Improving Codec Execution using Arm Cortex-M Processors

Author: Jacob Beningo,
Beningo Embedded Group

White Paper

"The drive to push more computing to the edge is forcing several new challenges on developers."

Abstract

Digital Signal Processing (DSP) has traditionally required the use of an expensive dedicated DSP processor. While solutions have been implemented in microcontrollers using fixed-point math libraries for decades, this does require software libraries that can use more processing cycles than a processor capable of executing DSP instructions. In this paper, we will explore how we can speed up DSP codecs using the DSP extensions built-in to the Arm Cortex-M processors.

Technology trend: moving processing to the edge of the network

The embedded systems industry is currently undergoing a significant transition that is being driven by the need to have intelligent, real-time, edge devices in the IoT. As systems need to do more at the edge to enable more compute performance, it's forcing devices to do far more in software than ever before. In the past, data could be streamed to the cloud for processing and analysis, but as intelligent real-time systems begin to fill the landscape, that processing needs to be done at the edge.

First, as more computing moves to the edge, more processing power is needed at the edge. This leads to other challenges, such as minimizing energy consumption, processing the plethora of signals that are available at the edge and finding ways to minimize bil-of-materials (BOM) costs in a processing-hungry environment.

As we move towards the edge device, developers need to find ways to improve the efficiency at which they process signals and data in a way that minimizes component count and BOM cost. To meet these needs, developers can leverage the DSP capabilities that are built into the Arm Cortex-M4, Cortex-M7, Cortex-M33 or Cortex-M35P processors.
In the past, developers had to grapple with software challenges such as:

- Expensive dedicated DSP processors
- Time-consuming software development
- Longer testing cycles
- Required expert knowledge
- Fixed point numerical libraries

Today, developers can leverage many different solutions to help them quickly, easily and cost-effectively utilize DSP in their designs. These include:

- Using a microcontroller with DSP extensions
- Utilizing modeling software that can generate the software algorithm
- Accessing free DSP libraries, such as CMSIS-DSP

In this paper, we will discuss how developers can improve DSP codec execution using the Arm Cortex-M DSP extensions and associated libraries.

What are DSP extensions?

DSP extensions provide a microcontroller with an extended instruction set that includes DSP instructions, such as single instruction multiple data (SIMD) that allow DSP algorithms to execute faster than on a standard microcontroller. A standard microcontroller is limited in its DSP instruction set, which means if DSP is needed, either an additional DSP processor must be used or more clock cycles on the microcontroller must be expended to execute a software library that can add that capability.

There are several benefits to using a microcontroller that has DSP extensions. These benefits include:

- Significant savings on the BOM (i.e. cost of products), by replacing two processors with one.
- Reduced system-level complexity by removing the need for shared memory and DSP communication, complex multiprocessor bus architectures, and other custom 'glue' logic between the processor and DSP.
- Reduced software development costs, because the entire project can be supported using a single compiler/debugger/IDE, and is programmable in a high-level programming language such as C or C++

"DSP extensions make processing codecs faster by including instructions that speed up common DSP operations."

DSP software can be an extremely powerful tool for developers who understand how to utilize it properly. To learn more about the benefits that DSP offers, check out my latest blog: 5 benefits to replacing analog components with DSP software.
You can also learn how to get started with DSP by reading my blog "5 Tips for Getting Started with digital signal processing on Arm Cortex®-M CPUs".

The DSP extension instruction set

At the lowest levels, the instruction set makes all the difference for how efficiently the DSP algorithms can be executed. Figure 1 shows a break-down of the different instructions that are supported by the Arm Cortex-M processors. As you can see, there are quite a few instructions supported by the Cortex-M4, Cortex-M7, Cortex-M33 and Cortex-M35P processors. There are several instructions set features that are particularly interesting to embedded developers working in DSP applications.

"This provides a level of parallel processing where the register is still 32-bits, allowing more performance to be squeezed out of the processor. "

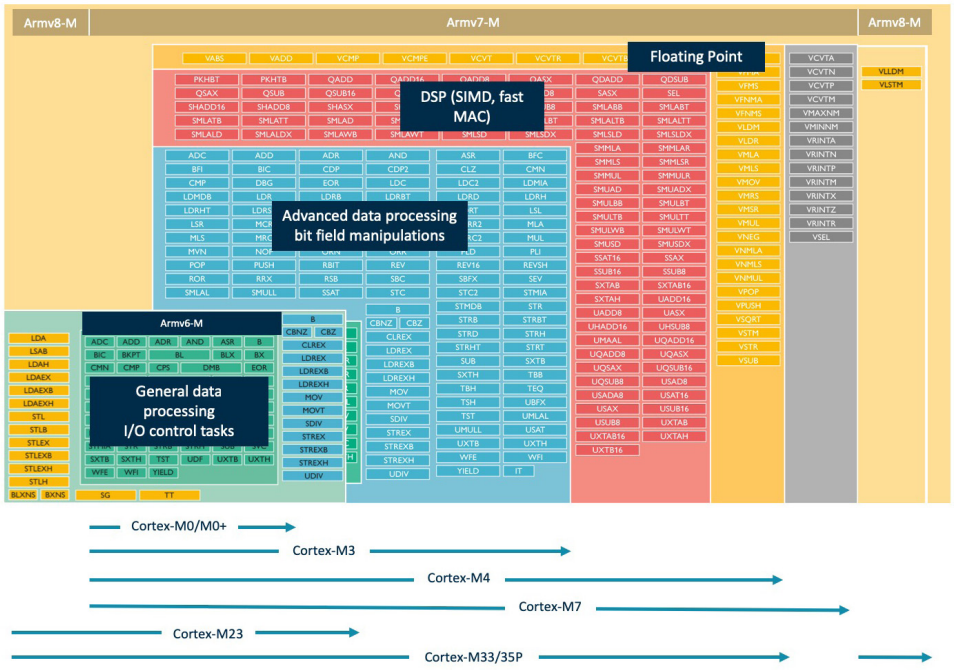

Figure 1 – The Arm Cortex-M family instruction set. It includes DSP and floating-point instructions to maximize performance.

First, the SIMD instructions can operate on 8-bit or 16-bit integers, which allows a single instruction to operate on multiple pieces of data within the same register. For example, the SIMD instruction can operate on either two 16-bit values or four 8-bit values. This is useful because 8-bit and 16-bit data types are most often used in audio and video processing, which do not require 32-bit precision and are the most advanced types of signals that need to be processed at the edge.

In addition to the SIMD instruction, there are many other useful instructions within the Arm Cortex-M instruction set. These features include:

- 32-bit accumulate
- Signed multiply with up to 64-bit accumulate
- Unsigned multiply
- Saturate
- Packing and unpacking
- Parallel addition and subtraction

MAC instructions are used in many filter applications such as IIR and FIR filters which form a major foundation in DSP applications. To see a full list of Arm Cortex-M DSP instructions, please visit the Arm website.

Utilizing the Floating Point Unit (FPU) to improve performance

Floating point mathematics can be a challenge from a performance stand-point for a microcontroller-based system especially at the edge. Many signal processing algorithms require floating point mathematics. In a microcontroller without a floating-point unit, a software library is provided that performs the calculations, but these calculations require more computing cycles and more code space in order to get the job done. The FPU can not only help decrease the calculation time, but can also decrease the code size and improve the determinism of the calculation.

In a recent issue of The Embedded Muse1, Jack Ganssle published some relative numbers on how the Cortex-M4F floating point unit behaved. In his experiment, the Cortex-M4F was running at 120 MHz with no cache, a single wait state and the interrupts were disabled. Using the IAR compiler, Ganssle created an executable that was approximately 15 kB that measured 32-bit floating point numbers and 64-bit floating point numbers with and without the FPU. His analysis looked at a differing number of parameters which could result in a range of execution times and also measured the relative code size. Below, in Figure 3 and Figure 4, you can see the results from running 32-bit operations with and without the FPU.

32-bit Operation (no FPU)	Execution Range (ns)	Relative Code Size
Addition	181 – 321	0 bytes
Subtraction	181 - 321	12 bytes
Multiplication	259	0 bytes
Division	458 - 533	- 20 bytes

Figure 3 – Jack Ganssle's Cortex-MF FPU measurement results for 32-bit floating point without an FPU from The Embedded Muse Issue 3691.

32-bit Operation (no FPU)	Execution Range (ns)	Relative Code Size
Addition	17	-786 bytes
Subtraction	17	-786 bytes
Multiplication	17	-786 bytes
Division	83	-786 bytes

Figure 4 – Jack Ganssle's Cortex-MF FPU measurement results for 32-bit floating point with an FPU from The Embedded Muse Issue 3691.

"For 32-bit operations, we can see from this experiment that using the FPU increases the performance by at least 10x"

As you can see, the no FPU versions require a range of computing time as low as 181 nanoseconds (ns) for addition and up to 533ns for division. The code size for the operation remained pretty close to constant. However, with the FPU, with a range of parameters passed, Ganssle found that the FPU executed the operation in 17ns for addition, subtraction and multiplication and then 83ns for division. The code size also decreased by almost 800 bytes.

64-bit Operation (no FPU)	Execution Range (ns)	Relative Code Size
Addition	293 – 745	488 bytes
Subtraction	290 - 703	488 bytes
Multiplication	525 – 533	-100 bytes
Division	1035 - 1329	-100 bytes

Figure 5 – Jack Ganssle's Cortex-MF FPU measurement results for 64-bit floating point without an FPU from The Embedded Muse Issue 3691.

64-bit Operation (no FPU)	Execution Range (ns)	Relative Code Size
Addition	17	-786 bytes
Subtraction	17	-786 bytes
Multiplication	17	-786 bytes
Division	83	-786 bytes

Figure 6 – Jack Ganssle's Cortex-MF FPU measurement results for 64-bit floating point with an FPU from The Embedded Muse Issue 3691.

Depending on the operation being performed the improvement can be significantly better. For this reason, including an FPU in your DSP applications can be extremely important depending on the data types that you are working with. Ganssle also did a comparison of 64-bit operations which can be seen below in Figure 5 and Figure 6.
As you can see from these experiments, the FPU made little difference in execution speed for these large data types but the FPU was able to help decrease the relative code size.

The CMSIS-DSP Library

The most efficient way to utilize the DSP extensions in a Cortex-M processor is to leverage the CMSIS-DSP library. The CMSIS-DSP library is a collection of over 60 free algorithms that make developing DSP software easier. Figure 7 shows an example of the main categories that developers can find in the library.

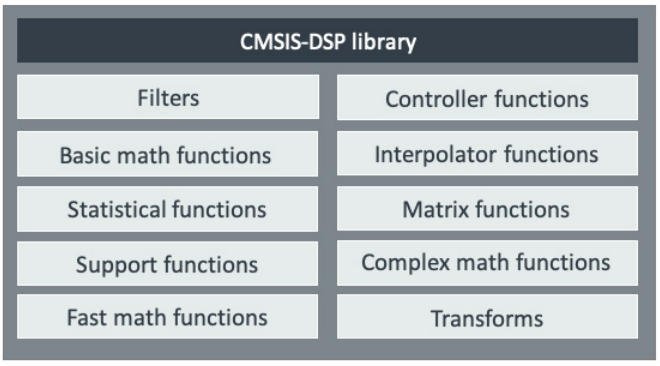

Figure 7 –The CMSIS-DSP library is a free library containing over 60 algorithms that developers can use to speed up their DSP software development.

The library contains functions for designing filters, calculating interpolations, performing complex mathematics and performing transforms. A common use of the CMSIS-DSP library is to create IIR and FIR filters in addition to calculating a Fast Fourier Transform (FFT).

To dig deeper into the CMSIS-DSP library, you can review the CMSIS-DSP library documentation here.

Let's look at an example implementation of an IIR filter that utilizes the CMSIS-DSP library using ASN Designer.

"With the automatically generated code, not only are possible mistakes removed, but developers can easily fine-tune their filters without having to go back and rewrite or rework their code. "

IIR Filter Deployment using ASN Designer

Advanced Solutions Nederland BV is an Arm DSP partner that provides a tool called the ASN Filter Designer. It allows developers to create different filters, such as FIR and IIR, and generate embedded code that implements that filter. You can see below in Figure 8 an example IIR filter header that has been designed in ASN Designer and automatically generated into code. Normally a developer would have to design their filter in an excel spreadsheet and then write all the code for the filter from scratch. Instead, this can be done through high-level software and then the low-level code generated. This IIR example has 2 passbands between 0 – 25 Hz and 64.1 and 250 Hz.

```
// ASN Filter Designer Professional v4.0.5
// Wed, 18 Oct 2017 09:50:05 GMT

// ** Primary Filter (H1)**

////Band#       Frequencies (Hz)     Att/Ripple (dB)
//    1          0.000,   25.000          0.001
//    2         64.104,  250.000         40.533
////
// Arithmetic = 'Floating Point (Single Precision)';
// Architecture = 'IIR';
// Structure = 'Direct Form II Transposed';
// Response = 'Lowpass';
// Method = 'User Defined';
// Biquad = 'Yes';
// Stable = 'Yes';
// Fs = 500.0000; //Hz
// Filter Order = 6;

// ** ASN Filter Designer Automatic Code Generator **
// ** Deployment to ARM CMSIS DSP Framework        **
```

Figure 8 – An IIR filter that has bands between 0 and 25 Hz and 64.104 and 250 Hz.

Figure 9 shows all the code necessary to setup the IIR filter. You can notice that there are buffers for data which can be seen in OutputValues and InputValues. The transform for the IIR filter is clearly documented in the comments. The filter coefficients are also in a table that is easy to read.

```
//#define ARM_MATH_CM4   // Cortex-M4 (default)
#include "arm_math.h"
#include "cmsis_os2.h"                  // ::CMSIS:RTOS2


#define TEST_LENGTH_SAMPLES  512
#define BLOCKSIZE 32
#define NUMBLOCKS  (TEST_LENGTH_SAMPLES/BLOCKSIZE)

float32_t OutputValues[TEST_LENGTH_SAMPLES];
float32_t InputValues[TEST_LENGTH_SAMPLES];

float32_t *InputValuesf32_ptr = &InputValues[0];  // declare Input pointer
float32_t *OutputValuesf32_ptr = &OutputValues[0]; // declare Output pointer

arm_biquad_cascade_df2T_instance_f32 S;

#define NUM_SECTIONS_IIR 3

// ** IIR Direct Form II Transposed Biquad Implementation **
// y[n] = b0 * x[n] + w1
// w1 = b1 * x[n] + a1 * y[n] + w2
// w2 = b2 * x[n] + a2 * y[n]

// IIR Coefficients
float32_t iirStatesf32[NUM_SECTIONS_IIR*5];
float32_t iirCoeffsf32[NUM_SECTIONS_IIR*5] =
          {// b0, b1, b2, a1, a2
            0.1215975,  0.0751514,  0.1215975,  0.9095030, -0.2278495,
            0.2081503, -0.1286440,  0.2081503,  1.2015820, -0.4892390,
            0.7326008, -1.1853730,  0.7326008,  1.5383070, -0.8181361
          };
```

Figure 9 – This figure shows the declarations necessary to setup the IIR filter. Notice that the IIR filter uses 32-bit floating point numbers.

The interesting part for us, is that we want to look and see how the CMSIS-DSP libraries are used. As you can see in Figure 10, there are three different CMSIS-DSP calls to the following functions:

- arm_sin_f32
- arm_biquad_cascade_df2T_init_f32
- arm_biquad_cascade_df2T_f32

The arm_sin_f32 function is used in conjunction with a custom white_noise_gen function to create a test signal that is used for input into the IIR filter. The arm_biquad_cascade_ df2T_init_f32 function is used to setup the coefficients that will be used in the IIR filter. Finally, arm_biquad_cascade_df2T_f32 is used to perform the filter operation.

```
void app_main (void *argument) {

  uint32_t n,k;

  long seed=12357;
  float32_t Amplitude = 0.21;
  // setup test sinusoid input
  for (n=0; n<TEST_LENGTH_SAMPLES; n++)
    InputValues[n]= Amplitude + (arm_sin_f32(2*PI*50.00*n/500)/3)+white_noise_gen(&seed)/25;

  // Initialise Biquads
  arm_biquad_cascade_df2T_init_f32 (
                                    &S, NUM_SECTIONS_IIR,
                                    &(iirCoeffsf32[0]),
                                    &(iirStatesf32[0]));

  // Perform IIR filtering operation
  for (k=0; k < NUMBLOCKS; k++)
    arm_biquad_cascade_df2T_f32 (&S,
                                 InputValuesf32_ptr + (k*BLOCKSIZE),
                                 OutputValuesf32_ptr + (k*BLOCKSIZE),
                                 BLOCKSIZE);    // perform filtering

  for (int i=0; i<TEST_LENGTH_SAMPLES; i++) {
    osDelay(1);
    dbgout = OutputValues[i];
    dbgin= InputValues[i];
  }
  for (;;) {}
}
```

Figure 10 – The automatically generated output that creates and executes the filter on test data.

Conclusion

Many factors within the embedded systems industry are driving the need to reduce costs and improve execution performance at the edge. As we start to push processing to the edge, embedded developers will need to leverage DSP capabilities more and more. The Arm Cortex-M processors and ecosystem have built the foundation that developers need to easily implement DSP codecs while improving computing capabilities through DSP extensions and libraries, while minimizing BOM and costs.

To learn more about DSP on the Arm Cortex-M processor, consider the following free webinars to get started:

- How to choose between analog hardware and digital signal processing
- Running DSP Algorithms on the Arm Cortex-M
- DSP software development masterclass with Arm and MathWorks

Additional resources to help you get started:

- Arm Cortex processors with signal processing
- Arm Cortex-M processors with signal processing
- CMSIS-DSP software library
- Use ASN Filter Designer to generate CMSIS-DSP code
- Arm's DSP ecosystem partners

References

1) Embedded Muse Issue 369 by Jack Ganssle