

# ECSE 444: Microprocessors

## Lab 1: Assembly, C, and Optimization

### Abstract

In this lab you will (a) learn how to time the execution of your code using the Cortex-M4 debugging interface, (b) learn how to integrate C and assembly source files in a single project, and (c) compare assembly, C, and CMSIS-DSP implementations of common signal processing functions.

### Deliverables for demonstration (~~50~~100%)

- Demo no longer than 10 minutes
- Discussion of each implemented algorithm
- Discussion of latency measurements for each implemented algorithm

### ~~Deliverable Report (50%)~~

- ~~• Report no longer than one (1) page (10-pt font, 1" margins, *no cover page*)~~
- ~~• Explanation of timing analysis and results~~

### Grading Summary for Demo and Report

- Square root calculation
  - 10% FPU
  - 10% CMSIS-DSP
  - 10% C
  - 10% Timing analysis
- Solving transcendental functions
  - 20% C
  - 20% Assembly
  - 20% Timing analysis of C and assembly implementations

### Changelog

- 13-Sep-2023 Deliverables and grading adjusted: no report will be submitted for Lab 1.
- 07-Jul-2023 Fall 2023 revision.
- 15-Sep-2020 Note about HardFault\_Handler() added in Putting it All Together.
- 22-Jun-2020 Initial revision.

### Overview

In this lab, we'll learn how to (a) integrate C and ARM assembly, (b) make use of highly optimized CMSIS library functions, and (c) measure the execution latency of our code. At the end of this lab, you'll be able to write and call C and assembly functions, call CMSIS-DSP functions, modify the STM32CubeIDE environment so it all compiles correctly, and profile your

code using the Instrumentation Trace Microcell (ITM). As a result, you'll also experience first hand the relative performance benefits of writing C or assembly, and using libraries that have been carefully tuned for the Cortex-M4.

## Resources

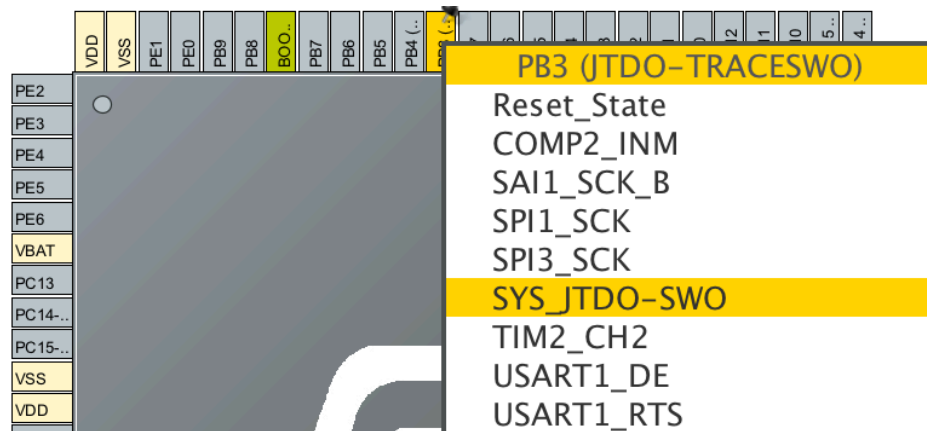
[ARM® and Thumb®-2 Instruction Set Quick Reference Card](#)  
[Vector Floating Point Instruction Set Quick Reference Card](#)

## Configuring the Board

For this lab, we'll be utilizing the debugging interface to time the execution of our code. In this case, we need to enable the pins used by the Single Wire Output (SWO) interface. Start a new project, and clear the pinout. Before doing anything else, check that the system clock (HCLK) is 80 MHz (or 120 MHz, depending on your board; refer to Lab 0 for more information).

We'll configure one different pin as follows:

- Set PB3 to SYS\_JTDO-SWO



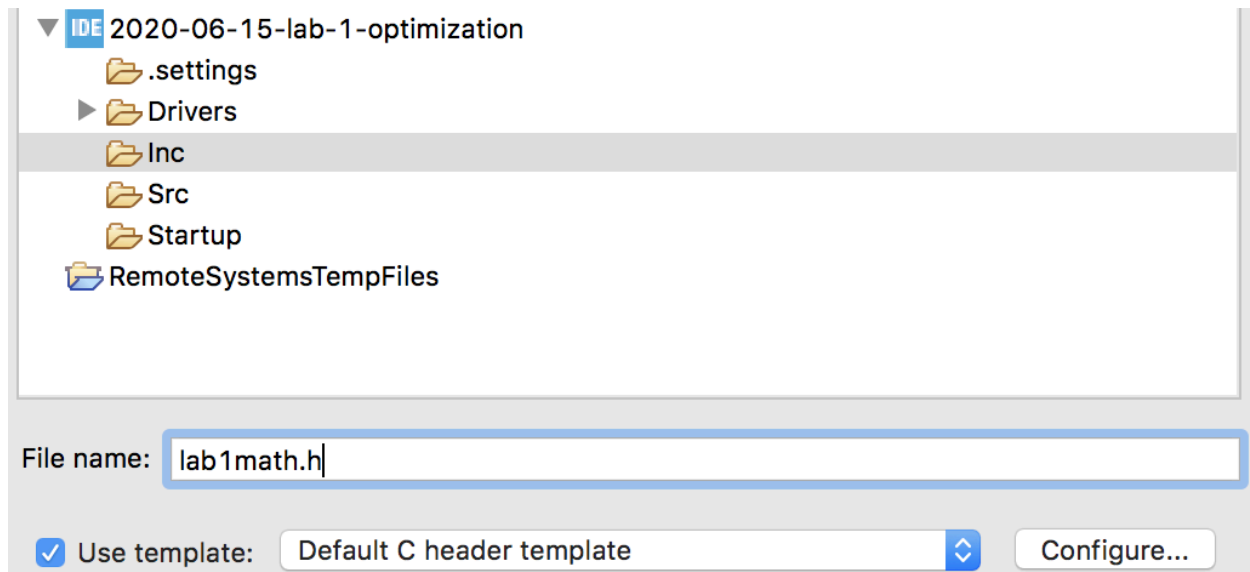
## Finding the Maximum Value of an Array in C

In this lab, we are comparing the performance of assembly, C, and library implementations of various mathematical functions. We'll start with finding the maximum value in an array. To get started, we need to create new header and source files for our code.

We'll define our new functions, both those written in C and in assembly, in a new header file, `lab1math.h`. Include that in `USER CODE BEGIN Includes`.

```
24 /* Private includes -----*/
25 /* USER CODE BEGIN Includes */
26 #include "lab1math.h"
27 /* USER CODE END Includes */
```

Now we need to create that file. In *Project Explorer*, right-click on the Inc directory, and choose *New > File from Template*. Once you name the file something.h, IDE will automatically select the appropriate template: *Default C header template*.



Click *Finish* to create the file.

Repeat this process in the Src directory to create `cmax.c`; this is where we'll write our C implementation of the max function.

We'll write the header file first, and then complete the implementation. Our first function finds the maximum value, and associated index, in an array of floating point values. Here's our function prototype:

```
void cMax(float *array, uint32_t size, float *max, uint32_t *maxIndex);
```

We pass a pointer to the array, and its size; we also pass a pointer to a variable to hold the max value, and another to hold the index of the max value. We're writing procedural C in this course (as opposed to C++); that means that when we don't want to use a struct, we pass arguments by reference, and modify them in functions. Once the function returns, we have the return values in the variables we passed as arguments (though our functions may often return void).

Note that we've used `uint32_t` instead of `int` above. Data types matter in embedded system programming; compilers can often improve performance by using data types that require fewer

than 32 bits. Being intentional about using signed and unsigned data types can also simplify things for the compiler. Such data types, however, are not part of the C standard. In order to be able to use them in header and C files without warnings and errors, be sure to `#include "main.h"` (e.g., in `cMax.c`).

If you haven't written C in a while, it is worthwhile refreshing yourself with this function; a working implementation can be found below. To facilitate incremental testing, we'll write the function call in `main.c` first, and then the implementation in `cmax.c`.

Starting at `USER CODE BEGIN 2`, we'll set up the variables needed to test the function. Remember: if you put your code in the marked `USER CODE` sections, it will be preserved if you have to regenerate your code skeleton.

```
/* USER CODE BEGIN 2 */
// define our variables and array
float max = 0;
uint32_t maxIndex;
// the max is 88.49 at index 5
float array[10] = {48.21, 79.48, 24.27, 28.82, 78.24, 88.49, 31.19, 5.52,
82.70, 77.73};
/* USER CODE END 2 */
```

Now write the function call; for lack of a better place, put it in `USER CODE BEGIN 3`. This is in the infinite while loop; the function will be called over and over again forever.

```
/* USER CODE BEGIN 3 */
    cMax(&array, 10, &max, &maxIndex);
}
/* USER CODE END 3 */
```

Since we allocated `max`, `maxIndex`, and `array` directly (on the stack, incidentally; more about this later in the semester), rather than pointers to them (resulting in allocation on the heap, incidentally), we pass their addresses to the function. This will result in a variety of warnings, since the prototype expects pointers, but doesn't affect correctness.

Now it's time to implement `cMax`. Remember: since we want to pass values back in `max` and `maxIndex`, we want to change the values pointed to, not the pointer itself (which is a memory address). We have to dereference the variables to access and change their values. I.e., we may start our function by setting `max` and `maxIndex` to correspond to first element of the array:

```
(*max) = array[0];
(*maxIndex) = 0;
```

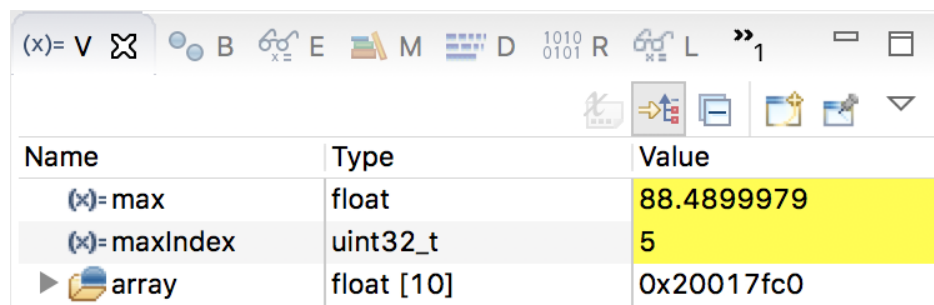
When you're confident in your implementation, or want to test something, build it, set a breakpoint at `while (1)` by double-clicking to the left of the line number, and start the debugger.

```

100  while (1)
101  {
102      /* USER CODE END WHILE */
103
104      /* USER CODE BEGIN 3 */
105      cMax(&array, 10, &max, &maxIndex);
106  }
107      /* USER CODE END 3 */

```

Resume execution once, and the debugger should advance to the `while` loop. Resume once more, and we can check if we got the right answer. It should be open by default, but if not, open the *Variables* view. From the *Window* pull-down menu, select *Show View > Variables*. This shows the values of all the variables in scope.



Name	Type	Value
(x)= max	float	88.4899979
(x)= maxIndex	uint32_t	5
array	float [10]	0x20017fc0

Inspection of the test array indicates that index 5 has the max value, 88.49; `max` has a different value than this simply because floating point number representations introduce error (more on this later in the semester).

Having problems? Here's a working implementation of `cMax`:

```

void cMax(float *array, uint32_t size, float *max, uint32_t *maxIndex) {
    (*max) = array[0];
    (*maxIndex) = 0;

    for (uint32_t i = 1; i < size; i++) {
        if (array[i] > (*max)) {
            (*max) = array[i];
            (*maxIndex) = i;
        } // if
    } // for
} // cMax

```

## Timing Code Execution using the Instrumentation Trace Microcell (ITM)

Though STM32CubeIDE doesn't come with a simulator (and so if we want to run our code, we need a hardware platform to deploy to), the debugger and related functionality is incredibly powerful when it comes to monitoring software for correct functionality and measuring performance. Before we look at assembly and library implementations of our max function, we will add a bit of code to time the execution of our C implementation using the built-in debugging hardware that comes with the Cortex-M4. Specifically, we will use the Instrumentation Trace Microcell (ITM), which is designed to add timestamps to trace events. If we create a timestamp before and after cMax, we have an approximation of how long cMax requires to execute.

Since we've already enabled SWO, we can profile the execution of cMax with the addition of just three lines of code, and a couple of other small changes to the configuration of the debugger. The ITM has memory address space reserved; writing to these addresses prompts a debugging event, which can be displayed in IDE. To get started, we'll define a macro for writing to this address space. Remember to define this macro in a USER\_CODE region.


```
/* Private define
-----*/
/* USER CODE BEGIN PD */
#define ITM_Port32(n) (*((volatile unsigned long *) (0xE0000000+4*n)))
/* USER CODE END PD */
```

ITM\_Port32(n) is a location in memory; setting it to a value will generate a trace packet with that value as the data. This also generates a timestamp in terms of elapsed cycle count and wall-clock time. Note: the debugging interface has limited bandwidth, so we need to take measures to intentionally spread out when trace packets are generated, e.g., by putting our relatively short function inside a loop that repeats it a number of times. (A loop isn't necessary if you single step through the ITM accesses and function calls.) Change your code as follows, adding ITM\_Port32(31) memory accesses before and after the calls to cMax.

```
/* USER CODE BEGIN 3 */
    ITM_Port32(31) = 1;
    for (uint32_t i=0; i<1000; i++)
        cMax(&array, 10, &max, &maxIndex);
    ITM_Port32(31) = 2;
}
/* USER CODE END 3 */
```

This sends data values 1 and 2 to ITM port 31; the timestamp for value 1 indicates the *approximate* start time, and for value 2, the *approximate* end time, of the execution of 1000 calls to cMax. Why are the times approximate? Because it takes time to write to the ITM port, and using a for loop to time 1000 iterations of cMax introduces overhead.



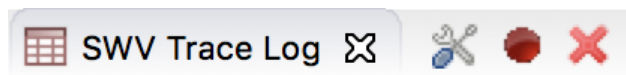
Build , and start the debugger . The first time you do so, you'll have the option to configure the debugging environment. Open the Debugger tab.

Under *Serial Wire Viewer (SWV)*, tick the “Enable” box, and set the Core Clock to 80.0 MHz (120.0 MHz). Apply the changes and click OK. This will deploy and start your code; like usual, it will automatically stop at the first line of main.

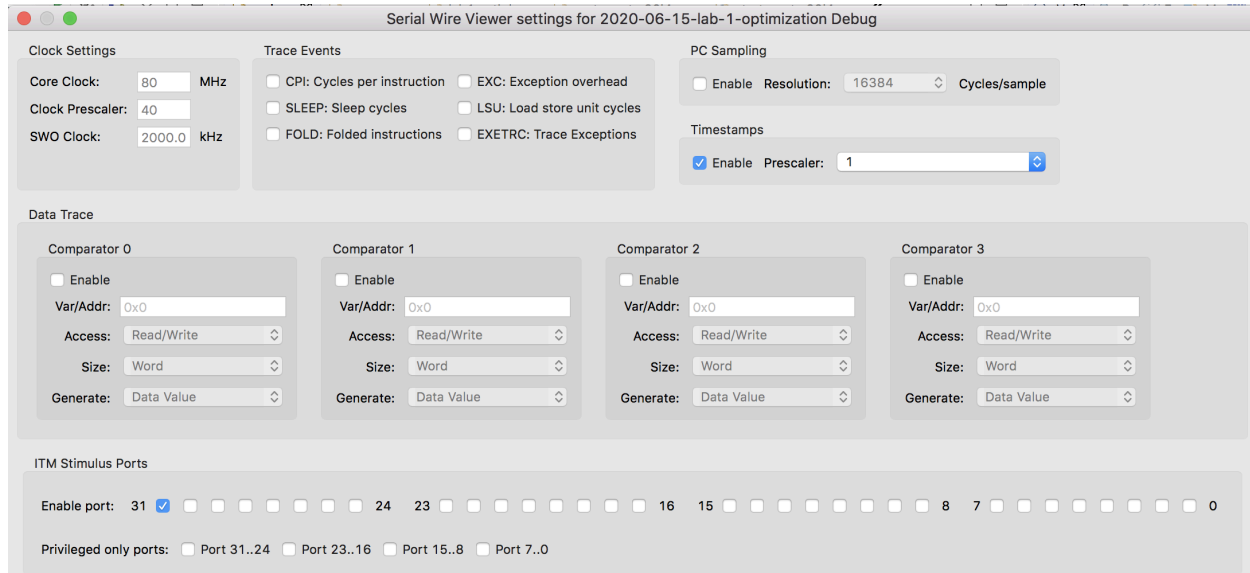
In order to see the ITM trace packets, we need to open and configure another view in IDE, and then turn on trace recording. From the *Window* pull-down menu, select *Show View > SWV > SWV Trace Log*. This opens a window that will display all of the trace packets as they arrive. In the future, we'll look at how other SWV views can be used to track how variables change with time, or capture calls to printf(\*).




Next, click the wrench to configure the SWV Trace Log.



In our code, we write to ITM port 31; we therefore need to enable the tracing of port 31. Verify that timestamps are enabled; nothing else needs to be changed.



Finally, click the record button , and resume execution. If you still have a breakpoint at while (1), execution will stop there; resume once more, and cMax will run once. Two packets should now be displayed in the Trace Log.

2	ITM Port 31	1	190546	2.381825 ms
3	ITM Port 31	2	503779	6.297238 ms

In the above case, we observe that the loop takes 313233 cycles, or about 313 cycles per call (Your numbers may be slightly different; mine differ from test to test.) Note: this includes loop overhead. It is possible to time single function calls, provided you are single-stepping through your code; too many accesses to the ITM, however, will overwhelm it, and timestamps will not reliably be generated.

*There are other ways to measure performance. If you found doing something else was easier, showcase it in your demo, and let me know.*

## Finding the Maximum Value of an Array in Assembly

Now it's time to implement an alternative: finding the maximum value of the array using a function in assembly. Add a new file to the Src directory, `asmmax.s`. The implementation, with comments, is given on the following page. Add a new function prototype to `lab1math.h`, too.

```
extern void asmMax(float *array, uint32_t size, float *max, uint32_t
*maxIndex);
```



The extern keyword indicates that, though defined here, it is implemented elsewhere.

Note that IDE uses GNU assembly syntax. E.g., “;” is not recognized as starting a comment; instead, use // as you would in C. Assembler directives are also formatted differently. For an example with side by side comparison, refer to [this](#).

Once your assembly implementation is written, add a loop in which to call it, and additional writes to the ITM to log timestamps. You should observe that the assembly implementation is considerably faster than unoptimized C, at ~157 cycles, or 50% less latency.

```

/*
 * asmmax.s
 */

// unified indicates that we're using a mix of different ARM instructions,
// e.g., 16-bit Thumb and 32-bit ARM instructions may be present (and are)
.syntax unified

// .global exports the label asmMax, which is expected by lab1math.h
.global asmMax

// .section marks a new section in assembly. .text identifies it as source code;
// .rodata marks it as read-only, setting it to go in FLASH, not SRAM
.section .text.rodata

/**
 * void asmMax(float *array, uint32_t size, float *max, uint32_t *maxIndex);
 *
 * R0 = pointer to array
 * R1 = size
 * R2 = pointer to max
 * R3 = pointer to maxIndex
 */

asmMax:
    PUSH        {R4, R5}           // saving R4 and R5 according to calling convention
    VLDR.f32    S0, [R0]           // max = array[0] (fp register S0 is used for max)
    MOV         R4, #0              // maxIndex = 0 (register R4 is used for maxIndex)

loop:
    SUBS        R1, R1, #1          // size = size - 1
    BLT         done               // loop finishes when R1 < 0
    ADD         R5, R0, R1, LSL #2  // calculate base address (in R5) for array element
    VLDR.f32    S1, [R5]           // load element into fp register S1 (from address in R5)
    VCMPI.f32    S0, S1             // compare new element with current max
    VMRS        APSR_nzvc, FPSCR   // load the FP PSR to branch using FP conditions
    BGT         continue           // if max > new element, on to the next element
    VMOV.f32     S0, S1             // otherwise, max = new element
    MOV         R4, R1             // update maxIndex

continue:
    B           loop               // next iteration

done:
    VSTR.f32     S0, [R2]           // store max value in the pointer to max variable given
    STR         R4, [R3]           // store max index in the pointer to maxIndex given
    POP         {R4, R5}           // restore context
    BX LR                         // return

```

## Finding the Maximum Value of an Array Using CMSIS-DSP

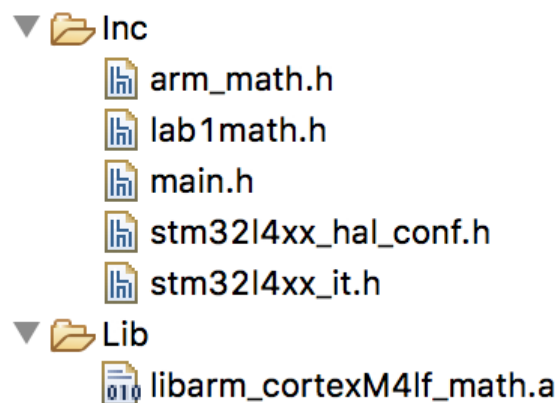
For our last alternative implementation, we'll use functions from an extensive ARM library, CMSIS-DSP. This library has been specifically optimized for our processor, and takes full advantage of its features. In this case, `arm_max_f32()` is a perfect match; it even has the exact same list of arguments. Add the call, and another loop and ITM access, to your code.

Since we're making a call to a function in a library, we need to do the following in order to ensure the appropriate instructions are included in our executable.

1. Copy header and library files into our project.
2. Include the appropriate header file in `main.c`.
3. Direct IDE to link using the appropriate library file.

All the files required to use CMSIS-DSP are included in the installation for STM32CubeIDE; in OS X, these are in `STM32Cube/` by default. For Windows, go to your user directory (`C:\Users\user_name`), there you should find the `STM32CubeIDE` directory containing the Repository. Go to this directory and,

1. Find `arm_math.h` in `Repository/STM32Cube_FW_L4_V1.17.2/Drivers/CMSIS/DSP/` Include. *Note that the `FW_L4_VX.XX.X` part of this directory name will change as firmware versions change.* Copy `arm_math.h` into your project `Inc/` directory.
2. Find `libarm_cortexM4lf_math.a` in `Repository/STM32Cube_FW_L4_V1.17.2/Drivers/CMSIS/DSP/Lib/GCC`. Copy it into a new directory in your project, `Lib/`.



Next, we need to include `arm_math.h` in our source code. `arm_math.h` is a complex header file that works differently for different Cortex processors; add the `#define` statement below to indicate our processor is a Cortex-M4.

```
/* USER CODE BEGIN Includes */
#define ARM_MATH_CM4
#include "arm_math.h"

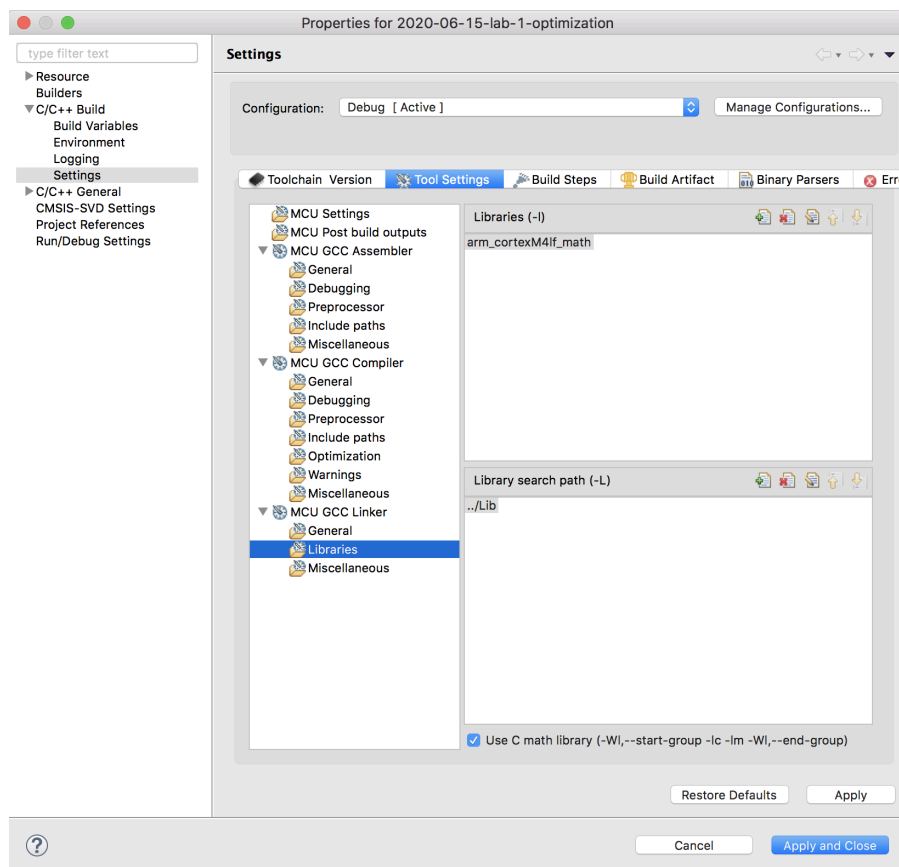
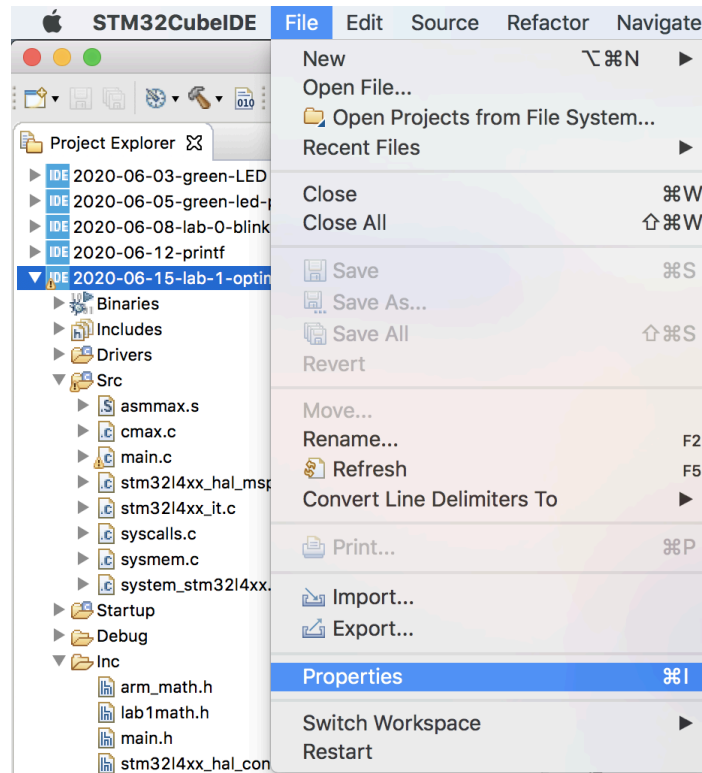
#include "lab1math.h"
/* USER CODE END Includes */
```

Now all we need to do is to direct IDE to link using the library. This requires that we change two parameters passed to GCC: libraries to look for, and where to look for them. Select the project in Project Explorer, and then under the *File* pulldown menu, select *Properties*. Then, under *C/C++ Build*, select *Settings*. Next select *Tool Settings*, and under *MCU GCC Linker*, select *Libraries*.

Now we need to add the library (akin to passing the `-l` argument to GCC). Click the + on the *Libraries (-l)* part of the configuration window, and enter `arm_cortexM4lf_math`. Note: the `lib` prefix and `.a` extension should be removed from the file name when adding libraries.

Next, we need to add our new Lib directory to the set of directories that are searched for libraries (akin to passing the `-L` argument to GCC). Click on the + on the *Library search path (-L)* part of the configuration window, and either enter `../Core/Lib`, or navigate to and select this directory using the Workspace or File system interfaces.

*Apply and Close*, and *Build*! Note: setting paths is one of the most frustrating parts of working with IDEs and C, so if your compilation suddenly fails due to errors, check the spelling of the `#define` statement, library, library location, etc.



You should observe that the heavily optimized `arm_max_f32()` routine is even faster than our hand-coded assembly, at ~127 cycles, a nearly 60% reduction in latency over the unoptimized C implementation.

*Additional exercise: what happens to the timing of the three implementations if you allow GCC to optimize your code? Note that debugging compiler-optimized code is much more challenging, as optimization passes often manipulate what code executes when, though the `-Og` option strikes an interesting balance. Also note that `-Og` performs dead-code elimination: if a variable is not used after being updated, the calculation performing the update may be removed from the executable.*

## Putting it All Together

Now that you know how to mix C, assembly, and library calls, complete source code that implements the following functionality in each, and measure the differences in execution latency.

*Note:* if during debugging your code halts in `HardFault_Handler()`, this simply means an unrecoverable error, e.g., something like a segmentation fault or stack overflow, has occurred. Slowly step through your code to find where the fault is occurring. Check the arguments you're passing to functions.

## Square Root

Our Cortex-M4 processor has a floating point unit, and can natively perform square root. How much faster is that than the various other options available to us? Implement and compare the performance of the following approaches to calculating square root of  $x$ :

- [Cortex-M4 FPU](#)
- [CMSIS-DSP](#)
- [Newton-Raphson Method](#) in C

$x$  must be a parameter to whatever functions you write to implement square root.

*Note:* it is recommended that you begin with the CMSIS-DSP implementation, and for ease of implementation and testing, employ the same function prototype for your C and assembly implementations. This will give you a ground-truth result for debugging purposes.

Measure the latency of each. How much faster is the FPU?

### Transcendental Functions

While it is relatively easy to express polynomials using the basic mathematical operations available in a CPU/FPU, some functions are more difficult to work with, e.g.,  $\sin(x)$ . The CMSIS-DSP library includes implementations of [trigonometric functions](#) which use a combination of lookup tables and interpolation. Implement and compare the performance of following approaches to finding a value of  $x$  such that  $x^2 = \cosine(\omega x + \phi)$ :

- [Newton-Raphson Method](#) in C
- [Newton-Raphson Method](#) in ARM assembly

$x$ ,  $\omega$ , and  $\phi$  must all be parameters to whatever functions you write to implement your solution.

You may use the CMSIS-DSP trig functions in each of the above implementations. It is recommended that you begin with the C implementation.

Measure the latency of each, and experiment with optimization settings. What do you observe?

### Deliverables

Your demo is limited to 10 minutes. It is useful to highlight that your software computes correct partial and final answers. Draw our attention to variable values to demonstrate that your software operates as expected.

Your demo will be graded by assessing, for each algorithm, the correctness of the observed behavior, and the correctness of your description of that behavior. *Be sure to highlight your latency measurements, both how you took them, and their values.*

~~In your report, for each set of algorithms describe:~~

- ~~• Your approach, including basic software architecture (function prototypes, software parameters, toolchain parameters, etc)~~
- ~~• Your testing methodology, including corner cases~~
- ~~• The results of your timing analysis, and, to the extent possible, an explanation for whatever differences are observed~~

~~Your report is limited to one page. It will be graded by assessing, for each set of algorithms, your report's clarity, organization, and technical content.~~

## Grading

~~Your demo and report are equally weighted.~~ The breakdown for each is as follows:

- Square root calculation
  - 10% FPU
  - 10% CMSIS-DSP
  - 10% C
  - 10% Timing analysis
- Solving transcendental functions
  - 20% C
  - 20% Assembly
  - 20% Timing analysis of C and assembly implementations

Each part of the demo will be graded for (a) clarity, (b) technical content, and (c) correctness:

- 1pt *clarity*: the demo is clear and easy to follow
- 1pt *technical content*: correct terms are used to describe your software
- 2pt *correctness, part 1*: given an input, the correct output is clearly demonstrated
- 1pt *correctness, part 2*: timing analysis is demonstrated and explained

~~Each part of the report will be graded for: (a) clarity, (b) organization, and (c) technical content:~~

- ~~• 1pt *clarity*: grammar, syntax, word choice~~
- ~~• 1pt *organization*: clear narrative flow from problem description, approach, testing, challenges, etc.~~
- ~~• 3pt *technical content*: appropriate use of terms, description of proposed approach, description of testing and results, etc.~~

## Submission

Please submit, on MyCourses, your:

- Source code used to demo (only files you modified, including IOC file).
- ~~• Source code used for your report (only files you modified).~~
- ~~• Report~~