Prof. Dr.-Ing. L. Leutelt

Digital Systems

Lab

March 16, 2022

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Dieses Dokument wurde mit Hilfe von KOMA-Script und LaTeX gesetzt.

Contents

Chapter 1

Synchronous Serial Communication with an Audio Codec

1.1 Objectives of the first lab Session

You will learn in this lab session, how to develop a serial communication interface to another digital module and how to exchange data with an audio codec. In addition to the modules known from the lecture, such as frequency divider, finite state machine (FSM) to control sampling timing, serial-to-parallel converter, etc. you will design a VHDL model for a parallel-serial converter, simulate, verify and test it on an FPGA. In the end, you will be able to send analog audio signals through the system consisting of audio codec and FPGA and hear them again as analog output, initially unchanged except for quantization.

The focus will be on (re)learning the development tools Mentor ModelSim and Xilinx Vivado and investigating the timing behavior of the circuit by measuring the analog and digital input and output signals with the oscilloscope.

Preparation time: approx. 4-6 hours

1.2 Audio codec subsystem overview

The module `AUDIO_CODEC_COM`, which is to be implemented in the FPGA, establishes the data exchange with the audio codec PCM3006, which is located on an expansion board (see Figure 1.1). The block diagram in Figure 1.2, which was already presented in the lecture, shows the individual components of the subsystem: frequency divider, serial-to-parallel converter, parallel-to-serial converter, and FSM for controlling the sampling times. The frequency divider provides three clock signals SYSCLK, BCK and LRC for the audio codec. LRC determines the sampling rate to be set to approximately 48 kHz, BCK is the bit clock, and SYSCLK is the clock for oversampling the internal filters in the audio codec. The frequency ratios of these three clocks are given in Table 1.1.

All three clocks are derived from the system clock CLK, which is provided by an on-board oscillator of the Modsys 2.0 board to the Artix-7 FPGA (see Figure 1.8). This system clock is set to 100 MHz,

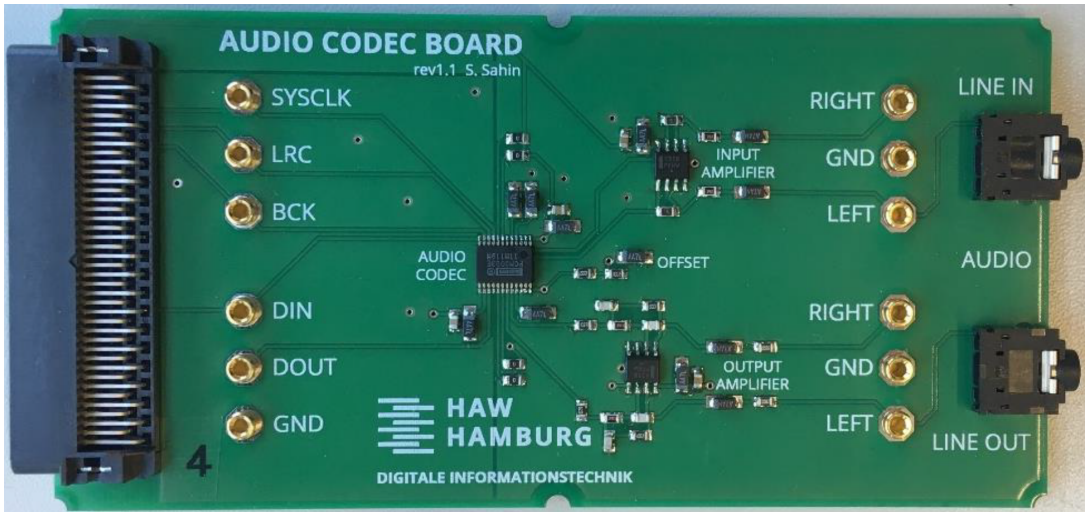


Figure 1.1: Expansion board "'AUDIO CODEC BOARD"' with the audio codec PCM3006.

and the derived clocks SYSCLK, BCK, LRC are generated inside the FPGA by dividing the system clock frequency by appropriate powers of two.

Prep Task 1

If you use the system clock from the oscillator of the Modsys 2.0 board ($f_{\mathrm{SYSCLK}} = 100\ \mathrm{MHz}$), you obviously cannot exactly set a sampling frequency 48 kHz, because the ratio of the two frequencies is not exactly a power of two.

- What sampling frequency $\hat{f}_{\mathrm{LRC}}$ can you set that is closest to f_{LRC}=48 kHz?
- Calculate the relative error $\epsilon_{\mathrm{rel}} = (f_{\mathrm{LRC}} - \hat{f}_{\mathrm{LRC}})/f_{\mathrm{LRC}}$.
- What ratio $f_{\mathrm{CLK}}/f_{\mathrm{SYSCLK}}$ did you assume?

In the following, the individual parts of the AUDIO_CODEC_COM module are briefly introduced once again. Further details can be found in the lecture slides and the source code in EMIL.

Frequency divider: CODEC_CLK_GEN The three clock signals LRC, BCK and SYSCLK are generated by this module. The ratio $P = f_{\mathrm{CLK}}\ / f_{\mathrm{SYSCLK}}$ is set in the prescaler module FRQ_PRE_DIV, which enables the internal counter in the frequency divider CODEC_CLK_GEN only in every P-th cycle ($P = 2^p$) of the system clock CLK.

Timing of actions: AUDIO_CDC_FSM The main task of this module is to control the timing of

Clock Signal	Frequency
FPGA system clock f_{CLK}	100 MHz
Audio codec oversampling f_{SYSCLK}	$256 \cdot f_{\mathrm{LRC}}$
Bit clock f_{BCK}	$32 \cdot f_{\mathrm{LRC}}$
Sampling clock f_{LRC}	approx. 48 kHz

Table 1.1: Frequency ratios of the clock signals occurring in the FPGA system.

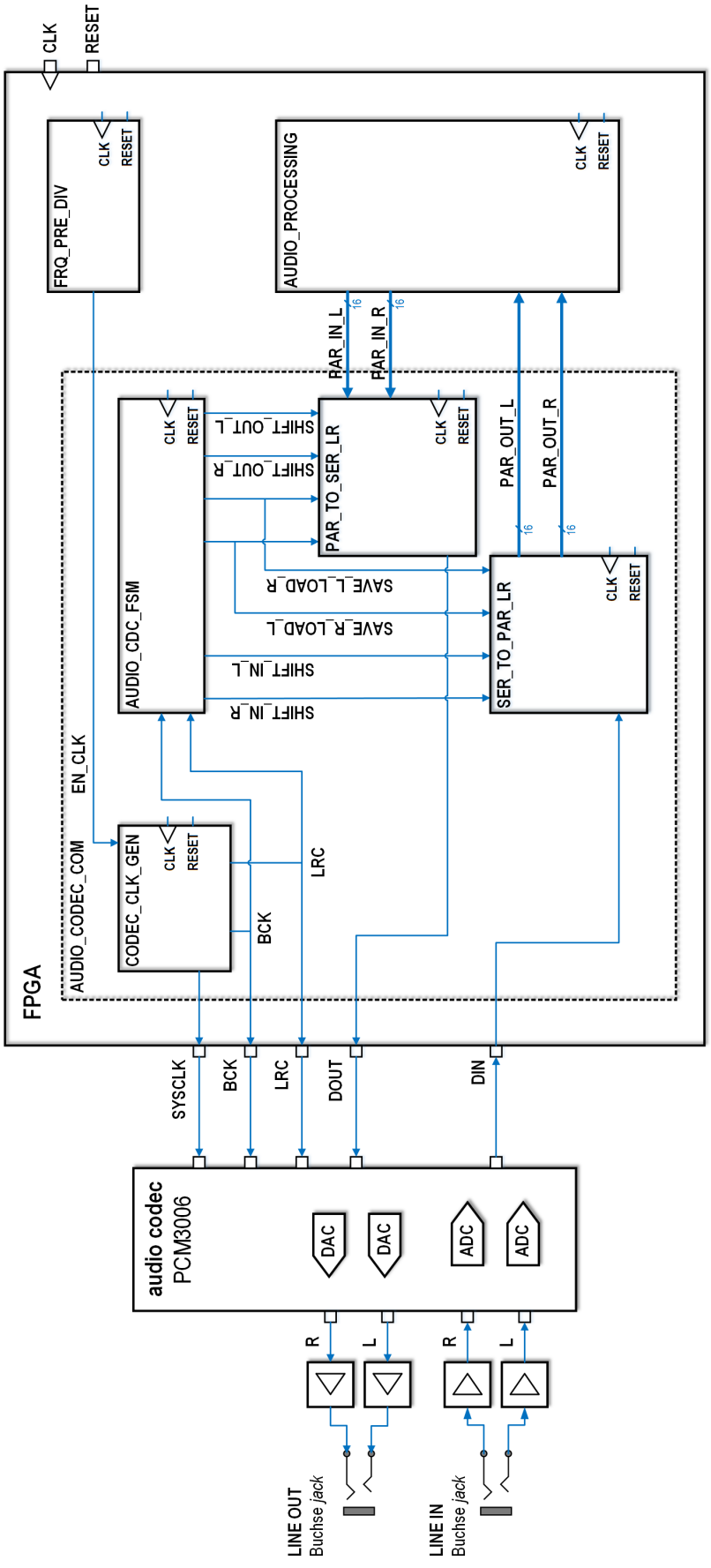


Figure 1.2: Block diagram of the overall system.

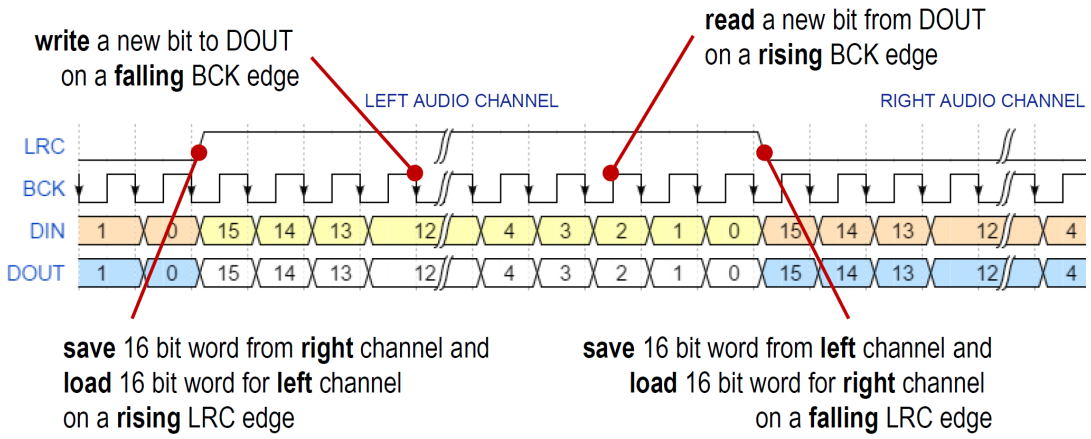


Figure 1.3: Important points in time during serial-to-parallel and parallel-to-serial conversion.

serial-to-parallel and parallel-to-serial conversion. In particular, a regular Finite State Machine also controls the times at which bits are written to and read from the audio codec. Figure 1.3 shows this as an example for the left channel. Table 1.2 indicates which actions in the converters are triggered by the signals.

Serial-to-parallel converter SER_TO_PAR_LR This module consists of one shift register per channel L/R, into which bits arriving at input DIN are shifted with each rising edge of BCK. If all 16 values of an audio sample have been read, the content of the respective shift register is copied into an associated parallel register. There the sample value remains stored and is available for further processing until it has to make room for a new value. The value of the signal LRC determines which pair of shift and parallel registers (for left or right channel) is currently updated.

Parallel-to-serial converter PAR_TO_SER_LR It is your task to develop this module in this lab session. The module converts a 16-bit audio value calculated by the AUDIO_PROCESSING module into a serial data stream and starts outputting it at output DOUT with each falling edge of BCK. The module does not need a parallel register, since we can assume that the AUDIO_PROCESSING module will provide the processed audio signal values in time, and the values can be read directly into the shift register. Only one shift register shall be used for both channels L/R, in contrast to the serial-to-parallel converter. Remember: you have already started deriving a block diagram for a parallel-to-serial converter in the lecture.

1.3 Development of the parallel-to-serial converter

1.3.1 VHDL template PAR_TO_SER_LR_DUMMY.vhd

All VHDL modules necessary for communication with the audio codec are provided in EMIL. For the development of the parallel-to-serial converter PAR_TO_SER_LR the file PAR_TO_SER_LR_DUMMY.vhd shall be used. This consists of the correct entity and architecture declarations, but without a meaningful behavioral description. Nevertheless, the module as well as the overall system can be simulated and synthesized.

Signal	Meaning
SHIFT_IN_L, SHIFT_IN_R	shift a bit at input DIN (both L and R channel) into serial-to-parallel converter (SER_TO_PAR_LR)
SHIFT_OUT_L, SHIFT_OUT_R	shift a bit out of parallel-to-serial converter (PAR_TO_SER_LR) to output DOUT (both L and R channel)
SAVE_R_LOAD_L	SER_TO_PAR_LR: copy content of shift register into parallel register (R channel) PAR_TO_SER_LR: load new word into shift register (L channel)
SAVE_L_LOAD_R	SER_TO_PAR_LR: copy content of shift register into parallel register (L channel) PAR_TO_SER_LR: load new word into shift register (R channel)

Table 1.2: Meaning of output signals of Finite State Machine AUDIO_CDC_FSM.

Input and output signals of the PAR_TO_SER_LR module to be developed are shown in Figure 1.4 and explained in Table 1.2 and the timing diagram in Figure 1.5.

The source code of PAR_TO_SER_LR_DUMMY.vhd is shown in listing 1.1.

Listing 1.1: Template for the VHDL model PAR_TO_SER_LR_DUMMY.vhd to be developed.

```
1  library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
3
   entity PAR_TO_SER_LR is
5  generic(W : integer := 16);
   port(
7     CLK, SRESETN: in std_logic;
      PAR_IN_L, PAR_IN_R: in std_logic_vector(W-1 downto 0);
9      -- control signals for shifting bit out
      SHIFT_OUT_L, SHIFT_OUT_R    : in std_logic;
11     -- control signals for load in parallel word
       SAVE_R_LOAD_L, SAVE_L_LOAD_R    : in std_logic;
13    SER_OUT: out std_logic
      );
15 end PAR_TO_SER_LR;

17 architecture BEHAVIORAL of PAR_TO_SER_LR is
```

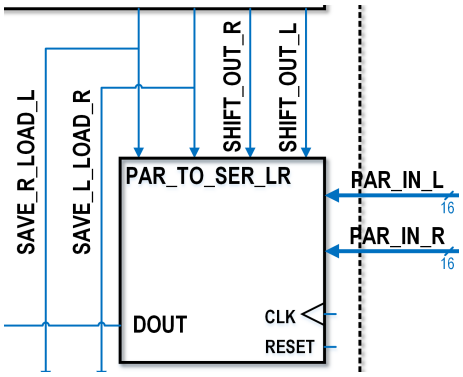


Figure 1.4: Ein- und Augsgangssignale des zu entwickelnden Moduls PAR_TO_SER_LR.

```
19 -- single shift reg for both channels
   signal SHIFT_REG_NEXT,SHIFT_REG : std_logic_vector( W-1 downto 0);
21
   begin
23
   -- process defining 1 shift reg
25 P2S_REGS: process(CLK)
   begin
27    if CLK = '1' and CLK'event then
         if SRESETN = '0' then
29          SHIFT_REG <= (others=>'0') after 2 ns;
           else
31          SHIFT_REG <= SHIFT_REG_NEXT after 2 ns;
         end if;
33    end if;
   end process;
35
   ------------------------------------------------------------------
37 -- insert your code below for next-state logic and output logic
   ------------------------------------------------------------------
39 P2S_COMBIN: process(SHIFT_REG) -- modify sensitivity list as needed
   begin
41    -- delete next lines and replace by your code
      SHIFT_REG_NEXT <= not SHIFT_REG after 2 ns;
43    SER_OUT <= SHIFT_REG(0) after 2 ns;

45 end process;

47 end BEHAVIORAL;
```

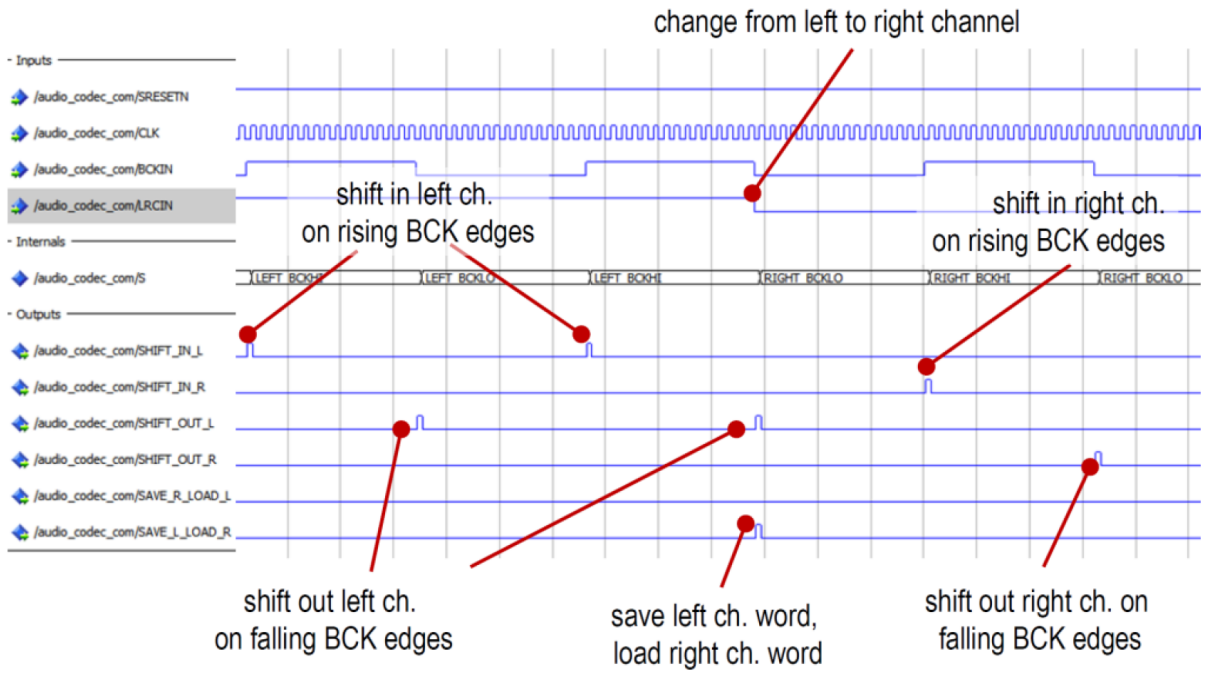


Figure 1.5: Timing of the control signals for the PAR_TO_SER_LR parallel-serial converter.

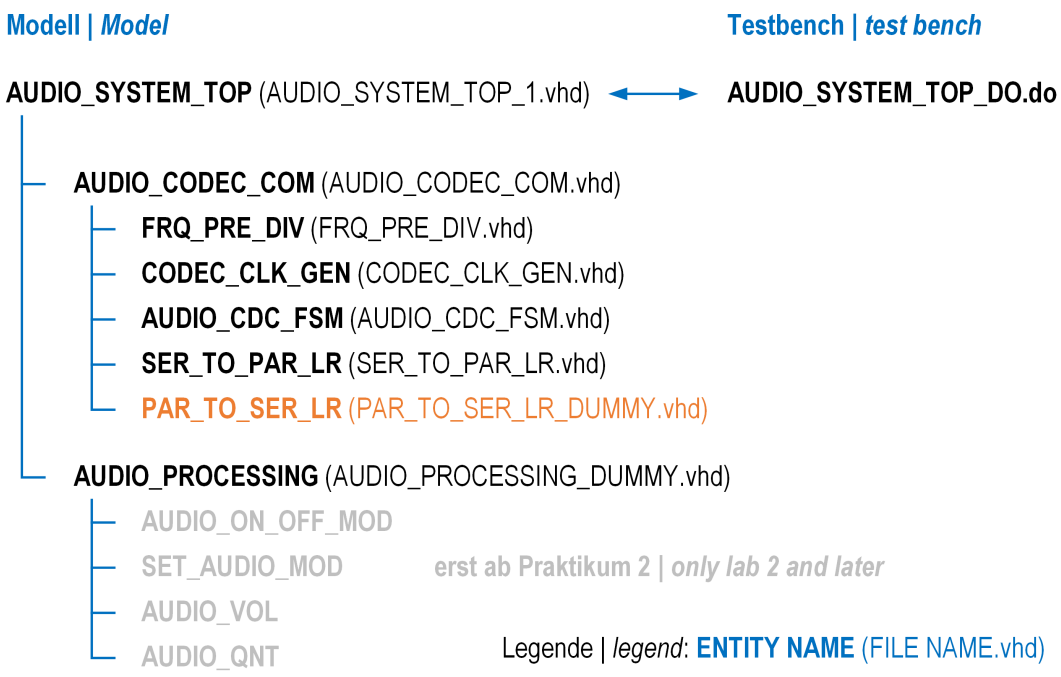


Figure 1.6: Hierarchical structure of the modules.

1.3.2 Hierarchical VHDL design

The VHDL design is in general hierarchical, i.e. the functionality is divided into several modules, some of which are subdivided into several modules again. The top-level module AUDIO_SYSTEM_TOP instantiates the modules AUDIO_PROCESSING and AUDIO_CODEC_COM, each of which instantiates others. The files available in EMIL show the exact VHDL syntax for how to include so-called components. Overall, you can assume that the provided modules all instantiate correctly, as well as correctly compile, simulate, and synthesize. Figure 1.6 shows the design hierarchy of the modules.

A number of port signals are commented out, which are only needed in later lab session. The AUDIO_PROCESSING module in the AUDIO_PROCESSING_DUMMY.vhd file is merely a placeholder and simply forwards the PAR_IN_L or PAR_IN_R input signals unchanged to PAR_OUT_L or PAR_OUT_R, respectively.

With the file AUDIO_SYSTEM_TOP_DO.do a simple testbench is provided. It simulates the complete system with a simple input bit sequence. With this you can test if you have included all files correctly in ModelSim. Of course the output signal DOUT does not show a meaningful signal shape yet, because you still have to design your own parallel-to-serial converter. Adjust this testbench for your needs, i.e. change the input sequences with the `force` command and add more signals from the PAR_TO_SER_LR module to the waveform viewer with the `add wave` command. Further changes are not necessary in the do. file.

Adding internal signals from modules instantiated as components by other modules is done by a 'path specification'. For example,

```
add wave INST_AUDIO_CODEC_COM/INST_SER_TO_PAR_LR/SHIFT_REG_R
```

adds the current value of the shift register signal SHIFT_REG_R of the serial-to-parallel converter to the waveform viewer. The path consists of the names of the instances of the components leading to the shift register. Note that the module name (entity name) and the name of the instances of this module must always be different (so don't use the entity name!). You can get these names from the VHDL source code or more comfortably from the instance browser of ModelSim (see Figure 1.7). After starting the simulation, click on the `sim` tab, then select the desired component in the `Instance` window and then the desired signal in the `Objects` window. Select it and drag it to the `Transcript` window, where the text path to be copied to the .do file appears.

Prep Task 2

Design a behavioral model in VHDL of the PAR_TO_SER_LR parallel-to-serial converter and verify it.

- First, simulate the overall system as it was provided to you – incomplete yet, but ready for simulation. If you succeed in the simulation without errors, start modeling the parallel-to-series converter.
- Use the file PAR_TO_SER_LR_DUMMY.vdh as starting point and use the the block diagram from the lecture as a guideline for implementation.
- Please mind, the module should have only one shift register for both channels and no parallel registers.
- Write 3 (non-trivial) test cases that will give you confidence that your parallel-to-serial converter will work. One test case
  - has a meaningful title (for your manager who is responsible for the project) and

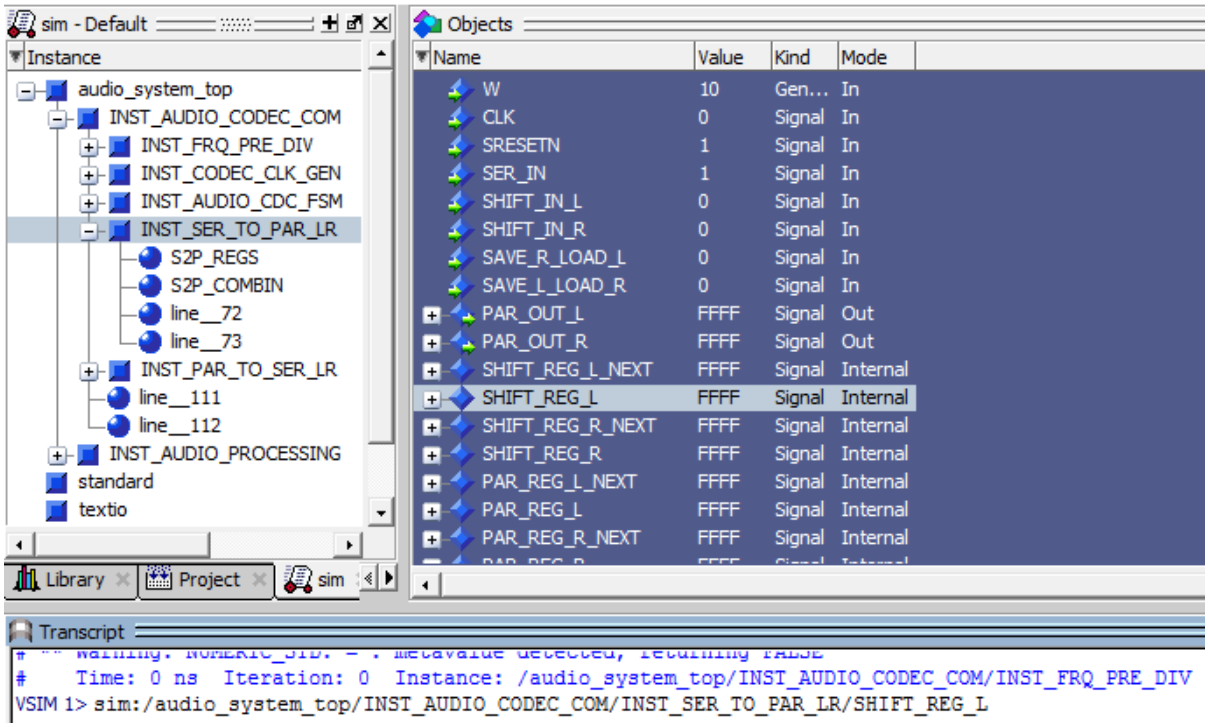


Figure 1.7: Selection of signals of interest for simulation in ModelSim.

- a detailed description of the scenario (for your intern who is writing the testbench).
- This includes describing what you need to observe in the waveform viewer to know that the test case has passed.

- Simulate at least these 3 test cases and copy per test case a relevant part of your ModelSim simulation (waveforms) into the preparation sheet, based on which you explain why the test passed (if necessary why you think not).
  Please note: Always simulate your module in the overall system. It is duplicate work to write a testbench only for the parallel-to-serial converter, because a good testbench already exists. Better adapt this one appropriately.

Congratulations if you got this far in your preparation. That was quite a bit of work and you had to familiarize yourself with VHDL modeling and tools. Now you can look calmly at the lab session.

1.4 Implementation on FPGA and test

If the functional simulation was successful, we can move on to implementing the circuit on the FPGA. Check out the video tutorial or the tutorial in EMIL on how to synthesize with Xilinx Vivado, place & route and configure the device. The FPGA used is a Xilinx Artix-7 XC7A100T-2CSG324 on the ModSys 2.0 evaluation board (see Figure 1.8). The board provides a 100 MHz oscillator with clock divider as source for the system clock and allows to connect peripheral boards via four PCB connectors. .xdc files are used to define the mapping between FPGA pins and peripheral module devices.

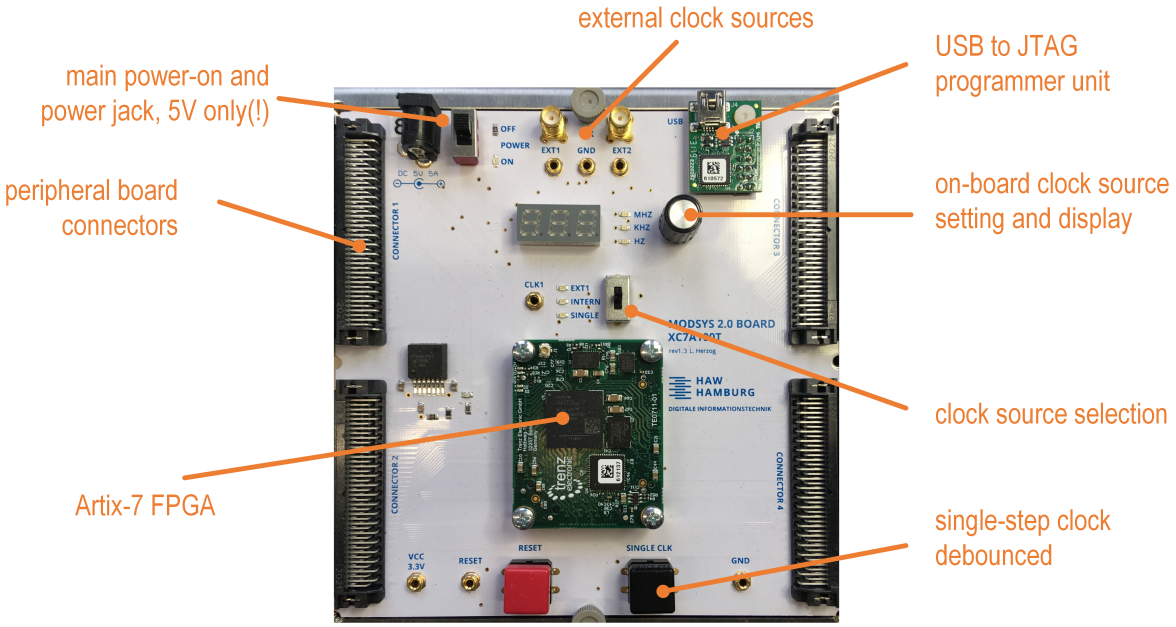


Figure 1.8: ModSys2.0 board with Artix-7 FPGA for the implementation of the audio system.

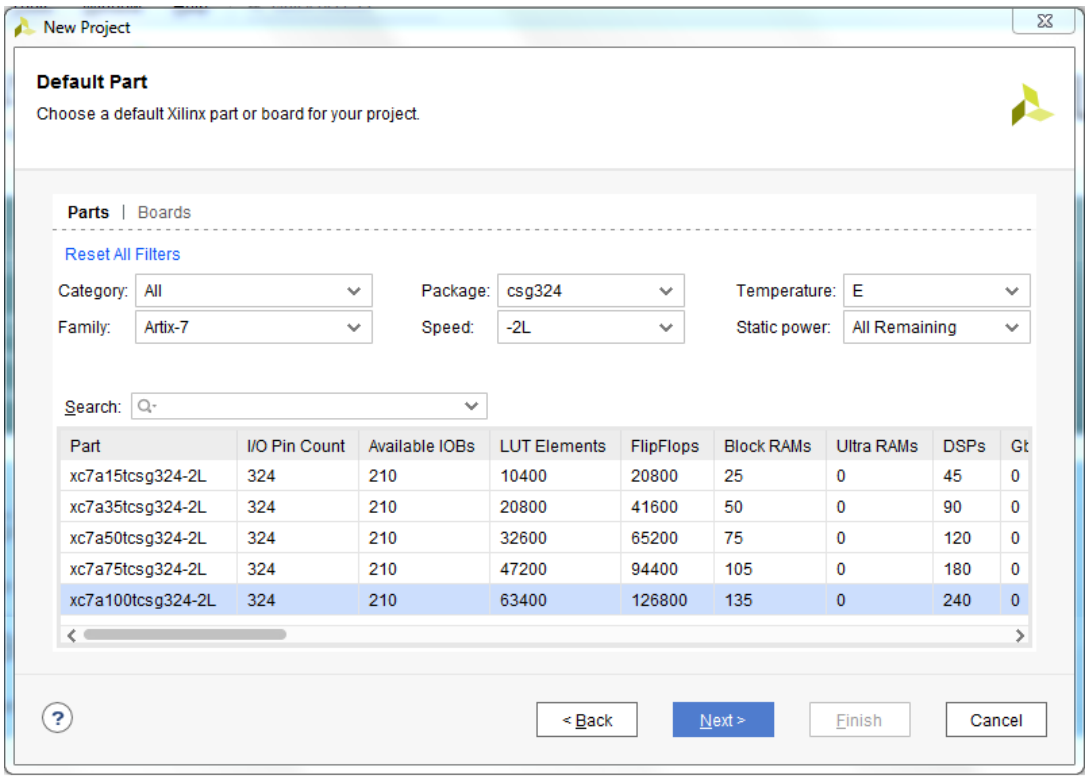


Figure 1.9: Settings in Xilinx Vivado for the XC7A100T.

When creating a new project you have to select the FPGA settings as shown in Figure 1.9. Connect the AudioCodecBoard to the lower connector, the JTAG cable for programming to the JTAG socket and the power supply to the socket left of the "'Power On Switch"'. Turn on the power with this switch before configuring (programming) with Vivado. The configuration of Xilinx FPGAs is volatile, i.e. is lost when the board is switched off. Only reconfiguration helps, unless the configuration was stored in the external memory and the FPGA is configured to load it automatically from the memory at power-on reset (we do not need this function in the lab).

The analog inputs of the AudioCodecBoard (LINE IN) have been specified for so called *line levels* with max. $\pm 1,4\mathrm{V}$ amplitude.

Adjust the function generator so that these levels are not exceeded (always check with the oscilloscope) to prevent damage to the hardware.

Lab Task 1: Test of the system with a sinusoidal signal

With this test you can see if you have implemented everything correctly. The input signal at LINE IN should appear as a time-shifted version at the output LINE OUT when measured with the oscilloscope. If this is not the case, first see if you have configured the FPGA successfully (*Programming succeded*) and the power is on at the Modsys 2.0 board. If the output still does not meet the expectation, continue first with Lab Task 2 for troubleshooting and return later.

- Apply a sinusoidal voltage of e.g. 1 V and 1 kHz to the LINE IN input. Check the

amplitude and frequency with the oscilloscope before connecting the function generator. Display both, the input and output signal, on the oscilloscope.

- If the output signal was as expected, slowly turn up the frequency of the sinusoidal signal to 30 kHz. Describe what you observe and name some characteristic frequency and amplitude values.

Lab Task 2: Analysis of serial bit streams

In this task, we look at the serial, digital data that is exchanged between the FPGA and the audio codec. If Lab Task 1 did not bring the desired results, here is the possibility to make an error analysis based on the digital data. Maybe your VHDL model is correct, but an amplifier or the audio codec is broken...

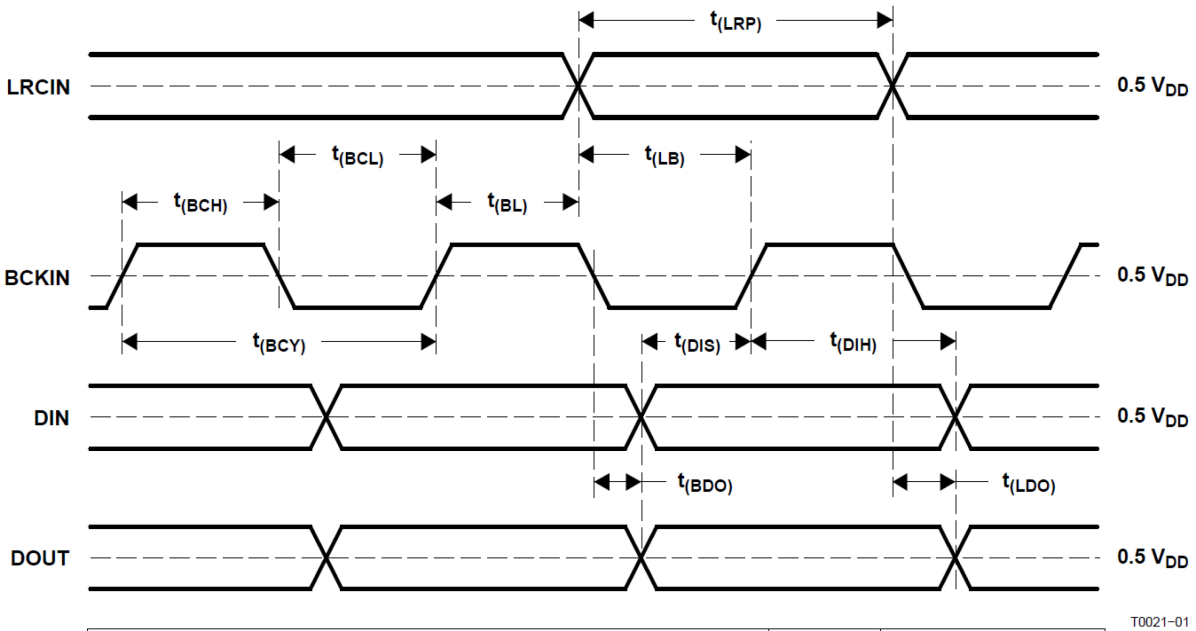
- Apply a sinusoidal voltage of e.g. 1 V and 20 kHz to the LINE IN input. Take a complete data frame of DIN and DOUT together with the BCK and LRC signal. One data frame corresponds to one complete period of the LRC signal with one sample each for left and right channel.
- Determine the 16 bit sampling values for left and right channel and express them as decimal numbers. Are the values of DIN and DOUT the same? Explain.
- Record two consecutive data frames. What can you now state about DIN and DOUT?
- Document the results with oscilloscope images (screenshots, but *not* cell phone photos).

The data sheet of the PCM3006 [3] provides information about the timing requirements of the clock signals BCK and LRC as well as the serial data streams DIN and DOUT to be met for error-free communication (see Figure 1.10). In the following tasks, we will check whether our FPGA meets these timing requirements under the given lab conditions.

It should be noted here that the datasheet uses different signal names that are audio codec-centric, while the ones we use are FPGA-centric. Table 1.3 gives the correspondences between the signal names as used in the VHDL code (and as labeled on the Audio Codec Board) and those from the PCM3006 datasheet. It should be noted in particular that DIN and DOUT swap their roles.

Label Audio Codec Board	Datasheet PCM3006
LRC	LRCIN
BCK	BCKIN
DIN	DOUT
DOUT	DIN

Table 1.3: Different signal names on the Audio Codec Board and in the PCM3006 data sheet.



BCKIN pulse cycle time	$t_{(BCY)}$	300 ns (min)
BCKIN pulse duration, HIGH	$t_{(BCH)}$	120 ns (min)
BCKIN pulse duration, LOW	$t_{(BCL)}$	120 ns (min)
BCKIN rising edge to LRCIN edge	$t_{(BL)}$	40 ns (min)
LRCIN edge to BCKIN rising edge	$t_{(LB)}$	40 ns (min)
LRCIN pulse duration	$t_{(LRP)}$	$t_{(BCY)}$ (min)
DIN setup time	$t_{(DIS)}$	40 ns (min)
DIN hold time	$t_{(DIH)}$	40 ns (min)
DOUT delay time to BCKIN falling edge	$t_{(BDO)}$	40 ns (max)
DOUT delay time to LRCIN edge	$t_{(LDO)}$	40 ns (max)
Rising time of all signals	$t_{(RISE)}$	20 ns (max)
Falling time of all signals	$t_{(FALL)}$	20 ns (max)

Figure 1.10: Timing requirements of the PCM3006 audio codec [3].

Lab Task 3: Bit Timing

In this part you are to check whether the specifications of the data sheet for the timing behavior of the signals are met. The signal names according to the left column in table1.3 are used in the following. Measure the following times and compare them with the data sheet.

- BCK pulse cycle time $t_{(BCY)}$ (measure at BCK)
- DOUT delay time to BCK falling edge $t_{(BDO)}$ (measur at BCK and DIN)
- DIN setup time $t_{(DIS)}$ (measure at DOUT and BCK)
- DIN hold time $t_{(DIH)}$, (measure at DOUT and BCK)

Lab Task 4: Delay of the overall system

First measure the time delay between the analog LINE IN input and the LINE OUT output. Then you are to estimate which part of the delay is due to the audio codec and which part is due to the FPGA.

- What is the risk if you measure the delay with a sinusoidal signal?
- Therefore, perform the measurement with a rectangular 1 kHz pulse with a duty cycle of 10% and ± 1 V amplitude. How do you measure the delay you are looking for?
- Explain the shape of the output signal. Is it still rectangular?
- Which part of the delay is due to the audio codec, which due to the FPGA? Use the results of Lab Task 2 for your considerations, too.

Great, you have successfully completed the first lab session! Now you can design simple state machines in VHDL, apply the tools for simulation and synthesis and use function generator and oscilloscope for testing your circuit.