

# TSIU03: Lab 4 - Audio Codec

Petter Källström, Mario Garrido

September 11, 2015

## 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 board.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>	<b>5.4</b>	<b>Module SndDriver . . . . .</b>	<b>4</b>
<b>2</b>	<b>Your Task</b>	<b>1</b>	<b>6</b>	<b>Simulation</b>	<b>6</b>
			6.1	The Test Bench . . . . .	6
<b>3</b>	<b>Requirements to Pass</b>	<b>2</b>	6.2	A ModelSim Trick . . . . .	6
<b>4</b>	<b>Common Errors</b>	<b>2</b>	<b>7</b>	<b>The Result</b>	<b>6</b>
4.1	Malfunctioning Implementation . . . .	2	<b>Appendix A</b>	<b>DE2 Overview</b>	<b>7</b>
<b>5</b>	<b>The FPGA Implementation</b>	<b>2</b>	<b>Appendix B</b>	<b>The Sound Chip WM8731</b>	<b>7</b>
5.1	Module Sound: Top Module . . . . .	3	B.1	The Serial Interface . . . . .	7
5.2	The SndBus . . . . .	3	<b>Appendix C</b>	<b>Timings</b>	<b>8</b>
5.3	Module Application . . . . .	3			

## 1 Introduction

In this lab there is a system (illustrated in Fig 1), where you should create the module “SndDriver”. This works as a translator between the sound processing module (the “Application”) and the bit serial interface used by the sound chip on the DE2 board (WM8731).

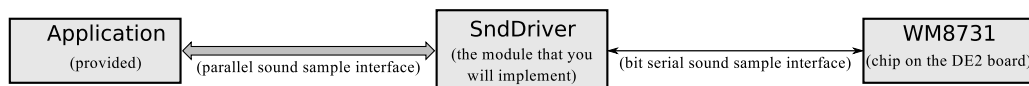


Figure 1: An overview of the system, where your task is to create the module in the middle.

Appendix A describes the used components on the DE2 board.

Appendix B describes the sound chip (including the bit serial sample interface).

Appendix C describes the timing of the control signals in the system.

## 2 Your Task

Your task is to implement the module SndDriver, which is described in Sec. 5.4.

There is a lab skeleton on U:\da\TSIU03\Labs\Lab4\_Audio\\* – Copy this to somewhere on H:\.

In the skeleton there is also a group number generator. Set your group number.

You should also simulate the SndDriver, using a VHDL test bench.

You do *not* have to do the pin placement, since this is done.

### 3 Requirements to Pass

General requirements are given in the “Lab Demonstration” in the FAQ [2].

- You must implement the SndDriver.
- The functions in “Application” must work (See Table 1). The “Noise” must not be heard.
- You must understand your implementation (not the Application module).
- You must complete a testbench and use it in a simulation.

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

### 4 Common Errors

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

- **Mistakes in the schematics**  $\Rightarrow$  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**  $\Rightarrow$  If you change the pins of a sub module, you have to update its symbol file (File $\rightarrow$ Create/Update $\rightarrow$ Create Symbol Files for Current File), and the symbol in its “calling” schematic (right click the symbol $\rightarrow$ update...). Rewire if needed (if pinns changed place etc).
- **ADC Shift error**  $\Rightarrow$  You should shift in exactly 16 bits per sample. Not more, not less.
- **DAC Shift error**  $\Rightarrow$  The first bit must be available on `dacdat` as soon as `daclrc` switches, *not* one `bclk` cycle later.

Other common errors are mentioned in the section “Common Errors” in the FAQ [1].

#### 4.1 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 <sup>1</sup> )	
Error in the SndBus interface	The signal <code>lrssel</code> is not toggling. ★
Neither input nor output work (silent, no LEDs except for internal sound <sup>1</sup> )	
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 <code>Channel_Mod</code> ?. ★
Output does not work (silent)	
Error in the transmitter	Check the content of the <code>dacdat</code> signal. ★
Error in the SndBus interface	Do you read the DAC signal in <code>Channel_Mod</code> ? ★
Output does not work (white noise)	
Mixing up left/right	You read from the “other” DAC channel in the Snd-Bus. ★
Output does not work (strong noise)	
Additional DFF in transmitter	Do not assign <code>dacdat&lt;=...</code> in a process... ★

<sup>1</sup> “Internal sound” is the sound generated in the Application (that should be indicated on the LED bar).

★ Possible to detect in a simulation.

### 5 The FPGA Implementation

The FPGA application is an almost complete sound processing system. The only thing missing is a communication module. This module is your task to complete.

The system clock frequency  $f_{clk} = 50$  MHz, and the sample frequency  $f_s = \frac{f_{clk}}{1024} \approx 48.828$  kHz.

## 5.1 Module Sound: Top Module

The top module is only a “glue together” unit, depicted in Fig. 2, with the sub modules **Application** and **SndDriver**. They communicate via the bus *SndBus* (several signals).

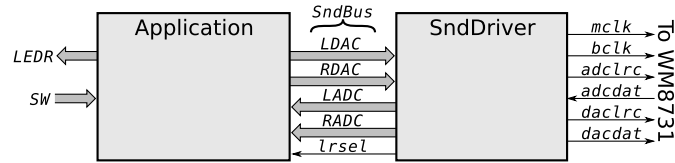


Figure 2: The top module schematics.

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

## 5.2 The SndBus

The SndBus is a parallel interface used in this application. It contains four 16 bits (signed) sample channels, LADC, RADC, LDAC, RDAC, and one control signal, *lrsl*. 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. *lrsl* defines which are the selected channel. *lrsl*='1' for left active, and *lrsl*='0' for right active, as depicted in Fig. 3.

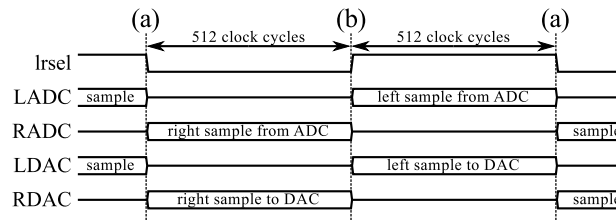


Figure 3: The timing of the SndBus signals during one sample period

In Fig. 3, at times (a), **SndDriver** reads a sample on LDAC, turns *lrsl*='0', and provides a sample on RADC. Then **Application** detects the fall on *lrsl*, 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 *lrsl*='1', and provides a sample on LADC. Then **Application** detects the rise on *lrsl* etc.

## 5.3 Module Application

The module **Application** performs some digital sound processing. **This module is already implemented, and you don't have to modify it.**

The functions implemented in the **Application** are listed in Table 1.

<b>Forward sound</b>	It passes the incoming sound directly to the output.
<b>Generate right</b>	It generates and adds two sinusoids, 440 and 660 Hz, on the right channel when SW6 is ON.
<b>Generate left</b>	It generates and adds two sinusoids, 440 and 550 Hz, on the left channel when SW7 is ON.
<b>Mute</b>	It mutes the output when SW5 is ON.
<b>Noise</b>	It generates white noise on the LDAC/RDAC that is “not used” (and hence will not be heard).
<b>Analyse sound</b>	It writes some kind of low pass filtered logarithmic amplitude indicator of the output on the red LEDs.

Table 1: Functions in the 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 white noise is implemented as a linear feedback shift register (LFSR).
- 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!

## 5.4 Module SndDriver (TODO: Complete this)

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 (un)signed vector types where suitable. Sound samples should be signed, counters should be unsigned.

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.

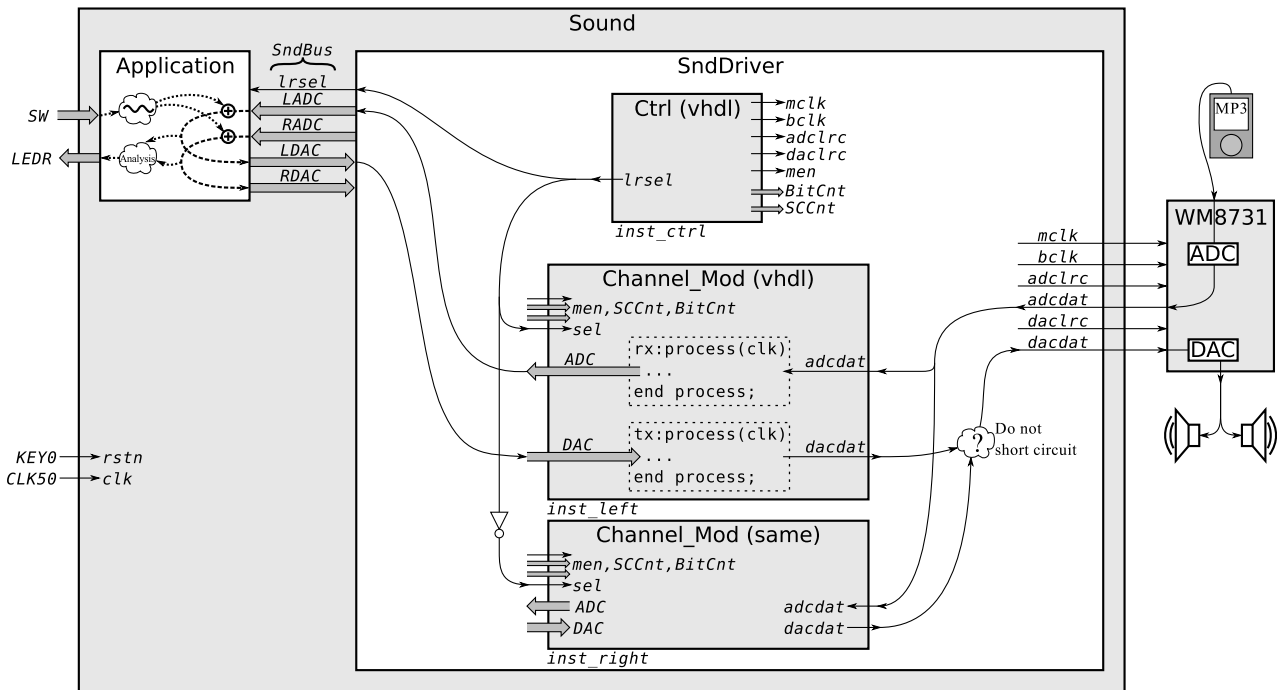


Figure 4: A structural view of **SndDriver** in its environment.

The sub module **Channel\_Mod** decodes one bidirectional channel. This is instantiated twice, one instance for the left and one for the right channel.

### 5.4.1 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**, **lrsl** ⇒ The **SndBus**, as described above.
- **mclk**, **bclk**, **adclrc**, **daclrc**, **adcdat**, **dacdat** ⇒ The serial signals to/from the WM8731 chip. Those are described in App. B (“The Sound Chip WM8731”).

You should also have the internal control signals, explained more in detailed in Sec. 5.4.2

- **men** ⇒ Master Enable signal.
- **SCCnt** ⇒ Sub Cycle Counter.
- **BitCnt** ⇒ Bit Counter.

Appendix C (last page) provides a timing diagram of all those signals.

### 5.4.2 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.

Have a look in App. C to understand how the signals should work. Figure 5 illustrates a few signals (where the counter is called `cntr`).

- `mclk`  $\Rightarrow$  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`  $\Rightarrow$  Bit clock should be a quarter of `mclk` (3.125 MHz). Where in `cntr` do you have this behaviour?
- `men`  $\Rightarrow$  Master Enable should be '1' just before the rising flank of `mclk`.
- `SCCnt`  $\Rightarrow$  The sub cycle counter counts the four `mclk` pulses within each `bclk` pulse. It must be two bits.
- `BitCnt`  $\Rightarrow$  The bit counter counts the 32 `bclk` periods for each sample (though only 16 of those are used).
- `adclrd`, `daclrc`  $\Rightarrow$  The left/right clock for the bit serial adc/dac channels. Those should be equal.

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.

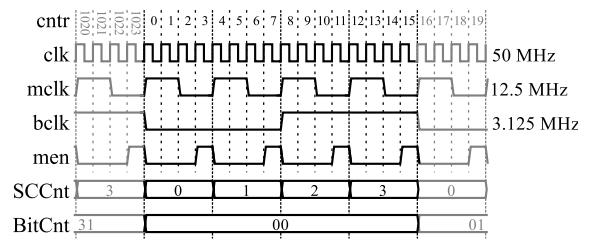


Figure 5: Clock timings.

### 5.4.3 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. B) that the samples are sent MSB first through `adcdat` and `dacdat`.

`RXReg` should, when `sel='0'`, shift in `adcdat` from the right<sup>1</sup> 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. B, we know that `dacdat` should come

<sup>1</sup>Shift in from the right, so the first incoming bit (MSB) will be shifted all way to the left.

from the left channel when `adclrc='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'?`).

## 6 Simulation

You have to simulate the `SndDriver`, in a way that detects any kind of error you may do.

In order to do so, there are a number of things you must do:

- 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.
- Compile and simulate the test bench and all the VHDL files related to `SndDriver`. Do not add the other VHDL files (`Sound.vhd` or `application.vhd`), since you will only simulate the driver.

### 6.1 The Test Bench

Have a look at the VHDL file for the test bench. You can observe that the structure of the test bench is:

- A clock generator part, that generates a 50 MHz clock, a reset signal, and a `done` signal (after 1 ms).
- An `adcdat/dacdat` translator part, that generates `adcdat` from an ADC sample stimuli register, and generates a DAC sample output register from the `dacdat`.
- A process that verifies the `*clk`, `adclrc`, `daclrc` and `lrsel`, in the following steps:
  1. Wait for 1  $\mu$ s, so the signals are stabelized.
  2. Wait until two rising edges of the `mclk`. Measure the time between
  3. Wait until two rising edges of the `bclk`. Measure the time between.
  4. Wait until two rising edges of the `adclrc`. Measure the time between.
  5. Verify that `mclk=1` and `bclk=0` after the `adclrc` edge.
  6. Report the result so far.
  7. Verify in a loop that `adclrc = daclrc  $\neq$  lrsel` for the rest of the simulation.
- A part that provides stimuli to the DUT (`adcdat` and the DAC bus).
- A process that verifies that the output of the DUT (the ADC bus, and `dacdat`) corresponds to what was sent in the previous sample.

You can see that the test bench is not complete. Your task is to finish the part between the comment “TO FILL IN:”, and the comment “STOP FILL IN”. The comments in that part will help you.

### 6.2 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”  $\Rightarrow$  “Analog (custom)...”. Max = 32767, Min = -32768.

## 7 The Result

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

## References

- [1] TSIU03: FAQ, section *Common Errors*.
- [2] TSIU03: FAQ, section *Lab Demonstration*.
- [3] The WM8731 Manual, U:\da\TSIU03\Manuals\WM8731\.

## Appendix A DE2 Overview

The used components on the DE2 board is

- The FPGA (EP2C35).
- The Sound chip (WM8731).
- The 50 MHz oscillator.
- The Key0 (as inverted reset).
- The SW5-7 (to control **Application**).
- The HEX6,HEX7 (to show the group number).
- The LEDR bar graph (to show the sound analysis).

## Appendix B The Sound Chip WM8731

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

The sound samples are transmitted bit serially via a digital interface, described below in sec. B.1.

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<sup>2</sup>C). This is done automatically when the DE2 are restarted, and nothing you have to care about.

If you are interested, have a look in the data sheet [3], 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 (kilo Sample per second).
- Two channels means a total sample rate of  $\approx 96$  kSps.
- The slave mode means that we (on the FPGA) must provide clock and control signals (see the serial interface below).
- It affects the digital interface drastically, this is described below.

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

### B.1 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 ( $\frac{\text{mclk}}{4}$ ).
- **adclrc**  $\Rightarrow$  A left/right selector for **adcdat**. **adclrc**='1' for left.
- **adcdat**  $\Leftarrow$  Serial bits from the ADC (one bit per **bclk** pulse).
- **daclrc**  $\Rightarrow$  A left/right selector for **dacdat**. **daclrc**='1' for left.
- **dacdat**  $\Rightarrow$  Serial bits to the DAC (one bit per **bclk** pulse).

In the following, **\*lrc** refers to **adclrc** or **daclrc**, **\*dat** to **adcdat**/**dacdat**, **adc\*** to **adclrc**/**adcdat** and **dac\*** to **daclrc**/**dacdat**.

The **adc\*** and the **dac\*** signals works in exactly 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 FPGA generates both the **\*lrc**.

For **adc\*** the WM8731 is the transmitter, and the FPGA the receiver. For **dac\*** it the opposite.

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

In this lab it is suitable if `adclrc = dacclrc`, i.e., you read and write the right channel data simultaneously, and then the left channel data simultaneously. Note that the receive 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 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.

## Appendix C Timings

Figure 6 shows a timing diagram that illustrates how the signals are related to each other in time.

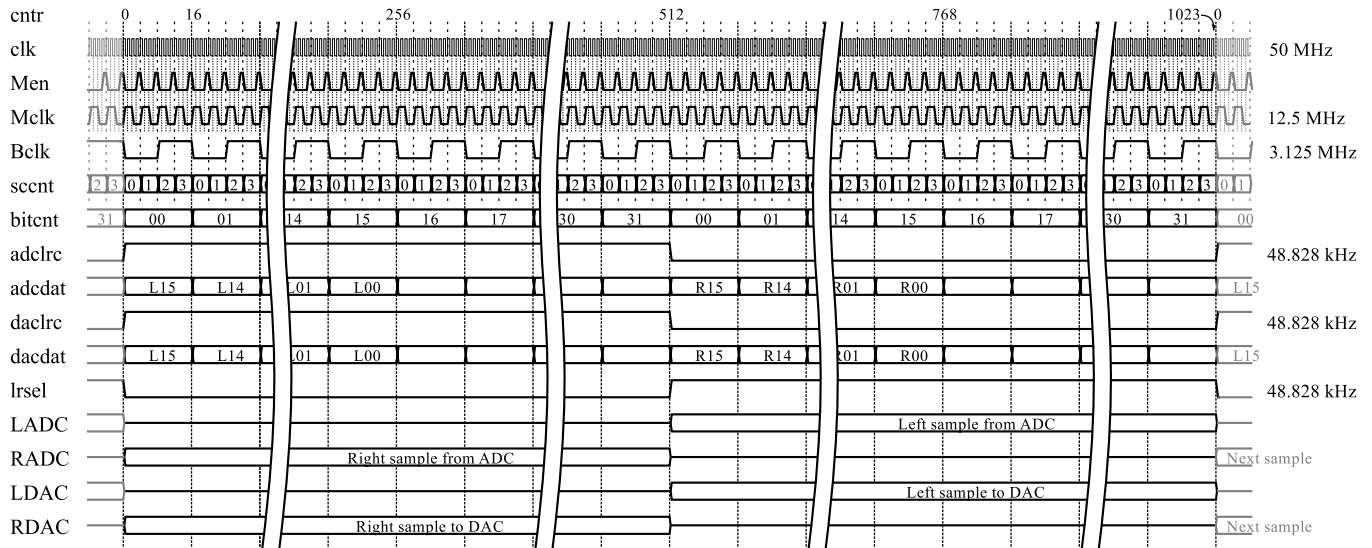


Figure 6: Timing information for almost all control signals in the system.

Note that the data on the `*DAC` are copied to the `dacdat` in the same clock cycle as `BitCnt` are reset and `dacclrc` changes.