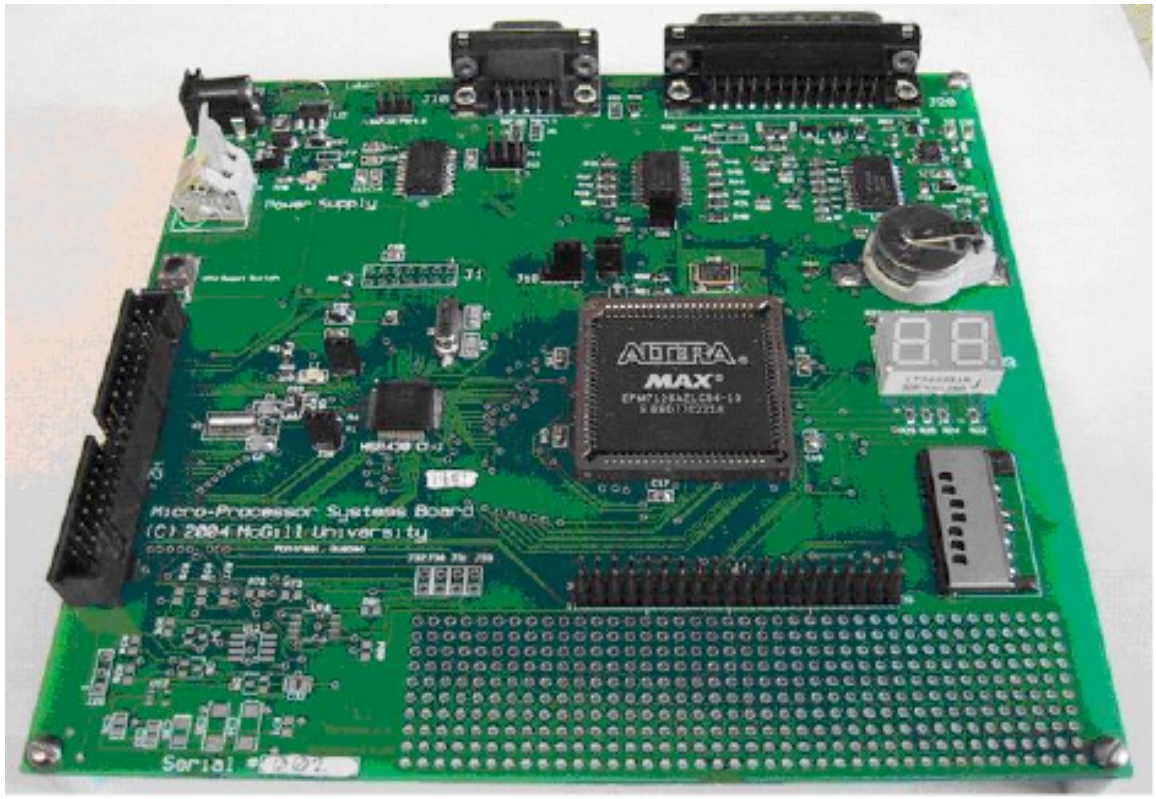
ECSE 426
Tools, Architecture and HW

Zeljko Zilic

zeljko.zilic@mcgill.ca



Acknowledgments: to STMicroelectronics for material on processors and the board

Outline

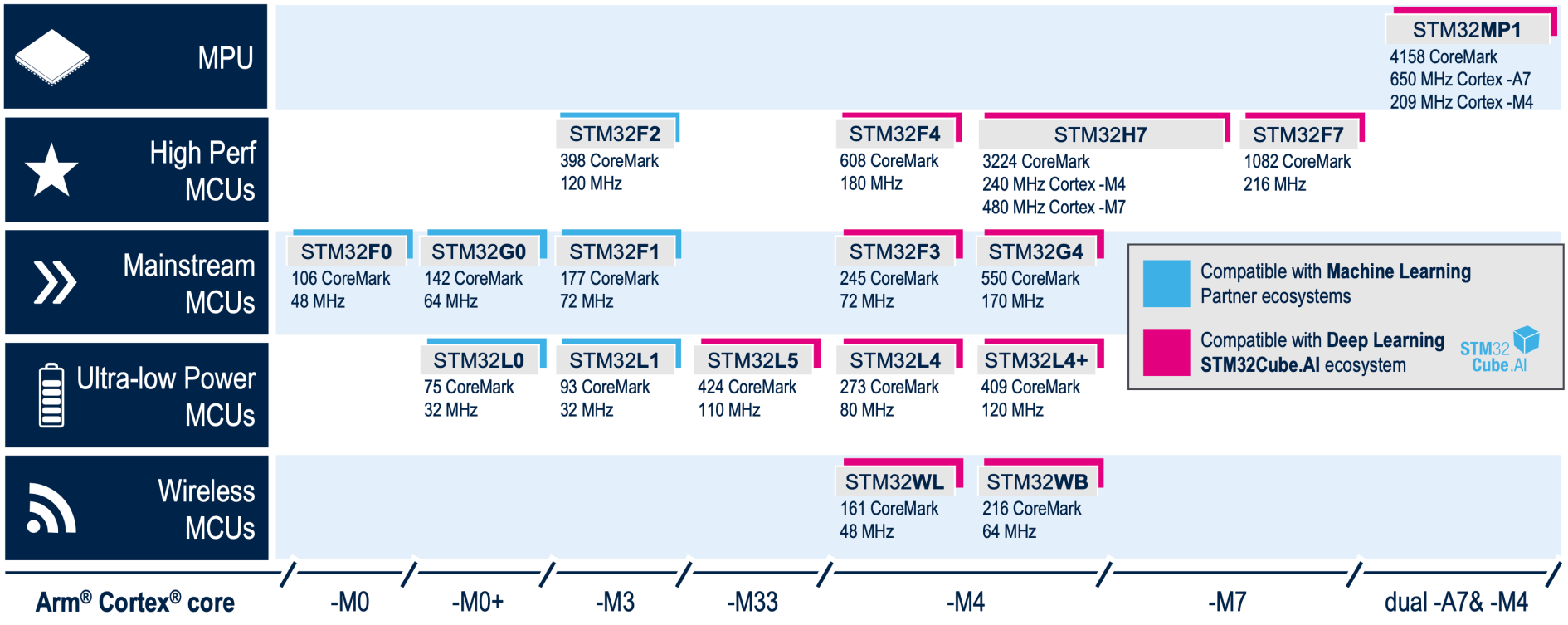
- ARM Cortex M3 & M4 Families
- Floating-Point Use Recap
- Tools Overview, C use
- Practical Lab Issues
- Processor Microarchitecture
- CMSIS-DSP
- In Tutorial/Lab: SW Infrastructure: CUBE, Q&A

ARM Cortex M Processors

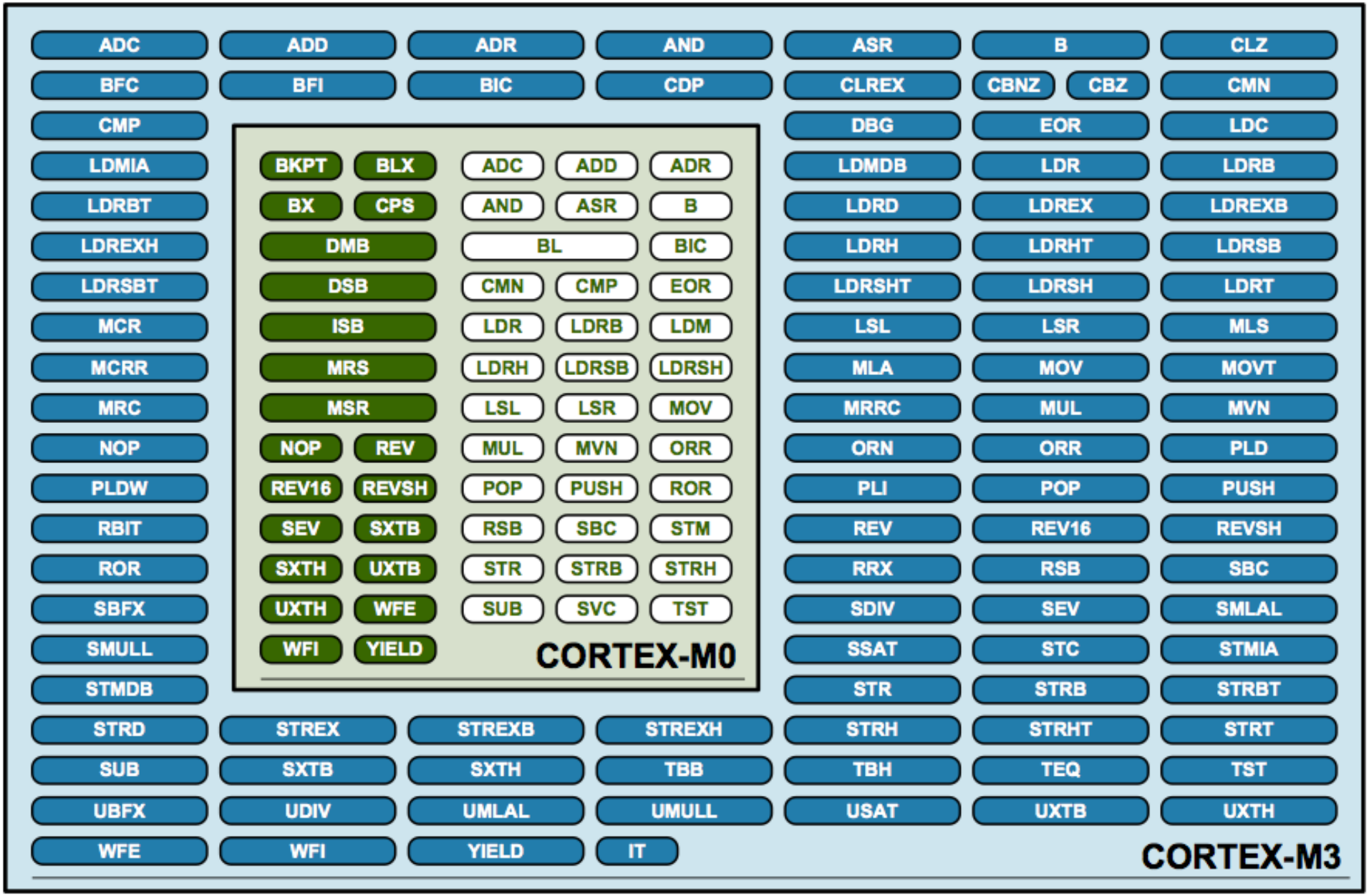
- Established Cortex M architectures
  - Cortex M0, M0+: low cost (V.6-M)
  - Cortex M1: for FPGA logic (V.6-M)
  - Cortex M3: “mainstream” (V.7-M)
  - Cortex M4: higher performance (V.7-M)
  - Cortex M7: highest performance (V.7E-M)
- Evolving, similar to Cortex A, Cortex R
  - V8-M: TrustZone; secure/non-secure core
    - Cortex M23, M33, M35P, M55 (V8.1)

STMicroelectronics Offerings

Market segmentation at work:



Cortex M3 ISA at Glance



Cortex-M4 processors

Cortex
Low-Power Leadership from ARM

Cortex-M4

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

CORTEX-M3

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-M0/M1

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

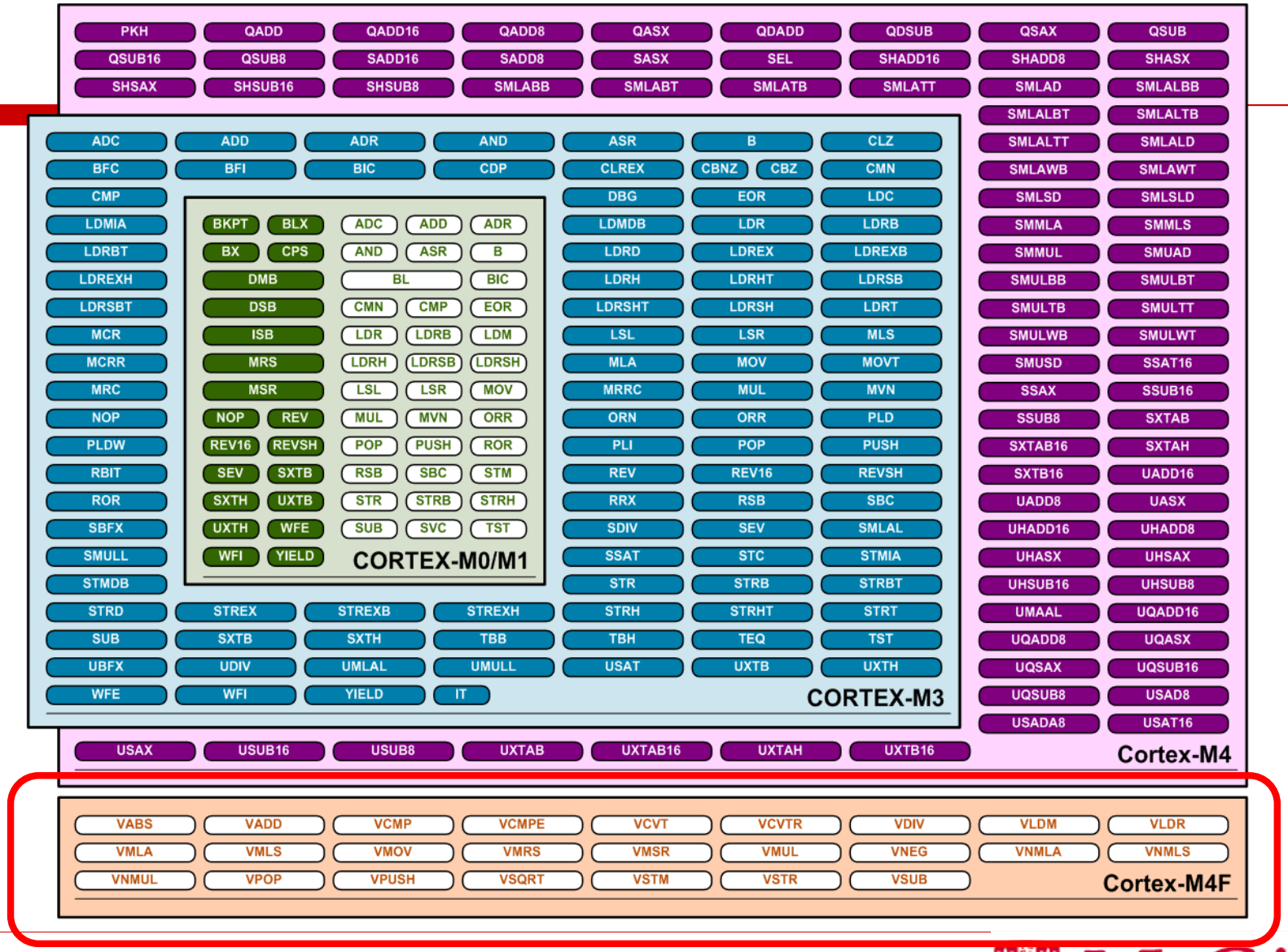
Cortex-M4

USAX, USUB16, USUB8, UXTAB, UXTAB16, UXTAH, UXTB16

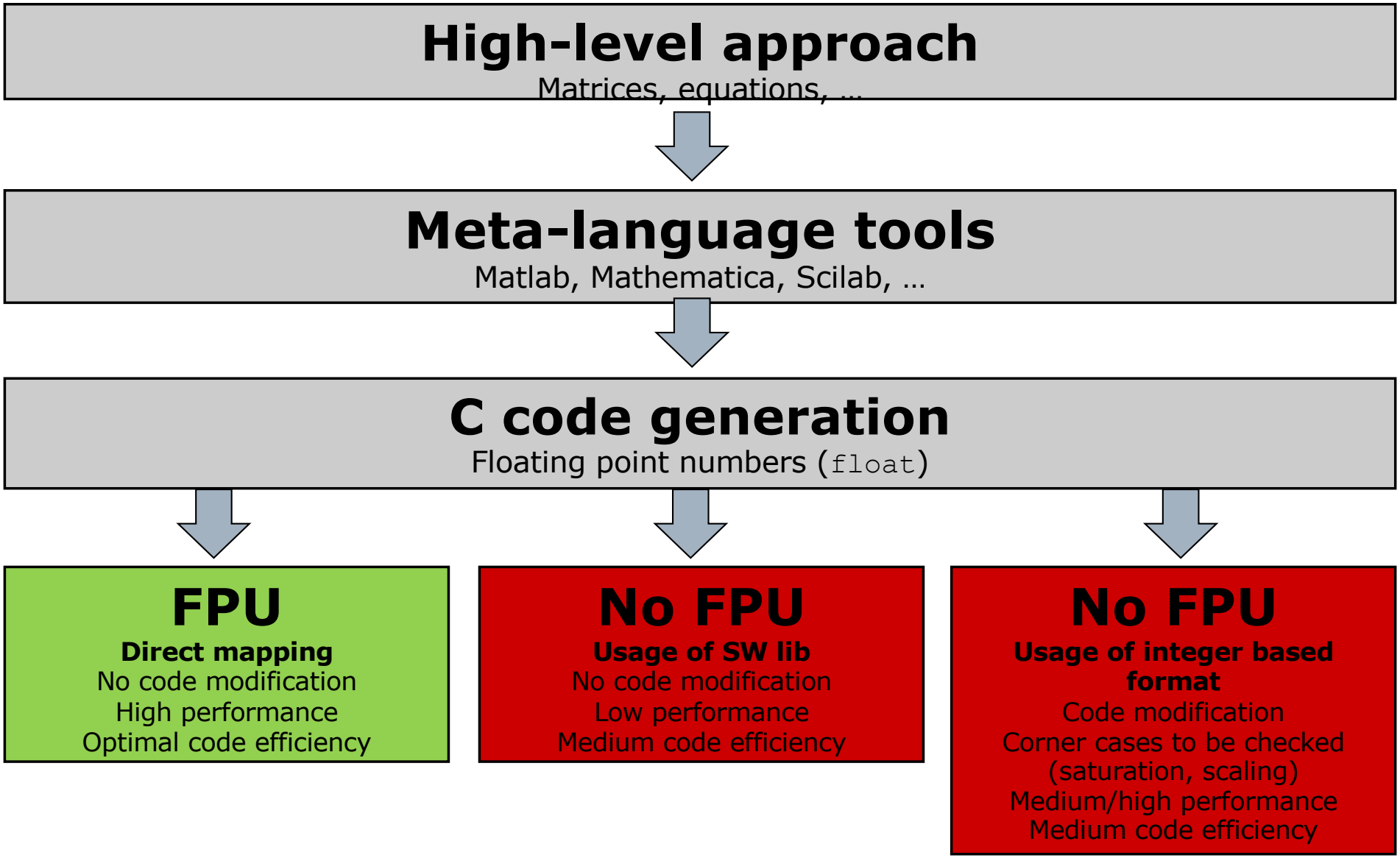
Cortex-M4F

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

FPU Instructions



FPU usage


High-level approach
Matrices, equations, …
Meta-language tools
Matlab, Mathematica, Scilab, …
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 cases to be checked (saturation, scaling)
Medium/high performance
Medium code efficiency

Benefits of FPUs

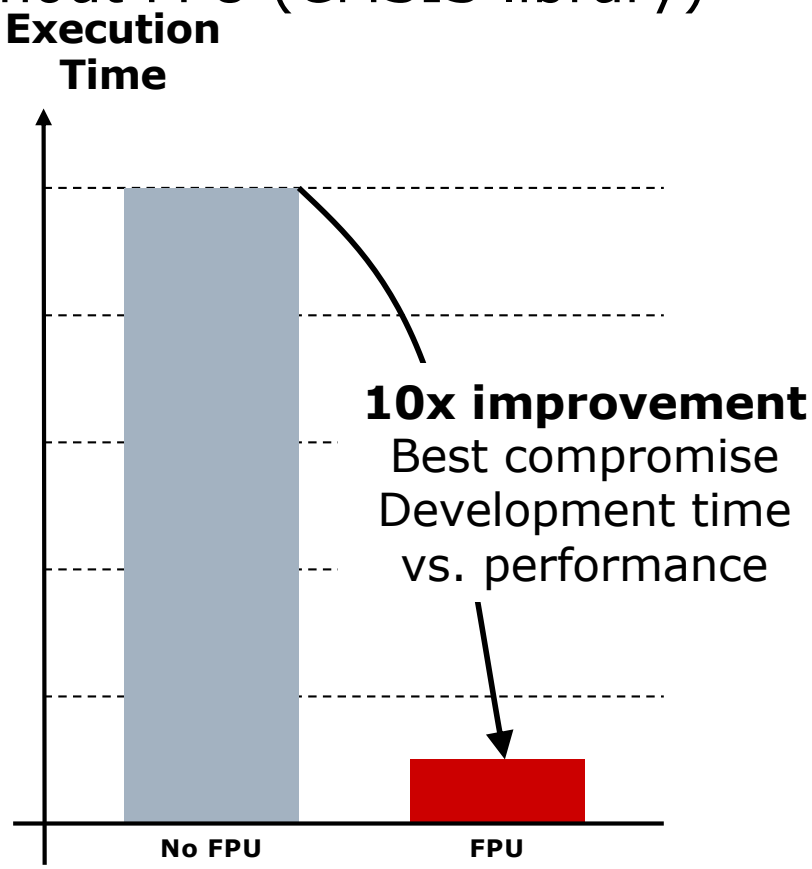
- **Handle "real" numbers (C float) without penalty**
- If FPU is not available
  - Need to emulate fp operation by software
  - Need to rework all its algorithm and fixed point implementation to handle scaling and saturation issues

- FPU eases usage of high-level design tools (MatLab/Simulink)
  - Now part of embedded development flow for advanced applications
  - Derivate code directly using native floating point leads to :
    - Faster development
    - More reliable application code as no post modification are needed (no critical scaling operations to move to fixed point)

McGill

Performance

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



C Language Translation

```
float function1(float number1, float number2)
{
        float temp1, temp2;

                temp1 = number1 + number2;
                temp2 = number1/temp1;

        return temp2;
}
```

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

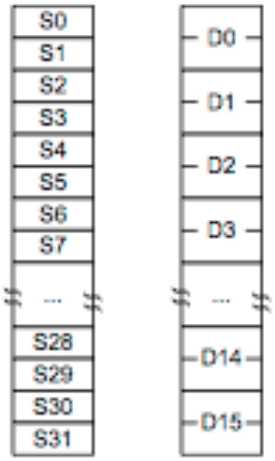
1 Assembly instruction

Call Soft-FPU

```
# float function1(float number1, float
number2)
# {
          PUSH      {R4,LR}
          MOVS      R4,R0
          MOVS      R0,R1
#           float temp1, temp2;
#
#           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}
# }
```

FPU and Precision

- Single precision FPU
- Dedicated 32 FPU registers
  - Single precision registers (S0-S31)
  - Can act as 16 FP Double registers for load/store operations (D0-D15)
  - FPSCR for status & configuration
- Conversion between
  - Integer numbers
  - Single precision floating point numbers
  - Half precision floating point numbers
- Floating point exceptions - interrupt

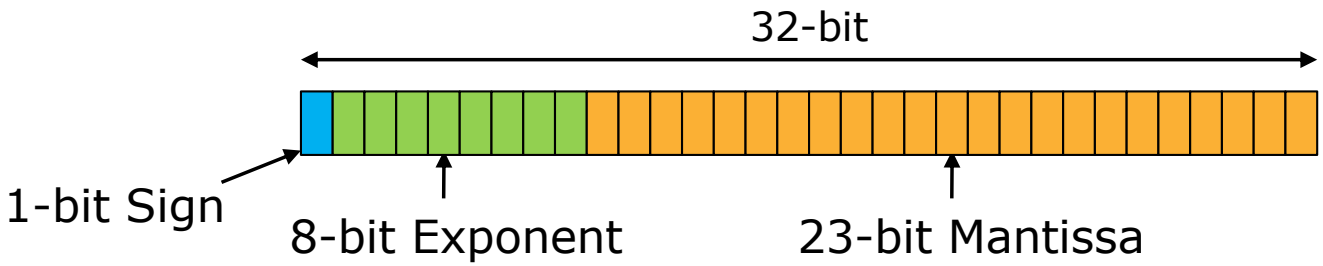


Rounding issues

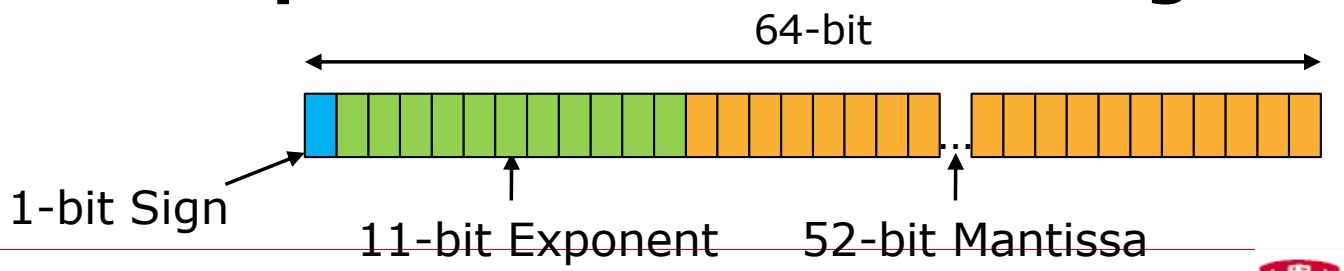
- The precision has some limits
  - Rounding errors accumulate leading to inaccurate results
- Examples
  - If you are working on two numbers in different base, the hardware automatically *denormalizes* to make the calculation in the same base
  - Subtracting two close numbers means losing the relative precision (also called cancellation error)
- Reorganizing operations may not yield the same result because of the rounding errors…

Floating-point: Number format

- Sign
- Biased exponent (exponent plus a constant bias)
- Fractions (or mantissa)
- **Single precision : 32-bit coding**



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



Number value

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

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

FP Special values

- **Denormalized (Exponent field all "0", Mantisa non 0)**
  - Too small to be normalized (but some can be afterwards)
  - $\mathbf{(-1)^s} \times \mathbf{(\Sigma(N_i.2^{-i}) \times 2^{-bias}}$
- **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

IEEE 754 Encoding Summary

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

Cortex M4: Relation to IEEE 754

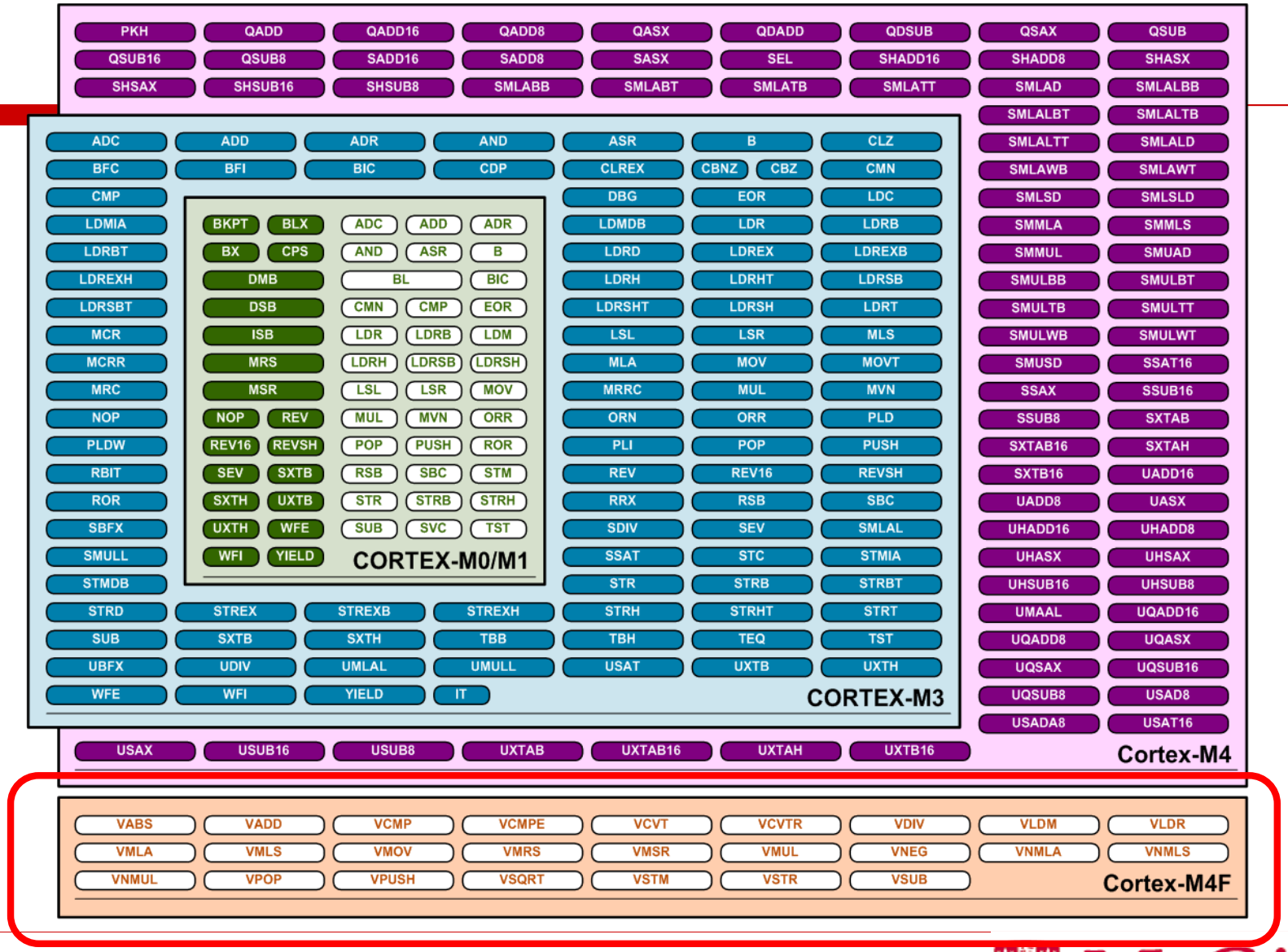
- Full Compliance mode
  - Process operations according to IEEE 754
- Alternative Half-Precision format
  - $\mathbf{(-1)^s} \times \mathbf{(1 + \Sigma(N_i.2^{-i}))} \times \mathbf{2^{16}}$, no de-normalized number
- Default NaN mode
  - Any operation with an NaN as an input or that generates a NaN returns the default NaN
- Flush-to-zero mode
  - De-normalized numbers are treated as zero
  - Associated flags for input and output flush

Complete implementation

- **Cortex-M4F does <u>NOT</u> 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

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

FP Comparison and Conversion

Operation	Description	Assembler	Cycle
Compare	float with register or zero float with register or zero	VCMP.F32 VCMPE.F32	1 1
Convert	between integer, fixed-point, half precision and float	VCVT.F32	1

FP: Load/Store Instructions

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

Floating-point Code Example

- Data movement and calculation

```
VLDR        s1, [r0, #0x0C]    // load var1 from the state variable (struct)

VLDR        s2, [r0, #0x00]    // load var2
VADD.F32    s1, s1, s2         // var1 <- var1 + var2

VSTR        s1, [r0, #0x0C]    // store v1 back in state variable
```

Lab 1: Kalman Filter using FP Ops

Step 1: Kalman filter in ARM+FP assembly languages

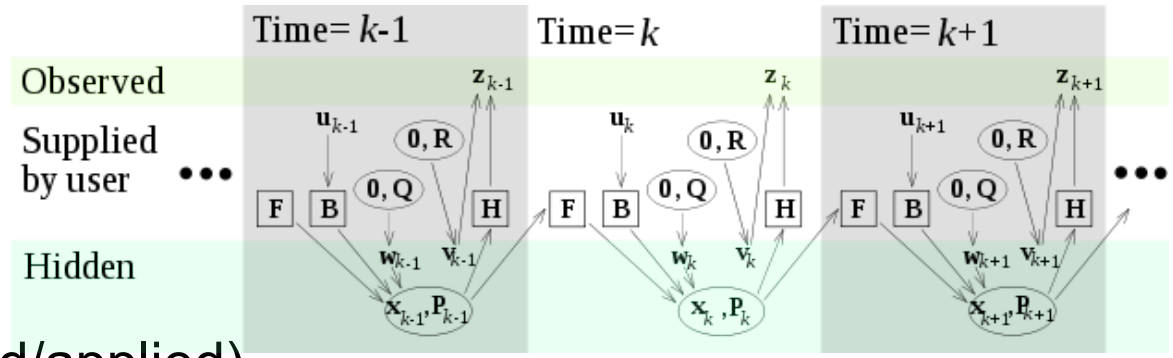
Step 2: Incorporate into a C program

Step 3: Validate the execution, compare to C code

Step 4: Apply ARM's CMSIS-DSP for Validation

Lab 1: Kalman Filter

- Kalman filter, a discrete-time system estimation
- <u>Find</u>: internal *state* **x** (non-observable)



- <u>From:</u>
  - Output: **z** (measured)
  - Input/control: **u** (measured/applied)
  - Multivariate normal distributed noise: **w** (covariances), **v**
- <u>As</u>

$$\mathbf{x}_k=\mathbf{F}_k\mathbf{x}_{k-1}+\mathbf{B}_k\mathbf{u}_k+\mathbf{w}_k$$

$$\mathbf{z}_k=\mathbf{H}_k\mathbf{x}_k+\mathbf{v}_k$$

Iterative prediction of state **x**, error covariances **P** from **z** prediction mismatch

Kalman Filter: Simple Code

- Code (Python programming language) for single variable x

```
class KalmanFilter(object):
        q = 0.0 # process noise variance, i.e., E(w^2)
        r = 0.0 # measurement noise variance, i.e., E(v^2)
        x = 0.0 # value
        p = 0.0 # estimation error covariance
        k = 0.0 # kalman gain

        def __init__(self, q, r, p=0.0, k=0.0, initial_value=0.0):
                self.q = q
                self.r = r
                self.p = p
                self.x = initial_value

        def update(self, measurement):
                self.p = self.p + self.q
                self.k = self.p / (self.p + self.r)
                self.x = self.x + self.k * (measurement - self.x)
                self.p = (1 - self.k) * self.p

                return self.x
```

McGill

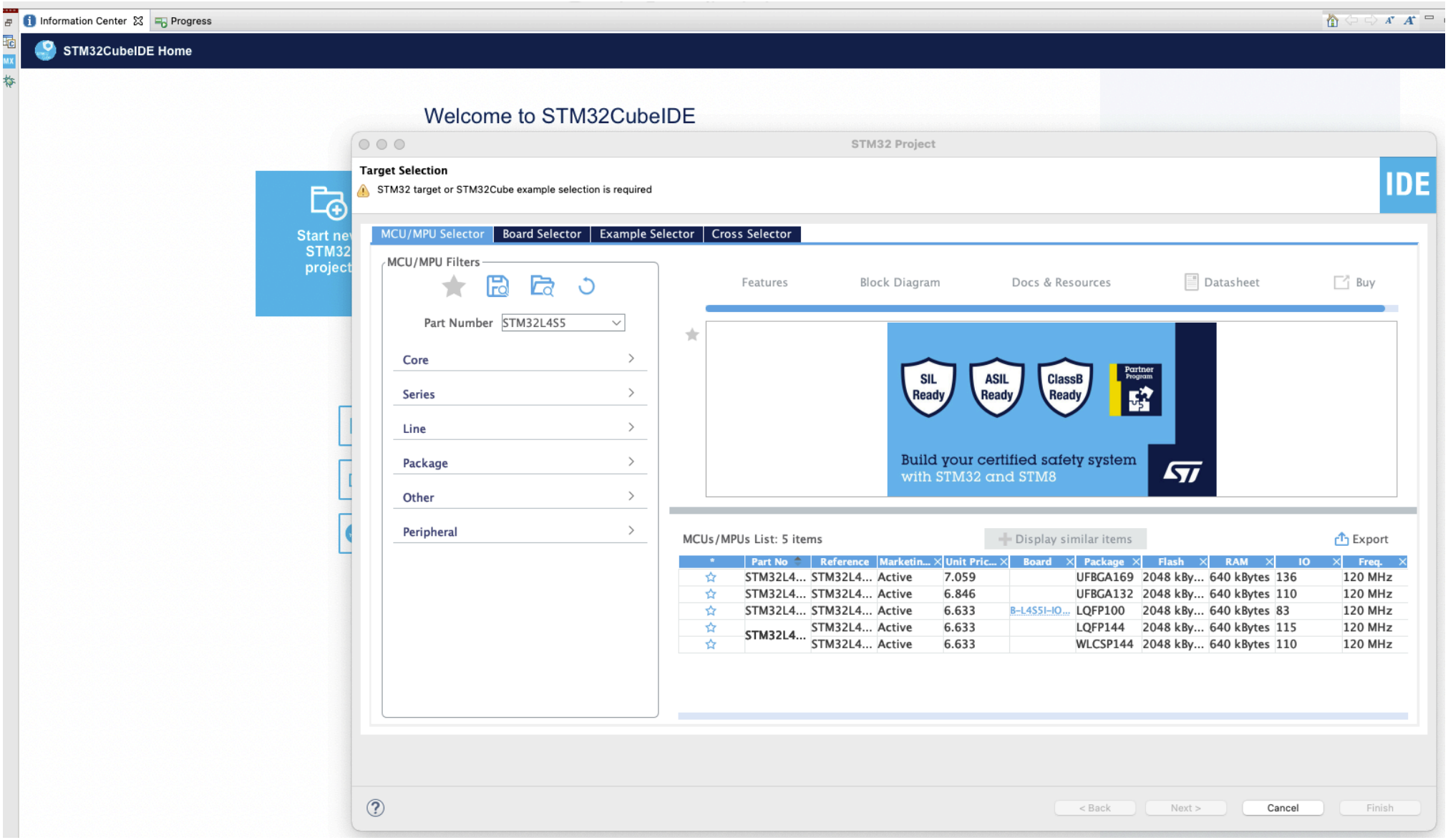
Lab 1: Tools

- Procedure: several options
  - Create project: assembler code only
  - When asked, include startup code
  - Edit startup code to branch to entry point
  - Add assembler code file, finish design
  - Compile/build and run

New Project in STM32CubeIDE

- From Information Center: Start New STM32 Project
- Choose the processor from your board
- Enter the project name
- Check all options: C code, executable target and CUBE project type

What Processor to Pick?



Project Options

workspace_1.4.0 - try/Core/Src/main.c - STM32CubeIDE

Information Center | Progress

STM32CubeIDE Home

Welcome to STM32CubeIDE

STM32 Project

Empty project name is not supported

IDE

Project

Project Name:

Use default location

Location: /Users/zeljko/STM32CubeIDE/workspace_1.4.0 Browse...

Options

Targeted Language

C C++

Targeted Binary Type

Executable Static Library

Targeted Project Type

STM32Cube Empty

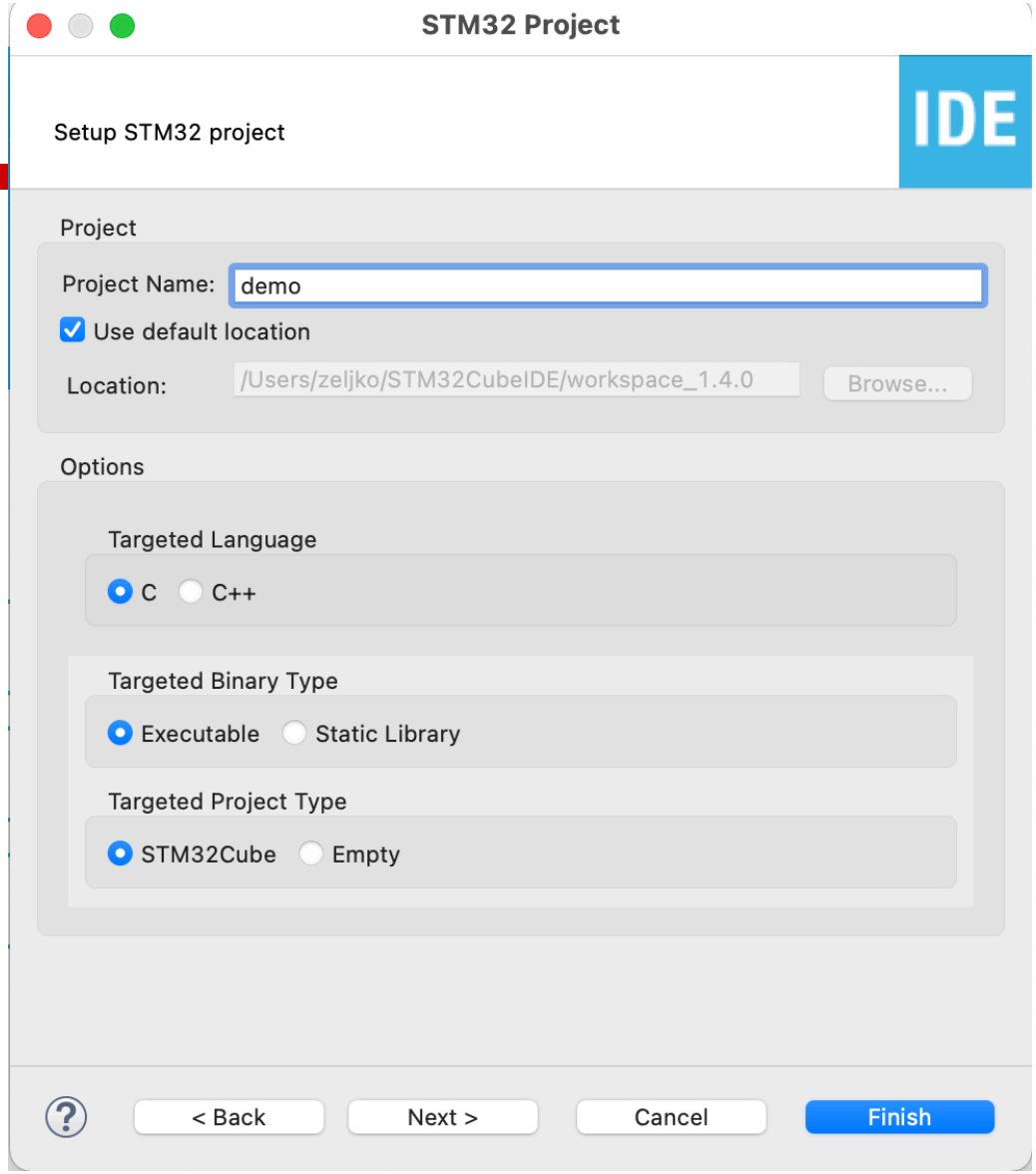
< Back Next > Cancel Finish

Support & Community

Twitter
Facebook
Youtube
ST Home
STM32MCU Wiki
STM32MPU Wiki
ST Community
ST Longevity Commitment

Project Options

- Default options



FPU programmer model - control

Address	Name	Type	Description
0xE000EF34	FPCSR	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

1. Note the address space: Peripherals (0xE0…)
2. FPU enabled by writing to CPACR(0xE000ED88)

```
SCR->CPACR |= 0x00F00000;
```

FP stack related registers

- **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

Status & Control Register

- **Floating-Point Default Status Control Register**
  - Details default values for Alternative half-precision mode, Default NaN mode, Flush to zero mode and Rounding mode

Status & Control Register

- **Floating-Point Default Status Control Register**
  - Default values for
    - Alternative half-precision mode (AHP)
    - Default NaN mode (DN)
    - Flush to zero mode (FZ)
    - Rounding mode (Rmode)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
N	Z	C	V	Reserved	AHP	DN	FZ	RMode		Reserved					
rw	rw	rw	rw		rw	rw	rw	rw	rw						

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								IDC	Reserved		IXC	UFC	OFC	DZC	IOC
								rw			rw	rw	rw	rw	rw

Media & FP Feature

- **Media & FP Feature Register 0**
  - Details supported mode, instructions and precision
- **Media & FP Feature Register 1**
  - Details supported instructions and additional hardware support

FP Exceptions

- **Invalid operation**
  - Resulting in a NaN
- **Division by zero**
- **Overflow**
  - The result depends on the rounding mode and can produce a +/-oo or the +/-Max value to be written in the destination register
- **Underflow**
  - Write the denormalized number in the destination register
- **Inexact result**
  - Caused by rounding

Register Content: Hints

- **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**
  - Inexact result flag may not be passed to the interrupt controller…

Processing Arithmetic Conditions

- Two ways:
  - In software, by checking FP status control (condition) register
  - Through interrupts (traps) - not yet for us
- Checking for FP arithmetic conditions
  - FP compare (VCMP instruction)
  - Move FP Status Control to (integer) registers: VMRS
  - Comparison and Jump/Conditional execution
- Checking in C: trick that (mostly) does the job

```
if (var1 == var2)
```

FPU Cookbook: FP SCR Register

- Condition codes register FPSCR is not in the main processor SCR!
- In assembly:

```
//Check the FPSCR for errors
VMRS R0, FPSCR //Load FPSCR to R0
```

- In C (via cmsis_gcc):

```
//Return FPSCR as an error flag
i = __get_FPSCR();
return i&0x0000000F;
```

Problems with Low-level Languages

- Programs in low-level languages require detailed documentation, as otherwise they hard to read for people who were not involved in the process of the program creation
- Example: Company "Ostrich" has recently re-developed their embedded software for flagship products
  - Developed in assembly, 80 percent working, 2000 lines of code
- Suddenly it has been realized that the product is not shippable
- Bugs: system lock-ups indicative of major design flaws or implementation errors + major product performance issues
- Designer has left the company and provided few notes or comments
- You are hired as a consultant. Do you:
  - Fix existing code?
  - Perform complete software redesign and implementation? In this case, which language?

Problem-oriented Language layer

- Compiled to assembly or instruction set level
- You will be using embedded C
- How does this differ from usual use of C?
  - Directly write to registers to control the operation of the processor
  - All of the registers have been mapped to macros
  - Important bit combinations have macros – use these, please !
  - Registers are 32 bits, so int type is 4 bytes
  - Register values may change without your specific instructions
  - Limited output system
  - Floating point operations very inefficient, divide + square-root to be avoided