Lab 1: CMSIS-DSP Functions for Optimized Code

ECSE 444: Microprocessors

Winter 2020

Introduction

In this lab you will be introduced to the CMSIS-DSP software library, which is a collection of common signal processing functions that ARM has optimized for the various Cortex-M processor cores.

In particular, you will:

- Write two simple tasks in C code

- Re-write the tasks using the CMSIS-DSP library functions.

- Learn how to time the code to evaluate the speed-up

- Write the tasks in assembly to evaluate the speed-up (or lack thereof) of hand written assembly v/s C.

Changelog

- 4-Sep-2019   Initial release.

- 9-Sep-2019   Fixed variance formula.

Grading

- 25 % C code for array multiplication

- 10 % CMSIS-DSP implementation of array multiplication

- 5 % Timing and analysis of both implementations

- 25 % C code for standard deviation

- 10 % CMSIS-DSP implementation of standard deviation

- 5 % Timing and analysis of both implementations

- 10 % Assembly code for array multiplication

- 10 % Assembly code for standard deviation

Note: *Please remember that your assembly code must run faster than your C code!*

Background

This section will introduce you to all the concepts you need to succeed in this lab.

As an example, we will provide the code to find the maximum value in an array of floating point elements. You are provided with a base-project **GXX_Lab1** which contains the files **main.c**, **asm_math.s** and **asm_math.h**.

IMPORTANT: If you are using the lab computers, please ensure that you save the project in C:\Users\<your_name_goes_here> otherwise it will not work in Keil.

```c
1   #include <stdio.h>
2   #include <arm_math.h>
3
4   #include <asm_max.h>
5
6   float fl0_array[10] = {48.21, 79.48, 24.27, 28.82, 78.24, 88.49, 31.19, 5.52, 82.70, 77.73};
7
8   int main() {
9
10      float max;
11      int max_idx;
12
13      float a_max;
14      int a_max_idx;
15
16      float DSP_max;
17      uint32_t DSP_max_idx;
18
19      int i;
20
21      /* LOOP TO FIND MAX*/
22      max = fl0_array[0];
23      max_idx = 0;
24      for(i=0 ; i<10 ; i++) {
25          if(fl0_array[i] > max) {
26              max = fl0_array[i];
27              max_idx = i;
28          }
29      }
30
31      /* Assembly max function */
32      asm_max(fl0_array, 10, &a_max, &a_max_idx);
33
34      /* CMSIS-DSP max function */
35      arm_max_f32(fl0_array, 10, &DSP_max, &DSP_max_idx);
36
37      printf("Pure C   : Max element f[%d] = %f\n", max_idx, max);
38      printf("Assembly : Max element f[%d] = %f\n", a_max_idx, a_max);
39      printf("C: DSP   : Max element f[%d] = %f\n", DSP_max_idx, DSP_max);
40      return 0;
41  }
42
```

Figure 1: **main.c**

```c
1  #ifndef _ASM_MAX
2  #define _ASM_MAX
3
4      void asm_max(float *f_arr, int N, float *max, int *max_idx);
5
6  #endif
7
```

Figure 2: **asm_max.h**
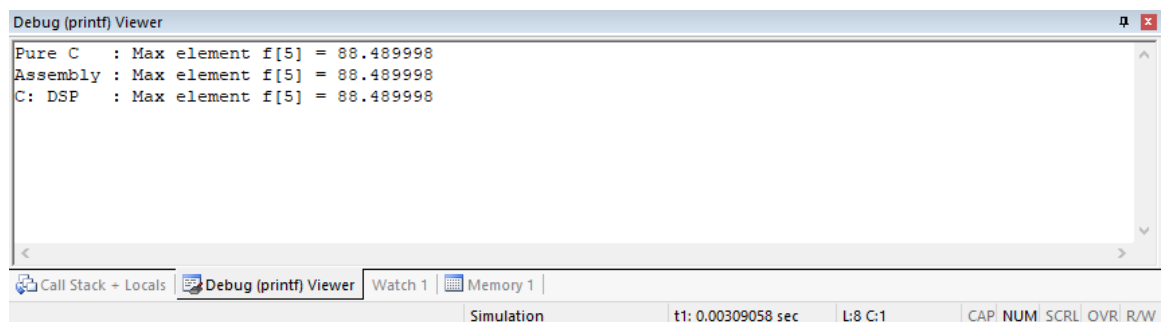
```
1       AREA test, CODE, READONLY
2
3       export asm_max              ; label must be exported if it is to be used as a function in C
4   asm_max
5
6       PUSH{R4, R5}                ; saving context according to calling convention
7
8       VLDR.f32 S0, [R0]           ; max = f[0] (register S0 used for max)
9       MOV R4, #0                  ; max_index = 0 (register R4 used for max_idx)
10  loop
11      SUBS R1, R1, #1             ; loop counter (N = N-1)
12      BLT done                   ; loop has finished?
13      ADD R5, R0, R1, LSL #2      ; creating base address for the next element in the array
14      VLDR.f32 S1, [R5]           ; load next element into S1
15      VCMP.f32 S0, S1            ; compare element with current max
16      VMRS APSR_nzcv, FPSCR      ; need to move the FPU status register to achive floating point conditional execution
17      BGT continue               ; if the current max is greater or equal to new element, no need to update, continue
18      VMOV.f32 S0, S1            ; if not, then update current max
19      MOV R4, R1                  ; also update max index
20
21  continue
22      B loop                     ; loop
23
24  done
25      VSTR.f32 S0, [R2]          ; store the max value in the pointer (float *max) that was provided
26      STR R4, [R3]               ; store the index of the max value in the pointer (int *max_idx) that was provided
27
28      POP{R4, R5}                ; restore context
29      BX LR                      ; return
30
31      END
32  |
```

Figure 3: **asm_max.s**

Assembly code to find the maximum value in the array can be found in **asm_math.s**, while the C implementation as well as the CMSIS-DSP library implementation (using the arm_max_f32() function) is found in **main.c**. Take some time to study the code.

Figure 4 illustrates the expected output upon entering *Debug* mode and executing the code. We will now shift our attention to timing the code.

```
Debug (printf) Viewer                                                                    ⏷ ☒
Pure C    : Max element f[5] = 88.489998
Assembly : Max element f[5] = 88.489998
C: DSP    : Max element f[5] = 88.489998



 <

 Call Stack + Locals | Debug (printf) Viewer | Watch 1 | Memory 1
                              Simulation        t1: 0.00309058 sec    L:8 C:1      CAP NUM SCRL OVR R/W
```

Figure 4: Expected output of the *max* example provided.

Timing

We can time the code by placing breakpoints around the sections of code that need to be timed and using the *t1 stopwatch* provided by Keil. Figure 5 shows the breakpoints added around the three different implementations we wish to time, as well as the menu for the stopwatch *t1* and the option used to reset the time.
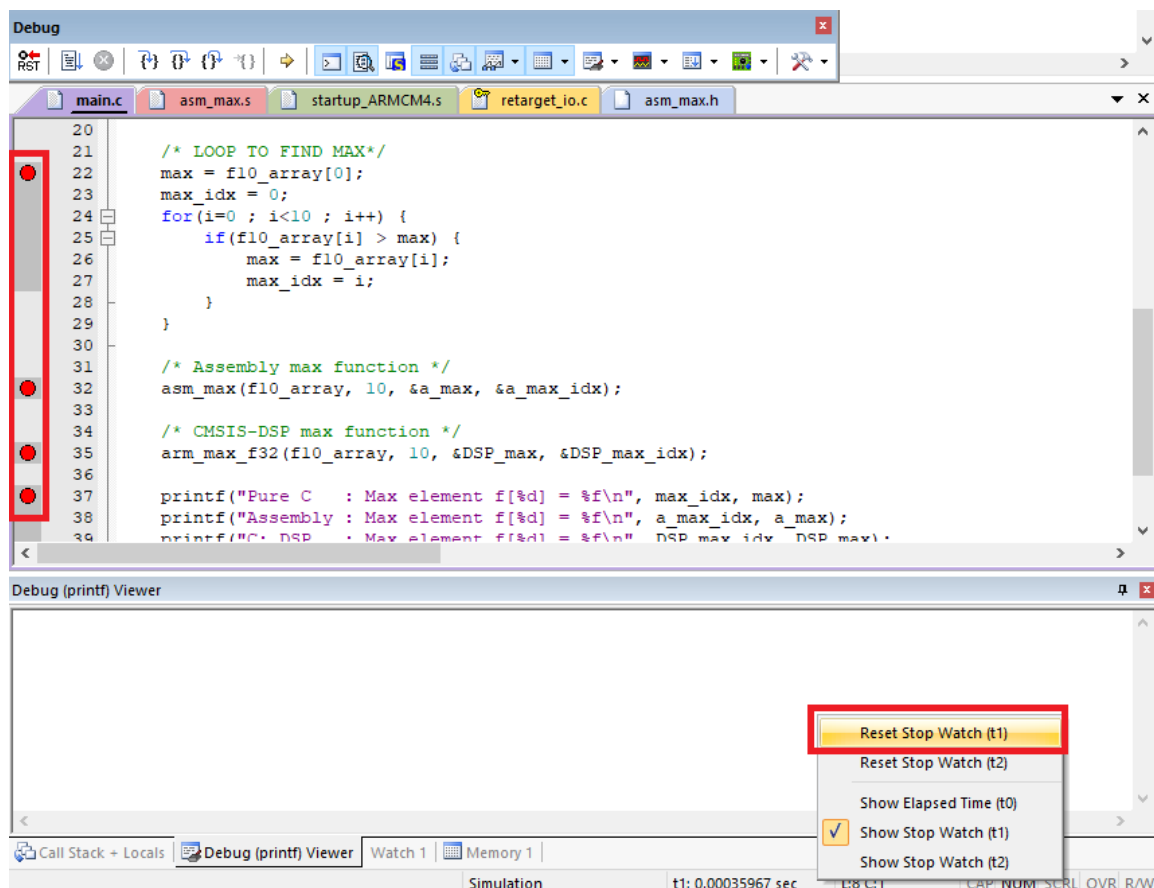

Figure 5: Adding breakpoints and the stopwatch menu.

1. We begin by executing the code until the first breakpoint and then resetting the stopwatch. We are now ready to time the C implementation.
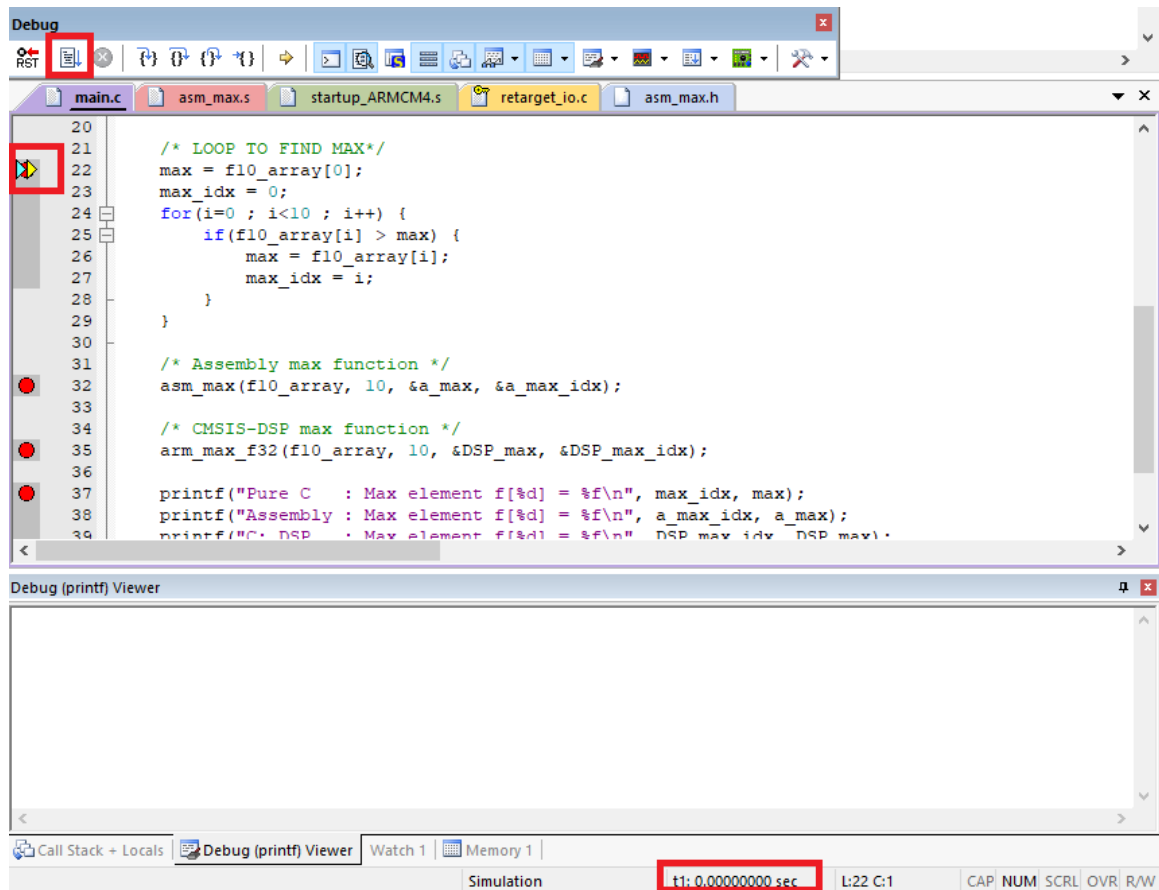
Figure 6: Run till first breakpoint and reset the stopwatch.

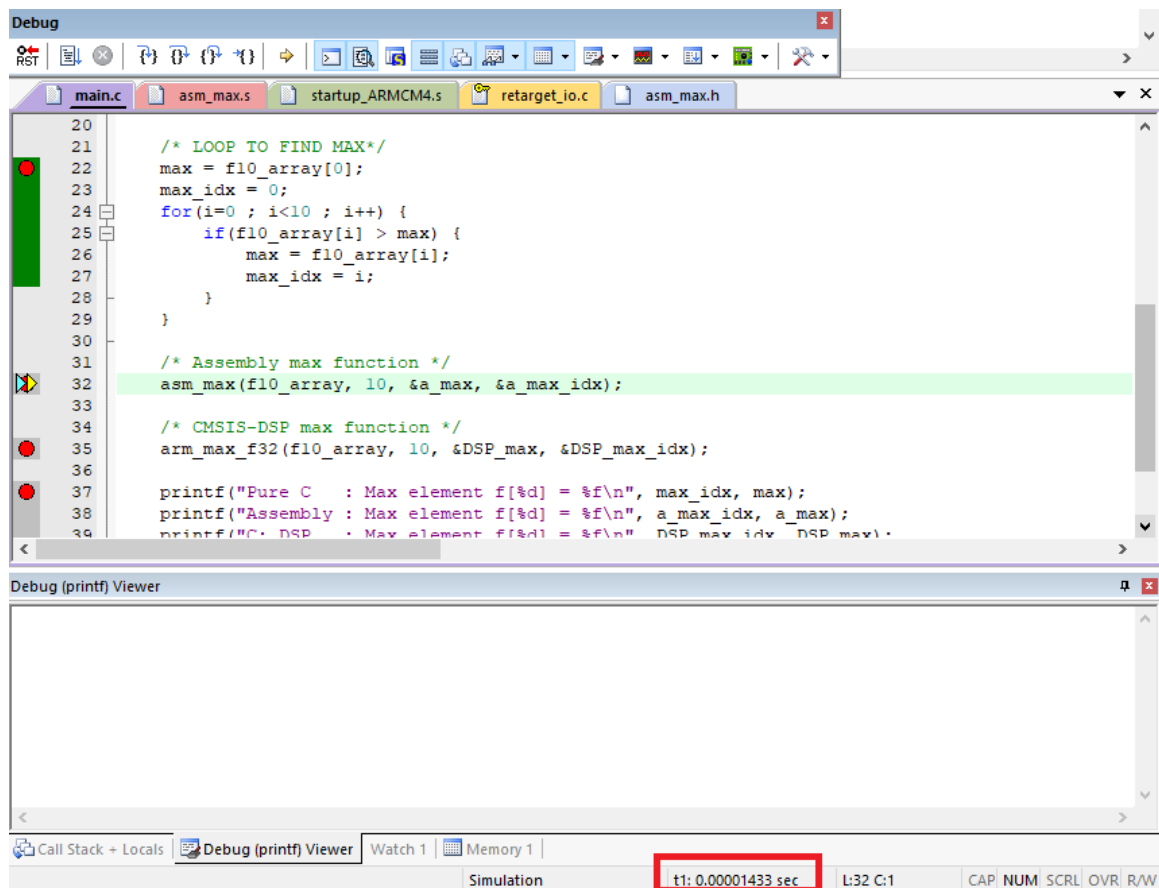2. We run the code till the second breakpoint and note down the time taken, which is 14.33 μs.

Figure 7: Run till second breakpoint and note down the time elapsed for the C implementation.

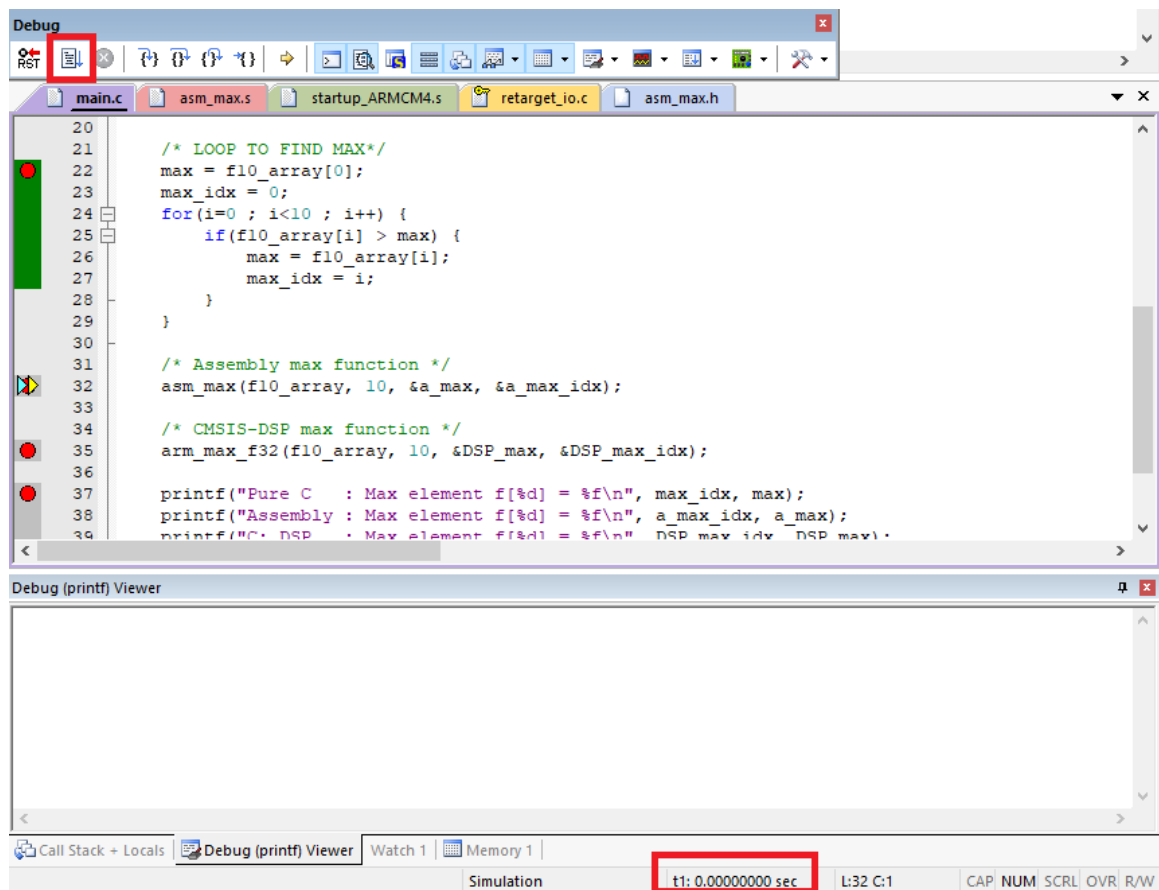3. We now reset the stopwatch in order to time the assembly implementation.

Figure 8: Reset the stopwatch.

4. We execute the code to the next breakpoint and note down the time taken, which is 13.17 μs. Therefore the assembly implementation is roughly 8.81 % faster than the C implementation.
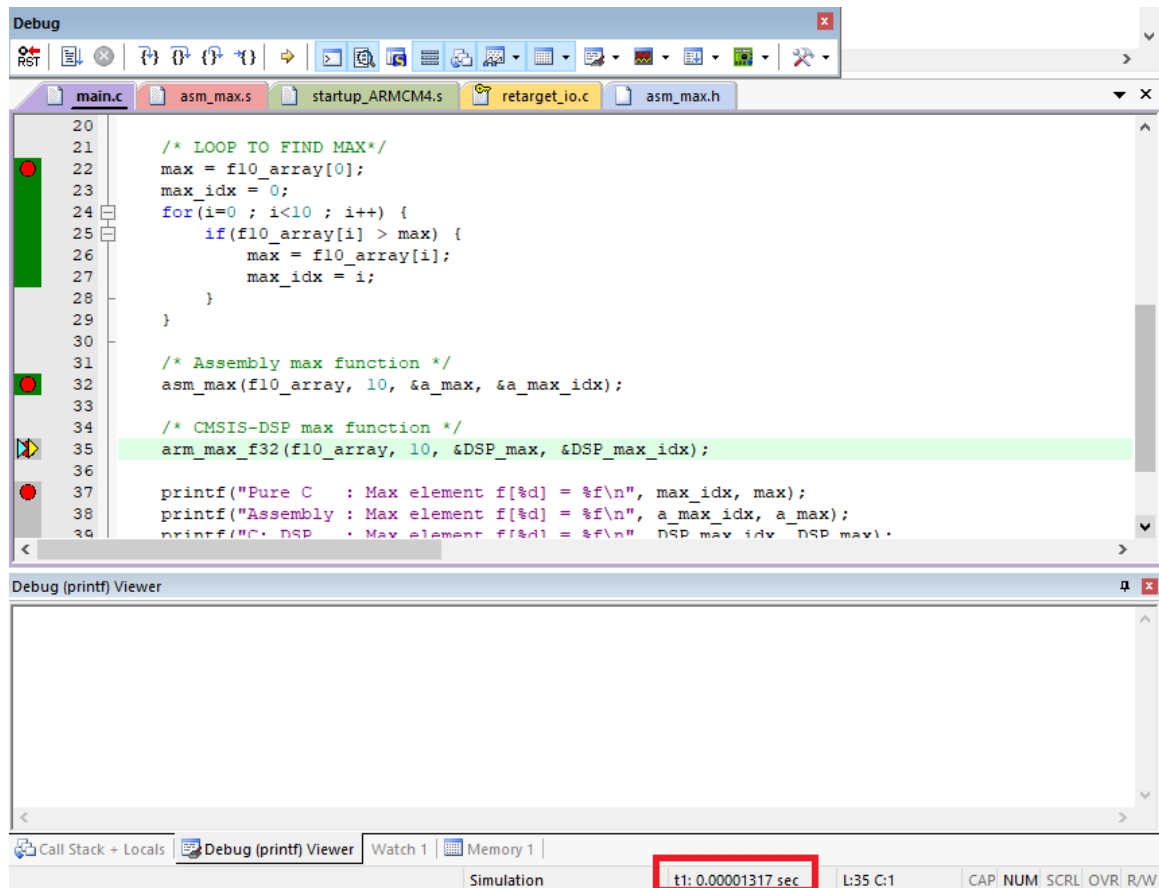

Figure 9: Run till third breakpoint and note down the time elapsed for the assembly implementation.

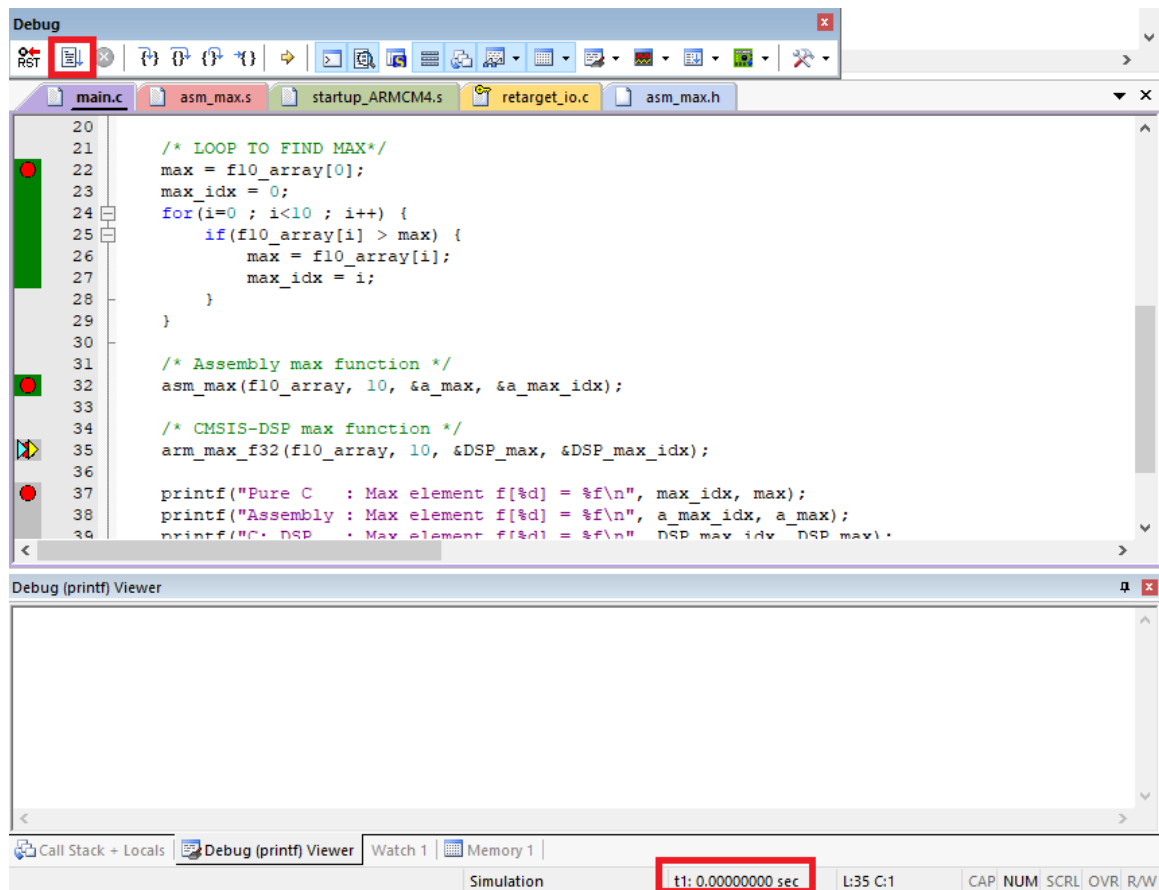5. We now reset the stopwatch in order to time the CMSIS-DSP implementation.

Figure 10: Reset the stopwatch.

6. We execute the code to the next breakpoint and note down the time taken, which is 9.92 μs. The CMSIS-DSP implementation is 44.5% and 32.8% faster than the C and assembly implementation respectively. This simple example clearly illustrates the advantage of using the optimized CMSIS-DSP library. **Ensure that you can repeat this exercise using the base-project and source code provided.**
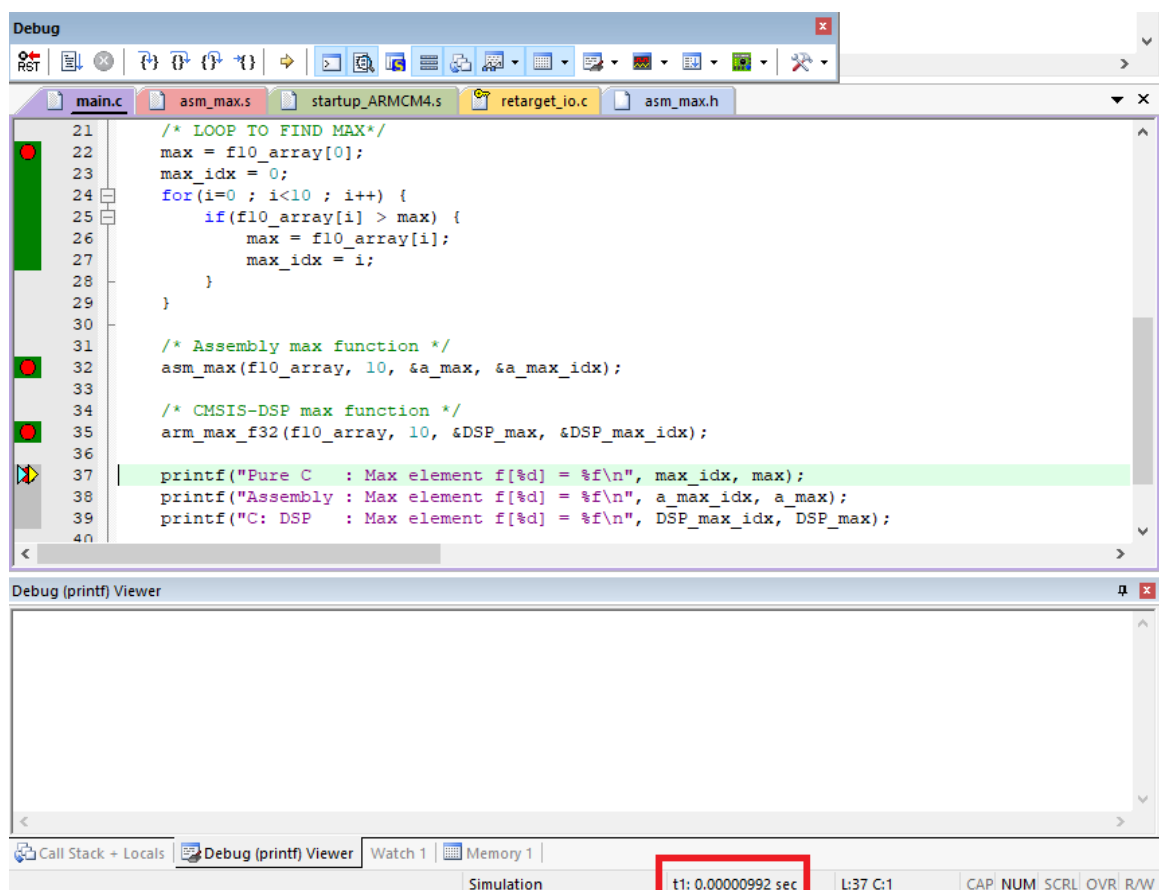

Figure 11: Run till final breakpoint and measure elapsed time for the CMSIS-DSP implementation.

Lab tasks

In this lab, you will complete two tasks based on the concepts introduced. You may use the same base-project provided to write code in, and refer to the CMSIS-DSP documentation to locate the library functions you may need.

Element-wise Array Multiplication

Given two vectors (arrays) A and B each of length N, the element-wise multiplication Y can be computed by:

$$Y(i) = A(i) \times B(i)$$

- Write plain C code to calculate the array multiplication. Then make use of the CMSIS-DSP library function (arm_mult_f32). Report the time taken by both implementations and the speed-up to your TA when presenting your code.

- Write an assembly implementation of array multiplication and measure the time taken.

Standard Deviation

Given a vector (array) A of length N, the standard deviation σ can be computed by:

$$\sigma = \sqrt{\frac{\sum\limits_{i=0}^{N-1}(A(i) - \mu)^2}{N - 1}}$$

where μ is the mean of vector A, computed by:

$$\mu = \frac{\sum\limits_{i=0}^{N-1} A(i)}{N}$$

- Write plain C code to calculate the standard deviation. Then make use of a CMSIS-DSP library function (arm_std_f32). Report the time taken by both implementations and the speed-up to your TA when presenting your code.

- Write an assembly implementation of the standard deviation and measure the time taken.

Assumptions

- All arrays and results are 32-bit floating point numbers.

- Use only the **f32** versions of CMSIS-DSP functions.

- Measure execution time for an array size of $N = 1000$ for both tasks.