ELEC3342: Digital System Designs
First Semester, 2022–23

Homework 3 (r1.1)

```
Due:
```

- `Part (A) -- 24 Nov, 2022, 11:59pm`
- `Part (B) -- 24 Nov, 2022, 11:59pm`

Instruction: Submit your answers electronically through Moodle.

There are 2 major parts in this homework. Part A includes questions that aim to help you with understanding the lecture materials. They resemble the kind of questions you will encounter in quizzes and the final exam. Some of these questions may also ask you to implement and test small designs in VHDL. This part of homework must be completed individually.

Part B of this homework contains a mini-project that you should work in groups of 2. Your submitted work will be graded by an auto tester and therefore you should make sure your submitted files conform to the required format.

The following summarizes the 2 parts.

Part	Type	Indv/Grp
A	Basic problem set	Individual
B	Mini-project	Group of 4

In all cases, you are encouraged to discuss the homework problems offline or online using Piazza. However, you should not ask for or give out solution directly as that defeat the idea of having homework exercise. Giving out answers or copying answers directly will likely constitute an act of plagiarism, which is a serious offence.

Part A: Problem Set

A.1 Simple Artificial Synapse

Recent success in AI and machine learning is built on top of decades of research in understanding how the human brain works. In this question, you will create a simple artificial synapse (SAS) that models a human neuron with a simple mathematical model.

A basic model of how human brain works is that spikes of signals are transmitted across the neural system. With the arrival of each spike, or electrical pulse, the surface potential of the synapse of a neuron increases slightly. On the other hand, when there is not spiking activity, the surface potential decays over time. If the surface potential exceeds certain threshold, the synapse *fires a pulse* and returns to a resting state.

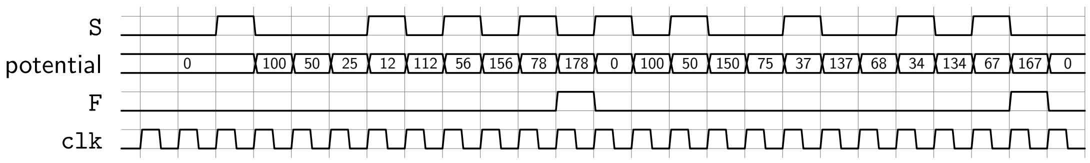
For your SAS, you will model the neuron as a circuit that operates with the following rules:

- SAS has one input `S` and one output `F`.
- A spike is represented as one cycle of '1'. In other words, when there is no spiking activity, `S` is zero and output `F` should be kept at zero.
- With each spike SAS receives in `S`, its internal potential is incremented by a value of 100.
- When there is no spike input, SAS internal potential decreases by half every cycle. That is, let p_i be its internal potential at cycle i, then

$$p_{i+1} = \lfloor p_i/2 \rfloor$$

- SAS fires if the internal potential exceeds the value of 160.
- Once SAS fires, the internal potential is reset to the value of 0.

The following demonstrates the operation of an SAS:



A.1.1 Implement the design of SAS by hand. You can use any of the building blocks covered in class. Label clearly your input, output, as well as the bus notation and any bit slicing you have performed. Submit a short report with the circuit design and a short description of how it operates.

A.1.2 Implement your design in VHDL. Your should use synthesizable VHDL only. Submit your VHDL file.

A.2 Circuit Pipelining

In theory, any feedforward circuits can be sped up by pipelining the datapath. For instance, Figure A.1 shows several circuits with different pipelined datapaths that perform the same mathematical function:

$$Y = A(B - C) + (B - C)(A + B)$$

The diagram shows a correctly pipelined circuit that performs the same function except it has an additional pipeline stage and thus incurs an additional cycle of latency. The diagram also shows an *incorrectly pipelined* circuit that looks similar but produces wrong answers.

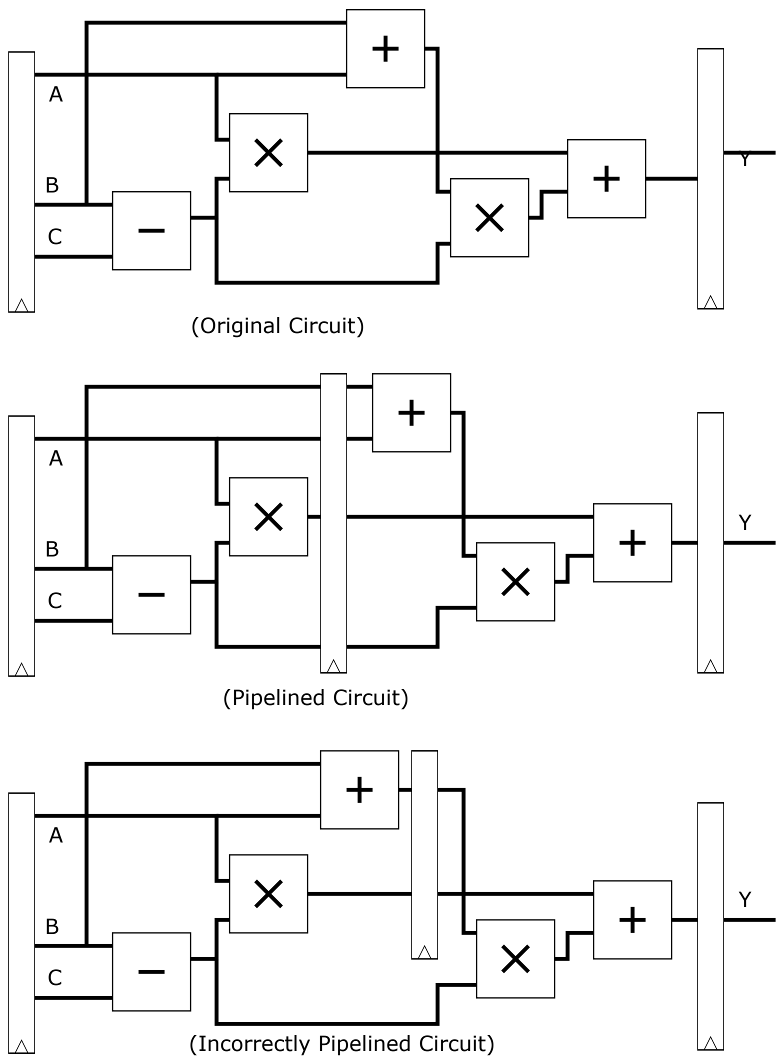


Figure A.1: Pipelining a feedforward circuit. (a) original circuit; (b) with 1 pipeline register; (c) incorrectly pipelined circuit.

Let the setup and hold time of each register to be t_{setup} and t_{hold} respectively. The register has a clock-to-Q propagation delay t_{pcq} and a clock-to-Q contamination delay of t_{ccq}. Also, let the propagation and contamination delay of an adder/subtractor be t_{pd} and t_{cd} respectively. Finally, let the propagation and contamination delay of a multiplier be $5t_{pd}$ and $5t_{cd}$

A.2.1 For each of the circuit (both (a) and (b)) in Figure A.1, determine the following:

(i) The maximum *clock frequency* that the circuit may run at without violating any of the setup and hold time constraints.

(ii) The minimum clock frequency that the circuit may run at without violation any of the setup and hold time constraints.

(iii) Throughput of the circuit.

(iv) Latency of the circuit.

Assume the values of all input arrive at every cycle, i.e., the values $A[n], B[n], C[n]$ are available at the *input* of the (leftmost) input register in cycle n where $n = 0, 1, \ldots$.

For (ii), the answer may be infinitely slow, i.e., no constraint on minimum. For (iii), processing *throughput* is measured as the number of answer ($Y[n]$) produces per second. For (iv), processing *latency* of the circuit is L if output $Y[0]$ is ready at the *output* of the final register in cycle L.

A.2.2 Explain, by tracing the number of added latency to the circuit that affect the index n, why the *incorrectly pipelined* circuit does not produce the same result?

A.2.3 Redesign the circuit in Figure A.1 such that it can run with maximum processing throughput (number of results per second) while keeping the overall function unchanged. You may make zero or more of the following changes:

- Add or remove 0 or more registers
- Add or remove 0 or more Adder/Subtractor/Multiplier
- Rearrange the Adder/Subtractor/Multiplier connection

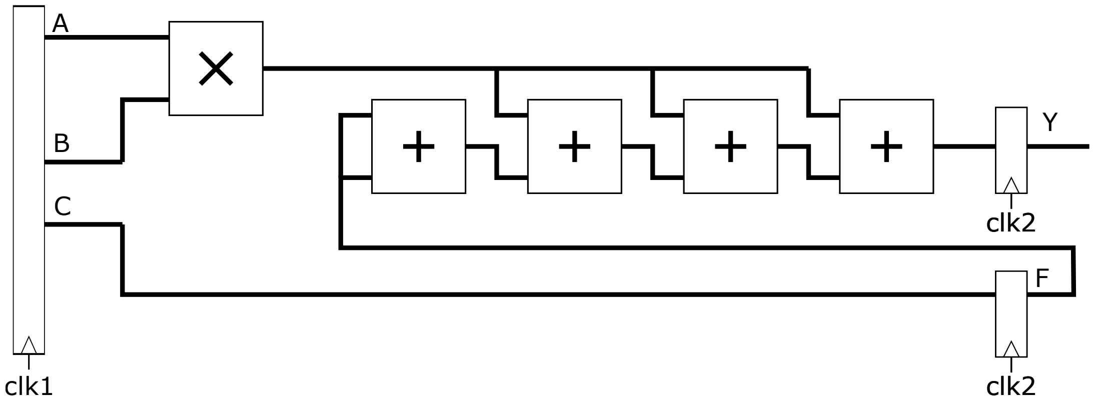
Submit a schematic of your new circuit, and determine its throughput.

A.3 Synchronizer MTBF

For your next design, you would like to build a synchronizer that can receive asynchronous inputs with an MTBF of 1 year. You are building the synchronizer using a sampling flip-flop with $\tau = 100\,\text{ps}$, $T_0 = 110\,\text{ps}$, and $t_{\text{setup}} = 70\,\text{ps}$. The synchronizer receives a new asynchronous input on average 4 times per second. What is the required probability of failure to satisfy this MTBF? How long would you have to wait before reading the sampled input signal to give that probability of error? [*based on DDCA 3.37*]

A.4 Circuit Timing

(*adopted from a past exam*)
The following shows the circuit that is currently the performance bottleneck of your system.



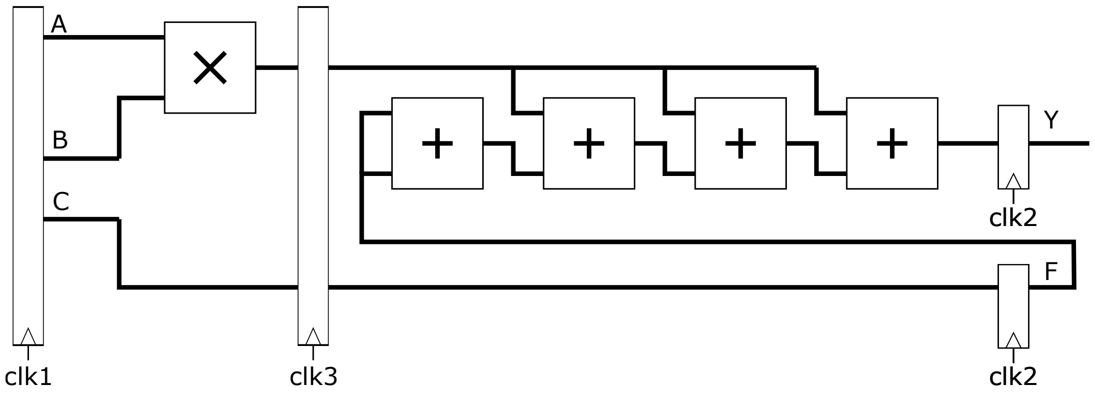
The following table lists the timing properties of this circuit:

Parameter	**Value** (ps)	**Remark**
t_{pcq}	69	C-to-Q Propagation delay of a register
t_{ccq}	38	C-to-Q Contamination delay of a register
t_{setup}	45	setup time of a register
t_{hold}	15	hold time of a register
$t_{\text{PD},add}$	450	Propagation delay from any input to any output of an adder
$t_{CD,add}$	65	Contamination delay from any input to any output of an adder
$t_{\text{PD},mult}$	1500	Propagation delay from any input to any output of a multiplier
$t_{CD,mult}$	50	Contamination delay from any input to any output of a multiplier

A.4.1 Assuming there is no clock skew between `clk1` and `clk2`, what is the maximum clock frequency that this circuit can run at? Show your steps by first finding the critical path of the circuit. *Hint:* Be aware of the feedback path with the value F.

A.4.2 If you want to run the circuit at 300 MHz, what is the maximum clock skew allowable between `clk1` and `clk2`.

A.4.3 To further improve throughput of the circuit, you have decided to pipeline the circuit as follows:



Assuming there is no clock skew among `clk1`, `clk2` and `clk3`, what is the maximum clock frequency that this circuit can run at? Show your steps by first finding the critical path of the circuit.

A.4.4 You now have the option to connect either `clk1` or `clk2` to the pipeline register in place of `clk3`. There is no skew between registers that are connected to the same clock. The maximum skew between `clk1` and `clk2` is 100 ps in either direction, i.e. `clk1` may be faster or slower than `clk2` by *up to* 100 ps. Which clock (`clk1` or `clk2`) should you connect to the pipeline register in order to maximize the clock frequency that this circuit can run at? Explain your choice quantitatively by showing the effect of clock skew in both cases.

A.4.5 Now, you may insert as many pipeline register as you want. For each of the additional register that you have added, connect it to either `clk1` or `clk2`. What is the maximum frequency you can achieve?

Part B: Mini-Project

The goal of this part of the homework is to complete the Music code decoder project and implement the design on an FPGA as shown in Figure B.1

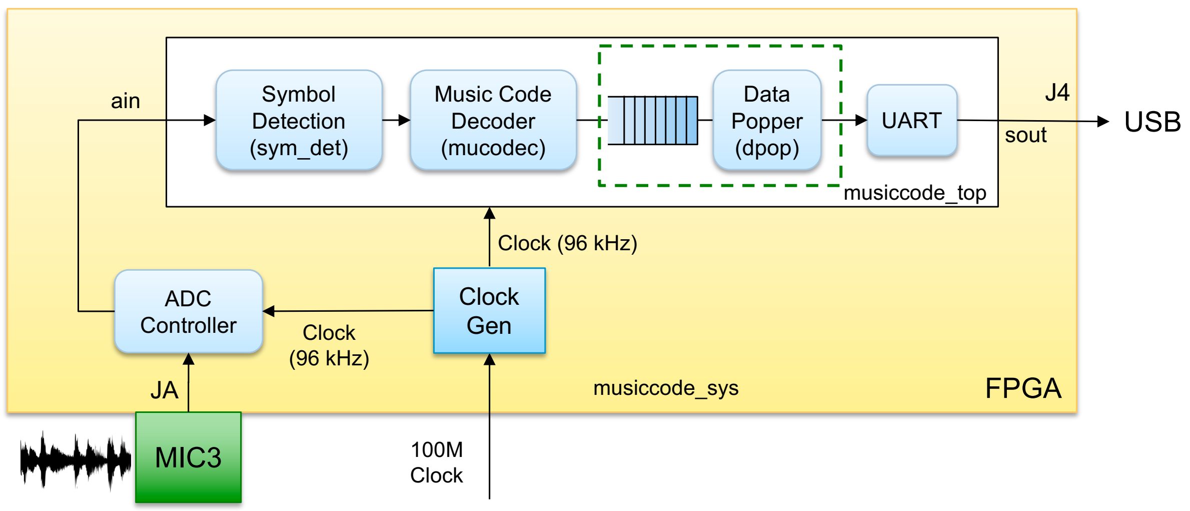


Figure B.1: Top-level block diagram of the entire design as implemented on an FPGA.

To make this project manageable, you should approach it by achieving the following milestones:

- **Milestone 1**: Full Music code decoder simulation
- **Milestone 2**: FPGA implementation
- **Milestone 3**: Extension

B.1 Milestone 1: Full Music Code decoder simulation

The goal of this milestone is to complete the entire Music Code decoder and verify its function through simulation. Essentially, you will implement the block labeled as `musiccode_top` in Figure B.1 in this milestone. Except for the `clk` and `clr` signals, `musiccode_top` simply takes audio samples as input (`ain`) and produces a serial signal (`sout`) for UART transmission. To complete this milestone, you need to demonstrate through simulation, that your `musiccode_top` design can take in the audio sample from a testbench, decode the signal into the corresponding ASCII characters, and produce the necessary serial output from the UART.

Since most of the blocks needed to implement `musiccode_top` have already been designed in homework 1 and 2, this milestone basically only requires you to combine your previously designed blocks into a single system. However, there are two additional circuits or modifications you will need for your design to function correct in the final design:

1. Modify `sym_det` from homework 2 or add an additional module to the system such that it will read the values from the ADC correctly;
2. Add an optional FIFO and the associate small data popper (`dpop` to connect the output of the FIFO to the input of UART.

Specifically, you will need to perform one of the followings:

B.1.1 Top level module To design and simulate the top-level module `musiccode_top`, you need to create a new project that includes all the modules that you will use. A few notes about this integration:

- You need to create your own top level VHDL file, `musiccode_top`, that combines all the modules. This top level VHDL entity has a single input port (`ain(11 downto 0)`) and clock (`clk`) and asynchronous clear (`clr`) and produces output the same as the UART (a single bit `sout`).
- The modules `sym_det`, `mucodec` and UART are from Homework 1 and 2. You need to include the design from your own group from homework 1 and 2. You may, and you should, improve the functionality of these modules if they were not fully functional when you turned in homework 1 and 2.
- You can assume your entire system operates on a 96 kHz clock, which will be passed to your system through the clock port (`clk`).

To simulate the behavior of the system, you need to build a new testbench file `tb_musiccode_top`, which reads a testbench file to pass as `ain`. It should also make sure you reset the system by using `clr` signal properly. See below section about FIFO.

B.1.2 Working with ADC input values In this project, you will use an analog-to-digital convertor (ADC) to convert the analog voltage produced by the microphone into digital signals (numbers) for processing on the FPGA. The external ADC is included in a small board called MIC3. Details can be found from the course website.

The sampled data from the ADC are encoded as simple binary numbers from `0000_0000_0000` to `1111_1111_1111`. A study of the MIC3 schematics show that the code `0000_0000_0000` represents the *most negative* air pressure while `1111_1111_1111` represents the *most positive* pressure. When there is no sound (0 air pressure, the reading is *around* `0111_1111_1111`. It is just an estimate as it depends on actual calibration of the circuit.

In other words, the data values you are receiving from the ADC are *NOT* represented by the usual 2s complement representation. You will need to subtract `1000_0000_0000` from the input to obtain the 2s complement representation of the data values.

B.1.3 Detecting Sound Source In order to detect the frequency of the symbol, you have to first detect if there is any sound in first place. In theory, when there is no sound, your ADC will output a solid value of 0. However, in practice, that is not the case. Since there are noises in the real world and in the electronic circuits, you almost never get a perfect zero (0) value even in absolute silent environment. Instead, you almost always get numbers that hover *around* zero during silent time, e.g., $-1, 0, 1, -1, -1, 1, 0, 1, -1, \ldots$. As you can see, using a naive zero crossing design like in homework 2, the above noise will inadvertently produce some additional false zero crossing.

So in the final project, you should improve your design from homework 2 if needed to ensure it can work robustly in real world environment. A simple scheme to detect sound is to perform a moving average of the input value to filter out the noise. Since amplitude of a wave represents the level of sound, your design may choose to count only when the *average* input sound amplitude has passed certain threshold.

B.1.4 Data popper In `musiccode_top`, there is an optional FIFO between the output of `mucodec` and the UART. It is optional because if you have only the basic Music Code protocol with a fixed 16 symbols per second rate, then your UART should be able to send out the data in time. However, if you increase the transmission rate as part of your extension task, then you will need a FIFO to buffer the data before being sent to computer.

If you decide to use a FIFO in your design (highly recommended), then note that the data that you have pushed into the FIFO will simply reside inside the FIFO until they are externally popped. At the same time, the UART would passively wait for data to be transmitted. As a result, you need to implement a simple circuit that connects the two. In this project, we call it a "data popper" (`dpop`). The design of `dpop` is quite straightforward:

`dpop` should monitor the control signals (e.g. `empty` from the FIFO and `ready` from UART), and when both of them are asserted (FIFO is not empty and UART is ready), then it should pop a data from the FIFO and send the data to the UART.

Implement `dpop` and use it in `musiccode_top`. To help with your debugging effort, you probably want to design and test it separately before using it in the over system.

B.1.5 FIFO If you use the built-in FIFO from the Vivado tools, then you must do the following in order to simulate the operation of the FIFO:

- Make sure you have the following library included (uncomment them if you produced your top level VHD file from Vivado) in your top level module `musiccode_top`:

```
library UNISIM;
use UNISIM.VComponents.all;
```

- Instantiate the FIFO in your design. HINT: You can copy-and-paste the port definition from the produced VHDL (`fifo_generator_0`) for your convenience.
- IMPORTANT: Make sure you RESET the system in the beginning of your simulation by setting your `clr` signal high for a short time before turning it back to '0'. The Xilinx produced FIFO **MUST** be initially reset at least once for it to behave correctly in simulation.

If you designed your own FIFO, then you may ignore the part about the use of `UNISIM` library.

B.1.6 Simulating Whole System With all the pieces, you should now be able to simulate the function of the entire system in Vivado. Remember to change the simulation duration to a larger value such that you can simulate the operation of the system for the complete audio input.

B.1.7 Milestone 1 Submission Show the simulation waveform using the provided audio files as inputs. Make sure you highlight (i) the output from the module `mucodec` with correctly decoded ASCII characters, and (ii) the serial output from the UART when the corresponding character is sent to the computer.

B.2 Milestone 2: Implementation on FPGA

In this part of the project, your task is to implement your design from Milestone 1 on the actual FPGA board. For your reference, the board you will be using is the Digilent Basys 3 board. The main FPGA on your board is a Xilinx Artix-7 FPGA (XC7A35T- 1CPG236C). For details of the board, please refer to the Basys 3 FPGA Board Reference Manual. (Note: on Digilent website the link is labeled as Datasheet.)

A sample project, called `musiccode_sys` is provided to you with all necessary additional circuits and configurations to help you implement the design on the FPGA board.

B.2.1 Clock An important part of a real project, as opposed to a simulation project, is that you need to run with an actual clock signal. On the Basys 3 board, a 100 MHz external clock source is available, and you will use that to drive your main system at 96 kHz.

In the provided system, the 96 kHz clock is labeled as `clk_96k` to avoid confusion. This slower clock is generated from the system 100 MHz clock precisely by using the FPGA's on-board multi-mode clock management (MMCM) circuit.

The circuits required to produce these clock signals are already provided to you. Feel free to explore how they work but DO NOT modify any of these circuits.

B.2.2 Audio Input On the actual FPGA board, an external microphone is used to listen to your real world audio input. The audio module you will be using is the Digilent Pmod MIC3 modules, which contains a Knowles Acoustics SPA2410LR5H-B microphone and a Texas Instruments ADCS7476 analog-to-digital converter.

A basic controller (`adccntlr` is provided to you. Its job is to communicate with the ADC to command the ADC to produce an audio *sample*. For this project, the controller sample the audio signal at a fixed rate of 96 kHz with $12 - bit$ accuracy. In other words, it produces a 12-bit sample every 1/96000 seconds. These audio data sample will then become the digital audio samples that can be processed by your system. Recall that your system is also running at 96 kHz by design. Therefore, it means the module is producing exactly 1 sample every cycle. It is designed specifically this way to make your design simple.

B.2.3 Implementing Design & Configuring FPGA You should refer to lab6 on how to implement and download a bitstream to the FPGA.

B.2.4 Serial Terminal In order to receive data from your Basys 3 board, you need to run a serial terminal program on your host computer. Make sure you set up your serial terminal with 9600 baudrate and 8-N-1 configuration.

The default configuration for the Basys 3 board will send a text string to the host computer when it first starts up through its UART. You should use this as a test of your serial connection before you test your own design.

If you set up is correct, when the Basys 3 starts up you should see a text string show on your computer screen.

Congratulation! If you can see the correctly decoded message on your computer screen, your Music Code decoder is completed!

B.3 Milestone 3: Extensions

Once you have completed Milestone 2, your group can discuss on an extension to be implemented as part of Milestone 3. Regardless of the topic of extension that your group has decided, you need to:

- Define your extension;
- Simulate your new design;
- Test your new design;
- Explain your design and results in the final report.

When you make change to the project for the extension, you may choose to do one or more of the following:

- Change the base Music Code protocol;
- Change your design from homework 1 and 2;
- Change the clock frequency on the FPGA board.

Note: even if you cannot complete implementing and testing your extension, you should attempt to adopt your design for the extension. In your project report, you should describe your extended design and state clearly at what stage of development has accomplished (conceptualized only, new design, simulated, tested on board, etc).

Below is a list of topics you may choose from:

1. Improve Speed of transmission:
   (a) Improve transmission symbol per second, i.e, faster, but still fixed symbol per second
   (b) Allow variable transmission speed, i.e. no fixed symbol per second
2. Improve the robustness of transmission:
   (a) Allow one actual musical instrument (e.g. piano, violin, flute, etc) instead of pure sine wave
   (b) Allow ANY musical instrument
   (c) Allow significant background noise
3. Expand the code book:
   (a) Allow Music Code to decode number + alphabet

You may implement other extensions as well, but you will need to seek approval from the teaching team beforehand. Please post a private message on Piazza with your ideas.

B.4 Project Report Submission & Demo

As a group, turn a report describing your design. In your report, you should include *at least* the following items:

- Your group number, and the *name* and *university number* for ALL members of your group.
- Explain the role of each member in the project.
- Explain the level of completeness for your project, e.g., which Milestone has your reached.
- Include the plots and results from Milestone 1 above.
- Explain the extension that you planned to perform in Milestone 3. What are the ideas? What changes have you made or planned to make to the design and/or the protocol? What is the final status of implementation?

Then during project demo, be prepared to show a working design. Before you attend the interview, set up the board and load the correct bitstream to the board for demo.