahb
        port map(hclk    => clk,
                 hresetn => rst,
                 dmain   => dmac_out,
                 dmaout  => dmac_in,
                 ahbin   => ahbin,
                 ahbout  => ahbout);

        dmac0 : dmac
        port map(hclk      => clk,
                 hrstn     => rst,
                 if_in     => if_in,
                 if_out    => if_out,
                 dmacinr   => dmacinr,
                 dmacoutr  => dmacoutr,
                 dmacinl   => dmac_in,
                 dmacoutl  => dmac_out);
end;
```

A.6. Interface VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use work.ac97link_pack.all;

library grlib;
use grlib.amba.all;

--------------------------
-- package delcaration
--------------------------
package apbac97if_pack is

------------------------------
-- inputs from dmac
------------------------------
type ac97if_in_type is record
        done_link : std_logic;
        busy_dma  : std_logic;
        wait_dma  : std_logic;
        done_dma  : std_logic;
end record;

-----------------------------------------------
-- outputs to dmac
-----------------------------------------------
type ac97if_out_type is record
        start : std_logic;
        base  : std_logic_vector(31 downto 0);
end record;

-------------------------------------------------
-- component declaration
-------------------------------------------------
component apbac97if_comp
 port (clk             : in  std_logic;
       rst             : in  std_logic;
       dmac_in         : in  ac97if_in_type;
       dmac_out        : out ac97if_out_type;
       apbac97if_inl   : in  apb_slv_in_type;
       apbac97if_inr   : in  aclink_outl_type;
       apbac97if_outl  : out apb_slv_out_type;
       apbac97if_outr  : out aclink_inl_type);
end component;

end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.ac97link_pack.all;
use work.apbac97if_pack.all;


library grlib;
use grlib.amba.all;
use grlib.devices.all;
```

```
------------------------------------------------
--          entity
------------------------------------------------
entity apb_ac97_if is
generic (pindex    :     integer := 10;
         paddr     :     integer := 10;
         pmask     :     integer := 16#FFF#;
         vendorid  : in  integer := 16#01#;
         deviceid  : in  integer := 16#08C#;
         version   : in  integer := 0);

port (clk             : in  std_logic;
      rst             : in  std_logic;
      dmac_in         : in  ac97if_in_type;
      dmac_out        : out ac97if_out_type;
      apbac97if_inl   : in  apb_slv_in_type;
      apbac97if_inr   : in  aclink_outl_type;
      apbac97if_outl  : out apb_slv_out_type;
      apbac97if_outr  : out aclink_inl_type);
end;

------------------------------------------------
-- architecture
------------------------------------------------
architecture apb_ac97_if_arch of apb_ac97_if is

------------------------------------------------
-- records
------------------------------------------------
type apbac97if_type is record
        valid           : std_logic;
        cmdrqstsig      : std_logic;
        ready           : std_logic;
        slv_enable      : std_logic;
        slv_write       : std_logic;
        equal           : std_logic;
        sync            : std_logic;
        ack             : std_logic;
        start_fetch     : std_logic;
        done_link       : std_logic;
        busy_dma        : std_logic;
        done_dma        : std_logic;
        wait_dma        : std_logic;
        donecopy        : std_logic_vector(7 downto 0);
        slv_sel         : std_logic_vector(0 to 15);
        comb            : std_logic_vector(7 downto 0);
        seq             : std_logic_vector(7 downto 0);
        slv_adres       : std_logic_vector(31 downto 0);
        slv_data        : std_logic_vector(31 downto 0);
end record;
signal signals : apbac97if_type;

------------------------------------------------------
-- configuration grlib
------------------------------------------------------
constant pconfig : apb_config_type :=
(0 => ahb_device_reg (VENDOR_GAISLER, GAISLER_AC97, 0, 0, 0),
 1 => apb_iobar(paddr, pmask));

type register_type is array(0 to 255)of std_logic_vector(31 downto 0);
type adres_mem is array(0 to 255)of std_logic_vector(31 downto 0);
```

```
type data_mem is array(0 to 255)of std_logic_vector(31 downto 0);

signal registers  : register_type;
signal adres      : adres_mem;
signal data       : data_mem;

-----------------------------------------------------
---------              notx function
-----------------------------------------------------
function notx(d : std_logic_vector) return boolean is
 variable res : boolean;
 begin
  res := true;
  -- pragma translate_off
  res := not is_x(d);
  -- pragma translate_on
 return (res);
end;

begin
  -------------------------------------
  -----  combinational process --------
  -------------------------------------
  combinational : process(rst, signals)
    variable count  : integer;
    variable v      : apbac97if_type;
    variable temp   : std_logic_vector(31 downto 0);
    variable temp2  : std_logic_vector(31 downto 0);
    variable temp3  : std_logic_vector(31 downto 0);

  begin
  ------------------------------------------------------------
  -- write into the registers
  ------------------------------------------------------------
  if(signals.slv_write and signals.slv_sel(pindex) and
     signals.slv_enable) = '1' then

   if notx(signals.slv_adres) or notx(signals.seq) then
     temp := signals.slv_data;
     registers(conv_integer(signals.slv_adres(7 downto 0))) <= temp;

     ------------------------------------------------------------------
     -- start the data fetching from memory and get the base address
     ------------------------------------------------------------------
     if (signals.slv_adres(7 downto 0) = x"80") then
         dmac_out.start              <= '1';
         apbac97if_outr.start_fetch  <= '1';
         signals.start_fetch         <= '1';
         dmac_out.base               <= signals.slv_data;

     ------------------------------------------------------------
     -- storing the values inside the memory
     ------------------------------------------------------------
     else
         count              := conv_integer(signals.seq);
         v.comb             := signals.comb + 1;
         signals.seq        <= v.comb;
         adres(conv_integer(signals.seq)) <= signals.slv_adres;
         data(conv_integer(signals.seq))<=x"0000"&
                                          signals.slv_data(15 downto 0);
     end if;
```

```
 end if;
else
end if;

---------------------------------
-- stop fetching data
---------------------------------
if (signals.done_link = '1') then
    dmac_out.start              <= '0';
    signals.start_fetch         <= '0';
    apbac97if_outr.start_fetch  <= '0';
else
end if;

-------------------------------------------------------------
-- check if somehting new has been written
-------------------------------------------------------------
if notx(signals.donecopy) or notx(signals.seq)then
 if (signals.donecopy = signals.seq) then signals.equal <= '1';
 else signals.equal <= '0';
 end if;
 apbac97if_outr.equal <= signals.equal;
end if;

---------------------------------
-- asserting valid signal
---------------------------------
if notx(signals.donecopy) then
 if (signals.cmdrqstsig = '1' and count > 0 and
     signals.equal = '0' and signals.start_fetch ='0') then
      signals.valid <= '1';

      ------------------------------------------------------------
      --             sending the adres/data to the link
      ------------------------------------------------------------
      temp2 := adres(conv_integer(signals.donecopy));
      apbac97if_outr.adres_if <= '0' & temp2(31 downto 1);
      temp3 := data(conv_integer(signals.donecopy));
      apbac97if_outr.data_if  <= temp3;

 else
 end if;
end if;

-----------------------------------------------------
--- asserting ready signal/de-asserting valid signal
-----------------------------------------------------
if (signals.ack = '1' and signals.sync = '1') then
            signals.ready <= '1';
            signals.valid <= '0';
else
            signals.ready <= '0';
end if;
```

```
  ---------------------------------------------
  -- reset
  if (rst = '0') then
         count                        := 0;
         signals.valid                <= '0';
         signals.equal                <= '0';
         dmac_out.start               <= '0';
         signals.start_fetch          <= '0';
         apbac97if_outr.start_fetch   <= '0';
         apbac97if_outr.adres_if      <= (others => '0');
         apbac97if_outr.data_if       <= (others => '0');
         apbac97if_outl.prdata        <= (others => '0');
         signals.seq                  <= (others => '0');
         adres                        <= (others => (others => '0'));
         data                         <= (others => (others => '0'));
         registers                    <= (others => (others => '0'));
  else
  end if;
  ------------------------------------------------------
  --read from the registers
  if notx(signals.slv_adres) then
  ---------------------------------------------
  -- dma status bits
  apbac97if_outl.pirq(2) <= signals.busy_dma;
  apbac97if_outl.pirq(1) <= signals.done_dma;
  apbac97if_outl.pirq(0) <= signals.wait_dma;
  ---------------------------------------------
  apbac97if_outl.prdata(15 downto 0) <=
  registers(conv_integer(signals.slv_adres(7 downto 0)))(15 downto 0);
  end if;
  apbac97if_outl.pconfig  <= pconfig;
  apbac97if_outl.pindex   <= pindex;
  apbac97if_outr.valid_if <= signals.valid;
  apbac97if_outr.ready_if <= signals.ready;
  apbac97if_outr.done_dma <= signals.done_dma;
end process;
-----------------------------------------
-- sequential process
sequential : process(clk)
begin
     if (rising_edge(clk)) then
          signals.slv_sel     <= apbac97if_inl.psel;
          signals.slv_enable  <= apbac97if_inl.penable;
          signals.slv_adres   <= apbac97if_inl.paddr;
          signals.slv_write   <= apbac97if_inl.pwrite;
          signals.slv_data    <= apbac97if_inl.pwdata;
          signals.comb        <= signals.seq;
          signals.cmdrqstsig  <= apbac97if_inr.cmd_rqst;
          signals.donecopy    <= apbac97if_inr.donecopy;
          signals.sync        <= apbac97if_inr.start;
          signals.ack         <= apbac97if_inr.ack;
          signals.done_dma    <= dmac_in.done_dma;
          signals.busy_dma    <= dmac_in.busy_dma;
          signals.wait_dma    <= dmac_in.wait_dma;
          signals.done_link   <= dmac_in.done_link;
     else
     end if;
end process;
end;
```

A.7. Link Clocks VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use work.ac97link_pack.all;

---------------------------------------
-- entity
---------------------------------------
entity linkclks is
port (clk       : in  std_logic;
      rst       : in  std_logic;
      bit_clk   : in  std_logic;
      data_in   : in  std_logic;
      inl       : in  aclink_inl_type;
      in_pcm    : in  pcm_in_type;
      out_pcm   : out pcm_out_type;
      codec_rst : out std_logic;
      outl      : out aclink_outl_type;
      data_out  : out std_logic;
      sync      : out std_logic);
end;

-----------------------------------------
-- architecture
-----------------------------------------
architecture linkclks_arch of linkclks is

-------------------------------------------
-- components to use
-------------------------------------------
component ac97link
        port (clk       : in  std_logic;
              rst       : in  std_logic;
              clks_in   : in  clks_in_type;
              inl_link  : in  aclink_inl_type;
              data_in   : in  std_logic;
              in_pcm    : in  pcm_in_type;
              out_pcm   : out pcm_out_type;
              outl_link : out aclink_outl_type;
              data_out  : out std_logic;
              sync      : out std_logic);
end component;

component clks
        port (clk        : in  std_logic;
              rst        : in  std_logic;
              bit_clk    : in  std_logic;
              clks_out   : out clks_in_type;
              codec_rst  : out std_logic;
              warm_rst   : out std_logic);
end component;

---------------------------------
-- intermediate signals
---------------------------------
signal clks_in_out  : clks_in_type;
signal warmrst      : std_logic;
signal syncsig      : std_logic;
```

```
begin
        ----------------------------------------
        -- port map
        ----------------------------------------
        clks0 : clks
        port map(clk       => clk,
                 rst       => rst,
                 bit_clk   => bit_clk,
                 clks_out  => clks_in_out,
                 codec_rst => codec_rst,
                 warm_rst  => warmrst);

        ac97link0 : ac97link
        port map(clk       => clk,
                 rst       => rst,
                 clks_in   => clks_in_out,
                 inl_link  => inl,
                 data_in   => data_in,
                 in_pcm    => in_pcm,
                 out_pcm   => out_pcm,
                 outl_link => outl,
                 data_out  => data_out,
                 sync      => syncsig);

        sync <= syncsig or warmrst;

end;
```

A.8. Stand-Alone Test bench VHDL Code for the First Stage of the Design

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

library gaisler;
library grlib;
use grlib.stdlib.conv_std_logic_vector;
use grlib.stdlib.conv_integer;
use grlib.stdlib.conv_std_logic;
use grlib.stdlib.tost;
use grlib.stdlib."+";
use grlib.testlib.print;
use grlib.amba.all;
use grlib.at_pkg.all;
use grlib.at_ahb_mst_pkg.all;
use work.ac97link_pack.all;
use work.apbac97if_pack.all;

------------------------------
-- entity
entity ac97_tb is
-------------------------------------------------------------
-- generics
generic(sysperiod_g     : integer := 20; --systemperiod in ns
        apbaddr_g       : integer := 16#800#;
        pindex          : integer := 0;
        paddr           : integer := 0;
        pmask           : integer := 16#FFF#;
        console         : integer := 0;
        pirq            : integer := 0;
        parity          : integer := 1;
        flow            : integer := 1;
        fifosize        : integer range 1 to 32 := 32;
        abits           : integer := 8);
end entity ac97_tb;
-----------------------------------------
-- architecture
architecture behavioural of ac97_tb is
-----------------------------------------
-- component
component ac97top
port (rst               : in  std_logic;
      clk               : in  std_logic;
      bit_clk           : in  std_logic;
      data_in           : in  std_logic;
      ac97top_inl       : in  apb_slv_in_type;
      codec_rst         : out std_logic;
      ac97top_outl      : out apb_slv_out_type;
      data_out          : out std_logic;
      sync              : out std_logic);
end component;

-- Tests
constant do_basic_read_test   : boolean := true;
constant do_basic_write_test  : boolean := true;
```

```
constant vmode        : boolean := false;
constant sysperiod_c  : time := sysperiod_g * 1 ns;
constant datareg_c    : std_logic_vector(31 downto 0):=uartaddr_c;
constant statusreg_c  : std_logic_vector(31 downto 0):=uartaddr_c+4;
constant ctrlreg_c    : std_logic_vector(31 downto 0):=uartaddr_c+8;
constant scalerreg_c  : std_logic_vector(31 downto 0):=uartaddr_c+12;
constant fifodbgreg_c : std_logic_vector(31 downto 0):=uartaddr_c+16;
constant uartaddr_c   : std_logic_vector(31 downto 0):=
apbaddr_c+(conv_std_logic_vector(paddr,12) and
conv_std_logic_vector(pmask,12));

constant apbaddr_c    : std_logic_vector(31 downto 0):=
conv_std_logic_vector(apbaddr_g, 12) & x"00000";

constant size : integer := 127;
constant frame : integer := 255;

type bit_array  is array (size downto 0) of std_logic;
type bit32_array is array (size downto 0) of
std_logic_vector(31 downto 0);
type reg_values is array (frame downto 0) of
std_logic_vector(31 downto 0);
type frame_array  is array (frame downto 0) of std_logic;

signal rstn              : std_ulogic := '0';
signal clk               : std_ulogic := '1';
signal apbi              : apb_slv_in_type;
signal apbo              : apb_slv_out_vector:=(others => apb_none);
signal ahbmi             : ahb_mst_in_type;
signal ahbmo             : ahb_mst_out_vector:=(others => ahbm_none);
signal ahbsi             : ahb_slv_in_type;
signal ahbso             : ahb_slv_out_vector:=(others => ahbs_none);
signal atmi              : at_ahb_mst_in_type;
signal atmo              : at_ahb_mst_out_type;
signal enablemon         : std_ulogic;
signal irqdetected       : std_ulogic;
signal clearirq          : std_ulogic;
signal bit_clk           : std_logic;
signal data_in           : std_logic;
signal codec_rst         : std_logic;
signal data_out          : std_logic;
signal sync              : std_logic;
signal data_to_write_2   : bit32_array := (others => (others => '0'));
signal frame_1           : frame_array := (others => '0');
signal frame_2           : frame_array := (others => '0');
signal frame_3           : frame_array := (others => '0');
signal frame_4           : frame_array := (others => '0');
signal frame_5           : frame_array := (others => '0');
signal i                 : integer := 0;
```

```
-------------------------------------------
--   functions
-------------------------------------------
function bin (mychar : character) return std_logic is
 variable bin : std_logic;
 begin
 case mychar is
             when '0'        => bin := '0';
             when '1'        => bin := '1';
             when 'x'        => bin := '0'
when others => assert (false) report "no binary character read"
severity failure;
end case;
return bin;
end bin;
-------------------------------------------------------------
function load_bit (filename : string) return bit_array is
 file objectfile : text open read_mode is filename;
 variable memory : bit_array;
 variable l      : line;
 variable i      : integer := 0;
 variable mychar : character;
 begin
  while not endfile(objectfile) loop readline(objectfile, l);
   read(l, mychar);
   memory(i) := bin(mychar);
   i := i + 1;
  end loop;
 return memory;
end load_bit;
-------------------------------------------------------------
function load_frame (filename : string) return frame_array is
 file objectfile : text open read_mode is filename;
 variable memory : frame_array;
 variable l      : line;
 variable i      : integer := 0;
 variable mychar : character;
 begin
  while not endfile(objectfile) loop readline(objectfile, l);
   read(l, mychar);
   memory(i) := bin(mychar);
   i := i + 1;
  end loop;
 return memory;
end load_frame;
-------------------------------------------------------------
function load_regvals (filename : string) return reg_values is
 file objectfile : text open read_mode is filename;
 variable memory : reg_values;
 variable l      : line;
 variable index  : natural := 0;
 variable mychar : character;
 begin
  while not endfile(objectfile) loop readline(objectfile, l);
   for i in 31 downto 0 loop
    read(l, mychar);
    memory(index)(i) := bin(mychar);
   end loop;index := index + 1;
  end loop;
 return memory;
end load_regvals;
```

```
------------------------------------------------------------------
function load_32bit (filename : string) return bit32_array is
 file objectfile : text open read_mode is filename;
 variable memory : bit32_array;
 variable l      : line;
 variable index  : natural := 0;
 variable mychar : character;
 begin
  while not endfile(objectfile) loop readline(objectfile, l);
   for i in 31 downto 0 loop
    read(l, mychar);
    memory(index)(i) := bin(mychar);
   end loop;
   index := index + 1;
  end loop;
  return memory;
end load_32bit;

begin
--------------------------------------
-- clk generation
--------------------------------------
clk <= not clk after (sysperiod_c/2);
rstn <= '0', '1' after 300 ns;

------------------------------------------------------------------
-- AMBA infrastructure
------------------------------------------------------------------
ahb0 : at_ahb_ctrl             -- AHB arbiter/multiplexer
generic map (defmast     => 0,
             split       => 1,
             enebterm    => 1,
             ebprob      => 1,
             rrobin      => 1,
             ioaddr      => 16#FFF#,
             hmstdisable => 16#4000#,
             ioen        => 1,
             nahbm       => 3,
             nahbs       => 2,
             hslvdisable => 16#600#,
             enbusmon    => 0,
             assertwarn  => 1,
             asserterr   => 1)
port map (rstn, clk, ahbmi, ahbmo, ahbsi, ahbso);

apb0 : apbctrl                     -- AHB/APB bridge
generic map (hindex      => 0,
             haddr       => apbaddr_g,
             enbusmon    => 0,
             asserterr   => 1,
             assertwarn  => 1,
             pslvdisable => 1,
             nslaves     => 1)
port map (rstn, clk, ahbsi, ahbso(0), apbi, apbo);

dma1 :  at_ahb_mst
generic map(hindex    => 0,
            vendorid  => 0,
            deviceid  => 0,
            version   => 0)
```

```
port map(hclk    => clk, -- AMBA AHB system signals
         hresetn => rstn,
         atmi    => atmi, -- Direct Memory Access Interface
         atmo    => atmo,
         ahbi    => ahbmi, -- AMBA AHB Master Interface
         ahbo    => ahbmo(0));

------------------------------
-- Component instantiation
------------------------------
inst0 : ac97top
port map(clk           => clk,
         rst           => rstn,
         bit_clk       => bit_clk,
         ac97top_inl   => apbi,
         data_in       => data_in,
         codec_rst     => codec_rst,
         ac97top_outl  => apbo(pindex),
         sync          => sync,
         data_out      => data_out);

-------------------------------------------
--   initialize test vectors
-------------------------------------------
init: process(i)
      begin
       if i = 0 then
        data_to_write_2 <=  load_32bit(string'("data_to_write_2.tv"));
        frame_1         <= load_frame(string'("frame_1.tv"));
        frame_2         <= load_frame(string'("frame_2.tv"));
        frame_3         <= load_frame(string'("frame_3.tv"));
        frame_4         <= load_frame(string'("frame_4.tv"));
        frame_5         <= load_frame(string'("frame_5.tv"));
       end if;
end process;

-----------------------------------------------------------------------
-- process to read or write
-----------------------------------------------------------------------
test_p : process is
        variable tp            : boolean;
        variable tpcounter     : integer;
        variable d             : std_logic_vector(31 downto 0);
        variable c             : std_logic_vector(31 downto 0);
        variable e             : std_logic;
        variable time0         : time;
        variable time1         : time;
        variable dummy         : boolean;

begin
        print("testbench start");
        at_init(atmi);
        wait until rstn = '1';
```

```
-------------------------------------------------------------------------
--                   test 1
-------------------------------------------------------------------------
if do_basic_write_test then print("test1:writing "& time'image(now));
 for i in 0 to 32 loop
  d := conv_std_logic_vector(i*4, 32);
  at_write_32(address           => apbaddr_c  + i*4,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput      => vmode,
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);
  print("written data at address "& tost(apbaddr_c+i*4)&" is "&
  tost(d));
 end loop;
end if;

-------------------------------------------------------------------------
--                   test 2
-------------------------------------------------------------------------
if do_basic_read_test then
 print("test 2: reading the written values...." & time'image(now));
 for i in 0 to size loop
  at_read_32(address           => apbaddr_c + i,
             waitcycles        => 0,
             lock              => false,
             hprot             => "0011",
             back2back         => true,
             screenoutput      => vmode,
             data              => d,
             atmi              => atmi,
             atmo              => atmo);
  print("read data at address " & tost(apbaddr_c + i) & " is " &
  tost(d));
  assert(apbo(0).prdata = data_to_write_2(i))
  report "error:wrong result vector at test 2 and vector" &
  integer'image(i)
  severity error;
 end loop;
end if;
wait for 7500 ns;

-------------------------------------------------------------------------
--                   test 3
-------------------------------------------------------------------------
if do_basic_write_test then
 print("test 3: writing the FIRST valid command at addres: " &
 tost(apbaddr_c) & "..." & time'image(now));
 d := x"00000400";
 for i in 0 to (frame) loop
  at_write_32(address           => apbaddr_c,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput      => vmode,
```

```
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);

  if (i > 5) then
   assert(data_out = frame_1(i - 5))
   report "error: wrong result vector at test 3 and vector " &
   integer'image(i - 5)
   severity error;
  end if;
 end loop;
end if;

-------------------------------------------------------------------------
--                   test 4
-------------------------------------------------------------------------
if do_basic_write_test then
 print("test 4: writing the SECOND valid command at address: " &
 tost(apbaddr_c + 4) & "..." & time'image(now));
 d := x"00000808";
 for i in 0 to (frame) loop
  at_write_32(address           => apbaddr_c + 4,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput      => vmode,
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);
  if (i > 6) then
   assert(data_out = frame_2(i - 6))
   report "error: wrong result vector at test 4 and vector " &
   integer'image(i - 6)
   severity error;
  end if;
 end loop;
end if;

-------------------------------------------------------------------------
--                   test 5
-------------------------------------------------------------------------
if do_basic_write_test then
 print("test 5: writing the THIRD valid command at address: " &
 tost(apbaddr_c + 8) & "..." & time'image(now));
 d := x"00000808";
 for i in 0 to (frame) loop
  at_write_32(address           => apbaddr_c + 8,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput      => vmode,
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);
  if (i > 7) then
   assert(data_out = frame_3(i - 7))
   report "error: wrong result vector at test 5 and vector " &
```

```
   integer'image(i - 7)
   severity error;
  end if;
 end loop;
end if;

-------------------------------------------------------------------------
--                   test 6
-------------------------------------------------------------------------
if do_basic_write_test then
 print("test 6: writing the FOURTH valid command at address: " &
 tost(apbaddr_c + 20) & "..." & time'image(now));
 d := x"00008000";
 for i in 0 to (frame) loop
  at_write_32(address           => apbaddr_c + 20,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput      => vmode,
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);
  if (i > 8) then
   assert(data_out = frame_4(i - 8))
   report "error: wrong result vector at test 6 and vector " &
   integer'image(i - 8)
   severity error;
  end if;
 end loop;
end if;
-------------------------------------------------------------------------
--                   test 7
-------------------------------------------------------------------------
if do_basic_write_test then
 print("test 7: writing the FIFTH valid command at address: " &
 tost(apbaddr_c + 48) & "..." & time'image(now));
 d := x"00000808";
 for i in 0 to (frame) loop
  at_write_32(address           => apbaddr_c + 48,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput      => vmode,
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);
  if (i > 9) then
   assert(data_out = frame_5(i - 9))
   report "error: wrong result vector at test 7 and vector " &
   integer'image(i - 9)
   severity error;
  end if;
 end loop;
end if;
assert false report "testbench ended!" severity failure;
end process;
end architecture;
```

A.9. Stand-Alone Test bench VHDL Code for the Second Stage of the Design

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

library gaisler;
library grlib;
use grlib.stdlib.conv_std_logic_vector;
use grlib.stdlib.conv_integer;
use grlib.stdlib.conv_std_logic;
use grlib.stdlib.tost;
use grlib.stdlib."+";
use grlib.testlib.print;
use grlib.amba.all;
use grlib.at_pkg.all;
use grlib.at_ahb_mst_pkg.all;
use grlib.at_ahb_slv_pkg.all;
use grlib.dma2ahb_package.all;
use work.ac97link_pack.all;
use work.dmac_pack.all;

------------------------
-- entity
entity ac97_tb is
generic(sysperiod_g : integer := 20; --systemperiod in ns
        apbaddr_g   : integer := 16#800#;
        pindex      : integer := 0;
        paddr       : integer := 0;
        pmask       : integer := 16#FFF#;
        console     : integer := 0;
        pirq        : integer := 0;
        parity      : integer := 1;
        flow        : integer := 1;
        fifosize    : integer range 1 to 32 := 32;
        abits       : integer := 8);
end entity ac97_tb;
---------------------------------------
-- architecture
architecture behavioural of ac97_tb is
---------------------------------------
-- component
component ac97top
port (rst               : in  std_logic;
      clk               : in  std_logic;
      bit_clk           : in  std_logic;
      data_in           : in  std_logic;
      ac97top_inl   : in  apb_slv_in_type;
      ac97top_outl  : out apb_slv_out_type;
      ac97dma_in        : in  ahb_mst_in_type;
      ac97dma_out   : out ahb_mst_out_type;
      ac97dmac_inm  : in  at_ahb_mst_out_type;
      debug             : out dmac_debug_type;
      codec_rst         : out std_logic;
      data_out          : out std_logic;
      sync              : out std_logic);
end component;


constant do_basic_read_test   : boolean := true;
```

```
constant do_basic_write_test  : boolean := true;

constant vmode        : boolean := false;
constant sysperiod_c  : time := sysperiod_g * 1 ns;
constant datareg_c    : std_logic_vector(31 downto 0):=uartaddr_c;
constant statusreg_c  : std_logic_vector(31 downto 0):=uartaddr_c+4;
constant ctrlreg_c    : std_logic_vector(31 downto 0):=uartaddr_c+8;
constant scalerreg_c  : std_logic_vector(31 downto 0):=uartaddr_c+12;
constant fifodbgreg_c : std_logic_vector(31 downto 0):=uartaddr_c+16;

constant uartaddr_c   : std_logic_vector(31 downto 0):=
apbaddr_c+(conv_std_logic_vector(paddr,12) and
conv_std_logic_vector(pmask,12));

constant apbaddr_c    : std_logic_vector(31 downto 0):=
conv_std_logic_vector(apbaddr_g, 12) & x"00000";

signal rstn             : std_ulogic := '0';
signal clk              : std_ulogic := '1';
signal apbi             : apb_slv_in_type;
signal apbo             : apb_slv_out_vector :=(others => apb_none);
signal ahbmi            : ahb_mst_in_type;
signal ahbmo            : ahb_mst_out_vector :=(others => ahbm_none);
signal ahbsi            : ahb_slv_in_type;
signal ahbso            : ahb_slv_out_vector :=(others => ahbs_none);
signal atmi             : at_ahb_mst_in_type;
signal atmo             : at_ahb_mst_out_type;
signal dbgi             : at_slv_dbg_in_type;
signal dbgo             : at_slv_dbg_out_type;

signal enablemon        : std_ulogic;
signal irqdetected      : std_ulogic;
signal clearirq         : std_ulogic;

signal deb            : dmac_debug_type;
signal bit_clk        : std_logic;
signal data_in        : std_logic;
signal codec_rst      : std_logic;
signal data_out       : std_logic;
signal sync           : std_logic;

begin
--------------------------------------
-- Component instantiation
--------------------------------------
inst0 : ac97top
port map(clk          => clk,
         rst          => rstn,
         bit_clk      => bit_clk,
         ac97top_inl  => apbi,
         ac97top_outl => apbo(pindex),
         ac97dma_in   => ahbmi,
         ac97dma_out  => ahbmo(1),
         ac97dmac_inm => atmo,
         debug        => deb,
         data_in      => data_in,
         codec_rst    => codec_rst,
         sync         => sync,
         data_out     => data_out);
```

```
--------------------------------------
-- clk generation
--------------------------------------
clk <= not clk after (sysperiod_c/2);
rstn <= '0', '1' after 300 ns;

------------------------------------------------------------
-- AMBA infrastructure
------------------------------------------------------------
ahb0 : at_ahb_ctrl             -- AHB arbiter/multiplexer
generic map (defmast     => 0,
             split       => 1,
             enebterm    => 1,
             ebprob      => 1,
             rrobin      => 1,
             ioaddr      => 16#FFF#,
             hmstdisable => 16#4000#,
             ioen        => 1,
             nahbm       => 3,
             nahbs       => 2,
             hslvdisable => 16#600#,
             enbusmon    => 0,
             assertwarn  => 1,
              asserterr  => 1)
port map (rstn, clk, ahbmi, ahbmo, ahbsi, ahbso);

apb0 : apbctrl                     -- AHB/APB bridge
generic map (hindex      => 0,
             haddr       => apbaddr_g,
             enbusmon    => 0,
             asserterr   => 1,
             assertwarn  => 1,
             pslvdisable => 1,
             nslaves     => 1)
port map (rstn, clk, ahbsi, ahbso(0), apbi, apbo);

dma1 :  at_ahb_mst
generic map(hindex    => 0,
            vendorid  => 0,
            deviceid  => 0,
            version   => 0)

port map(hclk    => clk, -- AMBA AHB system signals
         hresetn => rstn,
         atmi    => atmi, -- Direct Memory Access Interface
         atmo    => atmo,
         ahbi    => ahbmi, -- AMBA AHB Master Interface
         ahbo    => ahbmo(0));

------------------------------------------------------------------------
-- AHB memory
ahbslv0 : at_ahb_slv
generic map (hindex            => 1,
             -- Bank 0 configuration;
             bank0addr         => 16#400#,
             bank0mask         => 16#FFF#,
             bank0type         => AT_AHBSLV_MEM,
             bank0cache        => 1,
             bank0prefetch     => 1,
             bank0ws           => 1,
             bank0rws          => AT_AHBSLV_FIXED_WS,
```

```
            bank0dataload => 0,
            bank0datafile => "none")

port map (rstn => rstn, clk => clk, ahbsi => ahbsi, ahbso => ahbso(1),
            dbgi => dbgi, dbgo => dbgo);

----------------------------------------------------------------
-- process to read or write
----------------------------------------------------------------
test_p : process is
        variable tp             : boolean;
        variable tpcounter      : integer;
        variable d              : std_logic_vector(31 downto 0);
        variable c              : std_logic_vector(31 downto 0);
        variable e              : std_logic;
        variable time0          : time;
        variable time1          : time;
        variable dummy          : boolean;

begin
        print("testbench start");
        at_init(atmi);
        wait until rstn = '1';

-------------------------------------------------------------------
if do_basic_write_test then
 print("volume 2... " & tost(apbaddr_c + 48) & "..." &
 time'image(now));
 d := x"00000099";
 at_write_32(address            => apbaddr_c + 48,
            data                => d,
            waitcycles          => 0,
            lock                => false,
            hprot               => "0011",
            back2back           => true,
            screenoutput        => vmode,
            errorresp           => dummy,
            atmi                => atmi,
            atmo                => atmo);
end if;
-----------------------------------------------------------------------
if do_basic_write_test then
 print("volume 2... " & tost(apbaddr_c + 12) & "..." &
 time'image(now));
 d := x"00000001";
 at_write_32(address            => apbaddr_c + 12,
            data                => d,
            waitcycles          => 0,
            lock                => false,
            hprot               => "0011",
            back2back           => true,
            screenoutput        => vmode,
            errorresp           => dummy,
            atmi                => atmi,
            atmo                => atmo);
end if;
```

```
-------------------------------------------------------------------------
if do_basic_write_test then
 print("volume 2... " & tost(apbaddr_c + 4) & "..." &
 time'image(now));
 d := x"000000f0";
 at_write_32(address           => apbaddr_c + 4,
             data              => d,
             waitcycles        => 0,
             lock              => false,
             hprot             => "0011",
             back2back         => true,
             screenoutput      => vmode,
             errorresp         => dummy,
             atmi              => atmi,
             atmo              => atmo);
end if;
wait for 100000 ns;
-------------------------------------------------------------------------
-- writing into the memory
-------------------------------------------------------------------------
if do_basic_write_test then
 print("test 1: writing...." & time'image(now));
 for i in 0 to 35 loop -- 262143
  d := conv_std_logic_vector(26 + i, 32);
  at_write_32(address           => x"40000000" + i*4,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput  => vmode,
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);
  print("written data at address " & tost(x"40000000" + i*4) & " is "
  & tost(d));
 end loop;
end if;
wait for 1500 ns;
-------------------------------------------------------------------------
if do_basic_write_test then
 print("start fetching... " & tost(x"80") & "..." & time'image(now));
 d := x"40000000";
 at_write_32(address           => apbaddr_c + x"80",
             data              => d,
             waitcycles        => 0,
             lock              => false,
             hprot             => "0011",
             back2back         => true,
             screenoutput      => vmode,
             errorresp         => dummy,
             atmi              => atmi,
             atmo              => atmo);
end if;
wait for 1000000 ns;
```

```
-------------------------------------------------------------------------
-- writing into the memory
-------------------------------------------------------------------------
if do_basic_write_test then
 print("test 2: writing...." & time'image(now));
 for i in 0 to 25 loop
  d := conv_std_logic_vector(46 + i, 32);
  at_write_32(address           => x"40008000" + i*4,
              data              => d,
              waitcycles        => 0,
              lock              => false,
              hprot             => "0011",
              back2back         => true,
              screenoutput      => vmode,
              errorresp         => dummy,
              atmi              => atmi,
              atmo              => atmo);
  print("written data at address " & tost(x"40000000" + i*4) & " is "
  & tost(d));
 end loop;
end if;
wait for 1500 ns;

-------------------------------------------------------------------------
if do_basic_write_test then
 print("start fetching... " & tost(x"80") & "..." & time'image(now));
 d := x"40008000";
 at_write_32(address           => apbaddr_c + x"80",
             data              => d,
             waitcycles        => 0,
             lock              => false,
             hprot             => "0011",
             back2back         => true,
             screenoutput  => vmode,
             errorresp         => dummy,
             atmi              => atmi,
            atmo              => atmo);
end if;

wait for 1000000 ns;
assert false report "testbench ended!" severity failure;

end process;

end architecture;
```

A.10. System Test C Code for The First Stage of The Design

```
#include <stdio.h>

int main(void){

  int i, aux, aux2, j, piv;
  printf("\nTESTING: AC97 Controller \n");

  volatile unsigned int *reg0 = (unsigned int*)0x80000A00;
  *reg0 = 0x0;
  printf("reset.... \n");

  volatile unsigned int *reg1 = (unsigned int*)0x80000A08;
  *reg1 = 0x0;
  printf("headphone volume.... \n");

  volatile unsigned int *reg2 = (unsigned int*)0x80000A30;
  *reg2 = 0x0;
  printf("pcm out volume.... \n");

  volatile unsigned int *reg3 = (unsigned int*)0x80000A54;
  *reg3 = 0xf0f;
  printf("PC Beep.... \n");

  volatile unsigned int *reg4 = (unsigned int*)0x80000A58;
  *reg4 = 0x800;
  printf("PC Beep.... \n");

  volatile unsigned int *reg5 = (unsigned int*)0x80000AC0;
  *reg5 = 0x0;
  printf("PC Beep.... \n");

  aux = 0x80000A08;

  printf("\n");
  for (j = 0; j <= 3855; j = j + 257) {
    volatile unsigned int *reg = (unsigned int*)(aux);
    *reg = j;
    printf("volume DOWN... = %x \n", *reg);
  }
  printf("\n");

  for (j = 3855; j >= 0; j = j - 257){
    volatile unsigned int *reg = (unsigned int*)(aux);
    *reg = j;
    printf("volume UP... = %x \n", *reg);
  }
  printf("\n");

  volatile unsigned int *reg = (unsigned int*)(aux);
  *reg = 0x0;
  printf("\nMAX volume...\n\n");

  aux2 = 0x80000A58;
  for (j = 40092; j >= 32; j = j - 257){
    volatile unsigned int *reg = (unsigned int*)(aux2);
    *reg = j;
    printf("sweeping frequencies... = %x \n", *reg);
  }
  printf("\n");
```

```
  for (i = 0; i <= 2; i++){
    for (j = 0; j <= 3871; j = j + 257) {
       volatile unsigned int *reg = (unsigned int*)(aux2);
       *reg = j;
       printf("sweeping frequencies... = %x \n", *reg);
    }
    printf("\n");

    for (j = 3871; j >= 0; j = j - 257) {
       volatile unsigned int *reg = (unsigned int*)(aux2);
       *reg = j;
       printf("sweeping frequencies... = %x \n", *reg);
    }
    printf("\n");
  }

  *reg = 0xFFFF;
  printf("\nPC beep OFF \n");
  printf("\nAC97 Controller TEST finished!!! \n");
  return 0;
}
```

A.11. System Test C Code for The Second Stage of The Design

```
#include <stdio.h>

static unsigned int array[] = {};

int main(void){

        int i, j;
        printf("\nTESTING: AC97 Controller \n\n");

        //----------- INITIALIZING AUDIO REGISTERS --------------
        volatile unsigned int *reg0 = (unsigned int*)0x80000A00;
        *reg0 = 0x0;
        printf("Reset.... \n");

        volatile unsigned int *reg1 = (unsigned int*)0x80000A08;
        *reg1 = 0x0;
        printf("Headphone Volume ON.... \n");

        volatile unsigned int *reg2 = (unsigned int*)0x80000A30;
        *reg2 = 0x0;
        printf("PCM Out Volume ON.... \n");

        for (i = 0; i < 3; i++){ printf(" waiting... \n"); }

        //------- WRITING RAW DATA INTO MEMORY -------------------

        for (i = 0; i < 12; i++){ array[i] = 0xFFFF0000 + i; }

        volatile unsigned int *ptr0 = (unsigned int*)0x80000A80;
        *ptr0 = (unsigned int)array;
        printf("Start Fetching Data at Address: %x \n", array);

        for (i = 0; i < 24; i++){ printf(" waiting... \n"); }

        //--------- WRITING RAW DATA INTO MEMORY ------------------

        for (i = 0; i < 24; i++){ array[i] = 0xFFFF0000 + i; }

        volatile unsigned int *ptr1 = (unsigned int*)0x80000A80;
        *ptr1 = (unsigned int)array;
        printf("Start Fetching Data at Address: %x \n", array);

        for (i = 0; i < 48; i++){ printf(" waiting... \n"); }

        printf("\nAC97 Controller TEST finished!!! \n\n");
        return 0;

}
```