Assignment #2

Due Friday February 26, 2018

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 you write to help in program development

Problems:

For the following problems I will expect demos, but I also want a lab report turned in 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 3×3 matrix

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

a.) Write a C function having function prototype

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

for `float32_t` using the *matrix of cofactors* method as shown below:

$$\mathbf{A}^{-1} = \frac{1}{\det[\mathbf{A}]}\begin{bmatrix} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{13} & a_{12} \\ a_{33} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix} \\ \begin{vmatrix} a_{23} & a_{21} \\ a_{33} & a_{31} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{13} & a_{11} \\ a_{23} & a_{21} \end{vmatrix} \\ \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{12} & a_{11} \\ a_{32} & a_{31} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \end{bmatrix}$$

where the cofactors are evaluated as the determinate of the 2×2 matrices of the form

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

and det[**A**] is the determinant of the 3×3 matrix. Verify that your inverse calculation is correct using the test matrix below:

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

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

```
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 3×3 identity matrix. Use the CMSIS-DSP function

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

Look at the J. Yiutext 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:

```
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

```
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).

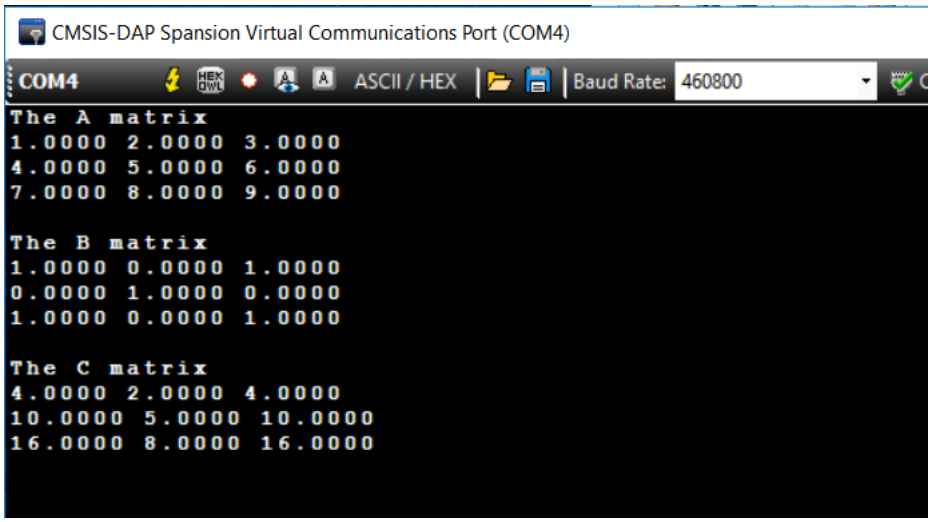
Note: If you want to preserve the original values in the data array `AData`, you will need to make a working copy of `A`, say `AW`. The `arm_mat_inv` function write over the original during the inverse solution. As a check on this approach verify as in part (a) that the product of the two matrices gives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

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

To assist with the display of matrices at the terminal, the sample main module, `main_matrix_demo.c`, is included in the ZIP package. The module `helper.c`/`helper.h` contains the formatting functions. A portion of main_matrix_demo is shown below:

```
int main(void){
arm_status status;
static float32_t AMat[N*N] = {1.0f,2.0f,3.0f,
                              4.0f,5.0f,6.0f,
                              7.0f,8.0f,9.0f};

static float32_t BMat[N*N] = {1.0f,0.0f,1.0f,
                              0.0f,1.0f,0.0f,
                              1.0f,0.0f,1.0f};
static float32_t CMat[N*N] = {0.0f,0.0f,0.0f,
                              0.0f,0.0f,0.0f,
                              0.0f,0.0f,0.0f};

arm_matrix_instance_f32 A = {N,N,AMat};
arm_matrix_instance_f32 B = {N,N,BMat};
arm_matrix_instance_f32 C = {N,N,CMat};
char message[80];

...

//Matrix multiply experiment
// Display A and B matrices
print_mat(AMat,3,3,message);
write_uart0("The A matrix\n");
write_uart0(message);
write_uart0("\n");
print_mat(BMat,3,3,message);
write_uart0("The B matrix\n");
write_uart0(message);
write_uart0("\n");

//Multiply A (AMat) times B (BMat)
status = arm_mat_mult_f32(&A,&B,&C);
print_mat(CMat,3,3,message);
write_uart0("The C matrix\n");
write_uart0(message);
write_uart0("\n");

...
}
```

A screen shot of the corresponding terminal output is given below:



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

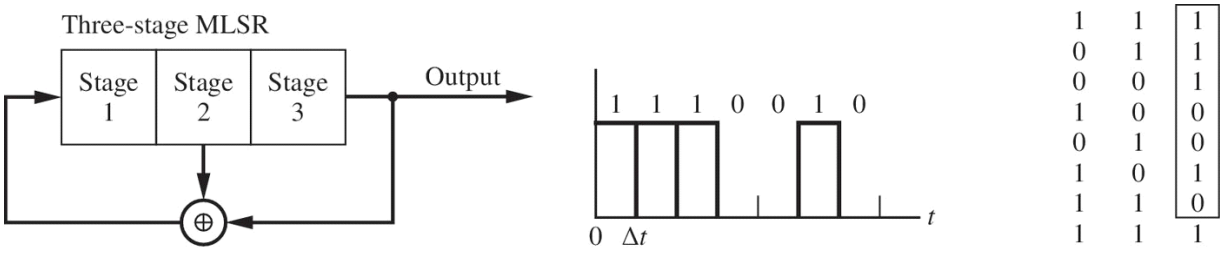
can find the accuracy of the standard library square root.

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 *PN* Sequence Generation with Data Structure**: Pseudo-random sequences find application in digital communications system. The most common sequences are known as *M*-sequences, where *M* stands for maximal length. In GPS for example Gold codes of length 1023 are uniquely assigned to the GPS satellites so that the transmissions from the satellites may share the same frequency spectrum, but be separated by the properties of the Gold codes which make nearly *mutually orthogonal*. Gold codes and M-sequences are related. In this problem the focus is on building an *M*-sequence generator in C using a data structure to manage the states and other details of the generator.

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

Table 1: Taps settings for *M* = 3 to 10

M	Taps
3	[0, 1, 1]
4	[0, 0, 1, 1]
5	[0, 0, 1, 0, 1]
6	[0, 0, 0, 0, 1, 1]
7	[0, 0, 0, 1, 0, 0, 1]
8	[0, 0, 0, 1, 1, 1, 0, 1]
9	[0, 0, 0, 0, 1, 0, 0, 0, 1
10	[0, 0, 0, 0, 0, 0, 1, 0, 0, 1]

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. The function prototype to employ and data structure `Mseq` is shown below. This makes for an efficient function call.

```
// gen_PN header
// Mark Wickert February 2015

#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;
	uint16_t tap1;
	uint16_t tap2;
	uint16_t mask1;
	uint16_t mask2;
	uint16_t sync_mask;
	uint16_t SR;
	uint16_t output_bit;
	uint16_t sync_bit;
};

void gen_PN_init(struct Mseq* PN, uint16_t M, uint16_t tap1, uint16_t tap2, uint16_t
SR);
void gen_PN(struct Mseq* mseq);
```

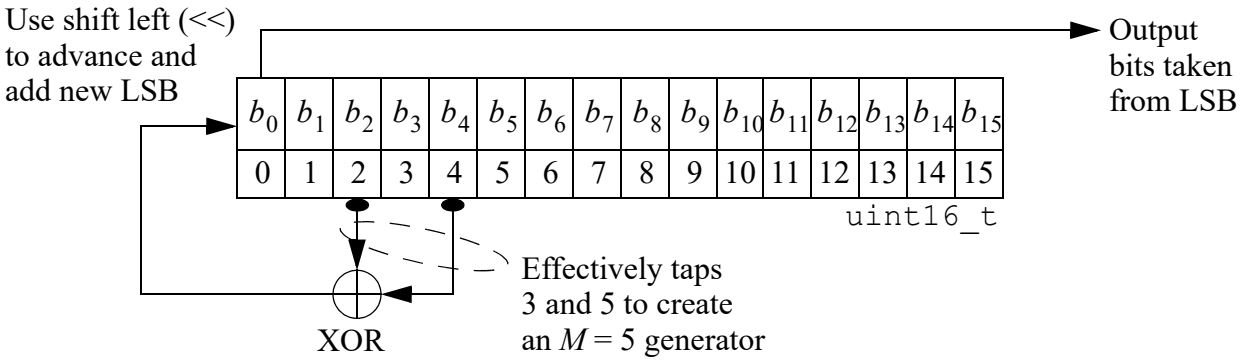
Implement both `gen_PN()` and `gen_PN_init()`. An example of usage of the above is found in the file `fm4_PN_intr_GUI.c` which is in the `src` folder of the ZIP package.

```
#include "gen_PN.h"

#define L_UP 20 % upsampling factor

// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;
struct Mseq PN1;
struct Mseq PN2;
int16_t xLh, xRh;
volatile int16_t idx_up = 0;

void PRGCRC_I2S_IRQHandler(void)
{
  union WM8731_data sample;
  int16_t xL, xR;

  gpio_set(DIAGNOSTIC_PIN,HIGH);
  // Get L/R codec sample
  sample.uint32bit = i2s_rx();
```

```
  //Process
  if (idx_up == 0) // Output a new sample every L_up samples
  {
    gen_PN(&PN1);
    gen_PN(&PN2);

    //write PN code bit to the left_output sample
    // and the PN sync bit to the right output sample
    if ((int16_t) FM4_GUI.P_vals[5] > 0)
    {
      xLh = 20000*(int16_t)PN2.output_bit - 10000;
      xRh = 20000*(int16_t)PN2.sync_bit - 10000;
    }
    else
    {
      xLh = 20000*(int16_t)PN1.output_bit - 10000;
      xRh = 20000*(int16_t)PN1.sync_bit - 10000;
    }
  }

  // Breakout and then process L and R samples with
  // slider parameters for gain control
  xL = (int16_t) (FM4_GUI.P_vals[0] * xLh);
  xR = (int16_t) (FM4_GUI.P_vals[1] * xRh);

  idx_up = (idx_up+1) % L_UP;

  // Return L/R samples to codec via C union
  sample.uint16bit[LEFT] = xL;
  sample.uint16bit[RIGHT] = xR;
  i2s_tx(sample.uint32bit);

  NVIC_ClearPendingIRQ(PRGCRC_I2S_IRQn);

  gpio_set(DIAGNOSTIC_PIN,LOW);
}

int main(void)
{

  // Initialize the slider interface by setting the baud rate (460800 or 921600)
  // and initial float values for each of the 6 slider parameters

  init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0);

  // Send a string to the PC terminal
  write_uart0("Hello FM4 World!\r\n");

  // Set up PN generator
  gen_PN_init(&PN1,5,3,5,0x1); // 5 stage
  gen_PN_init(&PN2,10,7,10,0x1); // 10 stage


  // Some #define options for initializing the audio codec interface:
  // FS_8000_HZ, FS_16000_HZ, FS_24000_HZ, FS_32000_HZ, FS_48000_HZ, FS_96000_HZ
  // IO_METHOD_INTR, IO_METHOD_DMA
  // WM8731_MIC_IN, WM8731_MIC_IN_BOOST, WM8731_LINE_IN
  fm4_wm8731_init (FS_48000_HZ,                // Sampling rate (sps)
                   WM8731_LINE_IN,            // Audio input port
```

```
                IO_METHOD_INTR,             // Audio samples handler
                WM8731_HP_OUT_GAIN_0_DB,    // Output headphone jack Gain (dB)
                WM8731_LINE_IN_GAIN_0_DB); // Line-in input gain (dB)
  while(1){
    // Update slider parameters
    update_slider_parameters(&FM4_GUI);
  }
}
```

To get started consider the following formulation:



For some hints on how to build this see the *M*-sequence generator used on the mbed in ECE 4670 Lab 2 at:

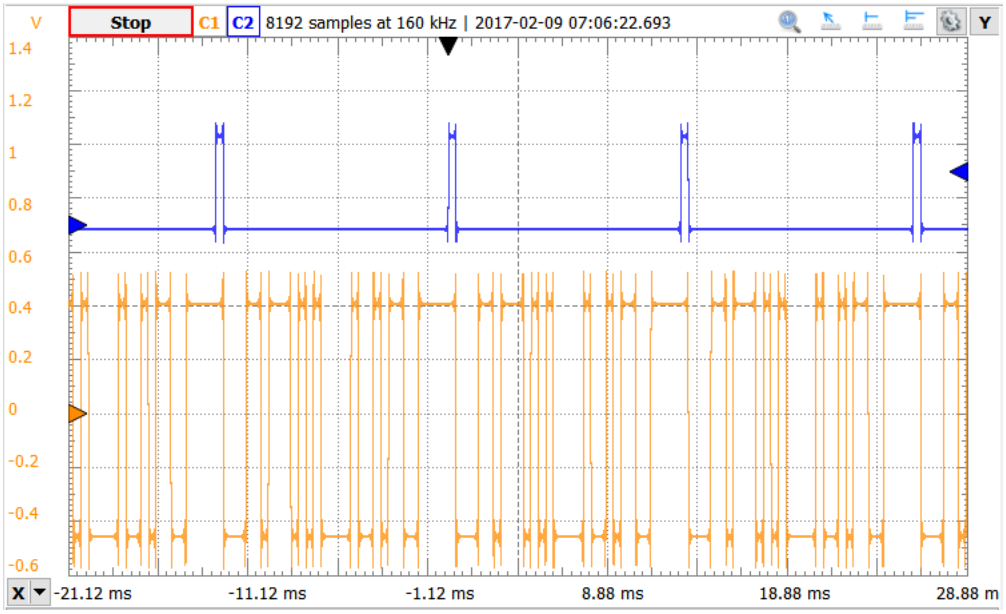
`http://www.eas.uccs.edu/wickert/ece4670/lecture_notes/PN_seq.cpp`

To writing the init function consider the snippet below to help you get started:

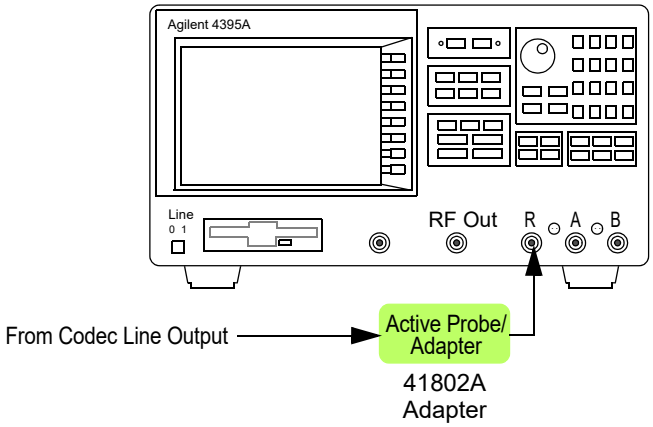
```
void gen_PN_init(struct Mseq* PN, uint16_t M, uint16_t tap1,
                 uint16_t tap2, uint16_t SR)
{
  PN->M = M;
  PN->tap1 = tap1-1;
  PN->tap2 = tap2-1;
  ...
}
```

Note the usage of the arrow operator when dealing with pointers to a data structure. Also consult the Python class found in the Assignment2 Jupyter notebook inside the Python folder of the ZIP package.

b.) Test the generator by calling it from within the `PRGCRC_I2S_IRQHandler` function of the `fm4_PN_intr_GUI.c` module found in the Assignment2 `src` folder. In this file the generator output and sync bit is written to the audio codec left and right channels so you can view the waveform on the scope. 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. The output for the $M = 5$, as captured on the Analog Discovery, is shown below.

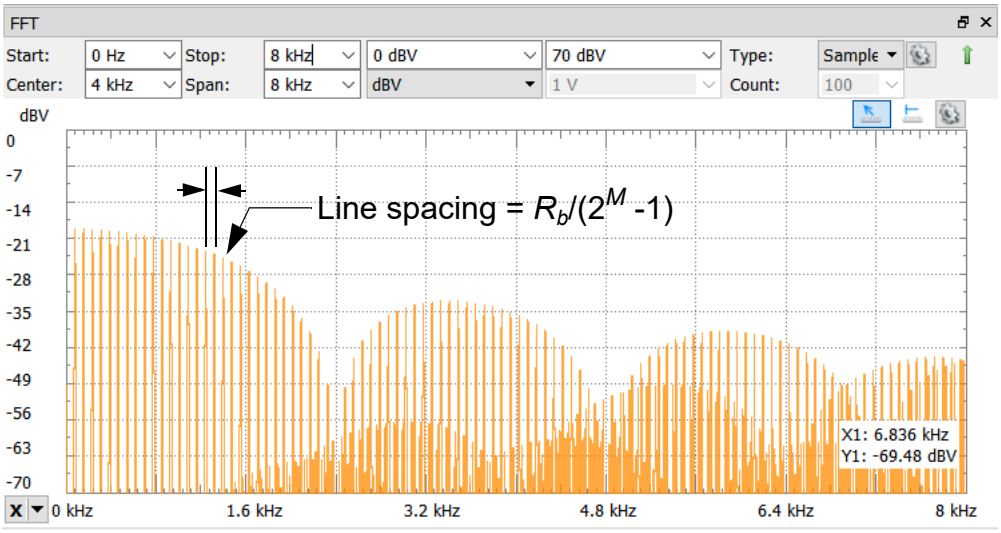


c.) Now you are ready for testing with the spectrum analyzer (Agilent 4395A) in the lab or using the Analog Discovery. Verify that the main lobe of the spectrum extends from 0 Hz

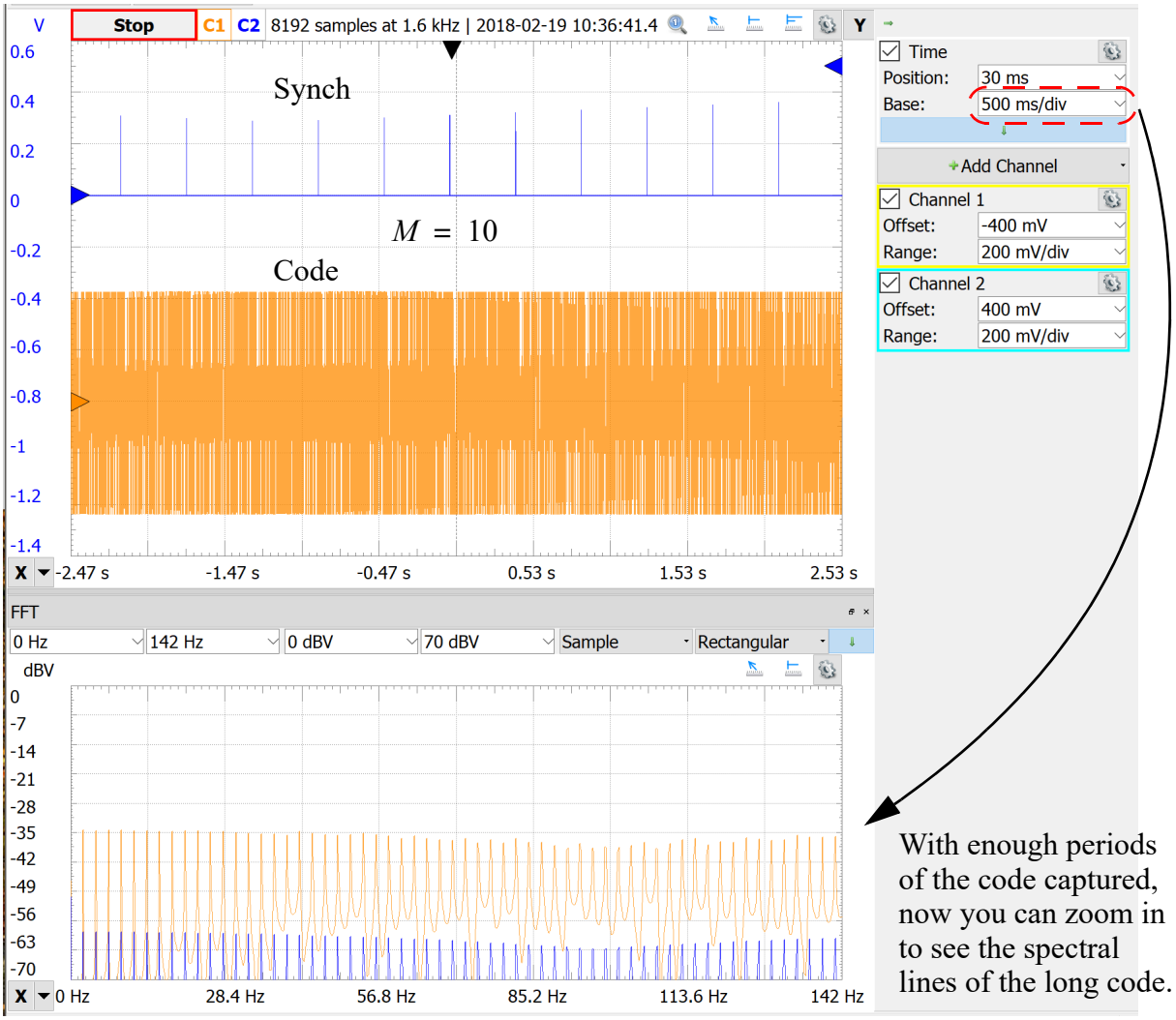


Agilent 4395A vector network with active probe input to port **R**.

to the bit rate $R_b = 48000/L_{\text{up}} = 48000/20 = 2400$ bits/s. This should be true for both $M = 5$ and $M = 10$. Also verify the spectral spacing of the individual spectral lines is $R_b/(2^M - 1)$ Hz for the $M = 5$ generator. If using the Analog Discovery the FFT analyzer that is part of the two channel scope works well. Consider a screen capture or import CSV data into the Jupyter notebook from the Analog Discovery FFT spectrum window (see figure below). To see the spectral lines when $M = 10$ is a bit more involved as much higher spectrum resolution is needed. On the Analog Discovery this means increasing the Time Base (see the second figure below).

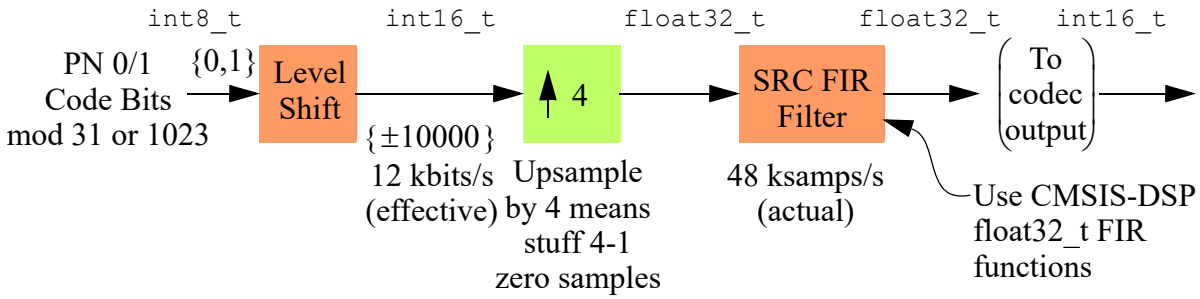


Below we increase the time base and then the FFT size can resolve the spectral lines:



Ask for help in configuring the spectrum analyzer if this is the first time you are using it.

d.) **20 pt Bonus**: Pulse shape the bit sequence using a square-root raised cosine pulse shape FIR filter defined in the header file `src_shape.h`. Additional signal processing is required to 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, this time try out filtering. The system block diagram is the following:



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 `SPI2_IRQHandler` function you will draw a CA code value from the code arrays scaled to ± 10000, modulo 1023. On the three remaining passes you insert 0 (zero). The values are passed into a linear filter as follows:

```
#include "b_RC4.h" // bring in filter coefficients b_RC4 and #define M_FIR
...
// Create two filter instances, one for the PN code and one for the sync pulse
// Start by declaring the working variables globally
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 the following to fill the filter data structures with the needed
// variables for FIR filtering one sample at a time.
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);
...
// Finally, in the ISR, for each channel, 1 and 2 (1 shown below)
// left output sample is +/- 10000.0f*codebit or 0.0f based on a modulo 4 index counter
// The codebit is taken from the PN generator as in part c.
// Repeat for producing the sync pulse out of the right channel.
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 (PN code bits per second). The header file `b_RC4.h/b_SRC.h` is supplied in the ZIP. The details of how to create it is included in the Jupyter notebook for Lab 2.