Digital Circuit Lab — Lab 3 Tutorial

Audio Recorder

Date: 05/02/2017

Team 05 Member: 郭子生 b03901032、曾耕森 b03901154、楊其昇 b03901101

Github: https://github.com/awinder0230/Digital-Circuit-Design-Lab

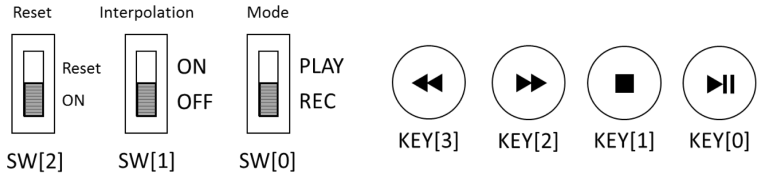
- **Introduction**

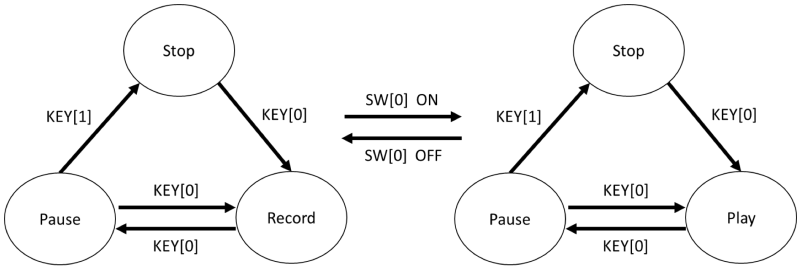
We are going to implement an audio recorder in this project. The audio recorder should be able to support the following functions, including **record**, **play**, **pause** and **stop**. Besides these fundamental functions, the audio recorder should be able to play at different speed as well as provide a user-friendly interface.

- **Support Functions**
  1. Main functions: record, play, pause, and stop.
  2. Sampling rate at 32kHz, each sample consumes 16 bits.
  3. Able to record for at most 32 seconds.
  4. Able to play faster at speed x2 ~ x8 as well as play slower at speed x1/2 ~ x1/8.
  5. Support both piecewise constant interpolation and linear interpolation for different speeds.
  6. An user-friendly interface implemented with seven segment display.

- **Tools**: Altera DE2-115 FPGA, System Verilog, Quartus II, RS-232 Qsys, Microphone, Speaker

- **Usage**
  - Connect your microphone and speaker to the FPGA board.
  - Record
    1. Switch SW[0] to REC.
    2. Press KEY[0] to start recording.
    3. You can press KEY[0] to pause the record and press again to continue.
    4. To stop recording, press KEY[0] to pause first and press KEY[1] to stop.
  - Play
    1. Switch SW[0] to PLAY.
    2. Press KEY[0] to play the recorded audio.
    3. You can press KEY[0] to pause and press again to continue.
    4. To stop playing, press KEY[0] to pause first and press KEY[1] to stop.
    5. You can speed up the audio by pressing KEY[2] or slow down by pressing KEY[3]. The player supports up to 8x and down to 1/8x speed.
    6. In slow-forward replaying, you can enable linear interpolation by switching SW[1] to ON.





- **SRAM Communication**

The SRAM on DE2_115 has 2MB memory capacity and is organized as 1024K words of 16 bits. There're seven signals associated with the SRAM:

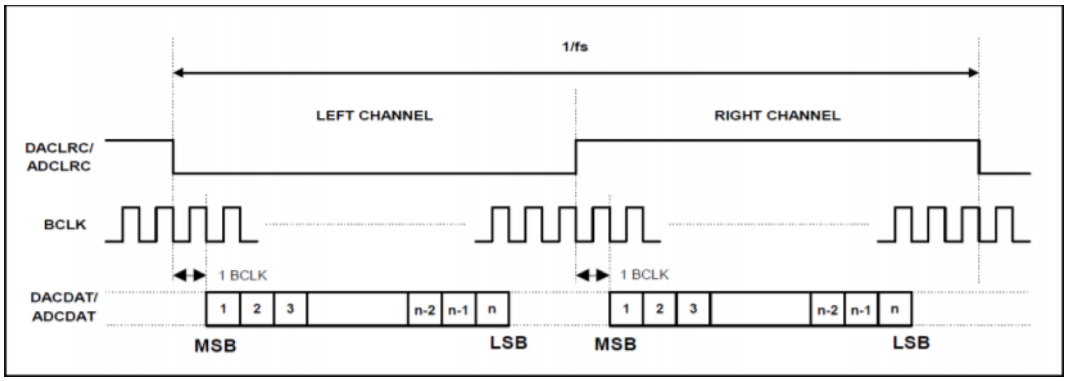
- SRAM_ADDR[19:0]: The read/write address of the SRAM.
- SRAM_DQ[15:0]: The 16 bits data to be read or stored in the SRAM. Note that this is a inout port, which means that one must beware of the control of this signal to avoid multiple driver of signal.
- SRAM_OE SRAM: Output Enable
- SRAM_WE SRAM: Write Enable
- SRAM_CE SRAM: Chip Enable
- SRAM_LB SRAM: Lower Byte Control
- SRAM_UB SRAM: Upper Byte Control

The control signals of the read/write operation are listed as below:

						I/O PIN		
Mode	$\overline{WE}$	$\overline{CE}$	$\overline{OE}$	$\overline{LB}$	$\overline{UB}$	I/O0-I/O7	I/O8-I/O15	V_{DD} Current
Not Selected	X	H	X	X	X	High-Z	High-Z	I_{SB1}, I_{SB2}
Output Disabled	H	L	H	X	X	High-Z	High-Z	I_{CC}
	X	L	X	H	H	High-Z	High-Z	
Read	H	L	L	L	H	D_{OUT}	High-Z	I_{CC}
	H	L	L	H	L	High-Z	D_{OUT}	
	H	L	L	L	L	D_{OUT}	D_{OUT}	
Write	L	L	X	L	H	D_{IN}	High-Z	I_{CC}
	L	L	X	H	L	High-Z	D_{IN}	
	L	L	X	L	L	D_{IN}	D_{IN}	

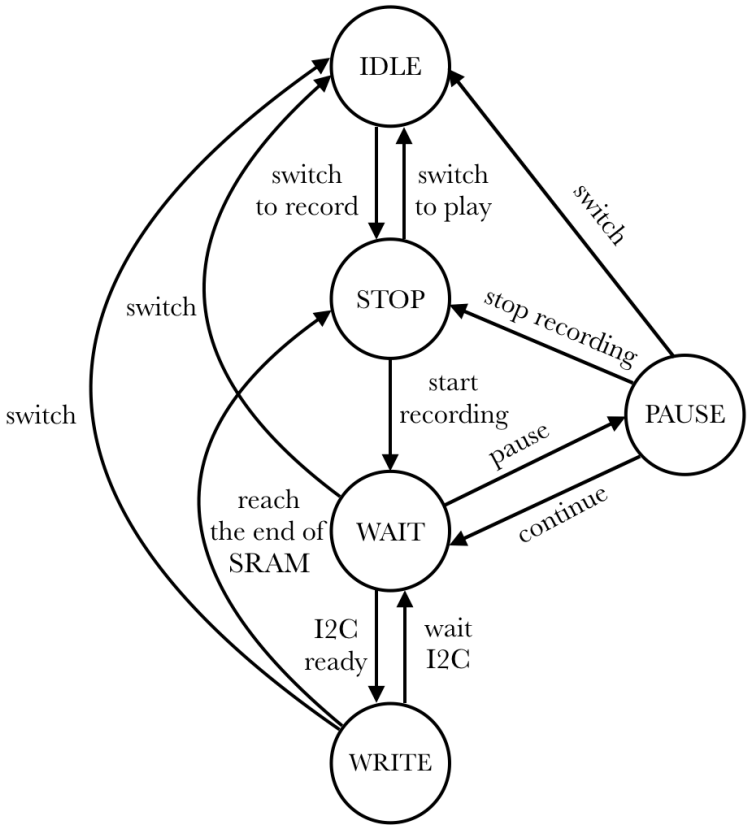
- **WM8731**

After initialization by passing configuration with I2C protocol, WM8731 will be activated as DAC and ADC between audio signal and the FPGA. We use the 12MHz BCLK generated by WM8731 as the main clock of our top design. The sample rate (i.e. the frequency of DACLRCK/ ADCLRCK) is set to 32kHz. Each DACLRCK/ADCLRCK clock cycle contains one sample of the audio signal which, in our case, is 16 bits long. The serial data transmission format follows the rules under I2S mode. I2S mode is one of the audio interfaces offered by WM8731. Please infer to the figure for details.



- **Recorder**

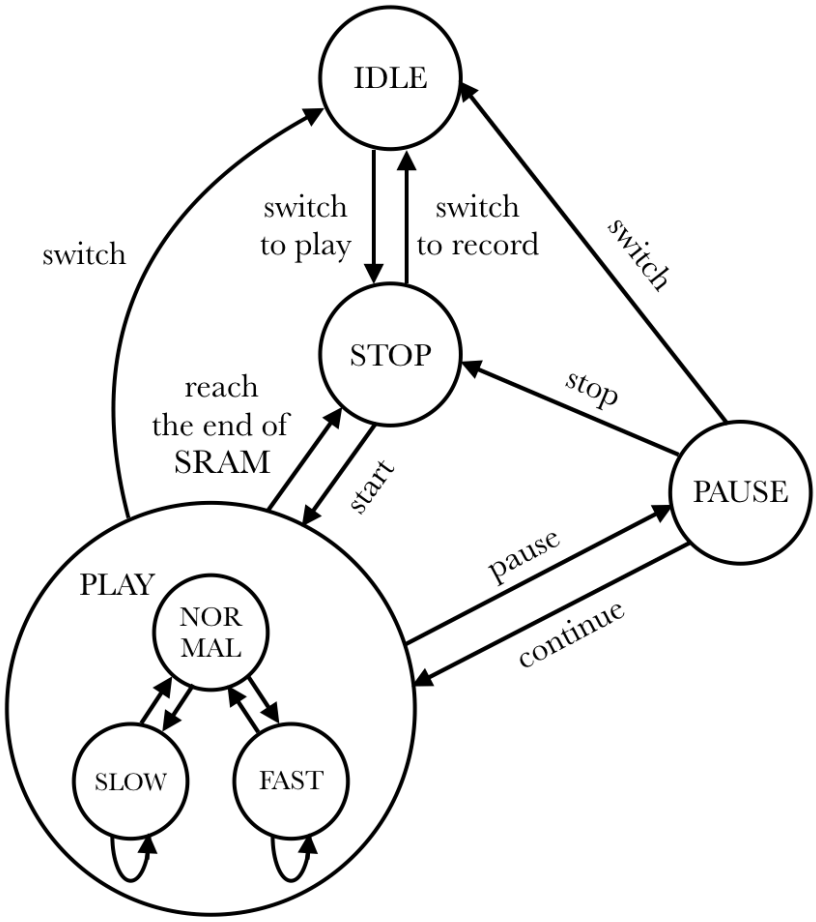
In record mode, we only record the signals from left channel. The FSM is shown as figure. In each ADCLRCK clock cycle, we receive a stream of 16bit serial data at the rate of BCLK from WM8731 and transform it to parallel signal. Note that there's a blank BCLK clock cycle after the edge of ADCLRCK. Then we enable the write signal of SRAM to write the data into SRAM and add the address by 1. When the machine runs at modes other than record mode, the output SRAM_DQ should be set to high impedance to avoid conflict when reading data from SRAM. The write enable signal should also be disabled.



- **Player**

In the play mode, we read the data from SRAM and pass it to the WM8731 by I2S interface. Then WM8731 will convert the digital data to analog signal and output it to audio devices (e.g. speakers). The replay process is pretty straightforward, just simply read the 16bit data from the SRAM for each address and pass it to the audio chip bit by bit. Note that we pass the same data to the left and right channel since we only records the audio from the left channel. To fast-forward the audio, say, play at 3x speed, we can add the address by 3 instead of 1 for each LRCK clock cycle.

Slow-forward is the trickiest part. Assume that we want to play at 1/2x speed. One simple way is to hold each data for two cycles. For more precise results, we can do linear interpolation. However, the calculation requires division and multiplication, which are expensive from the viewpoint of hardware, especially division. There are a few ways to approximate the results without using expensive calculations. For instance, we can calculate the step of each cylcle s = (b-a)/n and add the current data by s to avoid multiplication. Or, we can multiply the two data by the ratio and calculate their sum (e.g. a*0.25+b*0.75). The ratio is represented as binary floating point number. By this way we can avoid division.

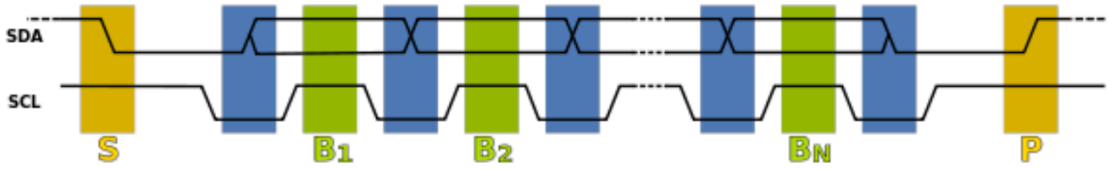


- **ALTPLL**

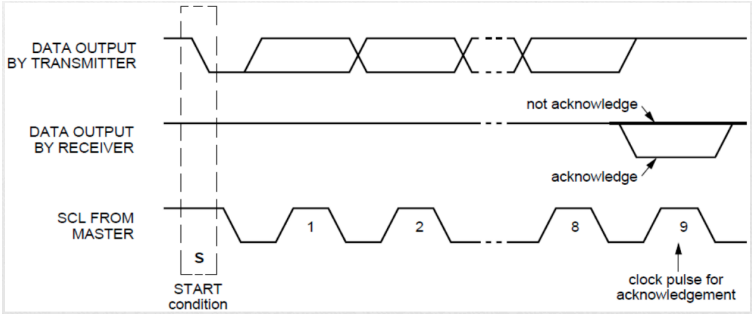
In our implementation, we use the clock signals generated by WM8731, which is set to 12MHz and 32kHz respectively, as our primary clock signals. We use PLL to convert the 50MHz clock to 12MHz and 100kHz. The 12MHz clock is fed to the AUD_XCK port and the 100kHz clock is used to initialize WM8731 (I2C protocol).

- **I2C Protocol**

In order to communicate with and initialize the chip WM8731, we will need the help from I2C protocol. The picture below is the waveform of I2C protocol:



When SCL=1, SDA would be recognized as valid data (green part). The exception would occur at the beginning and the end (brown part) of data transfer. Besides, SDA sets the transferred bit while SCL is low (blue part). After every 8 data bits in one direction, an "acknowledge" bit (0) is transmitted in the other direction. Last but not least, there are 10 registers that should be initialized in WM8731:



Left Line In	000_0000_0_1001_0111
Right Line In	000_0001_0_1001_0111
Left Headphone Out	000_0010_0_0111_1001
Right Headphone Out	000_0011_0_0111_1001
Analogue Audio Path Control	000_0100_0_0001_0101
Digital Audio Path Control	000_0101_0_0000_0000
Power Down Control	000_0110_0_0000_0000
Digital Audio Interface Format	000_0111_0_0100_0010
Sampling Control	000_1000_0_0001_1001
Active Control	000_1001_0_0000_0001

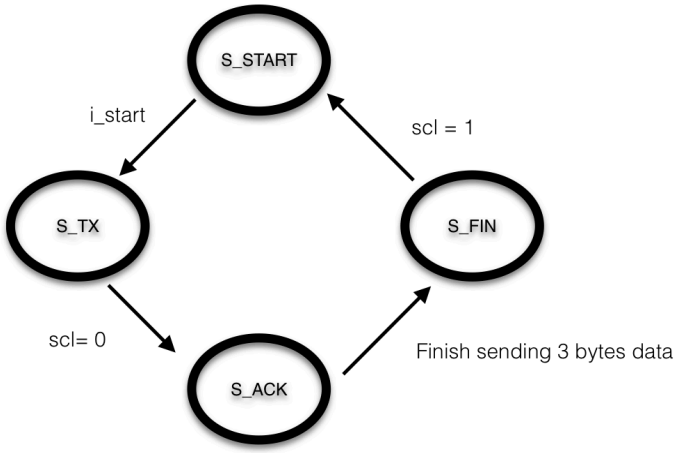
We have designed 2 modules to implemented I2C protocol:

1. I2Csender.sv:

In practice, each data transfer would transmit 24 bits data. 24 bits data contains 3 parts, and we take the recommended value of Left Line In for example:

a. Hardware location of WM8731(leftmost 7 bits): 0011010
b. Location of registers in WM8731(8th ~ 14th bits from the left): 0000000
c. Operation data for register: 0_1001_0111, for example.

The picture down below is the FSM of this module:



S_START:

When i_start is high, the module would read the input data (i.e. i_data_w = input), and initialize all the counters as well as output data to 0. After initialization, the state would be changed to S_TX.

S_TX:

When SCL is high, we would set SCL to low state, set output bit to the left most bit of the input data read in S_START, and shift one bit left of the input data (i.e. i_data_w = i_data_r << 1). After doing so, the next state would jump to S_ACK.

S_ACK:

According to the protocol, each time before sending 1 byte data, SCL should be changed from high to low. Therefore, we set SDL = 1 at the first clock cycle and SDL = 0 at the second clock cycle. After doing this, we will start to send the data bit by bit just like what we did in S_TX (i.e. assign output data and shift left 1 bit of input data).

After finishing sending one byte data, we would set the output bit to the ACK signal (1'bz). In practice, we have another signal output_enable to control above behavior. We actually set output_enable=0 after sending 8 bits data; otherwise, output_enable is 1. That is the reason why the Lab tutorial slides doing this: assign SDA = oe? a: 1'bz;

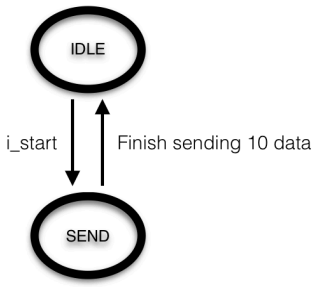
After sending 3 bytes data, the next state would be S_FIN.

S_FIN:

If SCL is high, we set both output bit and finish signal to high and the state would jump to S_START.

2. I2Cinitialize.sv:

In our design, we have to initialize 10 registers, and it means that we would use I2Csender.sv 10 times to send 24 bits data each time.



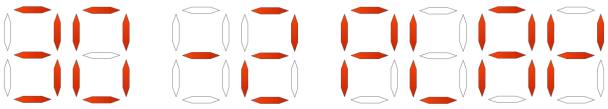
- **Seven Segment Display**

1. Timer

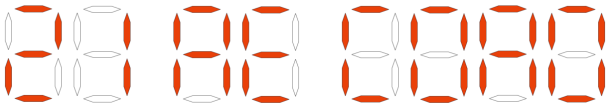
Two seven segment displays represent playing or recording time in seconds. As mentioned above, SRAM stores audio digital signals at address range from 0 to 2^{20}, we therefore split 0 to 2^{20} into 32 intervals (since the audio recorder records for at most 32 seconds), and by identifying the current reading/writing address interval of SRAM, we are able to show the correct timer of audio recorder by seven segment display. For example, when SRAM manipulating data at address 0001_1000_0000_0000_0001, seven segment display will show 2 digits number '04'. Note that we are able to check only most significant 5 bits to identify the address interval.

2. State Machine

For users to know the current state of the recorder, we utilize the rest of seven segment displays to show the current state of the machine. There are 6 states in total, including init, idle, record, stop, play, and pause. Further more, since user may play the recorder at different speeds, we also show the playing speed while the recorder is playing. Two examples of our seven segment displays are shown on the right.



Playing at Speed 1/2 to 30 seconds



Recording to 21 seconds

- **Debug Tools**

1. Test bench

Test benches are useful when you want to check whether your module works fine or not. After running ncverilog, you can view your waveforms on nWave and check if each signal behaves as you expected.

2. SignalTap logic analyzer

Quartus II provides a very useful tool to monitor your signal waveforms when running program on FPGA. It provides instant waveforms and thus is very helpful when debugging run-time errors.

- Go to Tools → SignalTap II Logic Analyzer.
- At the Setup section you can select signals you want to monitor, and set the basic clock at Signal Configuration.
- Save your settings as xxx.stp.
- Go back to your project and go to Assignments → Settings → SignalTap II Logic Analyzer and specify the stp file.
- Compile your design. SignalTap basically insert probes in your circuits to monitor the signals specified by the user, so we have to recompile the design to activate them.
- Program the sof file to your FPGA.
- After the program start running, press "Autorun Analysis" to view your instant waveforms.