RECOMMENDED ROUTINE BEFORE TO START WORKING:

1. **CREATE A FOLDER ON THE REMOTE DESKTOP** (Ex: MARCO-ROSSI)
2. **DOWNLOAD THE ASSIGNMENT (.ZIP) FROM *PORTALE DELLA DIDATTICA* AND PUT IT INSIDE YOUR FOLDER**
3. **EXTRACT THE FILES IN THE ZIPPED ASSIGNMENT INSIDE YOUR FOLDER**
4. **CREATE A FOLDER CALLED *Workspace* INSIDE YOUR FOLDER** (Ex: MARCO-ROSSI/Workspace)
   - Now your folder should contain the exctracted assignment files, the .zip, and an empty *Workspace* folder
5. **TYPE *CODE COMPOSER STUDIO* IN THE SEARCH BAR AND OPEN CCS**
6. **WHEN ASKED WHICH WORKSPACE TO USE, SELECT THE EMPTY *Workspace* FOLDER**
   - If CCS is not asking you to select a workspace, let CCS open and then click *File > Switch Workspace* and choose your empty *Workspace* folder
7. **FOLLOW THE GUIDE IN THE SLIDES TO IMPORT THE PROJECTS INSIDE YOUR EMPTY *Workspace*, MAKING SURE YOU TOGGLED THE *COPY PROJECTS INTO WORKSPACE* OPTION**
   - You should now have a copy of the assignment project inside your *Workspace*
8. **YOU SHOULD WRITE THE CODE IN THE FILES IN THE PROJECT INSIDE YOUR *Workspace*, AND CONTINUE WORKING ON THEM FOR THE REST OF THE ASSIGNMENT**
9. **YOU CAN LEAVE THE CODE IN YOUR LOCAL FOLDER ON THE REMOTE DESKTOP**
10. **PLEASE DO BACKUPS AT THE END OF EACH REMOTE LAB CLASS, WE ARE NOT RESPOSIBLE FOR DATA LOSS**

HOW TO PERFORM THE REQUESTED ACTIVITIES:

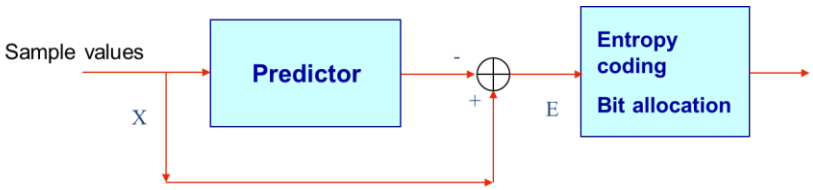
1. **DSPs WILL BE ALREADY BE PLUGGED IN THE REMOTE MACHINES SO THAT YOU CAN CONNECT AND PERFORM ACTIVITIES**
2. **CAREFULLY READ THE *04-DSP-Migliorati-CCS-quick-guide-2020* SLIDES BEFORE PROCEEDING**
   - Everything you'll be asked to do it's in there
3. **CAREFULLY READ THE ASSIGNMENT INSTRUCTIONS**
4. **REMEMBER THE CODE SNIPPETS IN *01-DSP-Migliorati-Theory-01-2020* AND *02-DSP-Migliorati-Theory-02-2020***
5. **IT'S RECOMMENDED TO WRITE DOWN NUMBERS AND RESULTS AS YOU PRODUCE THEM**
   - You can basically write the lab report while doing the experiments
6. **YOU SHOULD OBSERVE TRENDS AND BEHAVIORS WHILE CHANGING SOME PARAMETERS**
   - Ex: "the considered filtering technique is better than the other because as PARAMETER X increases PARAMETER Y decreases much faster…", or "as we change PARAMETER Z, PARAMETER W behaviour does not vary much", etc.
7. **IT'S IMPORTANT TO TRY AND COMPLETE ALL THE LAB EXPERIENCES, IF YOU GET STUCK WITH SOMETHING YOU CANNOT FIX GO TO THE FOLLOWING INSTRUCTION**

HOW TO WRITE THE LAB REPORT:

1. **WRITE ONE SINGLE LAB REPORT DESCRIBING BOTH OF THE ASSIGNMENTS** (Filtering and Compression)
2. **YOU SHOULD CLEARLY REPORT THE GROUP COMPONENTS** (Name, Surname, Number, and Group Number) **AT THE BEGINNING OF THE REPORT**
3. **LAB REPORTS SHOULD NOT BE LONGER THAN 10 PAGES: NO THEORY, NO CODE**
4. **LAB REPORTS SHOULD BE IN .PDF FORMAT**
5. **DISPLAY NUMBERS AND RESULTS IN A CLEAR, SELF-CONTAINED WAY** (Ex: tables, graphs, etc)
6. **OTHER THAN SHOWING RESULTS, YOU SHOULD BRIEFLY COMMENT ON YOUR FINDINGS**
7. **MAKE SURE ALL TABLES, PICTURES, ETC HAVE THEIR OWN CAPTION**
   - An unaware reader should be able to roughly understand what you're talking about and what they're observing in a table or a picture
8. **YOU CAN USE ANY WRITING SOFTWARE YOU WANT (WORD, LATEX, OPEN OFFICE)**
   - But please stick with a "standard" font (Times New Roman, CMU, etc)
9. **SEND THE LAB REPORTS AT LEAST 7 DAYS BEFORE THE EXAM**
   - Lab reports are the starting point for the oral exam
10. **YOU SHOULD BE ABLE TO WRITE IN DECENT ENGLISH**
    - Use writing tools such as https://app.grammarly.com/ which also has a Microsoft Word extension

Lossless Audio Compression LAB

The goal of this lab is to implement and test an audio compression/decompression algorithm on the DSP (**SEE THE SLIDES**). You will implement a lossless audio codec based on linear prediction, and entropy coding of the prediction error. The scheme of the encoder is shown in the following Figure:



Predictor

Two versions of the predictor should be considered:

1. In the first version, the predictor should be implemented as an FIR filter computing a polynomial prediction of the input signal. The following predictors, up to order 2, should be considered:
   - $p_0[n] = 0$ (i.e. no predictor has to be implemented for this configuration, the input signal is directly entropy coded)
   - $p_1[n] = x[n-1]$
   - $p_2[n] = 2x[n-1] - x[n-2]$

   Note that these filters have integer coefficients (not fixed-point), so no scaling is required after computation. As suggested in the sample code, you can implement separate specialized functions for each predictor order.

2. In the second version, the predictor should be implemented as a generic FIR filter taking as input parameter the filter coefficients, represented in Q3.12 format (hence fixed-point, so scaling is required).

The input signal x[n] will be a 16-bit signed integer. The prediction error e[n] = x[n] – p[n] can be implemented as a 16-bit signed integer. For simplicity, we will assume that the audio signal x[n] is sufficiently "smooth" and we have no overflow if we represent e[n] on 16 bits. However, care must be taken to avoid overflow on intermediate computations. For instance, the implementation should guarantee that the value of p[n] is within the range of a 16-bit signed integer, so it should clip it accordingly when needed: if p[n] > 32767 then p[n] = 32767, and if p[n] < -32768 then p[n] = -32768.

Entropy coding

The entropy encoder should use the following encoding: *Sign bit | Rice-Golomb code of the magnitude* (**SEE THE SLIDES**). For the sign bit, use the convention 0 = positive, 1 = negative. The parameter k of the Rice-Golomb code should be configurable between 0 and 15. Examples for k = 2 (vertical bars just divide sign, unary part, remainder part):

e[n] = -1 => code = 1|1|01 e[n] = 5 => code = 0|01|01 e[n] = -12 => code = 1|0001|00

The output of the encoder should be a continuous bitstream in which successive codewords are concatenated. This bitstream should be represented as a sequence of unsigned 16-bit integers, each holding a 16-bit portion of the bitstream (see the slides on bit packing algorithm).

Lossless Audio Codec

The audio codec will take as input a buffer of N audio samples, encoded as 16-bit signed integers, and provide as output an array of 16-bit unsigned integers containing the encoded bitstream. This can be implemented by writing two separate functions.

The first function computes the prediction error for each of the N input audio samples. This function should also manage an internal buffer of the same size as the predictor order. At function call, this buffer should contain the last L samples of the previous block of N samples, where L is the order of the predictor, and before returning the function should fill it with the last samples of the current block (exactly as the circular FIR filter implementation from the previous lab). When encoding the first audio block, these L samples should be initialized to a reference value (e.g., zeroes).

The second function encodes a buffer of N prediction errors using the specified entropy encoder, producing a variable-length bitstream. For simplicity, the output of the entropy coder can be written in a pre-allocated buffer of at least 8*N 16-bit words. The function should also return the length in bits of the encoded bitstream.

The predictor order L and filter coefficients will be fixed (decided at compilation time). The Rice-Golomb parameter k will be decided according to two versions of the codec:

- Fixed (decided at compilation time)
- Variable, computed block by block as $k = \left\lceil \log_2 \left(0.7 * \left(\frac{1}{N} \sum_{n=0}^{N-1} abs(e[n]) + 1 \right) \right) \right\rceil$

(adapted to the distribution of the prediction error e[n]: the sample code includes a function that estimates the parameter k from a buffer of prediction errors)

Task 1: testing the audio codec

Reference CCS project: ***audio_shorten***

In this task, you will implement prediction and entropy coding functions and test them using a reference dataset and a reference implementation of the decoder. The reference CCS project contains an implementation of the entropy decoder and of the reconstruction filters from prediction errors. The project also provides a test signal (8192 audio samples, 16 bit) in the header file "samples2.h" and coefficients for a 5th-order predictor in Q3.12 format in the header file "p_filter.h". Interfaces (function declarations) for the functions to be implemented are provided as well.

INSTRUCTIONS:

1. Import the reference project into the workspace (**SEE STEP-BY-STEP GUIDE AT THE BEGINNING**). Build the project and run it on DSP to verify that everything works.
2. Implement functions that compute the prediction error according to different polynomial predictors and/or generic prediction filters (**Hint**: *take a look at the already provided reconstruction functions: you should do the 'inverse' operation*).
3. Verify that the functions are correct by reconstructing the original signal using the provided reconstruction functions (**Hint:** *if the prediction functions you wrote are correct, since we are dealing with lossless coding, the reconstructed signal should be identical to the input, i.e. err = 0*)

4. Implement the entropy coder function according to the specifications in the **Entropy coding** section; the PACK_BITS macro is already provided, you should use it to write bit values in the encoded bitstream (**Hint**: *code blueprint for this part is already given*)
5. Verify that the function is correct by decoding its output using the provided decoding function (**Hint:** *same as the hint at 3.*)
6. Measure the number of output bits (**Hint**: *they are returned by the entropy coding function*) when encoding the references 8192 samples for different choices of the predictor (no predictor, polynomial 1-2, given 5^{th}-order pred.), and for different values of k for the Rice-Golomb code (use k in [5,7,9]). Report them on a table.
7. Measure the number of **CPU clock cycles per encoded sample** for the different versions, using no optimization and optimization level 2 and report them (**Hint**: to obtain *#CPUcycles / sample*, you should measure the CPU clock cycles for the total 8192, and then divide by …);

Task 2: real-time audio coding

Reference CCS project: ***audio_loop***

In this task, you will test the audio codec on a real-time audio loop. The reference CCS project provides an audio loop implemented with a double buffering technique. Samples are transferred from the ADC of the DSP board to one of two input buffers using DMA. While one input buffer is filling, the DSP can process the other one. Then, the content of two output buffers is alternatively transferred to the DAC using DMA as well. The current implementation simply copies each input buffer to the corresponding output buffer. Double buffering is implemented for both the left and the right channels. The aim of this task is to encode the input buffers using the functions developed and tested in task 1, for different choices of coding parameters, and estimate the average bit rate of the encoder. For using the code developed in task 2, the students can copy the corresponding files (*.c and *.h) to the project folder (use "Project\Add files…" command).

The size of the buffers can be set by changing the value of XMIT_BUFF_SIZE in ref_data_bypass.h. The default sampling frequency of ADC and DAC is 24 kHz. To change sampling frequency, you can edit the constants FS_48K, FS_24K, FS_16K, FS_12K in the file aic_test_i2c.asm.

```
FS_48K          .set    1       ; FS_48K = 1 to use 48 KHz sampling frequency
FS_24K          .set    0       ; FS_24K = 1 to use 24 KHz sampling frequency
FS_16K          .set    0       ; FS_16K = 1 to use 16 KHz sampling frequency
FS_12K          .set    0       ; FS_12K = 1 to use 12 KHz sampling frequency
```

(Set only one of the constants to 1 and the other ones to 0)

The project also uses the DSP real-time clock (RTC) for generating a periodic interrupt. This can be used to perform periodic tasks. The current code just switches the board led every 10 seconds and estimates the number of samples per second. You can set a watch on the variable samples_per_sec and verify that the number of samples per second corresponds to the sampling frequency (**SEE INSTRUCTIONS ON THE SLIDES ON HOW TO SET A WATCH AND ENABLE CONTINUOUS REFRESH**). The number of seconds between two interrupts can be controlled by changing the value of SEC_COUNT in rtc.h.

The students will be provided with Remote machines with connected DSPs and jack cables. If the Remote Connection has been set correcty, audio will be reproduced on the remote machine and not on the

student's local machine. In such fashion, students will not be able to hear the audio but it will be possible to observe the effects of filtering by analyzing the spectrum of the filtered audio by using the Matlab script "**start_spectrum_analyzer.m**". A sample audio file *.wav* is included in the zip folder. You can play it using Windows media player or any other software. Before executing the spectrum analyzer script, make sure you followed the steps in the .txt file **Sound Card Support** you can find the assignment zipped file.

Instructions:

1. Import the reference project into the workspace. Build the project and run it on DSP to verify that everything works.
2. Use the encoding functions developed in **Task 1** for estimating the bit rate of the encoder. You can sum the number of encoded bits for each buffer on a temporary variable and use the RTC interrupt value for periodically estimating the number of bits per second produced by the encoder. Remember that the state of the predictor should be shared between buffer 1 and buffer 2; again, the left and the right channel should be using separate state buffers.
3. Verify whether the encoder is functioning in real-time or not when in stereo mode by looking at the bit rate of the encoder for the following sampling frequencies (12 kHz, 16 kHz, 24 kHz, 48 kHz) and optimization levels (Disabled, 1, 2, 3). Write the estimated bit rates for the different configurations in a table. Which optimization and sampling frequency are required for real-time operations?
4. Choose the highest sampling frequency (48kHz) and measure the average bit-rate of the encoder for the different choices of predictors and k in (5,7,9). For consistency, estimate the average bit-rate over a 30 second time interval, trying to use the same starting point in each experiment (for example, time 1:00 in the provided *.wav* file). Record the values in a table and comment on your results. When discussing your results, compare the obtained bit-rate with the bit-rate of uncompressed audio: what is the best compression rate (i.e. the lowest) when working in real-time? Report on your findings.
5. Repeat the experiment 4 with a variable k. For computing the optimal k for each input buffer, you can use the function "rice_parameter" in rice_codec.c; comment your results.
6. Use the encoder with variable k and choose a predictor. Run the codec on an audio source of your choice (e.g., YouTube). Try to choose an audio source with some variability (both quiet parts and more energetic ones). Estimate the average bit-rate every 3 seconds. You can set a watch on a variable recording the bit-rate and enable continuous refresh to see bit-rate evolution in real-time (see instructions on slides). Observe the correlation between the kind of audio source and the obtained bit rate. Also, try to raise and/or decrease the volume and observe the effect on bit rate. Comment on your findings also considering how bit rate changes in correspondence with different parts (soft, loud) of the song.