STM32F3 Technical Training

For reference only

Refer to the latest documents for details

life.augmented

ARM Cortex M4 in few words

life.augmented

ARM

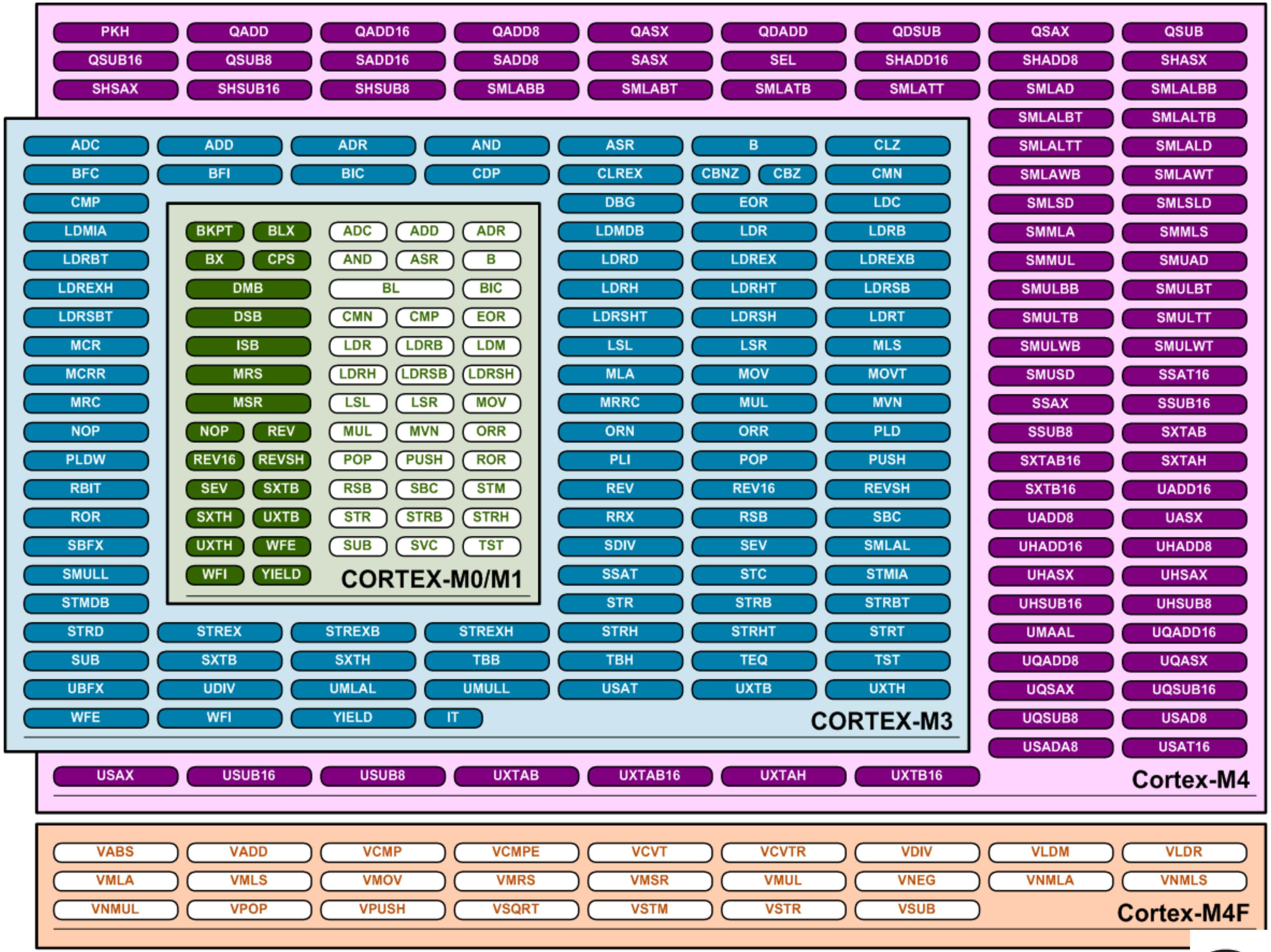
Cortex-M processors

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

Cortex-M0	Cortex-M3	Cortex-M4
"8/16-bit" applications	"16/32-bit" applications	"32-bit/DSC" applications

Binary and tool compatible

Cortex-M processors binary compatible



PKH QADD QADD16 QADD8 QASX QDADD QDSUB QSAX QSUB
QSUB16 QSUB8 SADD16 SADD8 SASX SEL SHADD16 SHADD8 SHASX
SHSAX SHSUB16 SHSUB8 SMLABB SMLABT SMLATB SMLATT SMLAD SMLALBB
SMLALBT SMLALTB
SMLALTT SMLALD
SMLAWB SMLAWT
SMLSD SMLSLD
SMMLA SMMLS
SMMUL SMUAD
SMULBB SMULBT
SMULTB SMULTT
SMULWB SMULWT
SMUSD SSAT16
SSAX SSUB16
SSUB8 SXTAB
SXTAB16 SXTAH
SXTB16 UADD16
UADD8 UASX
UHADD16 UHADD8
UHASX UHSAX
UHSUB16 UHSUB8
UMAAL UQADD16
UQADD8 UQASX
UQSAX UQSUB16
UQSUB8 USAD8
USADA8 USAT16
USAX USUB16 USUB8 UXTAB UXTAB16 UXTAH UXTB16

Cortex-M4

ADC ADD ADR AND ASR B CLZ
BFC BFI BIC CDP CLREX CBNZ CBZ CMN
CMP DBG EOR LDC
LDMIA LDMDB LDR LDRB
LDRBT LDRD LDREX LDREXB
LDREXH LDRH LDRHT LDRSB
LDRSBT LDRSHT LDRSH LDRT
MCR LSL LSR MLS
MCRR MLA MOV MOVT
MRC MRRC MUL MVN
NOP ORN ORR PLD
PLDW PLI POP PUSH
RBIT REV REV16 REVSH
ROR RRX RSB SBC
SBFX SDIV SEV SMLAL
SMULL SSAT STC STMIA
STMDB STR STRB STRBT
STRD STREX STREXB STREXH STRH STRHT STRT
SUB SXTB SXTH TBB TBH TEQ TST
UBFX UDIV UMLAL UMULL USAT UXTB UXTH
WFE WFI YIELD IT

CORTEX-M3

BKPT BLX ADC ADD ADR
BX CPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
ISB LDR LDRB LDM
MRS LDRH LDRSB LDRSH
MSR LSL LSR MOV
NOP REV MUL MVN ORR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD

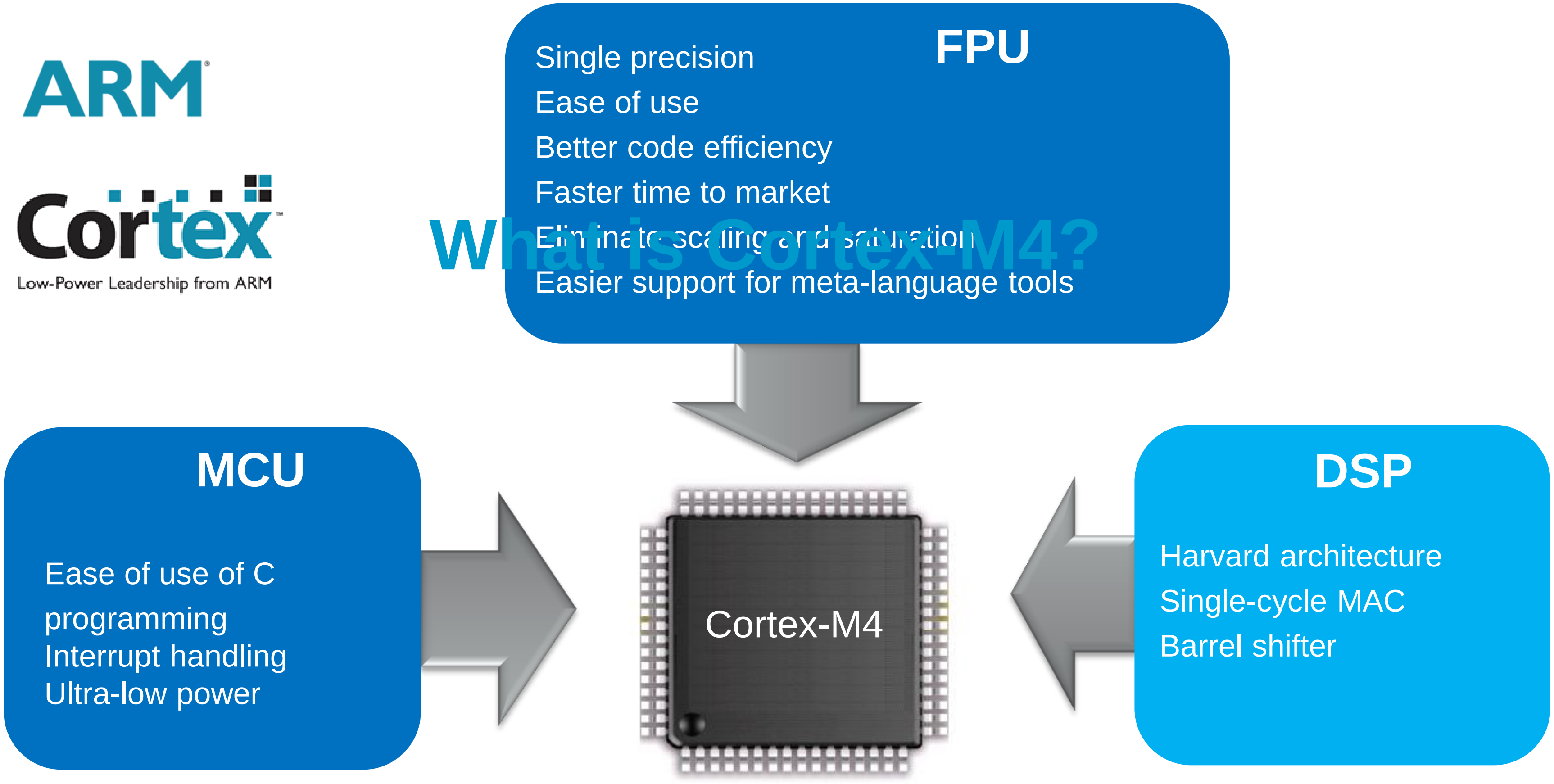
CORTEX-M0/M1

VABS VADD VCMP VCMPE VCVT VCVTR VDIV VLDM VLDR
VMLA VMLS VMOV VMRS VMSR VMUL VNEG VNMLA VNMLS
VNMUL VPOP VPUSH VSQRT VSTM VSTR VSUB

Cortex-M4F

ARM Cortex M4 Core

What is Cortex-M4?



FPU

Single precision
Ease of use
Better code efficiency
Faster time to market
Eliminate scaling and saturation
Easier support for meta-language tools

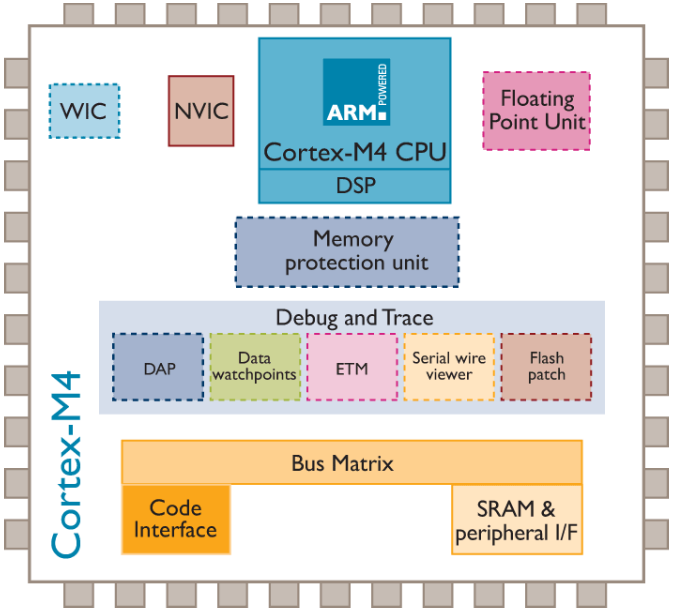
MCU

Ease of use of C programming
Interrupt handling
Ultra-low power

DSP

Harvard architecture
Single-cycle MAC
Barrel shifter

Cortex-M4 processor microarchitecure



- **ARMv7ME Architecture**
  - Thumb-2 Technology
  - DSP and SIMD extensions
  - Single cycle MAC (Up to 32 x 32 + 64 -> 64)
  - Optional 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

Cortex-M feature set comparison

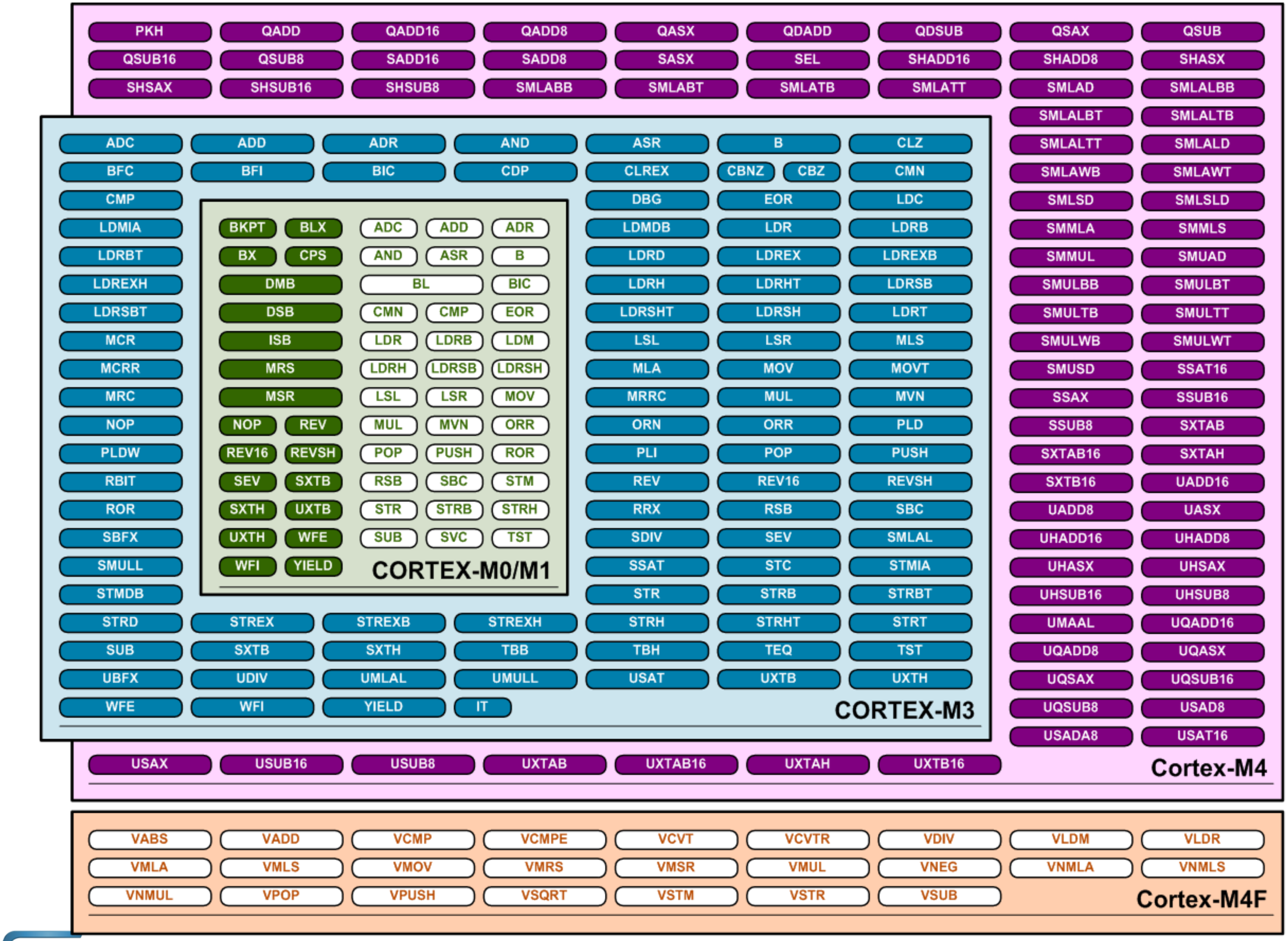
	Cortex-M0	Cortex-M3	Cortex-M4
Architecture Version	V6M	v7M	v7ME
Instruction set architecture	Thumb, Thumb-2 System Instructions	Thumb + Thumb-2	Thumb + Thumb-2, DSP, SIMD, FP
DMIPS/MHz	0.9	1.25	1.25
Bus interfaces	1	3	3
Integrated NVIC	Yes	Yes	Yes
Number interrupts	1-32 + NMI	1-240 + NMI	1-240 + NMI
Interrupt priorities	4	8-256	8-256
Breakpoints, Watchpoints	4/2/0, 2/1/0	8/4/0, 2/1/0	8/4/0, 2/1/0
Memory Protection Unit (MPU)	No	Yes (Option)	Yes (Option)
Integrated trace option (ETM)	No	Yes (Option)	Yes (Option)
Fault Robust Interface	No	Yes (Option)	No
Single Cycle Multiply	Yes (Option)	Yes	Yes
Hardware Divide	No	Yes	Yes
WIC Support	Yes	Yes	Yes
Bit banding support	No	Yes	Yes
Single cycle DSP/SIMD	No	No	**Yes**
Floating point hardware	No	No	**Yes**
Bus protocol	AHB Lite	AHB Lite, APB	AHB Lite, APB
CMSIS Support	Yes	Yes	Yes

life.augmented

Low-Power Leadership from ARM

Cortex M4 – DSP features

Cortex-M processors binary compatible



PKH QADD QADD16 QADD8 QASX QDADD QDSUB QSAX QSUB
QSUB16 QSUB8 SADD16 SADD8 SASX SEL SHADD16 SHADD8 SHASX
SHSAX SHSUB16 SHSUB8 SMLABB SMLABT SMLATB SMLATT SMLAD SMLALBB
SMLALBT SMLALTB
SMLALTT SMLALD
SMLAWB SMLAWT
SMLSD SMLSLD
SMMLA SMMLS
SMMUL SMUAD
SMULBB SMULBT
SMULTB SMULTT
SMULWB SMULWT
SMUSD SSAT16
SSAX SSUB16
SSUB8 SXTAB
SXTAB16 SXTAH
SXTB16 UADD16
UADD8 UASX
UHADD16 UHADD8
UHASX UHSAX
UHSUB16 UHSUB8
UMAAL UQADD16
UQADD8 UQASX
UQSAX UQSUB16
UQSUB8 USAD8
USADA8 USAT16
USAX USUB16 USUB8 UXTAB UXTAB16 UXTAH UXTB16

Cortex-M4

ADC ADD ADR AND ASR B CLZ
BFC BFI BIC CDP CLREX CBNZ CBZ CMN
CMP DBG EOR LDC
LDMIA LDMDB LDR LDRB
LDRBT LDRD LDREX LDREXB
LDREXH LDRH LDRHT LDRSB
LDRSBT LDRSHT LDRSH LDRT
MCR LSL LSR MLS
MCRR MLA MOV MOVT
MRC MRRC MUL MVN
NOP ORN ORR PLD
PLDW PLI POP PUSH
RBIT REV REV16 REVSH
ROR RRX RSB SBC
SBFX SDIV SEV SMLAL
SMULL SSAT STC STMIA
STMDB STR STRB STRBT
STRD STREX STREXB STREXH STRH STRHT STRT
SUB SXTB SXTH TBB TBH TEQ TST
UBFX UDIV UMLAL UMULL USAT UXTB UXTH
WFE WFI YIELD IT

CORTEX-M3

BKPT BLX ADC ADD ADR
BX CPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
ISB LDR LDRB LDM
MRS LDRH LDRSB LDRSH
MSR LSL LSR MOV
NOP REV MUL MVN ORR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD

CORTEX-M0/M1

VABS VADD VCMP VCMPE VCVT VCVTR VDIV VLDM VLDR
VMLA VMLS VMOV VMRS VMSR VMUL VNEG VNMLA VNMLS
VNMUL VPOP VPUSH VSQRT VSTM VSTR VSUB

Cortex-M4F

Cortex
Low-Power Leadership from ARM

life.augmented

Cortex-M4 overview

- Main Cortex-M4 processor features
  - ARMv7-ME architecture revision
    - Fully compatible with Cortex-M3 instruction set
  - Single-cycle multiply-accumulate (MAC) unit
  - Optimized single instruction multiple data (SIMD) instructions
  - Saturating arithmetic instructions
  - Optional single precision Floating-Point Unit (FPU)
  - Hardware Divide (2-12 Cycles), same as Cortex-M3
  - Barrel shifter (same as Cortex-M3)

Single-cycle multiply-accumulate 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 extended single cycle MAC

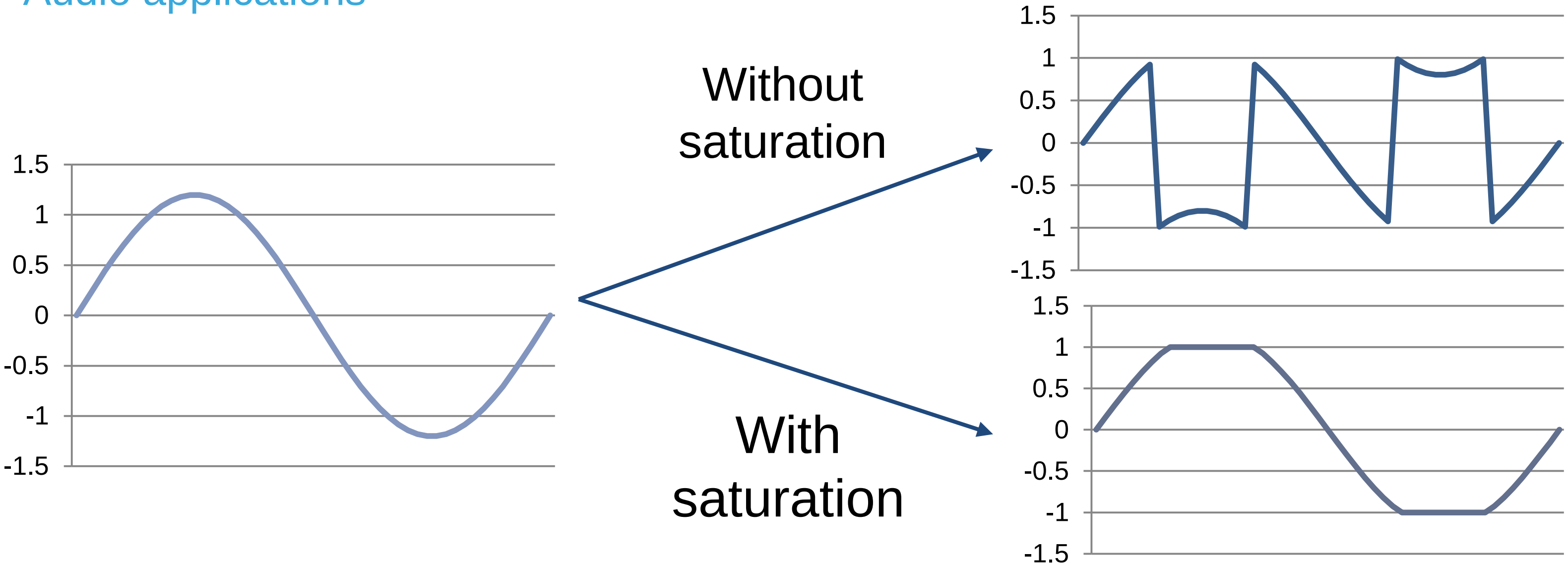
OPERATION	INSTRUCTIONS	CM3	CM4
16 x 16 = 32	SMULBB, SMULBT, SMULTB, SMULTT	n/a	1
16 x 16 + 32 = 32	SMLABB, SMLABT, SMLATB, SMLATT	n/a	1
16 x 16 + 64 = 64	SMLALBB, SMLALBT, SMLALTB, SMLALTT	n/a	1
16 x 32 = 32	SMULWB, SMULWT	n/a	1
(16 x 32) + 32 = 32	SMLAWB, SMLAWT	n/a	1
(16 x 16) ± (16 x 16) = 32	SMUAD, SMUADX, SMUSD, SMUSDX	n/a	1
(16 x 16) ± (16 x 16) + 32 = 32	SMLAD, SMLADX, SMLSD, SMLSDX	n/a	1
(16 x 16) ± (16 x 16) + 64 = 64	SMLALD, SMLALDX, SMLSLD, SMLSLDX	n/a	1
32 x 32 = 32	MUL	1	1
32 ± (32 x 32) = 32	MLA, MLS	2	1
32 x 32 = 64	SMULL, UMULL	5-7	1
(32 x 32) + 64 = 64	SMLAL, UMLAL	5-7	1
(32 x 32) + 32 + 32 = 64	UMAAL	n/a	1
32 ± (32 x 32) = 32 (upper)	SMMLA, SMMLAR, SMMLS, SMMLSR	n/a	1
(32 x 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
  - Audio applications



Without saturation

With saturation

  - 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

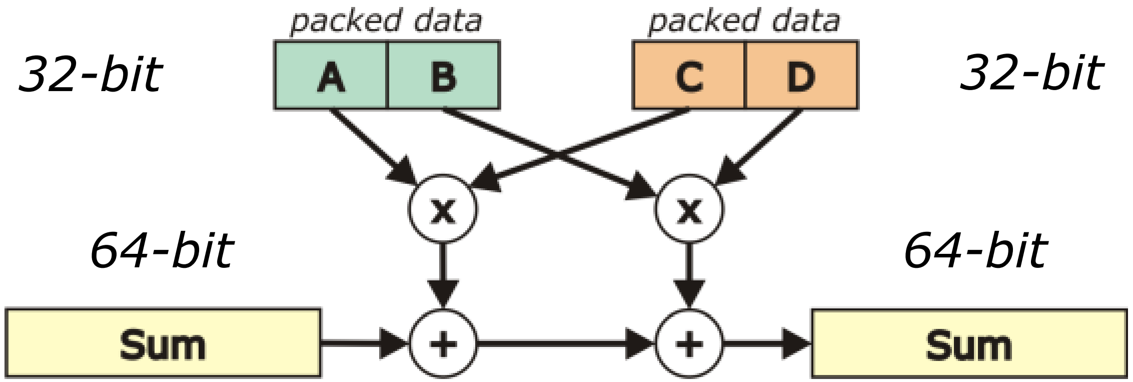
Single-cycle SIMD instructions

- 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 (Result = 16x16 + 16x16 + 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

  Sum = Sum + (A x C) + (B x D)



- SIMD techniques operate with packed data

Cortex-M4 DSP instructions compared

		Cycle counts	
CLASS	INSTRUCTION	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, SMLSLD	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

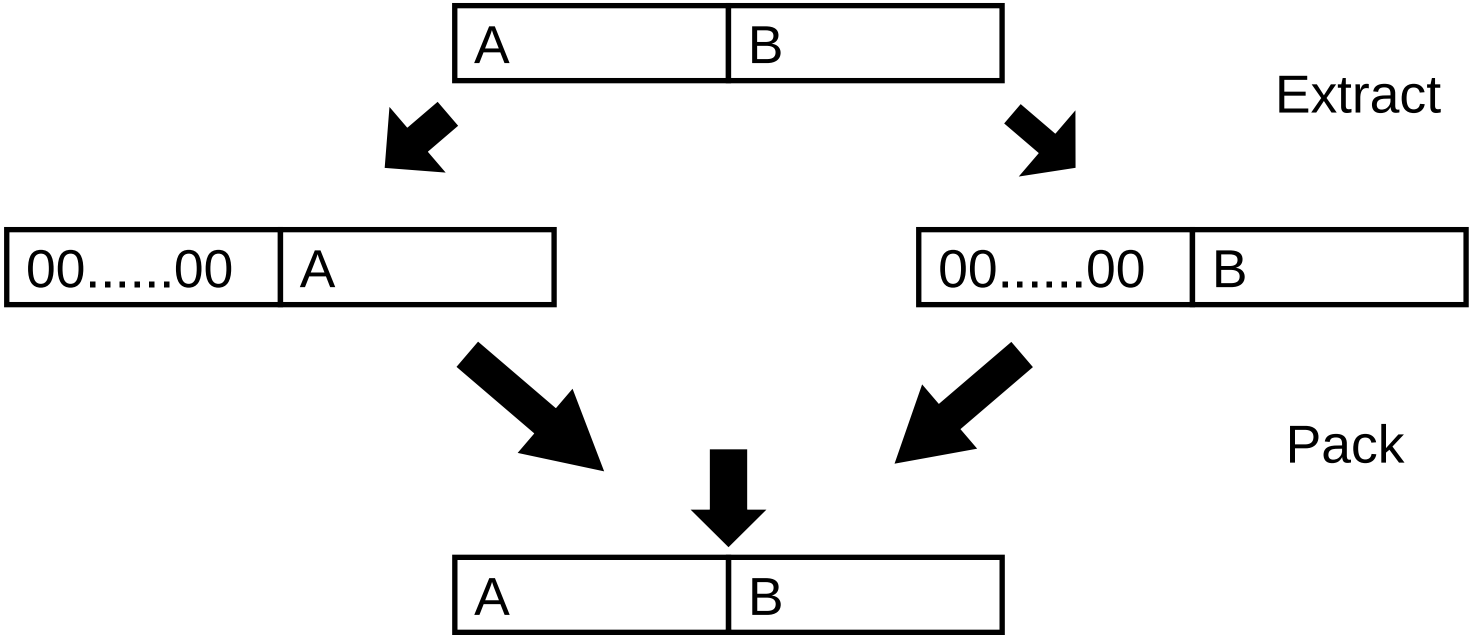
Single cycle MAC

Cortex-M4 non–DSP instructions

		Cycle counts	
CLASS	**INSTRUCTION**	**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, SXTH, 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

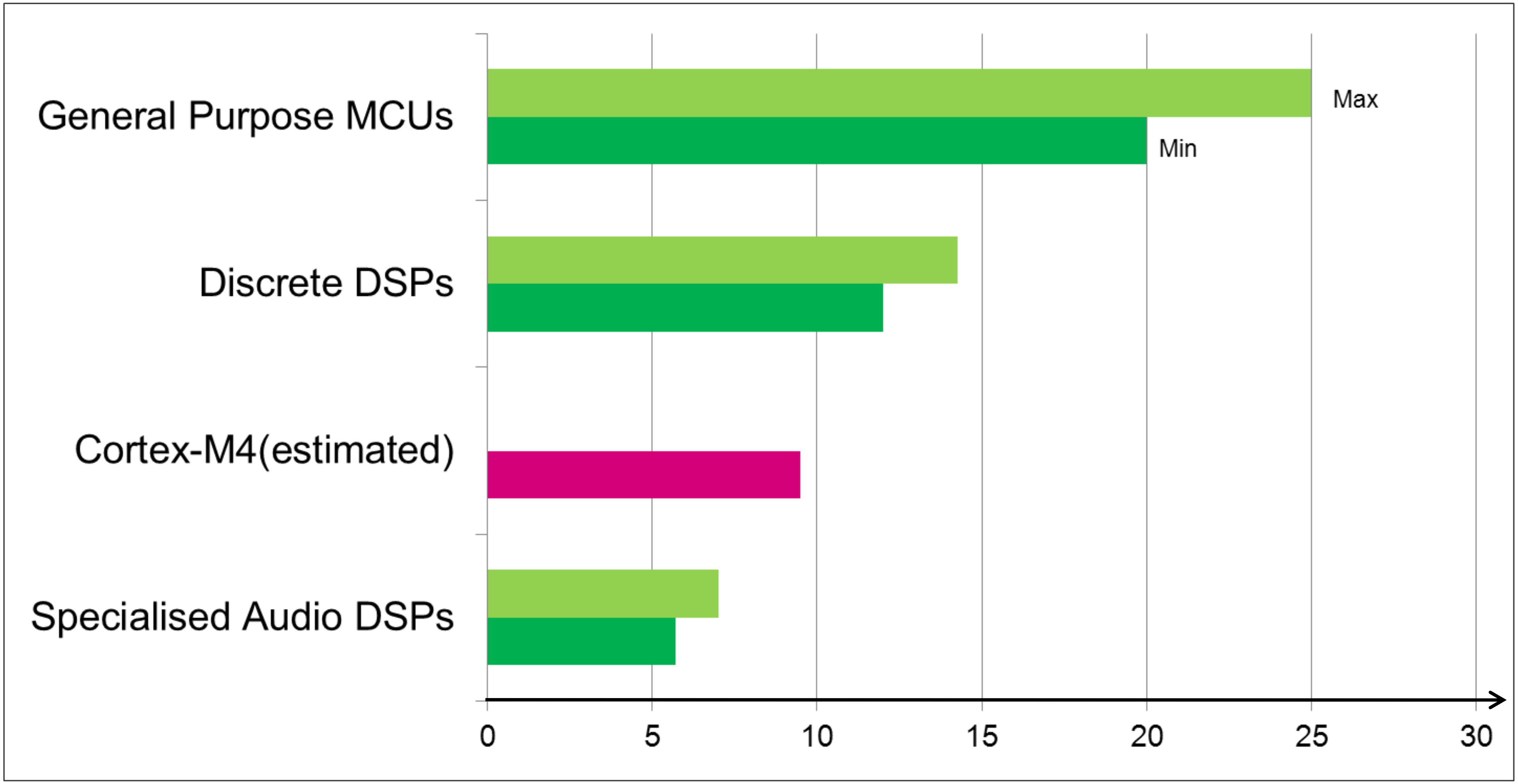


DSP performances for control application

- 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 !)

DSP concept from ARM (*)

DSP lib provided for free by ARM

- The benefits of software libraries for Cortex-M4
  - Enables end user to develop applications faster
    - Keeps end user abstracted from low level programming
  - Benchmarking vehicle during system development
  - Clear competitive positioning against incumbent DSP/DSC offerings
  - Accelerate third party software development

- Keeping it easy to access for end user
  - Minimal entry barrier - very easy to access and use

- One standard library – no duplicated efforts
  - ARM channels effort/resources with software partner
  - Value add through another level of software – eg: filter config tools

DSP lib function list 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
- Support functions – copy/fill arrays, data type conversions etc

Tools

- Matlab / Simulink
  - Embedded coder for code generation
  - Mathworks
    - Demo being developed (availability end of year)
  - Aimagin (Rapidstm32)

- Filter design tools
  - Lot of tools available, most of them commercial product, some with low-cost offer, few free
    - http://www.dspguru.com/dsp/links/digital-filter-design-software

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 !

Mathematical details

- FIR Filter

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

- IIR or recursive filter

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

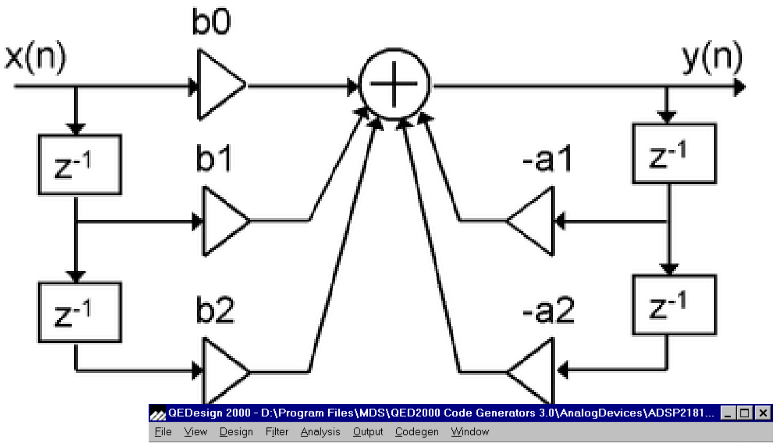
- FFT Butterfly (radix-2)

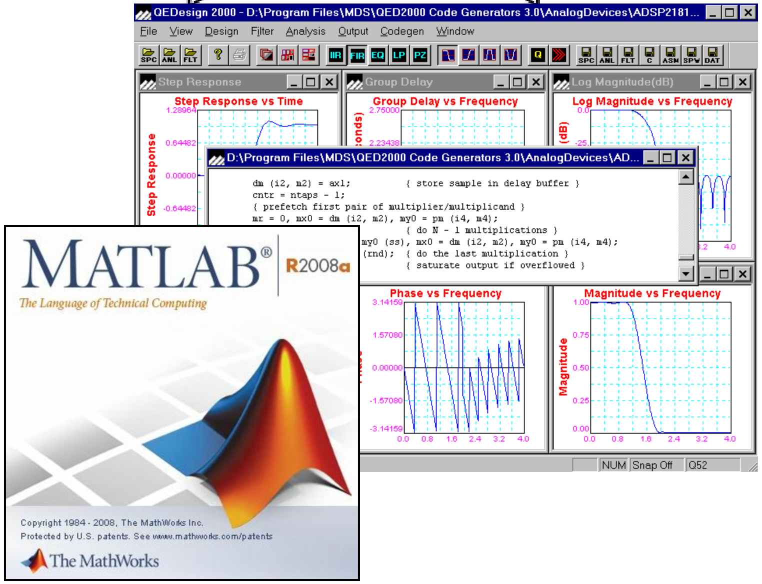
$$Y[k_1]=X[k_1]+X[k_2]$$
$$Y[k_2]=(X[k_1]-X[k_2])e^{-j\omega}$$

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




QEDesign 2000 - D:\Program Files\MDS\QED2000 Code Generators 3.0\AnalogDevices\ADSP2181...
File View Design Filter Analysis Output Codegen Window
SPC ANL FLT
IIR FIR EQ LP PZ
SPC ANL FLT C ASM SPW DAT
Step Response
Step Response vs Time
Group Delay
Group Delay vs Frequency
Log Magnitude(dB)
Log Magnitude vs Frequency
Phase vs Frequency
Magnitude vs Frequency
NUM Snap Off Q52
D:\Program Files\MDS\QED2000 Code Generators 3.0\AnalogDevices\AD...
MATLAB® R2008a
The Language of Technical Computing
The MathWorks

```
dm (i2, m2) = axl;          { store sample in delay buffer }
cntr = ntaps - 1;
{ prefetch first pair of multiplier/multiplicand }
mr = 0, mx0 = dm (i2, m2), my0 = pm (i4, m4);
                            { do N - 1 multiplications }
my0 (ss), mx0 = dm (i2, m2), my0 = pm (i4, m4);
(rnd);  { do the last multiplication }
                            { saturate output if overflowed }
```

Copyright 1984 - 2008, The MathWorks Inc.
Protected by U.S. patents. See www.mathworks.com/patents

IIR – single cycle MAC benefit

	Cortex-M3 cycle count	Cortex-M4 cycle count
`xN = *x++;`	2	2
`yN = xN * b0;`	3-7	1
`yN += xNm1 * b1;`	3-7	1
`yN += xNm2 * b2;`	3-7	1
`yN -= yNm1 * a1;`	3-7	1
`yN -= yNm2 * a2;`	3-7	1
`*y++ = yN;`	2	2
`xNm2 = xNm1;`	1	1
`xNm1 = xN;`	1	1
`yNm2 = yNm1;`	1	1
`yNm1 = yN;`	1	1
Decrement loop counter	1	1
Branch	2	2

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

- 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
- 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

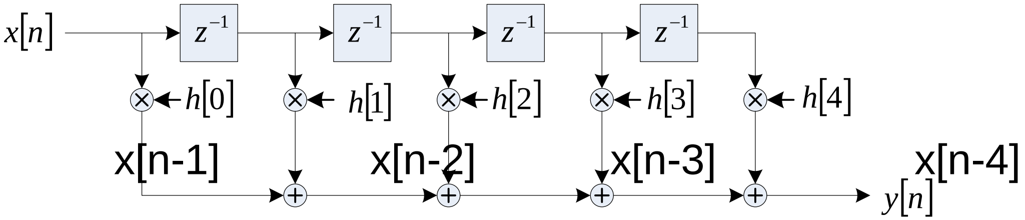
FIR Filter

- Occurs frequently in communications, audio, and video applications

- A filter of length N requires
  - N coefficients h[0], h[1], …, h[N-1]
  - N state variables x[n], x[n-1], …, x[n-(N-1)]
  - N multiply accumulates

- Classic function in signal processing

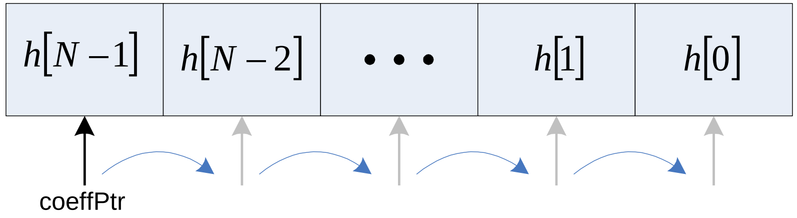
$$y[n]=\sum_{k=0}^{N-1} h[k]x[n-k]$$



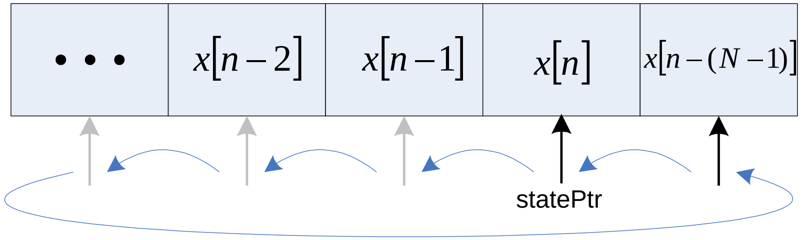
Circular Addressing

- 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

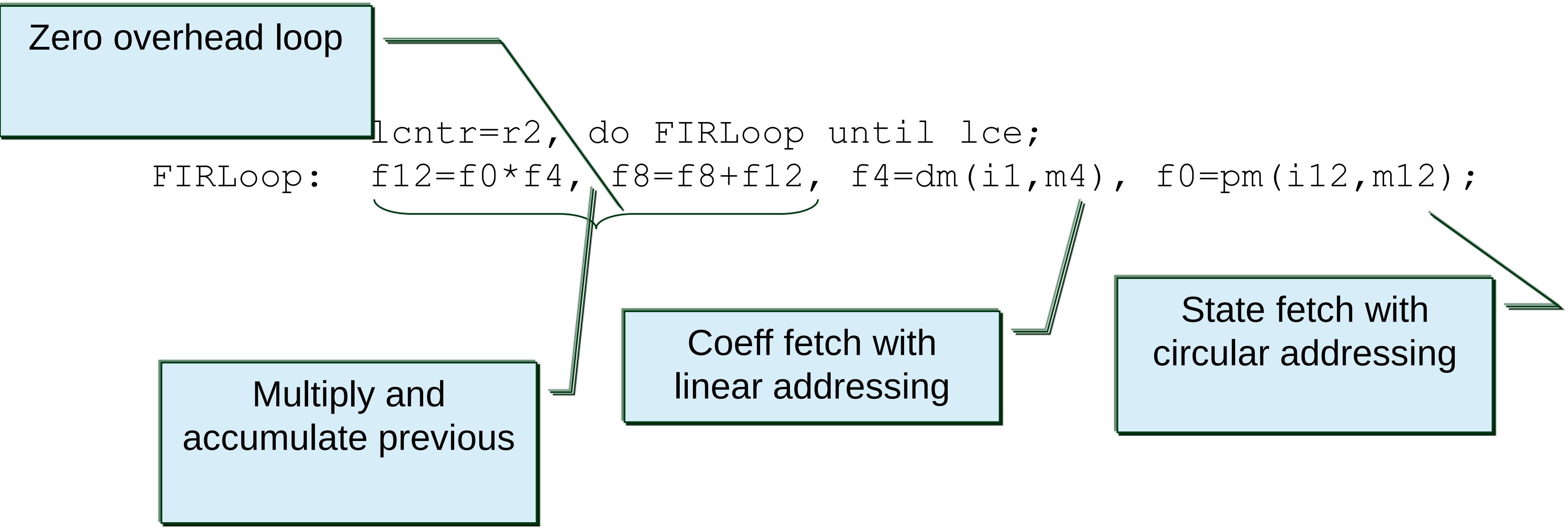
```
void fir(q31_t *in, q31_t *out, q31_t *coeffs, int *stateIndexPtr,
                    int filtLen, int blockSize)
{
  int sample;
  int k;
  q31_t sum;
  int stateIndex = *stateIndexPtr;

  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



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

Zero overhead loop

Multiply and accumulate previous

Coeff fetch with linear addressing

State fetch with circular 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

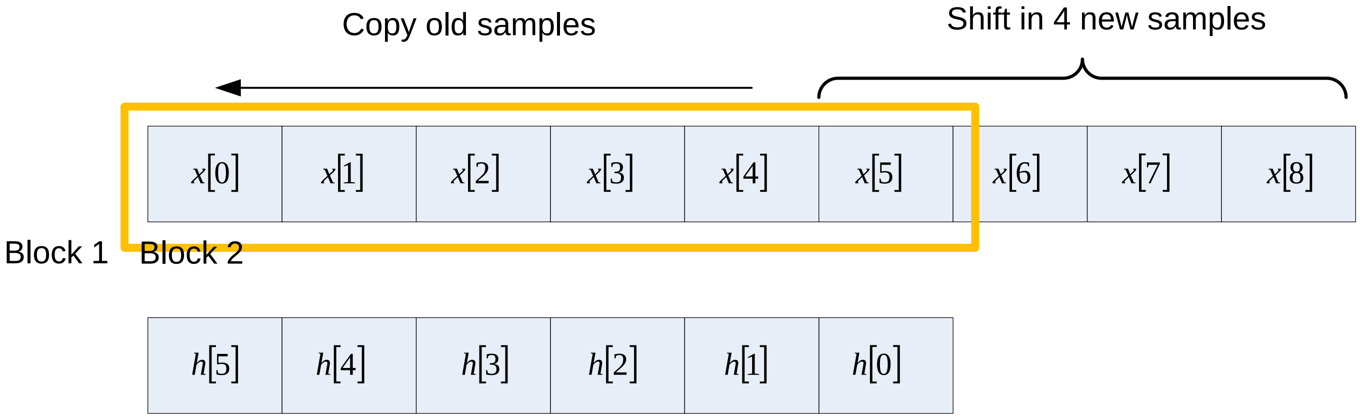
Total	12 cycles

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

How can this be improved on the Cortex-M4 ?

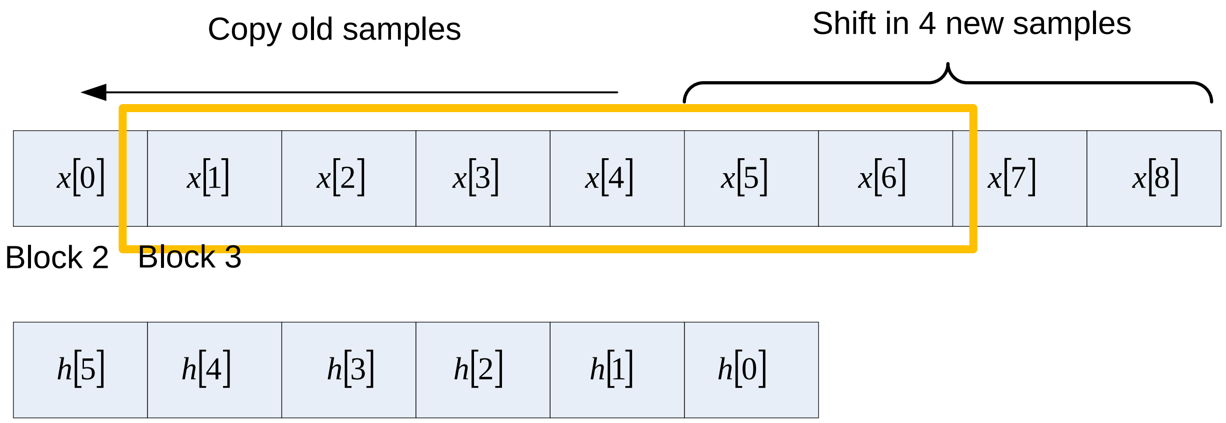
Circular addressing alternative

- 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.



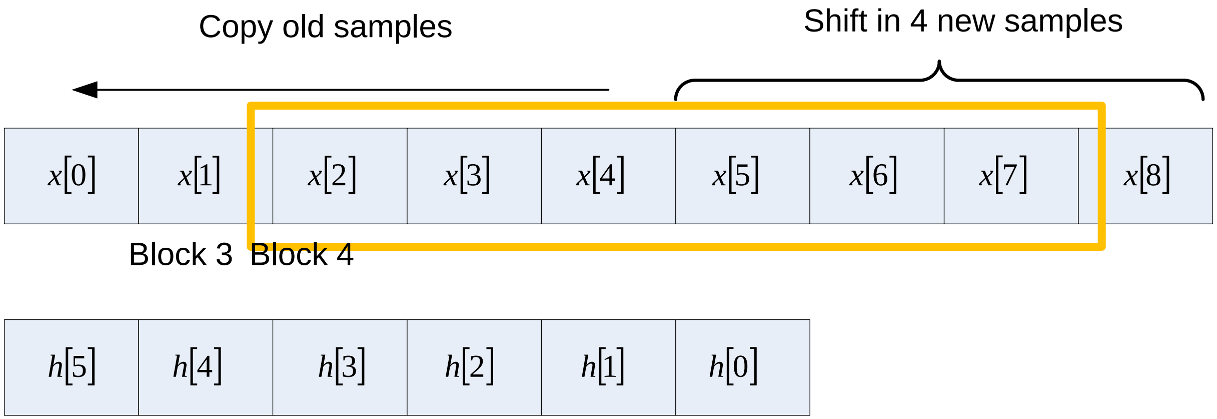
Circular addressing alternative

- 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 addressing 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

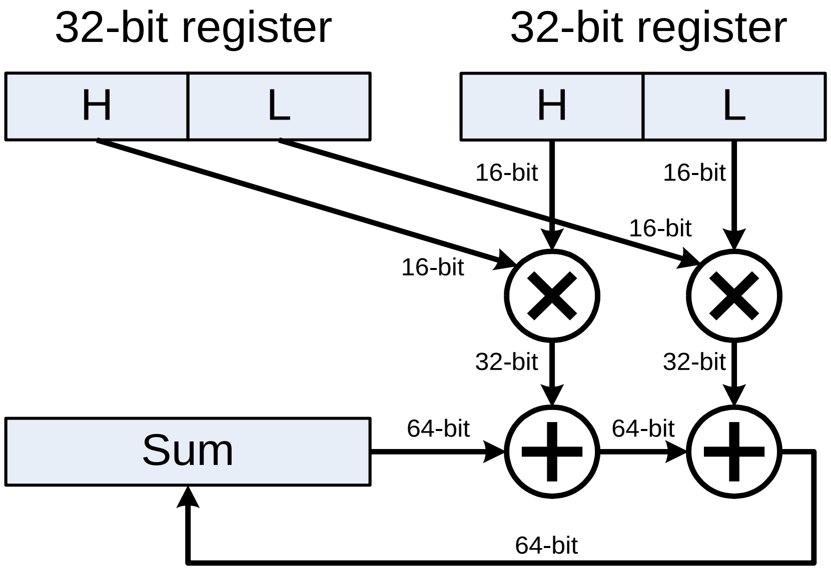
Lets try SIMD

Apply SIMD

- Many image, video processing, and communications applications use 8- or 16-bit data types.

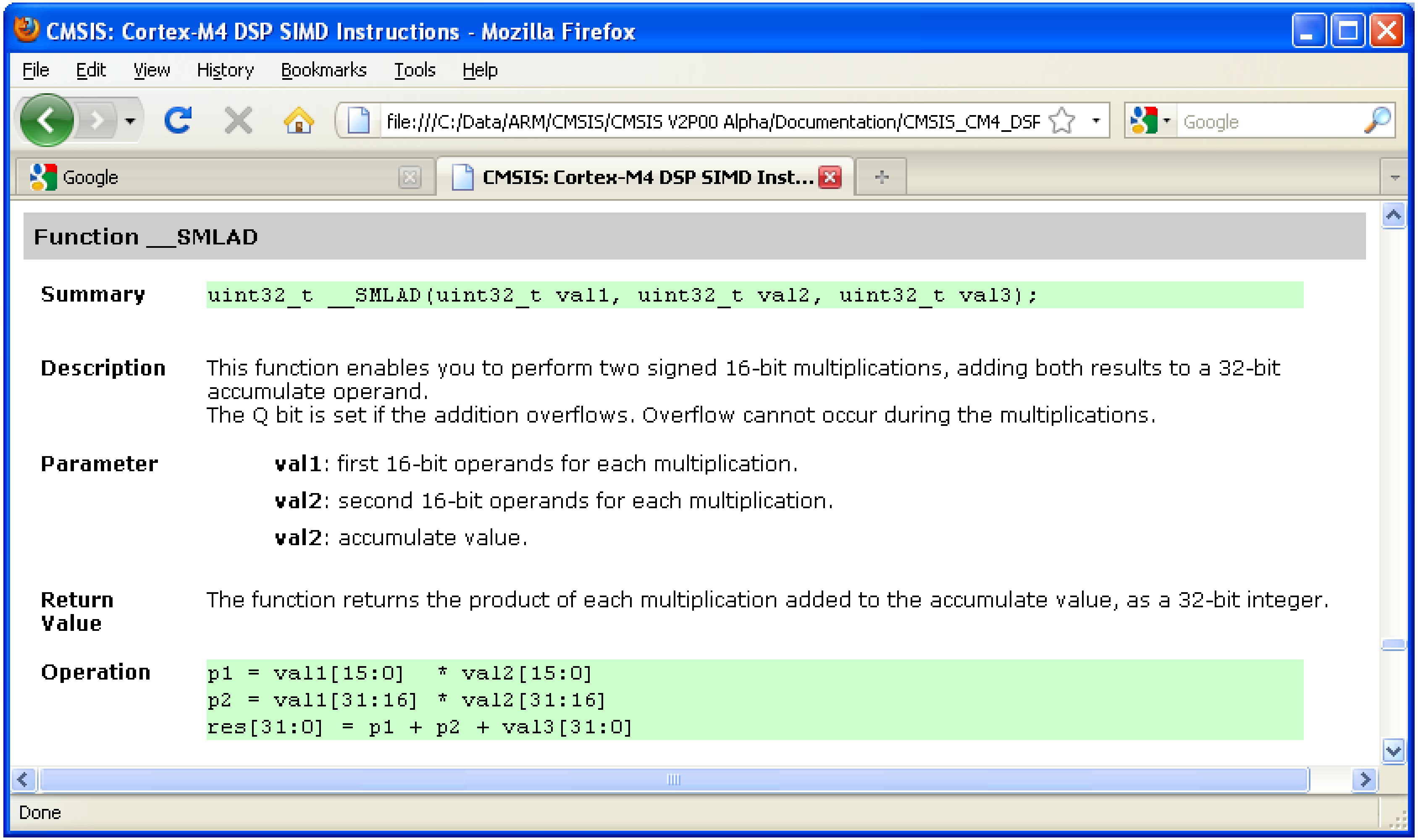
- SIMD speeds these up
  - 16-bit data yields a 2x speed improvement over 32-bit
  - 8-bit data yields a 4x speed improvement

- Access to SIMD is via compiler intrinsics



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

CMSIS Files


CMSIS: Cortex-M4 DSP SIMD Instructions - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Data/ARM/CMSIS/CMSIS V2P00 Alpha/Documentation/CMSIS_CM4_DSF
Google
Google
CMSIS: Cortex-M4 DSP SIMD Inst...
Done

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

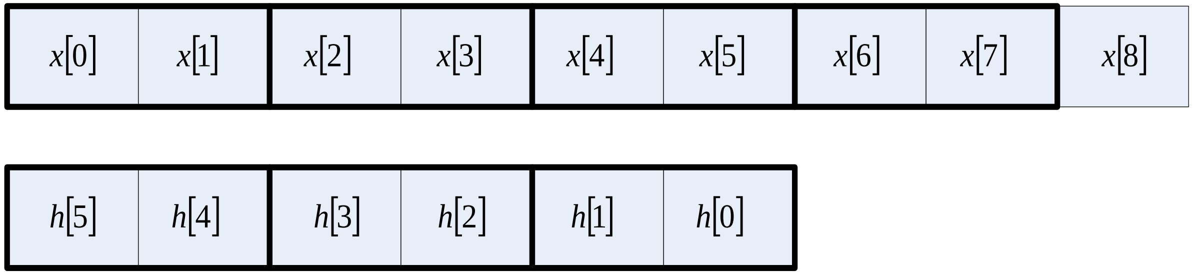
The function returns the product of each multiplication added to the accumulate value, as a 32-bit integer.

Operation

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

Data organization with SIMD

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



$x[0]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$x[6]$	$x[7]$	$x[8]$

$h[5]$	$h[4]$	$h[3]$	$h[2]$	$h[1]$	$h[0]$

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?

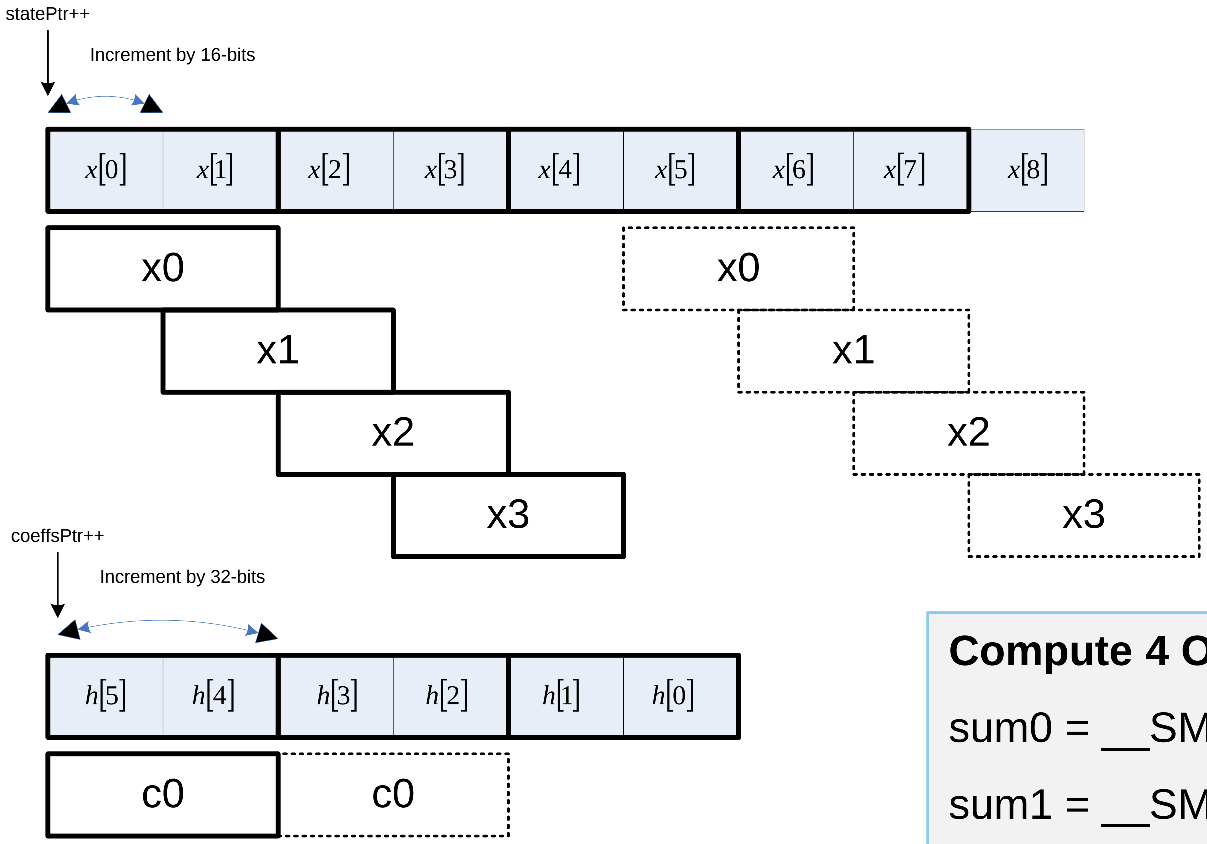
One more idea left

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!

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

Quick introduction to fixed point data format

- Fixed point format can be integer, fractional or a mix of integer and fractional.

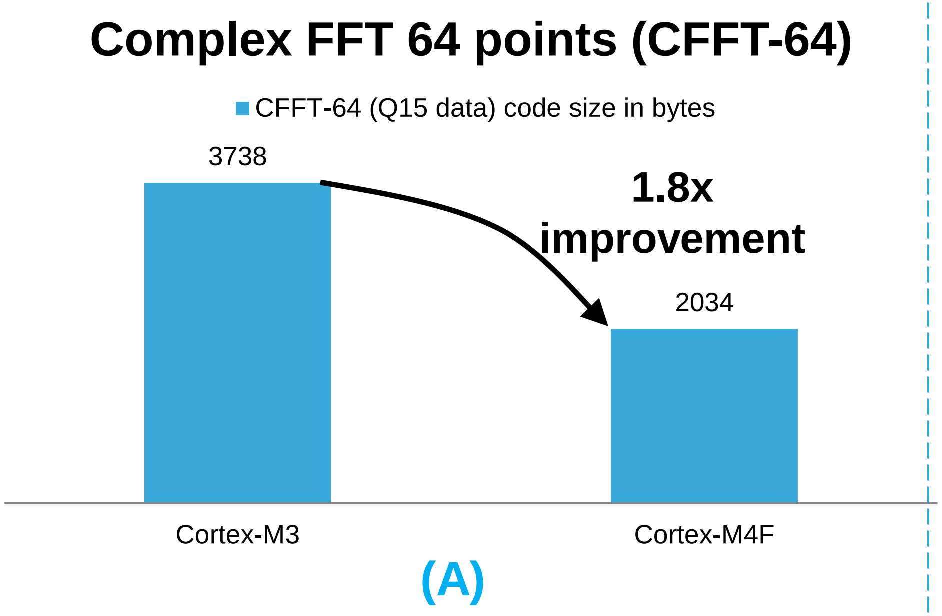
- Fixed point use Qx.y notation
  - X : number of integer bits
  - Y: number of fractional bits

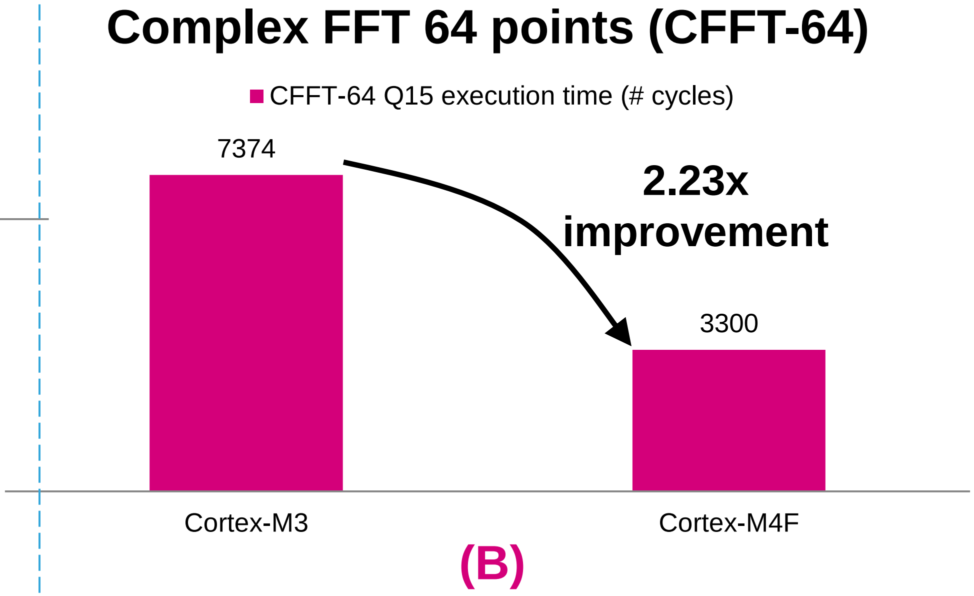
- Q2.13 denotes fixed point data type with 2 bits for integer and 13 bits for fractional part.

- Fixed point format used in CMSIS DSP library is Q0.7 (Q7), Q0.15 (Q15) and Q0.31 (Q31)
  - Only fractional bits to represent numbers between -1.0 and 1.0.
  - Value $= \sum b_{(15-i)} * 2^{(-1*i)}$: with i = 1..15
  - Example: 0.25 is represented as 0x2000 in Q15 format.

Cortex-M4F benefits

- Cortex-M4F benefits Vs. Cortex-M3
  - Improvement in code size (A)
  - Improvement in performance (B)



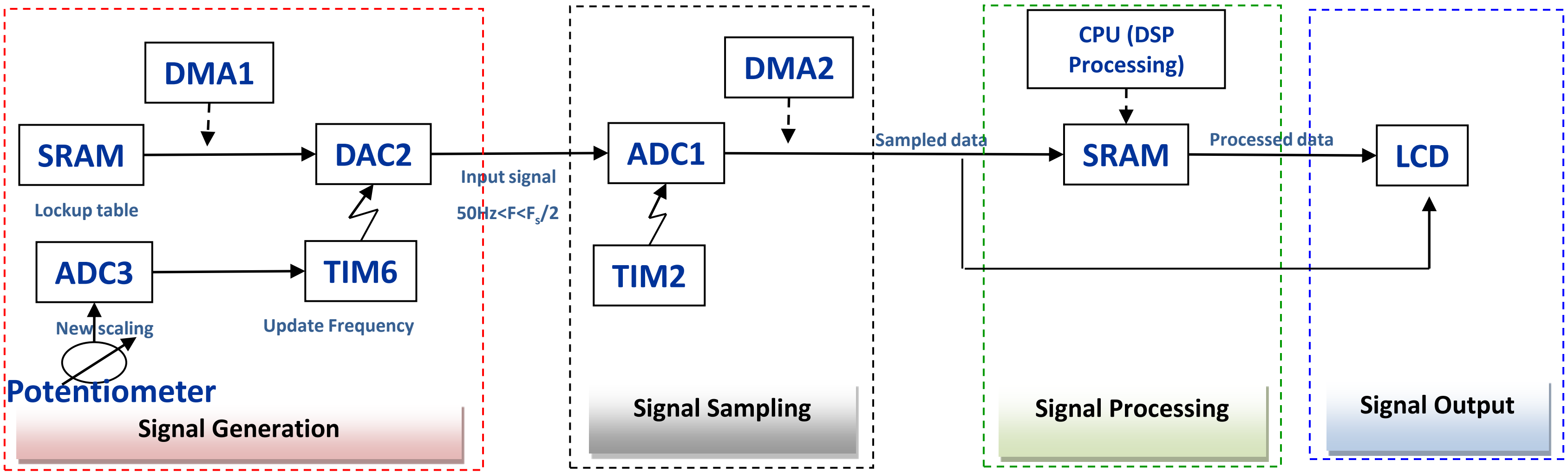


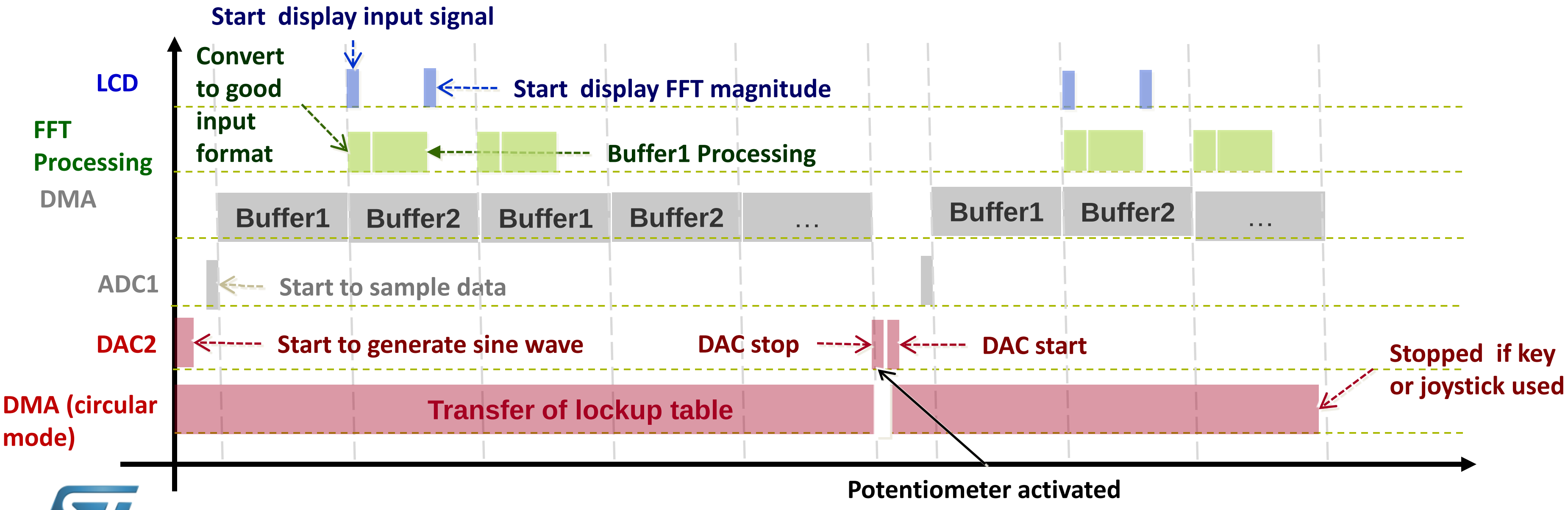
Fixed point DSP examples

- We will provide an overview on the new ARM CMSIS DSP library & give example of performance of **FIR** (Finite impulse response) filtering and **FFT** (Fast Fourier transform) with STM32F2, STM32F3 and STM32F4.

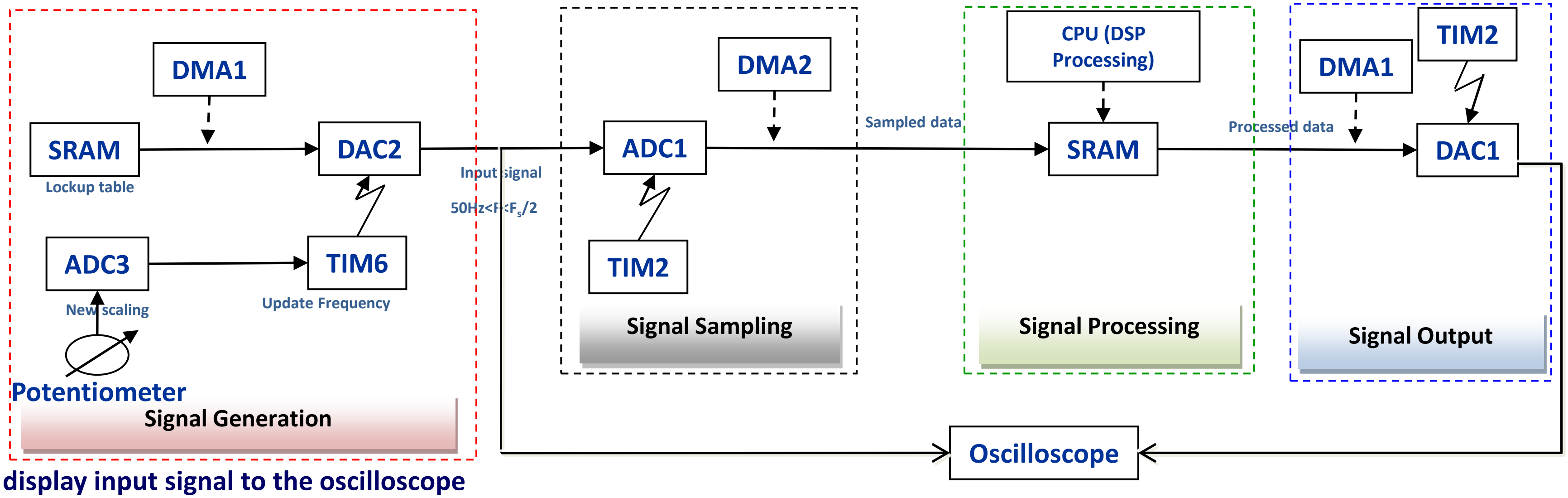
- FIR & FFT Examples
  - Benchmarking setup
  - Benchmarking results

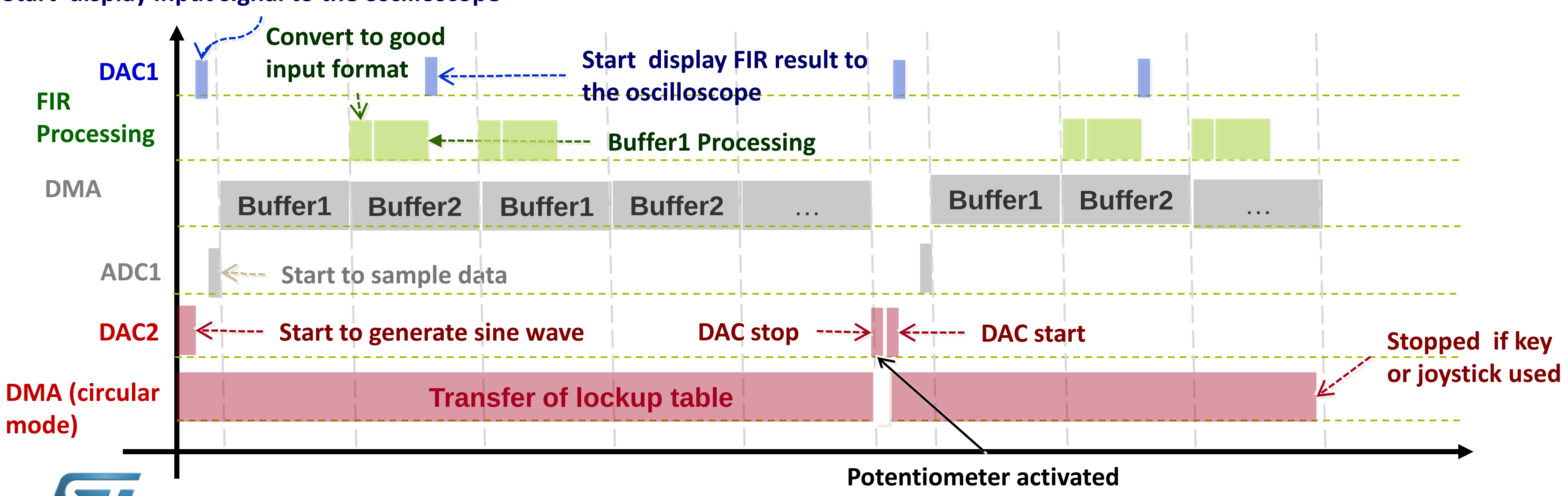
FFT: Hardware setup & data flow





FIR: Hardware setup & data flow





FFT benchmark results
Setup

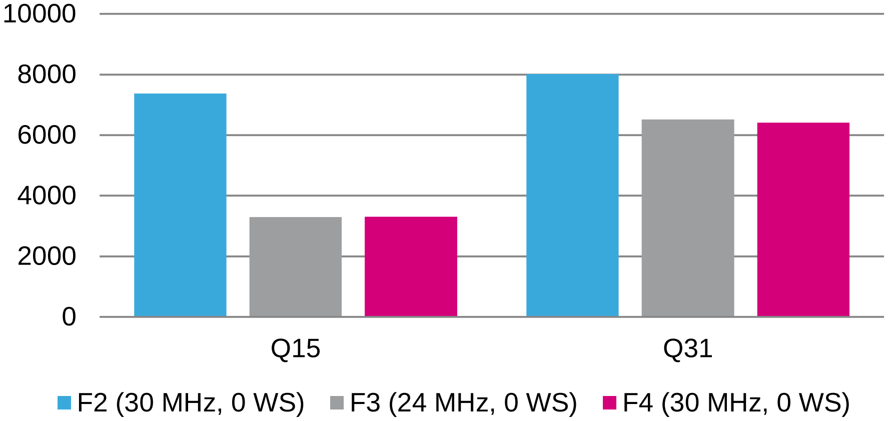
Input signal characteristics	
Input signal	a sine wave with a frequency F 50Hz <F<4KHz
Sampling frequency	8KHz

Measures done with MDK-ARM (4.23.00.0) toolchain

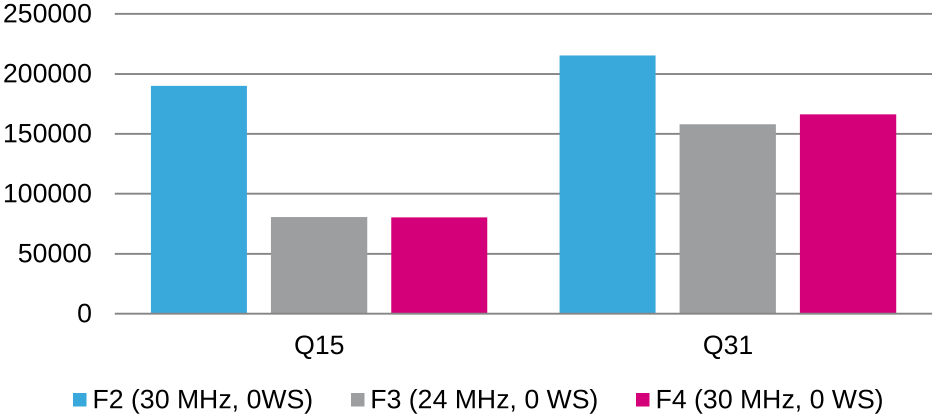
Level 3(-O3) for time optimization without MicroLib

FFT benchmark results
Cortex-M3 vs Cortex-M4F

FFT 64-points average processing time (# cycles)



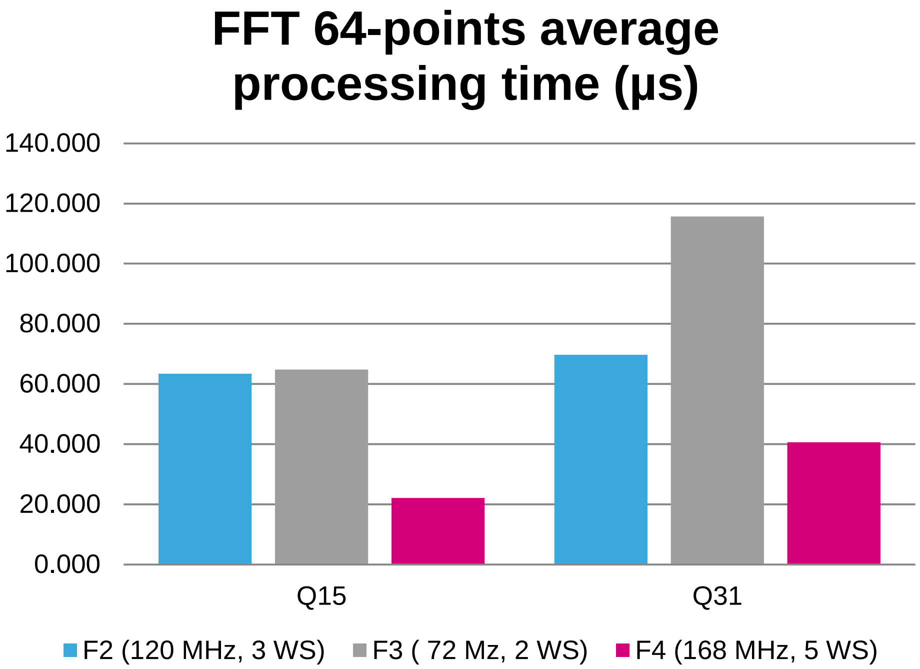
FFT 1024-points average processing time at 0 WS (# cycles)

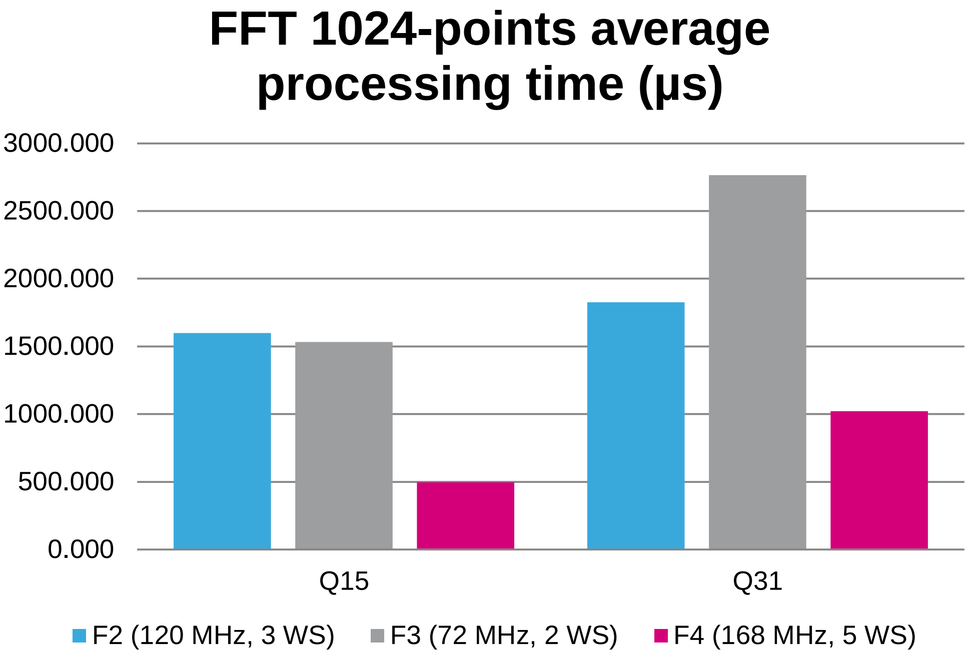


		F2(30MHz,0WS) (#cycles)	F3(24 MHz,0WS) (#cycles)	Gain (F2 vs F3)	F4(30MHz,0WS) (#cycles)	F3(24 MHz,0WS) (#cycles)
FFT (64-points)	Q15	7374	3300	x2.23	3307	3300
	Q31	8022	6522	x1.23	6410	6522
FFT (1024-points)	Q15	190028	80608	x2.36	80252	80608
	Q31	215505	158022	x1.36	166406	158022

life.augmented

FFT benchmarking results F2/F3/F4





		F2(120MHz,3WS) (µs)	F3 (72 MHz, 2WS) (µs)	F4(168MHz,5WS) (µs)	Gain F4/F3
FFT (64-points)	Q15	63.442	64.847	22.101	x 2.9
	Q31	69.683	115.694	40.679	x 2.8
FFT (1024-points)	Q15	1600.067	1532.139	496.952	x 3.08
	Q31	1825.642	2765.861	1021.208	x 2.7

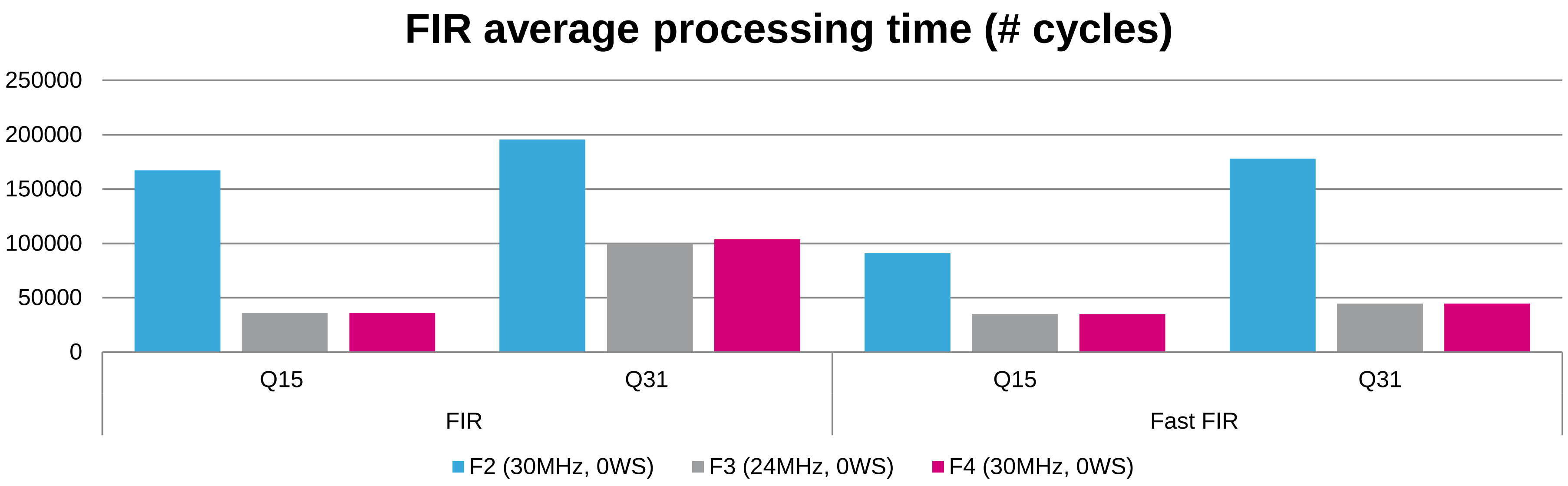
life.augmented

FIR benchmarking results
Setup

Filter & input signal characteristics	
Filter type	Stop Band
Filter order	165
Filter coefficients	166
Cut-off frequency	F_{STOP1}=1.9KHz, F_{STOP2}=2.1KHz
Sampling frequency	48KHz
Number of samples	128
Input signal	a sine wave with a frequency F 50Hz <F<4KHz

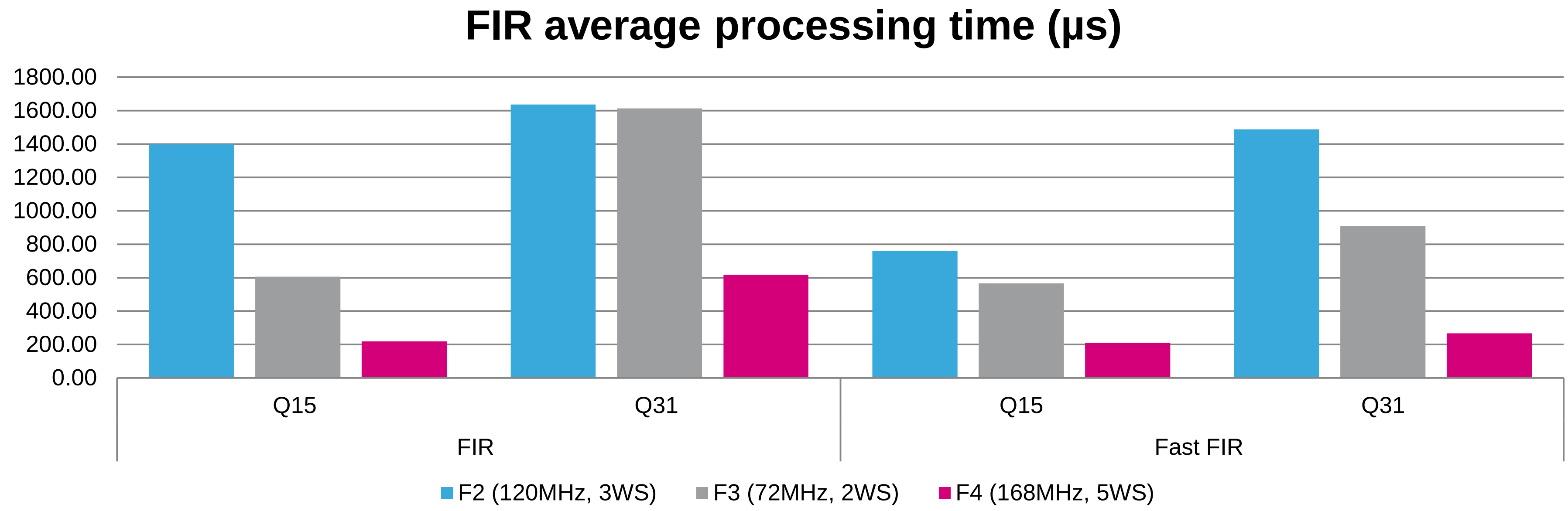
Measures done with MDK-ARM (4.23.00.0) toolchain
Level 3(-O3) for time optimization without MicroLib

FIR benchmarking results F2/F3/F4



		F2(30MHz,0WS) (#cycles)	F3(24 MHz,0WS) (#cycles)	Gain (F2 vs F3)	F4(30MHz,0WS) (#cycles)	F3(24 MHz,0WS) (#cycles)
FIR	Q15	167284	36339	x4.60	36374	36339
	Q31	195537	99861	x1.96	103745	99861
Fast FIR	Q15	90955	34916	x2.60	35079	34916
	Q31	177917	44599	x3.99	44736	44599

FIR benchmarking results F2/F3/F4



		F2(120MHz,3 WS)	Processing time per Tap	F3(72 MHz, 2 WS)	Processing time per Tap	F4(168MHz,5 WS)	Processing time per Tap
FIR	Q15	1396.99 µs	66ns	605.81 µs	28.5 ns	218.28 µs	10 ns
	Q31	1636.66 µs	77ns	1613.72 µs	76 ns	618.27 µs	29 ns
Fast FIR	Q15	760.68 µs	36ns	566.33 µs	26 ns	209.76 µs	10 ns
	Q31	1488.29 µs	70ns	907.73 µs	42 ns	267.46 µs	13 ns

CONTENTS

- **Cortex-M4F (DSP and Floating point Unit)**
  - **Cortex-M4 and DSP features**
  - **Floating point unit**

Floating Point Unit

Overview

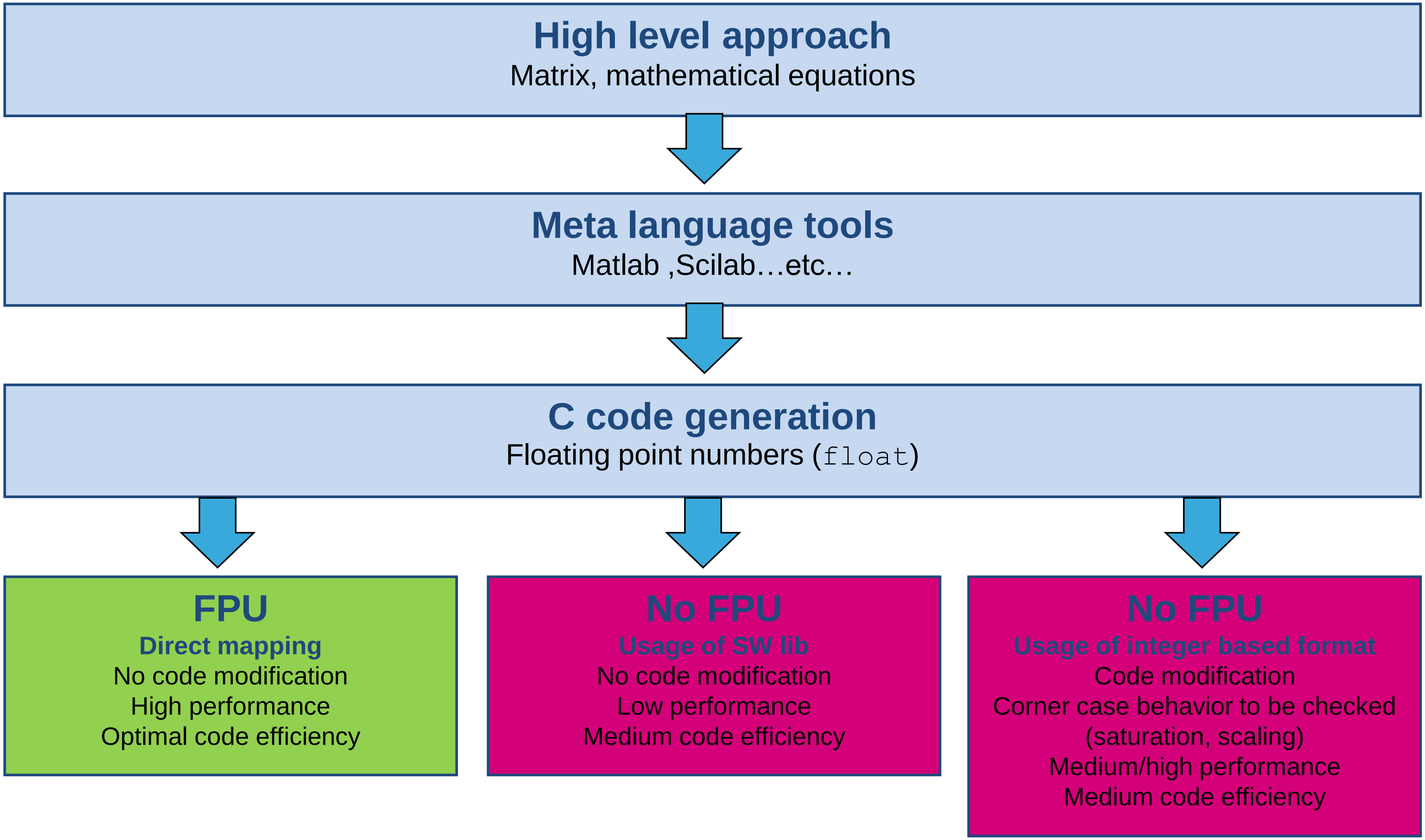
- **FPU : Floating Point Unit**
  - Handles “real” number computation
  - Standardized by **IEEE.754-2008**
    - Number format
    - Arithmetic operations
    - Number conversion
    - Special values
    - 4 rounding modes
    - 5 exceptions and their handling

- **ARM Cortex-M FPU ISA**
  - Supports
    - Add, subtract, multiply, divide
    - Multiply and accumulate
    - Square root operations

Floating Point Unit

- **Introduction**
  - FPU usage
  - Historical perspective
  - Benefit of floating point arithmetic
  - Example & performances
  - Rounding issues
- IEEE 754
- ARM FPv4-SP Single Precision FPU

FPU usage



High level approach
Matrix, mathematical equations

Meta language tools
Matlab ,Scilab…etc…

C code generation
Floating point numbers (`float`)

FPU
Direct mapping
No code modification
High performance
Optimal code efficiency

No FPU
Usage of SW lib
No code modification
Low performance
Medium code efficiency

No FPU
Usage of integer based format
Code modification
Corner case behavior to be checked
(saturation, scaling)
Medium/high performance
Medium code efficiency

Historical perspective

- Usage of floating point as always been a **need** for computers since the beginning (Konrad Zuse - 1935)
- But the **complexity of implementation** discarded their usage during decades (IBM 704 - 1956)
- Floating point unit where implemented in mainframes with various coding techniques depending of the manufacturer
- IBM PC where designed to have floating point capabilities through optional **arithmetic coprocessors** (80x87 series)
- The standardization of floating point coding was done in the 80’s through the **IEEE 754** standard in 1985
- The Intel 80387 was the first intel coprocessor to implement the full IEEE 754 standard in 1987

Benefits of a Floating-Point Unit

- **FPU allows to handled “real” numbers (C float) without penalty**

- **If no FPU**
  - Need to emulate it by software
  - Need to rework all its algorithm and fixed point implementation to handle scaling and saturation issues

- **FPU eases usage of high-level design tools (MatLab/Simulink)**
  - Now part of microcontroller development flow for advanced applications.
  - Derivate code directly using native floating point leads to :
    - quicker time to market (faster development)
    - easy code maintenance
    - more reliable application code as no post modification are needed (no critical scaling operations to move to fixed point)

C language example

```
float function1(float number1, float number2)
{       float temp1, temp2;
        temp1 = number1 + number2;
        temp2 = number1/temp1;
        return temp2;
}
```

Code compiled on Cortex-M3

```
# float function1(…)
# { …
#    temp1 = number1 + number2;
     MOVS      R1,R4
     BL        __aeabi_fadd
     MOVS      R1,R0
#    temp2 = number1/temp1;
     MOVS      R0,R4
     BL        __aeabi_fdiv
#    return temp2;
     POP       {R4,PC}
# }
```

Same code compiled on Cortex-M4F

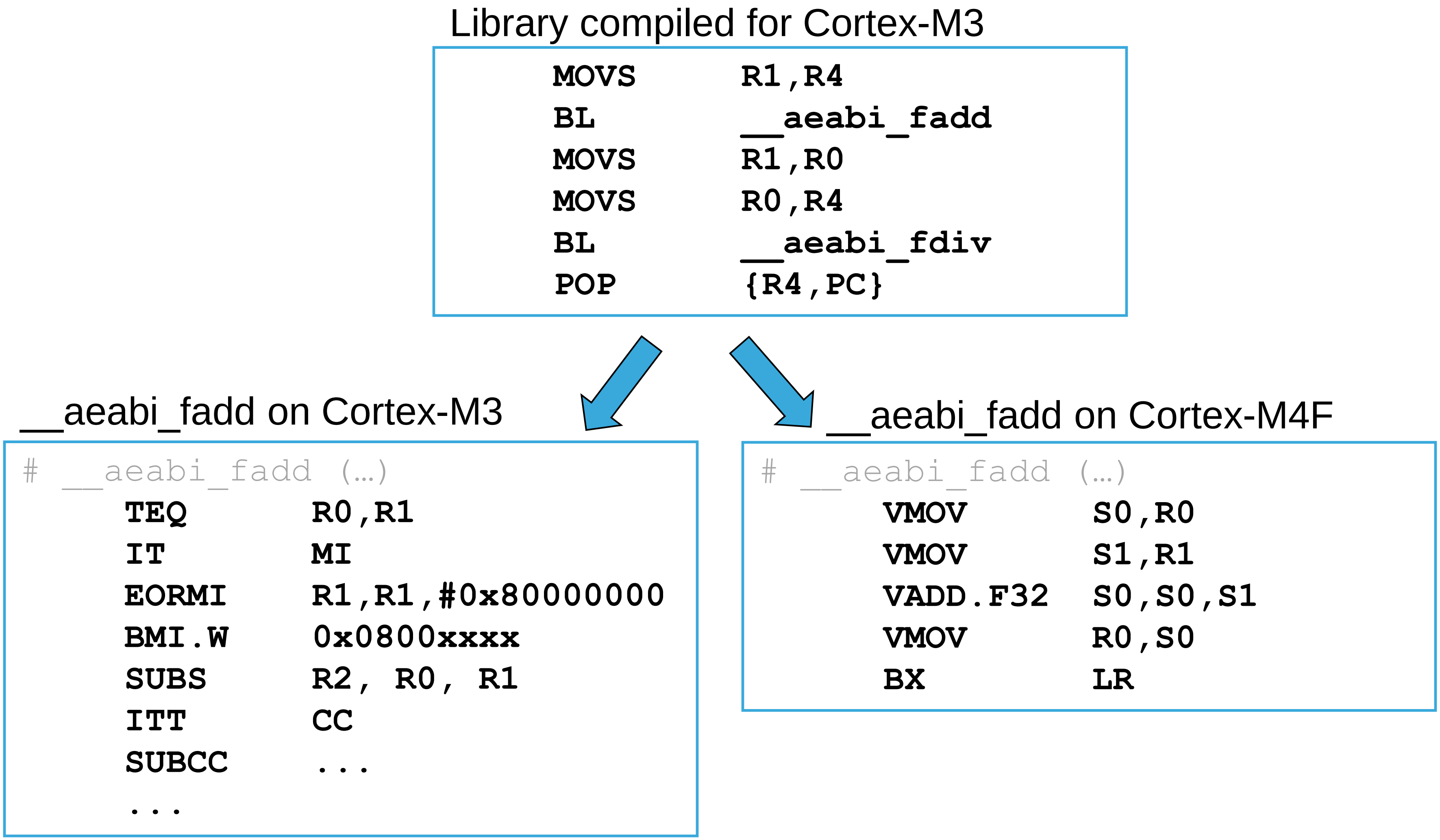
```
float function1(…)
# { …
#    temp1 = number1 + number2;
     VADD.F32 S1,S0,S1
#    temp2 = number1/temp1;
     VDIV.F32 S0,S0,S1
#
#    return temp2;
     BX        LR
# }
```

FPU assembly instructions

Call Soft-FPU (keil's software library)

life.augmented

Binary library example



Library compiled for Cortex-M3

```
MOVS       R1,R4
BL         __aeabi_fadd
MOVS       R1,R0
MOVS       R0,R4
BL         __aeabi_fdiv
POP        {R4,PC}
```

__aeabi_fadd on Cortex-M3

```
# __aeabi_fadd (…)
     TEQ        R0,R1
     IT         MI
     EORMI      R1,R1,#0x80000000
     BMI.W      0x0800xxxx
     SUBS       R2, R0, R1
     ITT        CC
     SUBCC      ...
     ...
```

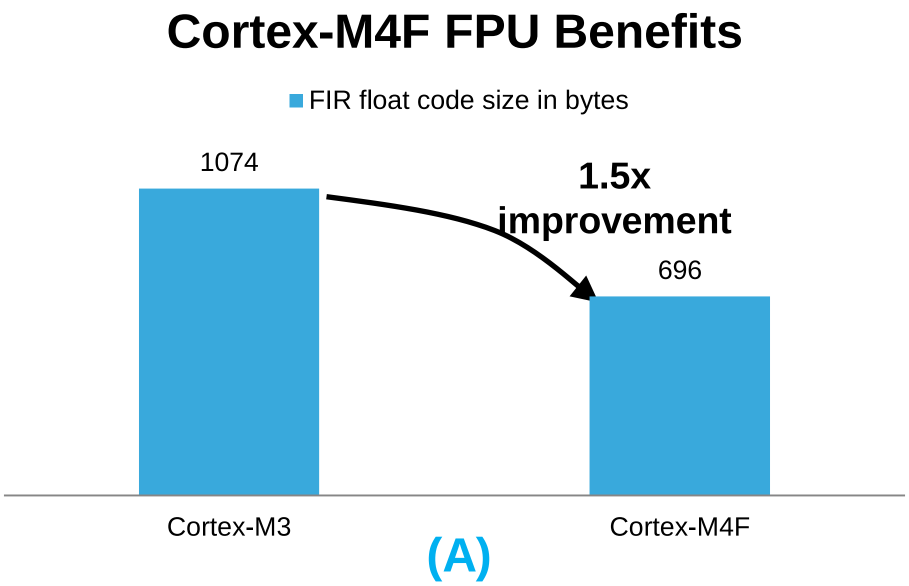
__aeabi_fadd on Cortex-M4F

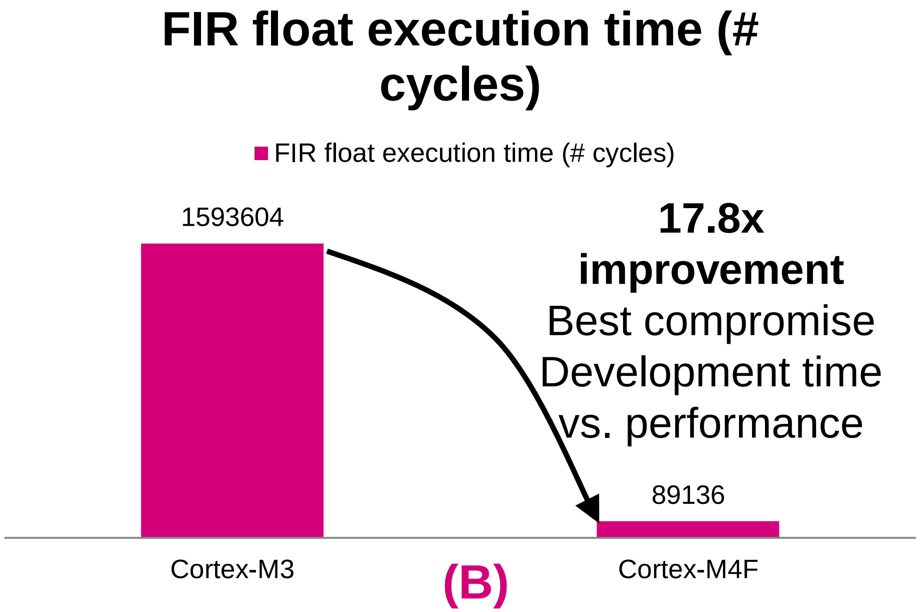
```
# __aeabi_fadd (…)
     VMOV       S0,R0
     VMOV       S1,R1
     VADD.F32   S0,S0,S1
     VMOV       R0,S0
     BX         LR
```

Reduced code size & Enhanced performances

Benefits of a Floating-Point Unit

- Comparison for a 166 coefficient FIR on float 32 with and without FPU (CMSIS library)
  - Improvement in code size (A)
  - Improvement in performance (B)





Cortex-M4 : Floating point unit Features

- **Single precision FPU**
- **Conversion between**
  - Integer numbers
  - Single precision floating point numbers
  - Half precision floating point numbers
- **Handling floating point exceptions** (Untrapped)
- **Dedicated registers**
  - 16 single precision registers (S0-S15) which can be viewed as 16 Doubleword registers for load/store operations (D0-D7)
  - FPSCR for status & configuration

Rounding issues

- **The precision has some limits**
  - Rounding errors can be accumulated along the various operations an may provide unaccurate results (do not do financial operations with floatings…)

- **Few examples**
  - If you are working on two numbers in different base, the hardware automatically « denormalize » on of the two number to make the calculation in the same base
  - If you are substracting two numbers very closed you are loosing the relative precision (also called cancellation error)

- **If you are « reorganizing » the various operations, you may not obtain the same result as because of the rounding errors…**
  - Value1 = ((2.0f - 1.99f) - 0.01f); /* Value1 = -9.313266E-9 */
  - Value2 = (2.0f - (1.99f + 0.01f)); /* Value2 = 0 */

IEEE 754

Floating Point Unit

- Introduction

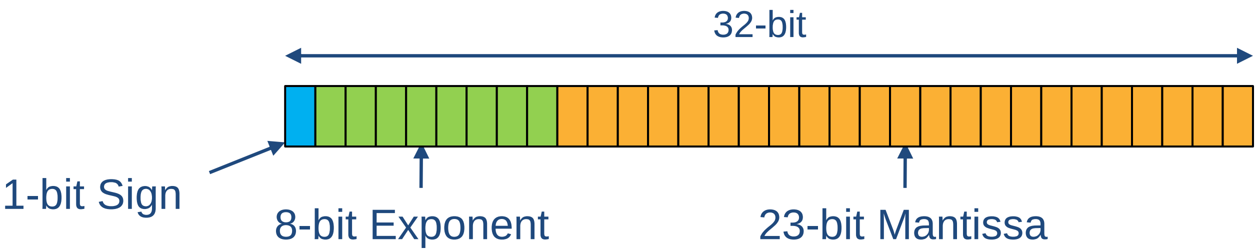
- **IEEE 754**
  - Number format
  - Arithmetic operations
  - Number conversion
  - Special values
  - 4 rounding modes
  - 5 exceptions and their handling

- ARM FPv4-SP Single Precision FPU

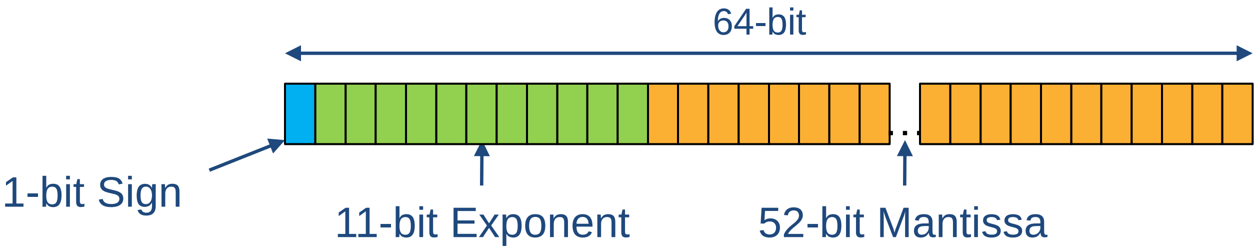
Number format

- **3 fields**
  - Sign
  - Biased exponent (sum of an exponent plus a constant bias)
  - Fractions (or mantissa)

- **Single precision : 32-bit coding**

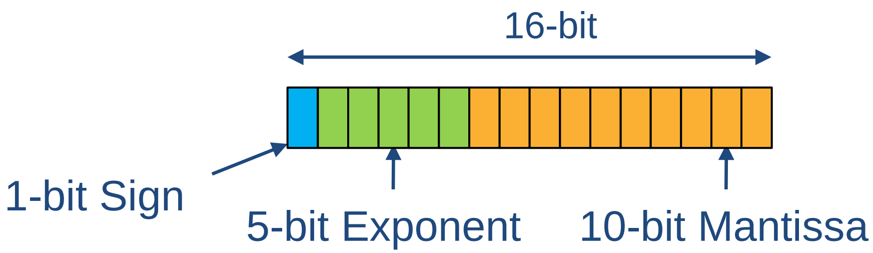


- **Double precision : 64-bit coding**



Number format

- **Half precision : 16-bit coding**



- Can also be used for storage in higher precision FPU
- ARM has an alternative coding for Half precision

Normalized number value

- **Normalized number**
  - Code a number as :

    A sign + Fixed point number between 1.0 and 2.0 multiplied by 2^N

- **Sign field (1-bit)**
  - 0 : positive
  - 1 : negative

- **Single precision exponent field (8-bit)**
  - **Exponent range** : 1 to 254 (0 and 255 reserved)
  - **Bias** : 127
  - **Exponent - bias range** : -126 to +127

- **Single precision fraction (or mantissa) (23-bit)**
  - **Fraction** : value between 0 and 1 : $\sum(N_i.2^{-i})$ with i in 1 to 24 range
  - The 23 N_i values are store in the fraction field

$$(-1)^s \times (1 + \sum(N_i.2^{-i})) \times 2^{exp-bias}$$

Number value

- **Single precision coding of -7**
  - **Sign bit** = 1
  - **7** = 1.75 x 4 = (1 + ½ + ¼) x 4 = (1 + ½ + ¼) x 2^2
    = (1 + 2^{-1} + 2^{-2}) x 2^2
  - **Exponent** = 2 + bias = 2 + 127 = 129 = 0b10000001
  - **Mantissa** = 2^{-1} + 2^{-2} = 0b11000000000000000000000

- **Result**
  - Binary coding : 0b 1 10000001 11000000000000000000000
  - Hexadecimal value : **0xC0E00000**

Special values

- **Denormalized (Exponent field all “0”, Mantisa non 0)**
  - Too small to be normalized (but some can be normalized afterward)
  - $(-1)^s$ x $(\sum(N_i.2^{-i})$ x 2^{-bias}

- **Infinity (Exponent field “all 1”, Mantissa “all 0”)**
  - Signed
  - Created by an overflow or a division by 0
  - Can not be an operand

- **Not a Number : NaN (Exponent filed “all1”, Mantisa non 0)**
  - Quiet NaN : propagated through the next operations (ex: 0/0)
  - Signalled NaN : generate an error

- **Signed zero**
  - Signed because of saturation

Summary of IEEE 754 number coding

Sign	Exponent	Mantissa	Number
0	0	0	+0
1	0	0	-0
0	Max	0	+oo
1	Max	0	-oo
-	Max	!=0 MSB=1	QNaN
-	Max	!=0 MSB=0	SNaN
-	0	!=0	Denormalized number
-	[1, Max-1]	-	Normalized number

Floating-point rounding

- **Round to nearest**
  - Default rounding mode
  - If the two nearest are equally near : select the one with the LSB equal to 0
- **Directed rounding**
  - 3 user-selectable directed rounding modes
  - Round toward +oo, -oo or 0
- **Usage**
  - Program through FPU configuration registers

Floating-point operations

- **Add**
- **Subtract**
- **Multiply**
- **Divide**
- **Remainder**
- **Square root**

Floating-point format conversion

- **Floating-point and Integer**
- **Round-floating point number to integer value**
- **Binary-Decimal**
- **Comparison**

Exceptions

- **Invalid operation**
  - Resulting in a NaN

- **Division by zero**

- **Overflow**
  - The result depend of the rounding mode and can produce a +/-oo or the +/-Max value to be written in the destination register

- **Underflow**
  - Write the denormalize number in the destination register

- **Inexact result**
  - Caused by rounding

Exception handling

- A **TRAP** can be requested by the user for any of the 5 exception with a specific handler

- The **TRAP handler** can return a **value to be used instead** of the exceptional operation result

ARM Cortex-M FPU

Cortex™

Intelligent Processors by ARM®

life.augmented

Floating Point Unit

- Introduction
- IEEE 754
- **ARM FPv4-SP Single Precision FPU**
  - Introduction
  - FPUv4-SP vs IEEE 754-2008
  - FP Status & Control Register
  - FPU instructions
  - Exception management
  - FPU programmers model

Introduction

- **Single precision FPU**
- **Conversion between**
  - Integer numbers
  - Single precision floating point numbers
  - Half precision floating point numbers
- **Handling floating point exceptions** (Untrapped)
- **Dedicated registers**
  - 32 single precision registers (S0-S31) which can be viewed as 16 Doubleword registers for load/store operations (D0-D15)
  - FPSCR for status & configuration

Modifications vs IEEE 754

- **Full Compliance mode**
  - Process all operations according to IEEE 754

- **Alternative Half-Precision format**
  - $(-1)^s \times (1 + \sum(N_i.2^{-i})) \times 2^{16}$ and no de-normalize number support

- **Flush-to-zero mode**
  - De-normalized numbers are treated as zero
  - Associated flags for input and output flush

- **Default NaN mode**
  - Any operation with an NaN as an input or that generates a NaN returns the default NaN

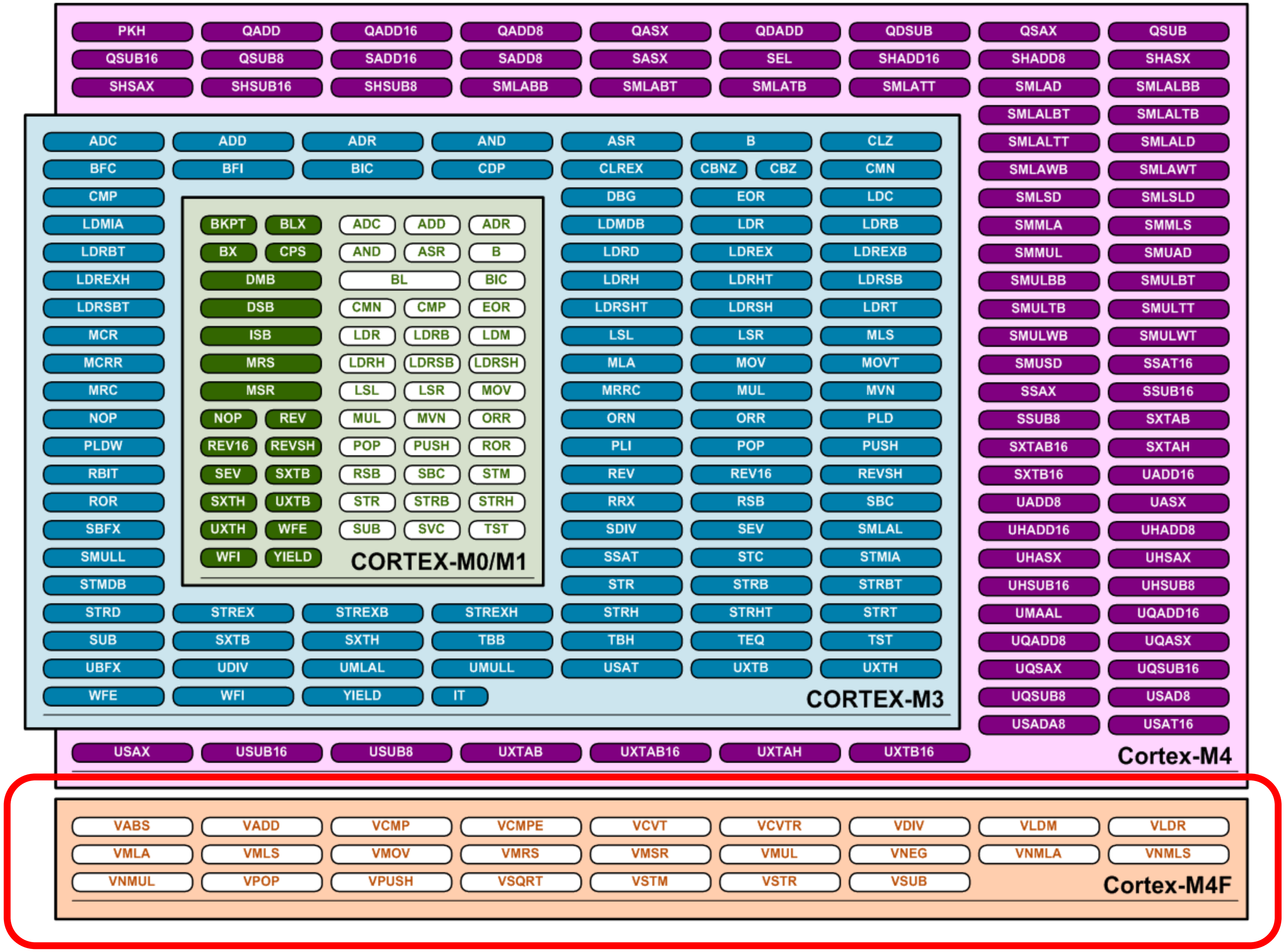
Complete implementation

- **Cortex-M4F does NOT support all operations of IEEE 754-2008**
- **Full implementation is done by software**
- **Unsupported operations**
  - Remainder
  - Round FP number to integer-value FP number
  - Binary to decimal conversions
  - Decimal to binary conversions
  - Direct comparison of SP and DP values

Floating-Point Status & Control Register

- **Condition code bits**
  - negative, zero, carry and overflow (update on compare operations)
- **ARM special operating mode configuration**
  - half-precision, default NaN and flush-to-zero mode
- **The rounding mode configuration**
  - nearest, zero, plus infinity or minus infinity
- **The exception flags are routed to interrupt controller**
  - Masks allow to Enable/Disable exception to generate FPU interruption
  - Inexact result flag is by default masked,…

FPU instructions



PKH QADD QADD16 QADD8 QASX QDADD QDSUB QSAX QSUB
QSUB16 QSUB8 SADD16 SADD8 SASX SEL SHADD16 SHADD8 SHASX
SHSAX SHSUB16 SHSUB8 SMLABB SMLABT SMLATB SMLATT SMLAD SMLALBB
SMLALBT SMLALTB
SMLALTT SMLALD
SMLAWB SMLAWT
SMLSD SMLSLD
SMMLA SMMLS
SMMUL SMUAD
SMULBB SMULBT
SMULTB SMULTT
SMULWB SMULWT
SMUSD SSAT16
SSAX SSUB16
SSUB8 SXTAB
SXTAB16 SXTAH
SXTB16 UADD16
UADD8 UASX
UHADD16 UHADD8
UHASX UHSAX
UHSUB16 UHSUB8
UMAAL UQADD16
UQADD8 UQASX
UQSAX UQSUB16
UQSUB8 USAD8
USADA8 USAT16
USAX USUB16 USUB8 UXTAB UXTAB16 UXTAH UXTB16

Cortex-M4

ADC ADD ADR AND ASR B CLZ
BFC BFI BIC CDP CLREX CBNZ CBZ CMN
CMP DBG EOR LDC
LDMIA LDMDB LDR LDRB
LDRBT LDRD LDREX LDREXB
LDREXH LDRH LDRHT LDRSB
LDRSBT LDRSHT LDRSH LDRT
MCR LSL LSR MLS
MCRR MLA MOV MOVT
MRC MRRC MUL MVN
NOP ORN ORR PLD
PLDW PLI POP PUSH
RBIT REV REV16 REVSH
ROR RRX RSB SBC
SBFX SDIV SEV SMLAL
SMULL SSAT STC STMIA
STMDB STR STRB STRBT
STRD STREX STREXB STREXH STRH STRHT STRT
SUB SXTB SXTH TBB TBH TEQ TST
UBFX UDIV UMLAL UMULL USAT UXTB UXTH
WFE WFI YIELD IT

CORTEX-M3

BKPT BLX ADC ADD ADR
BX CPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
ISB LDR LDRB LDM
MRS LDRH LDRSB LDRSH
MSR LSL LSR MOV
NOP REV MUL MVN ORR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD

CORTEX-M0/M1

VABS VADD VCMP VCMPE VCVT VCVTR VDIV VLDM VLDR
VMLA VMLS VMOV VMRS VMSR VMUL VNEG VNMLA VNMLS
VNMUL VPOP VPUSH VSQRT VSTM VSTR VSUB

Cortex-M4F

FPU arithmetic instructions

Operation	Description	Assembler	Cycle
Absolute value	of float	VABS.F32	1
Negate	float and multiply float	VNEG.F32 VNMUL.F32	1 1
Addition	floating point	VADD.F32	1
Subtract	float	VSUB.F32	1
Multiply	float then accumulate float then subtract float then accumulate then negate float the subtract the negate float	VMUL.F32 VMLA.F32 VMLS.F32 VNMLA.F32 VNMLS.F32	1 3 3 3 3
Multiply (fused)	then accumulate float then subtract float then accumulate then negate float then subtract then negate float	VFMA.F32 VFMS.F32 VFNMA.F32 VFNMS.F32	3 3 3 3
Divide	float	VDIV.F32	14
Square-root	of float	VSQRT.F32	14

FPU Load/Store/Compare/Convert

Operation	Description	Assembler	Cycle
Load	multiple doubles (N doubles)	VLDM.64	1+2*N
	multiple floats (N floats)	VLDM.32	1+N
	single double	VLDR.64	3
	single float	VLDR.32	2
Store	multiple double registers (N doubles)	VSTM.64	1+2*N
	multiple float registers (N doubles)	VSTM.32	1+N
	single double register	VSTR.64	3
	single float register	VSTR.32	2
Move	top/bottom half of double to/from core register	VMOV	1
	immediate/float to float-register	VMOV	1
	two floats/one double to/from core registers	VMOV	2
	one float to/from core register	VMOV	1
	floating-point control/status to core register	VMRS	1
	core register to floating-point control/status	VMSR	1
Pop	double registers from stack	VPOP.64	1+2*N
	float registers from stack	VPOP.32	1+N
Push	double registers to stack	VPUSH.64	1+2*N
	float registers to stack	VPUSH.32	1+N
Compare	float with register or zero	VCMP.F32	1
	float with register or zero	VCMPE.F32	1
Convert	between integer, fixed-point, half precision and float	VCVT.F32	1

Exception management

- **No TRAP function : exception through interrupt controller**

- **FP register saving modes** (when FPU is enabled)
    - No FP registers saving
    - Lazy saving/restoring (only space allocation in the stack)
    - Automatic FP registers saving/restoring

- **Stack frame**
    - 17 entries in the stack (FPSCR + S0-S15)

IEEE754 compliancy

The Cortex-M4 Floating Point Unit is IEEE754 compliant :

- The rounding more is selected in the FPSCR register (nearest even value by default)

Sign	Exponent	Mantissa	Compliant options FZ=0 and AHP=0 and DN=0	Non compliant option FZ=1 or AHP=1 or DN=1
-	0	!=0	De-normalized number	Flush to zero
0	Max	0	+infinity	Alternate Half Precision
1	Max	0	-infinity	Alternate Half Precision
-	Max	!=0 MSB=1	QNaN (Quiet Not a Number)	Default NaN Alternate Half Precision
-	Max	!=0 MSB=0	SNaN (Signaling Not a Number)	Default NaN Alternate Half Precision

Some non compliant options are available in the FPSCR Register:

- Flush to zero (FZ bit) :
  - de-normalized numbers are flushed to zero
- Alternate Half Precision formation (AHP bit):
  - special numbers (exp = all “1”) = normalized numbers
- Default NaN (DN bit):
  - Different way to handle the Not A Number values

STM32 - Floating point exceptions

The FPU supports the 5 IEEE754 exceptions +1

Invalid operation (IEEE754)	Underflow (IEEE754)
Division by zero (IEEE754)	Inexact (IEEE754)
Overflow (IEEE754)	Input denormal (Fluh to zero mode only)

- Comments
  - These flags are in the FPSCR register
  - When flush to zero mode is used:
    - the FPU add a specific exception : input denormal
    - the FPU handles the underflow and Inexact exception in a non-IEEE754 way
  - The exception are not trapped
    - This is compliant with IEEE754
    - The value returned by the instruction generating an exception is a default result.
  - Examples
    - 1234 / 0 => division by zero flag is set / the returned value is +infinity
    - Sqrt(-1) => Invalid Operation flag is set / the returned value is QNaN

Note: For details on each exception as well as the default returned value when such exceptions occurs, please refer to ARM-7M architecture reference manual

FPU programmers model

Address	Name	Type	Description
0xE000EF34	FPCCR	RW	FP Context Control Register
0xE000EF38	FPCAR	RW	FP Context Address Register
0xE000EF3C	FPDSCR	RW	FP Default Status Control Register
0xE000EF40	MVFR0	RO	Media and VFP Feature Register 0
0xE000EF44	MVFR1	RO	Media and VFP Feature Register 1

- **Floating-Point Context Control Register**
  - Indicates the context when the FP stack frame has been allocated
  - Context preservation setting
- **Floating-Point Context Address Register**
  - Points to the stack location reserved for S0
  - Valid only for lazy context saving mode
- **Floating-Point Default Status Control Register**
  - Details default values for Alternative half-precision mode, Default NaN mode, Flush to zero mode and Rounding mode
- **Media & FP Feature Register 0 & 1**
  - Details supported mode, instructions precision and and additional hardware support

About the Stack Frame

There is a difference between the stack frame with or without FPU

0x1C	xPSR
0x18	ReturnAddress
0x14	LR (R14)
0x10	R12
0x0C	R3
0x08	R2
0x04	R1
0x00	R0

Basic Frame

Frame without FPU

0x64	Reserved
0x60	FPSCR
0x5C	S15
…	…
0x20	S0
0x1C	xPSR
0x18	ReturnAddress
0x14	LR (R14)
0x10	R12
0x0C	R3
0x08	R2
0x04	R1
0x00	R0

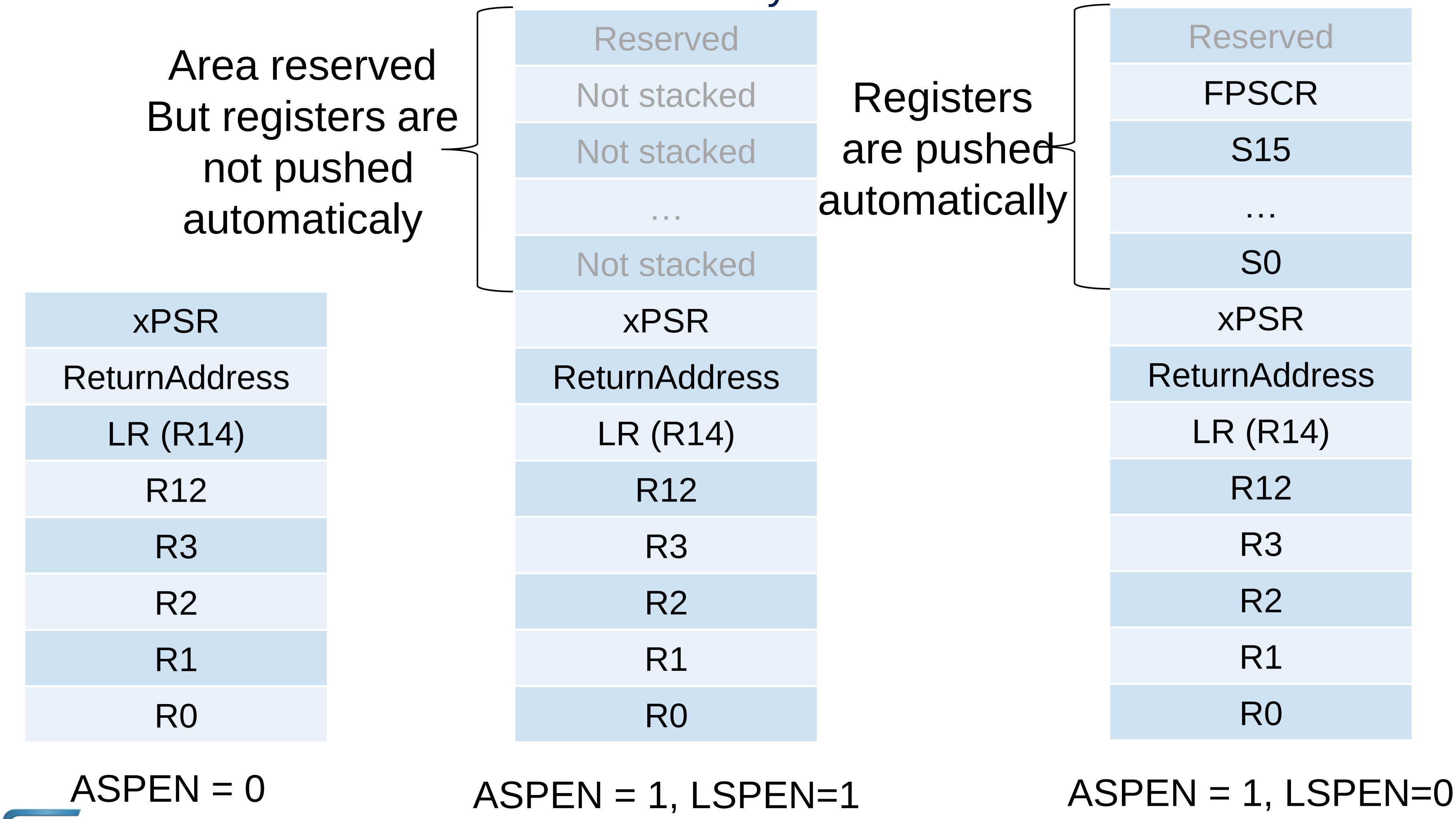
Extended Frame

Frame with FPU

Note : the FPU registers S16 to S31 are not in the frame

About the Stack Frame

Depending on the Floating-Point Context Control Register configuration, the core handle the stack in different ways



Area reserved
But registers are
not pushed
automaticaly

Registers
are pushed
automatically

ASPEN = 0	ASPEN = 1, LSPEN=1	ASPEN = 1, LSPEN=0
	Reserved	Reserved
	Not stacked	FPSCR
	Not stacked	S15
	…	…
	Not stacked	S0
xPSR	xPSR	xPSR
ReturnAddress	ReturnAddress	ReturnAddress
LR (R14)	LR (R14)	LR (R14)
R12	R12	R12
R3	R3	R3
R2	R2	R2
R1	R1	R1
R0	R0	R0

Lazy context save (default after reset)

Reserved
Not stacked
Not stacked
…
Not stacked
xPSR
ReturnAddress
LR (R14)
R12
R3
R2
R1
R0

ASPEN = 1
LSPEN=1

In Lazy mode, the FP context is not saved

- This reduces the exception latency.
- This keep it simple for the user to push the value if needed

If a floating point instruction is needed when lazy context save is active, the processor first :

- Retrieve the address of the reserved area from FPCAR register
- Save the FP state, S0-S15 and the FPSCR,
- Sets the FPCCR.LSPACT bit to 0, to indicate that lazy state preservation is no longer active,
- It can then processes the FPU instruction.

Enabling FPU exception interruption

- Six exception flags (IDC, IXC, UFC, OFC, DZC, IOC) are ORed and connected to the interrupt controller.
- There is an individual mask to enable/disable FPU interrupt for each exception.
- FPU exception mask is done at product level : System configuration controller configuration register 1 (SYSCFG_SFGR1).
- FPU interruption enable/disable is done at interrupt controller level.

Clearing FPU exception interruption flags

- Clearing the FPU exception interruption source flags depends on FPU context save/restore configuration

FP registers save/restore mode	How to clear	Comment

None	Interrupt source must be cleared in FP Status and Control Register (*FPSCR*).	Using CMSIS functions:
		__get_FPSCR()
		__set_FPSCR()

Lazy	Interrupt source must be cleared in the stack. FPSCR register address : *FPU->FPCAR + 0x40*	A dummy read access should be made to FP register to force context saving.
Automatic	Interrupt source must be cleared in the stack.	Check LR value to determine which stack was used to preserve context.

Note : Please refer to Cortex-M4F programming manual for more details

Exception entry & LR values

Depending on the CPU mode and configuration, context format & destination stack varies.

- LR register value gives details on which mode/configuration was active when entering the exception.

LR Values	Return to (Mode)	Return Stack	Frame Type
0xFFFF_FFF1	Handler Mode	Main	Basic
0xFFFF_FFE1	Handler Mode	Main	Extended
0xFFFF_FFF9	Thread mode	Main	Basic
0xFFFF_FFE9	Thread mode	Main	Extended
0xFFFF_FFFD	Thread mode	Process	Basic
0xFFFF_FFED	Thread mode	Process	Extended

FPU exception interruption benefits

- Boost the priority of FPU exception handler (via NVIC software interruption priorities configuration)

- Optimize over all performance
  - Example handling Divide-by-Zero Exception:

With polling	With Divide-by-Zero Interruption

float x = 2.5f;
for(index = 0; index < 0xFFFF;i++)
{
   x = 1.0f/(x*x);
   if(__get_FPSCR() & 0x00000002)
      {
         DivZeroExc_Handler();
      }
}
	float x = 2.5f;
	SYSCFG_ITConfig(SYSCFG_IT_DZC, ENABLE);
	for(index = 0; index < 0xFFFF; index ++)
	{
	   x = 1.0f/(x*x);
	}
	SYSCFG_ITConfig(SYSCFG_IT_DZC, DISABLE);

	void FPU_IRQHandler(void)
	{
	   DivZeroExc_Handler();
	}

What can reduce FPU performances?

- Accidentally used double precision data/functions
  - The compiler will use double precision software library instead of using single precision Hardware FPU
  - Explicitly cast your constant data to float type.
- Compiler/library settings (e.g. hard VFP vs soft VFP)
- Bad instructions scheduling (when writing in assembly)
  - Pipelining instruction execution between Cortex-M4 core and FPU co-processor improves over all performance.
  - Basic example : float division(VDIV 14 cycles) can hide load (LDR, LDM, …) overhead, …

Floating point DSP examples

- Comparison of DSP (ARM CMSIS DSP library) algorithm execution time on Cortex-M4F:
  - with and without FPU
  - version using FPU insructions Vs version using DSP instructions

FFT benchmarking results
Setup

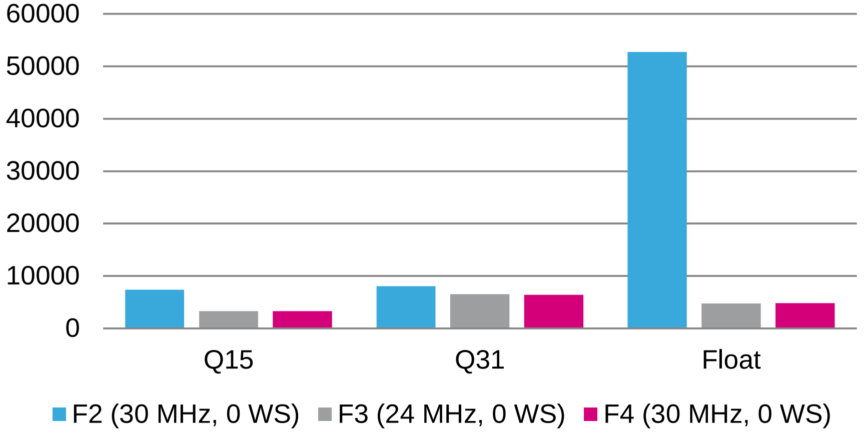
Input signal characteristics	
Input signal	a sine wave with a frequency F 50Hz <F<4KHz
Sampling frequency	8KHz

Measures done with MDK-ARM (4.23.00.0) toolchain

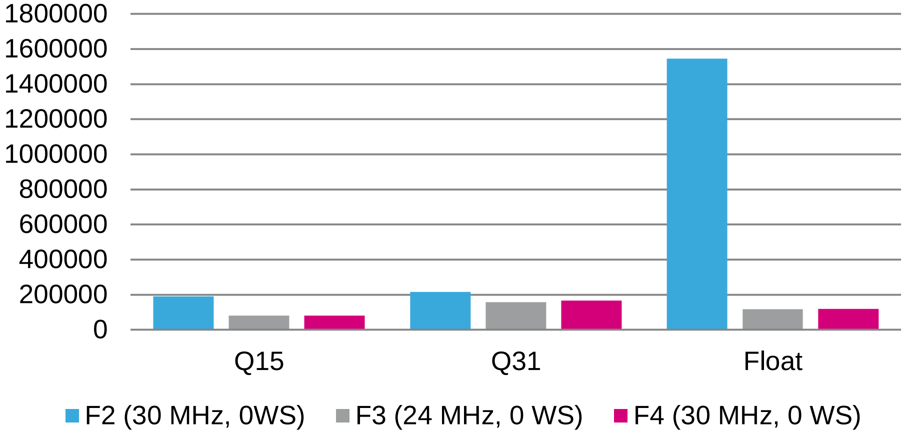
Level 3(-O3) for time optimization without MicroLib

FFT benchmark results Cortex-M3 vs Cortex-M4F (2/2)

FFT 64-points average processing time at 0 WS (# cycles)



FFT 1024-points average processing time at 0 WS (# cycles)



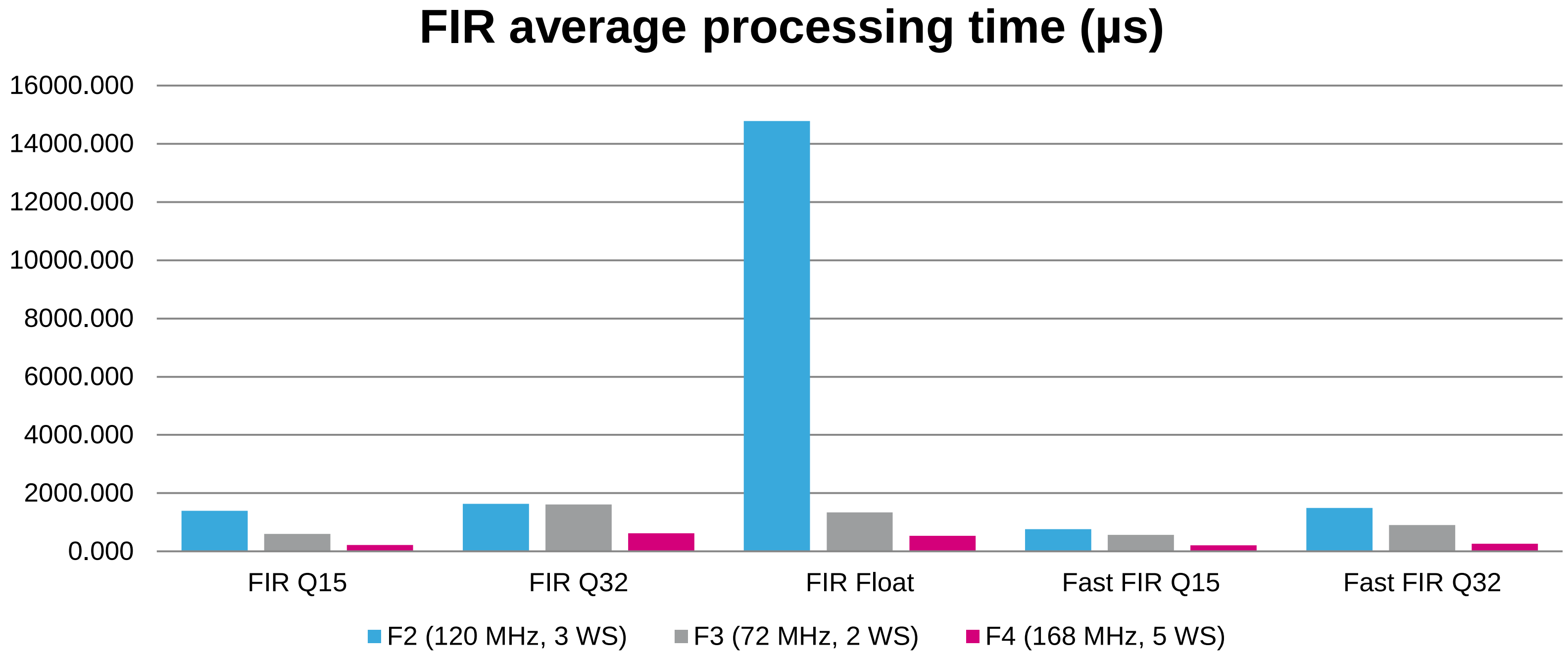
		F2(30MHz,0WS) (#cycles)	F3(24 MHz,0WS) (#cycles)	F4(30MHz,0WS) (#cycles)	Gain (F2 vs F3)
FFT (64-points)	Q15	7374	3300	3307	x2.23
	Q31	8022	6522	6410	x1.23
	Float	52763	4725	4793	x11.17
FFT (1024-points)	Q15	190028	80608	80252	x2.36
	Q31	215505	158022	166406	x1.36
	Float	1544676	116576	118633	x13.25

FIR benchmarking results
Setup

Filter & input signal characteristics	
Filter type	Stop Band
Filter order	165
Filter coefficients	166
Cut-off frequency	F_{STOP1}=1.9KHz, F_{STOP2}=2.1KHz
Sampling frequency	48KHz
Number of samples	128
Input signal	a sine wave with a frequency F 50Hz <F<4KHz

Measures done with MDK-ARM (4.23.00.0) toolchain
Level 3(-O3) for time optimization without MicroLib

FIR benchmarking results F2/F3/F4



		F2(120MHz,3 WS)	Processing time per Tap	F3(72 MHz, 2 WS)	Processing time per Tap	F4(168MHz,5 WS)	Processing time per Tap
FIR	Q15	1396.992 µs	66ns	605.819 µs	28.5 ns	218.280 µs	10 ns
	Q31	1636.658 µs	77ns	1613.722 µs	76 ns	618.273 µs	29 ns
	Float	14782.510 µs	696ns	1338.528 µs	63 ns	531.160 µs	25 ns
Fast FIR	Q15	760.675 µs	36ns	566.333 µs	26 ns	209.761 µs	10 ns
	Q31	1488.292 µs	70ns	907.736 µs	42 ns	267.464 µs	13 ns

Summary

- FPU is a key benefit for many application tasks that require precision (to name just a few) :
  - loop control,
  - audio processing,
  - sensor signal conditioning,
  - motor control,
  - digital filtering, …
- FPU gives developers unparalleled flexibility:
  - Applications requiring to apply mathematical models on data benefits from FPU where math formulas are translated directly to C code with no pains
  - Code is more easy to maintain compared to fixed point.
- Floating point number (Half precision) has larger dynamic range than fixed-point

STM32 Releasing your **creativity**

Thank you !