Lab 8

Nexys A7 Peripherals: Audio and PWM

Introduction

Students will be introduced to the audio peripherals of the Nexys A7 board. This project contains a skeleton file to interface with the on-board audio amplifier that is connected to the audio jack. Students will gain an understanding of how Pulse Width Modulation (PWM) signals translate to audio signals. They will then generate tones using the on-board slide switches.

Download the lab files here[1].

Collaboration Policy

Lab groups may collaborate in completing the lab but may not consult other lab groups. Violating this policy is considered a violation of the Duke Community Standard.

Equipment

- AUX Cord for speaker (if needed)
- Speakers (wired earbuds with standard jack are fine)
- Nexys A7
- MG90S Servo motor

Lab Tasks

You must complete all parts of this lab to receive credit. Ensure that the TA marks the completion of the tasks in Sakai. To receive credit for this lab, you must complete:

To receive credit for this lab, you must:

1. Explain how to use the PWM signal generator to produce a waveform.
2. Explain how to use the PWM signal generator to control the volume of a signal.
3. Derive a formula to find the counter limit for a signal with an arbitrary frequency.
4. Calculate the counter limit range that correlates to the range of human hearing.
5. Generate tones using the slide switches.
6. Implement the microphone to overlay captured audio with generated tones.
7. Control a servo motor using the slide switches.
8. Record 5 seconds of audio and play it back using a button, if time permits.

[1] Link updated 2024-10-23

Lab Instructions

Pulse Width Modulation (PWM)

The Nexys A7 uses a single pin (A11) to control audio output. Instead of using a complex digital to analog converter, the Nexys A7 utilizes signal encoding known as Pulse Width Modulation (PWM), which uses a single bit to encode an analog value.

The signal alternates between being on (1) and off (0) and uses these transitions to encode the desired value. Encoding occurs by modifying the duty cycle, or percentage of “on-time,” per pulse. In this lab, the duty cycle is a 10-bit number ranging from 0 to 1023. For example, a duty cycle of 512 corresponds to a signal being on (1) half the time and off (0) the other half. Complex waveforms can be produced by modifying the *widths of each pulse* according to a function or recorded values.

An analog output voltage is produced using a Digital to Audio Amplifier that takes the duty cycle of the input signal and decodes it to a corresponding analog value. The amplifier averages the duty cycles over a window to provide smooth voltage transitions. The output voltage of the amplifier can be estimated using the following equation:

$$V_{out} \cong \text{Duty Cycle} * \text{VDD}$$

Where VDD is the maximum FPGA output voltage, 3.3V for the Nexys A7. Figure 1 shows an example of a sinusoidal signal and its approximation using a square wave, generated by a PWM signal. Note how the signal peaks during the high duty cycles, where the signal is high (1) for most of the time, and is lowest during the lowest duty cycles, where the signal is low (0) for most of the time.

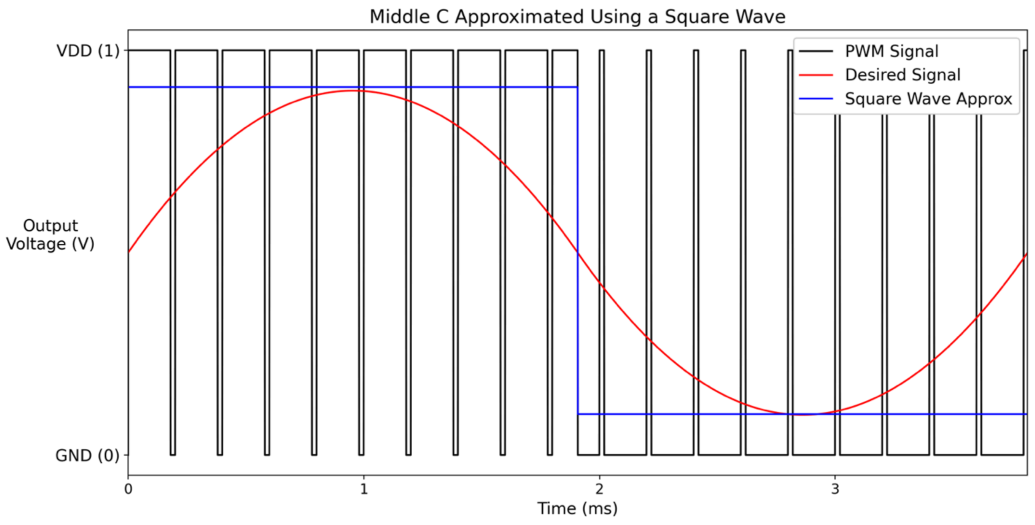
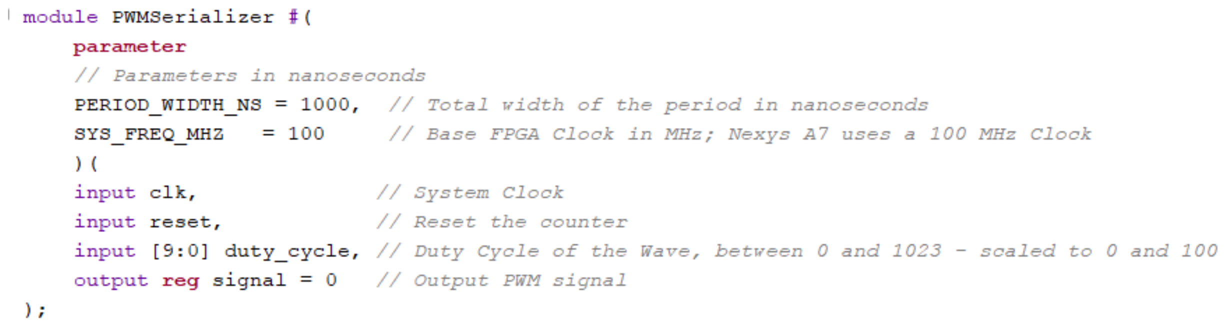


Figure 1: PWM Signal approximating a Sinusoidal Output Voltage

 Task 1: Explain how PWM signals can be used to generate a waveform.

Generating Tones

You are provided with a PWM Serializer module that produces a PWM signal with a specified duty cycle. The module is defined as follows:



```
module PWMSerializer #(
    parameter
    // Parameters in nanoseconds
    PERIOD_WIDTH_NS = 1000,  // Total width of the period in nanoseconds
    SYS_FREQ_MHZ   = 100     // Base FPGA Clock in MHz; Nexys A7 uses a 100 MHz Clock
    ) (
    input clk,               // System Clock
    input reset,             // Reset the counter
    input [9:0] duty_cycle,  // Duty Cycle of the Wave, between 0 and 1023 - scaled to 0 and 100
    output reg signal = 0    // Output PWM signal
);
```

The clk input expects the 100 MHz system clock available on pin E3. The reset input sets all counters to 0. The duty_cycle input expects a 10-bit binary value, between 0 and 1023, representing the desired duty cycle of the output signal. The signal output is a PWM signal with the specified duty cycle. The optional parameter SYS_FREQ_MHZ can be changed to reflect different clock frequencies. PERIOD_WIDTH_NS likewise can be changed to specify a different period width.

PWM signals are usually centered around a 512 duty cycle, 1/2 VDD, to allow for the signal to evenly oscillate in either direction. The volume of a sound wave is dependent on its magnitude of oscillation, the difference between the highest and lowest points; therefore, a signal that oscillates between 0 and 1023 duty cycles is louder than a signal that oscillates between 256 and 768 duty cycles.

 Task 2: How could you control the volume of the generated signal?

You can use a counter to modify the duty cycle of a signal at a specified interval. Modifying the limit of this counter will change the rate at which the value changes, thereby changing the frequency of a generated signal. Using a counter to drive a component at a slower frequency is known as clock division. Below is an example of using a clock divider to create a 1 MHz clock from the 100 MHz system clock:

```
reg clk1MHz = 0;       // Define the desired clock as a register
reg[17:0] counter = 0;             // Define the counter register
always @(posedge clk) begin        // Every clock cycle
      if(counter < CounterLimit)  // Half period of desired freq
            counter <= counter + 1; // Increment counter
      else begin
            counter <= 0;          // Reset counter
            clk1MHz <= ~clk1MHz;   // Toggle desired clock
      end
end
```

In the example, the new clock is defined as a register to toggle the signal using the always block. The counter threshold is half of the system clock frequency divided by the desired frequency, minus 1. We subtract 1 from the threshold as during the first cycle, the counter value equals 0. The 50th cycle occurs when the counter value equals 49. The new clock will toggle from 0 to 1 every 100 cycles, dividing the system clock by 100 and creating a 1 MHz signal.

A direct application of a clock divider is generating a square wave. A square wave is a basic approximation for a sine wave that alternates between two values, high and low. By modifying the clock divider skeleton above, a multibit value can be toggled when the counter threshold is reached, driving the duty cycle of the wave at a slower frequency than the system clock. Figure 1 shows an example of approximating a sine wave using a square wave.

Since we are toggling between two multibit values, in Verilog we cannot simply use the "~" operator to reassign the duty cycle value. Instead, we can use a mux to dictate which duty cycle to use. An example of using a mux to set the duty cycle using the clock divider example is shown below:

```
wire[9:0] duty_cycle; \\ Define input to the PWM Serializer module
assign duty_cycle = clk1MHz ? Max : Min; \\ Assign Duty Cycle
```

The duty cycle in the example is modified every 50 clock cycles, with the values repeating every 100 clock cycles. We have generated a 1 MHz square wave with a high value of 1023 and a low value of 0. By using this square wave as the input to the PWM Serializer module, we can generate an audio waveform that behaves similarly to our input square wave.

☐ Task 3: How can we calculate the limit for a specific frequency? Derive a formula for the counter limit in terms of the desired signal and clock frequencies.

(Young) humans can hear audio signals with frequencies ranging from 20 Hz to 20 kHz; however, the notes on a piano range in frequencies from 27.5 Hz (A0) to 4186 Hz (C8).

☐ Task 4: What is the range of the limit on your counter to stay in the human hearing range? In the piano range?

We will be using an extended C Major scale, a common musical scale. This scale is made up of 12 tones, starting with middle C (C4) and does not include sharp or flat notes. The notes in this scale, plus four other notes correspond to the following frequencies:

Note	C4	D4	E4	F4	G4	A4	B4	C5	D5	E5	F5	G5	A5	B5	C6	D6
Frequency (Hz)	261	293	329	349	391	440	493	523	587	659	698	783	880	987	1046	1174

You are provided with a skeleton file, AudioController.v, that creates registers with each note's frequency and instantiates the inputs and outputs for this lab. Using your counter, the formula from Task 3, and 4 slide switches on the FPGA, write code in the AudioController module to generate the 16 tones in the extended C Major Scale on the FPGA board. Each combination of the 4 slide switches must produce a unique tone. Make sure to add the constraints file (constraints.xdc) and memory file (Freqs.mem) to your project.

Verilog allows you to index memory arrays in a similar way to high-level programming languages. Thus you can obtain a frequency from the FREQs array by using:

```
FREQs[select]
```

☐ Task 5: Show your tone generator to the TA by uploading the bit file to your FPGA and playing the tones on your speakers.

Using the Microphone

Square waves, being crude approximations of sine waves, will not capture the small, individual transitions between output values and instead create large jumps between different output voltages. The sudden change between output voltages creates a tinny, video game-like timbre that will not sound natural. When mimicking real audio, it is important to capture the smooth transitions of real audio waveforms and avoid large jumps between values. This can be accomplished by modifying the PWM duty cycle to reflect the behavior of a sine wave, with many intermediate values to change the output voltage more precisely. Figure 2 shows an example of a PWM signal creating a better approximation of a sine wave.

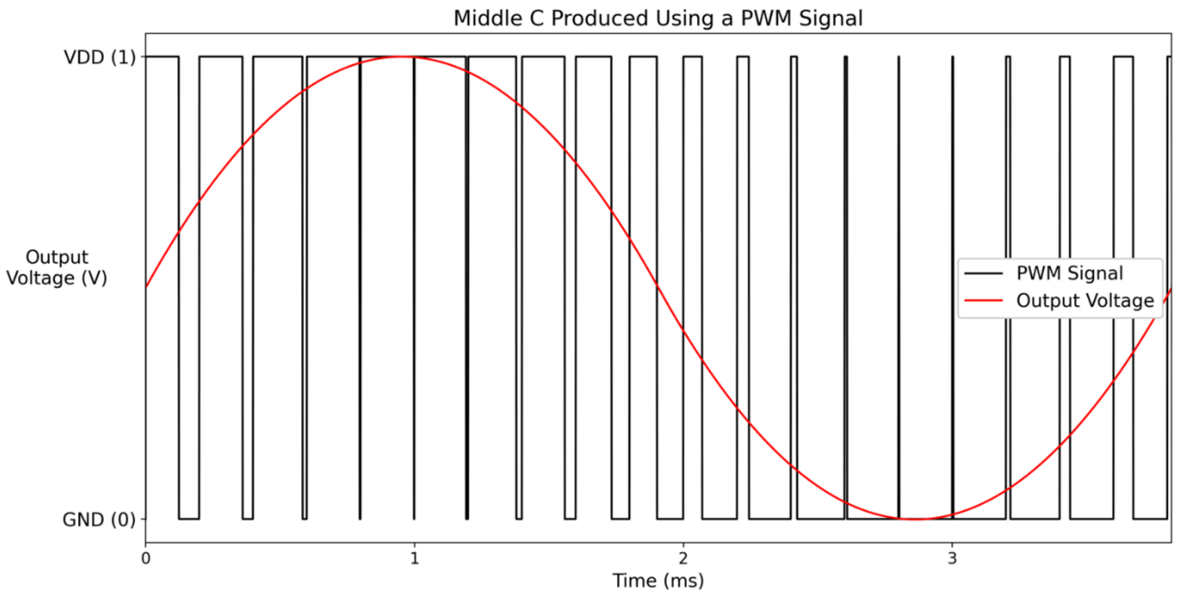
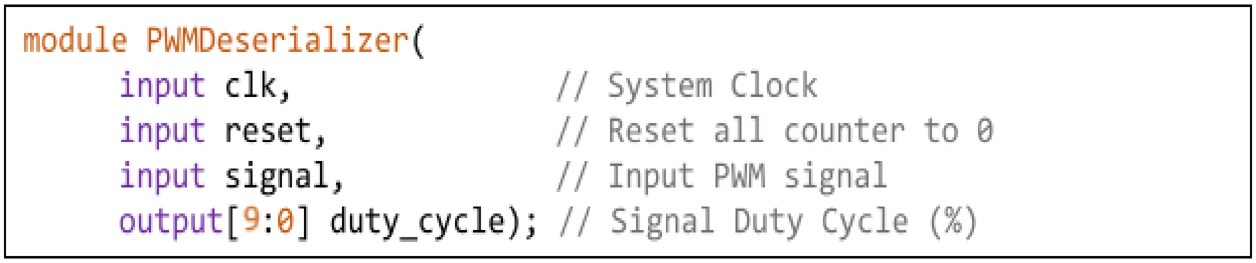


Figure 2: PWM Signal creating a Sinusoidal Output Voltage

The Nexys A7's built-in microphone outputs a PWM signal of its own using the more accurate approach above. The microphone requires a 1 - 3.3 MHz clock, and the PWM signal is updated on the edges of the provided clock. It is recommended you create a 1 MHz microphone clock signal using a clock divider.

The channel select output of the AudioLab module determines if the data is available on the rising or falling edge of the clock. It is recommended to capture the data using a register as the microphone output is not stable between clock edges.

You are provided with a PWM Deserializer module, which outputs the duty cycle of an input PWM signal. The module is defined as follows:



```
module PWMDeserializer(
     input clk,              // System Clock
     input reset,            // Reset all counter to 0
     input signal,           // Input PWM signal
     output[9:0] duty_cycle); // Signal Duty Cycle (%)
```

The clk input expects the 100MHz system clock. The reset input sets all counters to 0. The signal input expects a PWM signal where each pulse is updated every 1MHz (the microphone clock frequency). The duty_cycle output represents a 10-bit binary value between 0 and 1023. The duty_cycle output is updated after 100 microphone clock cycles; therefore, the duty_cycle output updates at a 10 kHz rate. The duty cycle output is meant to be read on the falling edge of a 10kHz clock.

You may use the below clock divider for your mic clock:

```
localparam SYSTEM_FREQ = 100000000; // 100 MHz
wire[5:0]  micThresh;
assign micThresh = (SYSTEM_FREQ / (1*MHz)) >> 1;


reg capturedBit = 0;
reg[31:0] micCounter = 0;
always @(posedge clk) begin
     if (micCounter < micThresh - 1)
          micCounter <= micCounter + 1;
     else begin
          micCounter <= 0;
          micClk <= ~micClk;
     end
end

always @(posedge micClk) begin
     capturedBit <= micData; // Assign mic input on the clock edges
end
```

Now, using your tone generator and the PWM deserializer module, overlay the microphone onto your generated tones. If every slide switch is off, then the microphone should be the only active audio signal. You must reduce the intensity of both signals by a factor of two to avoid saturating the speaker and distorting the audio signal.

☐ Task 6: Show your new karaoke machine to your TA.

Controlling a Servo

PWM can also be used for applications other than audio, a common one being servo control. A servo is a special kind of motor that has precise position control. The one you will be using today can rotate its output shaft between 0° and 180° based on the duty cycle of the PWM signal (Fig. 3). Note the range of the allowed duty cycles - not all values correspond to a servo position!

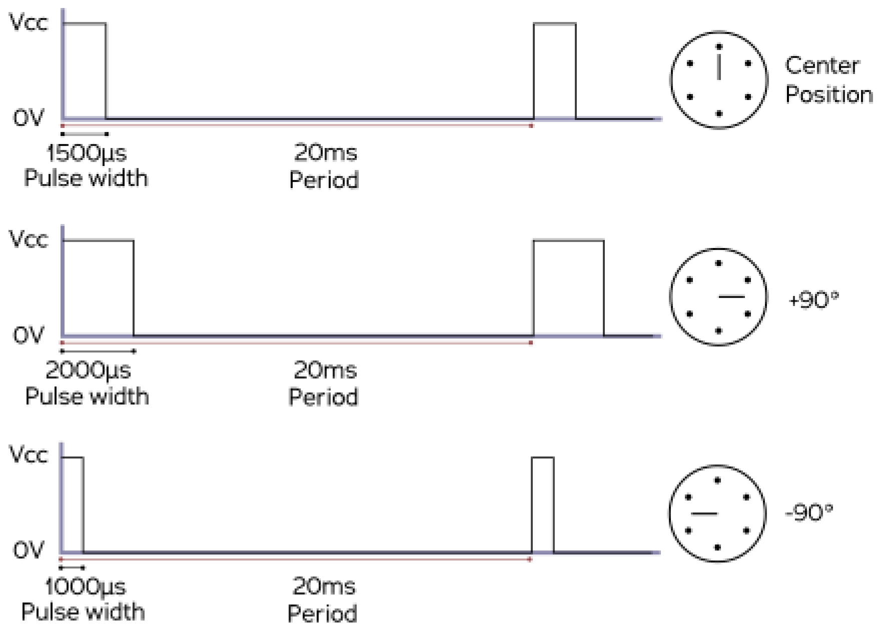


Figure 3: Duty cycles for a standard servo

Before wiring the servo to the FPGA, first turn off your FPGA by using the switch on the top left of the board or by unplugging the FPGA from the computer. Then, refer to Figure 4 to connect the servo to the FPGA. You will be using pin one on Pmod JB (Figure 5). This is defined in the constraints file as servoSignal. Do **not** turn your FPGA back on at this point.

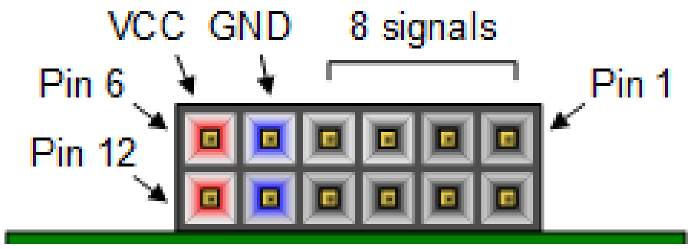


Figure 4a: Nexys A7 Pmod pinout

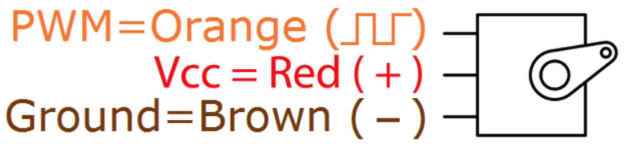


Figure 4b: Servo wiring (Orange Red Brown)

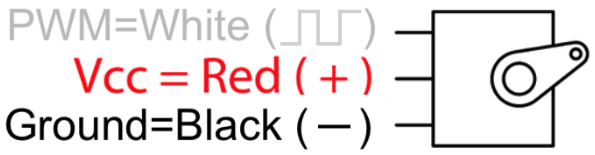


Figure 4c: Servo wiring (White Red Black)

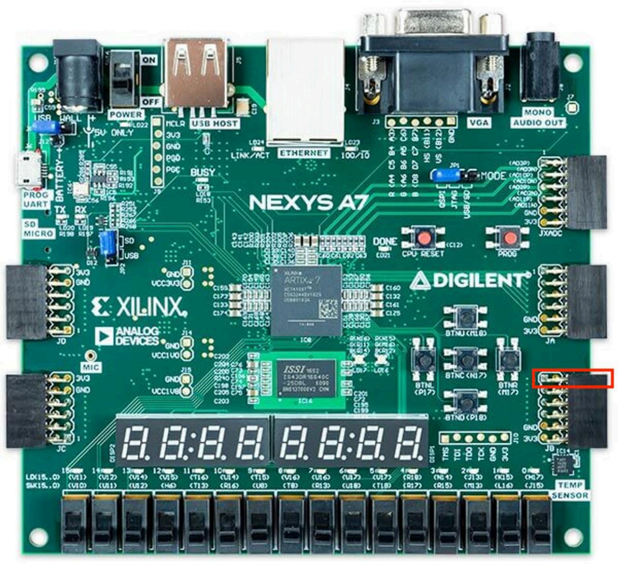


Figure 5: JB[1] location

 Task 7: Show your servo wiring to the TA

Once the TA has checked off your wiring, you can turn the FPGA back on. Modify ServoController.v so that you can use the slide switches to control the position of the servo. You may need to set ServoController.v as your top-level module - right click ServoController.v and select “Set as Top” from the dropdown (Figure 6).

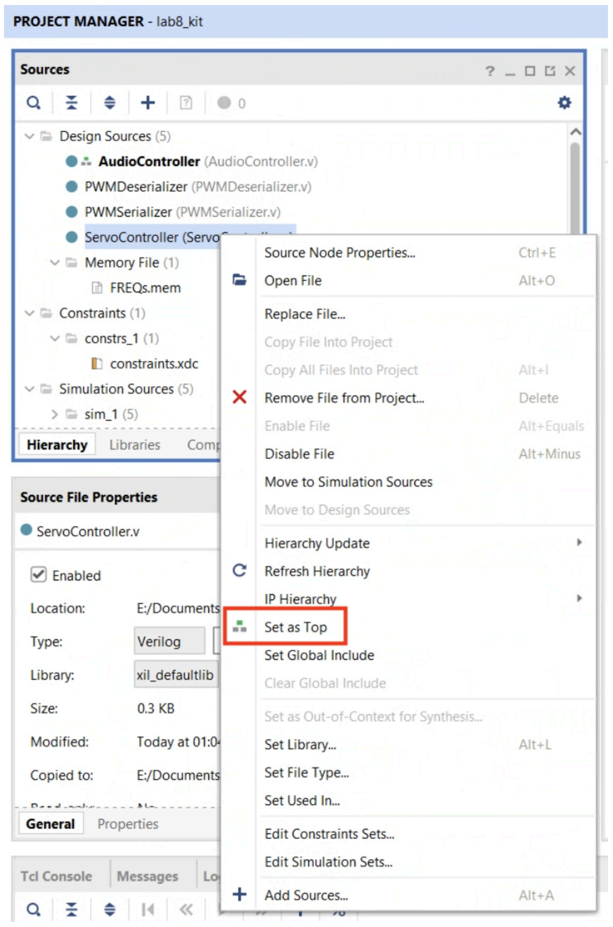


Figure 6: How to set a module as top

You will also need to change the optional parameter PERIOD_WIDTH_NS to reflect the 20ms period. The syntax for that can be found in Figure 7. **Note**: although not modified earlier, these parameters also exist for the PWMDeserializer.

```
PWMSerializer #(
    .PERIOD_WIDTH_NS(/*Enter value here*/),
    .SYS_FREQ_MHZ(/*Enter value here*/)
) ServoSerializer(

      . . .
);
```

Figure 7: Syntax for defining a module’s parameters

Hint: calculate what duty cycle values are allowed for a servo based on Figure 3. Keep in mind the duty cycle parameter is scaled.

☐ Task 8: Show your servo moving to your TA

Implementing Memory (Time Permitting)

You will now be utilizing the FPGA's built in ram to record your audio and play it back on command. Using the on-board RAM allows vastly more efficient storage of moderate amounts of data over fabricating memory elements out of the flip flops in the FPGA's LUTs (which we have been doing until now), something that could be quite useful in your final projects. The details of accessing the RAM are pretty involved, however, so we have abstracted a lot of this complexity for you here.

You were provided with a RAM module in the Lab 6-7 kit. This particular one is, by default, a 256-byte memory (256 8-bit storage locations). The module is defined as follows:

```
module RAM #( parameter
      DATA_WIDTH = 8,
      ADDRESS_WIDTH = 8,
      DEPTH = 256,
      MEMFILE = "") (
      input clk, // RAM Clock
      input wEn, // Write Enable
      input[ADDRESS_WIDTH-1:0] addr, // RAM Address
      input[DATA_WIDTH-1:0] dataIn, // Ram Data In
      output reg[DATA_WIDTH-1:0] dataOut); // Ram Data Out
```

The clk input is the clock speed at which to run the module. In this lab, the clk input expects a 10 kHz clock in order to capture the output of the PWM Deserializer. The wEn input determines if the RAM is being written to. The addr input addresses the data in the RAM. Its width is determined by the ADDRESS_WIDTH parameter whose default value is 8 bits. The dataIn input is the data to write at the specified address. Its width is determined by the DATA_WIDTH parameter whose default value is 8 bits. The dataOut output is the data stored in the specified address. Its width is determined by the DATA_WIDTH parameter whose default value is 8 bits.

Change the parameters so that it can store 5.6 seconds of recorded audio. Use the clock speed of the RAM and the recording time to calculate how many addresses you will need in your RAM and set the ADDRESS_WIDTH and DEPTH parameters accordingly. Also determine the DATA_WIDTH parameter based on the PWM modules. The MEMFILE parameter specifies an initialization .mem file. It is not necessary to set this, but it can help test your modules capability of outputting from the RAM. We have provided a `song.mem` file to do just that.

You will need to include a button to start the recording and playback of your recorded audio. It may also be useful to use the LEDs to indicate the progress of your recording. We have provided a song.mem file so you can test your playback.

☐ Task 9: Show your module to your TA by recording your voice and playing it back.

Clean Up Workstation

If you are in the lab, clean up your workstation, making sure to share any code written on the lab computer with your group for reference later.

Grading

Completing Lab Tasks: 100 points (pass/fail)