Designing with the Cortex-M4

Kishore Dasari
Managing Director/GM
www dspconcepts.com
Agenda

• Introduction

• Review of signal processing design

• Software building blocks for signal processing

• Optimization techniques

• Conclusion

• Quick Demos
Who is DSP Concepts?

- an engineering services company specializing in embedded audio product and technology development
Market Size vs. Sophistication of Audio Processing

Markets DSP Concepts has experience in

Markets to focus on

Complexity of Audio Processing

Low

Medium

High

Annual unit sales

1k 10k 100k 1M 10M 100M 1B

Audio is Primary

Broadcast

Aircraft Radio

PA & Evacuation

Professional

Aftermarket Auto Amplifier

Guitar Pedal

OEM Auto Amplifier

Home Theater

Audio is Secondary

PCs

Game Consoles

Head unit

Set top box

Cameras

iPod

Cell phone

iPod

With Speakers

Only decoding
Digital signal control - blend

**MCU**
- Low costs
- Ease of use
- C Programming
- Interrupt handling
- Ultra low power

**Digital Signal Controller**

**DSP**
- Harvard architecture
- Single cycle MAC
- Floating Point
- Barrel shifter
Mathematical details

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

- **IIR or recursive filter**
  \[ y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + a_1 y[n-1] + a_2 y[n-2] \]

- **FFT Butterfly (radix-2)**
  \[ Y[k_1] = X[k_1] + X[k_2] \]
  \[ Y[k_2] = (X[k_1] - X[k_2]) e^{-j\omega} \]

Most operations are dominated by MACs
These can be on 8, 16 or 32 bit operations
## Powerful MAC instructions

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

All the above operations are single cycle on the Cortex-M4 processor
Floating point hardware

- IEEE 754 standard compliance

- 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>SP FP OPERATION</th>
<th>CYCLE COUNT USING FPU</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>
Design Example

- 7-band Graphic Equalizer
  - Cortex-M3 LPC1768 running at 120MHz
  - Cortex-M4 running at 120MHz
- Designed using DSP Concept’s Audio Weaver development environment
  - a graphical drag-and-drop design environment and a set of optimized audio processing libraries.
DSP example - graphic equalizer
Audio Weaver signal flow

Real-time Demo

- 7 band parametric EQ
- 32-bit precision
- Stereo processing
- 48 kHz sample rate
Results

Performance
- Cortex-M3 needed 1291 cycles (47.4% processor loading)
- Cortex-M4 needed only 299 cycles (11% processor loading).
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!
Circular Addressing

- Data in the delay chain is right shifted every sample. This is very wasteful. How can we avoid this?
- Circular addressing avoids this data movement

Linear addressing of coefficients.

Circular addressing of states
FIR Filter Standard C Code

```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
FIR Filter DSP Code

- 32-bit DSP processor assembly code
- Only the inner loop is shown, executes in a single cycle
- Optimized assembly code, cannot be achieved in C

```
lcntr=r2, do FIRLoop until lce;
FIRLoop: f12=f0*f4, f8=f8+f12, f4=dm(i1,m4), f0=pm(i12,m12);
```
Cortex-M inner loop

```c
for(k=0;k<filtLen;k++)
{
    sum += coeffs[k] * state[stateIndex];
    stateIndex--;
    if (stateIndex < 0)
    {
        stateIndex = filtLen-1;
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch coeffs[k]</td>
<td>2</td>
</tr>
<tr>
<td>Fetch state[stateIndex]</td>
<td>1</td>
</tr>
<tr>
<td>MAC</td>
<td>1</td>
</tr>
<tr>
<td>stateIndex--</td>
<td>1</td>
</tr>
<tr>
<td>Circular wrap</td>
<td>4</td>
</tr>
<tr>
<td>Loop overhead</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>--------</td>
</tr>
<tr>
<td>Total</td>
<td>12</td>
</tr>
</tbody>
</table>

Even though the MAC executes in 1 cycle, there is overhead compared to a DSP.

How can this be improved on the Cortex-M4?
Optimization strategies

• Circular addressing alternatives

• Loop unrolling

• Caching of intermediate variables

• Extensive use of SIMD and intrinsics
Circular Buffering alternative

- Create a buffer of length $N + \text{blockSize} - 1$ and shift this once per block

|--------|--------|--------|--------|--------|--------|--------|--------|--------|

Copy old samples

Shift in 4 new samples

|--------|--------|--------|--------|--------|--------|
Circular Buffering alternative

- Create a circular buffer of length \( N + \text{blockSize} - 1 \) and shift this once per block
- Example. \( N = 6 \), \( \text{blockSize} = 4 \). Size of state buffer = 9.
Circular Buffering alternative

- Create a circular buffer of length $N + \text{blockSize} - 1$ and shift this once per block
Cortex-M4 code with change

for (k = 0; k < filtLen; k++)
{
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch coeffs[k]</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Fetch state[stateIndex]</td>
<td>1 cycle</td>
</tr>
<tr>
<td>MAC</td>
<td>1 cycle</td>
</tr>
<tr>
<td>stateIndex++</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Loop overhead</td>
<td>3 cycles</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>8 cycles</strong></td>
</tr>
</tbody>
</table>
Improvement in performance

- DSP assembly code = 1 cycle
- Cortex-M4 standard C code takes 12 cycles
  - Using circular addressing alternative = 8 cycles

33% better but still not comparable to the DSP

Lets try loop unrolling
Loop unrolling

- This is an efficient language-independent optimization technique and makes up for the lack of a zero overhead loop on the Cortex-M4.

- There is overhead inherent in every loop for checking the loop counter and incrementing it for every iteration (3 cycles on the Cortex-M.)

- Loop unrolling processes ‘n’ loop indexes in one loop iteration, reducing the overhead by ‘n’ times.
Unroll Inner Loop by 4

for (k=0; k<filtLen; k++)
{
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
}

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch coeffs[k]</td>
<td>2 x 4</td>
<td>8</td>
</tr>
<tr>
<td>Fetch state[stateIndex]</td>
<td>1 x 4</td>
<td>4</td>
</tr>
<tr>
<td>MAC</td>
<td>1 x 4</td>
<td>4</td>
</tr>
<tr>
<td>stateIndex++</td>
<td>1 x 4</td>
<td>4</td>
</tr>
<tr>
<td>Loop overhead</td>
<td>3 x 1</td>
<td>3</td>
</tr>
</tbody>
</table>

------------

Total: 23 cycles for 4 taps
= 5.75 cycles per tap
Improvement in performance

- DSP assembly code = 1 cycle
- Cortex-M4 standard C code takes 12 cycles
  - Using circular addressing alternative = 8 cycles
  - After loop unrolling < 6 cycles

25% further improvement
But a large gap still exists

Let's try SIMD
Many image and video processing, and communications applications use 8- or 16-bit data types.

SIMD speeds these up
- 16-bit data yields a 2x speed improvement over 32-bit
- 8-bit data yields a 4x speed improvement

Access to SIMD is via compiler intrinsics

Example dual 16-bit MAC
- \texttt{SUM=\_SMLALD(C, S, SUM)}
Data organization with SIMD

- 16-bit example
- Access two neighbouring values using a single 32-bit memory read

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inner Loop with 16-bit SIMD

```c
filtLen = filtLen << 2;
for(k = 0; k < filtLen; k++)
{
    c = *coeffs++; // 2 cycles
    s = *state++; // 1 cycle
    sum = __SMLALD(c, s, sum); // 1 cycle
    c = *coeffs++; // 2 cycles
    s = *state++; // 1 cycle
    sum = __SMLALD(c, s, sum); // 1 cycle
    c = *coeffs++; // 2 cycles
    s = *state++; // 1 cycle
    sum = __SMLALD(c, s, sum); // 1 cycle
    c = *coeffs++; // 2 cycles
    s = *state++; // 1 cycle
    sum = __SMLALD(c, s, sum); // 1 cycle
}
```

19 cycles total. Computes 8 MACs

2.375 cycles per filter tap
Improvement in performance

- DSP assembly code = 1 cycle

- Cortex-M4 standard C code takes 12 cycles
  - Using circular addressing alternative = 8 cycles
  - After loop unrolling < 6 cycles
  - After using SIMD instructions < 2.5 cycles

That’s much better!
But is there anything more?

One more idea left
Caching Intermediate Values

- FIR filter is extremely memory intensive. 12 out of 19 cycles in the last code portion deal with memory accesses
  - 2 consecutive loads take
    - 4 cycles on Cortex-M3, 3 cycles on Cortex-M4
  - MAC takes
    - 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
Data Organization with Caching

Compute 4 Outputs Simultaneously:

\[
\begin{align*}
\text{sum0} &= \_\_\text{SMLALD}(x_0, c_0, \text{sum0}) \\
\text{sum1} &= \_\_\text{SMLALD}(x_1, c_0, \text{sum1}) \\
\text{sum2} &= \_\_\text{SMLALD}(x_2, c_0, \text{sum2}) \\
\text{sum3} &= \_\_\text{SMLALD}(x_3, c_0, \text{sum3})
\end{align*}
\]
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 = __SMLALD(x0, c0, sum0);
      sum1 = __SMLALD(x1, c0, sum1);
      sum2 = __SMLALD(x2, c0, sum2);
      sum3 = __SMLALD(x3, c0, sum3);
      c0 = *(coeffPtr++);
      x0 = *(q31_t *) (statePtr++);
      x1 = *(q31_t *) (statePtr++);
      sum0 = __SMLALD(x0, c0, sum0);
      sum1 = __SMLALD(x1, c0, sum1);
      sum2 = __SMLALD(x2, c0, sum2);
      sum3 = __SMLALD(x3, 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);
FIR Application - use case
Cortex-M4 FIR performance

- DSP assembly code = 1 cycle
- Cortex-M4 standard C code takes 12 cycles
  - Using circular addressing alternative = 8 cycles
  - After loop unrolling < 6 cycles
  - After using SIMD instructions < 2.5 cycles
  - After caching intermediate values ~ 1.6 cycles

Cortex-M4 C code now comparable in performance
Summary of optimizations

• Basic Cortex-M4 C code quite reasonable performance for simple algorithms

• Through simple optimizations, you can get to high performance on the Cortex-M4

• You DO NOT have to write Cortex-M4 assembly, all optimizations can be done completely in C
CMSIS DSP library snapshot

- Basic math – vector mathematics
- Fast math – sin, cos, sqrt etc
- Interpolation – linear, bilinear
- Complex math
- Statistics – max, min, RMS etc
- Filtering – IIR, FIR, LMS etc
- Transforms – FFT(real and complex), Cosine transform etc
- Matrix functions
- PID Controller, Clarke and Park transforms
- Support functions – copy/fill arrays, data type conversions etc

Variants for functions across q7, q15, q31 and f32 data types
Cortex-M4 approaches specialized audio DSP performance!
Quick Demos