# Lab 3A Handout

Microphone: Computing FFT

## Lab Overview

Lab 3 makes use of a custom IP Core enabling the use of the Nexys4 DDR omnidirectional MEMS microphone. The goal of Lab3A and Lab3B is to build a Chromatic Tuner, which is a tool used by musicians to properly tune their instruments. First, the musician selects an octave (frequency range). Next the musician begins to play the instrument at a frequency, for example, 440Hz is A4, or the note A in the 4th octave. The Chromatic Tuner responds by displaying the closest note and how far the tone of the instrument is off from the note. To do this, we need to implement an efficient way of computing FFT to determine the frequency of the note being played. **This is the objective of Lab3A. For Lab3A, you need to demonstrate your FFT implementation by testing for input frequencies between 200 Hz and 5-10 KHz and displaying the note and its frequency on the terminal. The GUI and implementation of Chromatic Tuner will be completed in Lab 3B.**

The default software application provided to you samples the microphone input using a custom stream grabber port into a Microblaze buffer and displays the base frequency on an equal-tempered diatonic scale (A=440Hz). However, you will discover that the default FFT implementation is rather inefficient. You will need to improve the FFT code so as to accurately determine the input frequency over a large frequency range with a quicker response time.

## 1 Block Design

While the focus of Lab3A is on efficient FFT computation, it is a good idea to add all the hardware peripherals you will need in Lab3B now itself. You can do the software configuration of the peripherals (not being used in Lab3A) in Lab3B. You should start from your Lab 2B project.

### 1.1 Rotary Encoder

Make sure the Vivado project is properly configured for the rotary encoder, just like it was in Lab 2B.

### 1.2 LCD Display

Make sure Vivado project has a properly configured interface for the LCD.

### 1.3 Customizing Microblaze

To build the Chromatic Tuner it is useful to further customize the Microblaze processor to increase its arithmetic performance. Screenshots of a recommended configuration for the Chromatic Tuner are included at the back of the Lab Write Up. Right click on your Microblaze and copy the suggested configuration.

## 1.4 Microphone

The Microphone integrated circuit (IC) consists of two inputs and one output. The output signal is a Pulse Density Modulated single bit stream **mic_data** which transmits at the rate of the input **mic_clk**. The **nic_lr_sel** signal is a phase selector that allows both left and right channels of audio to be transmitted on a single wire. We'll leave that constant once set to a default value. The microphone is an Analog Device ADMP421 chip which has a high signal to noise ratio (SNR) of 61dBA and high sensitivity of -26 dBFS. It also has a flat frequency response ranging from 100Hz to 15kHz. The digitized audio is output in the pulse density modulated (PDM) bit stream (synchronized to the input clock signal).

A *Hierarchy* hardware driver is created in the Vivado Block Design to accommodate the custom IP (Intellectual Property) Cores used to interface to the Analog Device ADMP241 chip:

First a filter/decimator block (in nopll_mic_block) filters the incoming 3.125Mb/s PDM bit stream down to a 48kHz 27-bit integer sampled data stream. The specific form of PDM is 4th order Sigma-Delta. The output sample rate, more precisely, is 48.828125 kHz or $100MHz/32/64$. This module also generates the clock to drive the microphone.

Next, we must get this stream into the processor without losing random samples. For this a hardware stream_grabber block is used. This accepts input samples from the filter block, but discards them by default. When triggered by the processor, it grabs a contiguous sequence of samples into its internal buffer. It always fills the buffer (up to 4096 samples). The processor can poll the device to see if enough samples have been received. When re-triggered it will start grabbing a new block of samples.

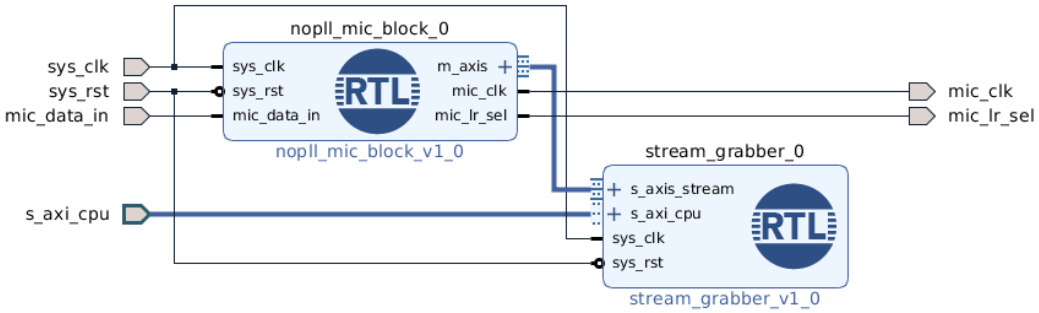



Figure 1: The microphone block consists of a microphone control block that outputs a continuous stream of data, and a stream grabber that, when commanded by the processor, grabs a section into its buffer to be read back into the processor buffer. **IMPORTANT**: Make a list of the module interconnections and hand it in with your lab report. If these connections are completed incorrectly and you managed to generate and export a bitstream, you may see the error in SDK "**Unable to Stop Processor**". Power off your board, close xSDK, fix your Vivado Design, and regenerate the bit-stream.

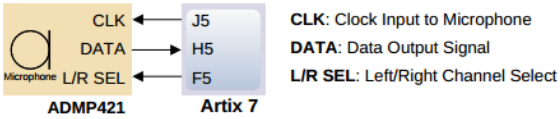



Figure 2: The Nexys4 onboard microphone requires the connection of 3 pins. mic_clk, mic_data and mic_lr_sel. Make these 3 pins external, by right clicking the ports and selecting 'Make External'. Edit the Nexys4_master_DDR.xdc file to connect to the pins.

### 1.4.1 Creating the Microphone Hierarchy Block

1. Add the provided nopll_mic_block.v and stream_grabber.v to your project using the "Add sources" in the file menu.
2. Add the nopll_mic_block to your design, and also a stream_grabber. To add a block from Verilog source open the block diagram, and in the sources window (on the left) right click on the source file and choose "Add to block diagram".
3. Choose the tool tip "Select Area". Completely select both blocks without wiring.
4. Right click and choose 'Create Hierarchy'. Name this sub-diagram mic_block.
5. Connect the AXI stream interface between the two modules: from m_axis on the nopll_mic_blk to the s_axis_stream on the stream_grabber.
6. The option to "Run Connection Automation" should appear as a banner a the top of the diagram. option should appear. Use this to connect s_axi_cpu from the stream_grabber to the interconnect and to connect sys_clk to the processor clock (ui_clk from the MIG). **Ensure this clock is set right, the wizard might choose the wrong default.**

   This should also make sys_rst connect to the peripheral_aresetn of the processor reset, same as your other peripherals. Ensure that this is the case.
7. Make the mic_data_in, mic_clk and mic_lr_sel ports external.
8. Add the constraints given in mic_constraints.xdc to your main XDC file to hook up the microphone pins externally.
9. If the pin names from "Make External" end up with a trailing "_0" remove this so the names agree with the constraints.
10. Go to the Address Editor tab and ensure mic_block/stream_grabber_0 has been assigned an address. If not it will show up under an "Unmapped Slave" sub-category. If so, right-click on it and choose auto-assign address.
11. After the usual cycle of "Generate Bitstream", "Export Hardware" and "Re-generate BSP Sources", the sample code should now compile. If it does not, please check the names assigned to stream grabber, as its address is hard-coded into stream_grabber.c. Find the right version of XPAR_[Block Name]_[Instance Name]_BASEADDR in xparameters.h and update stream_grabber.c
12. In SDK, do not forget to run the linker script, map all sections of the memory to the DDR (mig_7series_0_memaddr...) and **not** the BRAM. Also make sure that the the heap and stack sizes are 4096 (4 KB).

# 2 Octaves

The frequencies detectable by our microphone range from tens of hertz to the kilohertz range. Octaves are used to divide this wide range of frequencies into smaller groups of frequencies on a logarithmic scale that are labeled 0-9. Each group contains 12 notes:

$$C, C\#, D, D\#, E, F, F\#, G, G\#, A, A\#, B$$

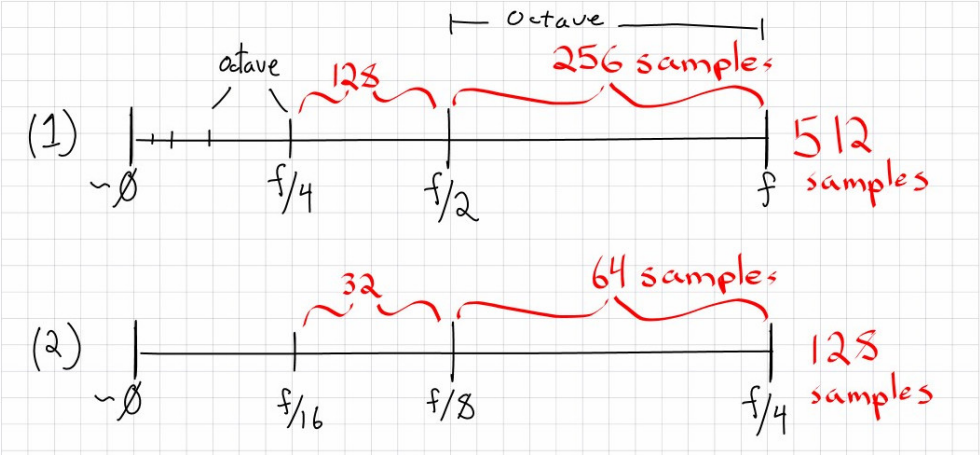
The frequency corresponding to a note is described by the following formula:

$$f = 440Hz * 2^{\frac{(n-9)}{12}+k-4}$$

where $n$ is the number corresponding to a note (C is 0, A is 9, A# is 10, etc.) and $k$ is the number corresponding to an octave. So $A4$ is 440 Hz, $A1$ is 55 Hz, $C4$ is 261.626 Hz etc. The use of octaves to identify the note is very useful when calculating the FFT and can help decrease the time to compute the FFT by allowing a smaller, lower resolution FFT to be used selectively. (ref: Scratch Wiki list of MIDI notes)

## 3 Fast Fourier Transform

The Fast Fourier transform places recorded samples into frequency bins. The FFT is calculated up to the Nyquist frequency (half the sampling frequency) which here is 24.4140625KHz. The resolution of the frequencies placed in the upper half of the range is much greater than those placed in the lower half of the range. This is shown in the figure below. Looking at Fig. 3:plot(1) If we are analyzing the raw samples from the microphone, the upper half would contain frequencies ranging from 12 KHz to 24 KHz. This means that if we are trying to detect a lower frequency, like 100Hz, only a few samples contribute. Furthermore the microphone flat response begins to fall off at 15kHz and there is very little music which is fundamentally tuned in this range, we are making little use of the most accurate part of the FFT in the default setup.




There is a way we could fix the effective resolution, and potentially increase the speed of the FFT. This is shown above in Fig. 3:plot(2), which computes the FFT using 128 samples. The 128 samples could either be a decimated read from the buffer, where we take only every 4th value, or the samples could be averaged values from the original 512 samples. 4-way averaging the samples makes an effective 128 samples at $48KHz/4 = 12KHz$. Unlike decimation, averaging also acts as a low-pass filter to lower the noise floor by removing higher frequency components as well as their noise. Although the bin spacing is identical, the noise is lowered which improves the accuracy of the parabolic fitting to estimate the measured frequency.

On the other hand, we could grab, say, 2048 samples at 48MHz and average down to 512 samples at 12kHz to get higher resolution at 3-6kHz, or even 128 samples at 3kHz for a computationally efficient look at the 750Hz-1.5kHz range. This techniques improves the noise and improves the FFT bin-spacing as well. Of course, the longer sampling lowers the speed at which one can report the higher resolution result, increasing measurement latency. The sample decimation lowers the Nyquist rate for the measurement, so if the frequency goes beyond the Nyquist bound (1/2 of the effective sample frequency), aliasing will occur.

The above discussion shows that there is a considerable space of trade-offs in resolution vs. run-time. In a practical application, any delays over 50mS are noticeable, leading to design trade-offs in tuners. If the goal is extreme accuracy, i.e.

a few cents at very low frequencies, sample times of several seconds are necessary. (Consider that the spacing of the lowest note on a pipe-organ $C$ (32.7Hz) to $C\#$ (34.6Hz) is only 1.9 Hz). Alternatively, rapid arpeggios in music can exceed 20 notes/second but such notes can only be audibly distinguished if pitched into or containing harmonics at higher frequencies. At such rates, the tuner readout latency also becomes an issue.

The audio amplitudes as captured by the microphone are sampled at 48.828125 KHz (100MHz/2048), with a default of 512 samples per read of the buffer. In the raw output, all frequencies are present. To increase the resolution of the FFT at a particular octave, average the samples together to optimize the FFT for the frequency range your musician is playing a note for.

Lab 3A provides default source code to verify functionality of the microphone. The demo code tells the stream grabber to start grabbing data from the microphone, waits for it to fill its buffer, then copies that into a software buffer. This occurs in read_fsl_values(). The sample code provided in **src/fft.c** computes the Fast Fourier Transform (FFT) and xil_printf() outputs the estimated base frequency. The FFT response time for the default code is too long and must be improved. To improve the code it is necessary to to conduct a performance analysis. There are three sections of code in particular to focus on.

1. Focus on the timing around **read_fsl_values()**, which is the function responsible for reading the 512 microphone sample from the Microblaze buffer.

   (a) The existing code starts the sample grabber, waits until it gets 512 values into the grabber's buffer, then copies them into a software buffer. In fact the grabber keeps going up to 4096 samples but the software doesn't need to check this. The hardware is fixed at the microphone's sampling rate so lower effective sampling rates must be achieved in software.

   (b) The existing code gets 512 values into the grabber, then reads them into the microblaze. The input is fixed point, and the code is not clever about when conversions to floating point representations are made.

   (c) Given this, experiment with different effective sample frequencies (after decimation) to speed up your FFT and increase the accuracy of your design. Review the earlier section on the FFT and octaves, and think about how you could change the code of the FFT to make sure the target octave falls in the upper half of the FFT.

   (d) Note that the grabber, once started, runs in the background. Thus one can either pipeline the system (start the next grab while computing the current FFT), or use the non-blocking sample checks in a GUI to be more responsive while waiting for data.

2. Study the FFT code, and focus on finding small changes which result in noticeable speed-ups of your code. The original code uses floating-point arithmetic. Rewriting the FFT to use a fixed-point FFT (which would remove the need for floating point numbers altogether) would allow for a much faster FFT, but this is tricky to get right – e.g. a 512 sample FFT of 24-bit data can overflow a 32-bit fixed point FFT. Alternatively, the current implementation is quite redundant in running function calls on the same operands- these calls should be cached or pre-computed into tables.

   This class of optimization is particularly important since if you can run a 512-point FFT in the time it would take someone else to run a 128-point FFT you can have that extra speed without the accuracy trade-off of a smaller FFT.

Your goal should be to build the base for a fluid interface for the Chromatic Tuner, that has a quick response time with reasonably high accuracy.

# 4 Performance Analysis

The simplest way to do performance analysis is to measure time with the AXI timer. For more detailed analysis, a method called "profiling" is used. Conventional profiling requires additional code to be compiled into functions to keep track of entry and exit statistics while the code is run. Instead, we add an assembly language trick to sample where the program counter is as a simple way to do profiling. We call this trick the "performance monitor". It allows us to measure the relative proportion of time spent by the program in different sections of the code.

## 4.1 Performance Monitor tool

A performance monitor is a small interrupt driven program, which runs alongside your main program. The performance monitor interrupts on a regular basis, and maintains a log of where the current hardware program counter is at the time of the interrupt.

Specifically it maintains a table of how many times it is found between addresses denoting the lower and upper bounds of the code of each of the procedures in the base line code.

Your performance monitor code needs to be fast and as small as possible, because it must fit in the program space, along with that of the program under analysis.

## 4.2 Performance Monitor Details

In order to analyze the performance of your code, you need a periodic interrupt. When you enter the interrupt handler, you can see where the code was, by checking the return address of the interrupt. This address is stored in register 14 and can be accessed via inline assembly. The necessary code for this is:

$$uint32\ a;$$
$$asm(\text{"}add\ \%0, r0, r14\text{"} : \text{"} = r\text{"}(a));$$

The above code adds zero to register 14 and stores the result in **a**. Once the address is found, compare it to the address in the symbol table of your executable, **your_xsdk_name**.elf . The executable is located in

*YOUR_VIVADO_NAME*.sdk/SDK/SDK_Export/*YOUR_XSDK_NAME*/Debug

To access the symbol table, open the elf file. If you are on Linux, you can also open your regular terminal, navigate to the folder containing your .elf and type:

nm **your_xsdk_name**.elf

Another alternative is to use **objdump**, which has additional options for viewing more information about your code. To see the symbol table using **objdump** type:

objdump -t **your_xsdk_name**.elf

### 4.3 Performance Optimization

Use the above analysis and your knowledge of what each segment of code is doing to lower the 512-point FFT latency below 25 mS, using only software changes in the C code. Hint: This is easier than it sounds. Look for small pieces of code that cost large numbers of cycles and think about what is really needed. Typically one can achieve 8-10 mS for floating point versions and 1-2mS for fixed-point making only software changes to the project. Minimizing sampling length (latency) creates an accuracy trade-off, especially for low frequencies. Often, a rough FFT at full bandwidth is run on a small number of samples to check if sample averaging is safe (will not alias).

# 5 Reporting

Write a concise lab report on the following topics:

1. Include a Block Design Schematic of your Vivado Project.
2. Include a list of the Mic_Block internal wiring.
3. Include a description of the steps taken for optimizing the original FFT code.
4. Was there a difference in the FFT computation for low and high frequencies?
5. How does the performance of your FFT code compare to that of the default code?
6. Include results from your performance analyzer. The performance analysis should consist of an estimate of the fraction of total execution time spent in each program code module. Estimate the average time you spend in your profile code per sample interrupt (times the size of its code segment). This is the 'goodness' metric for your profiler.

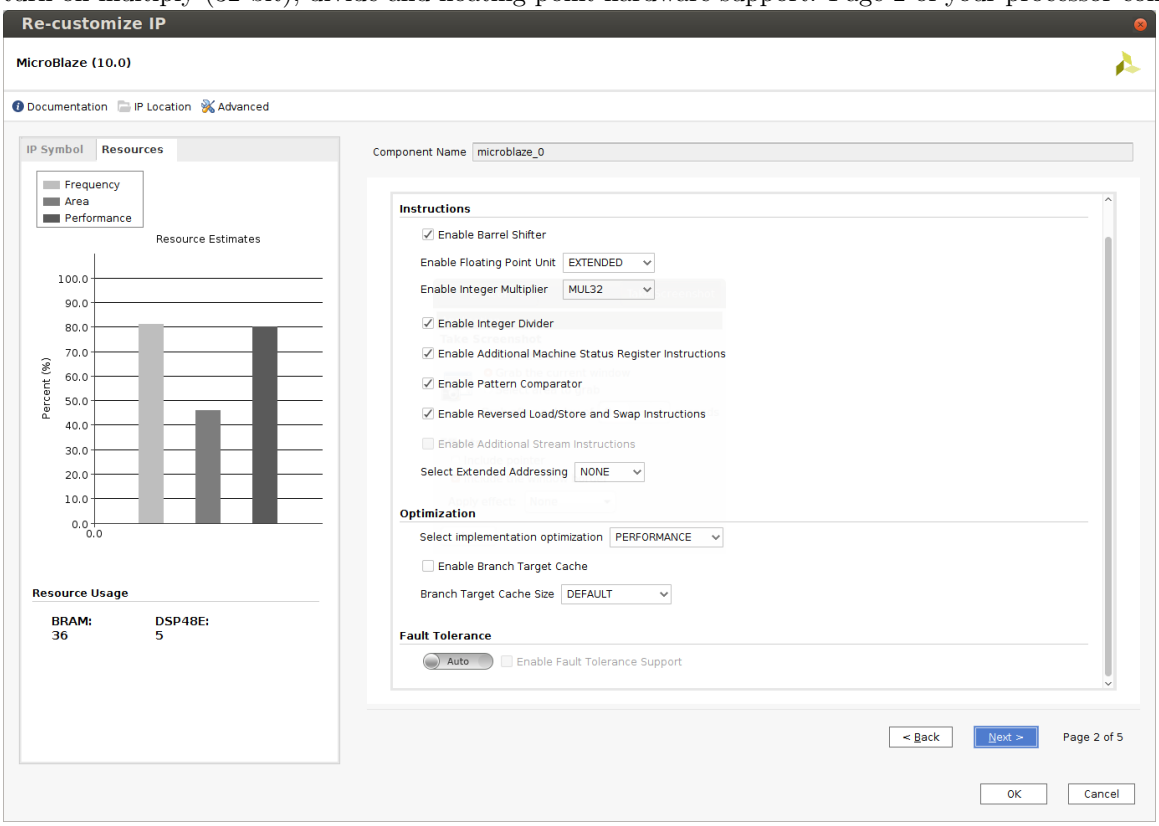
# 6 Video Submission Guidelines

To test the code, you can use a web or phone application to generate and play frequencies into the onboard microphone. Submit a short video (less than 60 seconds long) displaying on the console the computed frequency and time to do the computation with the following input frequencies (sine waves):

- 200 Hz
- 440 Hz
- 880 Hz
- 2 KHz
- 3.6 KHz
- 5 KHz

# 7 Appendix: Microblaze Configuration

Review your Microblaze configuration for suitability for your final project. To achieve good performance for your chromatic tuner, it is helpful to add arithmetic performance features to your processor. In particular, you should go to reconfigure your processor and turn on multiply (32-bit), divide and floating point hardware support. Page 2 of your processor config should look like this:




# 8 Appendix: Trouble Shooting

If you are sure that your hardware setup is correct, but cannot get the code to identify the components, you can backup your code, delete the sdk folder in your project folder, and export hardware again. By doing so, you delete all the old hardware platforms and bsp files.