Introduction to Digital Signal Processing For High Performance Cortex M3 and M4

May 2010
Outline

- Introduction
- DSP for Cortex M3
- M4 Target Applications
- Cortex M4
  - Instruction set
  - Performance
- Signal Processing
  - Overview
  - Optimizations
NXP is a leader in ARM Flash MCUs

- Clear strategy: 100% focus on ARM
- Top performance through leading technology & architecture
- Design flexibility through pin- and software-compatible solutions
  - Scalable memory sizes
  - Widest range of peripherals
- Unlimited choice through complete families for multiple cores

Go with the winner!

Cortex-M4
ARM9
Cortex-M3
ARM7
Cortex-M0

50+ options
10+ options
15+ options
20+ options
Bridging Two Great Families
ARM7 & Cortex-M3
NXP ARM Cortex™ Microcontrollers

Product Series Overview

- **Cortex-M0**
  - 40 - 50 MHz
  - LPC1100
  - Cortex-M0 ready to replace 8/16-bit

- **Cortex-M3 V2**
  - 60 - 70 MHz
  - LPC1300
  - Plug & Play with USB!

- **Cortex-M3 V2**
  - 100+ MHz
  - LPC1700
  - Fastest Cortex M3 @ 120 MHz
DSP for Cortex M3

- LP176X optimized for 120MHz operation from flash
- Optimized M3 DSP library
  - FFT
  - Supports both 32 and 16 bit data lengths
  - Block sizes of 64, 256 and 1024
  - FIR and IIR filters
    - 16-bit single stage Biquad
    - 32-bit single stage Biquad
  - PID controller
  - Resonator function
  - Random number generator
  - Dot Product
  - Cross product of vectors
Why is NXP adopting Cortex-M4?

- Enables us to address new markets requiring DSC
  - Digital Signal (Processor + Micro) Controller
  - An intelligent blend of MCU and DSP features demanded
    - High performance mixed signal peripherals demand much more signal processing
    - Upper limits of bandwidth challenged in general purpose MCUs
    - Hard to learn/program technology in many licensable DSP cores

- Extend the Cortex-M portfolio to cover new markets
  - Cortex-M0 for mixed signal devices and state machine replacements
  - Cortex-M3 for mainstream 32-bit MCUs
  - Opportunity: High end MCUs and DSC market

- Introduce ARM strengths to DSC market
  - Very high energy efficiency - more processing in less milliwatts
  - Strong s/w ecosystem - easy to program and use
M4 Target Applications

- **Power Applications**
  - Motor control
  - Power conversion
  - Lighting
- **Audio**
- **Communications**
- **Peripherals are as important as the core**
Motor Control

- Sensor less Field Oriented control (FOC)
- **Peripherals**
  - ADC and PWM blocks must be synchronized
  - 12 bit Simultaneous sampling ADC. 3 current and 3 voltage
  - Center aligned with configurable dead time.
Power Conversion/Energy Metering

- Power Factor Correction
- Peak Power Tracking
- Peripherals
  - 12 bit Simultaneous sampling ADC
  - Center/edge aligned with configurable dead time
  - High resolution PWM
Audio

- Audio Processing
  - Filters/Mixing
  - Compression/ Decompression

- User Interface (With Voice and Audio Functionality)
  - Voice Recognition: Vocal Commands
  - Speech Synthesis: Spoken Output

- Peripherals
  - Ethernet
  - I2C
  - Multiple I2S
  - Audio PLL for USB streaming hardware sample rate conversion
Communications

- Power Line Communications
- Modems
- Peripherals
  - Ethernet
  - Multiple UARTS
  - SPI
  - ADC
  - PWM
Some Implementation Details

- NXP will include Cortex M4 Options
  - Floating Point Unit
  - Memory Protection Unit
  - WIC
  - Full Debug Options
    - ETM
    - DAP
    - FLASH Patch
- Min 150 MHz from Flash
Cortex-M4
Energy efficient, easy-to-use digital signal control

Shyam Sadasivan, ARM
General signal processing system

- Signal processing is almost exclusively in the digital domain.

- Digital Signal Processing heavily leans on MAC (Multiply Accumulate) operations inside large loops.

- Yet most algorithms have a lot of control code as well.

- Most embedded markets demanding an efficient blend of traditional RISC processing features with DSP capabilities.
Microcontroller characteristics

- Easy to use
- Efficient instruction set architecture
- Ultra low power—sleep modes etc
- Excellent interrupt control and latency
- Low cost debug and trace
- Memory protection—process separation
- Excellent software ecosystem
DSP characteristics

- Harvard architecture
- High performance MAC
- Saturating math
- SIMD instructions
- Barrel shifters
- Circular addressing
- Zero overhead loops
- Load/store in parallel with math
Digital signal control – efficient blend
Cortex-M processors

- Forget traditional 8/16/32-bit classifications
  - Seamless architecture across all applications
  - Every product optimised for ultra low power and ease of use

<table>
<thead>
<tr>
<th>Cortex-M0</th>
<th>Cortex-M3</th>
<th>Cortex-M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>“8/16-bit” applications</td>
<td>“16/32-bit” applications</td>
<td>“32-bit/DSC” applications</td>
</tr>
</tbody>
</table>

Binary and tool compatible
Cortex-M4 for digital signal control

- **Cortex-M4 processor**
  - Thumb-2 Technology
  - DSP and SIMD instructions
  - Single cycle MAC (Up to $32 \times 32 + 64 \rightarrow 64$)
  - Optional decoupled single precision FPU
  - Integrated configurable NVIC
  - Compatible with Cortex-M3

- **Microarchitecture**
  - 3-stage pipeline with branch speculation
  - 3x AHB-Lite Bus Interfaces

- **Configurable for ultra low power**
  - Deep Sleep Mode, Wakeup Interrupt Controller
  - Power down features for Floating Point Unit

- **Flexible configurations for wider applicability**
  - Configurable Interrupt Controller (1-240 Interrupts and Priorities)
  - Optional Memory Protection Unit
  - Optional Debug & Trace

Dotted boxes denote optional blocks
Powerful Cortex-M instruction set
Cortex-M4 SIMD instructions

- SIMD extensions perform multiple operations in one cycle

\[ Sum = Sum + (A \times C) + (B \times D) \]

- SIMD techniques operate with packed data
  - Algorithm developers can also reduce memory load instructions
## Powerful single cycle MAC – Cortex-M4

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>INSTRUCTION</th>
<th>CYCLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 \times 16 = 32$</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT</td>
<td>1</td>
</tr>
<tr>
<td>$16 \times 16 + 32 = 32$</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT</td>
<td>1</td>
</tr>
<tr>
<td>$16 \times 16 + 64 = 64$</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT</td>
<td>1</td>
</tr>
<tr>
<td>$16 \times 32 = 32$</td>
<td>SMULWB, SMULWT</td>
<td>1</td>
</tr>
<tr>
<td>$(16 \times 32) + 32 = 32$</td>
<td>SMLAWB, SMLAWT</td>
<td>1</td>
</tr>
<tr>
<td>$(16 \times 16) \pm (16 \times 16) = 32$</td>
<td>SMUAD, SMUADX, SMUSD, SMUSDX</td>
<td>1</td>
</tr>
<tr>
<td>$(16 \times 16) \pm (16 \times 16) + 32 = 32$</td>
<td>SMLAD, SMLADX, SMLSD, SMLSDX</td>
<td>1</td>
</tr>
<tr>
<td>$(16 \times 16) \pm (16 \times 16) + 64 = 64$</td>
<td>SMLALD, SMLALDX, SMLSLD, SMLSLDX</td>
<td>1</td>
</tr>
<tr>
<td>$32 \times 32 = 32$</td>
<td>MUL</td>
<td>1</td>
</tr>
<tr>
<td>$32 \pm (32 \times 32) = 32$</td>
<td>MLA, MLS</td>
<td>1</td>
</tr>
<tr>
<td>$32 \times 32 = 64$</td>
<td>SMULL, UMULL</td>
<td>1</td>
</tr>
<tr>
<td>$(32 \times 32) + 64 = 64$</td>
<td>SMLAL, UMLAL</td>
<td>1</td>
</tr>
<tr>
<td>$(32 \times 32) + 32 + 32 = 64$</td>
<td>UMAAL</td>
<td>1</td>
</tr>
<tr>
<td>$32 \pm (32 \times 32) = 32$ (upper)</td>
<td>SMMLA, SMMLAR, SMMLS, SMMLSR</td>
<td>1</td>
</tr>
<tr>
<td>$(32 \times 32) = 32$ (upper)</td>
<td>SMMUL, SMMULR</td>
<td>1</td>
</tr>
</tbody>
</table>
Cortex-M 32-bit functions cycle count

- FIR
- IIR
- Complex FFT
- PID
- Matrix Addition
- Matrix Subtraction
- Matrix Multiplication

Comparisons between Cortex-M3 and Cortex-M4.
Floating point hardware capabilities

- IEEE 754 standard compliance
- Floating point hardware improves performance
- Single-precision floating point math key to some algorithms
  - Add, subtract, multiply, divide, MAC and square root
  - Fused MAC – provides higher precision

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CYCLE COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Subtract</td>
<td>1</td>
</tr>
<tr>
<td>Divide</td>
<td>14</td>
</tr>
<tr>
<td>Multiply</td>
<td>1</td>
</tr>
<tr>
<td>Multiply Accumulate (MAC)</td>
<td>3</td>
</tr>
<tr>
<td>Fused MAC</td>
<td>3</td>
</tr>
<tr>
<td>Square Root</td>
<td>14</td>
</tr>
</tbody>
</table>
Signal processing optimization techniques on Cortex-M processors

Paul Beckmann
www.dspconcepts.com
## Why signal processing in digital domain?

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced algorithms</td>
<td>Bandwidth limited by sample rate</td>
</tr>
<tr>
<td>Precision and accuracy</td>
<td>Limited dynamic range</td>
</tr>
<tr>
<td>Stability</td>
<td>Quantization noise</td>
</tr>
<tr>
<td>Software based design leads to reconfigurability</td>
<td>Higher power consumption</td>
</tr>
<tr>
<td>Increasing capabilities of processors opens up new applications on the high end</td>
<td>Cost of analog components continues to fall</td>
</tr>
<tr>
<td>Cost of processors continues to fall and opens up new applications on the low end</td>
<td></td>
</tr>
</tbody>
</table>
Main DSP operations and applications

- Finite impulse response (FIR) filters
  - Data communications
  - Echo cancellation (adaptive versions)
  - Smoothing data

\[ y[n] = \sum_{k=0}^{N-1} h[k] x[n-k] \]

- Infinite impulse response (IIR) filters
  - Audio equalization
  - Motor control

\[ y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + a_1 y[n-1] + a_2 y[n-2] \]

- Fast Fourier transforms (FFT)
  - Audio compression
  - Spread spectrum communication
  - Noise removal

\[ Y[k_1] = X[k_1] + X[k_2] \]
\[ Y[k_2] = (X[k_1] - X[k_2])e^{-j\omega} \]

Most operations are dominated by Multiply-Accumulates (MACs)
How to program – assembly or C?

Assembly?
+ Can result in highest performance
  – Difficult learning curve, longer development cycles
  – Code reuse difficult – not portable

C?
+ Easy to write and maintain code, faster development cycles
+ Code reuse possible, using third party software is easier
+ Intrinsics provide direct access to certain processor features
  – Highest performance might not be possible
  – Get to know your compiler!

C is definitely the preferred approach!
2nd order IIR Filter – AKA “Biquad”

- Commonly used for control and audio filtering
- Implemented using a difference equation.
- Direct Form 1 structure is the most numerically robust - shown below
- Has 5 coefficients and 4 state variables
- Coefficients determine the response of the filter (lowpass, highpass, etc.) and may be computed in a number of different ways
  - Simple design equations running on the MCU
  - External tools such as MATLAB

\[
y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2]
\]
Cortex-M Biquad implementation

### Cortex-M3

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
<th>Cortex-M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>xN = *x++;</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>yN = xN * b0;</td>
<td>3-7</td>
<td>1</td>
</tr>
<tr>
<td>yN += xNm1 * b1;</td>
<td>3-7</td>
<td>1</td>
</tr>
<tr>
<td>yN += xNm2 * b2;</td>
<td>3-7</td>
<td>1</td>
</tr>
<tr>
<td>yN -= yNm1 * a1;</td>
<td>3-7</td>
<td>1</td>
</tr>
<tr>
<td>yN -= yNm2 * a2;</td>
<td>3-7</td>
<td>1</td>
</tr>
<tr>
<td>*y++ = yN;</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>xNm2 = xNm1;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>xNm1 = xN;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>yNm2 = yNm1;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>yNm1 = yN;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decrement loop counter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

### Cortex-M4

\[ y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2] \]

- 27-47 cycles
- 16 cycles

- Only looking at the inner loop, making these assumptions
  - Function operates on a block of samples
  - Coefficients b0, b1, b2, a1, and a2 are in registers
  - Previous states, x[n-1], x[n-2], y[n-1], and y[n-2] are in registers

*Cortex-M4 40-65% higher performance!
Optimize by unrolling the loop by 3

\[
\begin{align*}
x_0 &= *x++; \quad (2 \text{ cycles}) \\
y_0 &= x_0 \times b_0; \quad \text{(1 cycle)} \\
y_0 &= y_0 + x_1 \times b_1; \quad \text{(1 cycle)} \\
y_0 &= y_0 + x_2 \times b_2; \quad \text{(1 cycle)} \\
y_0 &= y_0 - y_1 \times a_1; \quad \text{(1 cycle)} \\
y_0 &= y_0 - y_2 \times a_2; \quad \text{(1 cycle)} \\
*y++ &= y_0; \quad (2 \text{ cycles}) \\
\end{align*}
\]

\[
\begin{align*}
x_2 &= *x++; \quad (2 \text{ cycles}) \\
y_2 &= x_2 \times b_0; \quad \text{(1 cycle)} \\
y_2 &= y_2 + x_0 \times b_1; \quad \text{(1 cycle)} \\
y_2 &= y_2 + x_1 \times b_2; \quad \text{(1 cycle)} \\
y_2 &= y_2 - y_0 \times a_1; \quad \text{(1 cycle)} \\
y_2 &= y_2 - y_1 \times a_2; \quad \text{(1 cycle)} \\
*y++ &= y_2; \quad (2 \text{ cycles}) \\
\end{align*}
\]

\[
\begin{align*}
x_1 &= *x++; \quad (2 \text{ cycles}) \\
y_1 &= x_1 \times b_0; \quad \text{(1 cycle)} \\
y_1 &= y_1 + x_2 \times b_1; \quad \text{(1 cycle)} \\
y_1 &= y_1 + x_0 \times b_2; \quad \text{(1 cycle)} \\
y_1 &= y_1 - y_2 \times a_1; \quad \text{(1 cycle)} \\
y_1 &= y_1 - y_0 \times a_2; \quad \text{(1 cycle)} \\
*y++ &= y_1; \quad (2 \text{ cycles}) \\
\end{align*}
\]

- Reduces loop overhead
- Eliminates the need to shift down state variables
- 30 cycles on Cortex-M4 to for 3 output samples
  \[\rightarrow 10 \text{ cycles per sample}\]
FIR Filter

- Occurs frequently in communications, audio, and video applications
- A filter of length N requires
  - N coefficients $h[0], h[1], \ldots, h[N-1]$
  - N state variables $x[n], x[n-1], \ldots, x[n-(N-1)]$
  - N multiply accumulates
- Classic function for which DSPs were designed for
FIR filter standard C implementation

```c
void fir(q31_t *in, q31_t *out, q31_t *coeffs, int *stateIndexPtr,
         int filtLen, int blockSize)
{
    int sample;
    int k;
    q31_t sum;
    int stateIndex = *stateIndexPtr;

    for(sample=0; sample < blockSize; sample++)
    {
        state[stateIndex++] = in[sample];
        sum=0;
        for(k=0;k<filtLen;k++)
        {
            sum += coeffs[k] * state[stateIndex];
            stateIndex--;
            if (stateIndex < 0)
            {
                stateIndex = filtLen-1;
            }
        }
        out[sample]=sum;
    }
    *stateIndexPtr = stateIndex;
}
```

- Block based processing
- Inner loop consists of:
  - Dual memory fetches
  - MAC
  - Pointer updates with circular addressing
Circular addressing on DSPs

- Data in the delay chain is right shifted every sample. This is very wasteful. How can we avoid this?
- DSPs typically utilize circular addressing to avoid this data movement

![Diagram showing linear and circular addressing of coefficients and states](image)
Circular addressing on Cortex-M

- Cortex-M processors do not have circular addressing
- Two alternatives on Cortex-M processors
  - Create a circular buffer of length \( N + \text{blockSize} \) and shift this once per block.
  - Break the inner loop into two separate portions: pre-wrap and post-wrap. (Use if \( N \gg \text{blockSize} \))

![Diagram showing circular addressing and buffer](attachment:image.png)
Caching of coefficients and states

- The FIR filter is extremely memory intensive.
- Computing a single output sample requires N coefficient and N state variable fetches.
- These memory accesses dominate the computation:
  - 2 consecutive loads = 4 cycles on Cortex-M3, 3 cycles on Cortex-M4
  - MAC = 3-7 cycles on Cortex-M3, 1 cycle on Cortex-M4
- When operating on a block of data, memory bandwidth can be reduced by simultaneously computing multiple outputs and caching several coefficients and state variables.
Use SIMD for 8- or 16-bit data types

- Many applications use 8- or 16-bit data types
- FIR filters insensitive to coefficient quantization
- Ideal application of SIMD!
  - 16-bit data yields a 2x speed improvement over 32-bit
  - 8-bit data yields a 4x speed improvement
Cortex-M4 FIR example with intrinsics

```c
sample = blockSize/4;
do
{
  sum0 = sum1 = sum2 = sum3 = 0;
  statePtr = stateBasePtr;
  coeffPtr = (q31_t *)(S->coeffs);
  x0 = *(q31_t *)(statePtr++);
  x1 = *(q31_t *)(statePtr++);
  i = numTaps>>2;
  do
  {
    c0 = *(coeffPtr++);
    x2 = *(q31_t *)(statePtr++);
    x3 = *(q31_t *)(statePtr++);
    sum0  += __smlad(x0, c0, sum0);
    sum1  += __smlad(x1, c0, sum1);
    sum2  += __smlad(x2, c0, sum2);
    sum3  += __smlad(x3, c0, sum3);
    c0 = *(coeffPtr++);
    x0 = *(q31_t *)(statePtr++);
    x1 = *(q31_t *)(statePtr++);
    sum0 += __smlad(x2, c0, sum0);
    sum1 += __smlad(x3, c0, sum1);
    sum2 += __smlad(x0, c0, sum2);
    sum3 += __smlad(x1, c0, sum3);
  } while(--i);
  *pDst++ = (q15_t) (sum0>>15);
  *pDst++ = (q15_t) (sum1>>15);
  *pDst++ = (q15_t) (sum2>>15);
  *pDst++ = (q15_t) (sum3>>15);
  stateBasePtr= stateBasePtr + 4;
} while(--sample);
```

Uses loop unrolling, SIMD intrinsics, caching of states and coefficients, and works around circular addressing by using a large state buffer.
Cortex-M4 FIR example with intrinsics

```c
sample = blockSize/4;
do {
    sum0 = sum1 = sum2 = sum3 = 0;
    statePtr = stateBasePtr;
    coeffPtr = (q31_t *)(S->coeffs);
    x0 = *(q31_t *)(statePtr++);
    x1 = *(q31_t *)(statePtr++);
    i = numTaps>>2;
    do {
        c0 = *(coeffPtr++);
        x2 = *(q31_t *)(statePtr++);
        x3 = *(q31_t *)(statePtr++);
        sum0  += __smlad(x0, c0, sum0);
        sum1  += __smlad(x1, c0, sum1);
        sum2  += __smlad(x2, c0, sum2);
        sum3  += __smlad(x3, c0, sum3);
        c0 = *(coeffPtr++);
        x0 = *(q31_t *)(statePtr++);
        x1 = *(q31_t *)(statePtr++);
        sum0 += __smlad(x2, c0, sum0);
        sum1 += __smlad(x3, c0, sum1);
        sum2 += __smlad(x0, c0, sum2);
        sum3 += __smlad(x1, c0, sum3);
    } while(--i);
    *pDst++ = (q15_t) (sum0>>15);
    *pDst++ = (q15_t) (sum1>>15);
    *pDst++ = (q15_t) (sum2>>15);
    *pDst++ = (q15_t) (sum3>>15);
    stateBasePtr = stateBasePtr + 4;
} while(--sample);
```

```
x0 = [ x[0] | x[1] ]
c0 = [ c[0] | c[1] ]
```

Inner loop is 22 cycles for a total of 16, 16-bit MACs.

Only 1.375 cycles per filter tap!
DSP example – MP3 audio playback

- General Purpose MCUs
- Discrete DSPs
- Cortex-M4
- Specialised Audio DSPs

MHz bandwidth requirement for MP3 decode

Cortex-M4 approaches specialized audio DSP performance!
DSP example – graphic equalizer

Real-time Demo
- 7 band parametric EQ
- 32-bit precision
- Stereo processing
- 48 kHz sample rate

Performance
- Cortex-M3 57 MHz
- Cortex-M4 13.2 MHz