Integer Fast Fourier Transform on an ARM Cortex M3

Ethan Goldstein,

Electrical Engineering, University of Victoria

V00770062, [gstein@uvic.ca](mailto:gstein@uvic.ca)

David Nelson,

Electrical Engineering, University of Victoria

V00223232, [nelsonda@uvic.ca](mailto:nelsonda@uvic.ca)

Submitted on: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[Introduction 3](#_Toc457475710)

[Theory 4](#_Toc457475711)

[Software Optimizations 6](#_Toc457475712)

[General Information 6](#_Toc457475713)

[Refining function specificity 6](#_Toc457475714)

[Replace calculations for cos values with look-up function 6](#_Toc457475715)

[Make use of temporary variables 6](#_Toc457475716)

[Reduce load/store operations 6](#_Toc457475717)

[Aid pipeline optimization 7](#_Toc457475718)

[Loop Unrolling 8](#_Toc457475719)

[Rewrite code to expose hidden operator efficiencies 10](#_Toc457475720)

[Compiler based optimizations 11](#_Toc457475721)

[Restricted Arrays 11](#_Toc457475722)

[Hardware Optimizations 11](#_Toc457475723)

[Digital Filter Block (DFB) 11](#_Toc457475724)

[VHDL 12](#_Toc457475725)

[SOC 13](#_Toc457475726)

[Performance and Cost Evaluation 14](#_Toc457475727)

[Conclusion 16](#_Toc457475728)

[References 16](#_Toc457475729)

Table of Figures

[Figure 1: Bit reversal of time domain input. 4](#_Toc457475730)

[Figure 2: Basic Butterfly operation with twiddle factor. 4](#_Toc457475731)

[Figure 3: Diagram for FFT operations. 5](#_Toc457475732)

[Figure 4: Graphical representation of the input, outputs and error. 14](#_Toc457475733)

Table of Tables

[Table 1: Performance metrics. 14](#_Toc457475734)

# Introduction

The FFT is a type of DFT, which is a common algorithm implemented in digital system to compute the Fourier Transform of a data set. The FFT converts a time domain signal into a frequency domain signal, which is useful in a wide variety of applications. The computational complexity of the DFT is O(N2*)* while the computation complexity of the FFT is O(Nlog2(N)). The key to decreasing the computational complexity is to make N a power of 2, and if there is not enough data to fill up the power of 2, then it can be padded with zeroes[[1]](#footnote-1). The FFT was originally invented by Carl Friedrich Gauss in 1796 as a way to interpolate positions of asteroids[[2]](#footnote-2). Two centuries later the algorithm was reinvented by Cooley and Tukey in 1965, to detect nuclear tests by the Soviets, through the analysis of ground vibration sensors spread out over the globe.

The ARM Cortex M3 is a popular architecture used in many commercial embedded systems. The goal of this project is to optimize the FFT algorithm for the ARM Cortex M3 architecture, to obtain the fastest transform possible, while avoiding the use of inline assembly, or pure assembly. Initially, the project started with Integer FFT code that was created for a generic architecture, resulting in slow performance. To test the Integer FFT, code was found on Matlab’s website that performed an Integer FFT, which yields very similar results to the FFT executed on the M3. The total list of changes to the FFT code to optimize for the ARM M3 are as follows:

1. Re – write code to expose hidden operator inefficiencies
2. Used Builtins, to increase performance of loops and memory disambiguation
3. Created two LUTs with modifier type const
4. Completely unrolled the bit reversal algorithm
5. Fixed the FFT size to 1024 to increase loop iteration performance
6. Used MLA and MLS instructions, replacing ineffective separate MUL, ADD and SUB
7. Increased pipelining performance through grouping load/store and arithmetic instructions

The work for this project was split between Ethan Goldstein and David Nelson. Git was used as the revision control system, using the website [www.github.com](http://www.github.com) as a server. The initial code for the FFT performed on the ARM is found at [www.jjj.de/crs4/integer\_fft.c](http://www.jjj.de/crs4/integer_fft.c). Additionally, the generation of the LUTs and the bit reversal unrolling was generated using a python script, which is included in the appendix. The python script uses code found from <http://www.maxim-ic.com/images/appnotes/3722/3722Firmware.zip> as a guide in unrolling the bit reversal algorithm.

# Theory

The FFT is comprised of two main sections, the first section comprises of a bit reversal algorithm (reordering the data) and the second section comprises of multiplications and additions. Bit reversal reorders the input array into an order that after the second stage has processed the data, the data is in the correct order for frequency domain. The reordering at each stage of the bit reversal is called an interlace decomposition. The code for the bit reversal algorithm can be found in the appendix. The image below shows bit reversal for a 16 point FFT.

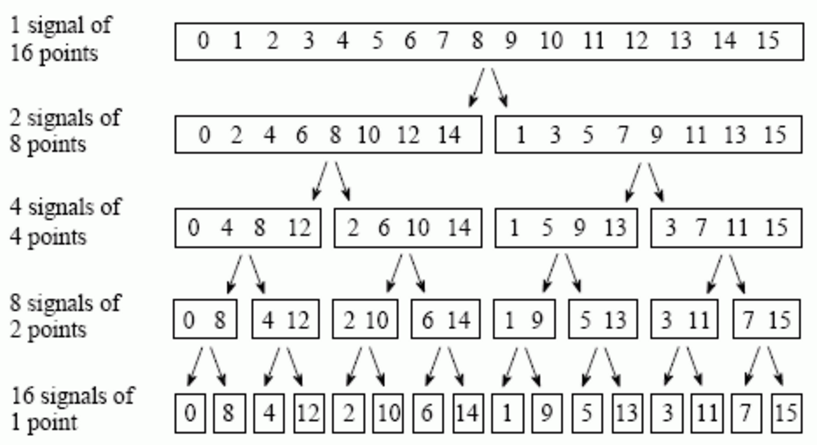


Figure 1: Bit reversal of time domain input[[3]](#footnote-3).

After the bit reversal has occurred the data is passed to a butterfly unit, which performs many multiply and accumulate operations. The output of the butterfly operation is modified by multiplying it with a twiddle factor. A twiddle factor is trigonometric constant coefficient that is multiplied to the input of the butterfly operation[[4]](#footnote-4). The image below shows a single, generic butterfly operation, modified by a twiddle factor.

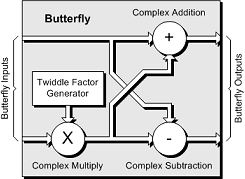


Figure : Basic Butterfly operation with twiddle factor[[5]](#footnote-5).

The FFT is comprised of many butterfly operations, initially starting out with the bit reversal array, and ending up with an array of data in the correct order, from lowest to highest frequency components. The diagram below shows the butterfly diagram for the 16point FFT. The twiddle factors are represented by W.

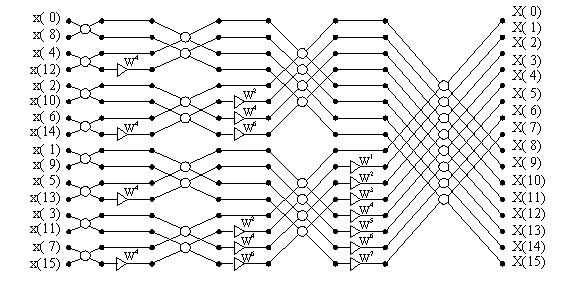


Figure : Diagram for FFT operations.

The 16 point FFT shown above is comprised of 4 separate stages, the number of stages is calculated according to the formula below. N is the number of input points, in this case it is 16.

The number of butterfly operations per stage is computed by the following formula.

At each stage, new twiddle factors must be computed. Fortunately, the twiddle factors can be computed offline, and stored in a LUT. There are 2 LUTs required for computing twiddle factors are a Sin LUT and Cos LUT, where the size of both LUTs is N/2. Both LUT start at the trigonometric function evaluated at 0, and the final element is the value of the trigonometric function evaluated at π. Furthermore, each element of the sin LUT is multiplied by -1. Shown below are samples of the sin LUT.

…..

# Software Optimizations

## General Information

In this section we will discuss techniques for producing efficient assembly code from C code. The first step in increasing code efficiency is change the level of optimization used by the complier. GCC supports 5 levels of optimization; none, minimal, high, speed, space, and debug. Speed optimization performs the most in-depth optimization routine. All compilations and comparisons of code were done using the -03 speed optimization flag. We also enabled the finline-functions flag, which can enable the in-lining of functions (replace of function in coded vs branching to function), although the compiler does not appear to inline some of our functions despite the fact that they have been qualified with inline keyword in C.

## Refining function specificity

The code for the FFT we started with had many noticeable inefficiencies. Due to the fact that the function could perform the FFT or the inverse FFT there was significant overhead involved in dynamically alternating the flow of control of the function. By separating the FFT and inverse FFT into two separate functions, we reduced the number of conditional checks and branch instruction within our FFT significantly. Since branch instructions have some of the highest penalties for a pipeline, this increased our code efficiency. It makes the flow of the algorithm easier to understand, which made further optimization easier.

## Replace calculations for cos values with look-up function

When retrieving the twiddle factors, the original code dynamically calculated the cosine values to retrieve by performing and addition on the sine index (effectively looking 90 degrees ahead). By converting the sinewave table into separate sine wave and cosine wave tables, we save an addition around each other outer loops. It also helped prevent buffer overflow that occurred when looking up some cosines values.

## Make use of temporary variables

Within the inner most loop of our FFT, there existed multiple cases where the same memory address was being read. We changes the code so that temporary variables are read the initial value of the memory address we wish to edit. All further calculations are performed on the temporary memory address, and the final result is written back to the memory address. This has two purposes.

Original C



Optimized C

****

### Reduce load/store operations

First, by using temporary variables in this way we eliminate the need for additional read operations each time the memory address is needed for a calculation. This reduced the number of load/store operation in called within the code, but puts additional pressure on the registers. Given that there are 12 general purpose registers in the M3’s architecture, and we have 16 temporary variables in the FFT, it is clear that some additional load/store operations will be needed when switching between difference portions of the function. The area of code we changed to use temporary variables is among the most frequently called section of code in our FFT algorithm, meaning savings in the critical section will have the largest overall impact of performance.

### Aid pipeline optimization

There is an additional advantage to using temporary variables. Based on ARM’s Programmer Model independent load operations are more easily pipelined [1]. Another advantage of using temporary variables is to eliminate the presence of DMB or DSB instructions. By changing the order of load/stores and arithmetic operations, the need for data synchronizations wait cycles can be reduced.

Original C Code



Original Assembly



Modified C



Modified ASM



By using temporary variables in this manner 4 total instructions are saved in .L2. Note that .L2 is the most frequently run loop in the whole program, so a savings of 4/34 (11.76%) reduction in instructions is significant in terms of the total time savings of the algorithm.

## Loop Unrolling

Loop unrolling, in general reduces the need for the compiler to check the branch equality during each iteration of the loop. This directly reduces the number of clock cycles that are needed to perform a given segment of code, at the expense of maxing the code larger and harder to maintain. Before the main part of calculation, sampled data is reordered. This reordering of data corresponds to a series of load/store operations. By removing intermediate operations for dealing with branching, loop unrolling produces assembly that the architecture can do a better parallelizing. Individual load and store operations are 2 cycle operations minus some factor for pipelining that happens when adjacent instructions are load stores. For a group of 3 load/store operations the total number of clock cycles needed is 4, for a savings of 2 clock cycles ( 1/3rd increase in performance) when compared with an un-pipelined implementation.

Original C



Original ASM



Modified C



Modified ASM



The above assembly code shows the instruction level parallelism. Lines with multiple MEM[] instructions are executing multiple parts of the load store operation. In the original code, the branching around the loop prohibits the complier from being able to pipeline the load stores in the same way. It is also interesting to note that due to compiler optimization, the order of load and stores in the assembly does not match the C code. The compiler is ‘smart’ enough to optimize blocks of the same load/store operation pipelining by the MPU hardware. This could save as much as 1/3 of clock cycles, so the speed up for bit reversal may be as much as 1/3rd after loop unrolling. Loop unrolling could be taken much further for our FFT. Since we have removed all branching that is not known at compile time, we could unroll the entire FFT in a manner similar to bit\_reversal. This would increase the size of the code, but would eliminate conditional checks and branching. It would also allow for a greater degree of hardware pipelining.

## Rewrite code to expose hidden operator efficiencies

The ARM Cortex M3 on which the Cypress PSOC 51xxx is based has a set of DSP specific instruction. Some of these DSP instructions like multiply and subtract, and multiply and add combine two operation into one. Looking at a critical section of the FFT, we noticed two segments of code where the value assigned to a register was the difference of two multiplications. By rewriting the code so that the first multiplication is assigned to the register, and the second multiplication is added or subtracted to the first multiplication, we managed to lead the complier to make use of MLA and MLS instructions.

Original C



Original C



Modified C



Modified ASM



It should be noted that the MLS and MLA instructions are 2 cycle instructions on the Cortex M3. This means that exposing the MLS and MLA instructions does not directly reduce cycle count. However, since the total number of instructions issued is less, there is

## Compiler based optimizations

We also experimented with GCC \_builtin\_ compiler flags. These flags give the compiler more information about the code, which allows for more efficient compilation. First we tried to change the cost associated with branching. Since most of the branching statements in our program are loops that are run many times, it may be possible for compiler or MPU architecture to use predictive branch detection. We used the \_\_builtin\_expect pragma for the loop conditions, with no noticeable effect on performance. This is not surprising. The Cortex M3 does not support branch prediction [2]. However, some embedded controllers like the Cortex R5 do support branch prediction, so this technique could be useful in the general case.

Macro definitions



Macro Use



### Restricted Arrays

In order to help the compiler to arrange memory accesses in the most efficient way possible, it is important for the compiler to know whether the arrays being passed to the function can be in the same location. This could create a situation where a read/write error occurs, so a wait may be needed to make sure there is no resource conflict. By using the \_\_restrict\_\_ pragma, we specified to the compiler that the real and imaginary components of the input signal are stored at unique locations.

Original C



Optimized C



# Hardware Optimizations

There are 3 techniques for using hardware to optimize the FFT algorithm that we will discuss here. The development board we used was a Cypress PSOC 5LP.

## Digital Filter Block (DFB)

One of the features of this board is a digital filter block (DFB). The DFB is specialized a pipeline for filtering application, and includes its own control and data path that is separate from the MPU. The control path is executed by a RAM-based finite state machine, which control how data enters and leaves the datapath. Data is usually transferred into and out of the DFB using DMA. The datapath includes a multiply and accumulate function as well as an ALU, and the DFB has its own instruction assembly instruction set.

It should be possible to configure the DFB and DMA to execute the FFT. Unfortunately, the DFB is not specifically designed for this application so likely will not perform the FFT as fast as the MPU. However, using the DFB would free MPU clock cycles which could be used to perform other task such as service a UI. For better performance from a hardware, more control over the specifics of the data path would be an advantage.

## VHDL

Using a hardware description language in conjunction with a synthesis tool and an FPGA offers another way to use hardware to accelerate the calculation of the FFT. One of the most common HDLs, VHDL, shares many syntactic features with C, but differs by offering concurrently executing statements. By allowing more operations to occur at the same time, clock cycles may be reduced. However, increased parallelism usually comes at the cost of needing more gates and more logic, which necessitates large FPGA. For example, the section of code below requires 4 multiplies and one add or subtract. Using VHDL this code



could be rewritten in VHDL as



In the code snippet above, tr and ti are calculated at the same time. Within the first statement, wr \* fr\_j\_temp and wi \* fi\_j\_temp may also be synthesised so that they are calculated at the same time. Following this logic, and allowing for 1 cycle multiplication, the above section of code could be calculated in at most 2 cycles. However, this would come at the expense of needing 4 multipliers, which is an incredible amount of chip real-estate. It is likely that instruction would need to be broken into multiple stages to make more efficient use of physical resources. Using a state machine with only one hardware multiplier would produce VHDL code as follows:



## SOC

The ultimate hardware support for a function is the creation of an SOC. The new SOC will augment the existing MPU datapath with a functional block to perform a subset of the operations needed in the FFT (for example the multiplication above). A tool like SystemC can be used simulate the FFT. First an all software solution to the FFT could be implemented. Parts of the FFT can then substituted for a hardware implementation. One the concept has been proven, actual design of the physical SOC could begin.

# Performance and Cost Evaluation

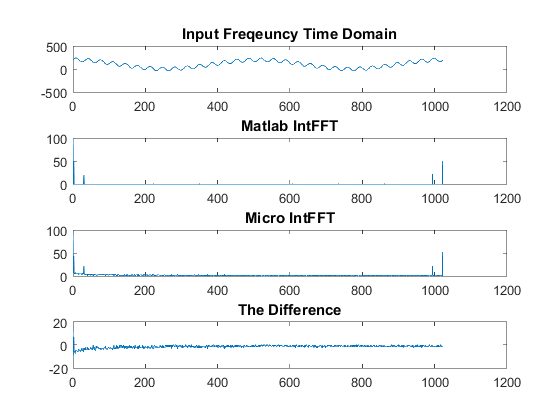
The output of the microcontroller FFT is very close to the FFT computed in Matlab. The image below shows the performance of the system.

Figure : Graphical representation of the input, outputs and error.

The table below shows key metrics when evaluating the performance of the system. The Clock Cycles were measured by setting up a hardware timer that began counting when the FFT started, and was stopped once the FFT completed.

Table 1: Performance metrics.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Not Optimized** | **Optimized** | **Improvement (%)** |
| **NOR Flash (Bytes)** | 12032 | 19968 | -66 |
| **SRAM (Bytes)** | 4541 | 2493 | +45 |
| **Clock Cycles** | 991488 | 428928 | +57 |

It is clear from the table that the optimization increased the speed by 57% and reduced the SRAM usage by 45%. However, flash usage increased by 66%. Furthermore, the execution time was reduced by 57%, which also corresponds to 57% less power consumed by the microcontroller. To analyze the cost of the software, two factors are taken into account, SRAM and Flash. The SRAM usage has been reduced by 2048 bytes, and the Flash usage has increased by 7936 bytes. This means that approximately 4 times more flash has been consumed than the SRAM has decreased. The cost of SRAM is significantly more than that of Flash, corresponding to the silicon area. A single bit of NOR flash, is a single transistor, while a CMOS SRAM bit occupies 6 transistors. Therefore, the assumption will be made that SRAM is 6 times more expensive than Flash. Using this assumption, the silicon area required by the new software roughly the same area as the non – optimized solution.

# Conclusion

The FFT is an important algorithm for a wide variety of modern applications ranging from video to music editing and more. Since this algorithm is used so much, and since the FFT is usually just a small part of a larger signal processing systems, the efficiency of this algorithm can be important to overall system performance. The importance of an efficient FFT algorithm made more difficult in embedded systems which lack some of the sophisticated hardware, such as floating point units, that is available to full feature processors such as Pentium. For our project, we took on the task of improving an existing integer arithmetic FFT and further optimized it through a series of software optimization techniques. Most of the software optimizations techniques are focused on modifying C code so that it produces more efficient assembly. Only by drilling down to the assembly level is it possible to see how the total number of instruction that particular section of code requires.

Looking at the assembly generated by the compiler, we were able to identify a few particularly useful optimizations. Fully unrolling the bit reversal portion of the algorithm reduced the number of comparison and branching operations. It also enabled a much higher degree of hardware optimization.

The core of the FFT algorithm is the butterfly. By rewriting this critical section of code, we enable the compiler to see a more efficient set of instructions (MLA, MLS) can be implemented. Also within the main body of the FFT, the use of temporary variables helped reduce the number of read/write operations. Using temporary variables also allowed blocks of load and store operations to be performed together, which allows the MPU to achieve a higher level of parallelism. Loading multiple store and load operation can save as much as 1/3rd of clock cycles.

Based on our analysis of the critical pathway for the FFT algorithm as being in the butterfly, we proposed different hardware based solutions that would allow the MPU to be freed of some or all of the computational load. The chip we used has a digital filter block with dedicated MAC hardware, which could be repurposed to computer part of our FFT. Alternatively, VHDL could be used to program the FFT to run an FPGA. The FPGA can allow for more or less parallelism, based on the amount of hardware available.

Ultimately, our optimization techniques produced an integer math FFT that is considerably better than the original. By using a system timer, we were able to measure that our time to calculate the FFT was reduced by 50% compared to the original code.

# References

1. <http://www.ece.rice.edu/~srs1/files/Circuits_10_26.pdf>
2. <http://www.cis.rit.edu/class/simg716/Gauss_History_FFT.pdf>
3. [www.github.com](http://www.github.com)
4. [www.jjj.de/crs4/integer\_fft.c](http://www.jjj.de/crs4/integer_fft.c)
5. <http://www.maxim-ic.com/images/appnotes/3722/3722Firmware.zip>
6. <http://www.dspguide.com/ch12/2.htm>
7. <https://en.wikipedia.org/wiki/Twiddle_factor>
8. <http://www.dilloneng.com/uploads/2/1/2/2/21220816/4887989_orig.gif>
9. <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337h/DDI0337H_cortex_m3_r2p0_trm.pdf>
10. <https://community.arm.com/thread/5337>

1. <http://www.ece.rice.edu/~srs1/files/Circuits_10_26.pdf> [↑](#footnote-ref-1)
2. <http://www.cis.rit.edu/class/simg716/Gauss_History_FFT.pdf> [↑](#footnote-ref-2)
3. <http://www.dspguide.com/ch12/2.htm> [↑](#footnote-ref-3)
4. <https://en.wikipedia.org/wiki/Twiddle_factor> [↑](#footnote-ref-4)
5. <http://www.dilloneng.com/uploads/2/1/2/2/21220816/4887989_orig.gif> [↑](#footnote-ref-5)