Designing with the Cortex-M4

Kishore Dasari
Managing Director/GM
www.dspconcepts.com

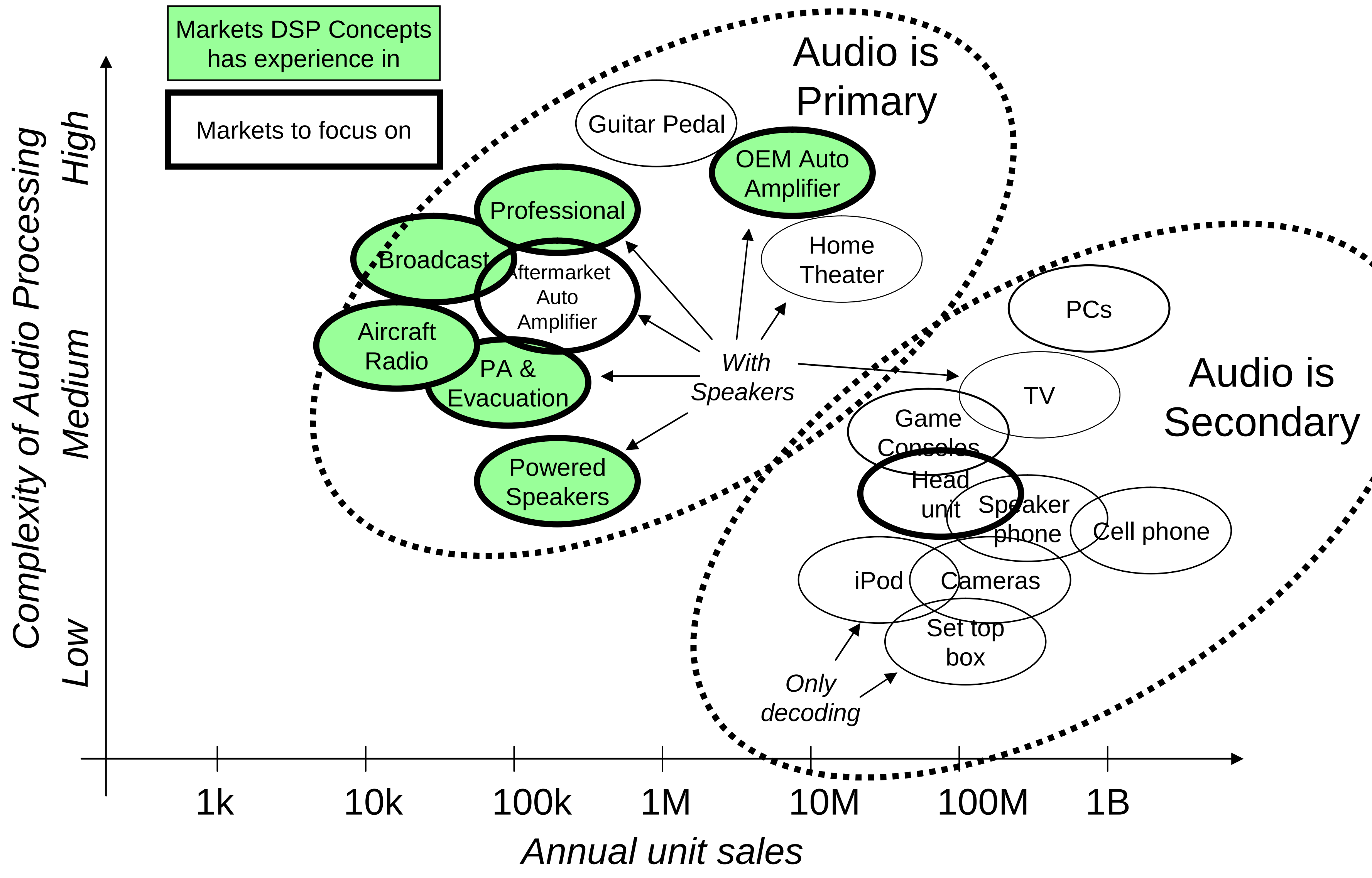
Agenda

- Introduction
- Review of signal processing design
- Software building blocks for signal processing
- Optimization techniques
- Conclusion
- Quick Demos

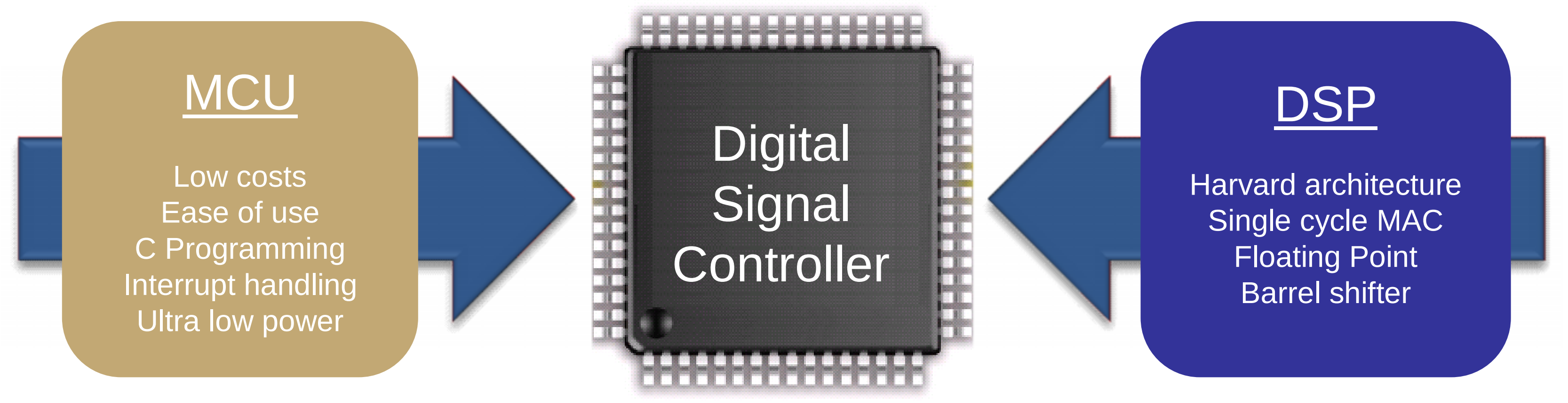
Who is DSP Concepts?

- an engineering services company specializing in embedded audio product and technology development

Market Size vs. Sophistication of Audio Processing



Digital signal control - blend



Mathematical details

- FIR Filter

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

- IIR or recursive filter

$$y[n]=b_0x[n]+b_1x[n-1]+b_2x[n-2]$$
$$+a_1y[n-1]+a_2y[n-2]$$

- FFT Butterfly (radix-2)

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

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

Powerful MAC instructions

OPERATION	INSTRUCTION
16 x 16 = 32	SMULBB, SMULBT, SMULTB, SMULTT
16 x 16 + 32 = 32	SMLABB, SMLABT, SMLATB, SMLATT
16 x 16 + 64 = 64	SMLALBB, SMLALBT, SMLALTB, SMLALTT
16 x 32 = 32	SMULWB, SMULWT
(16 x 32) + 32 = 32	SMLAWB, SMLAWT
(16 x 16) ± (16 x 16) = 32	SMUAD, SMUADX, SMUSD, SMUSDX
(16 x 16) ± (16 x 16) + 32 = 32	SMLAD, SMLADX, SMLSD, SMLSDX
(16 x 16) ± (16 x 16) + 64 = 64	SMLALD, SMLALDX, SMLSLD, SMLSLDX
32 x 32 = 32	MUL
32 ± (32 x 32) = 32	MLA, MLS
32 x 32 = 64	SMULL, UMULL
(32 x 32) + 64 = 64	SMLAL, UMLAL
(32 x 32) + 32 + 32 = 64	UMAAL
32 ± (32 x 32) = 32 (upper)	SMMLA, SMMLAR, SMMLS, SMMLSR
(32 x 32) = 32 (upper)	SMMUL, SMMULR

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

Floating point hardware

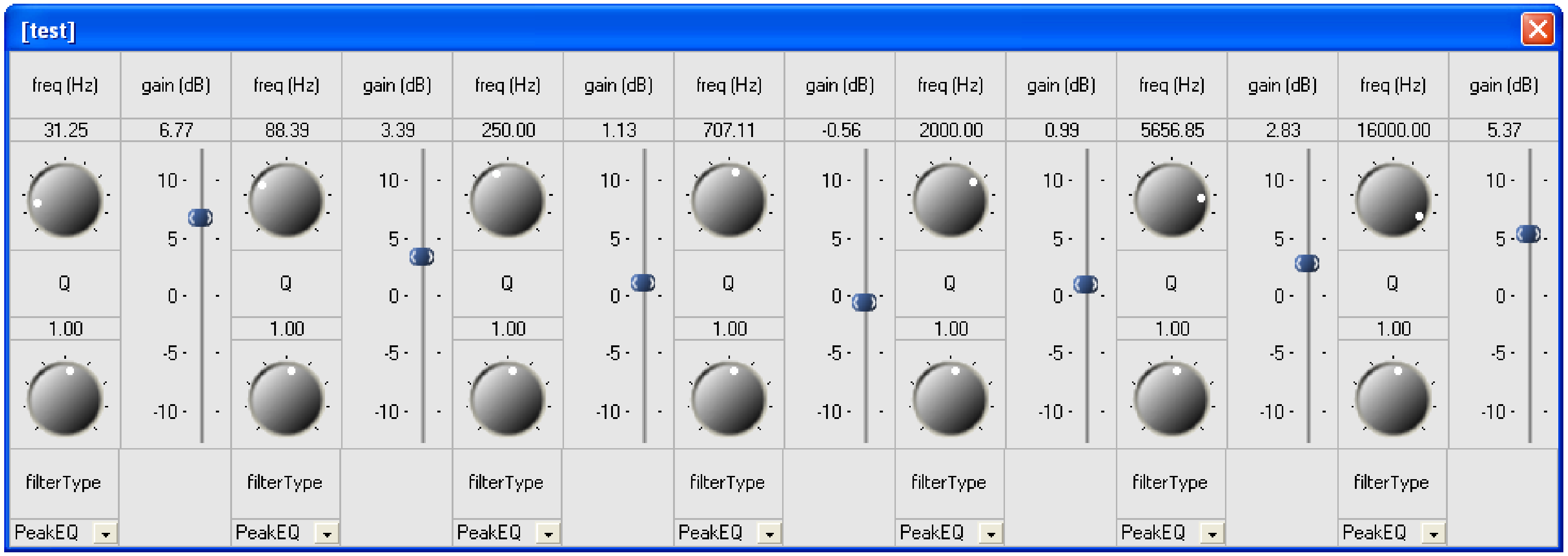
- IEEE 754 standard compliance
- Single-precision floating point math key to some algorithms
  - Add, subtract, multiply, divide, MAC and square root
  - Fused MAC – provides higher precision

SP FP OPERATION	CYCLE COUNT USING FPU
Add/Subtract	1
Divide	14
Multiply	1
Multiply Accumulate (MAC)	3
Fused MAC	3
Square Root	14

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.

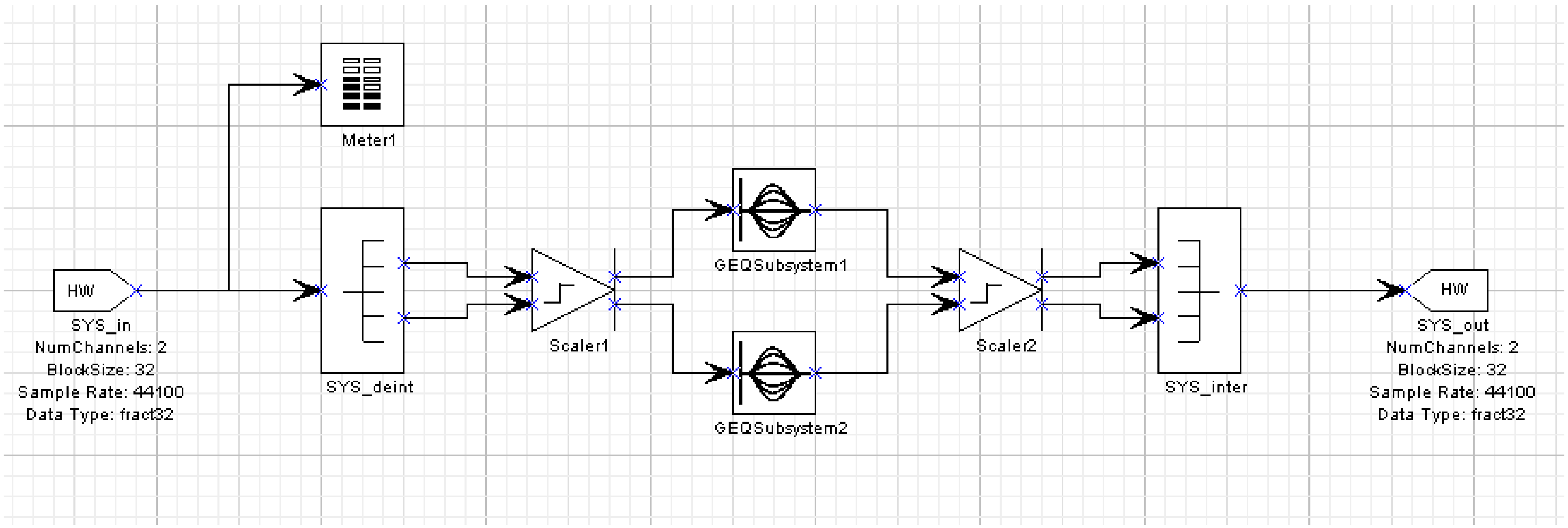
DSP example – graphic equalizer



[test]

freq (Hz)	gain (dB)	freq (Hz)	gain (dB)	freq (Hz)	gain (dB)	freq (Hz)	gain (dB)	freq (Hz)	gain (dB)	freq (Hz)	gain (dB)	freq (Hz)	gain (dB)
31.25	6.77	88.39	3.39	250.00	1.13	707.11	-0.56	2000.00	0.99	5656.85	2.83	16000.00	5.37
Q		Q		Q		Q		Q		Q		Q	
1.00		1.00		1.00		1.00		1.00		1.00		1.00	
filterType		filterType		filterType		filterType		filterType		filterType		filterType	
PeakEQ		PeakEQ		PeakEQ		PeakEQ		PeakEQ		PeakEQ		PeakEQ	

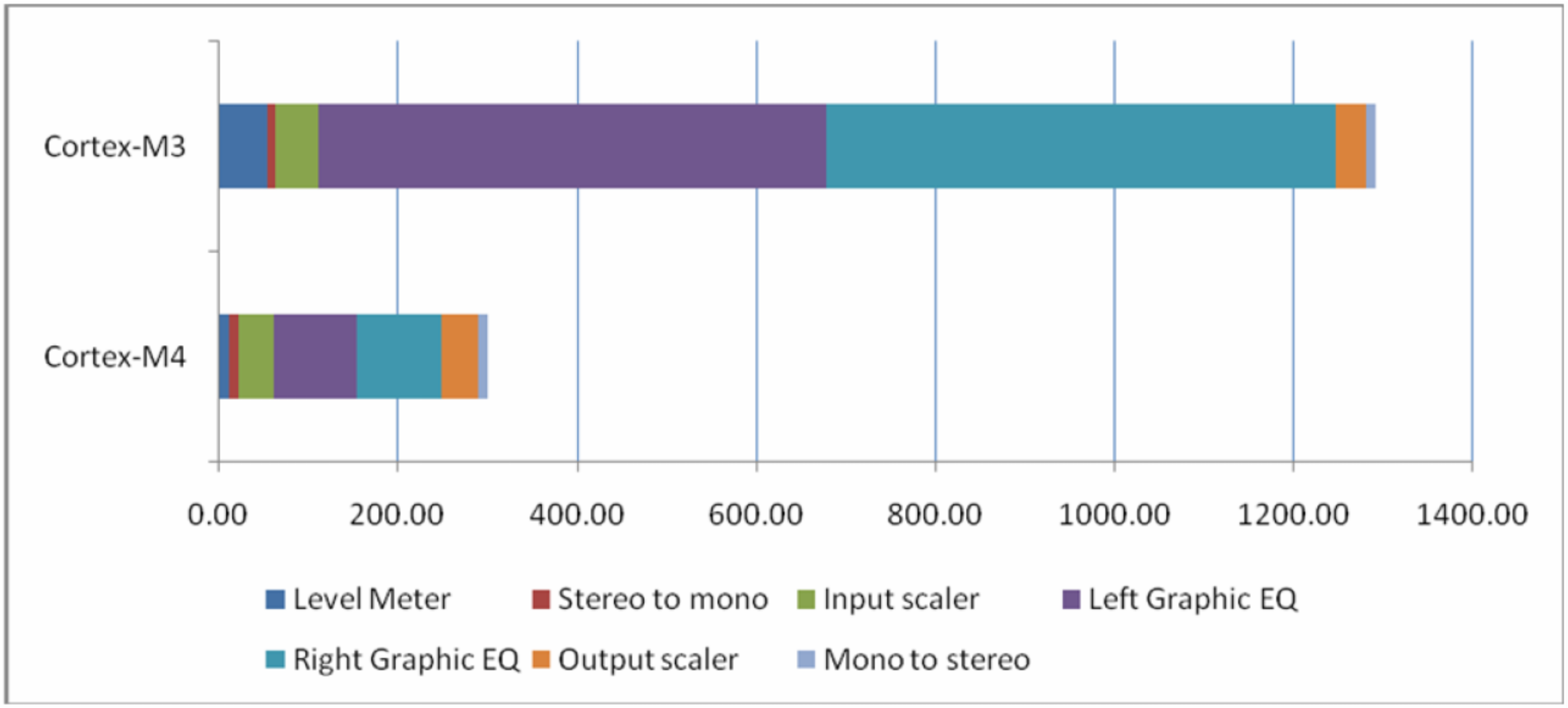
Audio Weaver signal flow



<u>Real-time Demo</u>

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

Results



<u>Performance</u>

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

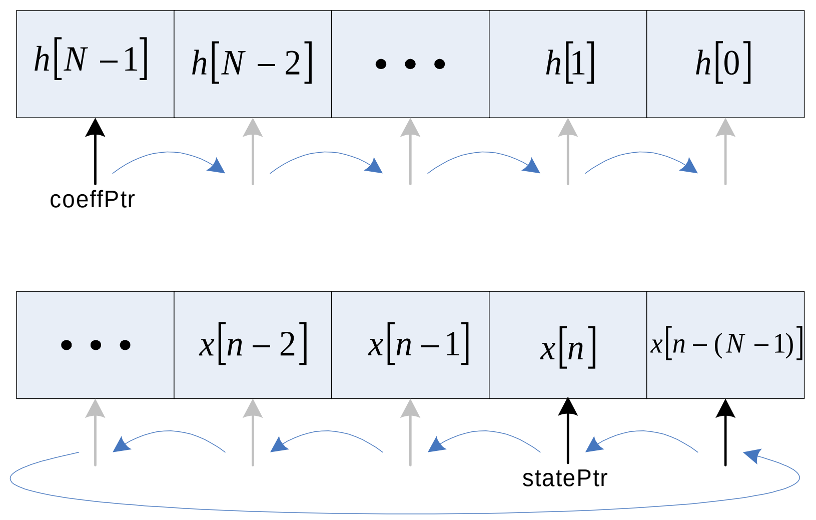
How to program – assembly or C?

- 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

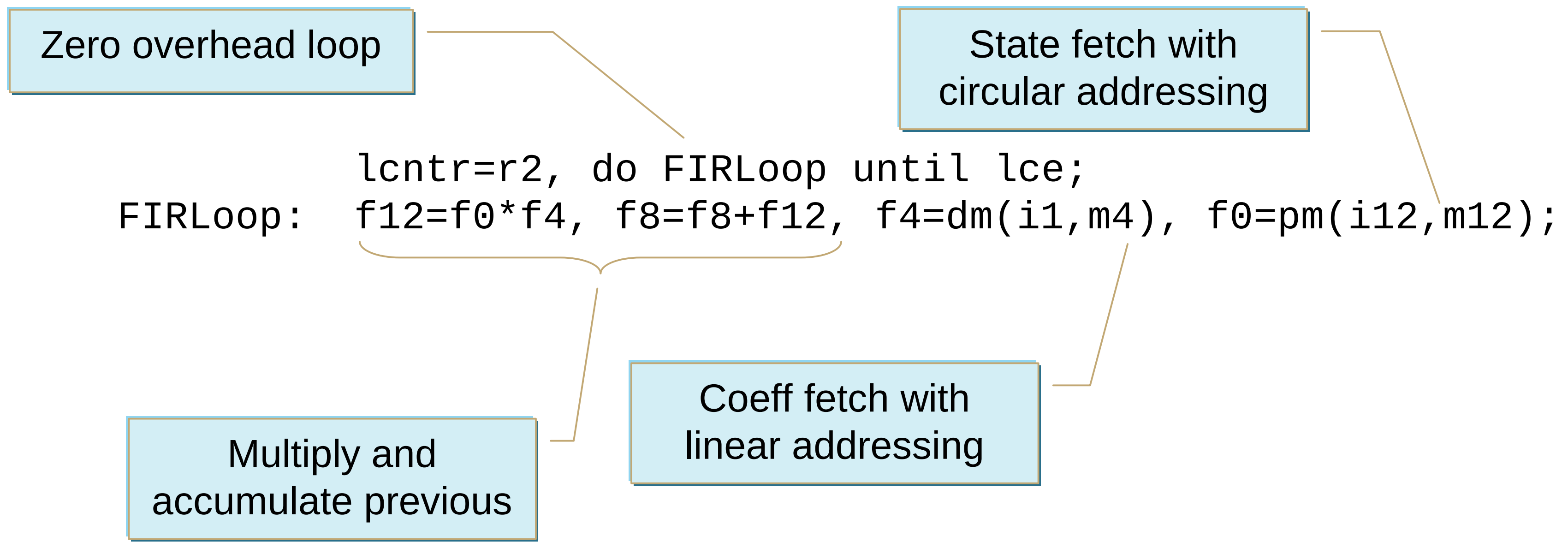
FIR Filter Standard C Code

```
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);
```

Zero overhead loop
State fetch with circular addressing
Multiply and accumulate previous
Coeff fetch with linear addressing

Cortex-M inner loop

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

Fetch coeffs[k]	2 cycles
Fetch state[stateIndex]	1 cycle
MAC	1 cycle
stateIndex--	1 cycle
Circular wrap	4 cycles
Loop overhead	3 cycles

Total	12 cycles

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

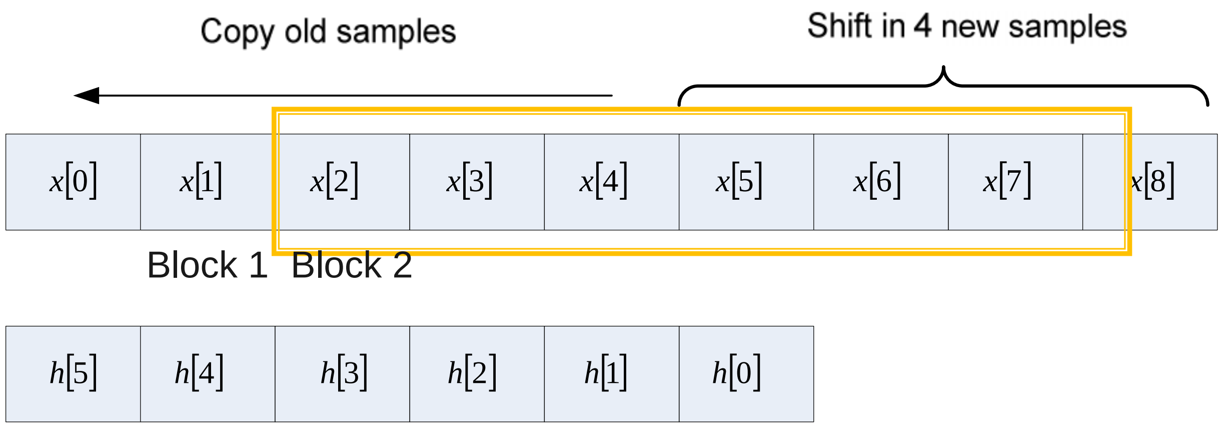
How can this be improved on the Cortex−M4 ?

Optimization strategies

- Circular addressing alternatives
- Loop unrolling
- Caching of intermediate variables
- Extensive use of SIMD and intrinsics

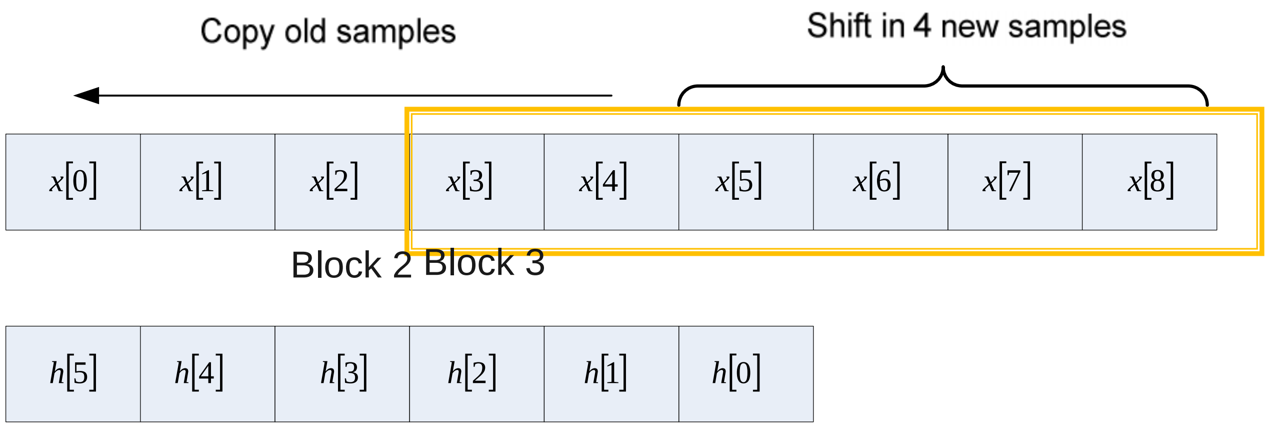
Circular Buffering alternative

- Create a buffer of length N + blockSize−1 and shift this once per block
- Example. N = 6, blockSize = 4. Size of state buffer = 9.



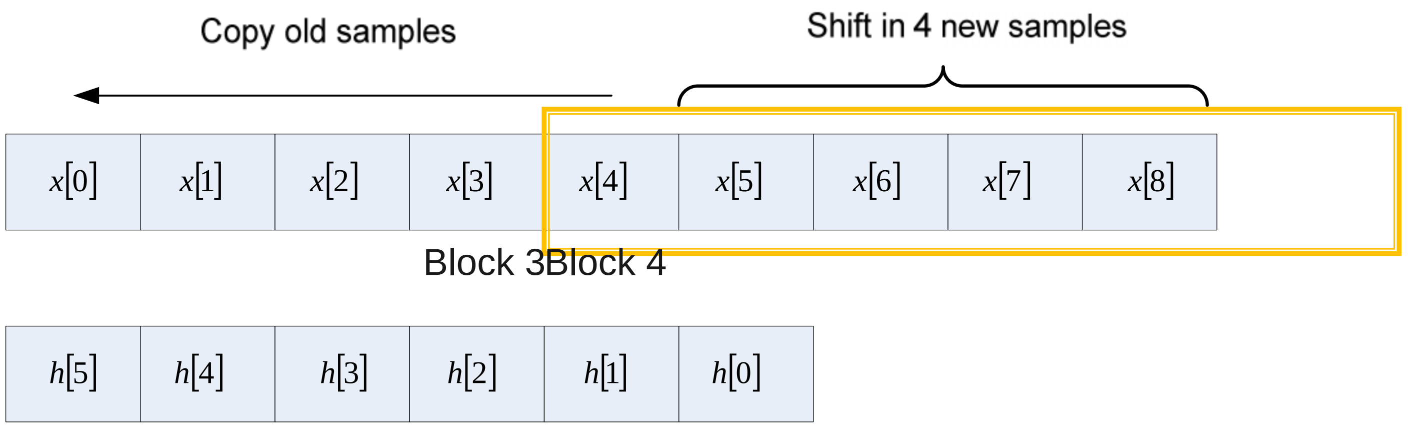
Circular Buffering alternative

- Create a circular buffer of length N + blockSize−1 and shift this once per block
- Example. N = 6, blockSize = 4. Size of state buffer = 9.



Circular Buffering alternative

- Create a circular buffer of length N + blockSize−1 and shift this once per block
- Example. N = 6, blockSize = 4. Size of state buffer = 9.



Cortex-M4 code with change

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

Fetch coeffs[k]	2 cycles
Fetch state[stateIndex]	1 cycle
MAC	1 cycle
stateIndex++	1 cycle
Loop overhead	3 cycles

Total	8 cycles

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++;
}
```

Fetch coeffs[k]	2 x 4	= 8 cycles
Fetch state[stateIndex]	1 x 4	= 4 cycles
MAC	1 x 4	= 4 cycles
stateIndex++	1 x 4	= 4 cycles
Loop overhead	3 x 1	= 3 cycles

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

Lets try SIMD

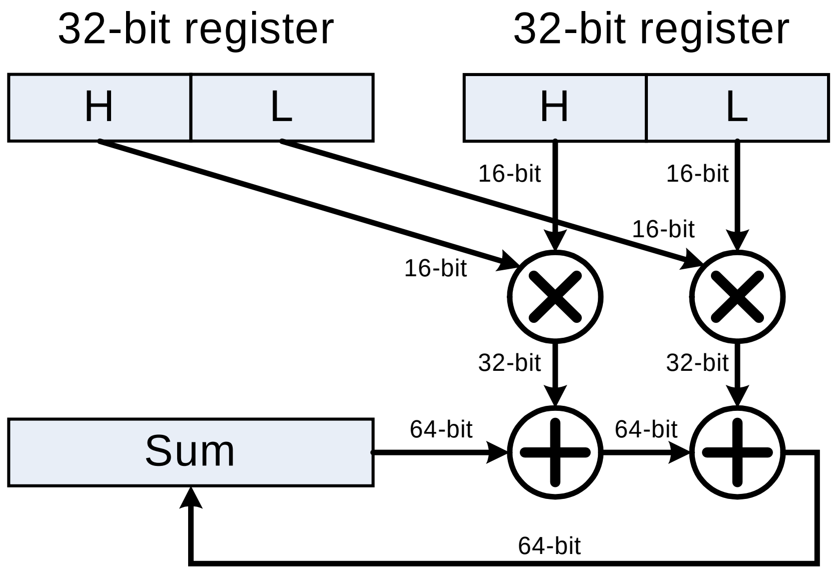
Apply SIMD

- 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

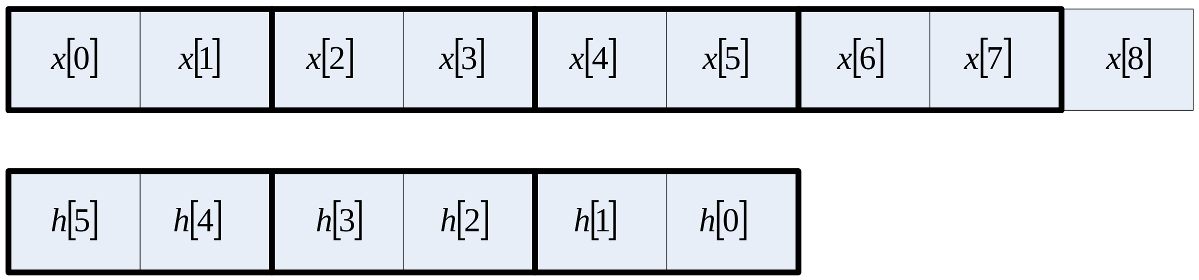
- Access to SIMD is via compiler intrinsics

- Example dual 16-bit MAC
  - SUM=__SMLALD(C, S, SUM)



Data organization with SIMD

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



Inner Loop with 16-bit SIMD

```
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
}                                             // 3 cycles
```

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

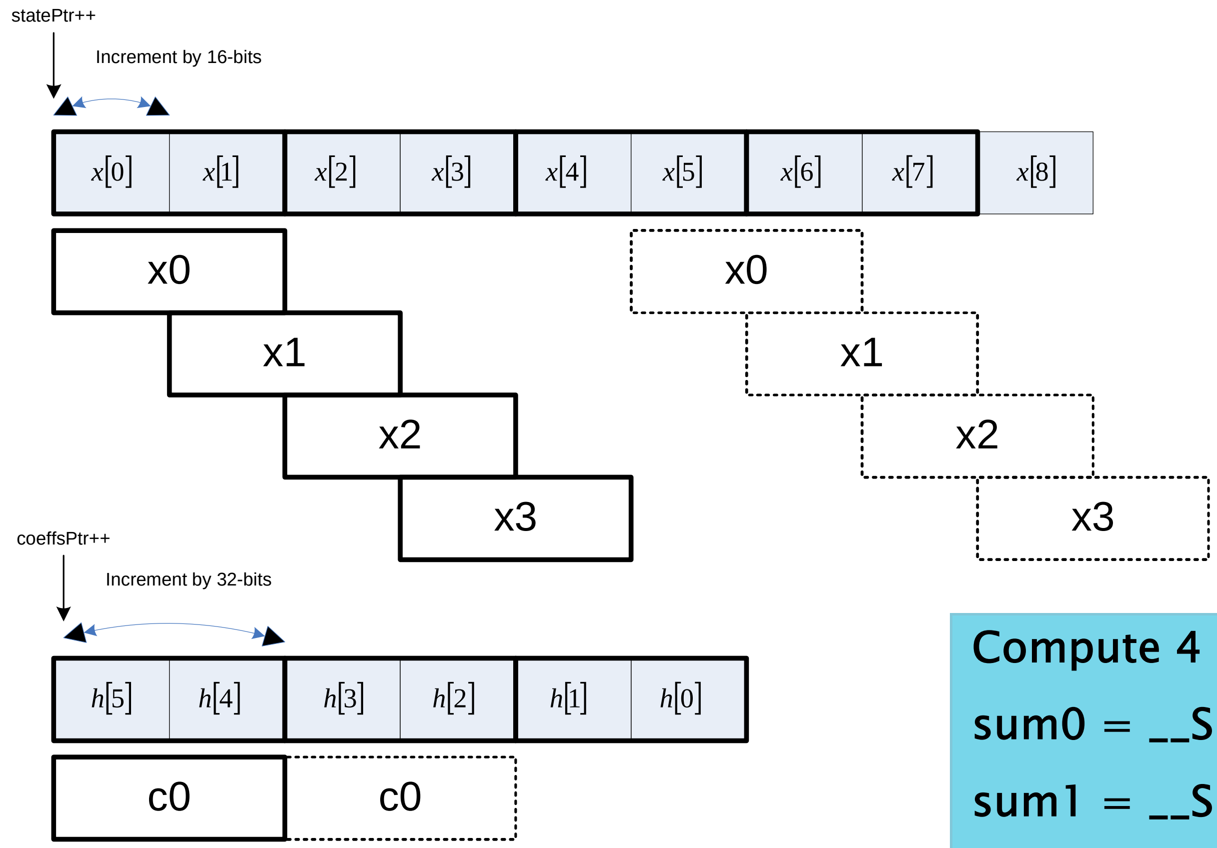
That's much better!
But is there anything more?

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:

```
sum0 = __SMLALD(x0, c0, sum0)
sum1 = __SMLALD(x1, c0, sum1)
sum2 = __SMLALD(x2, c0, sum2)
sum3 = __SMLALD(x3, c0, sum3)
```

Final FIR Code

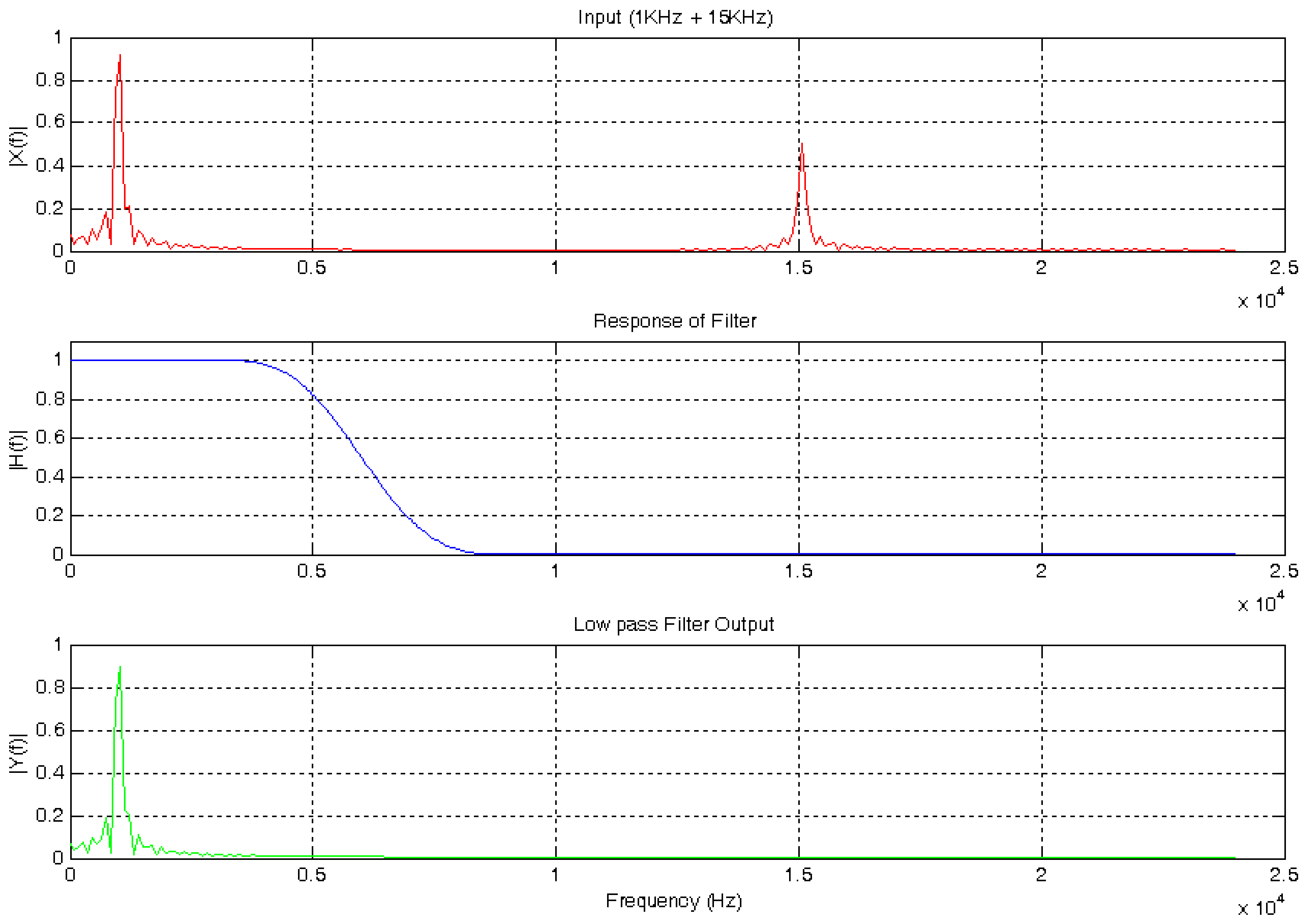
```
sample = blockSize/4;
    do
    {
       sum0 = sum1 = sum2 = sum3 = 0;
       statePtr = stateBasePtr;
       coeffPtr = (q31_t *)(S->coeffs);
       x0 = *(q31_t *)(statePtr++);
       x1 = *(q31_t *)(statePtr++);

       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);
```

Uses loop unrolling, SIMD intrinsics, caching of states and coefficients, and work around circular addressing by using a large state buffer.

Inner loop is 26 cycles for a total of 16, 16-bit MACs.

Only 1.625 cycles per filter tap!

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

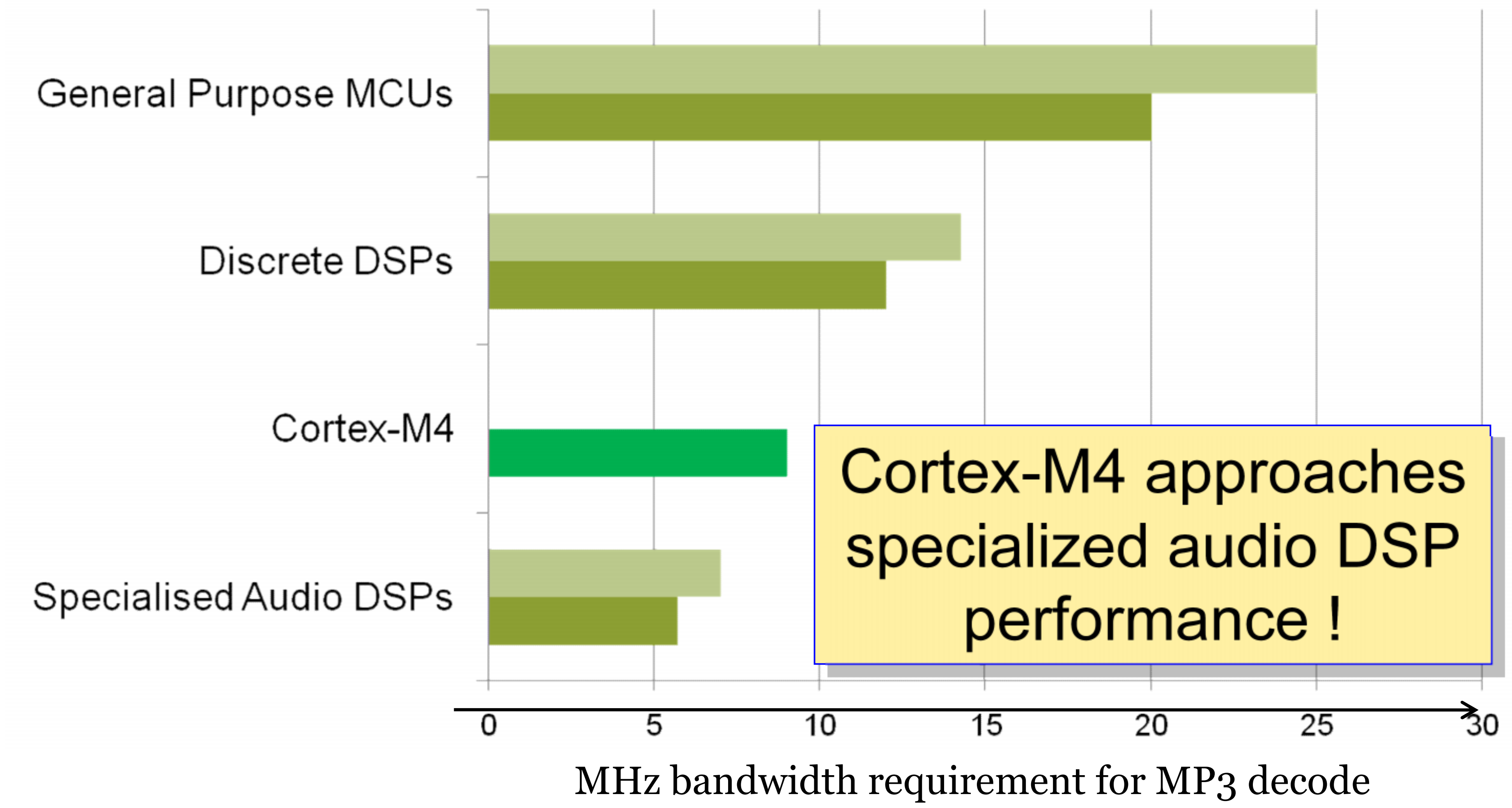
- Basic Cortex-M4 C code quite reasonable performance for simple algorithms
- Through simple optimizations, you can get to high performance on the Cortex-M4
- You DO NOT have to write Cortex-M4 assembly, all optimizations can be done completely in C

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

<u>Variants for functions across q7,q15,q31 and f32 data types</u>

DSP example – MP3 audio playback



Quick Demos