TSIU03: Lab 4 - Audio Codec

Petter Källström, Mario Garrido

September 1, 2021

Abstract

In this lab you will create a sound interface driver, that helps an existing application to communicate with the sound chip WM8731 on the DE2-115 board.

Contents

1 Introduction

We have a provided application for sound manipulation. It reads a stereo signal, manipulates it, and writes it back, using 16-bit parallel samples. We pretend that it's fancy, and call it "`my_fancy_application`".

To convert an analogue input signal into a digital stream of samples, and vice versa back to analogue, the external sound chip WM8731 is used. However, it uses a bit serial interface for the sample streams to and from the FPGA.

To let `my_fancy_application` communicate with the sound chip, we need a translator (driver) between the parallel and serial protocols. This is illustrated in Fig. 1. The driver is called "`SndDriver`", and it is your task to create this driver.

For testing, you must also finish a testbench and use it to verify the driver in Modelsim.

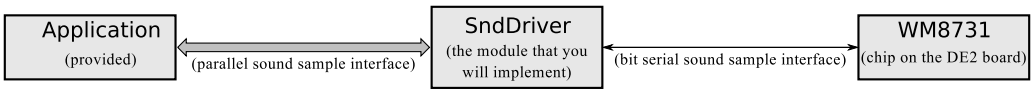


Figure 1: A brief overview of the system, where your task is to create the module in the middle.

2 System Overview

This section describes the surrounding of the `SndDriver`.

The sound communication contains in total four individual channels: Left and right in each direction. Each channel have a sample frequency of $f_s = \frac{f_{clk}}{1024} \approx 48.828$ kHz, where $f_{clk} = 50$ MHz is the usual clock.

2.1 About the Top Module "Sound"

The top module is only a "glue together" unit, depicted in Fig. 2, with the sub modules `my_fancy_application` (or just "Application" for short) and `SndDriver`. They communicate via the bus *SndBus* (several signals).

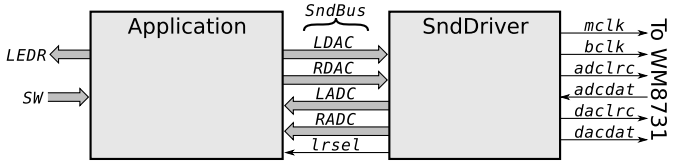


Figure 2: The top module schematics.

You should not do anything with the top module, except set your group number.

2.2 About My_fancy_Application

The module `my_fancy_application` performs some digital sound processing. **This module is already implemented, and you don't have to modify/understand it**. Its functionalities are listed in Table 1.

Forward sound	It passes the incoming sound directly to the output.
Generate right	It generates and adds two sinusoids, 440 and 660 Hz, on the right channel when SW6 is ON.
Generate left	It generates and adds two sinusoids, 440 and 550 Hz, on the left channel when SW7 is ON.
Mute	It mutes the output when SW5 is ON.
Analyse sound	It writes some kind of low pass filtered logarithmic amplitude indicator of the output on the red LEDs.
Noise	(It generates white noise on the inactive `LDAC`/`RDAC`, which must not be heard. What "inactive" means is explained later)

Table 1: Functions in the `my_fancy_application` module.

The generated sinusoids contains some minor noise, that is acceptable to hear.
Some brief comments about how this module is constructed (just in case you are interested):

- The sinusoid generator is implemented as a piece wise polynomial approximation. Since there are 512 clock cycles per sample, the same module can be reused to generate all three frequencies, using one phase accumulator per frequency.
- The sound analyser is implemented using a squarer, a first order low pass filter (LPF), and then it simply picks the bits from the filter register to generate a thermometer coded dB scale. Simple!
- The white noise is implemented as a linear feedback shift register (LFSR).

2.3 About the SndBus

The SndBus is the parallel interface used by `my_fancy_application`. It contains four 16 bits (signed) sample channels, `LADC`, `RADC`, `LDAC`, `RDAC`, and one control signal, `lrsel`. The channels are left and right samples, in both direction (ADC=incoming, DAC=outgoing).

The left and right channels are not active in the same time. `lrsel` defines which are the selected (active) channel. `lrsel='1'` for left active, and `lrsel='0'` for right active, as depicted in Fig. 3.

In Fig. 3, at times (a), `SndDriver` reads a sample on `LDAC`, turns `lrsel='0'`, and provides a sample on `RADC`. Then `my_fancy_application` detects the fall on `lrsel`, processes the sample on `RADC`, and write the result on `RDAC` within 512 clock cycles. At time (b), `SndDriver` reads the sample on `RDAC` and sends it out to the sound chip (WM8731), turns `lrsel='1'`, and provides a sample on `LADC`. Then `my_fancy_application` detects the rise on `lrsel` etc.

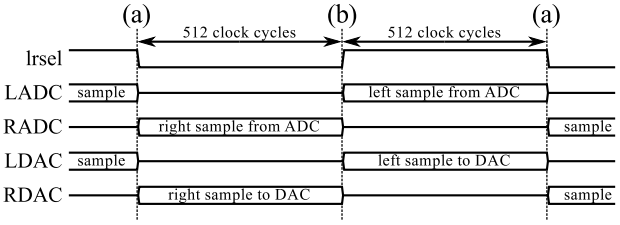


Figure 3: The timing of the SndBus signals during one sample period

2.4 About the Sound Chip WM8731

The WM8731 sound chip is an advanced audio chip. It's main feature is that it has two analogue-to-digital converters (ADCs) to convert an input analogue stereo sound signal into a stream of digital samples, and it has two corresponding digital-to-analogue converters (DACs).

The sound samples are transmitted bit serially via a digital interface, described below.

If you are interested: The WM8731 has several configuration parameters, such as sample frequency or precision, digital interface format, internal amplification/mute/balance etc. Those parameters are set via another digital interface (I^2C). This is done automatically when the DE2-115 are restarted, and nothing you have to care about. You can have a look in the data sheet [1], and consider the following settings: `R5=0x06`, `R7=0x01`, `R8=0x00`. All other gets their default values. Those settings means

- The sample rate aims for $f_s \approx 48$ kSps.
- Two channels means a total sample rate of ≈ 96 kSps.
- The slave mode means that we (on the FPGA) must provide clock and control signals (see the serial interface below).

The sample rate should be 48 kSps (kilo samples per second). For simplicity, we tweak it a little, into $\frac{50\text{ MHz}}{1024} \approx 48.828$ kSps.

2.5 About the Serial Interface

The sound samples are provided bit serially, using a few wires.

The samples are sent in a left-right-left-right-... time interleaved fashion to and from the chip.

With current settings, you should provide ($\Rightarrow$) or read ($\Leftarrow$) the following signals to/from the WM8731 chip:

- `mclk` $\Rightarrow$ A 12.5 MHz master clock. It's the WM8731's internal operation clock.
- `bclk` $\Rightarrow$ A 3.125 MHz bit clock ($\frac{\texttt{mclk}}{4}$).
- `adclrc` $\Rightarrow$ A left/right selector for `adcdat`. `adclrc='1'` for left.
- `adcdat` $\Leftarrow$ Serial bits from the ADCs (one bit per `bclk` pulse).
- `daclrc` $\Rightarrow$ A left/right selector for `dacdat`. `daclrc='1'` for left.
- `dacdat` $\Rightarrow$ Serial bits to the DACs (one bit per `bclk` pulse).

The `adc*` and the `dac*` signals works in the same way:

- Each sample is transferred bit serially.
- For each sample, 32 bits are transferred. The first 16 bits are the sample (MSB first). The remaining 16 bits are unused.
- The transmitter updates the bits on `*dat` at the rising flank of `bclk`.
- The receiver reads the `*dat` at the falling flank of `bclk`.

The `bclk` is $\frac{50\text{ MHz}}{16}$, i.e. 16 clock cycles long. Each sample transfer uses 32 `bclk` cycles, i.e. 512 clk cycles. There are two samples (left + right) upon each `*lrc` period, so the `*lrc` period is 1024 clk cycles long.

In this lab let `adclrc` = `daclrc`, i.e., read and write the right channel data simultaneously, and then the left channel data simultaneously. Note that the received sample is not the same as the transmit sample, so typically `dacdat` $\neq$ `adcdat`.

Finally, it must be mentioned that the 16 bits samples are in signed format, i.e. they can be any integer between -32768 and +32767. This will not affect you much in this lab, since you only need to convert the bits between serial and parallel format. In the project, however, you need to care about the value they represent.

3 Module SndDriver

The module `SndDriver` is a coder/decoder (codec): It translates the audio signal between the parallel format `SndBus`, used by the application, and the bit serial format used by the WM8731 chip. This includes generation of several control signals.

The `SndDriver` must use the `signed` type for samples, and `unsigned` for counters.

The intended structure of `SndDriver` is depicted in Fig. 4. There are two sub modules; `Ctrl` and `Channel_Mod`, and a number of internal signals.

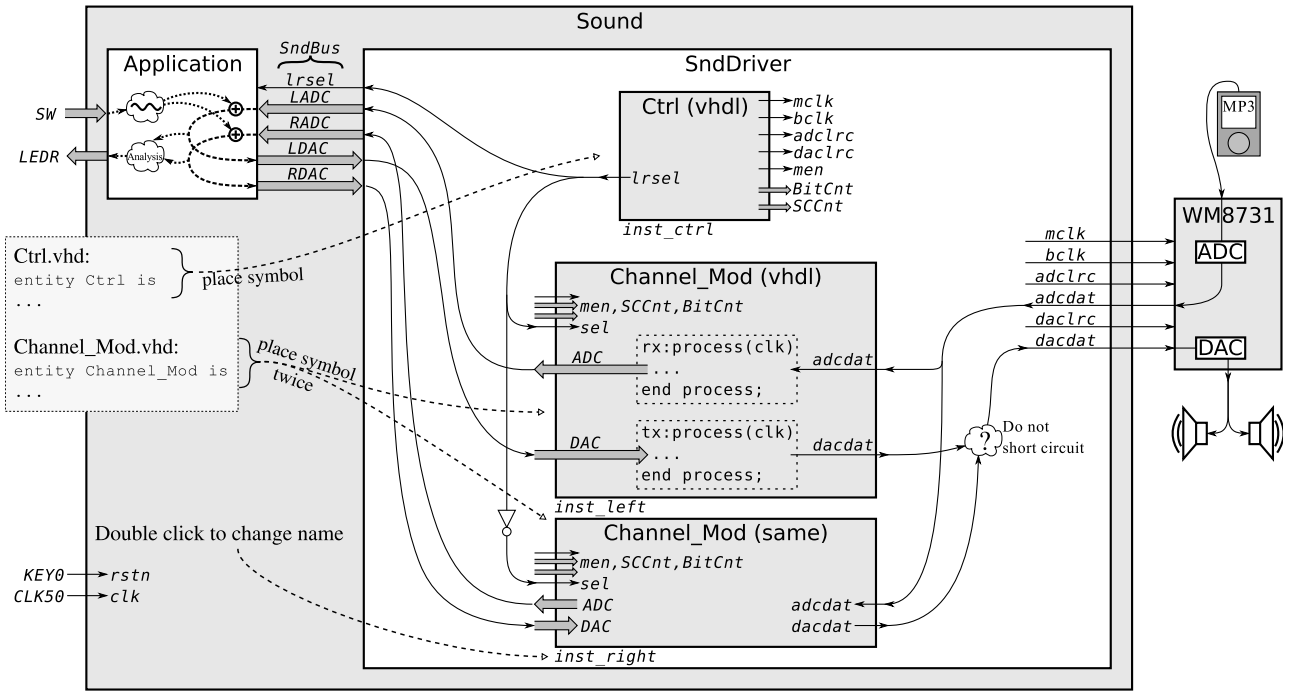


Figure 4: A structural view of `SndDriver` in its environment.

The sub module `Channel_Mod` decodes one bidirectional channel. This is instantiated twice, one instance for the left and one for the right channel.

3.1 Signal Description

The following signals are used as in/out for the module `SndDriver`:

- `clk`, `rstn` ⇒ System clock (50 MHz) and the active low reset.
- `LADC`, `RADC`, `LDAC`, `RDAC`, `lrsel` ⇒ The `SndBus`, as described above.
- `mclk`, `bclk`, `adclrc`, `daclrc`, `adcdat`, `dacdat` ⇒ The serial signals to/from the WM8731 chip. Also described above.

`SndDriver` should also have some internal control signals, generated by `Ctrl` (see Sec. 3.2):

- `men` ⇒ Master Enable signal ('1' the clock cycle before each rising edge of mclk).
- `SCCnt` ⇒ Sub Cycle Counter (counts mclk cycles within each bclk cycle).
- `BitCnt` ⇒ Bit Counter (counts 0 to 31 bits within each sample).

3.2 Control Block (Ctrl)

The system is controlled by a control block, which consists of a 10-bit counter. The control signals for the rest of the system are generated from the bits of the counter.

Figure 5 illustrates a few signals (where the counter is called `cntr`). A timing diagram of all signals over one entire sample is appended in the end of this document. Have a look at it to understand how the signals should work.

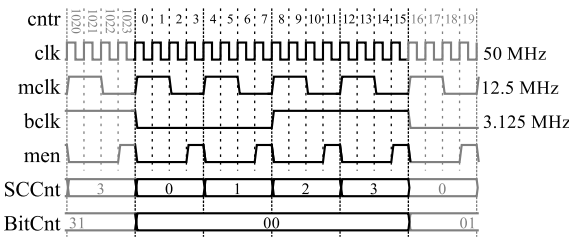


Figure 5: Clock timings.

- `mclk` ⇒ Master clock should be a quarter of `clk` (12.5 MHz). You have this behaviour in `cntr(1)`. Note that when any bit with more significance changes, `cntr(1)` flips from `'1'` to `'0'`, e.g., a falling flank. To get a rising flank behaviour, simply invert the bit: `mclk<=not cntr(1);`.
- `bclk` ⇒ Bit clock should be a quarter of `mclk` (3.125 MHz). Where in `cntr` do you have this behaviour?
- `men` ⇒ Master Enable should be `'1'` just before the rising flank of `mclk`.
- `SCCnt` ⇒ The sub cycle counter counts the four `mclk` pulses within each `bclk` pulse. It is two bits.
- `BitCnt` ⇒ The bit counter counts the 32 `bclk` periods per sample (though only 16 of those are used).
- `adclrc`, `daclrc` ⇒ The left/right clock for the bit serial adc/dac channels. Those should be equal.
- `lrsel` ⇒ The left/right clock for the SndBus channels. This should be inverted to `adclrc`.

Remember: *all* those signals are generated from the bits of `cntr`. To figure out how to generate these signals, you can draw a timing diagram of the counter (using paper and pencil), and it's different bits. After `"11...11"` comes `"00...00"`. Do not draw all 1024 counts, just as many as needed for your understanding.

Another hint is that `adclrc` = `daclrc` = `not lrsel`.

When you are done, generate a symbol file for `Ctrl`, and insert it into the `SndDriver` schematic. Double click the instance name just below the symbol, and name it "`inst_ctrl`".

3.3 Channel_Mod

`Channel_Mod` gets the signal `sel`, which indicates that the SndBus part is active. When `sel='0'`, the bit serial part is active (e.g., shift in/out the bits from/to `adcdat`/`dacdat`).

Remember: There is one `Channel_Mod`, that is instantiated once for left and once for right channel. Hence, in the VHDL code, we don't know if this is the left or the right channel (it will be used for both).

`Channel_Mod`...

- ...needs two shift registers, `RXReg` and `TXReg`, 16 bits each. No other signals are needed.
- ...should contain a process called `rx`, that handles the ADC part (`RXReg`).
- ...should contain a process called `tx`, that handles the DAC part (`TXReg`).
- ...may contain some combinational logic to solve the `dacdat` problem (see below).

Remember (from App. **??**) that the samples are sent MSB first through `adcdat` and `dacdat`.

`RXReg` should, when `sel='0'`, shift in `adcdat` from the right[1] when the `bclk` changes from `'0'` to `'1'` (i.e., when `SCCnt = "01"` and `men`). Only the first 16 bits must be shifted, then it must stop, so no bits of the sample are lost.

`RXReg` should, when `sel='1'`, provide its content on the `ADC` bus (and when not selected, i.e. `sel='0'`, it can do so as well, since it does not matter what is on the bus then).

`TXReg` should, when `sel='0'`, shift out the bits when `bclk` changes from `'1'` to `'0'` (i.e., when `SCCnt = "11"` and `men`), during the first 16 bits - and then it can continue, since it does not matter what value are driven on `dacdat` after that. The MSB of `TXReg` should be available on `dacdat` as soon as `sel='0'`, *NOT* one bit later. Therefore, it is suitable to let `dacdat` *be* the MSB of `TXReg`.

`TXReg` should, when `sel` changes from `'1'` to `'0'` (i.e., the last clock cycle when `sel='1'`), load the value from the `DAC` bus. It does not matter if the module loads data before the last clock cycle of the selected (`sel='1'`) period, as long as it also loads the last clock cycle. So for simplicity, it is easiest to load the register as long as the module is selected, since you then do not need to detect when the last clock cycle is.

The `dacdat` gives a problem. The two instances of `channel_mod` both provides one `dacdat`. The WM8731 chip needs only one. Somehow you have to solve this. From App. **??**, we know that dacdat should come from the left channel when `daclrc='1'` and right otherwise. It feels natural to implement a multiplexor for this. It can however be solved using only an AND or an OR gate, but that requires some extra logic in `Channel_Mod` (what comes out from `Channel_Mod` when `sel='1'`?).

When you are done, generate a symbol file for `Channel_Mod`, and insert it *twice* into the `SndDriver` schematic. Name the two instances "`inst_left`" and "`inst_right`" (double click on the instance names below the instances).

[1] Shift in from the right, so the first incoming bit (MSB) will be shifted all way to the left.

3.4 SndDriver

In SndDriver, you shall place one instance of `ctrl`, two of `channel_mod`, a NOT gate, and your solution to the `dacdat` problem. Remember from Lab1 how to place a NOT gate. **Important for simulation:** Name the wires to e.g. `SCCnt[1..0]` by single clicking them and start typing the name. Also the instances needs clever names (as suggested in Fig. 4).

Note: there seems to be a Quartus bug - simple gates like a two-input OR gate might give Quartus problem when generating HDL file from schematics. You may solve this by creating your own component, like "my_mux" or "my_OR".

4 Your Task

- ▶ Copy the lab skeleton on `K:\TSIU03\Labs\Lab4_Audio\*` to e.g. `X:\TSIU03\Lab4`, and open it.
- ▶ Set your group number in the top module.
- ▶ Open the graphical module `SndDriver`. It is almost empty.
- ▶ Create the VHDL files `ctrl.vhd` and `channel_mod.vhd`, and add them to the project.
- ▶ Implement the modules `ctrl` and `channel_mod`, and connect them in `Snd_Driver`, as described in sec 3.

When done, you should also simulate the `SndDriver`. Follow the instructions in sec. 5 ("Simulation").

Finally, you must also synthesize and verify on the DE2-115 board. You do *not* have to do any pin placement, since this is already done.

5 Simulation

You have to simulate the `SndDriver`, in a way that detects any kind of error you may do. Note that it is just this module that is simulated, not the entire FPGA project.

▶ Generate a VHDL file for the `SndDriver` schematic, and change the `std_logic_vector` into `unsigned` or `signed` where suitable, or you will get "Error loading design" in ModelSim.

▶ Complete the existing test bench "`TB_Audio.vhd`" in the `MSim` folder. Read sec. 5.1, and follow the instructions in the comments in the testbench.

▶ Compile and simulate the test bench and all the VHDL files related to `SndDriver`. Do not add the other VHDL files (Sound.vhd or my_fancy_application.vhd), since you will only simulate the driver. Add signals to the waveform in a colour coded way using `> do wave.do` before the `> run -a`.

5.1 The Test Bench

Have a look at the VHDL file for the test bench. The structure of it is depicted in Fig. 6. You can observe that the test bench architecture contains the parts described below.

A **clock generator** part, that generates a 50 MHz clock, a reset signal, and a `done` signal (after 1 ms).

A **sanity check for the clocks and lrsel**. This part will test the timings of the different clocks, and their relative phases. This is not completed, and your task is to finish it between the comments "`TO FILL IN:`", and "`STOP FILL IN`".

1. Measure the time between two rising edges of the `mclk`.
2. Measure the time between two rising edges of the `bclk`.
3. Measure the time between two rising edges of the `adclrc`.
4. Verify that `mclk=1` and `bclk=0` after the `adclrc` edge.
5. Verify that `adclrc` = `daclrc` ≠ `lrsel` for the rest of the simulation.

A **serial/parallel translator** part. This encodes parallel ADC stimuli signals to the `adcdat`, and decodes `dacdat` into parallel DAC result signals.

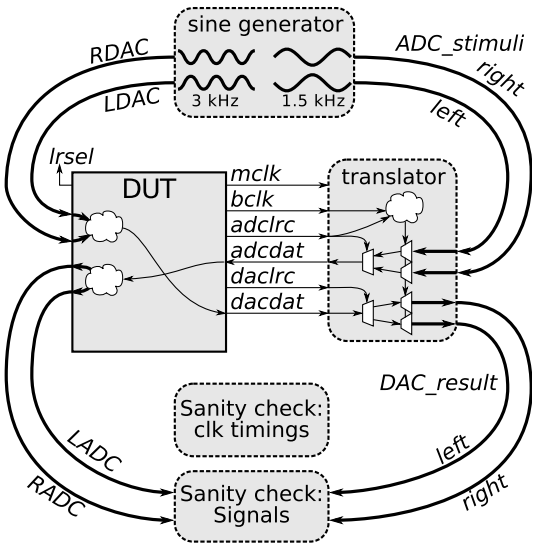


Figure 6: The test bench design.

A **stimuli generator** part. This creates four different sinusoids as digital signals. Two of the signals are 3 kHz tones, passed to the `*DAC` input of the SndDriver. The other two are 1.5 kHz tones, passed to the translators ADC stimuli input.

A **sanity check for the signals**. This verifies that the translator output the same DAC values, as was sent in to the SndDriver. It also verifies that the `*ADC` from the SndDriver is the same as the ADC stimuli.

5.2 A ModelSim Trick

The signals that corresponds to complete samples, represent an analogue level. This is handy to look at. Right click on, e.g., the `ADC` bus, and select "Format" ⇒ "Analog (custom)...". Max = 32767, Min = -32768.

6 Physical Verification

The intended behaviour of the result is specified in Tab. 1. All those functions must work (except the "noise", that must not be heard).

7 Requirements to Pass

General requirements are:

- You must implement and understand the SndDriver.
- You must complete the testbench and use it in a simulation.
- The functions in "`my_fancy_application`" must work (See Table 1). The "Noise" must not be heard.
- You should at least briefly understand the concept of samples, and how they form sound.

When you want to demonstrate, be ready with programmer, waveform, code and understanding.

References

[1] The WM8731 Manual, `K:\TSIU03\DE2_115_Documents\DE2_115_Datasheets\Audio CODEC`

Appendix A Common Errors

Apart from the common VHDL errors, there are some errors that can easily occur:

- **Mistakes in the schematics** ⇒ If you move a module, Quartus tries to move the wires along with it, but often fails to do it in a good way. Make sure you have not unintentionally short circuited anything.
- **Pin mismatch** ⇒ If you change the pins of a sub module, you have to update its symbol file (File→Create/Update→Create Symbol Files for Current File), and the symbol in its "calling" schematic (right click the symbol→update...). Rewire if needed (if pins changed place etc).
- **ADC Shift error** ⇒ You should shift in exactly 16 bits per sample. Not more, not less.
- **DAC Shift error** ⇒ The first bit must be available on `dacdat` as soon as `daclrc` switches, *not* one `bclk` cycle later.

A.1 Malfunctioning Implementation

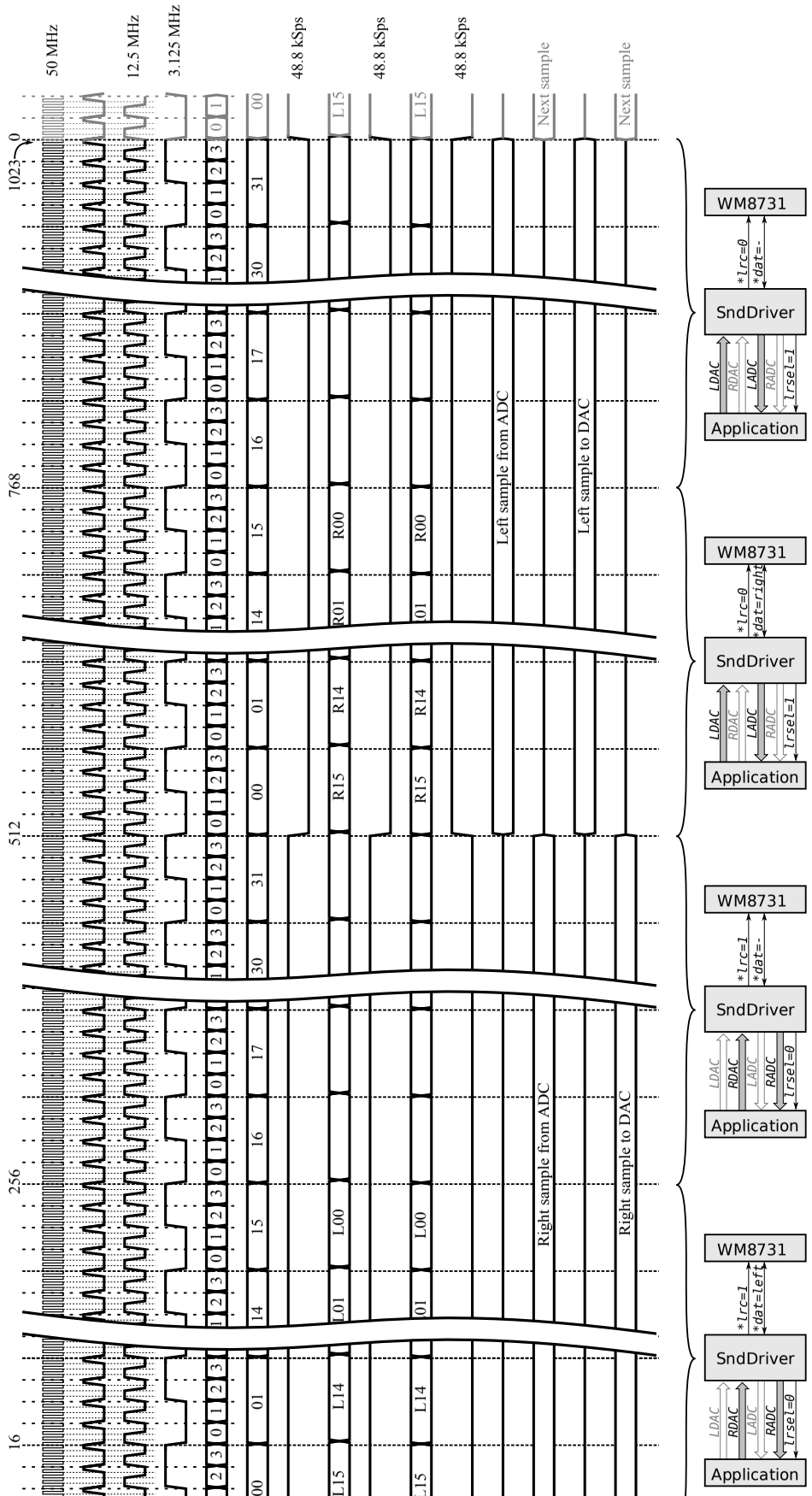
Here are some hints, if everything "should" work, but you don't get the correct result. First of all, verify on the HEX display that it is *your* system running on the FPGA.

Internal error (no LED indication even for internal sound[1])		
Error in the SndBus interface	The signal `lrsel` is not toggling.	⋆
Neither input nor output work (silent, no LEDs except for internal sound[1])		
Error in the WM8731 bus	Check the control signals in a simulation.	⋆
Error in the WM8731 configuration	Turn all switches to 0, then restart the FPGA board.	

Input does not work (no LED indication)		
No input stimuli	Do you feed the input with a sound source?	
Error in the receiver	Check the corresponding code.	⋆
Error in the SndBus interface	Never assigning the `ADC` signal in `Channel_Mod`?	⋆
Output does not work (silent)		
Error in the transmitter	Check the content of the `dacdat` signal.	⋆
Error in the SndBus interface	Do you read the `DAC` signal in `Channel_Mod`?	⋆
Output does not work (white noise)		
Mixing up left/right	You read from the "other" DAC channel in the SndBus.	⋆
Output does not work (strong noise)		
Additional DFF in transmitter	Do not assign `dacdat<=...` in a process...	⋆

[1] "Internal sound" is the sound generated in `my_fancy_application` (that should be indicated on the LED bar).

⋆ Possible to detect in a simulation.


cntr
clk
Men
Mclk
Bclk
scent
bitcnt
adclrc
adcdat
daclrc
dacdat
lrsel
LADC
RADC
LDAC
RDAC
50 MHz
12.5 MHz
3.125 MHz
48.8 kSps
Left sample from ADC
Left sample to DAC
Right sample from ADC
Right sample to DAC
Next sample
WM8731
SndDriver
Application
*lrc=1
*dat=left
*lrc=0
*dat=right
*dat=-
LDAC
RDAC
LADC
RADC
lrsel=0
lrsel=1