Lab 4: State Machine with Audio Output

EE478 Fall 2019

Objective

In this lab, you will implement a finite state machine to control an audio codec – a chip that interfaces a digital signal from our FPGA with a DAC and an audio output jack. **You will need headphones for this lab** so please bring a pair with you.

You will be starting from provided code that has a controller for the AD SSM2603 audio codec on our Zybo board. You will implement a finite state machine that changes the frequency of a square wave output every 1 second, producing a simple 4 note musical tune. A pushbutton will be used to start the tune.

Deliverables

Lab start: Beginning of lab session during week of 10/14.

Demo: Working finite state machine changing the frequency of the square wave that is used as input to the audio codec. Working design programmed onto the Zybo board, producing the 4-tone tune, with one second per tone, when a pushbutton is pressed. **No testbench required.**

Demo due: End of lab session during week of 10/21 (2 week lab)

Design Document due:

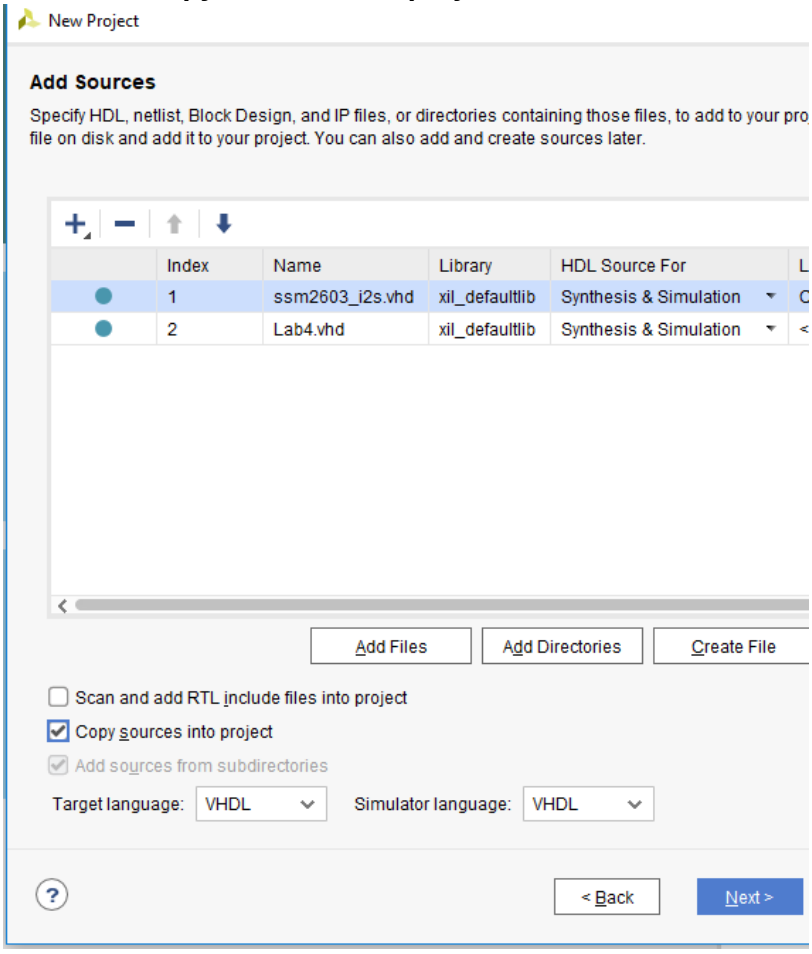
- On UBLearns by 11:59PM on 11/1. *Submit one per team*.

Lab Contents

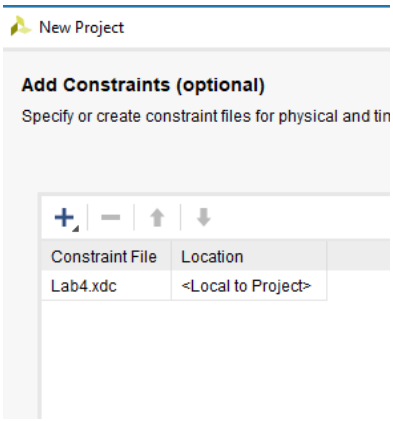
Part A: Creating the Lab 4 Project

In this lab, you will be provided with an existing vhd file for the AD SSM2603 device. You will create a Lab4.vhd module that instantiates this component and drives square waves into its inputs to play different notes. This section will walk you through downloading and setting up the project.

1. Download the ssm2603_i2s.vhd file from UBLearns under Documents – Labs – Lab4.
2. Create a new Vivado project and this time during project creation, select **Add Files** instead of just creating a new file. Browse to where you downloaded the provided code and select ssm2603_i2s.vhd.
3. Also press Create File and make a file called Lab4.vhd
4. Check the **Copy sources into project** checkbox



5. Press Next, then create a new Constraints file as well, called Lab4.xdc



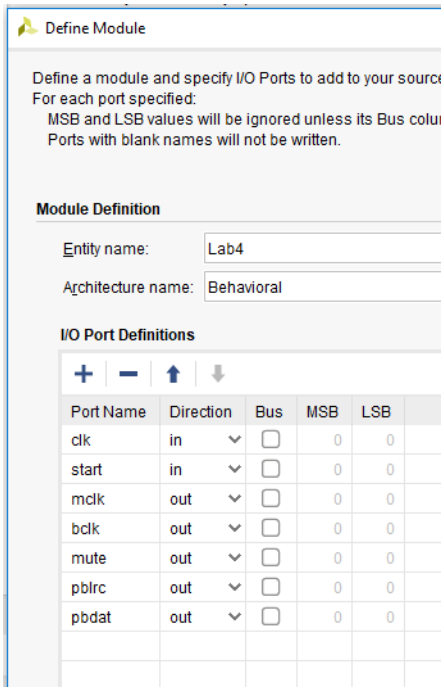
6. Finish setting up the project as usual, and then you will be prompted to set up the I/O for Lab4.vhd.

Create the following I/O signals for Lab4.vhd:

Inputs/Outputs:

- clk – 125MHz system clock
- start – start signal connected to the BTN0 pushbutton
- mclk – 12.288MHz master clock to the codec
- bclk – 3.072MHz bit clock to the codec
- mute – active low mute signal (connected to '1' so we aren't muting)
- pblrc – left/right channel select for the codec
- pbdat – serial I2S playback data

Take a look at the SSM2603 datasheet on UBLearns to understand what these signals are being used for.



Uncomment line 27 to allow us to use unsigned and signed data types in our design, as before.

Now we will instantiate the codec driver into our Lab4.vhd module. Follow these steps

- Declare internal signals called r_data and l_data that are 24-bit std_logic_vectors. These signals will be the actual audio data that is played through your headphones.
- Declare an internal signal called mclk_sig. This signal will be a 12.288MHz clock required by the audio codec chip, that we will also use as our own clock signal.
- Declare an internal signal called ready. This signal is output by the codec to tell us it is ready for a new audio sample.
- Declare a component for ssm2603_i2s.
- Instantiate the component, hooking up its clk, bclk, mute, pblrc, and pbdat signals to the top level I/O signals, and hooking up l_data, r_data, and mclk to our internal signals.

Your Lab4.vhd file architecture should now look similar to this:

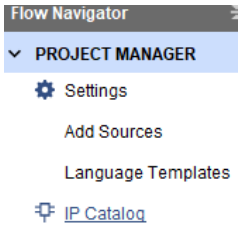
```
component ssm2603_i2s
    Port ( clk : in STD_LOGIC;
           r_data : in STD_LOGIC_VECTOR (23 downto 0);
           l_data : in STD_LOGIC_VECTOR (23 downto 0);
           bclk : out STD_LOGIC;
           pbdat : out STD_LOGIC;
           pblrc : out STD_LOGIC;
           mclk : out STD_LOGIC;
           mute : out STD_LOGIC;
           ready : out STD_LOGIC);
end component;

signal mclk_sig       : std_logic;
signal l_data, r_data : std_logic_vector(23 downto 0) := (others => '0');
signal ready          : std_logic;

begin

codec : ssm2603_i2s port map(
    clk    => clk,
    mclk   => mclk_sig,
    bclk   => bclk,
    mute   => mute,
    pblrc  => pblrc,
    pbdat  => pbdat,
    l_data => l_data,
    r_data => r_data,
    ready  => ready
);
mclk <= mclk_sig;
```

Part B: Creating a PLL

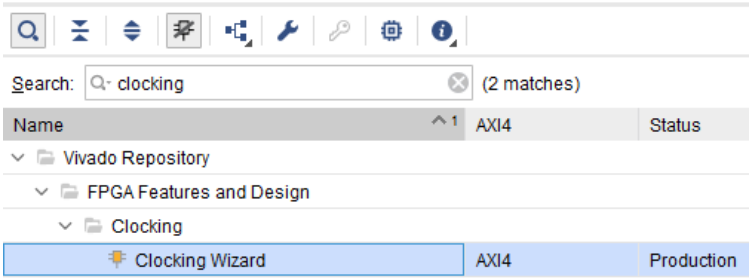
Open up the ssm2603_i2s.vhd file and notice that there is an error on line 58, where mclk_gen is being instantiated. The codec chip requires a 12.288MHz master clock signal for 48kHz audio playback, which cannot be easily created by dividing our 125MHz clock. Instead, we will create this clock by using a **Phase-Locked Loop** (PLL). A PLL is an analog control system that generates output signals with frequency and phase that are related to an input clock signal, and we can use PLLs in our designs to create clock signals with arbitrary frequencies. PLLs are extremely important in many branches of electrical engineering, so if you haven't come across them in class it's highly recommended that you study up on them a bit.

The Zynq chip on our board has two PLLs for us to use but we have to configure the project to use this **IP Core**. An IP Core is a set of Xilinx specific functionality that we can't access just through the VHDL language. We need to use a wizard in Vivado to set up the core for us.

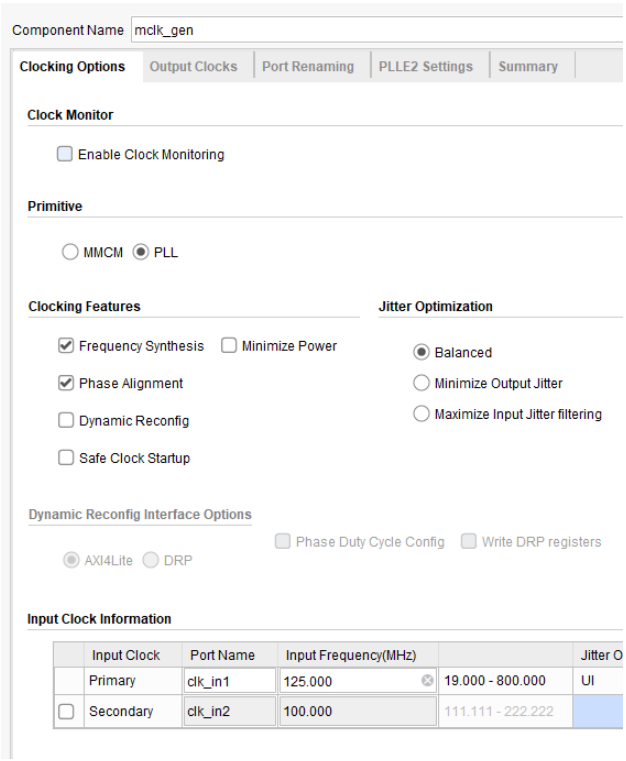
Under the Flow Navigator on the left, press the IP Catalog button.



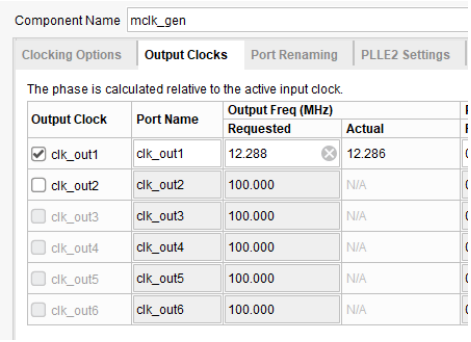
In the search box, type "clocking" and double click "Clocking Wizard"



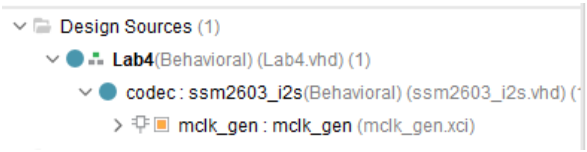
For Component Name, enter "mclk_gen." Change the Primitive type to PLL and under Input Clock Information, enter 125 for Primary Input Frequency:



On the Output Clocks tab, enter “12.288” for the requested Frequency of clk_out1, then press OK and OK again on the prompt.



Press Generate on the Generate Output Products window. After the core is generated, take a look at the Sources hierarchy, and verify that you see the mclk_gen as a child module of the ssm2603_i2s module:



In the code starting on line 58 in the ssm2603_i2s.vhd file, note that clk_in1 is connected to our 125MHz input clock, clk_out1 is connected to a signal called mclk_sig. This mclk signal is used for all of the processes in our design – it is a genuine, buffered clock signal and is ok to use in the rising_edge function. Note that Vivado may continue to flag an error on line 58 but it should still compile.

The locked output is not being used, but it indicates that the PLL output has stabilized to the correct frequency (which could take several microseconds from power on). Some designs may need to wait for this signal to go high. The reset input allows us to reset the PLL, which is also not used in this design.

Part C: Lab assignment overview

In this lab, you will be providing a square wave input of varying frequency to the SSM2603 chip, and listening to the result through a pair of headphones. The frequency of the square wave is controlled through an internal signal we will call **tone_terminal_count**. This count will be assigned to 4 possible values, called COUNT_C5, COUNT_E5, COUNT_G5, and COUNT_C6. These are the four notes of a simple tune; each note should be played for one second.

Assignment 1 – Slow Clock

Implement a 1Hz slow clock, the same way as we did in lab 3. We will use this slow clock to control the finite state machine. **Use the mclk_sig signal as your clock for all of your processes and not clk!** mclk_sig is a 12.288MHz clock, so your slow clock counter should count to 12,288,000.

Assignment 2 – Generate Square Wave

Now we will set things up to generate square waves of different frequencies for the l_data and r_data signals. First, define a 7-bit unsigned signal called tone_terminal_count. Next, define a

signal called tone_counter that will count up to this terminal count, also 7-bit unsigned. Also, define four unsigned constants for for different notes (constants are like signals that never change values). The values for these constants are shown below:

```
--Signals for generating the audio square wave
signal tone_counter : unsigned(6 downto 0) := (others => '0');
signal tone_terminal_count : unsigned(6 downto 0) := (others => '0');
--Clock cycle counts for 200Hz, 400Hz, 800Hz, 1600Hz
constant COUNT_C5  : unsigned(6 downto 0) := to_unsigned(92, 7);  --48KHz audio rate, 523Hz signal
constant COUNT_E5  : unsigned(6 downto 0) := to_unsigned(73, 7);
constant COUNT_G5  : unsigned(6 downto 0) := to_unsigned(61, 7);
constant COUNT_C6  : unsigned(6 downto 0) := to_unsigned(46, 7);
```

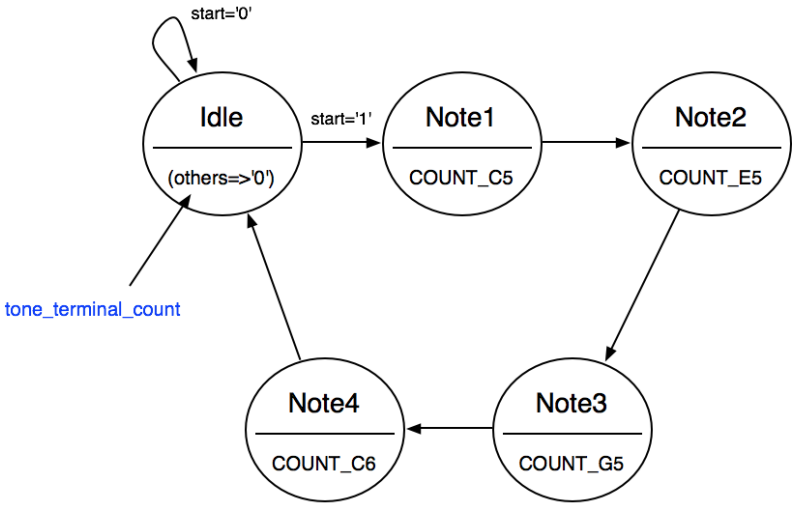
Next, in your architecture, create a process that will allow tone_counter to count up to tone_terminal_count and then resets to zero. Use ready as a clock enable so that the counter only updates when ready = '1'. I'll start it for you…

```
tone_counter_proc : process(mclk_sig)
begin
if rising_edge(mclk_sig) then
    if ready = '1' then
```

Lastly, we will use the tone counter to actually create the square wave for both l_data and r_data. You can use a when…else statement for both. Assign l_data and r_data to all 0's when tone_counter is less than tone_terminal_count / 2, else assign them to X"0FFFFF" (almost full amplitude output). This will create square waves between 0 and almost max at a frequency controlled by tone_terminal_count.

Assignment 3 – State Machine

We will implement a finite state machine that works as follows. The system will start in an idle state. When the "START" signal is '1', the system will move through four states, one per second, before returning to the idle state. Use the slow clock generated above to control the state machine. A state transition diagram is given below:



Note that we automatically move from Note1 to Note2 when the slow clock = ‘1’ (and similarly from Note2 to Note3 and so forth).

First, define your state type and state signal:

```
type state_type is (IDLE, NOTE1, NOTE2, NOTE3, NOTE4);
signal state : state_type := IDLE;
```

Then write a process to implement the state machine as shown above, started for you here:

```
state_proc : process(mclk_sig)
begin
    if rising_edge(mclk_sig) then
    case state is
        when IDLE =>
            if start = '1' then
```

Lastly, drive the **tone_terminal_count** internal signal to the appropriate value based on the state we are in using a when…else statement. For instance, when you are in the NOTE1 state, you should assign tone_terminal_count to COUNT_C5, and so forth. This will produce a different frequency note for each state. Now we’re done with the code for Lab4.vhd.

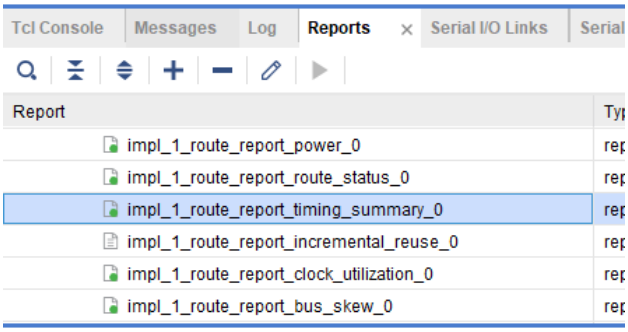
Assignment 4 – Constraints

You are not provided with an XDC constraints file for this lab. Rather, you need to create one and map the clk and pushbutton inputs as well as the inputs and outputs for the SSM2603 codec. Search the Zybo reference manual for the pin numbers of the mclk, bclk, mute, pblrc, and pbdat signals. Remember that our input clock is on pin **K17**. Make sure you define constraints for all of the inputs and outputs of Lab4.vhd.

We are also going to add **timing constraints** to our XDC file for this lab as well. Timing constraints are used to specify the frequencies of any clocks in our designs so that the synthesis tools can analyze and optimize the netlists to make sure they meet timing. The tools will check to make sure that all combinational logic circuits’ outputs will be stable for all input combinations before the rising edge of the clock. Basic timing constraints invovled just specifying the frequency of each clock in our design. For generated clocks, like mclk, we can look at the IP wizared to determine what the main clock is being multiplied by and divided by to generate mclk. In this case, I’ve found that clk is being multiplied by 23 and divided by 234 to get almost exactly 12.288MHz Just copy these lines into your xdc file:

```
create_clock -period 8 [get_ports clk]
create_generated_clock -name mclk -source [get_pins clk] -multiply_by 23 -divide_by 234 [get_pins mclk]
```

With these lines present, Vivido will analyze our design after it is implemented and make sure that all combinational logic meets timing for these clocks. After you implement your design, take a look at the timing report under Route Design in the Reports tab.



Around line 130, you should see the summary say that all of our timing constrants were met. You can look at this file further and discover that there is no *negative slack* found in the design; i.e., all of our combinational circuits compute their answers before the next rising edge of the clock.

```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints
    -------      -------  ---------------------  -------------------      -------      -------  ---------------------
     76.731        0.000                      0                  111        0.177        0.000                      0


All user specified timing constraints are met.
```

Part D: Audio Output System Requirements

1. The system shall have one start input (any pushbutton is fine for this)
2. The system shall produce audio output on the HP OUT jack on the Zybo board, as verified by plugging headphones into the port. (already implemented for you)
3. When the start button is pressed, the system shall output a four tone tune, spending one second on each tone, before returning to an idle state and waiting for the start button to be pressed again.

Part E: Connecting headphones

Connect your headphones to the **black HPH OUT** jack on the board, NOT the Blue, or Red jack.

Once you program your design on the board, note that you might have to hold down the BTN0 button for a second, until the next slow clock tick, then you should hear your four note pattern.

Appendix A: Audio Codec

An audio codec is a device or piece of software that codes or decodes a digital data stream of audio data. Software codecs are used to compress and decompress audio data stored in formats like mp3; hardware codecs refer to devices that have DACs and ADCs for conversion of binary data to and from analog audio data.

Appendix B: Sample Rate

If you look closer at the provided code in the Lab4.vhd file, you will notice that a square wave is generated based on the tone_terminal_count signal. The terminal counts that are output are based on the frequencies of four musical notes called C5, E5, G5, and C6. The codec requests data at a 48kHz rate (a standard rate for audio data). Every time this data is requested, the counter increments, and when the counter reaches half of the terminal value, it reverses to produce a square wave of the correct frequency at the 48kHz sample rate.

For extra credit, once you complete the main assignment, try extending into a longer, recognizable tune. This could be a great basis for your final project! To compute your own notes, take a look at http://www.szynalski.com/tone-generator/. Note the difference in the sound of a sine wave versus a square wave.

For *more extra credit*, you can create a test bench for your design and attach your simulation waveform to your design document. A test bench is not required for this lab.