Laboratory 1: Kalman Filter using Floating-Point Assembly Language Programming and its Evaluation with C and CMSIS-DSP

1st Karim Elgammal
Computer Engineering, McGill University
260920556

2nd Rafid Saif
Electrical Engineering, McGill University
260903802

***Abstract*—This investigation presents an exercise aimed at introducing the ARM Cortex and FP coprocessor assembly languages, instruction sets and their addressing modes. The laboratory also emphasizes the importance of respecting the ARM calling convention to enable integration with C programming language. The STM32CubeIDE, including the compiler and associated tools, is introduced through the lab.**

***Index Terms*—ARM Cortex, assembly language, floating-point coprocessor, instruction set, addressing modes, calling convention, STM32CubeIDE, C programming language, CMSIS-DSP API, microprocessor software interface, optimization.**

I. Introduction

This investigation is split into four distinctive parts; the first of which is concerned with writing working assembly language code for a single-variable Kalman filter. In the second part of the exercise, the assembly code developed in the first part is integrated into a larger program written in C using the Cortex Microprocessor Software Interface Standard (CMSIS-DSP) application programming interface (API). The CMSIS-DSP API incorporates a large set of routines optimized for different ARM Cortex processors. The third part examines an implementation of the same filter however, using plain C, without the CMSIS library. The exercise provides a practical introduction to the ARM assembly language and demonstrates the benefits of using the CMSIS-DSP API to optimize and accelerate the development of Cortex-based applications. It is noted that all plots are performed through Python by way of the Matplotlib library.

II. Methodology and Analysis

A. Assembly Code Kalman Filter Implementation

The first part of this investigation asks us to implement a Kalman filter in assembly code. The Kalman filter is a powerful tool for estimating physical processes. It's an adaptive filter that's well-suited for linear systems with Gaussian noise. What makes it unique is its ability to change its parameters based on the observed physical values and the current state of the system. This makes it more flexible than fixed linear filters. The Kalman filter achieves the best possible estimation error by updating its state in a series of discrete steps.

The assembly implementation within this section is based on the python implementation found in figure 1. More specifically, the implementation is of the update function which

```python
class KalmanFilter(object):
        q = 0.0 # process noise variance, i.e., E(w^2)
        r = 0.0 # measurement noise variance, i.e., E(v^2)
        x = 0.0 # value
        p = 0.0 # estimation error covariance
        k = 0.0 # kalman gain

        def __init__(self, q, r, p=0.0, k=0.0, initial_value=0.0):
                self.q = q
                self.r = r
                self.p = p
                self.x = initial_value

        def update(self, measurement):
                self.p = self.p + self.q
                self.k = self.p / (self.p + self.r)
                self.x = self.x + self.k * (measurement - self.x)
                self.p = (1 - self.k) * self.p

                return self.x
```

Fig. 1. Python Implementation of a Kalman Filter

is called on to the Kalman filter object instance and the measurement being taken. The initialization function init is written in C, along with a general KalmanFilter function which performs the overall steps of prediction and correction of the Kalman filter by calling the initialization function, and then calling the update function implemented in assembly.

```
6 // Take input array R0, outputarray R1, struct R2, length R3
7 kalman:
8     VPUSH {S1-S6}
9     MOV R1, R0 // Move address of struct into r1
10    VLDM R1, {S1-S5} // Load q,r,p,k,x into S1-S5
11    VADD.F32 S3, S3, S1 // p = p+q
12    VADD.F32 S6, S2, S3 // P+r and store in register
13    VDIV.F32 S4, S3, S6 // k = p/(p+r)
14    VSUB.F32 S6, S0, S5 // measurement - x
15    VMLA.F32 S5, S6, S4 // x = x + k*(measurement-x)
16    VMLS.F32 S3, S3, S4 // p = (1-k)*p
17    VMRS R0, FPSCR  // Read fpscr status
18    AND.S R0, R0, 0xF // AND with error bits
19    BNE exit // If error detected, don't update struct
20    VSTM R1, {S1-S5} // q,r,p,k,x changed in struct
21 exit:
22    VPOP {S1-S6}
23    BX LR
```

Fig. 2. Assembly Implementation of a Kalman Filter

The assembly implementation of the update function can be observed in figure 2. The implementation follows that of the python code presented in figure 1. It is noted that the program uses floating point registers to utilize the processor's FPU to conduct the calculations. This is done by enabling the FPU in the Cortex-M4 core by setting the CP10 and CP11 coprocessor access control registers. Then, we enable the FPU in the STM32L4S5 system control block by setting the CP10 and CP11 bits in the Floating-Point Context Control Register (FPCCR). This then allows us to use the floating-point instructions as seen in figure 2 with the instructions concatenated with a V. Furthermore, we optimized our implementation by using multiple store and load instructions such as in line 10 and 20 in figure 2. We also utilize the FPSCR to deal with the

corner cases within our calculations; this is done by fetching the flag bits and returning the specific flag being triggered by our update function.

Then, using the supplied measurments.h file, we ran the sample measurements of length 600 through our assembly implementation by using the 'extern' keyword to access the assembly file kalman.s. By doing this, we achieved an output array of the filtered measurements. This output was then plotted against the input measurements in order to arrive at a plot of both sets of data. The plot for the assembly implementation can be observed in figure 3. As expected, our filter causes a clear convergence towards the input stream; as the values of the output get closer and closer to the input stream, while accounting for the noise that the input stream may have experienced during measurement.

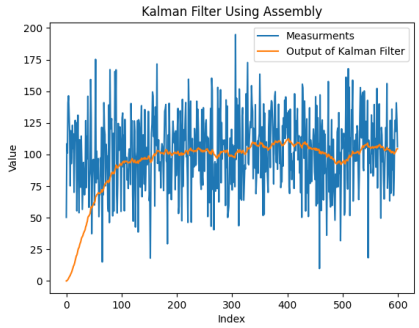

Fig. 3. Assembly Implementation Input/Output Stream Plot

Further, it is noted that our output stream deviates heavily from the input stream for up to 100 recursions of the filter, or 100 measurements. This is plausible as the filter adjusts its statistical factors to fit the fed data. As the filter receives more measurements, it alters the values of the estimation error covariance recursively until it converges to a value of 29.5041542, which seems to be the 'truest' estimation error covariance. The filter also continuously updates its Kalman gain property by each iteration of the recursion. This value starts from a default value of 0, and eventually converges to 0.0327823944. The filter then uses these values to estimate the expected measurement value that should be seen at the next estimation. As observed from our plot, our assembly implementation works as expected as the output stream converges to input stream, while taking measurement noise into consideration.

We then collected the output stream of our assembly implementation of the Kalman filter in order to analyze the difference between the input and output stream, standard deviation and average of that difference, as well as the correlation and convolution between the two streams. It is noted that these calculations were conducted using plain C code after obtaining the output by way of the assembly implementation. The difference between the two streams was then processed as the absolute value of these differences to arrive at the readable plot in figure 4. As seen from this plot, the difference in measurements starts from a very high range, and eventually converges to a band resembling the convergence to the input stream. The standard deviation and mean of this difference can be observed in table I.

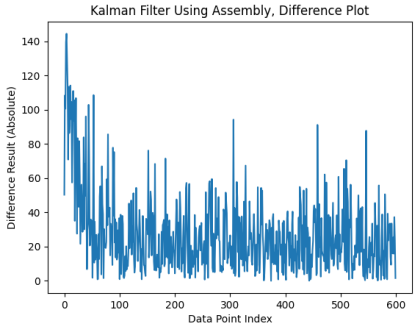

Fig. 4. Assembly Implementation Difference Plot

Next, we plotted the correlation and convolution of the two data sets. This plot can be observed in figure 5, where we see the plot exhibiting a smoothing of the input data stream. Additionally, it is understood that the correlation plot between the input and output data in a Kalman filter will typically show a peak at time zero (i.e., the correlation between the input and output at the same time step), and low or negligible correlations at other time lags. We can examine this characteristic of the correlation plot in figure 6, where our time zero is that of index 600 (as our data set is 600 elements long). And so, it is concluded that the assembly implementation of this Kalman filter is an optimized functioning solution.

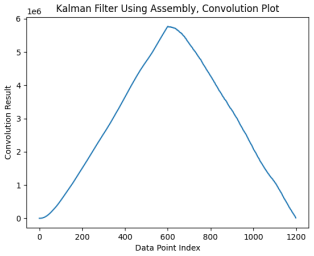

Fig. 5. Assembly Implementation Convolution Plot

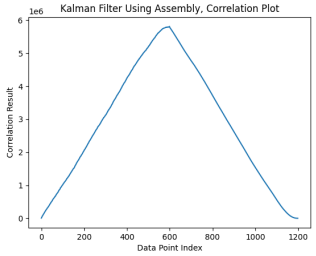

Fig. 6. Assembly Implementation Correlation Plot

B. CMSIS-DSP Kalman Filter Implementation

CMSIS-DSP is a library of optimized digital signal processing routines designed specifically for microcontroller applications, which provides a standardized API and various optimization techniques to efficiently implement DSP algorithms on Cortex-M based microcontrollers. We re-implemented the assembly code for the Kalman filter using the CMSIS-DSP library with the aims of cross validating our results from the assembly code as well as with our later discussed C implementation. The implementation follows the same structure of the code in figure 1, with discrepancies in function access name conventions that are described by the CMSIS library. We can

observe from the plot of the input and output data streams in figure 7, that our CMSIS-DSP solution results in a seemingly identical plot to that which we arrived at in the preceding section. This allows us to validate our assembly implementation using the derived standard deviation, difference and mean of the acquired output.

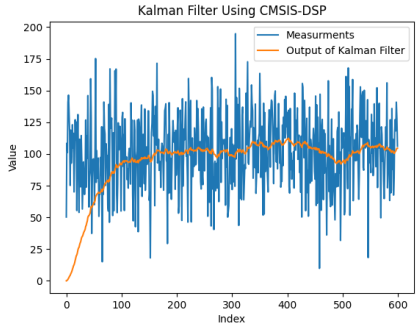

Fig. 7. CMSIS-DSP Implementation Input/Output Stream Plot

The library functions defined by CMSIS were then used in order to calculate the difference between the two data streams. This difference was then used along with the CMSIS defined functions to arrive at the standard deviation and mean of that difference. This difference is plotted in figure 8, while the values of the standard deviation and mean can be found in table I. As observed from the difference plot, our CMSIS calculation is seemingly identical to that of our difference plot arrived at through the assembly implementation; leading us to believe that our assembly solution in the preceding part is valid.

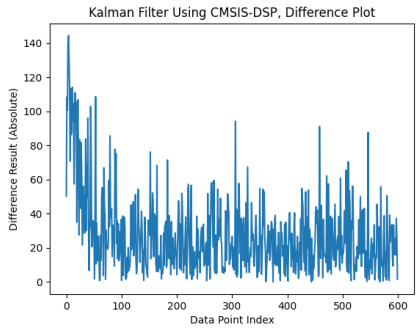

Fig. 8. CMSIS-DSP Implementation Difference Plot

Moreover, we used the CMSIS library defined function of correlation and convolution in order to arrive at the results in figure 9 and 10. As before, the results allow us to infer that the plots are seemingly identical to that of the preceding section.

C. Plain C Kalman Filter Implementation

We then rewrote our core Kalman filter function entirely in C without utilizing any of the built-in functions of the CMSIS library. This also included writing functions to perform the filter analysis steps of difference, standard deviation, mean, correlation and convolution. We encompassed all the analysis steps into one larger container function named AnalyzeFilter, which allowed us to perform each of the steps of the analyzation through one container, while implementing each of the steps separately in its own function declaration.

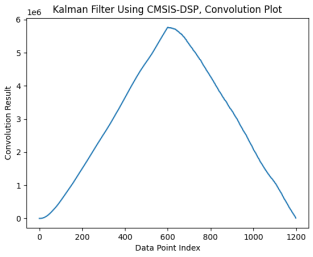

Fig. 9. CMSIS-DSP Implementation Convolution Plot

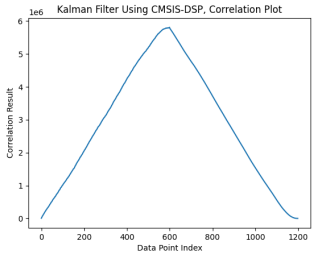

Fig. 10. CMSIS-DSP Implementation Correlation Plot

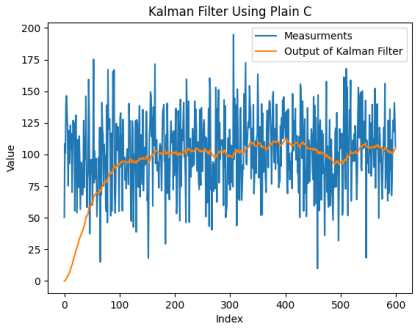

Fig. 11. Plain C Implementation Input/Output Stream Plot

Using this container function along with the capabilities of the debugging mode on our target processor, we are able to extract the output data stream of the C implemented Kalman filter. Like the preceding sections, the plot of the input and output stream (figure 11) showcases how the Kalman filter converges to the input data stream and accounts for the noise experienced by the system during measurement. The plot again, is seemingly identical to that of our assembly and CMSIS implementations; allowing us to infer that the same exact operations are being conducted in each of the separate implementations. This is also the case for the plot of the difference between the two data streams found in figure 12.

Implementation Type	Standard Deviation	Mean
Assembly	35.3178368	7.98565578
CMSIS-DSP	35.2043991	7.98565578
Plain C	35.3178368	7.98565578

TABLE I
STANDARD DEVIATION AND MEAN ACROSS DIFFERENT IMPLEMENTATIONS

We then utilized the capabilities of the debugger console/interface in order to log the results of our C implemented convolution and correlation operations. As seen from figure 13 and 14, the operations are also identical when implemented in C with the relevant data type declarations.

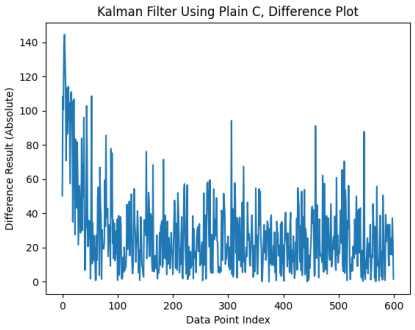

Fig. 12. Plain C Implementation Difference Plot

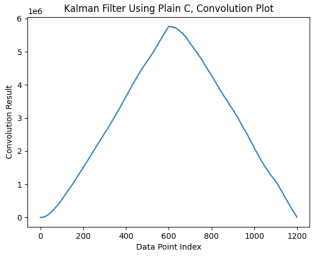

Fig. 13. Plain C Implementation Convolution Plot

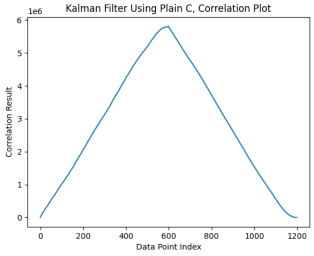

Fig. 14. Plain C Implementation Correlation Plot

Finally, we have left the discussion of the standard deviation and the mean of the difference between the two data streams; input and output for the last subsection of this section. Our results for all three implementations are presented in table I. As seen from the table, our three implementations all arrive at the same value for the standard deviation and mean of our testing data set. More so, the values match for up to 8 decimal points, leading us to believe that the operations being conducted are identical to each other. However, as observed from table I, the standard deviation of the CMSIS-DSP implementation varies from the other two implementations by a magnitude of 0.1134377. Albeit this difference is very small, it leads us to believe that the CMSIS calculation uses a varying technique to calculate the standard deviation between two sets of data. To validate this, we have rewritten the CMSIS-DSP implementation multiple times, some of which include the direct definition of the standard deviation calculation from the CMSIS reference manual, and yet all result in some form of deviation from the other implementations; and so we have opted for the structure which minimizes this discrepancy.

D. Code Profiling and Analysis

Implementation Type	Time Elapsed (ms)
Assembly	0.766408
CMSIS-DSP	48.645692
Plain C	635.157708

TABLE II
CODE PROFILING ACROSS DIFFERENT IMPLEMENTATIONS

Table II provides the code profiling results of our implementations. Code profiling was performed by utilizing the ITMPort32(31) which captures the time at execution. Two port objects were placed; one before the function call and one after, allowing us to examine the time elapsed as seen in table II. From our data we can observe that the assembly language implementation of the Kalman filter outperforms its C and CMSIS-DSP counterparts significantly. The data indicates that the assembly implementation has an execution time of 0.766408 ms, while the C and CMSIS-DSP implementations have execution times of 635.157708 ms and 48.645692 ms, respectively. This large discrepancy in execution times can be attributed to the fact that assembly language allows for low-level access to the underlying hardware, enabling developers to write code that directly manipulates registers and memory with minimal overhead. In contrast, C and CMSIS-DSP are higher-level languages that introduce additional layers of abstraction and safety, which can impact the execution time. Moreover, while C and CMSIS-DSP implementations are likely to be more portable, they may not be optimized for the specific processor architecture, whereas assembly code can be fine-tuned to take full advantage of the hardware platform. Therefore, the assembly implementation of the function should be preferred over the other two options if performance is a primary concern.

While this conclusion holds, it should also be mentioned that the assembly implementation does not call the analyze function, and hence a decreased time elapsed is represented here compared to the other two implementations. And so, while the magnitude of the differences is grand enough to assume that the source of difference is the abstraction level, the AnalyzeFilter function should also be taken into account.

We then used the debugger to inspect and modify the program variables while the code is running, to analyze which variables can be modified. By experimentation, we arrived at the following qualitative results: It is important to understand the rules governing which variables can be watched and modified, and how this can be done without interrupting the processor. In general, any variable that is in scope and accessible at the current point in the program can be watched and modified. This includes local variables, global variables, and static variables, as well as members of structs and classes. However, it is important to note that some variables may be optimized out by the compiler, which means they will not be available for inspection or modification.

To modify a variable using the debugger, the user can enter a new value into the field next to the variable. It is important to ensure that the value entered is of the same type as the variable, otherwise it may be automatically converted or truncated to fit the original type. Additionally, values can only be modified within the scope in which they were defined. Attempting to modify a variable outside its scope will result in a compiler error or undefined behavior at runtime.

III. Conclusion

To conclude, the provided data clearly shows that the assembly implementation of the Kalman filter function outperforms its C and CMSIS-DSP counterparts in terms of program execution time. The significant difference in execution times is due to the low-level access that assembly language provides to the hardware, allowing developers to write code that directly manipulates registers and memory with minimal overhead. While C and CMSIS-DSP are more portable, they introduce additional layers of abstraction and safety, which can impact execution time. If performance is a top priority, the assembly implementation of the Kaman filter function should be preferred. However, using assembly code requires a higher level of expertise and a more careful approach to development and maintenance.

This investigation highlights the benefits of using ARM assembly language and the CMSIS-DSP API for developing efficient Cortex-based applications. By integrating assembly code for a single-variable Kalman filter with the CMSIS-DSP API, we demonstrate the practical application of these technologies in accelerating the development of complex applications. The use of high-level APIs such as the CMSIS-DSP API streamlines this process and optimizes performance.