

ECSE 426

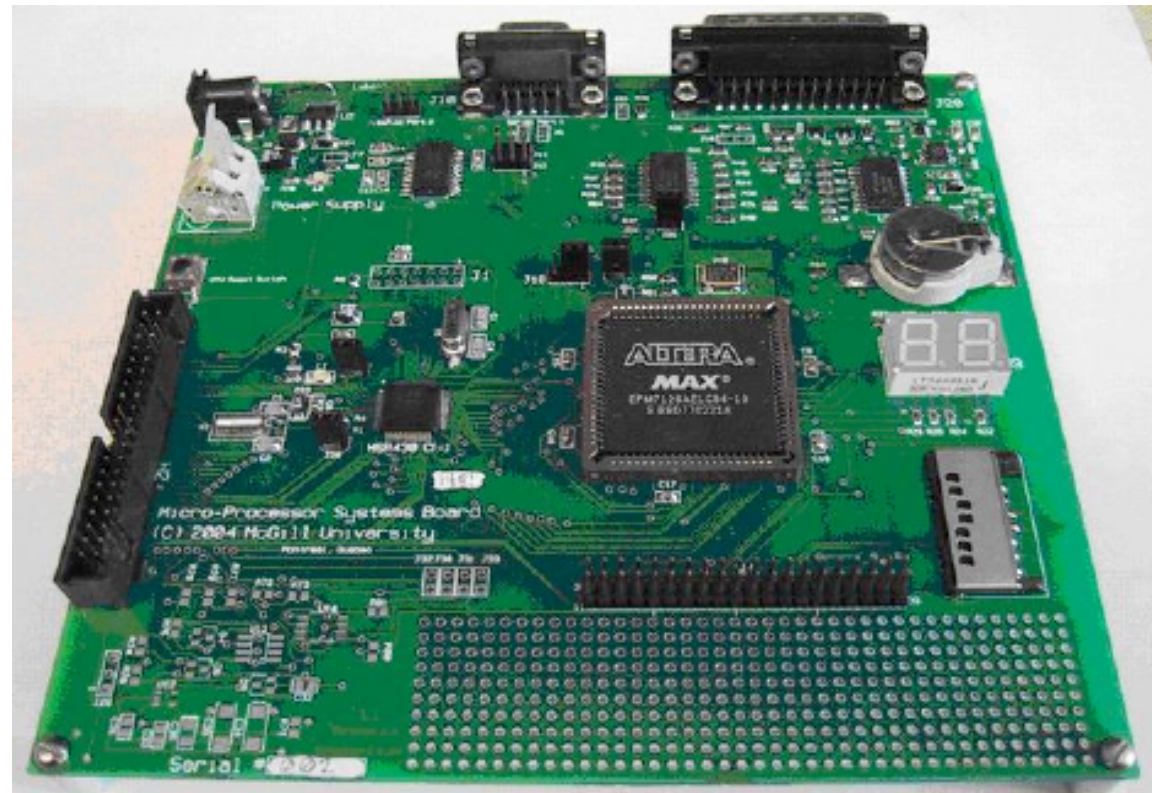
Microrchitecture and SIMD

Zeljko Zilic

zeljko.zilic@mcgill.ca



McGill



Acknowledgments: to STMicroelectronics for material on processors and the board

Outline

- ARM Cortex M4 Microarchitecture
- Important Hardware Blocks
 - Flash ART accelerator
 - SIMD/DSP Unit
- Debug interfaces – SWD and JTAG
- Lab: Part 1 of Lab 1
 - Coordinate with TAs
- Quiz 1 in 7 days (Feb. 8) in the class!

Cortex-M4 Microarch.

○ **ARMv7ME Architecture**

- Thumb-2 Technology
 - Compatible with Cortex-M3
- Single precision FPU
- DSP and SIMD extensions
- Single cycle MAC ($32 \times 32 + 64 \rightarrow 64$)

○ **Microarchitecture**

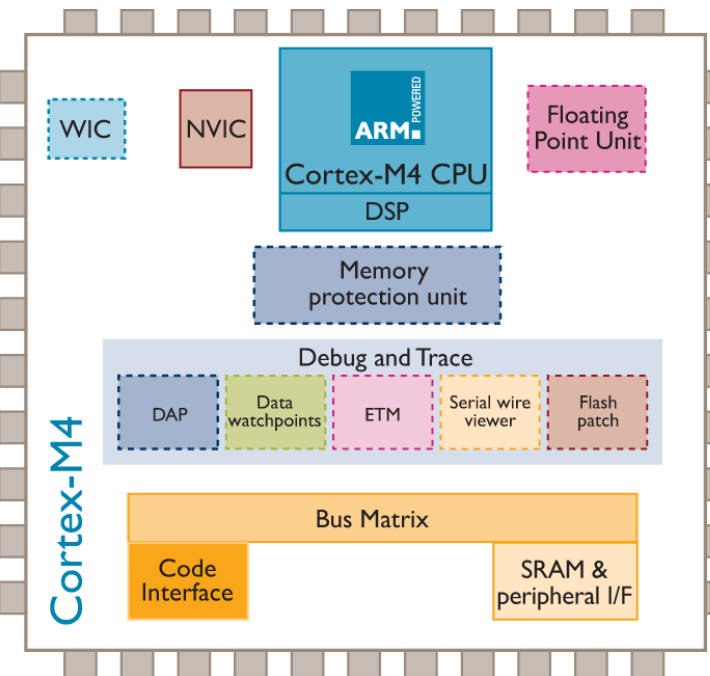
- 3-stage pipeline with branch speculation
- 3x AHB-Lite Bus Interfaces

○ **Configurable for ultra low power**

- Deep Sleep Mode, Wakeup Interrupt Controller
- Power down features for Floating Point Unit

○ **Flexible configurations for wider applicability**

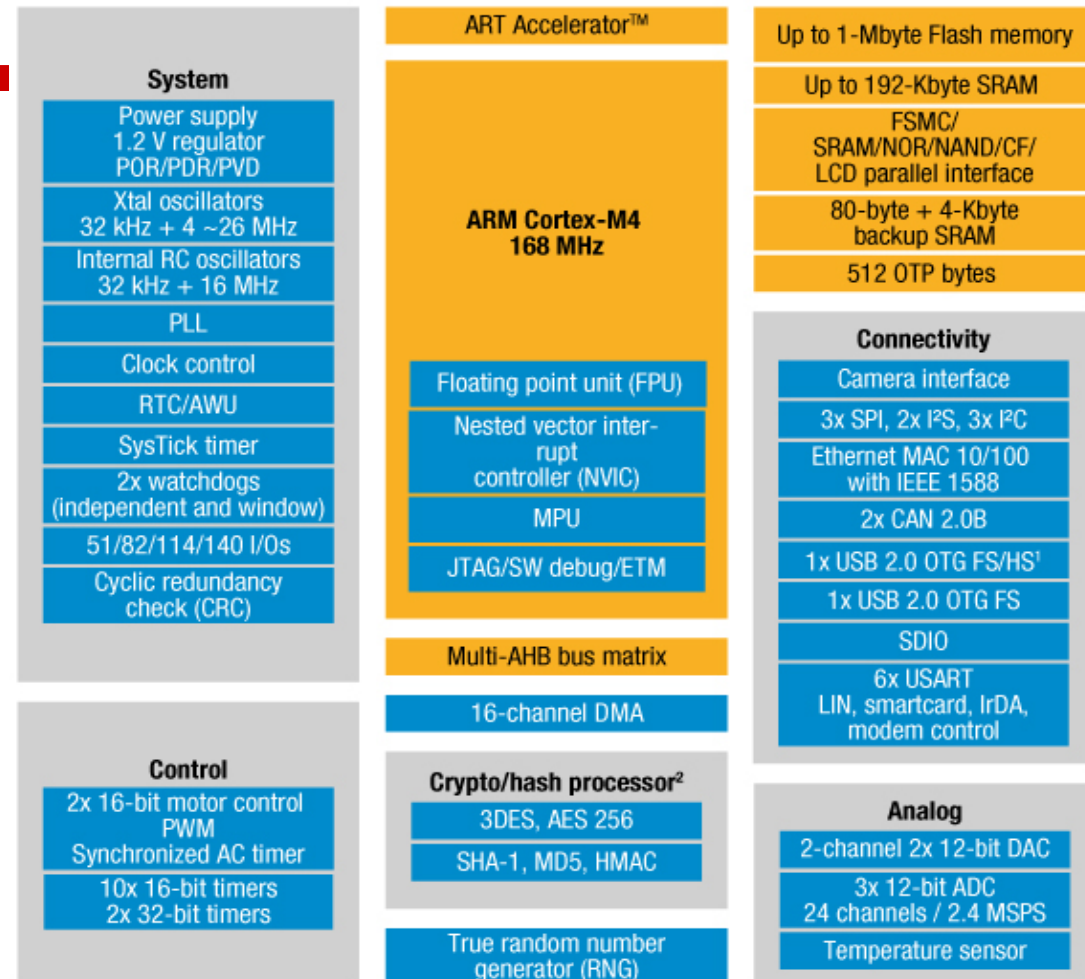
- Configurable Interrupt Controller (1-240 Interrupts and Priorities)
- Debug & Trace
- Optional Memory Protection Unit



STM32 F4 block diagram

Highlights

- 168 MHz Cortex-M4 CPU
 - Floating point unit (FPU)
 - ART Accelerator™
 - Multi-level AHB bus matrix
- 1-Mbyte Flash, 192-Kbyte SRAM
- 1.7 to 3.6 V supply
- Real-Time Clock (RTC): <1 µA typ, sub second accuracy
- 2x full duplex I²S
- 3x 12-bit ADC
0.41 µs/2.4 MSPS
- 168 MHz timers



Notes:

1. HS requires an external PHY connected to the ULPI interface
2. Crypto/hash processor on STM32F417 and STM32F415



McGill

STM32 F4 Family highlights

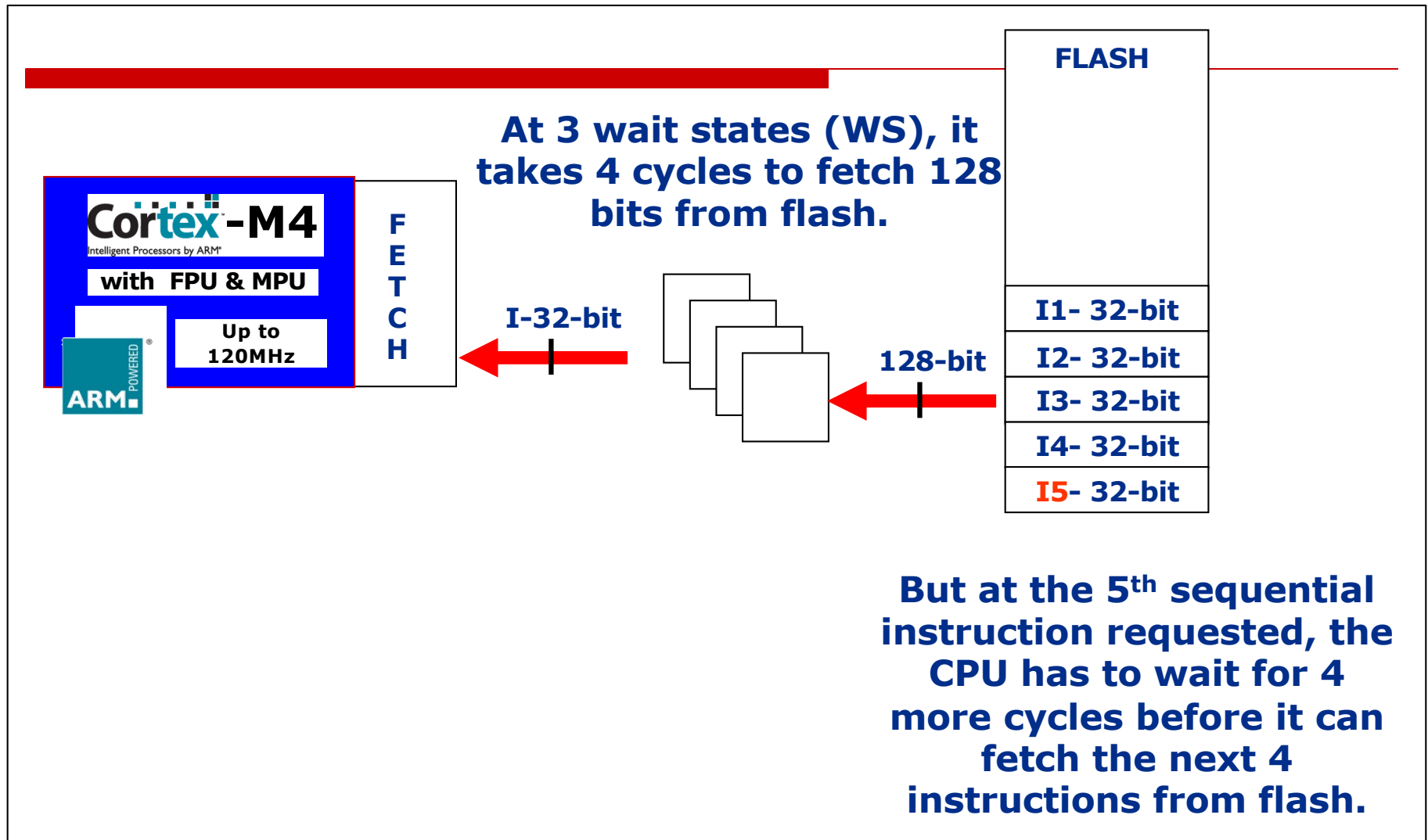
Advanced technology and process from ST:

- Memory accelerator: ART Accelerator™
- Multi AHB Bus Matrix

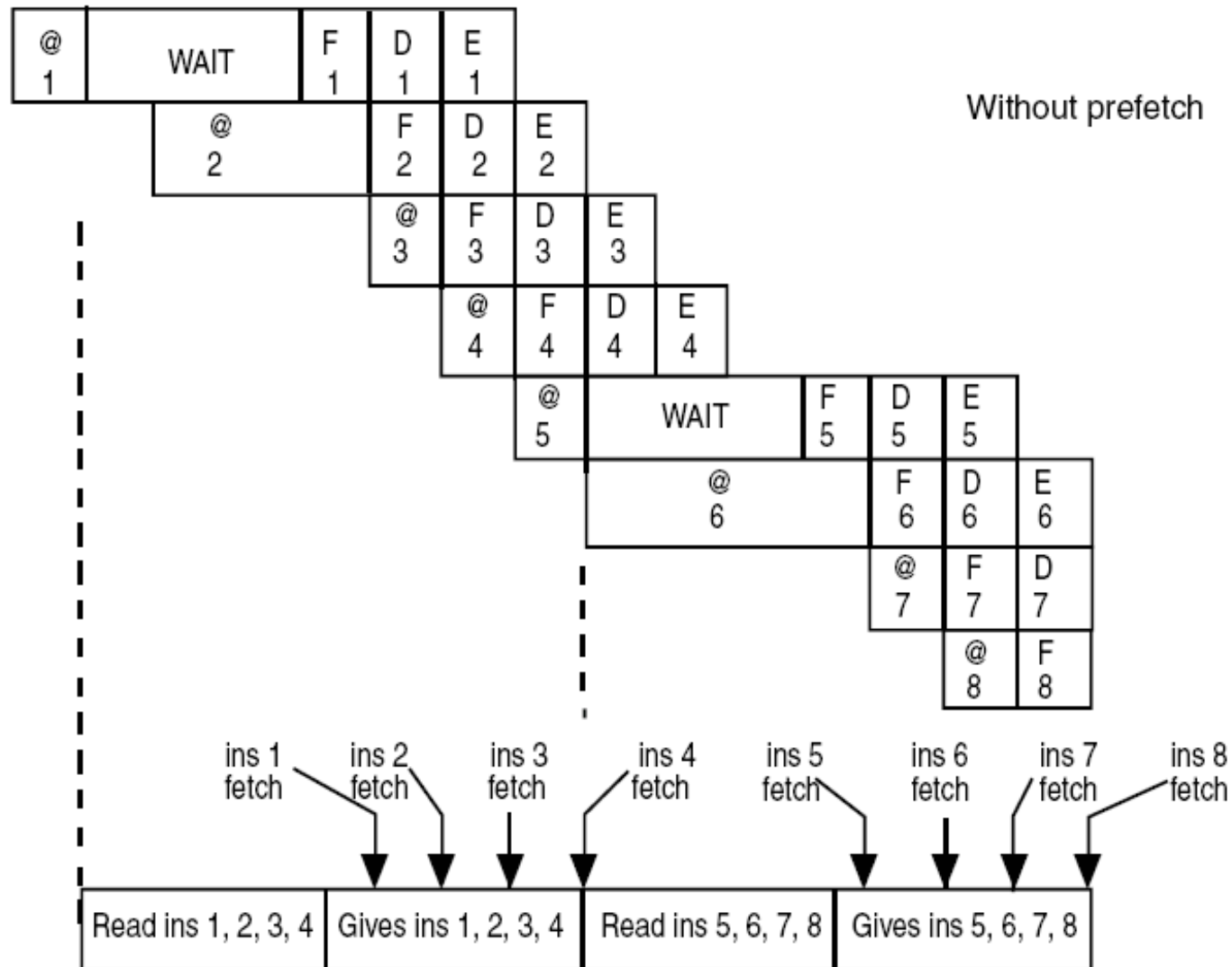
Outstanding performance results:

- 210DMIPS at 168Mhz
- Execution from Flash equivalent to 0-wait state performance up to 168Mhz due to ST ART Accelerator

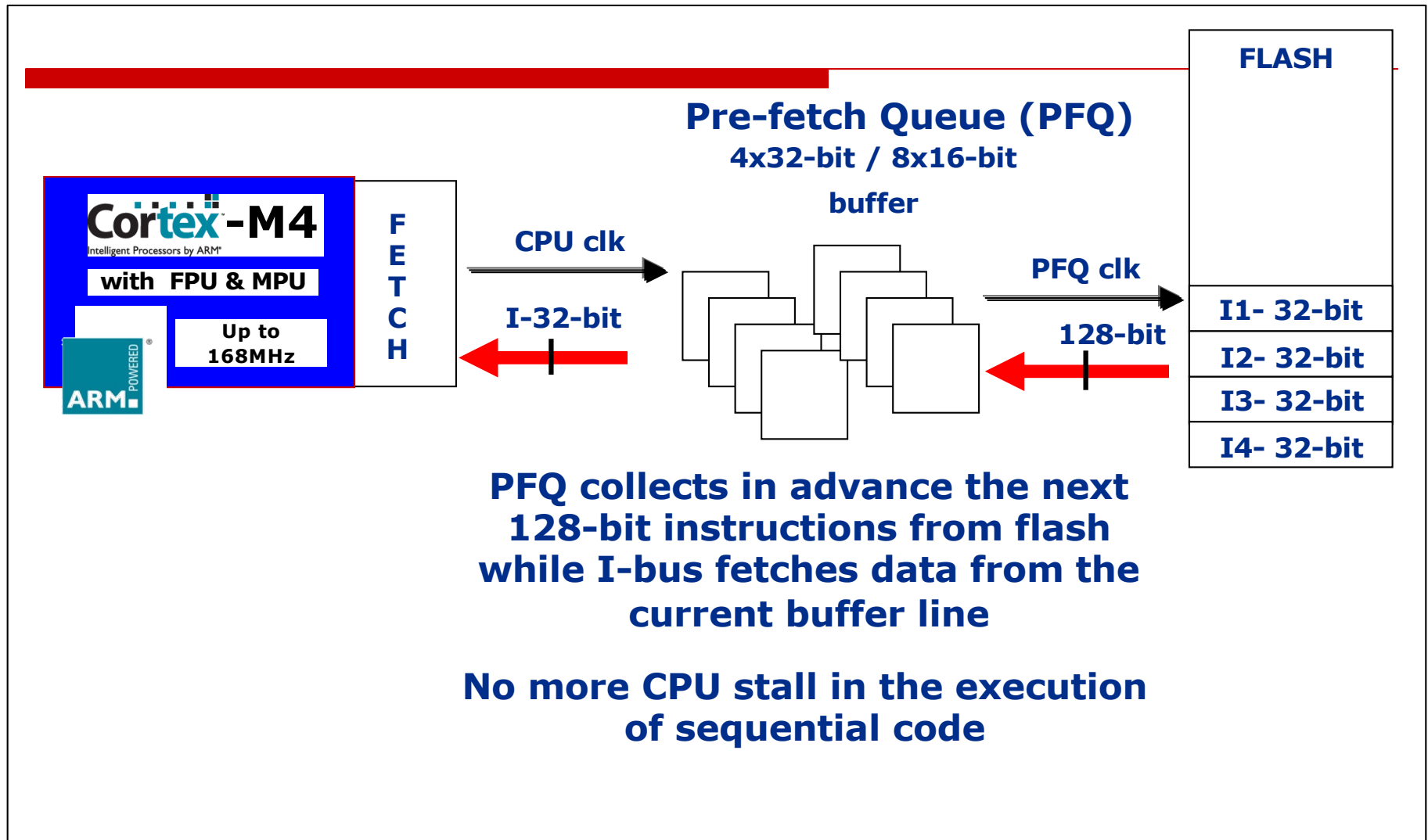
System Architecture – Flash Performance



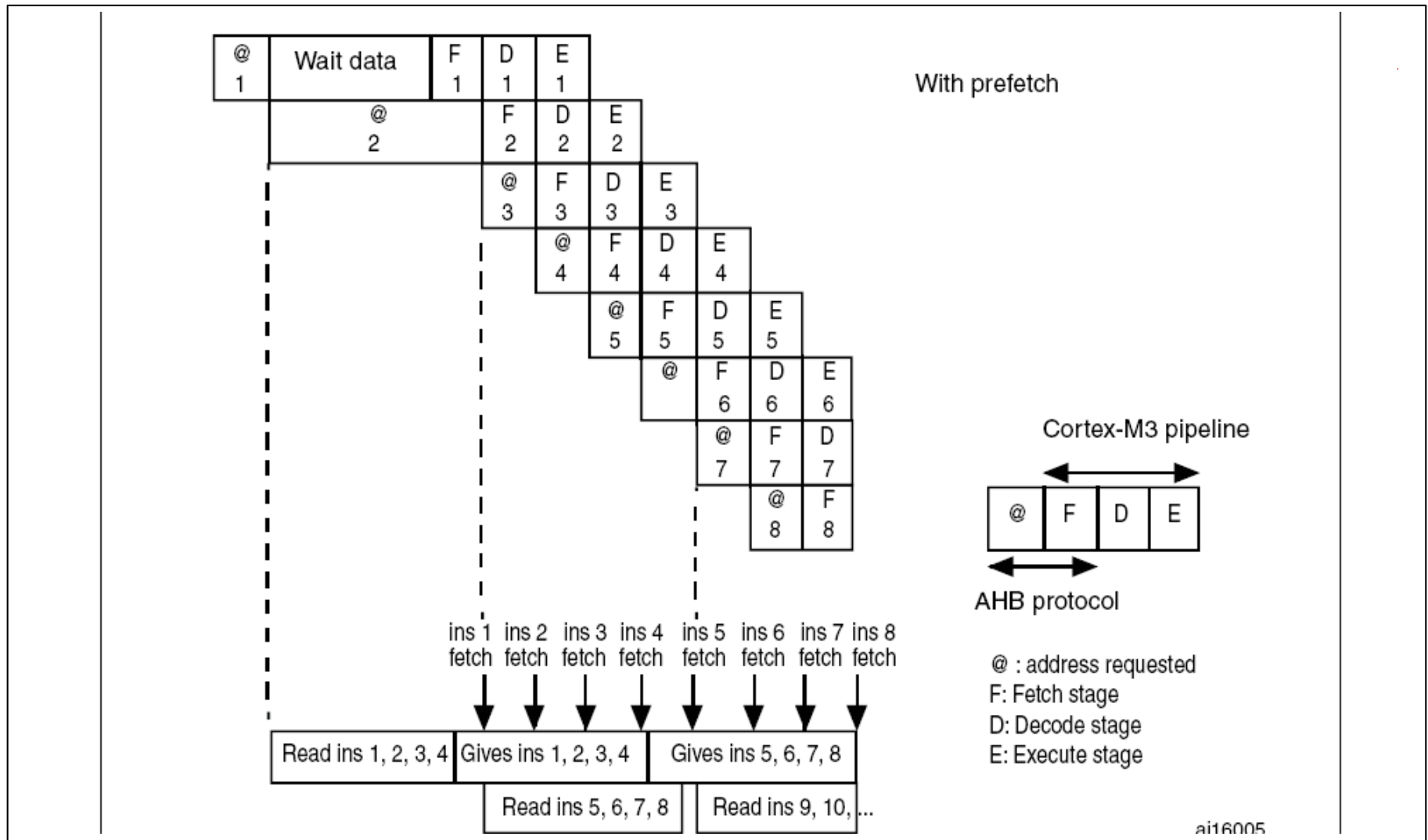
System Architecture – Flash performance



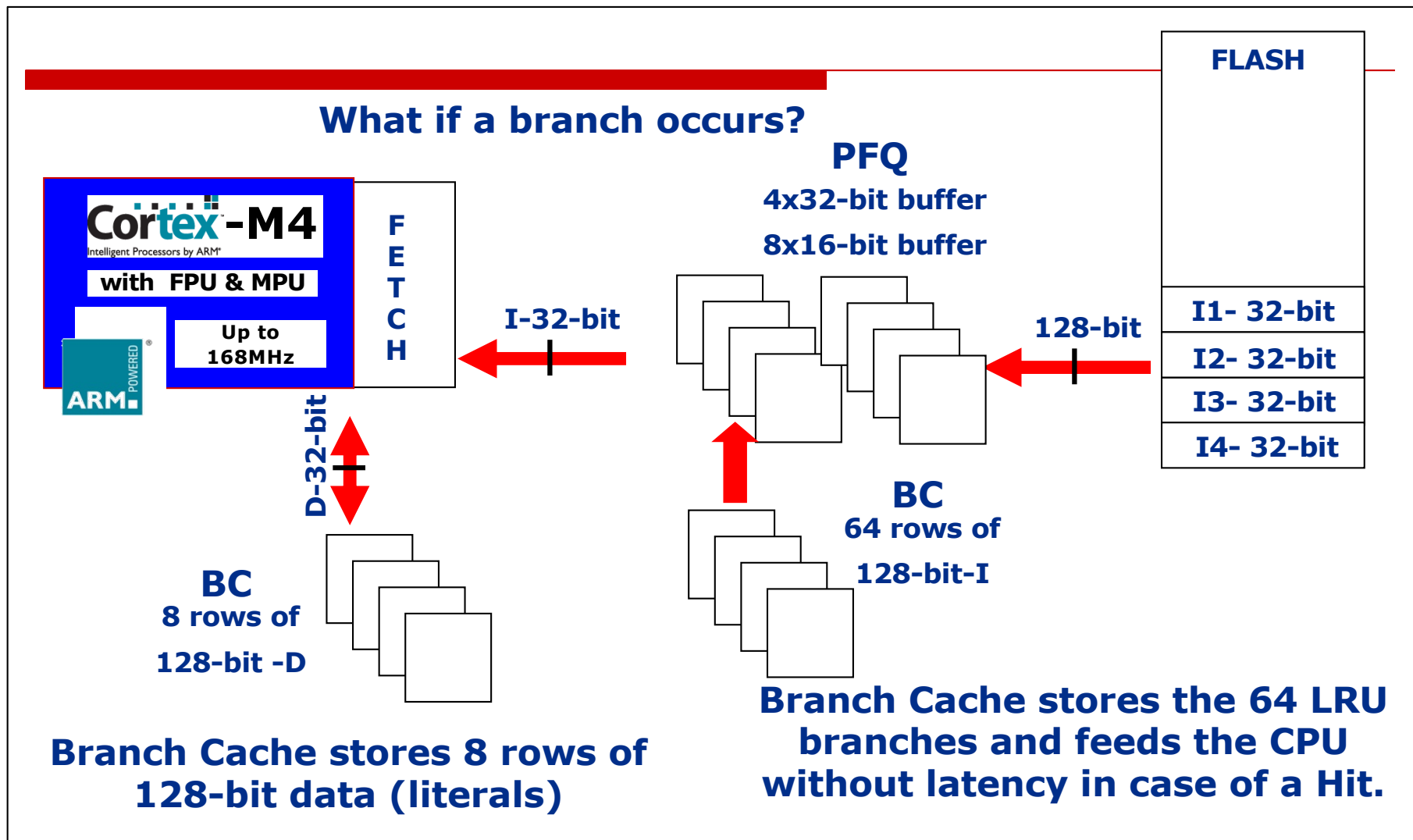
ART Accelerator Role - 1/2



System Architecture – Flash performance



ART Accelerator Role - 2/2



Single-cycle Multiply-accumulate (MAC) Unit

- The multiplier unit allows any MUL or MAC instructions to be executed in a single cycle
 - Signed/Unsigned Multiply
 - Signed/Unsigned Multiply-Accumulate
 - Signed/Unsigned Multiply-Accumulate Long (64-bit)
- Benefits : Speed improvement vs. Cortex-M3
 - 4x for 16-bit MAC (dual 16-bit MAC)
 - 2x for 32-bit MAC
 - up to 7x for 64-bit MAC

Cortex-M4 Single-cycle MAC

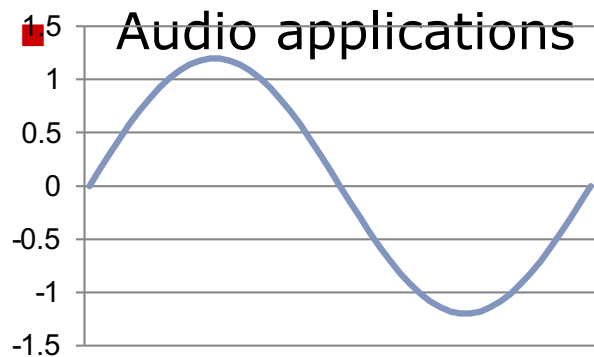
OPERATION	INSTRUCTIONS	CM3	CM4
$16 \times 16 = 32$	SMULBB, SMULBT, SMULTB, SMULTT	n/a	1
$16 \times 16 + 32 = 32$	SMLABB, SMLABT, SMLATB, SMLATT	n/a	1
$16 \times 16 + 64 = 64$	SMLALBB, SMLALBT, SMLALTB, SMLALTT	n/a	1
$16 \times 32 = 32$	SMULWB, SMULWT	n/a	1
$(16 \times 32) + 32 = 32$	SMLAWB, SMLAWT	n/a	1
$(16 \times 16) \pm (16 \times 16) = 32$	SMUAD, SMUADX, SMUSD, SMUSDX	n/a	1
$(16 \times 16) \pm (16 \times 16) + 32 = 32$	SMLAD, SMLADX, SMLSD, SMLSDX	n/a	1
$(16 \times 16) \pm (16 \times 16) + 64 = 64$	SMLALD, SMLALDX, SMLS LD, SMLS LD X	n/a	1
$32 \times 32 = 32$	MUL	1	1
$32 \pm (32 \times 32) = 32$	MLA, MLS	2	1
$32 \times 32 = 64$	SMULL, UMULL	5-7	1
$(32 \times 32) + 64 = 64$	SMLAL, UMLAL	5-7	1
$(32 \times 32) + 32 + 32 = 64$	UMAAL	n/a	1
$32 \pm (32 \times 32) = 32$ (upper)	SMMLA, SMMLAR, SMMLS, SMMLSR	n/a	1
$(32 \times 32) = 32$ (upper)	SMMUL, SMMULR	n/a	1

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

Saturated arithmetic

- Intrinsically prevents overflow of variable by clipping to min/max boundaries and remove CPU burden due to software range checks

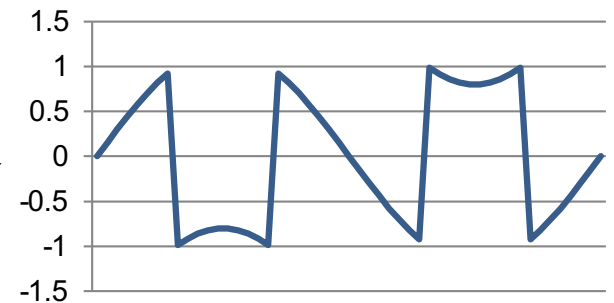
- Benefits



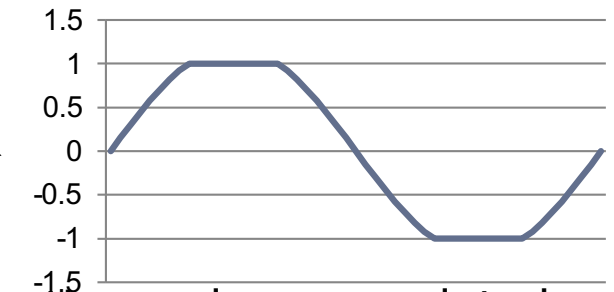
■ Control applications

- The PID controllers' integral term is continuously accumulated over time. The saturation automatically limits its value and saves several CPU cycles per regulators

Without
saturation



With
saturation



McGill

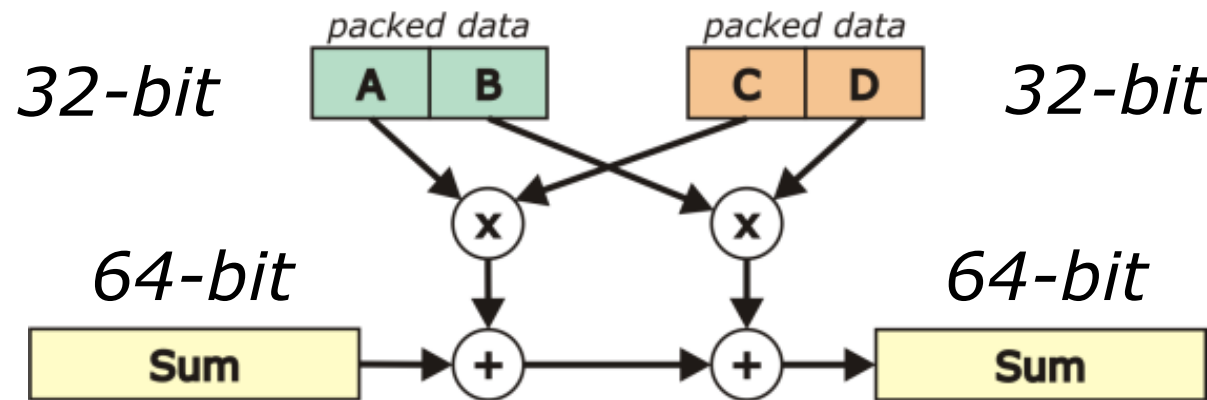
Single-cycle SIMD instructions

- Stands for Single Instruction Multiple Data
- Allows to do simultaneously several operations with 8-bit or 16-bit data format
 - Ex: dual 16-bit MAC ($\text{Result} = 16 \times 16 + 16 \times 16 + 32$)
 - Ex: Quad 8-bit SUB / ADD
- Benefits
 - Parallelizes operations (2x to 4x speed gain)
 - Minimizes the number of Load/Store instruction for exchanges between memory and register file (2 or 4 data transferred at once), if 32-bit is not necessary
 - Maximizes register file use (1 register holds 2 or 4 values)

SIMD operation example

- SIMD extensions perform multiple operations in one cycle

$$\text{Sum} = \text{Sum} + (A \times C) + (B \times D)$$



- SIMD techniques operate with packed data

Cortex-M4 DSP Instructions

CLASS	INSTRUCTION	Cycle counts	
		Cortex-M3	Cortex-M4
Arithmetic	ALU operation (not PC)	1	1
	ALU operation to PC	3	3
	CLZ	1	1
	QADD, QDADD, QSUB, QDSUB	n/a	1
	QADD8, QADD16, QSUB8, QSUB16	n/a	1
	QDADD, QDSUB	n/a	1
	QASX, QSAX, SASX, SSAX	n/a	1
	SHASX, SHSAX, UHASX, UHSAX	n/a	1
	SADD8, SADD16, SSUB8, SSUB16	n/a	1
	SHADD8, SHADD16, SHSUB8, SHSUB16	n/a	1
	UQADD8, UQADD16, UQSUB8, UQSUB16	n/a	1
	UHADD8, UHADD16, UHSUB8, UHSUB16	n/a	1
	UADD8, UADD16, USUB8, USUB16	n/a	1
	UQASX, UQSAX, USAX, UASX	n/a	1
	UXTAB, UXTAB16, UXTAH	n/a	1
	USAD8, USADA8	n/a	1
Multiplication	MUL, MLA	1 - 2	1
	MULS, MLAS	1 - 2	1
	SMULL, UMULL, SMLAL, UMLAL	5 - 7	1
	SMULBB, SMULBT, SMULTB, SMULTT	n/a	1
	SMLABB, SMLBT, SMLATB, SMLATT	n/a	1
	SMULWB, SMULWT, SMLAWB, SMLAWT	n/a	1
	SMLALBB, SMLALBT, SMLALTB, SMLALTT	n/a	1
	SMLAD, SMLADX, SMLALD, SMLALDX	n/a	1
	SMLSD, SMLSDX	n/a	1
	SMLSLD, SMLSXD	n/a	1
	SMMLA, SMMLAR, SMMLS, SMMLSR	n/a	1
	SMMUL, SMMULR	n/a	1
	SMUAD, SMUADX, SMUSD, SMUSDX	n/a	1
	UMAAL	n/a	1
Division	SDIV, UDIV	2 - 12	2 - 12

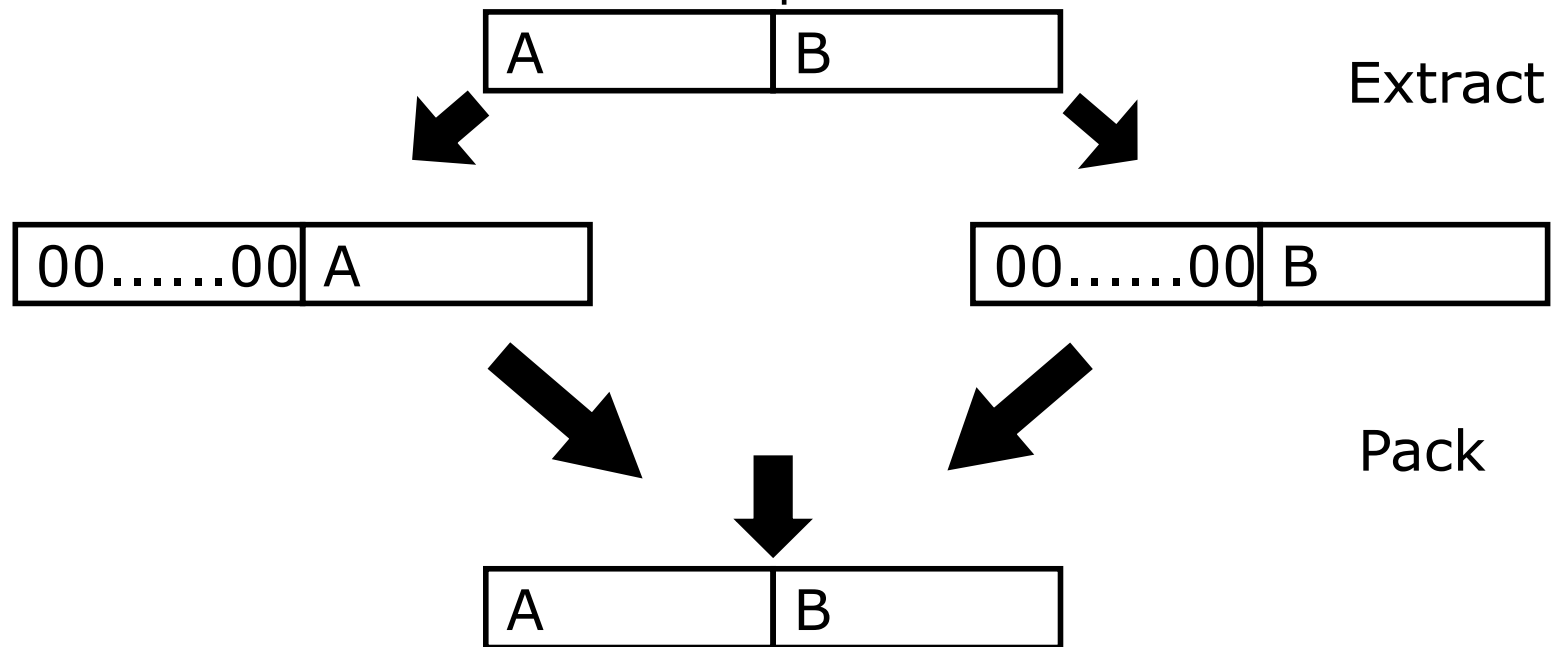
**Single
cycle
MAC**

Non-DSP Instructions

CLASS	INSTRUCTION	Cycle counts	
		Cortex-M3	Cortex-M4
Load/Store	Load single byte to R0-R14	1 - 3	1 - 3
	Load single halfword to R0-R14	1 - 3	1 - 3
	Load single word to R0-R14	1 - 3	1 - 3
	Load to PC	5	5
	Load double-word	3	3
	Store single word	1 - 2	1 - 2
	Store double word	3	3
	Load-multiple registers (not PC)	N+1	N+1
	Load-multiple registers plus PC	N+5	N+5
	Store-multiple registers	N+1	N+1
	Load/store exclusive	2	2
	SWP	n/a	n/a
Branch	B, BL, BX, BLX	2 - 3	2 - 3
	CBZ, CBNZ	3	3
	TBB, TBH	5	5
	IT	0 - 1	0 - 1
Special	MRS	1	1
	MSR	1	1
	CPS	1	1
Manipulation	BFI, BFC	1	1
	RBIT, REV, REV16, REVSH	1	1
	SBFX, UBFX	1	1
	UXTH, UXTB, SXTB, SXTB	1	1
	SSAT, USAT	1	1
	SEL	n/a	1
	SXTAB, SXTAB16, SXTAH	n/a	1
	UXTB16, SXTB16	n/a	1
	SSAT16, USAT16	n/a	1
	PKHTB, PKHBT	n/a	1

Packed data types

- Several instructions operate on “packed” data types
 - Byte or halfword quantities packed into words
 - Allows more efficient access to packed structure types
 - SIMD instructions can act on packed data
 - Instructions to extract and pack data

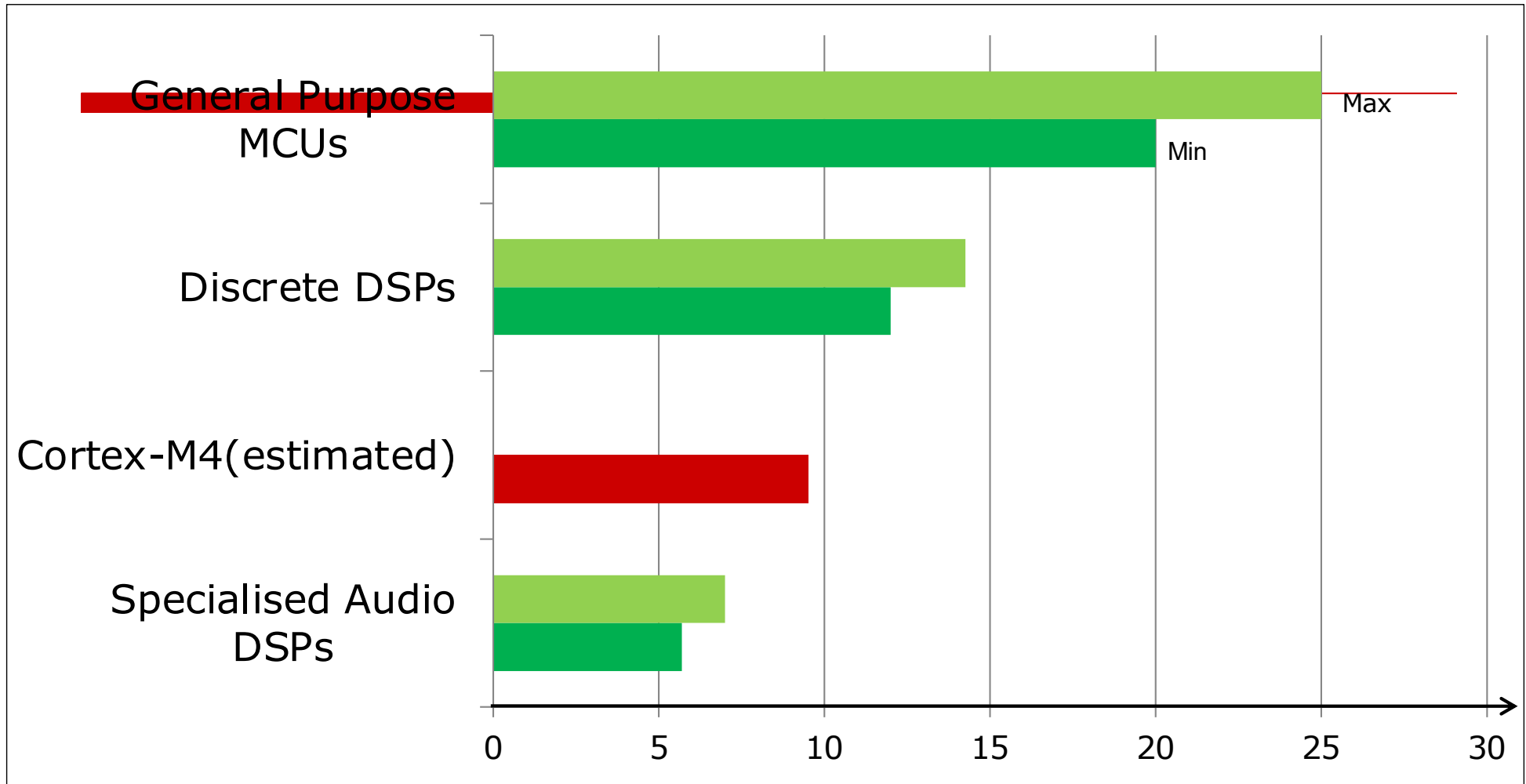


DSP performances for control application

- Example based on a complex formula used for sensorless motor drive
- Gain comes for load operations and SIMD instructions
- Total gain on this part is 25 to 35%

Cortex M3 (28-38 c.)	Cortex M4 (18-28 c.)
LDRSH R12,[R4, #+12]	LDR R10,[R4, #+12]
LDRSH R0,[SP, #+20]	(1 single 32-bit load replacing two 16-bit load with sign extension. Gain: 2 cycles)
SXTH LR,R8	
MUL R8,LR,R0	
LDR R1,[R4, #+44]	
SDIV R0,R1,R7	
LDRSH R2,[R4, #+24]	
LDRSH R3,[R4, #+26]	LDR R2,[R4, #+22]
LDRSH R10,[R4, #+22]	(1 single 32-bit load replacing to 16-bit with sign extension. Gain: 2 cycles)
SXTH R6,R6	
MLS R5,R6,R10,R5	
MLA R5,R9,R12,R5	SMLSD R5, R10, R6, R5 (1 SIMD instruction replacing two multiply-accumulate. Gain: 3 cycles)
ASR R6,R8,#+15	
MLA R5,R6,R3,R5	
SXTH R0,R0	
MLS R5,R0,R2,R5	SMLSD R5, R0, R2 (1 SIMD instruction replacing two multiply-accumulate. Gain: 3 cycles)
STR R5,[SP, #+12]	

DSP application example: MP3 audio playback



MHz required for MP3 decode (smaller is better !)

Main DSP operations

- Finite impulse response (FIR) filters
 - Data communications
 - Echo cancellation (adaptive versions)
 - Smoothing data
- Infinite impulse response (IIR) filters
 - Audio equalization
 - Motor control
- Fast Fourier transforms (FFT)
 - Audio compression
 - Spread spectrum communication
 - Noise removal

Assembly or C ?

- Assembly ?
 - Pros
 - Can result in highest performance
 - Cons
 - Difficult learning curve, longer development cycles
 - Code reuse difficult – not portable
- C ?
 - Pros
 - Easy to write and maintain code, faster development cycles
 - Code reuse possible, using third party software is easier
 - Cons
 - Highest performance might not be possible
 - Get to know your compiler !

DSP Processing Details

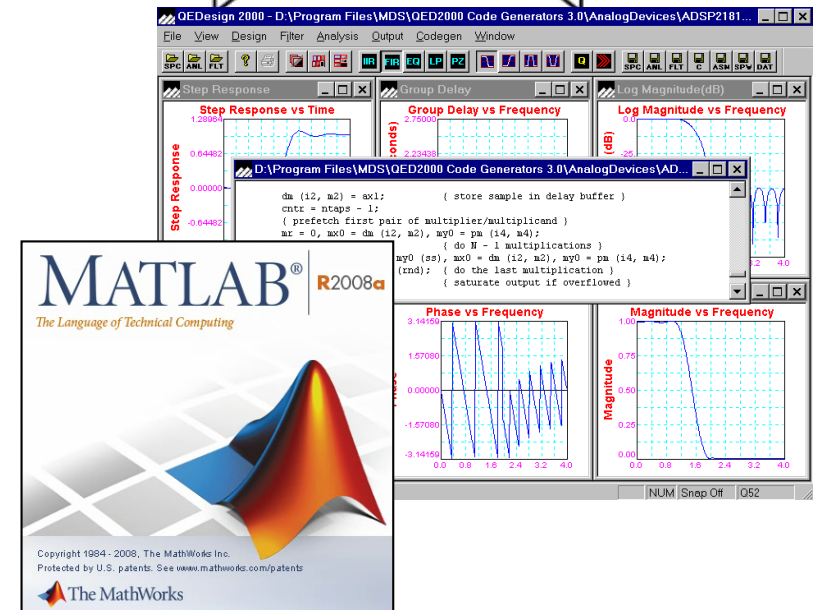
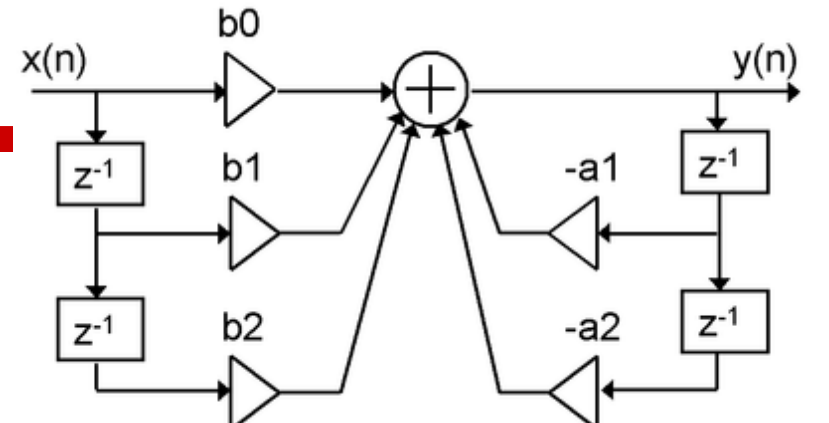
- FIR Filter
$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]$$
- IIR or recursive filter
$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + a_1y[n-1] + a_2y[n-2]$$
- FFT Butterfly (radix-2)
$$Y[k_1] = X[k_1] + X[k_2]$$
$$Y[k_2] = (X[k_1] - X[k_2])e^{-j\omega}$$

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

Computing Coefficients

- Variables in a DSP algorithm can be classified as “coefficients” or “state”
 - Coefficients – parameters that determine the response of the filter (e.g., lowpass, highpass, bandpass, etc.)
 - State – intermediate variables that update based on the input signal
- Coefficients may be computed in a number of different ways
 - Simple design equations running on the MCU
 - External tools such as MATLAB or QED Filter Design
- This structure is called a Direct Form 1 Biquad. It has 5 coefficients and 4 state variables.

http://www.dsprelated.com/dspbooks/filters/Four_Direct_Forms.html



IIR – Single-cycle MAC Benefit

Cortex-M3 Cortex-M4
cycle count cycle count

<code>xN = *x++;</code>	2	2
<code>yN = xN * b0;</code>	3-7	1
<code>yN += xNm1 * b1;</code>	3-7	1
<code>yN += xNm2 * b2;</code>	3-7	1
<code>yN -= yNm1 * a1;</code>	3-7	1
<code>yN -= yNm2 * a2;</code>	3-7	1
<code>*y++ = yN;</code>	2	2
<code>xNm2 = xNm1;</code>	1	1
<code>xNm1 = xN;</code>	1	1
<code>yNm2 = yNm1;</code>	1	1
<code>yNm1 = yN;</code>	1	1
Decrement loop counter	1	1
Branch	2	2

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

Only looking at the inner loop, making these assumptions

Function operates on a block of samples

Coefficients b_0 , b_1 , b_2 , a_1 , and a_2 are in registers

Previous states, $x[n-1]$, $x[n-2]$, $y[n-1]$, and $y[n-2]$ are in registers

Inner loop on Cortex-M3 takes 27-47 cycles per sample

Inner loop on Cortex-M4 takes 16 cycles per sample

Further optimization strategies

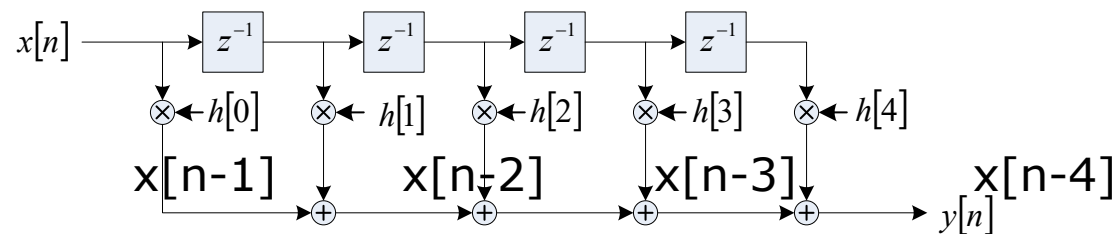
- Circular addressing alternatives
- Loop unrolling
- Caching of intermediate variables
- Extensive use of SIMD and intrinsics

These will be illustrated by looking at a
Finite Impulse Response (FIR) Filter

FIR Filter

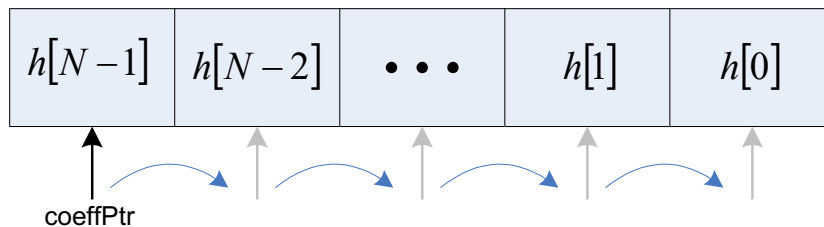
- Occurs frequently in communications, audio, and video applications
- A filter of length N requires
 - N coefficients $h[0], h[1], \dots, h[N-1]$
 - N state variables $x[n], x[n-1], \dots, x[n-(N-1)]$
 - N multiply accumulates
- Classic function in signal processing

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

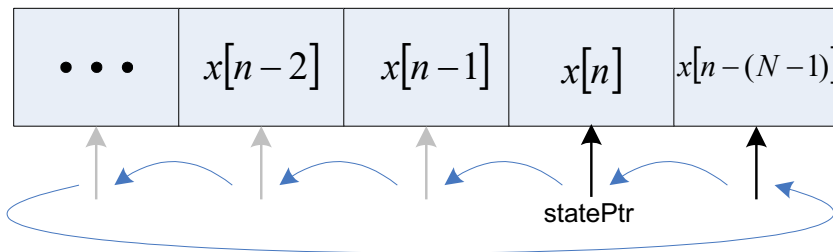


Circular Addressing

- 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;

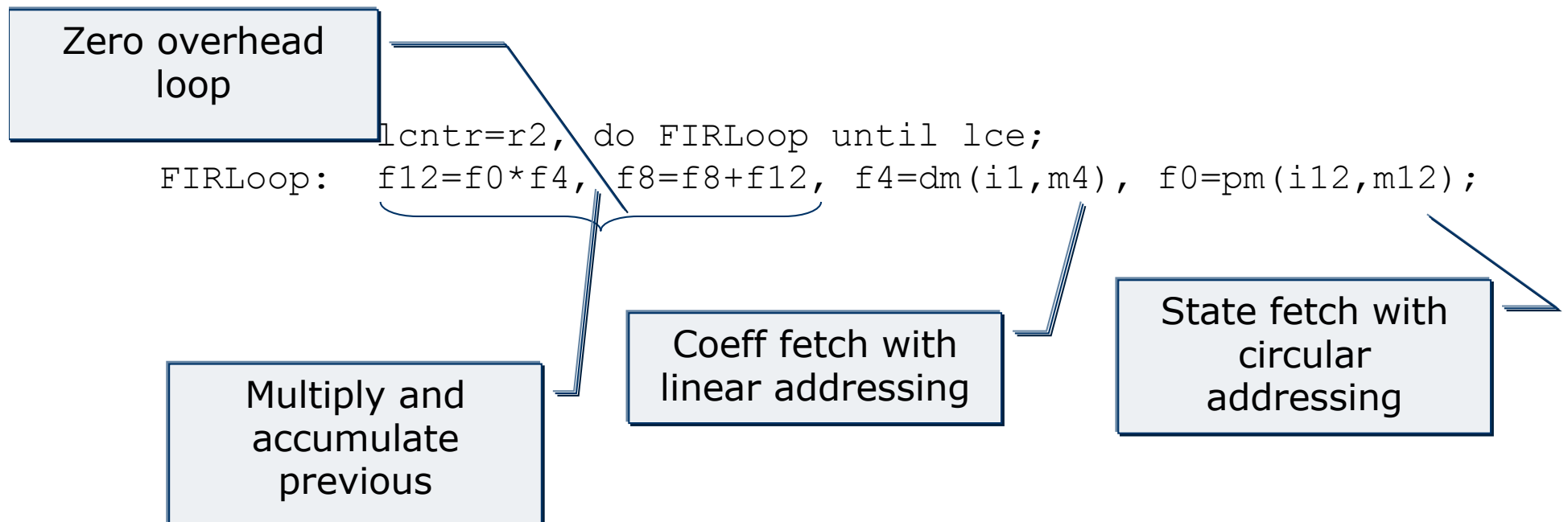
    for(sample=0; sample < blockSize; sample++)
    {
        state[stateIndex++] = in[sample];
        sum=0;
        for(k=0;k<filtLen;k++)
        {
            sum += coeffs[k] * state[stateIndex];
            stateIndex--;
            if (stateIndex < 0)
            {
                stateIndex = filtLen-1;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



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

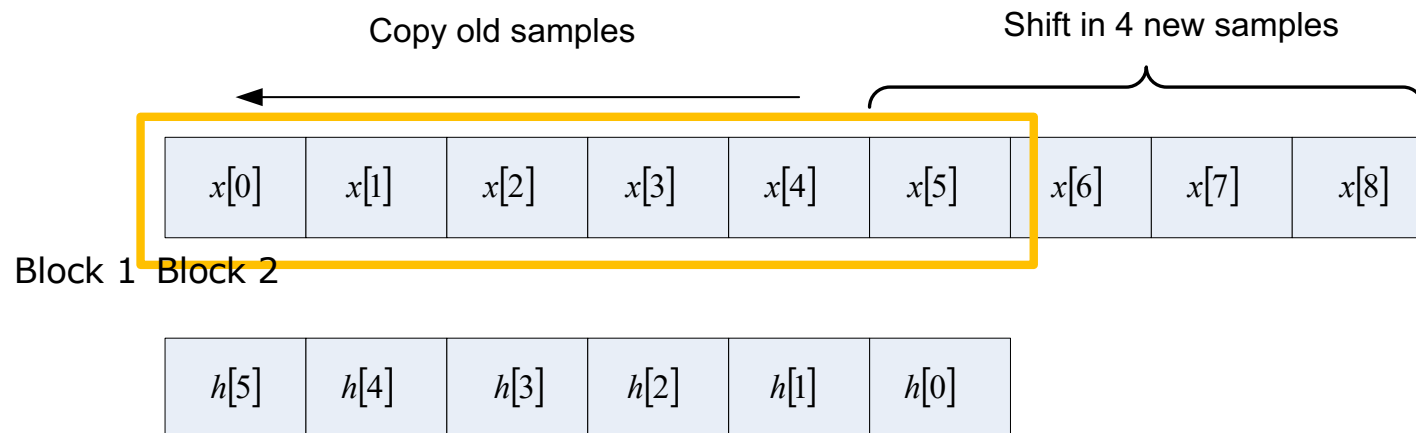
Total	12 cycles

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

How can this be improved on the Cortex-M4 ?

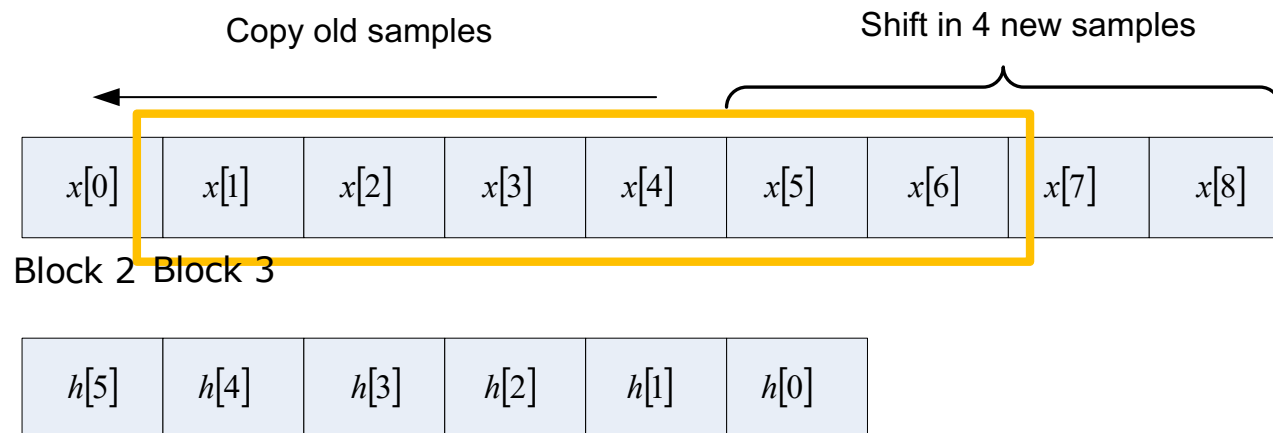
Circular Addressing Alternative

- 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.



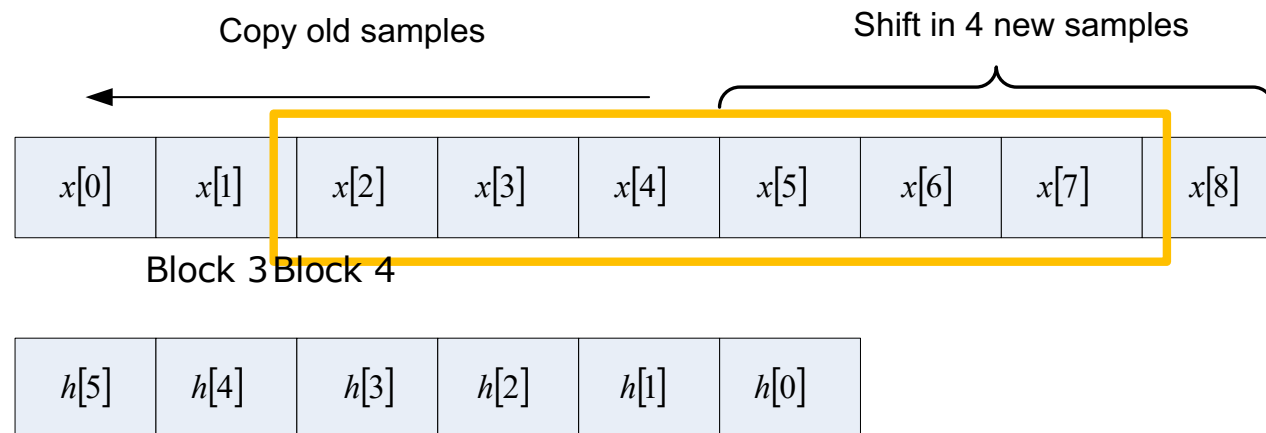
Circular Addressing Alternative

- 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 Addressing 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.



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

-
- An efficient language-independent optimization technique that 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

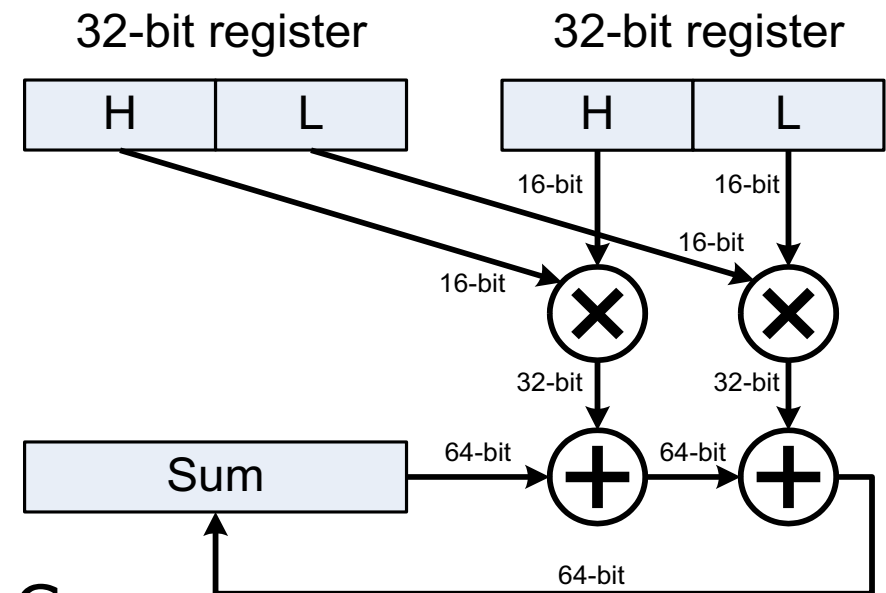
- DSP assembly code = 1 cycle
- Cortex-M4 standard C code takes 12 cycles
- Using circular addressing alternative = 8 cycles
- After loop unrolling < 6 cycles

25% further improvement
But a large gap still exists

Lets try SIMD

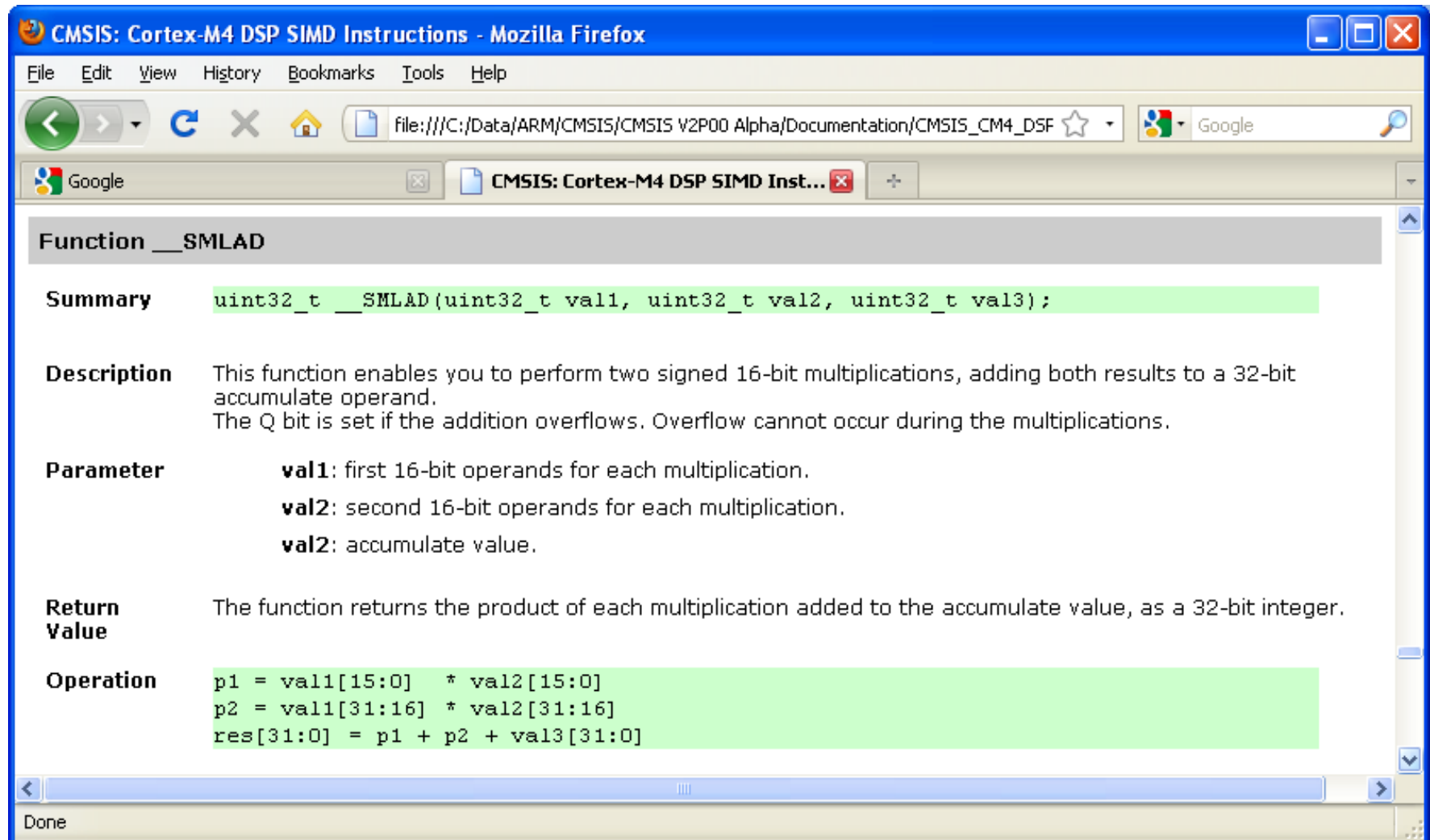
Applying SIMD

- Many image, 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)`



CMSIS Files

41



Function __SMLAD

Summary `uint32_t __SMLAD(uint32_t val1, uint32_t val2, uint32_t val3);`

Description This function enables you to perform two signed 16-bit multiplications, adding both results to a 32-bit accumulate operand. The Q bit is set if the addition overflows. Overflow cannot occur during the multiplications.

Parameter

- val1**: first 16-bit operands for each multiplication.
- val2**: second 16-bit operands for each multiplication.
- val2**: accumulate value.

Return Value The function returns the product of each multiplication added to the accumulate value, as a 32-bit integer.

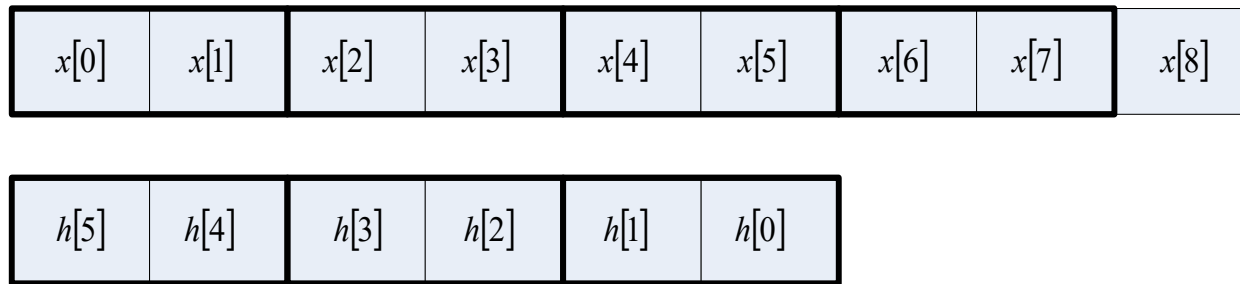
Operation

```
p1 = val1[15:0] * val2[15:0]
p2 = val1[31:16] * val2[31:16]
res[31:0] = p1 + p2 + val3[31:0]
```

Done

Data organization with SIMD

- 16-bit example
- Access two neighboring 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
}
```

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

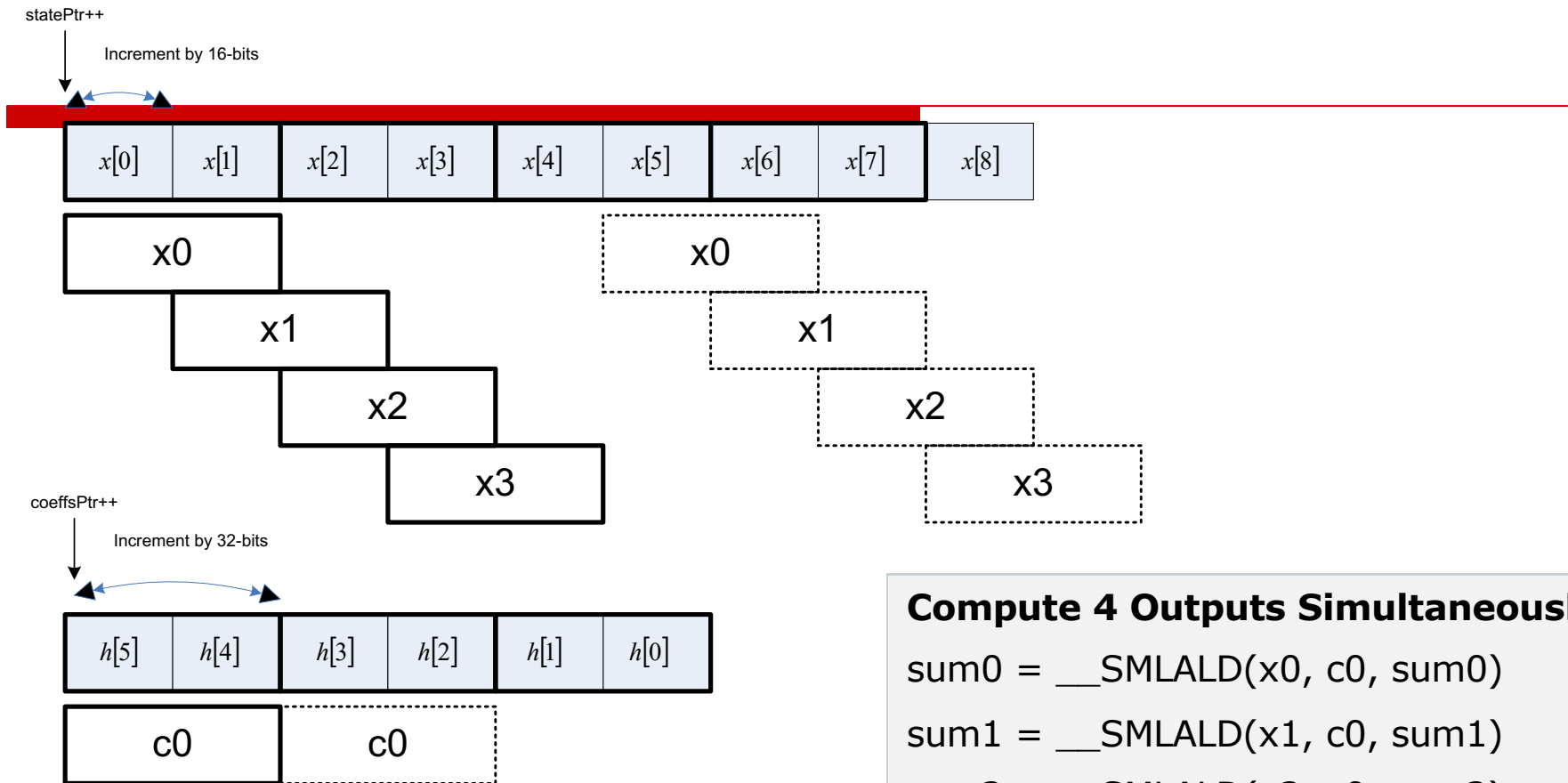
That's much better!
But is there anything more?

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:

$\text{sum0} = \text{__SMLALD}(x0, c0, \text{sum0})$

$\text{sum1} = \text{__SMLALD}(x1, c0, \text{sum1})$

$\text{sum2} = \text{__SMLALD}(x2, c0, \text{sum2})$

$\text{sum3} = \text{__SMLALD}(x3, c0, \text{sum3})$



McGill

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!



McGill

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