Assignment #2
Due March 14, 2016

Make note of the following:

- Each team of two will turn in documentation for the assigned problem(s), that is C or assembly source code as appropriate
- You may build Python/MATLAB/Mathematica prototypes of any C or assembly functions your write to help in program development

Problems:

For the following problems I will expect demos, but I also want a lab report turned which documents your source code, C, ASM, etc. Also include screen shots from Keil where appropriate.

1. Develop a C calling C function that implements the numerical calculation

\[ C = A - B \]

using the data type int16_t, where

\[ A = [a^2 + (a + 1)^2 + (a + 2)^2 + \cdots + (2a - 1)^2] \]
\[ B = [b^2 + (b + 1)^2 + (b + 2)^2 + \cdots + (2b - 1)^2] \]

a.) The function prototype should be of the form

```
int16_t sum_diff(int16_t a_scalar, int16_t b_scalar);
```

b.) Test your program using \( a = 3 \) and \( b = 2 \) by embedding the function call to `sum_diff()` in a main function. Set breakpoints around the function call to obtain both cycle count and the actual time at Level 0 and Level 3 optimization.

c.) For 20 bonus points: Implement as C calling assembly.

2. Consider the inverse of a 3 \times 3 matrix

\[
A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{bmatrix}
\]

a.) Write a C function having function prototype

```
inv3by3(float32_t* A, float32_t* invAData),
```

for float32_t using the matrix of cofactors method as shown below:
where the cofactors are evaluated as the determinate of the $2 \times 2$ matrices of the form

$$
det = \begin{vmatrix} A & B \\ C & D \end{vmatrix} = AD - BC
$$

and $\text{det}[A]$ is the determinant of the $3 \times 3$ matrix. Verify that your inverse calculation is correct using the test matrix below:

$$
\begin{bmatrix}
1 & 2 & 2 \\\4 & 6 & 10 \\
-3 & 6 & -8
\end{bmatrix}
$$

Provide storage for $A$ using a 1D array, i.e.,

```c
float32_t AData[N*N] = {1.0f,2.0f,2.0f,4.0f,6.0f,10.0f,-3.0f,6.0f,-8.0f};
```

This is known as row-major form and is the way C is used to store 2D arrays in numerical computations. This is also the way CMSIS-DSP stores matrices. The inverse needs to be stored in like fashion. Verify that the solution is correct by (1) comparing it with a Python/MATLAB/Mathematica solution and (2) numerically pre-multiply your solution by $AData$ and see that you get a $3 \times 3$ identity matrix. Use the CMSIS-DSP function

```c
arm_mat_mult_f32(&A, &invA, &AinvA);
```

Look at the J. Yiu text Example 22.6.1, p.732, to see how to use the CMSIS-DSP matrix library functions. **Hint:** you will have to create three matrix instance structures using:

```c
arm_matrix_instance_f32 Amat = {NROWS, NCOLS, AmatData};
```

b.) Profile your function at compiler optimization Level 0 (o0) and Level 3 (o3) using the test matrix
c.) Repeat part (a) except now use CMSIS-DSP for all of your calculations. In particular you will use the function

```c
arm_mat_inverse_f32(const arm_matrix_instance_f32 * pSrc, 
arm_matrix_instance_f32 * pDst)
```

For details on using this function see the example on p. 730–735 of the text (Yiu).
**Note:** If you want to preserve the original values in the data array AData, you will need to make a working copy of A, say AW. The arm_mat_inv function write over the original during the inverse solution. As a check on this approach verify as in part (a) that the product of the two matrices gives the identity matrix (you will need a working copy of A).

d.) Profile CMSIS-DSP solution and compare it with the part (b) results.

3. In this program you will convert the pseudo-code for a square-root algorithm shown below into C code for float32_t input/output variables.

```
Approximate square root with bisection method
INPUT: Argument x, endpoint values a, b, such that a < b
OUTPUT: value which differs from sqrt(x) by less than 1

done = 0
a = 0
b = square root of largest possible argument (e.g. -216).
c = -1
do {
    c_old = c
    c = (a+b)/2
    if (c*c == x) {
        done = 1
    } else if (c*c < x) {
        a = c
    } else {
        b = c
    }
} while (!done) && (c != c_old)
return c
```

a.) Code the above square root algorithm in C. Profile you code using the test values 23, 56.5, and 1023.7. Run tests at compiler optimization o0 and o3. Note: You will need to establish a stopping condition, as the present form is designed for integer math. I suggest modifying the line:

```
if (c*c == x) {
```

**to something like**

```
if (fabs(c*c - x) <= max_error) {
```

where max_error is initially set to $10^{-6}$. Realize that this value directly impacts the execution speed, as a smaller error requirement means more iterations are required. See if you can find the accuracy of the standard library square root.

b.) Compare the performance of your square root function at o3 with the standard math library function for float (float32_t), using float sqrtf(float x).

c.) Compare the performance of your square root function at o3 to the M4 FPU intrinsic function float32_t __sqrtf(float x).

4. **Real-time Gold Code sequence generation using a lookup table (LUT):** Pseudo-random sequences find application in digital communications system. The most common sequences are known as $M$-sequences, where $M$ stands for maximal length. A Gold Code formed by exclusive ORing two M sequences of the same length but of different phases. For example Gold codes of length 1023 are uniquely assigned to the GPS satellites so that the transmis-
sions from the satellites may share the same frequency spectrum, but be separated by the properties of the Gold codes which make nearly **mutually orthogonal**. In this problem you start by building an $M$-sequence generator in C.

a.) The block diagram of a three state linear feedback shift register (LFSR) is shown below:

Following each clock (note the clock input is implicitly assumed to be a part of the shift register) a new output bit is taken from Stage 3. The feedback taps for this $M = 3$ example are located at 2 and 3. On the far right of the figure you see the output pattern has length $2^M - 1$ bits before repeating. Note also that the initial shift register load is $[1, 1, 1]$. If you start the generator in the all zeros state it will fail to produce an output as the $M$ zeros in a row is not found in the output pattern. A pattern of $M$ ones occurs exactly once, which useful in deriving a synch waveform. The taps settings in Table 1 are not unique, but using an arbitrary tap set does not guarantee a maximal length sequence. Also note that the output can be taken from any shift register element. At the $M$ stage is convenient for drawing purposes.

Your task in (a) is to code a generator using a single 16-bit integer, i.e., `uint16_t`, to hold the elements of the shift register. A suggested function prototype is to employ a data structure such as, `Mseq`, as shown below. This makes for an efficient function call.

```c
// gen_PN header: gen_PN.h, implementation in gen_PN.c
// Mark Wickert February 2015
```

<table>
<thead>
<tr>
<th>$M$</th>
<th>Taps</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[0, 1, 1]</td>
</tr>
<tr>
<td>4</td>
<td>[0, 0, 1, 1]</td>
</tr>
<tr>
<td>5</td>
<td>[0, 0, 1, 0, 1]</td>
</tr>
<tr>
<td>6</td>
<td>[0, 0, 0, 1, 1]</td>
</tr>
<tr>
<td>7</td>
<td>[0, 0, 0, 1, 0, 0, 1]</td>
</tr>
<tr>
<td>8</td>
<td>[0, 0, 0, 1, 1, 1, 0, 1]</td>
</tr>
<tr>
<td>9</td>
<td>[0, 0, 0, 0, 1, 0, 0, 1]</td>
</tr>
<tr>
<td>10</td>
<td>[0, 0, 0, 0, 0, 1, 0, 0, 1]</td>
</tr>
</tbody>
</table>
#include <stdint.h>

// Structure to hold Mseq state information and make calling the
// generator function efficient by only requiring the passing of
// the structure address. For this to be implemented an initialization
// function is also required, hence the two function prototypes below.
struct Mseq
{
    uint16_t M;          // Holds SR length; cannot exceed 16
    uint16_t tap1;       // holds the tap1 position from Table 1
    uint16_t tap2;       // holds the tap2 position from Table 1
    uint16_t mask1;      // holds the bit mask for tap1
    uint16_t mask2;      // holds the bit mask for tap2
    uint16_t sync_mask;  // holds the bit mask to detect the M ones condition
    uint16_t SR;         // holds the 16-bit SR
    uint16_t output_bit; // holds the output bit
    uint16_t sync_bit;   // holds the synchronization bit (not a requirement)
};

void gen_PN_init(struct Mseq* PN, uint16_t M, uint16_t tap1, uint16_t tap2,
    uint16_t SR);  // initial SR load, e.g., 0x1, is input here

void gen_PN(struct Mseq* mseq); // pass structure by address use -> to access members

Your task is to implement at the very least the gen_PN() and perhaps also
    gen_PN_init(). An example of usage of the above is:

    // At the global level
    struct Mseq PN1; // note PN <=> pseudo noise sequence

    // In main
    gen_PN_init(&PN1, 5, 3, 5, 0x1);

    // In ISR
    gen_PN(&PN1);
    some_variable = PN1.output_bit;

    To get started consider the following formulation:

Use shift left (<<) to advance and add new LSB

XOR

Effectively taps 3 and 5 to create an M = 5 generator

For some hints on how to build this see the M-sequence generator used on the mbed in
b.) Test the generator by calling from within the SPI2_IRQHandler function of the stm32_loop_intr.c module you used in Lab1. Write the output to a GPIO pin so you can view the waveform on the scope/logic analyzer. You may also wish to fill a buffer so you can export the output to a file or perhaps the PC serial port. Test the generator with $M = 5$ and $M = 10$. Verify the period and search for the pattern of fives ones and 10 ones respectively.

c.) To explore Gold codes you will consider the case of the $M = 10$ (1023 bit patterns) used in GPS for coarse acquisition (CA). Note: Commerical GPS is limited to using only the CA codes. In the Lab 2 ZIP package you will find a text file, calthru37.txt, that contains 37 Gold codes arranged in columns. To get a particular code to run on the Cortex-M a utility that writes header files is available in the IPython notebook for lab two. This notebook also shows you how to read selected columns of a text file using the numpy loadtxt() function. Some of the IPython notebook code is shown below:

```python
import numpy as np
ca = np.loadtxt('caithru37.txt', dtype='int16', unpack=True)
```

Read in the entire set of codes and check the dimensions and the data type

```
def CA_code_header(fname_out, Nca):
    # Write 1023 bit CA (Gold) Code Header Files
    Mark Wickert February 2015
    ca = np.loadtxt('calthru37.txt', dtype='int16', usecols=(Nca-1), unpack=True)
    M = 1023  # code period
    N = 23    # code bits per line
    Sca = 'ca' + str(Nca)
    f = open(fname_out, 'wt')
    f.write('//define a CA code\n\n')
    f.write('#include <stdint.h>\n\n')
    f.write('#ifndef N_CA\n\n')
    f.write('#define N_CA %d\n' % M)
    f.write('#endif\n\n')
    f.write('#define S_CA %s\n\n' % Sca)
    f.write('#ifdef N_CA\n\n')
    f.write('#endif\n\n')
```

ECE 4670 Lab 2 at:
http://www.eas.uccs.edu/wickert/ece4670/lecture_notes/PN_seq.cpp
The last two lines of code are in a separate cell from the function code above it. These lines write header files containing the CA codes 1 and 12 respectively. The header file can then be imported into your Keil project and utilized just like the wave-table signal generator from Lab 1. Your task is to implement different CA codes (your choice) on each of the audio codec outputs.

Before jumping in consider what makes the Gold codes special. A fundamental property of both \(M\)-sequences and Gold codes is that they exhibit a strong correlation peak once per code period. For discrete-time signals the cross-correlation takes the form

\[
R_{ij}[k] = \frac{1}{N} \sum_{n=0}^{N-1} x_i[n]x_j[n+k]
\]

where \(N\) is the data record length used in the calculation (actually estimation).

Since the Gold codes form a family of codes, taking any pair codes \(i \neq j\) with result in only a small cross-correlation value. This means that \(i \neq j\) codes are nearly orthogonal and as signals can lie on top of each other cause minimal interference when a receiver uses the code of interest to recover via cross-correlation the information riding the transmitted signal. Here we use the function \(\text{Rij}, \text{lags_axis} = \text{dc.xcorr(xi,xj,lag_value_range)}\) to calculate the auto and cross-correlation between CA codes 1 and 2. The code module \text{digitalcom.py} contains the needed function.

Below is a sample calculation from the IPython notebook for Lab 2. Note that signaling is typically done using a bipolar waveform, that is the 0/1 values of the code are con-
Your task is to send code values for some code $i$ and $j$, $i \neq j$ to the codec left and right codec channels. Additional signal processing is required: (1) convert the 0/1 code values to ±10000 and implement a pulse shaping scheme using a raised cosine (RC) pulse shape. This will give you a chance to again use the CMSIS-DSP library. The system block diagram is the following:

```
import digitalcom as dc

R11, lags = dc.xcorr(Z*camat[0,:,:]-l, Z*camat[1,:,:]-l, 100)
R12, lags = dc.xcorr(Z*camat[0,:,:]-l, Z*camat[1,:,:]-l, 100)

plot(lags, R11)
plot(lags, R12)
xlabel(r'Lag $k$ in auto/cross-correlation')
ylabel(r'Normalized Correlation Amplitude')
title(r'CA Code Correlation Properties Over One Period')
legend((r'Autocorr of CA1', r'Crosscorr of CA1/2'), loc='best',)
grid();
```

![CA Code Correlation Properties Over One Period](image)

One sample per bit and over exactly one code period (1023 samples)

The pulse shaping operation is jumping ahead to give you a taste of FIR filtering and impulse train modulation from digital communications applications. An upsampling factor of four is employed, which means on every fourth pass through the I2S_HANDLER func-
tion you will draw a CA code value from the code arrays scaled to $\pm 10000$, modulo 1023. On the three remaining passes you insert 0 (zero). The values are passed into a linear filter as follows:

```c
#include "src_shape.h" // bring in filter coefficients h_FIR and #define M_FIR
...
float32_t x1, y1, state1[M_FIR]; // Working variables for channel 1
arm_fir_instance_f32 S1;
float32_t x2, y2, state2[M_FIR]; // Working variables for channel 2
arm_fir_instance_f32 S2;
...
// In Main insert
arm_fir_init_f32(&S1, M_FIR, h_FIR, state1, 1); // 1 => process one sample only
arm_fir_init_f32(&S2, M_FIR, h_FIR, state2, 1);
stm32_wm5102_init(FS_48000_HZ, WM5102_LINE_IN, IO_METHOD_INTR); // fs = 48 kHz
...
// In the ISR, for each channel, 1 and 2 (1 shown below)
// left output sample is +/- 10000*codebit or 0 based on a modulo 4 index counter
// The codebit is drawn modulo 1023 from the CA code array
x = (float32_t)left_out_sample;
arm_fir_f32(&S, &x, &y, 1);
left_out_sample = (short)(y);
...
```

The bit rate will be $48/4 = 12$ kbps (CA code chips per second). The header file `SRC_shape.h` is supplied in the ZIP. The details of how to create it is included in the IPython notebook for Lab 2.

d.) Now you are ready for testing via waveform data collection and auto- and cross-correlation calculations in Python. Along the way also view the left or right output channels on the spectrum analyzer (Agilent 4395A in the lab or using the Analog discovery). Verify that the main lobe of the spectrum extends from 0 Hz to about 8.1 KHz ($(12 \times 1.35)/2$).

Capture a long record or the left and right channels of at $f_s = 40$ kHz or higher and import into IPython.

![Agilent 4395A vector network with active probe input to port R.](image)
e.) Compute the auto-correlation and cross-correlation as shown earlier, except now you have multiple samples per bit in the waveform itself. The autocorrelation plot should have peaks spaced by the period of the code, which is $1023 \times \frac{1}{12\text{kHz}} = 85.25\text{ms}$. The cross-correlation should have no distinctive correlation peaks. The best way to collect a long data record (in stereo) is using the sound card on the PC as shown below.

![Waveform capture using the PC sound card.](image)

f.) Use the eyeplot tool in digitalcom.py (`dc.eyeplot()` to plot an eye plot of one of the two waveforms. For an example on the use of `dc.eyeplot()` see the Lab 2 IPython notebook. For this to work nicely you need to have an integer number of samples per bit. The best way to get this is to export a buffer samples from Keil to a text file or log the serial terminal, and then import them into Python. You cannot write to the serial port in real-time at 48 kspi due the overhead of using `sprintf()`. In any case you need to first fill a buffer of samples, say 1000 or more.

I know there are a lot of parts to Problem 4. We will talk in class on Monday.
Capturing Left and Right Channel CA Code Waveforms for Problem 4 Parts d, e, and f

There are two reasonable options within reach for capturing left and right audio samples from the Wolfson codec.

Option 1: PC Audio System using Goldwave or Soundcard Oscilloscope

The preferred method is to capture stereo samples using the PC sound system. Unfortunately in recent times both desktop and notebook PC, Mac books included, have been eliminating the line input jack. For our purposes the mic input is often too sensitive and easily subject to overload (clipping, etc) without some additional considerations.

The objective is to keep the record level in GoldWave in the green just below the yellow the yellow part of the display shown below:

Before getting to this point you need to the control panel to configure *Hardware and Sound*:

[Images of control panel screens showing device controls and sound settings]

I have an iMic USB audio system connected here
When you open the panel for the mic device that has an actual 3.5mm connector (front and/or back panel of the PC), you will see more options to configure:

![Microphone Properties Panel](image1.png)

Under the *Advanced* tab make sure to choose **2 channel**, 16 bit, 48000 Hz or higher quality. It is very important that you choose 2 channel, as it seems that the default for mic inputs is frequently 1 channel. With 1 channel the left and right channels will be summed together. A sure sign of this is when you import the wave file into Python and see that both channels are identical!

Next you need click on the *Levels* tab and be prepared to use the slider (see below) to adjust the recording level. With this control you make sure the Gold Wave recording leveling is not clipping. With the iMic system I use there is a switch from Mic to Line, so additional attenuation is available. With a mic only input you will need to set the gain slider to almost zero to avoid clipping. If you cannot get the level low enough an **extreme measure** is to insert a resistive voltage divider circuit as shown below:

![Voltage Divider Circuit](image2.png)

Assuming that you capture at 48 ksp, you will now have approximately the same the sample rate as the signal was generated at.
When using the Soundcard Oscilloscope app you work with the Audio Recorder portion of the app. Once you have a .wav file move on to post processing the file. In an Python notebook you now import the .wav file generate some plots.

```
import digitalcom as dc

fs, CA_wave = ssd.from_wav('cal_ca12_10s.wav')
fs

psd(CA_wave[:,0],2**10,48);  # plot the power spectrum
psd(CA_wave[:,1],2**10,48);  # plot the power spectrum
xlim([0,25]);
ylim([-80,-10]);
xlabel('Frequency (kHz)');

R_shaped_ca_00, lags = dc.xcorr(CA_wave[:,0],CA_wave[:,0],100)
R_shaped_ca_01, lags = dc.xcorr(CA_wave[:,0],CA_wave[:,1],100)
fs_ca = 80000
plot(lags*fsCa/fs_ca,R_shaped_ca_00.real)
plot(lags*fsCa/fs_ca,R_shaped_ca_01.real)
xlabel(r'\text{Lag 3k$\delta$ in auto/cross-correlation (bits)'}
xlabel(r'\text{Normalized Correlation Amplitude}')
title(r'\text{CA Pulse Shaped CA Code Correlation Properties}')
legend((r'Autocor of CA1',r'Crosscor of CA1/12'),loc='best')
grid();

dc.eye_plot(dc.time_delay(CA_wave[:,1],1.25*ones(len(CA_wave[:,1]))),2*4,4)
xlabel(r'\text{Time Index (~4 samples/bit)}')
xlim([0,8]);
```

Note the eye plot will likely no be as clear as what was shown during lecture. By only plotting 500
samples, as shown above, you can minimize the impact of clock drift. I have included a time delay function to try to locate at least one of the ~4 samples per bit at the maximum eye opening.

**Option 2: Analog Discovery**

If you have an Analog Discovery available you can get better results, but the record length you capture is only 8192 points long. To stay within this framework good scope settings are: Time base = 10 ms/div and channels C1 and C2 200 mv/div. Save the scope data as a CSV file. Three columns of data will saved: col 1 = time, col 2 = C1 samples, and col 3 = C2 samples. The sampling rate will be ~80000 samples/second. The data can be loaded into the Python workspace using `loadtxt` as shown below:

```python
#fs,calca12 = ssd.from.wav('calca12_10s.wav')
t_ca,cal_c,cal2_c = loadtxt('calca12_50ms_div.csv',delimiter=',',skiprows=1,usecols=(0,1,2),unpack=True)

1/(t_ca[1]-t_ca[0]) # check the sample rate
len(t_ca)
79999.99999964406

psd(cal_c,2**10,80); # plot the power spectrum
xlim([[0,25]]);
ylim([-40,-10]);
xlabel('r'Frequency (kHz)')

R_shaped_ca_00, lags = dc.xcorr(cal_c,cal1_c,100)
R_shaped_ca_01, lags = dc.xcorr(cal_c,cal2_c,100)
Rb = 12000
fs_c = 80000
plot(lags*Rb/fs_c,R_shaped_ca_00.real)
plot(lags*Rb/fs_c,R_shaped_ca_01.real)
xlabel('r'Lag $k$s in auto/coss-correlation (bits)')
ylabel('r'Normalized Correlation Amplitude')
title('r'RC Pulse Shaped CA Code Correlation Properties')
legend('r'Autocor of CA1',r'Crosscor of CA1/12',loc='best',)
grid();

dc.eye_plot(dc.farrow_resample(cal_c,1.0,1.05),2*7,4)
xlabel('r'Time Index (~7 samples/bit')
xlim([0,13]);
```

Note in the eye plot I am assuming ~7 samples per bit since 80 ksp/s/12 kbps = 6.6667 and using the Farrow resample function with rate increase of 1.05 gives $6.6667 \times 1.05 = 7$. 

Import the CSV saved from the scope app
Check the sample rate
Plot the power spectral density
Generate waveform autocorrelation and cross correlation. Here CA1 and CA12 was used in the real-time code.

Generate an eye plot