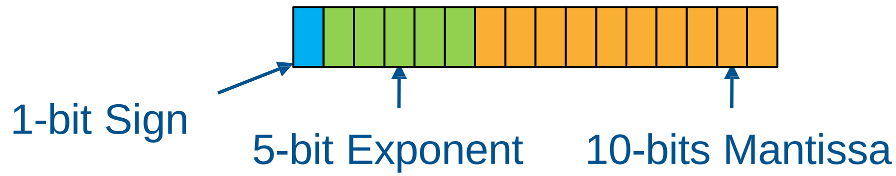
STM32F4 Core, DSP, FPU & Library

Content

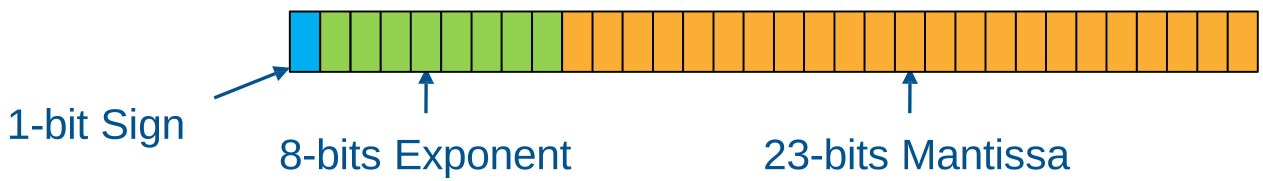
- A practical introduction to Fixed/Floating points
- A practical introduction to the floating point unit
- Tips & comments on floating points usage

Half / single / double precision

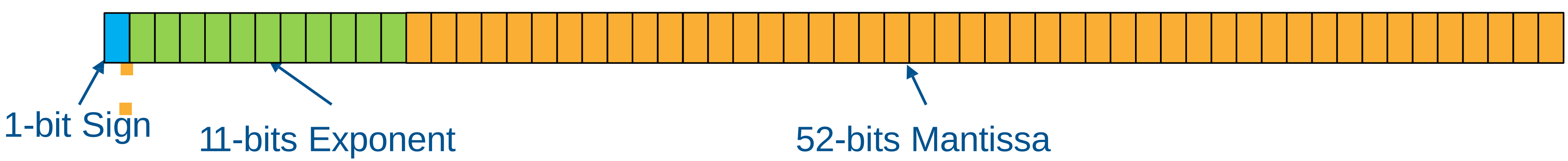
- **Half precision : 16-bits coding** ***(called binary16 in IEEE754-2008)***



- **Single precision : 32-bits coding** ***(called binary32 in IEEE754-2008)***



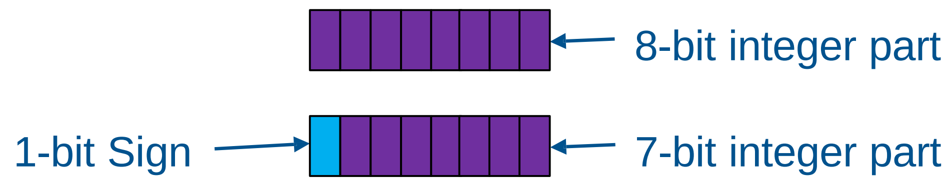
- **Double precision : 64-bits coding** ***(called binary64 in IEEE754-2008)***



Let's compare 8 bits formats

Integers format

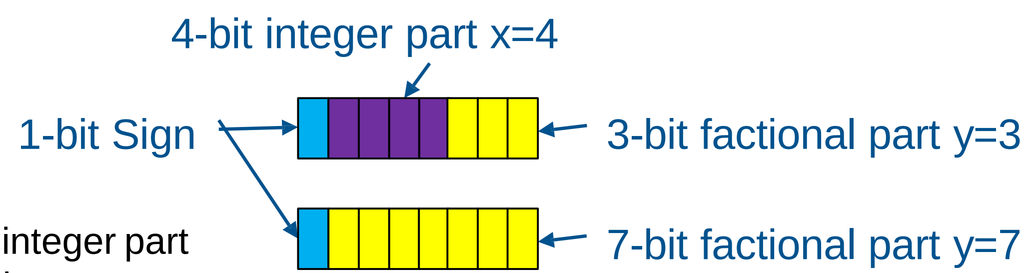
- unsigned integer
- Signed integer



Unsigned Represented value = 8bits integer part
Signed Represented value = $(-1)^{sign} \times$ 7bitsinteger part

Fixed point format Qx.y

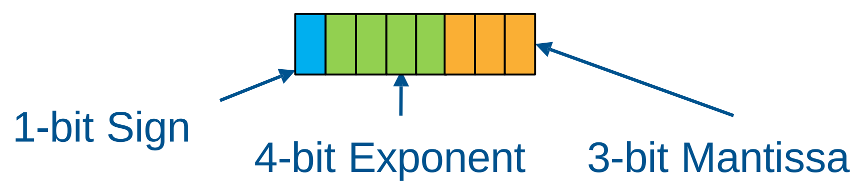
- Q4.3 format
- Q0.7 format



Q4.3 Represented value = $(-1)^{sign} \times 2^{-3} \times$ 7bits integer part
Q0.7 Represented value = $(-1)^{sign} \times 2^{-7} \times$ 7bits integer part

Floating point format

- IEEE754 Like
- Non IEEE754 Like

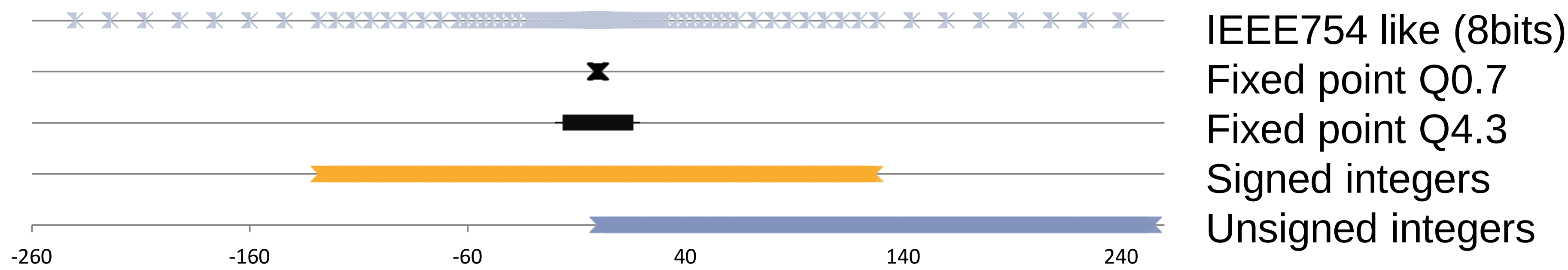


IEEE754 Like represented Normalized value : $(-1)^{sign}$ x $2^{(exponent - bias)}$ x **mantissa**
IEEE754 Like represented DeNormalized value : $(-1)^{sign}$ x $2^{(1- bias)}$ x **mantissa** (also called subnormal)

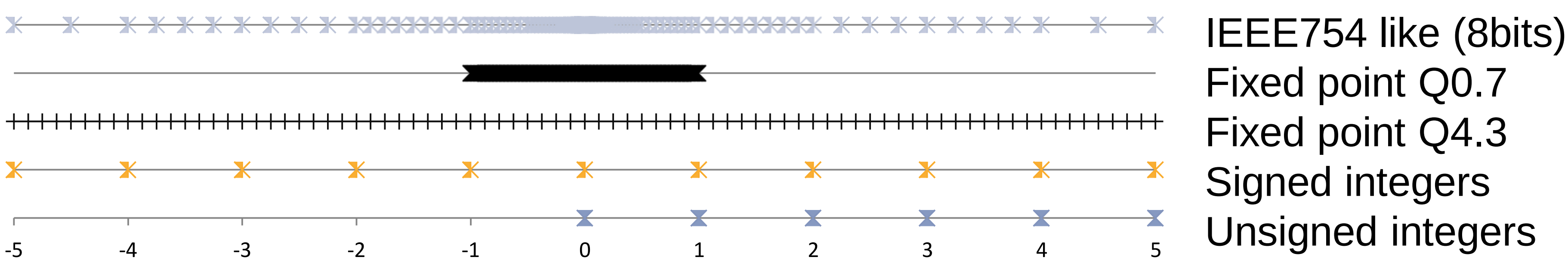
Note : this 8bits floating point format is not standard, it is used for illustration purpose

8bits formats comparison

Looking a the range -260 to +260



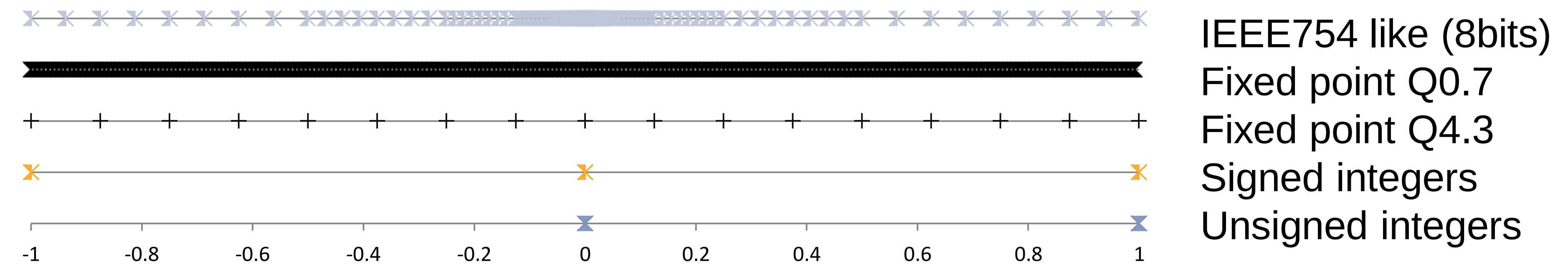
Looking at the range -5 to 5



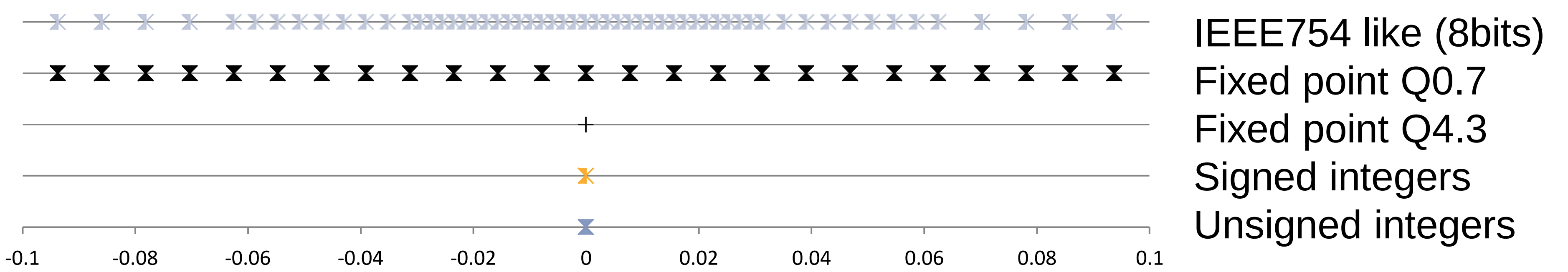
Note : All these formats have 256 dicrete values, only the repartition is different

8bits formats comparison (continued)

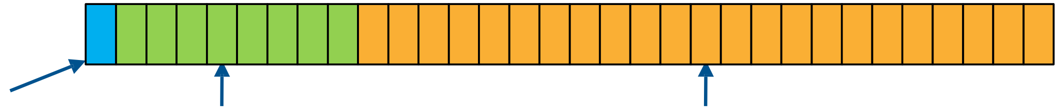
Looking a the range -1 to +1



Looking at the range -0.1 to 0.1



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

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

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

The IEEE754-2008 standard defines theses formats:

Format	Sign	Exponent	Mantissa
Binary16 / Half precision	1bit	5bits	10bits *(+1 implied bit for normalized numbers)*
Binary32 / Single precision	1bit	8bits	23bits *(+1 implied bit for normalized numbers)*
Binary64 / Double Precision	1bit	11bits	52bits *(+1 implied bit for normalized numbers)*

Normalized / Denormalized numbers

Sign	Exponent	Mantissa	IEEE754-2008
-	0	!=0	De-normalized number *(mantissa without implied MSB)*
-	[1, Max-1]	-	Normalized number *(mantissa with one implied MSB)*

Each of the format contains special numbers

Sign	Exponent	Mantissa	IEEE754-2008
0	0	0	+0
1	0	0	-0
0	Max	0	+infinity
1	Max	0	-infinity
-	Max	!=0 MSB=1	QNaN (Quiet Not a Number)
-	Max	!=0 MSB=0	SNaN (Signaling Not a Number)

Floating points : Rounding issues

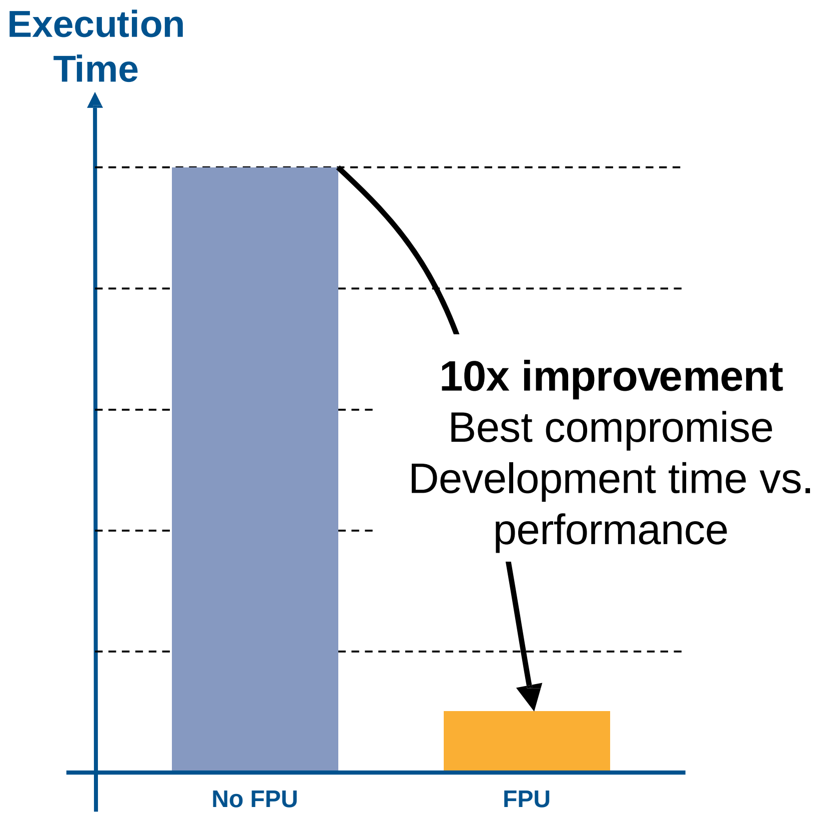
- **The precision has some limits**
  - Rounding errors can be accumulated along the various operations an may provide unaccurate results (do not do financial operations with floatings…)

- **Few examples**
  - If you are working on two numbers in different base, the hardware automatically « denormalize » one of the two numbers 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…**

Benefits of a Floating-Point Unit

Time execution comparison for a 29 coefficient FIR on float 32 with and without FPU (CMSIS library)



Code comparison with & without FPU

```
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)

Cortex-M4 : Floating point unit Features

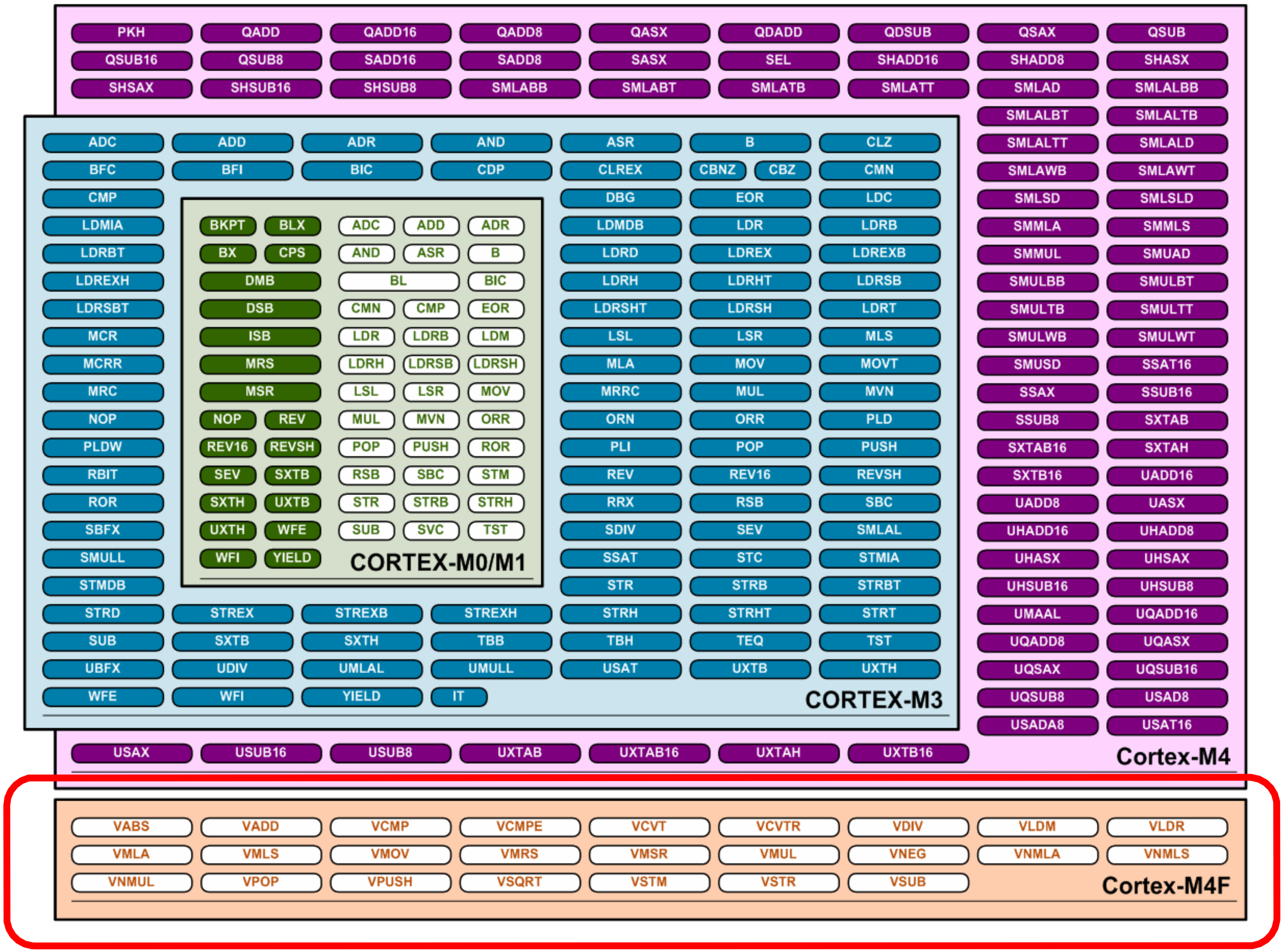
- **Single precision FPU**

- **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

FPU instructions



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

Important informations

- **The Floating point Unit IS compliant with IEEE754-2008**
- **The Floating point unit does NOT support all operations of IEEE 754-2008**
- **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
- **Full implementation is done by software**

IEEE754 compliancy

The Cortex-M4 Floating Point Unit is IEEE754 compliant :

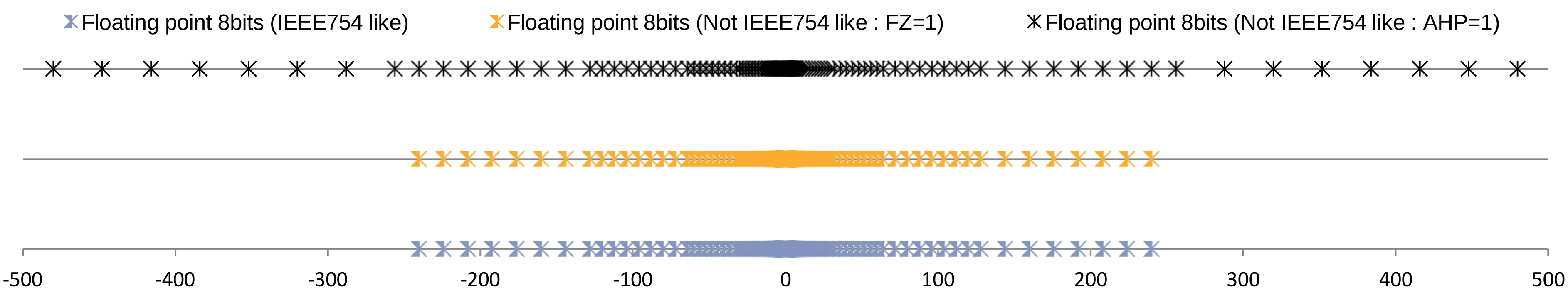
- The rounding more is selected in the FPSCR register (nearest even value by default)

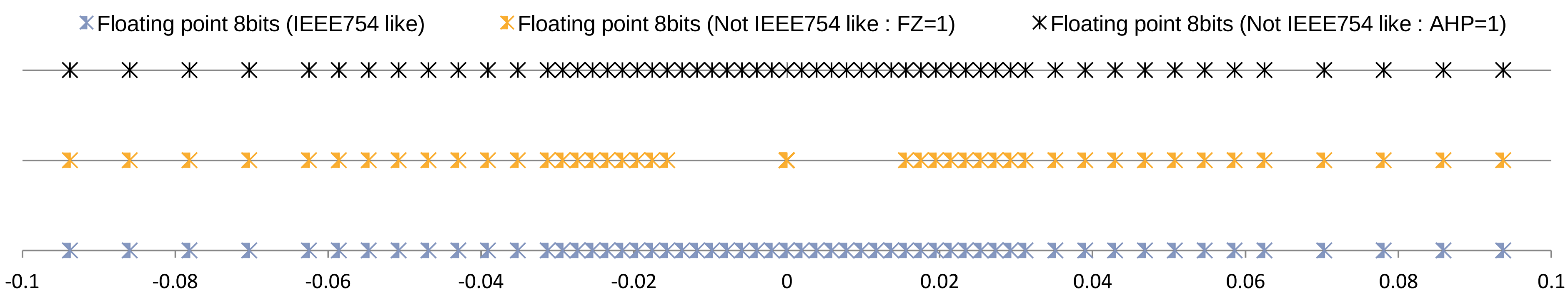
Sign	Exponent	Mantissa	Compliant options FZ=0 and AHP=0 and DN=0	Non compliant option FZ=1 or AHP=1 or DN=1
-	0	!=0	De-normalized number	Flush to zero
0	Max	0	+infinity	Alternate Half Precision
1	Max	0	-infinity	Alternate Half Precision
-	Max	!=0 MSB=1	QNaN (Quiet Not a Number)	Default NaN Alternate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

STM32F4 - Non IEEE754 compliant format





These are simulation using an 8bits format representation :

- Flush to zero (FZ bit) applies to 16bits (Half precision) & 32bits (single precision) formats
- Alternate Half Precision format (AHP bit) applies to 16bits (Half precision) format only

STM32F4 - Floating point exceptions

The FPU supports the 5 IEEE754 exceptions and adds a specific exception

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

- 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
- **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

About the Stack Frame

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

xPSR
ReturnAddress
LR (R14)
R12
R3
R2
R1
R0

ASPEN = 0

Area reserved
But registers are
not pushed
automaticaly

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

ASPEN = 1, LSPEN=1

Registers
are pushed
automatically

Reserved
FPSCR
S15
…
S0
xPSR
ReturnAddress
LR (R14)
R12
R3
R2
R1
R0

ASPEN = 1, LSPEN=0

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.
- While keeping it simple for the user to push the value if needed

If a floating point instruction is needed, the ISR need :

- To retrieve the address of the reserved area from the FPCAR register
- To 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.

Tips and comments on FP usage

What type to use ???

What is the difference between

- double a = (double) 1.1234
- double b = 1.1234
- double c = (float) 1.1234
- double d = 1.1234f

- float a = (double) 1.1234
- float b = 1.1234
- float c = (float) 1.1234
- float d = 1.1234f

- float e = a + b
- float f = a + b + (float) 1.1234
- float f = a + b + 1.1234
- float f = a + b + 1.1234f

To avoid :
-Compiler dependant behavior
- Implicit conversions
- the usage of an unexpected type
-the use of double precision software library when intending to use Hardware FPU

It is recommended to always explicitly specify the type using
float a = (float) 1.234
float a = 1.234f
double a = (double) 1.234

A practical example for rounding issue

```
sp_a = 0.9999996f;
sp_a += 0.0000001f;
sp_a += 0.0000001f;
sp_a += 0.0000001f;
sp_a += 0.0000001f;
sp_a += 0.0000001f;
sp_a += 0.0000001f;
sp_a += 0.0000001f;

sp_b = 0.9999996f;
sp_b += 0.0000007f;

if (sp_b == sp_a)
{ sp_a =1;}
else { sp_a =0;}
```

```
sp_a = 0.9999996f;
  sp_a += 0.0000001f;
  sp_a += 0.0000001f;
  sp_a += 0.0000001f;
  sp_a += 0.0000001f;

  sp_b = 0.9999996f;
  sp_b += 0.0000004f;

   if (sp_b == sp_a)
 { sp_a =1;}
  else  { sp_a =0;}
```

Floats cannot be compared directly

A better approach (but not perfect)
`if(sp_a-sp_b<delta) …`

But is there a suitable delta? What would be a suitable delta… depends on the application

…

Pay attention to rounding issues

- **Each operation adds some rounding errors**
- **A repeated addition = addition of rounding errors**
- **Never use this in looping condition**
  - The computed result may never exactly match the theorical value due to rounding errors

- **Another example : there is a difference between**

```
float a = 0.9999996f;
for (i=0; i<100, i++)  a+=0.0000001f ;
```
(rounded 100x)

And

```
float a = 0.9999996f;
a+= 100 * 0.0000001f ;
```
(rounded only once time)

Pay attention to substraction/addition

Substraction (and addition of negative numbers) can result in important loss of precision

For example :

- It may append that `a+b=a` even if `b≠0`
- It may append that `a-b=c` with c very different from the exact value
- It may append that `a-b=0` with even if the exact value is not zero at all

Cortex M4 – Focus on DSP Instructions

STMicroelectronics

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

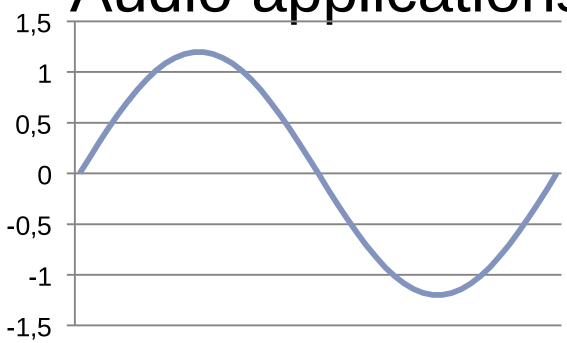
Cortex-M4 extended single cycle MAC

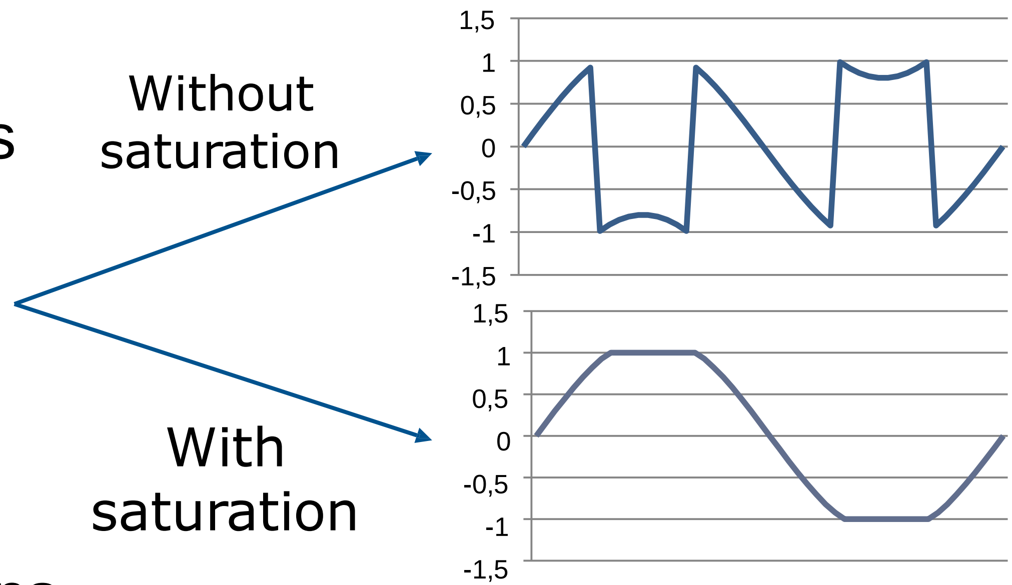
OPERATION	INSTRUCTIONS	CM3	CM4
16 x 16 = 32	SMULBB, SMULBT, SMULTB, SMULTT	n/a	1
16 x 16 + 32 = 32	SMLABB, SMLABT, SMLATB, SMLATT	n/a	1
16 x 16 + 64 = 64	SMLALBB, SMLALBT, SMLALTB, SMLALTT	n/a	1
16 x 32 = 32	SMULWB, SMULWT	n/a	1
(16 x 32) + 32 = 32	SMLAWB, SMLAWT	n/a	1
(16 x 16) ± (16 x 16) = 32	SMUAD, 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 <u>single cycle</u> 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

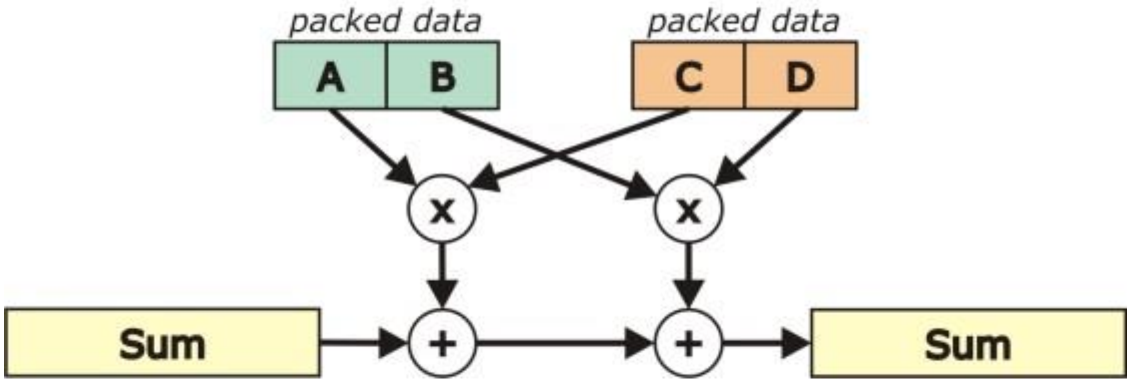




  - Control applications
    - The PID controllers' integral term is continuously accumulated over time. The saturation automatically limits its value and saves several CPU cycles per regulators

Single-cycle SIMD instructions

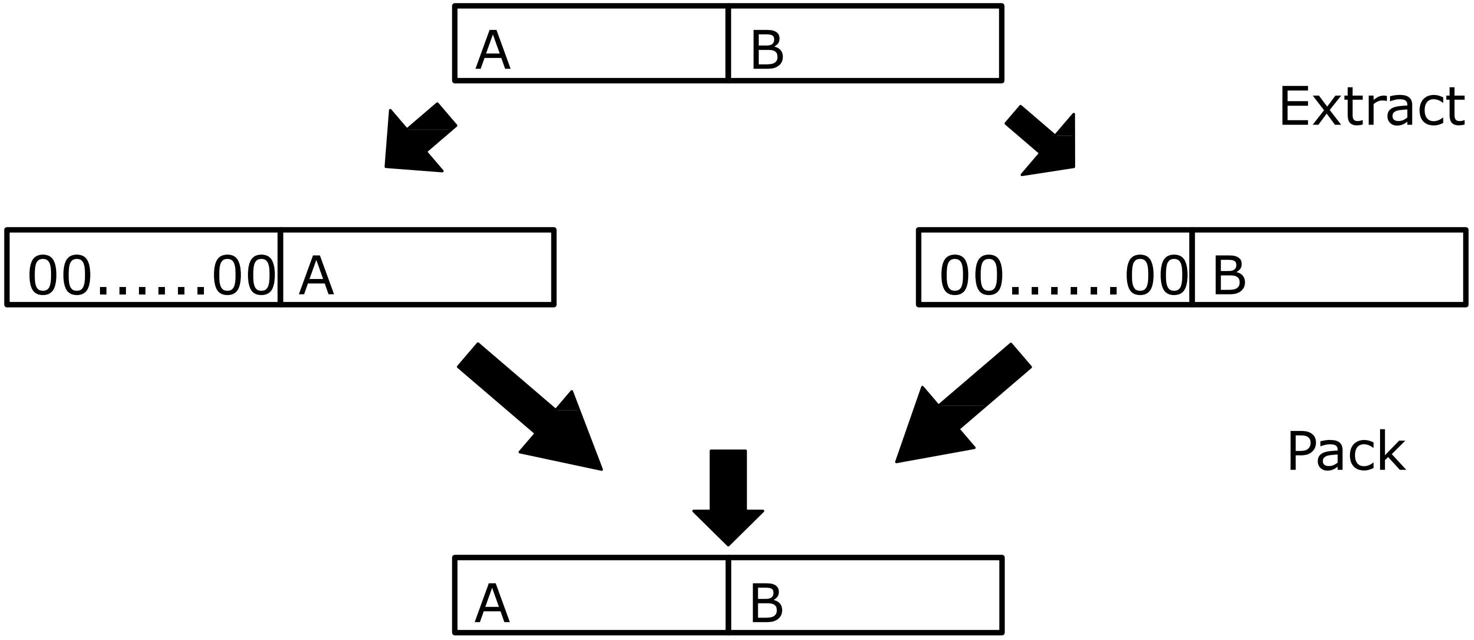
- Stands for **Single Instruction Multiple Data**
- It operates with **packed data**
- Allows to do simultaneously several operations with 8-bit or 16-bit data format
  - i.e.: dual 16-bit MAC (Result = 16x16 + 16x16 + 32)



- 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)

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



IIR – single cycle MAC benefit

	Cortex-M3 cy-count	Cortex-M4 cy-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

- 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

FIR Filter Standard C Code

```
void fir(q31_t *in, q31_t *out, q31_t *coeffs,     *stateIndexPtr,
int
                    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

Zero overhead loop

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

Multiply and accumulate previous

Coeff fetch with linear addressing

State fetch with circular addressing

Cortex-M4 - 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