

ESE 150 – Lab 08: Machine Level Language

LAB 08

In this lab we will gain an understanding of the instruction-level implementation of computation on a microprocessor by:

1. Using ItsyBitsy to perform the Fourier Transform on sampled data in the time domain, converting it to the frequency domain.
2. Timing your Fourier Transform to see how long the operation takes to perform.
3. Calculating number of cycles each instruction takes and calculating the time of execution using the assembly file.

Background:

Let us learn first what an .elf file and a .hex file are.

ELF is an acronym for Executable Linking Format. Files that contain the .elf file format are system files that store executable programs, shared libraries and memory dumps.

A HEX file is a hexadecimal source file typically used by programmable logic devices.

.elf and .hex files are both generated in the process of uploading Arduino code to the device.

In this lab you will be compiling Arduino code, which generates a .elf file. We will then generate assembly code from the .elf file to observe how your code works at the processor instruction level. You will see which instructions get executed in what order and how many times. This will give you an idea of which instructions take longer time and how to reduce your execution time.

In previous labs we've performed DFT for various applications without thinking about the implementation. We have had programs which run for several seconds and sometimes minutes!

Now we will look deeper at how the machine executes your program and how much time that takes.

From your lecture, you know what processor instructions are and how to read them. Let's work with them to understand the execution time of your program.

ESE 150 – Lab 08: Machine Level Language

Prelab: Obtaining the Assembly code

- In this section, we'll compile our Arduino code and obtain the location of a temporary .hex file
- We'll convert the .hex file to assembly code.

Optimizing our code requires that we view the generated machine code (hex) in assembly language. This allows us to know how things work at the instruction level. When we review assembly code we understand how the computer's hardware works and functions on a low-level. This allows us to calculate how many clock cycles each instruction takes and how to optimize our code/logic to achieve faster execution. Make sure to read the entire lab as a part of the pre-lab (including the description of ARM instructions in the Appendix).

1. We assume you have already downloaded the Arduino IDE and configured it for the Itsy Bitsy. Instructions are in Lab 1, Prelab, Step 3.
2. We will be compiling the code to compute the Fourier Transform of a discrete sample data and detect the peak frequency present in it. Go over the code (listed in next step) and understand the math behind it.
3. Paste the following code in the Arduino IDE and **name your file FT_PeakDetection** or open the file provided from the syllabus:

```
#define MAX_SAMPLES 128 //make it a power of 2
#define SAMPLING_TIME 0.00002 //Amount of time between samples
#define pi 3.1416
#define scale_factor 100
#define ADC ADC0
int samples[MAX_SAMPLES];
boolean samplesReceived = false;
int sineref[MAX_SAMPLES];
int cosref[MAX_SAMPLES];
long ask;
long ack;
double freq[MAX_SAMPLES];
double samp_freq;
int i;
int k;
double fft_abs;
void setup() {
  Serial.begin(9600); // setup serial monitor speed
  while (!Serial); // wait for serial to be ready
  Serial.println("Hello World;");
  pinMode(A0, INPUT_PULLUP);
  ADC -> CTRLA.reg &= 0b1111100011111111; // mask PRESCALER bits
```

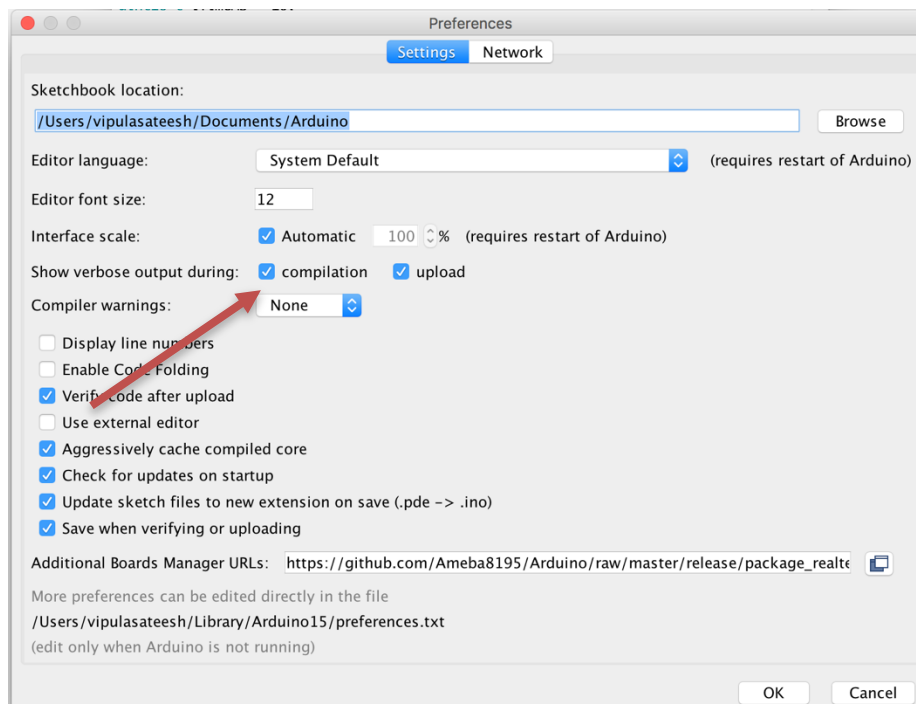
ESE 150 – Lab 08: Machine Level Language

```
ADC -> CTRLA.reg |= ADC_CTRLA_PRESCALER_DIV16; // divide Clock by 16 (original was 64)
// this PRESCALAR does appear to impact sample rate...smaller divider ... faster sample
ADC -> AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_1 | // take 1 sample
    ADC_AVGCTRL_ADJRES(0x00ul); // adjusting result by 0
ADC -> SAMPCTRL.reg = 0x00;
samp_freq = 1 / (SAMPLING_TIME + 0.0000185); // Includes ~analogread delay
}
long dotproduct(int length, int * a, int * b) {
    long sum = 0L; // Initializing long
    for (int i = 0; i < length; i++) {
        sum += (long) a[i] * (long) b[i]; //type casting
    }
    return (sum);
}
void loop() {
    for (int i = 0; i < MAX_SAMPLES; i++) {
        samples[i] = analogRead(A0); // read input from A0
        delayMicroseconds(SAMPLING_TIME * 1000000);
        samplesReceived = true;
    }
    if (samplesReceived) {
        for (int i = 0; i < MAX_SAMPLES; i++) {
            Serial.println(samples[i]);
        }
        for (k = 0; k < MAX_SAMPLES; k++) {
            for (i = 0; i < MAX_SAMPLES; i++) {
                sineref[i] = (int)(sin(2 * pi * k * i / MAX_SAMPLES) * (1 << 10)); //sine and cosine samples will be
                    between -1 and 1, so multiply those by 2^10 and round
                cosref[i] = (int)(cos(2 * pi * k * i / MAX_SAMPLES) * (1 << 10));
            }
            ask = (dotproduct(MAX_SAMPLES, samples, sineref)) / (MAX_SAMPLES * scale_factor); //scaling it
            ack = dotproduct(MAX_SAMPLES, samples, cosref) / (MAX_SAMPLES * scale_factor);
            freq[k] = k * samp_freq / (MAX_SAMPLES - 1);
            fft_abs = abs(ask) + abs(ack); // abs returns the modulus

            if (fft_abs > 200) {
                Serial.println("Peak found at ");
                Serial.println(freq[k]);
            }
        }
        while (1); //Run code once
    }
}
```

ESE 150 – Lab 08: Machine Level Language


- For MacOs follow steps 5 through 12. For Windows skip to step 13.
- Now to see compiler outputs, do the following to change the settings of the Arduino IDE:



- Enable “Show Verbose Output during Compilation and Upload”.
- Now, compile (verify) your code to see the output files (ELF, BIN, HEX files).
- Once you compile you will see something like this:

```
Done compiling.
Compiling sketch...
/Users/aaronshurberg/Library/Arduino15/packages/adafruit/tools/arm-none-eabi-gcc/9-2019q4/bin/arm-none-eabi-g++ -mcpu=cortex-m4 -m
Compiling libraries...
Compiling core...
Using previously compiled file: /var/folders/gj/qjbz18wn0s3fwszm4pmln180000gn/T/arduino_build_352763/core/variant.cpp.o
Using precompiled core: /var/folders/gj/qjbz18wn0s3fwszm4pmln180000gn/T/arduino_cache_709211/core/core_9954265f1d83f71ac49244a0596
Linking everything together...
/Users/aaronshurberg/Library/Arduino15/packages/adafruit/tools/arm-none-eabi-gcc/9-2019q4/bin/arm-none-eabi-g++ -L/var/folders/gj/q
/Users/aaronshurberg/Library/Arduino15/packages/adafruit/tools/arm-none-eabi-gcc/9-2019q4/bin/arm-none-eabi-objcopy -O binary /var
/Users/aaronshurberg/Library/Arduino15/packages/adafruit/tools/arm-none-eabi-gcc/9-2019q4/bin/arm-none-eabi-objcopy -O ihex -R .eep
/Users/aaronshurberg/Library/Arduino15/packages/adafruit/tools/arm-none-eabi-gcc/9-2019q4/bin/arm-none-eabi-size -A /var/folders/gj
Sketch uses 20324 bytes (4%) of program storage space. Maximum is 507904 bytes.
```

ESE 150 – Lab 08: Machine Level Language



```
e-linker-plugin -Wl,--gc-sections -mmcu=atmega328p -o "/var/folders/xm/gjy8bdj12qx81bm3jzfmnr  
om --set-section-flags=.eeprom=alloc,load --no-change-warnings --change-section-lma .eeprom=0  
om "/var/folders/xm/gjy8bdj12qx81bm3jzfmnlnm0000gn/T/arduino_build_960265/FFT_lab8.ino.elf" '
```

The directory referenced in the first photo,

`/Users/aaronshurberg/Library/Arduino15/packages/adafruit/tools/arm-none-eabi-gcc/9-2019q4/bin/` contains a file called `arm-none-eabi-objdump` which will be used in step 9. Scroll right to see the .elf file's location. This location will be specific to your machine.

The directory referenced in the second photo,

`/var/folders/gj/qjbzl8wn0s3fwszm4pmlndn180000gn/T/arduino_build_352763/FT_PeakDetection.ino.elf`, is where the elf file is stored.

- a. If you have trouble finding either path it may be easier to copy the output to a text editor and search around for it.
9. Now navigate to your desktop (`cd ~/Desktop`) or wherever you choose to store the lab files in terminal.
 - a. Create a new directory: `mkdir lab8`
 - b. Change into that directory: `cd lab8`

10. Now, generate the assembly file. Type the following in the terminal press enter:

```
/Users/aaronshurberg/Library/Arduino15/packages/adafruit/tools/arm-none-eabi-gcc/9-  
2019q4/bin/arm-none-eabi-objdump -S  
/var/folders/gj/qjbzl8wn0s3fwszm4pmlndn180000gn/T/arduino_build_352763/FT_PeakDetection.ino.e  
lf > d.txt
```

Here you should replace the location with the specific locations you obtained from the Arduino console. Also observe here, `d.txt` is the text file you're storing the assembly code in, and `FFT_ESE150Lab8.ino.elf` is the name of your Arduino file.

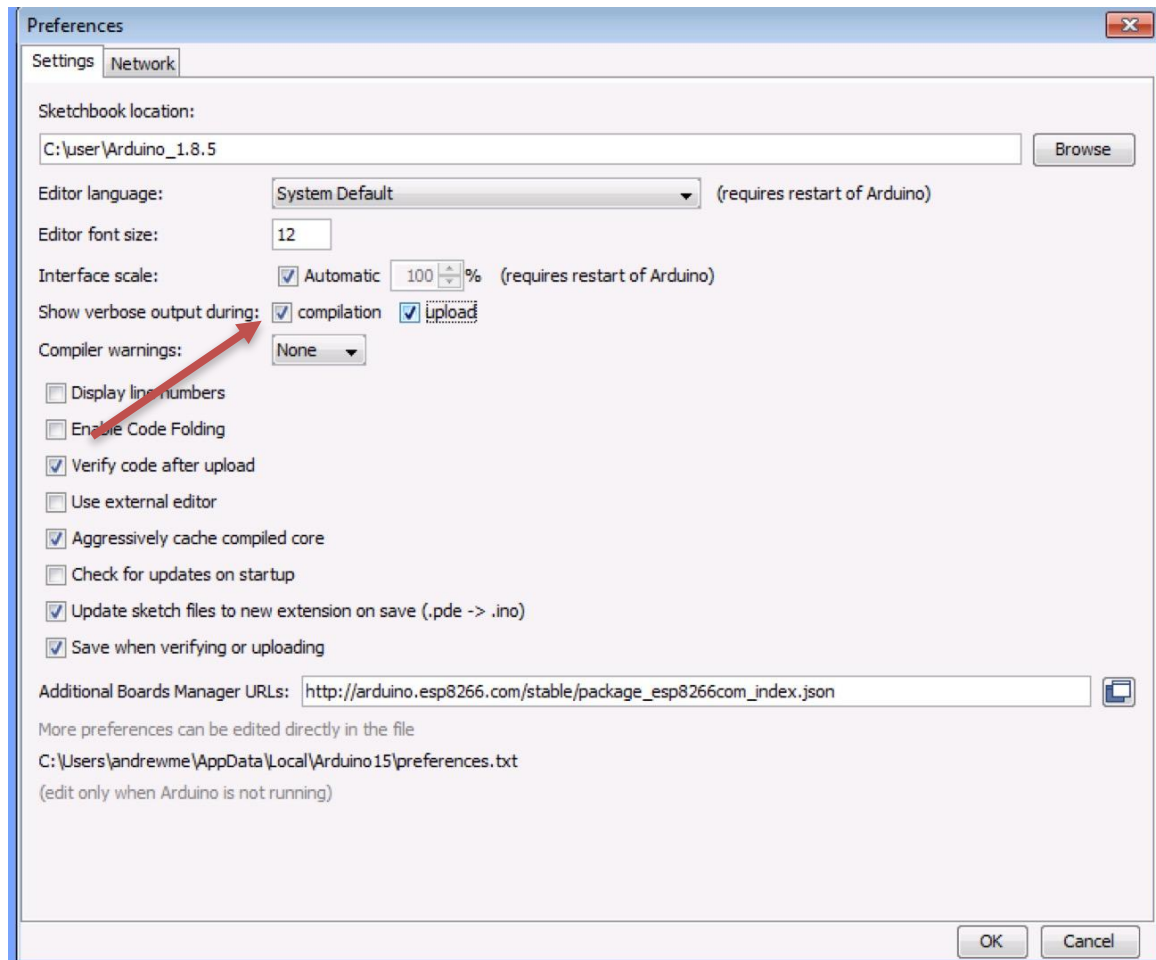
This command ("`>`") is redirecting stdout to a text file called "`d.txt`".

Note: There are other ways to generate the assembly code, and some applications do it for us by taking in the elf file. For more information on this you can refer to the link below as an example.

<https://sourceforge.net/projects/arduino-to-assembly-converter/>

- a. Note if either of your paths contain spaces then wrap them in quotes (`'`).
11. The file `d.txt` should now be saved in the directory from step 8.
 - a. This file should be included in the prelab canvas submission
 12. You can now skip to step 21.

ESE 150 – Lab 08: Machine Level Language



13. Go to File->Preferences and then enable “Show Verbose Output during Compilation and Upload”.
14. Now, compile your code to see the output files (ELF, BIN, HEX files).
15. Once you compile you will see something like this:

```
linking everything together...
C:\Users\kdepa\OneDrive\Documents\ArduinoData\packages\adafruit\tools\arm-none-eabi-gcc\9-2019q4/bin/arm-none-eabi-g++ "-LC:\Users\kdepa\AppData
C:\Users\kdepa\OneDrive\Documents\ArduinoData\packages\adafruit\tools\arm-none-eabi-gcc\9-2019q4/bin/arm-none-eabi-objcopy" -O binary "C:\Users\kde
C:\Users\kdepa\OneDrive\Documents\ArduinoData\packages\adafruit\tools\arm-none-eabi-gcc\9-2019q4/bin/arm-none-eabi-objcopy" -O ihex -R .eeprom "C:\
C:\Users\kdepa\OneDrive\Documents\ArduinoData\packages\adafruit\tools\arm-none-eabi-gcc\9-2019q4/bin/arm-none-eabi-size" -A "C:\Users\kdepa\AppData
Sketch uses 19852 bytes (3%) of program storage space. Maximum is 507904 bytes.
```

16. The directory "**C:\Users\kdepa\OneDrive\Documents\ArduinoData\packages\adafruit\tools\arm-none-eabi-gcc\9-2019q4/bin/**" contains a file called **arm-none-eabi-objdump** which will be used in step 19. The .elf's location is to the right. This location is specific to your machine.
17. Note the directory ("**C:\Users\kdepa\AppData\Local\Temp\arduino_build_413329**") where the **FT_PeakDetection.ino.elf** is stored highlighted below:

```
de7fdc69ea7bff9fd48e80a864b167.a
2019q4/bin/arm-none-eabi-g++ "-LC:\Users\kdepa\AppData\Local\Temp\arduino_build_413329" -Os -Wl,--gc-sections -save-temps -TC:\Users\kdepa\OneDri
2019q4/bin/arm-none-eabi-objcopy" -O binary "C:\Users\kdepa\AppData\Local\Temp\arduino_build_413329\FT_PeakDetection.ino.elf" "C:\Users\kdepa\AppData
2019q4/bin/arm-none-eabi-objcopy" -O ihex -R .eeprom "C:\Users\kdepa\AppData\Local\Temp\arduino_build_413329\FT_PeakDetection.ino.elf" "C:\Users\kdepa
2019q4/bin/arm-none-eabi-size" -A "C:\Users\kdepa\AppData\Local\Temp\arduino_build_413329\FT_PeakDetection.ino.elf"
```

- a. If you have trouble finding either path it may be easier to copy the output to a text editor and search around for it.
18. Now navigate to your Documents (cd C:\Users\kdepa\Documents) or wherever you choose to store the lab files in a Command Prompt terminal (search in the Start Menu if you don't know where to find it)

ESE 150 – Lab 08: Machine Level Language

- a. Create a new directory: `mkdir lab8`
- b. Change into that directory: `cd lab8`

19. Now let's generate assembly code from the .elf file generated. Run the following in the command prompt:

```
C:\Users\kdepa\OneDrive\Documents\ArduinoData\packages\adafruit\tools\arm-none-eabi-gcc\9-2019q4\bin\arm-none-eabi-objdump.exe" -S
```

```
"C:\Users\kdepa\AppData\Local\Temp\arduino_build_413329\FT_PeakDetection.ino.elf" > d.txt
```

```
C:\Users\kdepa\OneDrive\A Penn\ESE_150\labs\lab8>C:\Users\kdepa\OneDrive\Documents\ArduinoData\packages\adafruit\tools\arm-none-eabi-gcc\9-2019q4\bin\arm-none-eabi-objdump.exe" -S "C:\Users\kdepa\AppData\Local\Temp\arduino_build_413329\FT_PeakDetection.ino.elf" > d.txt
```

Here you should replace the location with the specific locations you obtained from the Arduino console. Also observe here, d.txt is the text file you're storing the assembly code in, and `FFT_ESE150Lab8.ino.elf` is the name of your Arduino file.

This command ("`>`") is redirecting stdout to a text file called "d.txt".

Note: There are other ways to generate the assembly code, and some applications do it for us by taking in the elf file. For more information on this you can refer to the link below as an example.

<https://sourceforge.net/projects/arduino-to-assembly-converter/>

20. The file d.txt should now be saved in the directory from step 18

- a. This file should be included in the prelab canvas submission

21. Now you have created d1.txt with your assembly code in it!

Familiarize yourself with the assembly code.

The more you look through it, the easier the lab will be.

Include this code in your Prelab Canvas submission.

22. Go over the code and identify the sections associated with a line of Arduino C code.

a) Search for third instance of the following (e.g. in the text editor Press CTRL+F and enter `sineref[i]=(int)(sin(2*pi*k*i/MAX_SAMPLES))*(1<<10));` to see how this command is executed.

b) You will obtain something like this:

```
439e:    4630        mov    r0, r6
43a0:    f003 fdf2    bl    7f88 <__aeabi_i2d>
43a4:    a342        add    r3, pc, #264 ; (adr r3, 44b0 <loop+0x178>)
43a6:    e9d3 2300    ldrd  r2, r3, [r3]
```

...

Recall we looked at instructions like add in lecture. Going line by line:

- i) "mov r0,r6" instruction r6 is moved to r0.
- ii) "bl 7f88 <__aeabi_i2d >" instruction sets PC to 0x00007f88 (calls subroutine)
- iii) "add r3,pc, #264" instruction computes $r3=pc+264$
- iv) "ldrd r2, r3, [r3]" instruction writes two words from memory simultaneously. That is $address=signextend(r3)$, $r2=Mem[address]$, and $r3=Mem[address+4]$

c) Look up the dotproduct:

```
int dotproduct(int length, int *a, int *b)
```

Hint: For the branch to the top of the loop, look for a branch that branches back to the code near the beginning of dotproduct().

ESE 150 – Lab 08: Machine Level Language

23. You will need these in Section 3 (and your report). Submit your raw (10 or 19) and extracted code (dot product from 20c) to the Prelab assignment on Canvas.

ESE 150 – Lab 08: Machine Level Language

Lab Procedure:

Lab – Section 1: Fourier Transform using Itsy Bitsy

- In this section we'll use the Itsy Bitsy to compute the Fourier Transform of a signal and recheck the most prominent frequency.
- We will use the same circuit from Lab 1 as follows:
 1. Grab the male headphone jack, a black wire (common convention used for ground) and any other wire of your choosing:
 - a. We will use this headphone jack to sample audio that comes from your computer or phone.
 - b. ***If your computer or phone does not have a standard female headphone jack, please let a TA know.***
 - c. Below is a picture of the headphone jack and some wires.



- d. Connect the black wire into the green socket on the headphone jack. This is the socket that is neither Left (L) or Right (R). The symbol you see is often used for ground. Ground is a reference point for any voltage reading. All readings are relative to ground.
- e. Next, connect the non-black wire into either the Left or Right socket. It shouldn't matter which one you choose. This wire will be considered as the audio input.
- f. Finally, connect the headphone jack to your audio source. This could either be your computer or your phone.
- g. So far, you should have something similar to the following picture:

ESE 150 – Lab 08: Machine Level Language

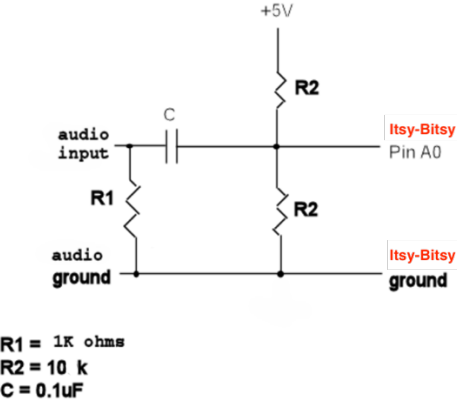


2. Next, grab **2x 10K Ohm**, **1x 1K Ohm** resistor and **1x 100nF** capacitor:
 - a. The resistors are the blue objects in the plastic bag. You can determine the value of the resistances by looking at the stamp near the bottom of the holding bands. It is important that you do not lose track of which resistors are which, so that you can reuse them in the future.
 - b. The 100nF (aka 0.1uF) capacitor is the red object in the plastic bin with a 104 label.
 - c. Here is a picture of the components:

ESE 150 – Lab 08: Machine Level Language



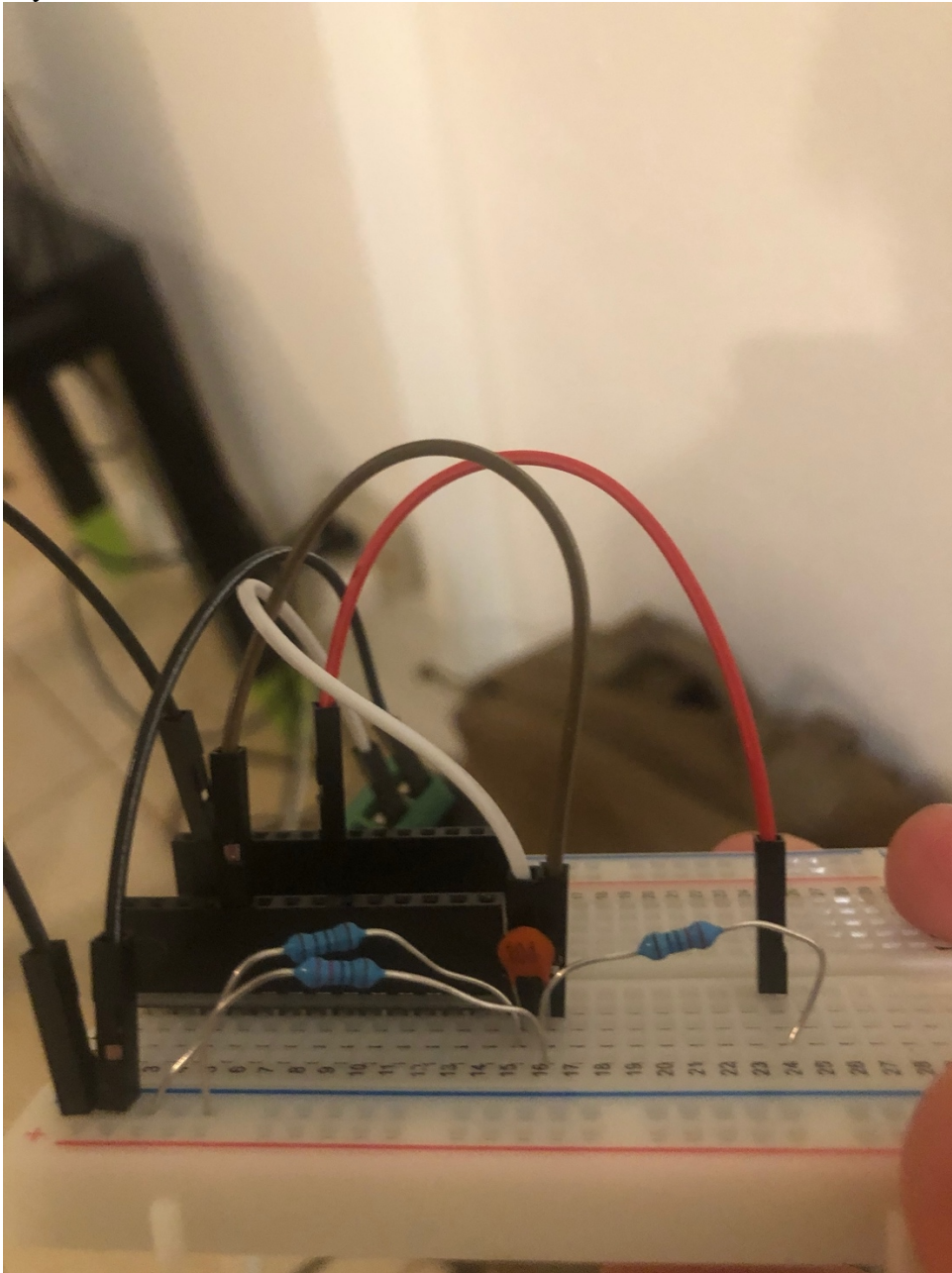
3. A diagram of the circuit you will build:



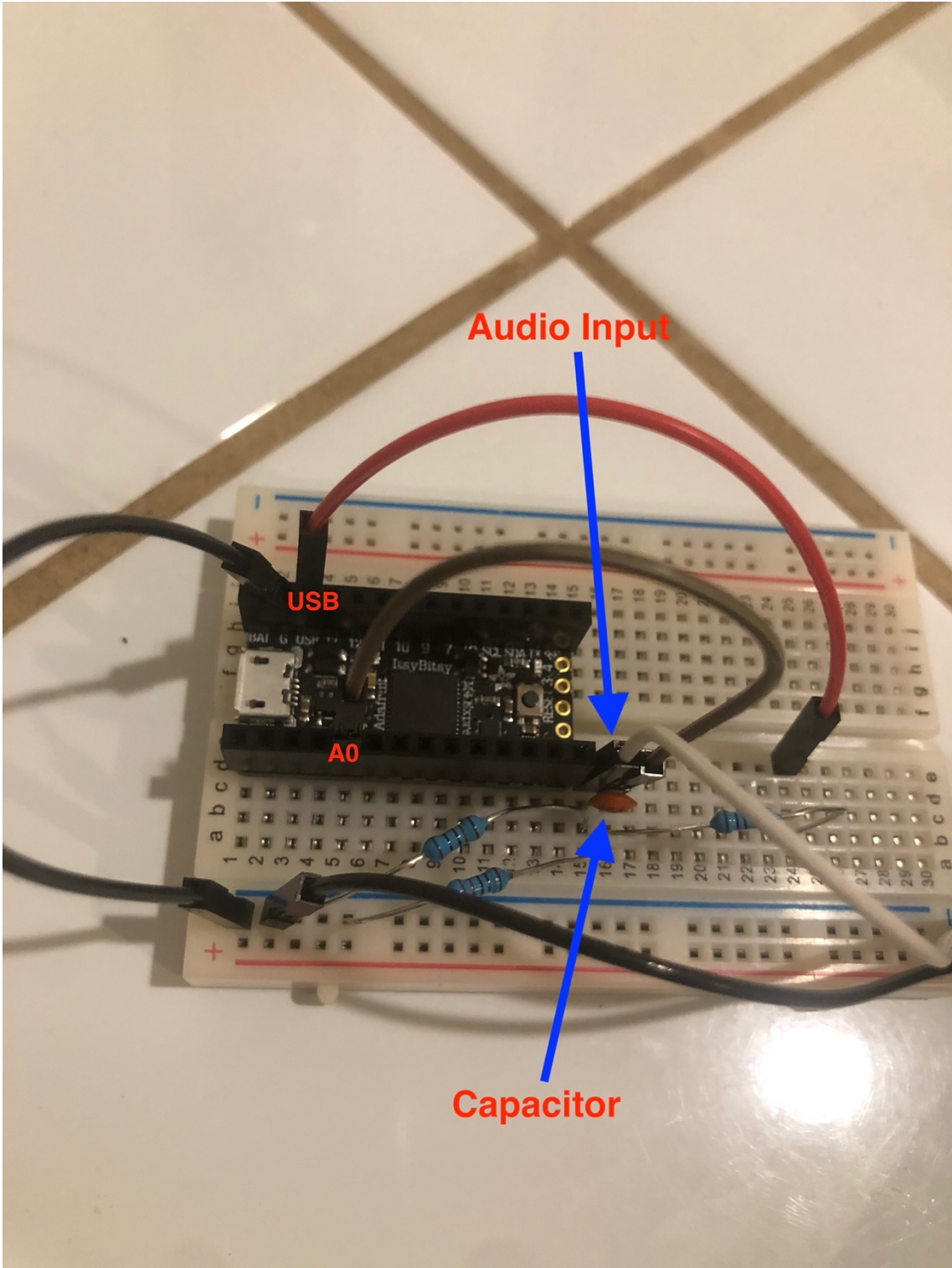
4. The next step is to connect the resistors and the capacitors. Both the 1K Ohm and one of the 10K Ohm resistors will connect to the common ground above, but they will go to different ends of the capacitor. It is best to have the capacitor away from any Itsy-Bitsy pins. The other 10K Ohm

ESE 150 – Lab 08: Machine Level Language

resistor will connect into the same side of the capacitor as its identical twin. This resistor can go to any other row on the breadboard.



5. Finally, connect the audio input wire into the same row of the capacitor as the 1kOhm resistor. Connect the other side of the last 10K Ohm resistor to the USB pin of the Itsy-Bitsy, and connect the other side of the capacitor to pin A0. The final product should look like this:



- 6. Download the provided 300 Hz MP3 from syllabus link.
- 7. Play the provided 300 Hz MP3 file. (Note that the file only runs for 1 minute; restart as necessary.)
- 8. Run the code from the prelab to sample the audio input and compute the Fourier Transform and print the most prominent frequencies.

ESE 150 – Lab 08: Machine Level Language

9. Identify which of the frequencies printed is the frequency of your signal and what the others could be. (Note: we haven't performed any calibration calculations to identify which frequency corresponds to each k.)

ESE 150 – Lab 08: Machine Level Language

Lab – Section2: Timing the Dot Product

- In this section, you'll time the dotproduct using the in-built micros() command.
- Finally, you'll calculate different size dotproducts and time taken for each.

1. Take a look at the micros() function in the Arduino library by clicking on the link below.

<https://www.arduino.cc/reference/en/language/functions/time/micros/>

2. Initialize two variables that will be used to calculate the time in micro seconds at two different points in your program to calculate how much time the Arduino takes to execute that section.

```
unsigned long microseconds1;  
unsigned long microseconds2;  
unsigned long time_elapsed;
```

3. Call the micros() function before and after the calculation of the dot product as shown below.

```
long dotproduct(int length, int *a, int *b) {  
    microseconds1 = micros();  
    long sum=0L;    // Initializing long  
    for(int i=0;i<length;i++) {  
        sum+=(long)a[i]*(long)b[i]; //type casting  
    }  
    microseconds2 = micros();  
    time_elapsed = (microseconds2 - microseconds1);  
    Serial.println(time_elapsed);  
    return(sum);  
}
```

4. The variable time_elapsed will contain the execution time in microseconds.
5. Now let's repeat the process for different size dotproducts and measure the execution time.
6. Replace the value of MAX_SAMPLES with different sizes (in powers of 2, like 256, 512, 1024,2048,4096) as shown below.
#define MAX_SAMPLES 256
7. Try increasing the number of samples to 16384 and run the program. Report the console message and reason behind it.
8. Tabulate and plot the values in Excel (time vs MAX_SAMPLES).
9. Show the TA the plot before you continue. Request for a TA to come to your breakout room on the assistance sheet
10. Review the plot and identify an equation that approximates the curve.

ESE 150 – Lab 08: Machine Level Language

Lab – Section 3: Computing the runtime for the Dot Product

- In this section we will go through the assembly code generated and check the number of cycles each instruction takes
- Finally, you'll calculate the runtime of the dotproduct

1. Refer to the dotproduct Assembly code generated and extracted in the pre-lab.
2. Now refer to the Appendix to see how many clock cycles each instruction takes (For example, add, mla, ldr, mov).
3. Next calculate clock cycles.
 - a. Total Cycles is defined by the following:

$$\sum_i (IC_i)(CC_i)$$

Where IC_i is the number of instructions for a given instruction type i , CC_i is the clock-cycles for that instruction type. The summation sums over all instruction types for a given benchmarking process.

- b. Make a spreadsheet table with one line for each instruction in dotproduct
 - i. Add two columns to the table for cycles for each of the instructions.
 1. One column for the maximum number of cycles the instruction takes
 2. One column for the minimum number of cycles the instruction takes
(see the Appendix for more details on this)
 - ii. Add another two columns for each instruction; one for the minimum number of times multiplied by MAX_SAMPLES and one for the maximum number of times multiplied by MAX_SAMPLES.
Hint: the number of times some instructions are called will depend on the value of MAX_SAMPLES, so actually create four columns here:
 - A max and min column for the number of multiples of MAX_SAMPLES the instruction is called
 - A max and min column for the number of times called that is not a multiple of MAX_SAMPLES
- c. Put MAX_SAMPLES in a spreadsheet cell.
- d. Add an equation that uses the table values to calculate total cycles.
Make this use the MAX_SAMPLES cell so you can get it to compute a Total Cycles estimate for each of the different MAX_SAMPLES values you previously measured.

4. Add an equation using the above to calculate the Execution time using the formula below:

$$\text{Execution Time} = \text{Total Cycles} \times \text{clock time}$$

Here the clock frequency is 120MHz.

ESE 150 – Lab 08: Machine Level Language

5. Compare the time obtained in this to the time obtained from Section 2.

6. Show the TAs the value and explain your discrepancy.

a. Request a TA come view your calculated values on the assistance sheet.

b. The TA will ask you a few questions.

c. This is your exit check off.

ESE 150 – Lab 08: Machine Level Language

Postlab

1. Reviewing the Instruction Set table in the Appendix:
 - Why does the MLA takes more clock cycles than ADD?
2. How much faster could you perform this computation with an architecture set to perform one multiply-accumulate in the dot-product per cycle.
 - The ARM has a powerful mla instruction that can perform a multiply-accumulate in 2 cycles.
 - However, in the dot-product loop, you don't get too issue that instruction every cycle. Instead you must perform a number of other operations to setup for each mla instruction.
 - 2.●.1. How many instructions and cycles must your ARM processor currently perform for each mla instruction?
 - We are imagining a more powerful processor that could do the useful part of all of these other instructions at the same time as the mla instruction.
 - How much faster would the dot product be computed when using an architecture with this single-cycle multiply-accumulate?
3. For what MAX_SAMPLE window sizes can you sample data and compute the Fourier Transform in real time?
 - For our 50000 samples per second rate, calculate how many clock cycles the Arduino could compute between samples. Call this ctime.
 - From the lab, you know how many clock cycles it takes to compute the dotproduct for a particular values of MAX_SAMPLES. Call this function dp_time(MAX_SAMPLES).
 - Further, you know it takes roughly $2 \times \text{MAX_SAMPLES} \times \text{dp_time}(\text{MAX_SAMPLES})$ to compute the Fourier Transform
 - For what values of MAX_SAMPLES is $2 \times \text{MAX_SAMPLES} \times \text{dp_time}(\text{MAX_SAMPLES}) < \text{ctime} \times \text{MAX_SAMPLES}$?
 - How does the maximum MAX_SAMPLES window size change if we sample at 16000 samples per second?

HOW TO TURN IN THE LAB

- Each student should assemble an individual writeup and upload as a PDF document to canvas, containing:
 - dotproduct \extracted code from prelab
 - Console message and explanation from Section 2, step 7
 - Arduino C code for time calculations
 - Table and plot of different DFT MAX_SAMPLES vs time taken for execution
 - Calculation of your Total Cycles with spreadsheet table and any assumptions made
 - Calculation of Execution time and inference from the comparison
 - Answers to any other highlighted questions in the lab
 - Answers to post-lab

ESE 150 – Lab 08: Machine Level Language

Appendix: Armv7-M Microcontroller Instructions

Section I:

To run code that is written in the Arduino IDE on the actual Itsy Bitsy Processor processor (ARM Cortex M4), a series of steps need to be taken. These will all be covered in more detail in later classes, starting with CIS240.

The processor cannot execute high level Arduino code that we write; what it executes is machine code, a series of bits that represent instructions telling the machine what to do. Each processor has a specific set of instructions that it “knows” what to do with; these are defined in the Instruction Set Architecture (ISA) of the device. In this lab we are using the ARM ISA of the Itsy Bitsy’s processor, but the concepts are extensible to any other device.

In between the high level code and the machine code is assembly. In assembly each line is a direct translation of the machine code but is human readable. In a high level code it’s not immediately clear what instructions correspond to complex structures such as a for loop, or an if statement. However, in assembly, those logical structures are already broken down into single lines of code that each reference one instruction. Each instruction may either reference a memory address, i.e. to jump to a subroutine stored at the address, or load data from that address, or it may reference “registers”. Registers are special locations in memory that can be used to store values when we want to do mathematical operations on them. (Additionally, there are certain special registers that are used to store information such as “C” the carry flag. Similarly, there is a status register that contains bits for N, Z, and V. These bits refer to negative, zero, and overflow, and are set by certain comparison, move, and arithmetic instructions. You can read more about these in the full datasheet linked below). To understand the assembly, we need to understand each instruction that the processor supports. Section II of this appendix details the instructions necessary for the lab; more details, as well as all the supported instructions can be found in the datasheet linked below.

A typical line of assembly could look like this:

```
43a4:      a342      add    r3, pc, #264    ; (adr r3, 44b0 <loop+0x178>)
```

Here 43a4 is the memory location, represented in hexadecimal, where this instruction is stored in the Arduino. The a342 is the actual string of bits (machine code), represented in hex, that are the instruction. The “add r3, pc, #264” is the human readable version of that instruction.

In assembly anything after a semicolon (;) is a comment, here the comment is letting us know that the memory location 43a4 that the assembly command is equivalent to the form address (adr) and putting 44b0 in r3 where 44b0 is <loop+0x178>. <loop+0x178> says the address 0x178 bytes after the loop label.

ESE 150 – Lab 08: Machine Level Language

Section II:

Below are the instructions that you will find in this lab and the relevant information for each. For the purposes of the lab, while it will be helpful to have a basic understanding of what each instruction does when identifying what parts of the Arduino code they refer to, you do not need to fully understand how to translate the Arduino code to assembly or vice-versa. This will be covered more in CIS240 and beyond. You should focus most on tracing through the execution of the assembly code and figuring out how many cycles each instruction takes.

Note that some instructions can take a variable number of cycles, indicated in the chart by the 2 columns for min and max cycles (depending on various things, such as if a condition is true or false when the instruction is executed). For the lab, this is what is meant by maximum and minimum number of cycles. For example, b might take 2 cycles if the branch prediction is correct but 4 if it is not. Note that whether the condition is true or not will affect what code runs, which further affects how long the entire program will take (i.e. in the high level code an if block might take a lot longer to run than the corresponding else block). For this lab we just want to look at the maximum and minimum number of cycles of each individual instruction, ignoring the effects taking a certain branch might have on the logical runtime of the code.

<u>Instruction Semantics</u>	<u>Instruction Description</u>	<u>Min Number of Cycles</u>	<u>Max Number of Cycles</u>
ldr <c><q> <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]	Load a value in memory into a register: <Rt>=Memory[<Rn>+(Rm << <shift>]	1	2
b<c><q> <label>	Branch based on condition <c> or unconditional if omitted. PC=PC+<label>	1 (not taken if conditional)	4
bl<c><q> <label>	Branch based on condition <c> or unconditional if omitted to a subroutine. PC=PC+<label>	2	4
mov{s}<c><q> <Rd>, #<const>	<Rd>=#<const> {s}-optionally updates flags with s set	1	1
cmp <q> <Rn>, <Rm> {, <shift>}	Computes Rn-Rm and updates the flags based on the result optionally	1	1

ESE 150 – Lab 08: Machine Level Language

mla <q> <Rd> <Rn> <Rm>	Multiply and accumulate. Rd=Rn*Rm	2	2
add{s} <Rd> <Rn> #<const>	Does the following addition Rd=Rn+const. {s}-optionally updates flags with s set	1	1
Push <registers>	Push registers onto the stack potentially including LR	1+N (where N is number of registers including LR)	1+N (where N is number of registers including LR)
Pop <registers>	Pop values off the stack into the list of registers potentially including PC	2+N	4+N

Notes:

<q> is either N or W with N meaning narrow (16-bit) encoding and W meaning wide (32-bit) encoding
N = number of register to process in list including PC/LR

LR = link register which contains the address to return when function call finishes

This appendix should suffice for our lab. To see more details of the ARMv7-M ISA see

<https://developer.arm.com/documentation/ddi0403/latest/> and

<https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions?lang=en>