TSIU03: Lab 4 - Audio Codec

Petter Källström, Mario Garrido

September 18, 2014

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

1 Introduction

In this lab there is a system (illustrated in Fig 1), where you should create the module "SndDriver", that 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).

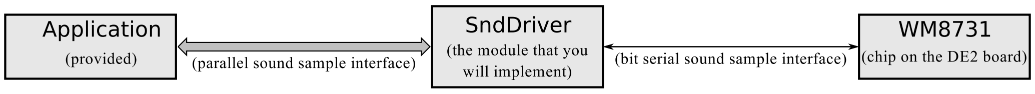
Application
(provided)
(parallel sound sample interface)
SndDriver
(the module that you will implement)
(bit serial sound sample interface)
WM8731
(chip on the DE2 board)

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 system (including the parallel sample interface).
Appendix D 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 App. C.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 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 pinns 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.

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

3.1 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 the Application (that should be indicated on the LED bar).

⋆ Possible to detect in a simulation.

4 Requirements to Pass

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

- You must implement the SndDriver, including its requirements from App. C.4.
- The functions in “Application” must work (See Appendix C.3). 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.

5 Implementation Hints

Appendix C.4 suggests some solution. Those are on a “design specification level”, and you might need some better implementation help.

Figure 2 illustrates the connection of different modules in the system at different levels.

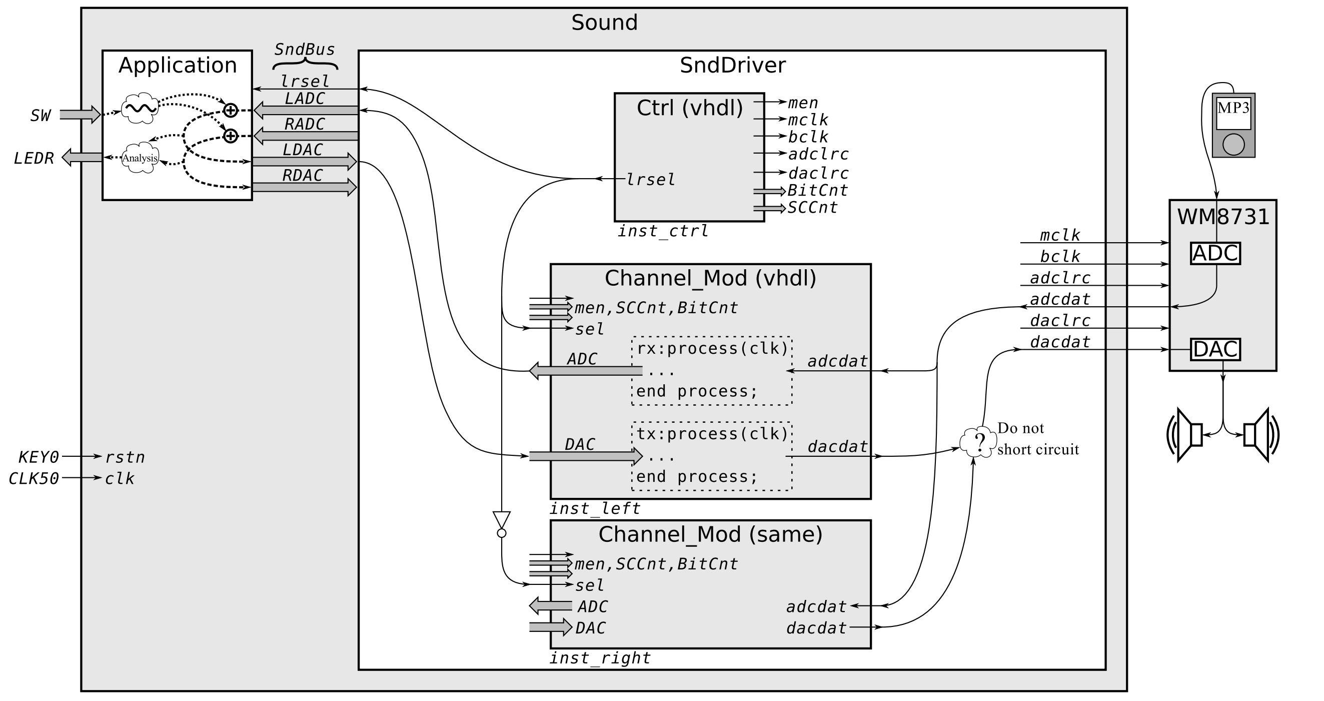

Figure 2: A structural view of `SndDriver` in its environment.

5.1 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. D to understand how the signals should work. Figure 3 illustrates a few signals (where the counter is called `cntr`).

- `mclk` ⇒ Master clock should be a quarter of `clk`. You have this behavior in `cntr(1)`. Note that when any bit with more significance changes, this bit flips from `'1'` to `'0'`, e.g., a falling flank. To get a rising flank behavior, simply invert the bit (hence, something like `mclk<=not cntr(1);`.
- `men` ⇒ Should be `'1'` just before the rising flank of `mclk`. Figure out how to generate this signal by logic operators on the bits of the counter.

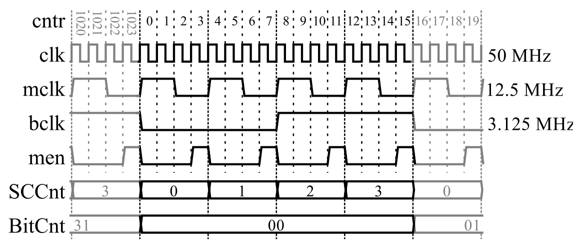

Figure 3: Clock timings.

- SCCnt and BitCnt can also be obtained from the bits of the counter.

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 "1111111111" comes "0000000000". Do not plot all 1024 counts, just as many as needed for your understanding.

5.2 Channel_Mod

Channel_Mod gets the signal sel, which indicates that the SndBus interface is active. When sel is zero, the serial WM8731 interface should be active (e.g., shift in/out the bits from/to adcdat/dacdat.

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 combinatorial logic to solve the dacdat problem.

RXReg should, when sel='0', shift in adcdat from the right[1] when the bclk changes from '0' to '1' (e.g., 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.

5.3 SndDriver

The two instances of Channel_Mod gives one dacdat each that should be merged. This is discussed in "Implementation Hints" in App. C.4.

If you solve the dacdat problem using an AND gate, an OR gate or a multiplexer, this logic is called "glue logic". Glue logic is used to "glue together" bigger modules, where they cannot be simply connected together.

Another use of glue logic is the sel inputs to the Channel_Mods, that need an inverter on one of the instances.

5.4 Top Module (Sound)

You must provide your group number to group_no in the top module, in its Generic Field, like in the VGA lab. Do not edit the top module in any other way .

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

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 µs, 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` ≠ `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" ⇒ "Analog (custom)...". Max = 32767, Min = -32768.

7 The Result

The intended behavior of the result is specified in Tab. 2. 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 SW0-4 (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 sound chip interface is described in details in the chip manual[3] (page 33 and forward). The chip contains lots of configuration registers, that can be set via an I^{2}C bus. Those settings are done by the default application on the DE2 board, so you don't have to do this, but you need to know which settings are done.

B.1 The Settings

The contents of the interesting registers, set by the default application, are shown in Table 1.

Register	Label	Value	Meaning
0x05	DEEMP	`11`	De-emphasis Control = 48 kHz. (the sample rate)
0x07	Format	`01`	Audio Data Format Select = MSB-First, left justified.
0x07	IWL	`00`	Input Audio Word Length = 16 bits.
0x07	LRP	`0`	DACLRC Phase Control = Right Channel DAC data when DACLRC low.
0x07	LRSWAP	`0`	DAC Left Right Clock Swap = Right Channel DAC Data Right.
0x07	MS	`0`	Master Slave Mode Control = Enable Slave Mode.
0x07	BCLKINV	`0`	Bit Clock Invert = Don't invert BCLK.
0x08	USB/NORMAL	`0`	Mode Select = Normal mode (256/384fs).
0x08	BOSR	`0`	Base Over-Sampling Rate = 256fs.
0x08	SR	`0000`	Sampling rate = 48 kHz.
0x08	CLKIDIV2	`0`	Core Clock divider select = Core Clock is MCLK.
0x08	CLKODIV2	`0`	CLKOUT divider select = CLOCKOUT is Core Clock.

Table 1: The content of some WM8731 registers, set by the default application.

This means that the sampling rate (for both the ADC and the DAC) is 48 kHz, and the master clock (MCLK) should be 256 times higher, e.g. 12.288 MHz.

For simplicity, set `mclk` to 12.5 MHz (50 MHz/4), which will give a sampling rate (fs, or f_s) of 48.828 kHz (per channel). Since there are two channels, it will result in 97.656 kSps (kilo-samples per second), which are delivered left-right-left-right-left-... in a time interleaved fashion, in both directions.

B.2 A Brief Summary of the Serial Interface

You should provide ($\Rightarrow$) or read ($\Leftarrow$) the following signals to/from the WM8731 chip:

- `mclk` $\Rightarrow$ A 12.5 MHz master clock.
- `bclk` $\Rightarrow$ A 3.125 MHz bit clock ($\frac{\texttt{mclk}}{4}$).
- `adclrc` $\Rightarrow$ A left/right selector for `adcdat`.
- `adcdat` $\Leftarrow$ Serial bits from the ADC (read one bit per rising flank of `bclk`).
- `daclrc` $\Rightarrow$ A left/right selector for `dacdat`.
- `dacdat` $\Rightarrow$ Serial bits to the DAC (write one bit per falling flank of `bclk`).

The `adclrc` and `daclrc` will act as a $\frac{\texttt{bclk}}{64}$ clock, which means that each channel will have 32 `bclk` cycles per sample. The first 16 are used to transfer bits on `adcdat` and `dacdat` (most significant bit first), and the rest are unused.

In this lab it is suitable if `adclrc` = `daclrc`, e.g., 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` ≠ `adcdat`.

The `*dat` signals are written on the rising flank of `bclk`, and read on the falling flank of `bclk`.

You should generate all these signals, except the `adcdat`, which is generated in the WM8731 when it receives the `bclk` falling flank (hence the small delays in the figures).

Finally, it must be mentioned that the 16 bits samples are in signed format, e.g. 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 The Sound System

This appendix describes how the system are/should be implemented, from a high level perspective.

C.1 Module Sound: Top Module

The top module is only a "glue together" module, depicted in Fig. 4, with the sub modules `Application` and `SndDriver`. They communicate via the bus *SndBus* (several signals). Both modules gets the clock, `clk`, (50 MHz) and the reset, `rstn`, (active low).

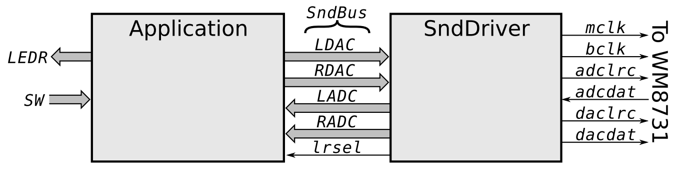

Figure 4: The top module schematics.

C.2 Interface SndBus

In the SndBus, `LADC` and `LDAC` are active when `lrsel='1'`, and `RADC` and `RDAC` are active when `lrsel='0'`, as depicted in Fig. 5.

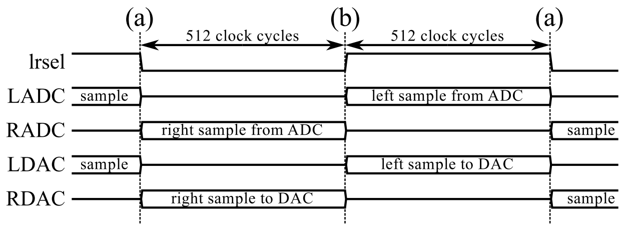

Figure 5: The timing of the SndBus signals during one sample period

While `lrsel` is low, the Application should read the sample on `RADC`, and write an output sample on `RDAC`, and the corresponding for the left channel while `lrsel` is high. It is up to Application to detect when lrsel changes, in order to process the samples.

When `SndDriver` turns `lrsel` from high to low, (a) in Fig. 5, it should read what is on `LDAC` (to be sent to the DAC), and make sure the new data (from the ADC) is available on `RADC`. Equally, in (b), when `lrsel` goes from low to high, `SndDriver` should read `RDAC` and write `LADC`.

Since `clk` is 50 MHz, which is divided with 1024 to get the sample frequency 48.828 kHz, there will be 512 clock cycles per sample, as depicted in Fig. 5.

C.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 2.

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.
Noise	It generates white noise on the `LDAC`/`RDAC` that is "not used" (and hence will not be heard).
Analyze sound	It writes some kind of low pass filtered logarithmic amplitude indicator of the output on the red LEDs.

Table 2: 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 implemented:

- 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 analyzer is implemented using a squarer, a first order low pass filter (LPF), and then it simply pick the bits from the filter register.

C.4 Module SndDriver

The module `SndDriver` is an adapter: It translates the audio signal between the parallel format used by the Application, and the bit serial format (including all control signals) used by the WM8731 chip.

C.4.1 Requirements

The requirements for `SndDriver` are listed in Table 3.

Bidirectional	It should work on both the receiver and transmitter in the same time.
Interface	It should implement both the SndBus interface and the WM8731 bus interface.
Control signals	It should generate all needed control signals in the interfaces.
Sub modules	It should be implemented with the sub modules in C.4.3 (Implementation Hints).
Data types	Proper use of the (`un`)`signed` vector types should be used.

Table 3: Requirements on the SndDriver.

C.4.2 Signal Description

The following signals will exist in the module `SndDriver`.

- `clk`, `rstn` ⇒ System clock (50 MHz) and the active low reset.
- `mclk` ⇒ Master Clock - a 12.5 MHz clock signal to the WM8731 chip.
- `bclk` ⇒ Bit Block - a 3.125 MHz clock to the WM8731 chip, specifying the serial bit rate.
- `adclrc`, `daclrc` ⇒ Left/Right select signal to the WM8731 for the ADC/DAC signals respectively.
- `lrsel` ⇒ Left/Right select signal for the SndBus interface to the Application.

- men ⇒ Master Enable signal within the SndDriver. This is '1' one clk cycle before each rising flank of mclk.
- SCCnt ⇒ Sub Cycle Counter, a two bit counter that counts the mclk cycles within each bclk cycle.
- BitCnt ⇒ Bit Counter, a five bit counter that counts bits in a word (0 to 31).
- adcdat, dacdat ⇒ Data bit signal from/to the WM8731.
- LADC, RADC, LDAC, RDAC ⇒ Parallel data buses to (*ADC) or from (*DAC) the application for left and right channel.

C.4.3 Sub Modules

The SndDriver should be implemented using two sub modules; Ctrl and Channel_Mod.

- Ctrl ⇒ generates all required control signals: mclk, bclk, adclrc, daclrc, lrsel, BitCnt, SCCnt and men. This module will just take the clk and rstn signals. It can be implemented as a ten bit counter, where you pick the correct bits for the values.
- Channel_Mod ⇒ will handle the serial/parallel conversion for one channel. It will receive the required control signals from Ctrl. It will also generate the *ADC signal in the SndBus interface, as well as the dacdat bit to the WM8731.

SndDriver should instantiate one instance of Ctrl (named to u_ctrl), and two instances of Channel_Mod, named u_left and u_right.

Each instance of Channel_Mod will output a dacdat signal, and SndDriver should merge those somehow and output the result. Since the final dacdat will send data from the "active" channel, this can be solved by, e.g., a multiplexer, that reads the two dacdat signals from the two Channel_Mods, and the daclrc as the select signal. Another solution is that Channel_Mod will output data when active, and only '0' when inactive, in this way, the multiplexer can be replaced by an OR gate.

Appendix D Timings

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

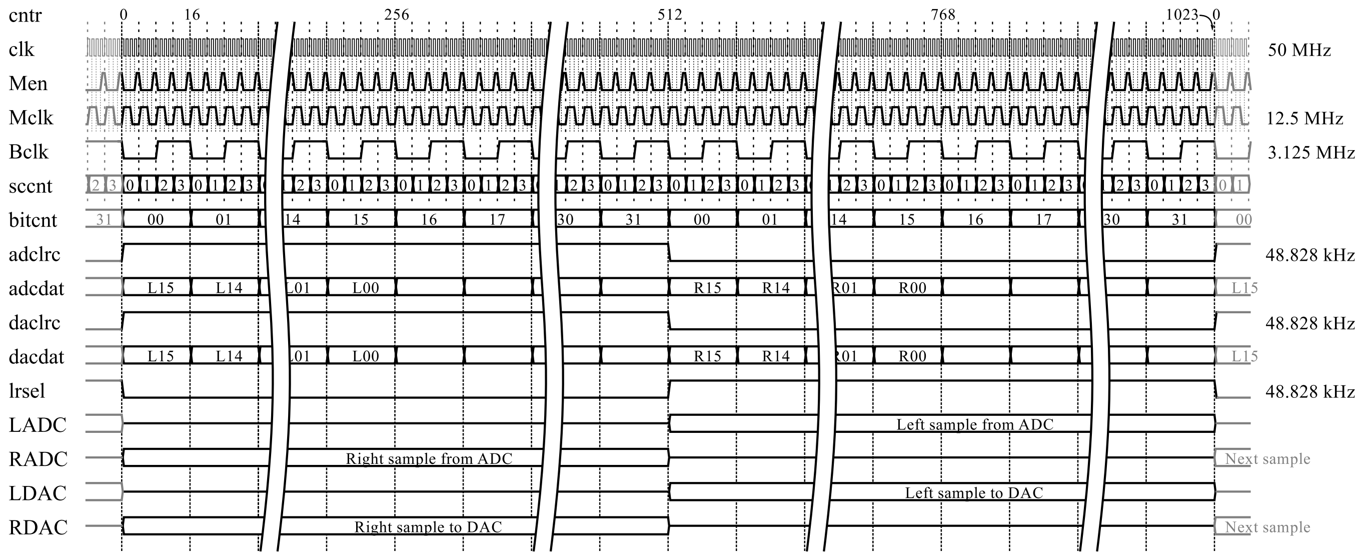

Figure 6: Timing information for almost all control signals in the system.

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