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 Arduino 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 a .elf file and .hex file is.
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.

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 Arduino instructions in the Appendix).

1. If you haven't already done so, download the Arduino IDE to your laptop: https://www.arduino.cc/en/Main/Software

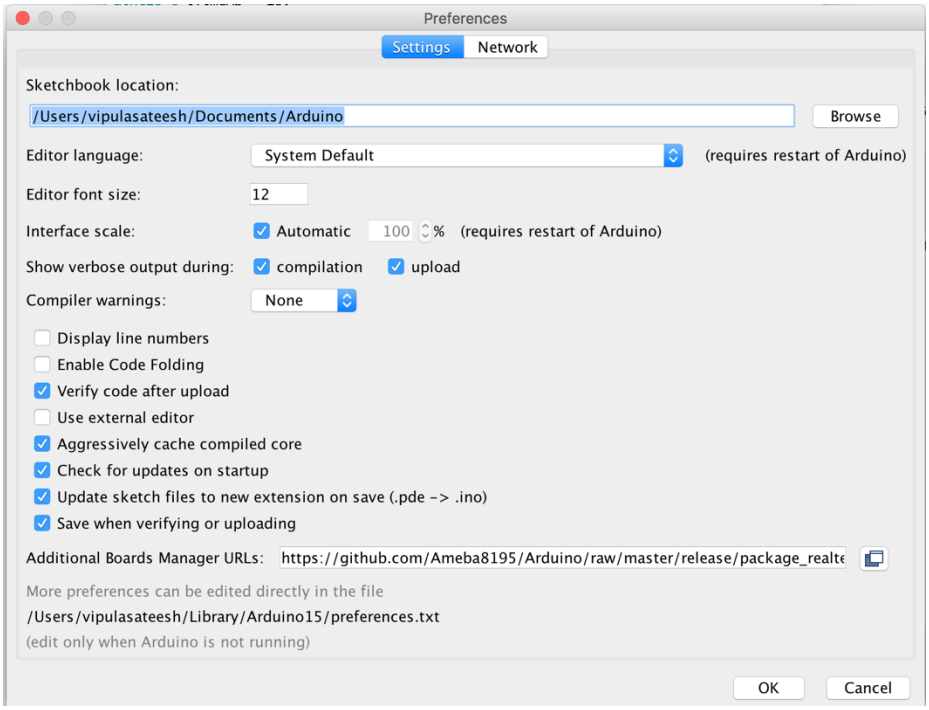
2. We will be compiling the code to compute the Fourier Transform of a discrete sample data and detect the peak frequency present in it. Go over the code (listed in next step) and understand the math behind it.

3. Paste the following code in the Arduino IDE and name your file FT_PeakDetection.

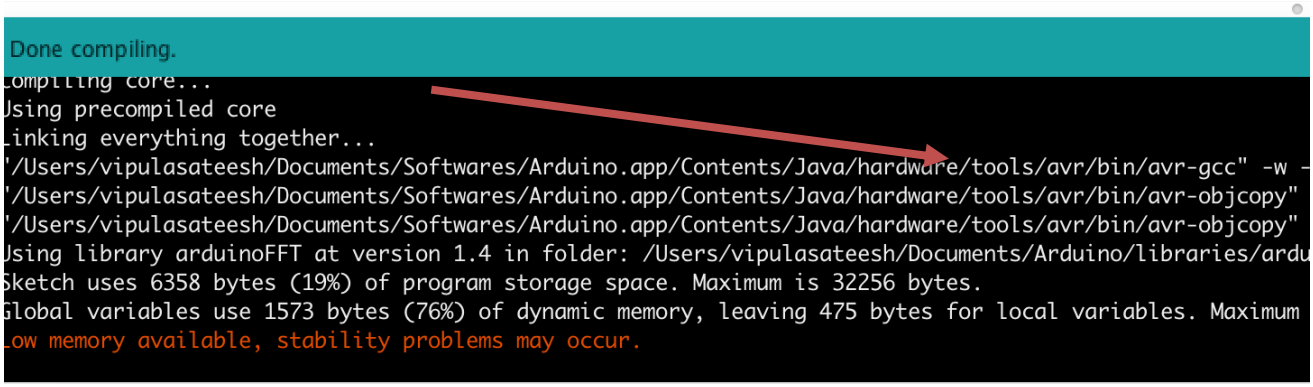
```
#define MAX_SAMPLES 128    //make it a power of 2
#define SAMPLING_TIME 0.0002 //Hz, must be less than 10000 due to ADC
#define pi 3.1416
#define scale_factor 100
int samples[MAX_SAMPLES] ;
boolean samplesReceived = false ;
int sineref[MAX_SAMPLES];
int cosref[MAX_SAMPLES];
long ask;
long ack;
double freq[MAX_SAMPLES];
double samp_freq;
int k = 0 ;
int i ;
double fft_abs;

void setup() {
  Serial.begin(9600);  // setup serial monitor speed
  samp_freq = 1/(SAMPLING_TIME + 0.000125);    //Analog read  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(200);
  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);
   /* Serial.print("For :");
     Serial.println(freq[k]);
     Serial.println(fft_abs);    printing the amplitude for each freq */
     fft_abs = abs(ask)+abs(ack);   // abs returns the modulous

    if (fft_abs>500){ // you may need to adjust this value
     Serial.println("Peak  found at ");
       Serial.println(freq[k]);
     }
  }
  while(1);       //Run code once
 }
}
```

4. Now to see compiler outputs, do the following to change the settings of the Arduino IDE:
   For MacOs follow steps 5 through 11. For Windows skip to step 12.



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



The directory /Users/vipulasateesh/Documents/Softwares/Arduino.app/Contents/Java/hardware/tools/avr/bin contains a file called avr-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.

8. Now let's generate assembly code from the .elf file generated. Open your terminal (search in Spotlight if you don't know where to find it) and type the following.
   a. cd /var/folders/xm/gjy8bdj12qx81bm3jzfmnnlm0000gn/T/arduino_build_232470

being sure to replace the path after cd with the specific path you found above.

   b. pwd
   c. ls

You should see something like:

```
[Vipulas-MacBook-Pro:arduino_build_839356 vipulasateesh$ cd ~                    ]
[Vipulas-MacBook-Pro:~ vipulasateesh$ cd /var/folders/xm/gjy8bdj12qx81bm3jzfmnnlm]
 0000gn/T/arduino_build_232470
[Vipulas-MacBook-Pro:arduino_build_232470 vipulasateesh$ pwd                     ]
 /var/folders/xm/gjy8bdj12qx81bm3jzfmnnlm0000gn/T/arduino_build_232470
[Vipulas-MacBook-Pro:arduino_build_232470 vipulasateesh$ ls                      ]
 FFT_ESE150Lab8.ino.eep                  core
 FFT_ESE150Lab8.ino.elf                  includes.cache
 FFT_ESE150Lab8.ino.hex                  libraries
 FFT_ESE150Lab8.ino.with_bootloader.hex  preproc
 build.options.json                      sketch
 Vipulas-MacBook-Pro:arduino_build_232470 vipulasateesh$
```

The command cd is used to open the directory which is mentioned after the command. The command ls is used to obtain all the files and folders in the current directory.

Note the FFT_ESE150Lab8.ino.elf file, which is used in our next step.

9. Now, generate the assembly file. ***Here again you should replace the location with the specific location 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.

/Users/vipulasateesh/Documents/Softwares/Arduino.app/Contents/Java/hardware/tools/avr/bin/avr-objdump -S FFT_ ESE150Lab8.ino.elf >d.txt

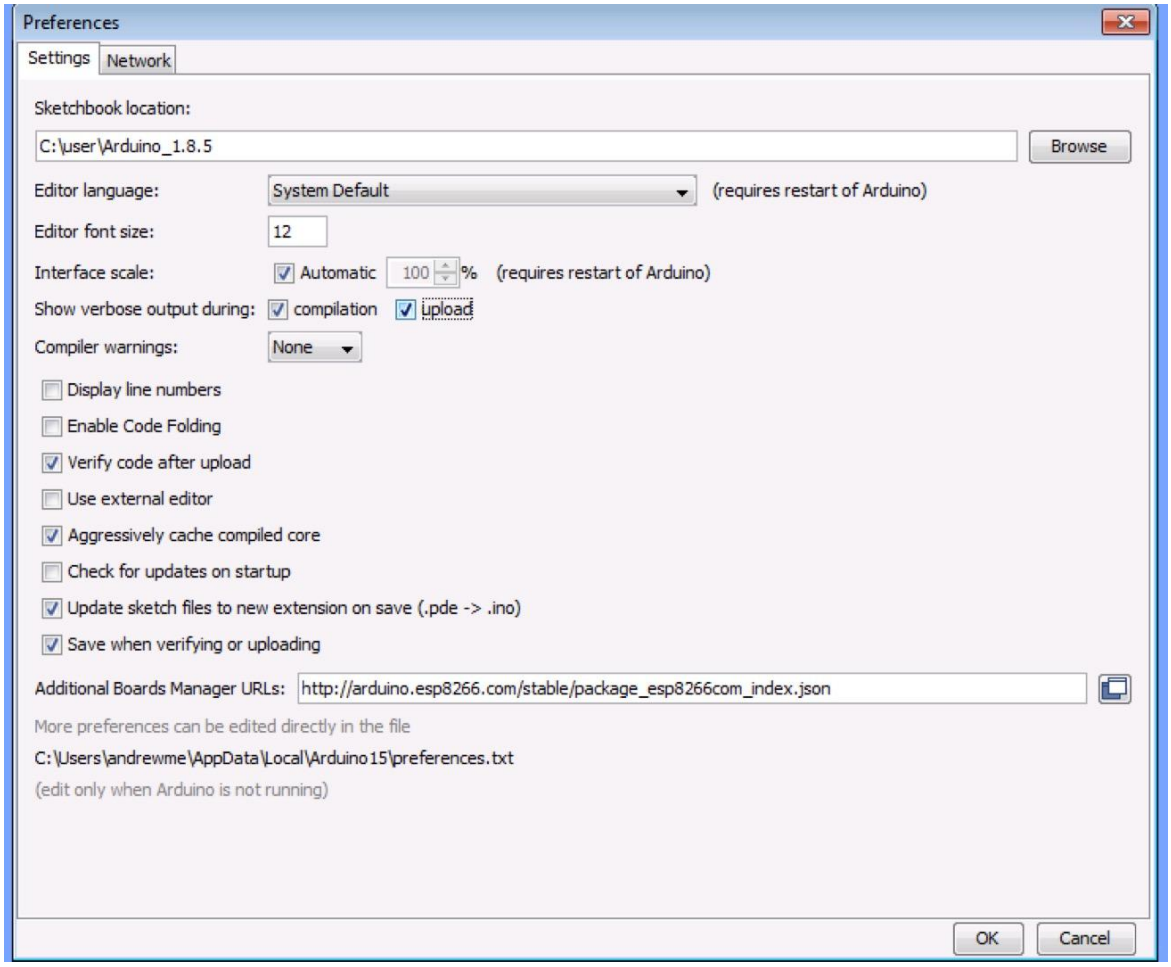
This command (">") is redirecting stdout to a text file called "d.txt".

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/

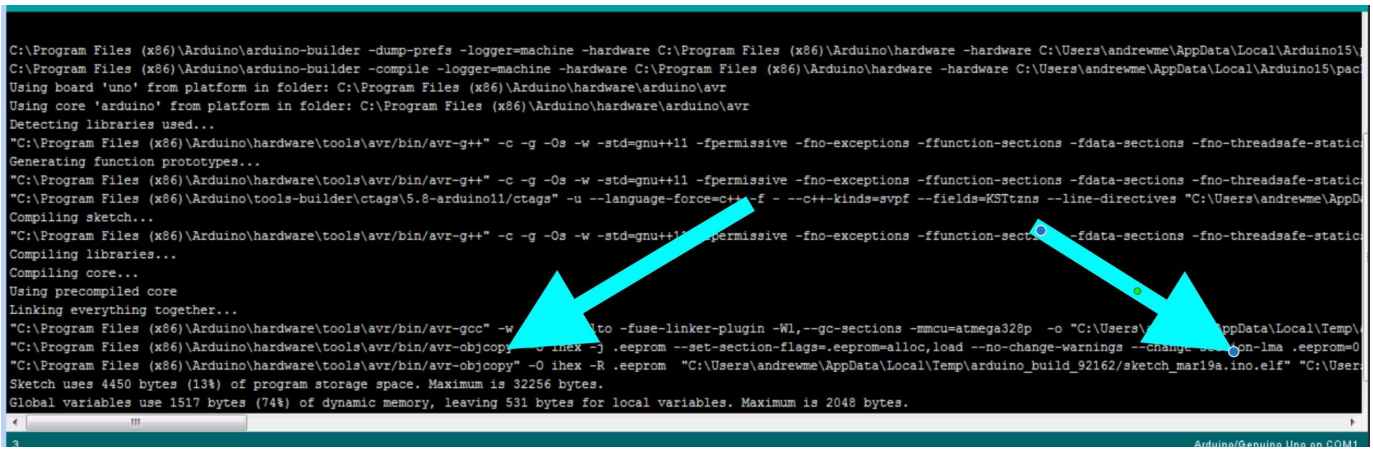
10. Copy paste the text file onto another text file which is more accessible. To do that type in the following command and ***replace the destination location with the directory of your choice***.

cp -X d.txt /Users/vipulasateesh/Documents/Penn/d1.txt

11. You can now skip to step 20.

12. 

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



15. The directory "C:\Program Files (x86)\Arduino\hardware\tools\avr/bin/" contains a file called avr-objdump which will be used in step 5. The .elf’s location is to the right.

16. Now let's generate assembly code from the .elf file generated. Open a PowerShell terminal (search in the Start Menu if you don't know where to find it) and type the following.
    a. cd "C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162" ***being sure to replace the path after cd with the specific path you found above.***
    b. pwd
    c. ls

You should see something like:

```
PS C:\Program Files (x86)\Arduino\hardware\tools\avr\bin> cd "C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162/"

PS C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162> pwd

Path
----
C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162


PS C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162> ls


    Directory: C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        3/19/2019   1:36 AM                core
d-----        3/19/2019   1:35 AM                libraries
d-----        3/19/2019   1:35 AM                preproc
d-----        3/19/2019   1:37 AM                sketch
-a----        3/19/2019   1:37 AM            935 build.options.json
-a----        3/19/2019   1:37 AM            459 includes.cache
-a----        3/19/2019   1:37 AM             13 sketch_mar19a.ino.eep
-a----        3/19/2019   1:37 AM          29524 sketch_mar19a.ino.elf
-a----        3/19/2019   1:37 AM          12540 sketch_mar19a.ino.hex
-a----        3/19/2019   1:37 AM          13680 sketch_mar19a.ino.with_bootloader.hex


PS C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162> _
```

17. The command cd is used to open the directory which is mentioned after the command. The command ls is used to obtain all the files and folders in the current directory.
    Note the sketch_mar19.ino.elf file, which is used in our next step.
18. Now, generate the assembly file. ***Here again you should replace the location with the specific location you obtained from the Arduino console***. Also observe here, d.txt is the text file you're storing the assembly code in, and sketch_mar19.ino.elf is the name of your Arduino file. **Note the & and ` characters as well as the location of the quotes.**

```
&"C:\Program Files (x86)\Arduino\hardware\tools\avr/bin/avr-objdump.exe" `-S
"C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162/sketch_mar19a.ino.elf"
> d.txt
```

```
d.txt
PS C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162> &"C:\Program Files (x86)\Arduino\hardware\tools\avr/bin/avr
-objdump.exe"  `-S "C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162/sketch_mar19a.ino.elf" > d.txt
PS C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162> _
```

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/

19. Copy paste the text file onto another text file which is more accessible. To do that type in the following command and ***replace the destination location with the directory of your choice***.

cp d.txt /Users/vipulasateesh/Documents/Penn/d1.txt

```
PS C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162> cp d.txt /Users/andrewme/Desktop/d1.txt
PS C:\Users\andrewme\AppData\Local\Temp\arduino_build_92162> _
```

20. 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.

21. Go over the code and identify the sections associated with a line of Arduino C code.
a) Search for (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:

```
70e:    41 01       movw  r8, r2
710:    03 2c       mov   r0, r3
712:    00 0c       add   r0, r0
714:    aa 08       sbc   r10, r10
716:    bb 08       sbc   r11, r11
718:    c5 01       movw  r24, r10
71a:    b4 01       movw  r22, r8
71c:    0e 94 f2 04 call  0x9e4  ; 0x9e4 <__floatsisf>
```

c) Recall we looked at instructions like add in lecture. Going line by line:
i) "movw r8,r2" instruction moves a full word (2 bytes) r2:r3 is moved to r8:r9.
ii) "mov r0,r3" instruction moves r3 to r0
iii) "add r0,r0" instruction computes r0=r0+r0
iv) "sbc r10, r10" instruction computes r10=r10-r10-C where C is the carry out bit that was previously set
v) "sbc r11, r11" instruction computes r11=r11-r11-C where C is the carry out bit that was previously set
vi) "movw r24,r10" instruction moves a full word (2 bytes) r10:r11 is moved to r24:r25
vii) "movw r22,r8" instruction moves a full word (2 bytes) r8:r9 is moved to r22:r23
viii) "call 0x9e4" instruction on the address 71c means the control is now being passed on to the instruction by the address 0xc9e <__floatsisf>. The call instruction supports operations like a procedure call in Java or MATLAB. It is paired with a ret (return) instruction that will return control to the instruction following the call instructions. See Appendix for more details of Arduino instructions.

d) Look up the dotproduct:

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

This is actually a bit tricky. The compiler copied the code for dotproduct into the loop() routine at the places where it was called, so you will actually see that signature line 4 times. To confuse matters further, it shows it twice for each time it is copied into the code (once for computing ask, once for computing ack). Starting from the first occurrence, find where it performs the multiply (hint: __mulhisi3 subroutine), where it does the accumulating addition, and where it performs a branch

back to the top of the loop. Notice that a divide follows the branch at the end of the loop. This divide (__divmodsi4 subroutine) corresponds to the division you see in loop() following the dotproduct() call in the C code. Following that you can see where it stores the result into the ask variable. From these landmarks, identify and extract the code for the dotproduct; include it in your Prelab Canvas submission.

Hint: Follow through the code for a few iterations and see the different subroutines that are jumped to when dotproduct() is called. For the branch to the top of the loop look for a branch that branches back to the code where dotproduct() begins.

Look further down and identify the dotproduct code for the computation of the ack variable. Note that it is the same sequence of instructions.

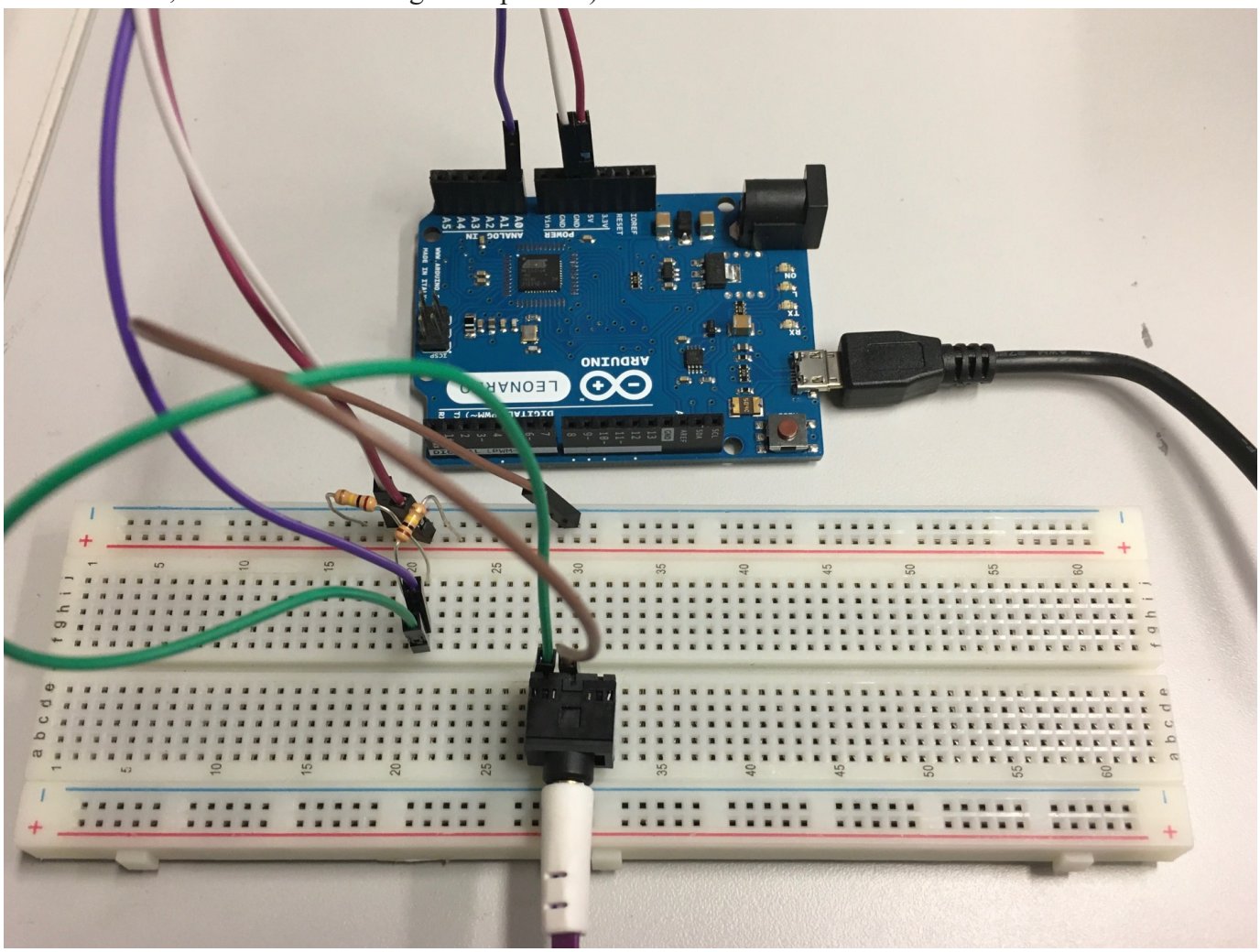
22. Extract the code for the __mulhisi3 subroutine that is used in the multiplication. Include all the routines that it calls or jumps to. Include all this extracted code in your Preclass Canvas submission.
23. You will need these in Section 3 (and your report). Submit your raw (20) and extracted code (21, 22) to the Prelab assignment on Canvas.

Lab Procedure:

Lab – Section 1: Fourier Transform using an Arduino

- In this section we'll use Arduino to compute the Fourier Transform of a signal and recheck the most prominent frequency.

1. Set up the Arduino with headphone jack, voltage divider, and audio cable.
    a. Connect the audio cable to the headphone jack and your computer's audio jack
    b. Connect the ground of the Arduino to the central pin of the jack
    c. Connect either of the outer pins of the jack to A0 on the Arduino
    d. Connect output pin of jack to power (+5V) and ground on the Arduino using the supplied 100KΩ resistors.
    e. (optionally: see https://forum.arduino.cc/index.php?topic=476900.0 for discussion of circuit details; we are not including the capacitor.)



2. Download the provided 300 Hz MP3 from syllabus link.
3. Play the provided 300 Hz MP3 file. (Note that the file only runs for 1 minute; restart as necessary.)

4. Run the code from the prelab to sample the audio input and compute the Fourier Transform and print the most prominent frequencies.
5. 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.)

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 64, 32,…,2) as shown below.

```
#define MAX_SAMPLES 64
```

7. Try increasing the number of samples to 264 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. Post a private message to the instructors on Piazza with your plot.
10. Review the plot and identify an equation that approximates the curve.

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, mul, ld, mov).
3. Next calculate clock cycles.
    a. Total Cycles is defined by the following:

$$\Sigma_i (IC_i)(CC_i)$$

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 and __umulhisi3 (and all the routines that __mulhsi3 calls).
        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:

Execution Time = $Total\ Cycles \times clock\ time$

Here the clock frequency is 16MHz.

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. Post your values (both measured and calculated max and min) in a private post in Piazza for a TA to review.
    b. The TA will followup with a private Zoom chat to ask you a few questions.
    c. This is your exit check off.

Postlab

1. Reviewing the Instruction Set table in the Appendix:
   - Why does the MUL takes more clock cycles than ADD?
2. If we were to design an enhanced Arduino (ATmega328) that included a hardware 16b×16b multiplier that was supported by a new instruction that could multiply two unsigned 16b integers in one cycle, how much faster would the dotproduct run on this enhanced Arduino compared to the Arduino you used in lab and evaluated in Section 3?
   - This instruction will replace the call to __mulhisi3. So, all the cycles that you calculate for making the call to __mulhisi3 and the cycles for the call will be replaced by a single cycle.
   - Hint: How many cycles does the current multiplication of 16-bit multiplication take? I.e. How long does __mulhisi take to run? Can you make the same chart as in Section 3, but with any instructions ran in __mulhisi removed?
3. For what MAX_SAMPLE window sizes can you sample data and compute the Fourier Transform in real time?
   - For our 5000 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×MAX_SAMPLES×dp_time(MAX_SAMPLES) to compute the Fourier Transform
   - When is 2×MAX_SAMPLES×dp_time(MAX_SAMPLES) < ctime×MAX_SAMPLES?
   - How does the maximum MAX_SAMPLES window size change if we sample at 1000 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 and __mulhisi3 extracted code from prelab
  - Console message and explanation from Section 2, step 7
  - Arduino C code for time calculations
  - Table and plot of different FFT 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

Appendix: Arduino (AVR) Microcontroller Instructions

Sectiom I:

To run code that is written in the Arduino IDE on the actual Arduino processor (ATmega328P), 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 ISA of the Arduino'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 and registers X Y and Z that are designed to store memory addresses. Similarly, there is a status register that contains bits for N, Z, and P. These bits refer to negative, zero, and positive, and are set by comparison instructions as well as implicitly set any time a value is stored in a register. They are used to make decisions in branch 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:

```
71c:    0e 94 f2 04     call    0x9e4  ; 0x9e4 <__floatsisf>
```

Here 71c is the memory location, represented in hexadecimal, where this instruction is stored in the Arduino. The 0e 94 f2 04 is the actual string of bits (machine code), represented in hex, that are the instruction. The call 0x9e4 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 0x9e4 that the assembly command references is referring to the subroutine titled "<__floatsisf>". So, this assembly instruction tells the Arduino to perform a call instruction to the __floatsisf subroutine. As with a procedure call in Java or MATLAB, when the called routine (__floatsisf) is done it returns to this routine. In the Arduino that is done with the ret instructions (see table below) that transfers control back to the instruction following this one.

Sectiom 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, brne might take 1 cycle if Z!=0 but 2 if Z=0. 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.

Instruction Semantics	Instruction Description	Min Number of Cycles	Max Number of Cycles
add Rd, Rr	Add without carry: Rd = Rr + Rd and C is set to the carry-out bit	1	1
adc Rd, Rr	Add with carrying: Rd = Rd + Rr + C (which was previously set); C is set to the new carry-out bit	1	1
and Rd, Rr	Logical And: Rd = Rd AND Rr	1	1
brne OFFSET	Branch Not Equal: If Z=0, Move the instruction execution (back or forward) by OFFSET.	1 `	2
brpl OFFSET	Branch if Positive: If N=0, Move the instruction execution (back or forward) by OFFSET.	1	2
call SUBROUTINE	Call subroutine: Move the instruction execution to instruction	4	4

	at SUBROUTINE, preparing to return when the subroutine is done		
cpc Rd, Rr	Compare with Carry: Compares Rd to Rr with subtraction, taking into account the previous carry bit, and sets the N and Z flags appropriately. Rd - Rr - C	1	1
cpi Rd, CONST	Compare with Constant: Compares Rd to CONST with subtraction, setting N and Z flags. Rd - CONST	1	1
eor Rd, Rr	Exclusive Or: Rd = Rd XOR Rr	1	1
jmp LOCATION	Jump: Move program execution to instruction at program memory LOCATION	3	3

ld Rd, X
ld Rd, Y
ld Rd, Z
	Load from Memory: Load data into Rd from memory at location specified in register X/Y/Z (that was set earlier). X/Y/Z leaves X/Y/Z unchanged. -(X/Y/Z) pre-decrements and (X/Y/Z)+ post-increments.	1	3

ldd Rd, (Y/Z)+OFFSET	Load from Memory with Displacement: Load data into Rd from memory at location (Y/Z) with OFFSET	1	3
ldi Rd, VALUE	Load Immediate: Load VALUE into Rd	1	1

mov Rd, Rr	Copy Register: Copies Rr into Rd	1	1
movw Rd, Rr	Copy Register Word: Copies a pair of registers (2 bytes = 1 word) to another pair. Rd = Rr, Rd + 1 = Rr + 1.	1	1
mul Rd, Rr	Multiply Unsigned: Multiplies unsigned numbers and places result in R0 and R1 (2 bytes). R1:R0 = Rd * Rr	2	2
ret	Return from Subroutine: Move program execution to location previously stored with CALL.	4	5
sbc Rd, Rr	Subtract with carrying: Rd = Rd - Rr - C (which was previously set); C is set to the new carry-out bit	1	1
sbrs Rr, BIT	Skip if Bit in Register Set: If bit number BIT in the value stored in Register Rr==1 then skip the next line of assembly code	1	3
std Y/Z + OFFSET, Rr	Store to Memory with Displacement: Store the contents of Register Rr into data memory location Y/Z, with OFFSET.	2	2
sts LOCATION, Rr	Store to Memory: Store the contents of Rd into data memory at LOCATION	1	2
sub Rd, Rr	Subtract without carry: Rd = Rd - Rd and C is set to the carry-out bit	1	1

This appendix should suffice for our lab. To see more details of the AVR Microcontroller and its instructions see http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf.