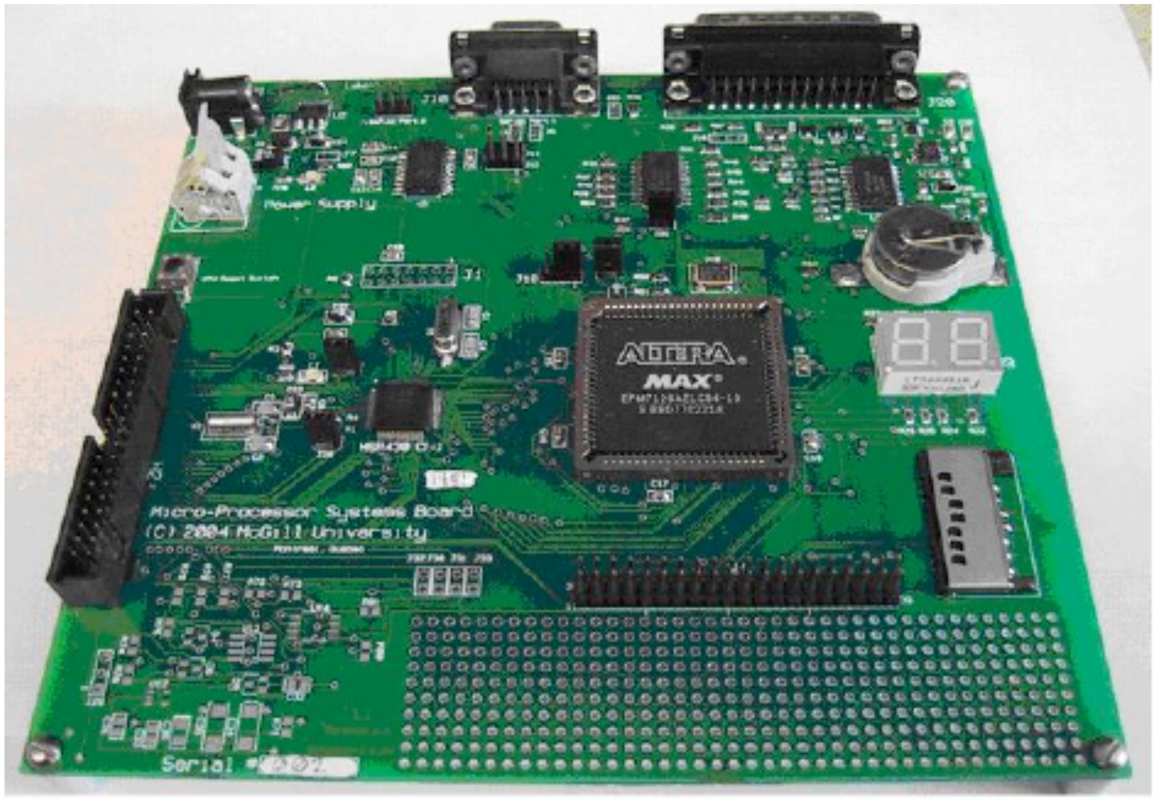
ECSE 426
Tools, Floating-Point and HW

Zeljko Zilic

zeljko.zilic@mcgill.ca



McGill

Acknowledgments: to STMicroelectronics for material on processors and the board

Outline

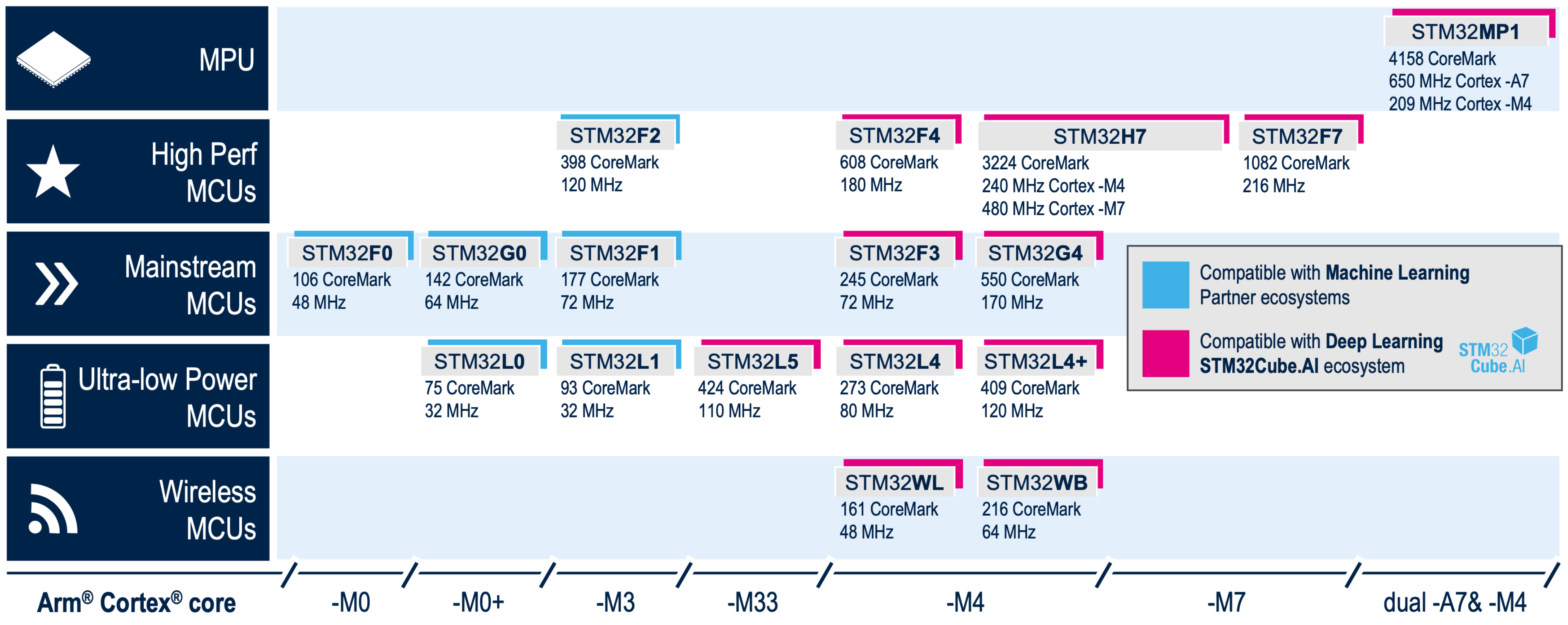
- ARM Cortex M3 & M4 Families
- Floating-Point Use Recap
- Tools Overview, C use
- Practical Lab Issues
- Processor Microarchitecture
- CMSIS-DSP
- In Tutorial/Lab: SW Infrastructure: CUBE, Q&A

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:

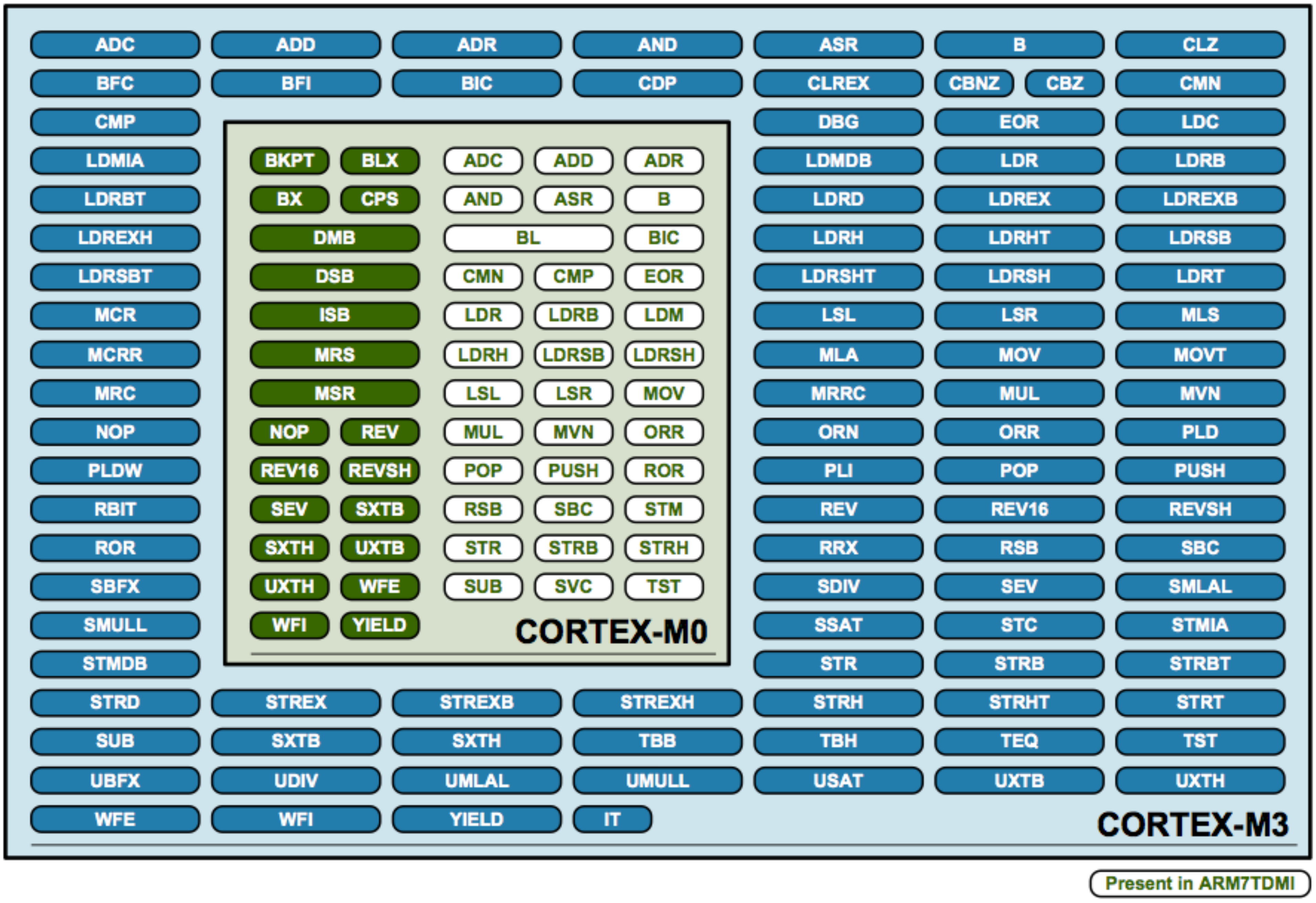


Arm® Cortex® core	-M0	-M0+	-M3	-M33	-M4	-M4	-M7	dual -A7& -M4
MPU								🟪 STM32**MP1** 4158 CoreMark 650 MHz Cortex -A7 209 MHz Cortex -M4
High Perf MCUs			🟦 STM32**F2** 398 CoreMark 120 MHz		🟪 STM32**F4** 608 CoreMark 180 MHz	🟪 STM32**H7** 3224 CoreMark 240 MHz Cortex -M4 480 MHz Cortex -M7	🟪 STM32**F7** 1082 CoreMark 216 MHz	
Mainstream MCUs	🟦 STM32**F0** 106 CoreMark 48 MHz	🟦 STM32**G0** 142 CoreMark 64 MHz	🟦 STM32**F1** 177 CoreMark 72 MHz		🟪 STM32**F3** 245 CoreMark 72 MHz	🟪 STM32**G4** 550 CoreMark 170 MHz		
Ultra-low Power MCUs		🟦 STM32**L0** 75 CoreMark 32 MHz	🟦 STM32**L1** 93 CoreMark 32 MHz	🟪 STM32**L5** 424 CoreMark 110 MHz	🟪 STM32**L4** 273 CoreMark 80 MHz	🟪 STM32**L4+** 409 CoreMark 120 MHz		
Wireless MCUs					🟪 STM32**WL** 161 CoreMark 48 MHz	🟪 STM32**WB** 216 CoreMark 64 MHz		

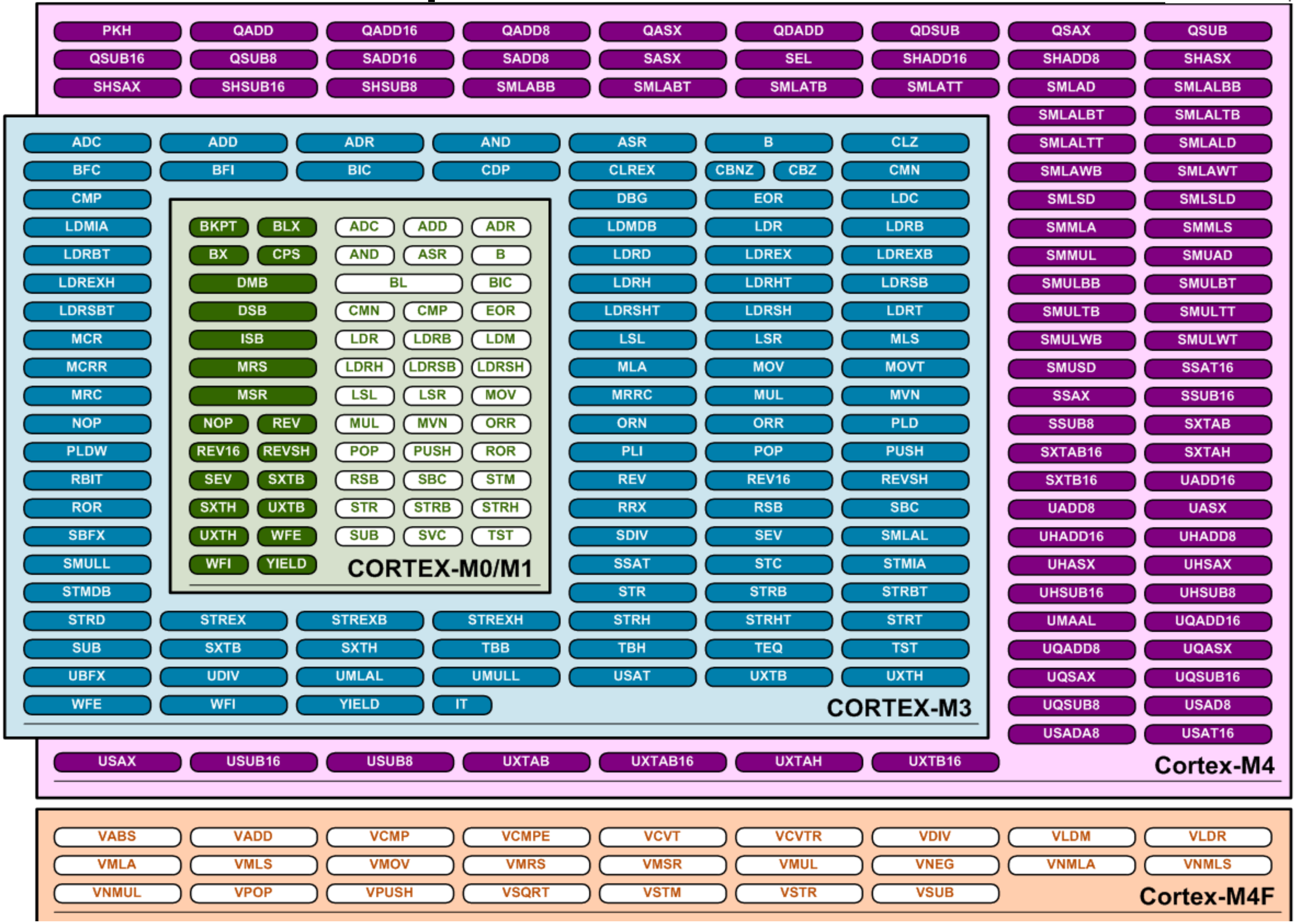
🟦 Compatible with **Machine Learning** Partner ecosystems

🟪 Compatible with **Deep Learning** **STM32Cube.AI** ecosystem

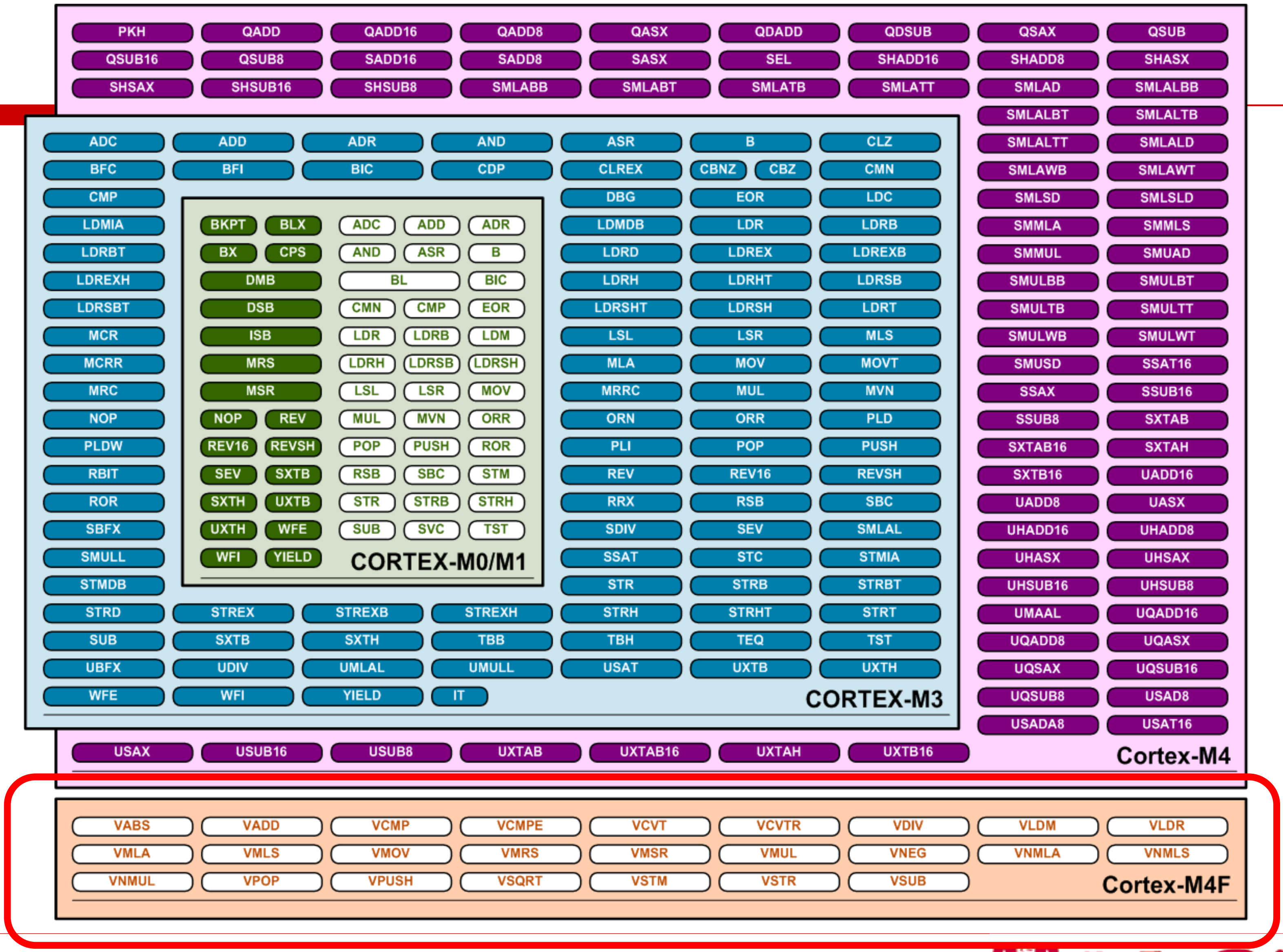
Cortex M3 ISA at Glance



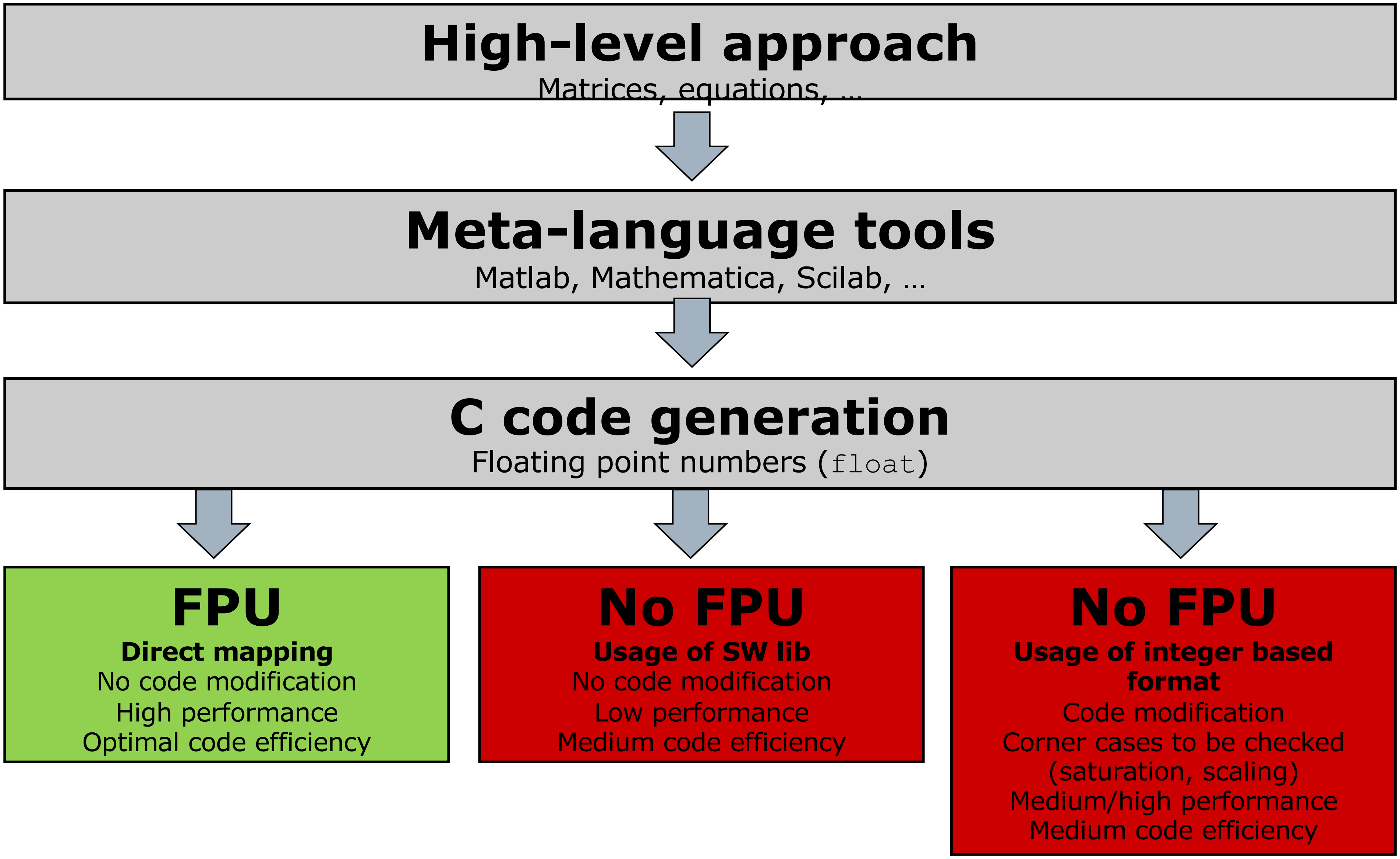
Cortex-M4 processors



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 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

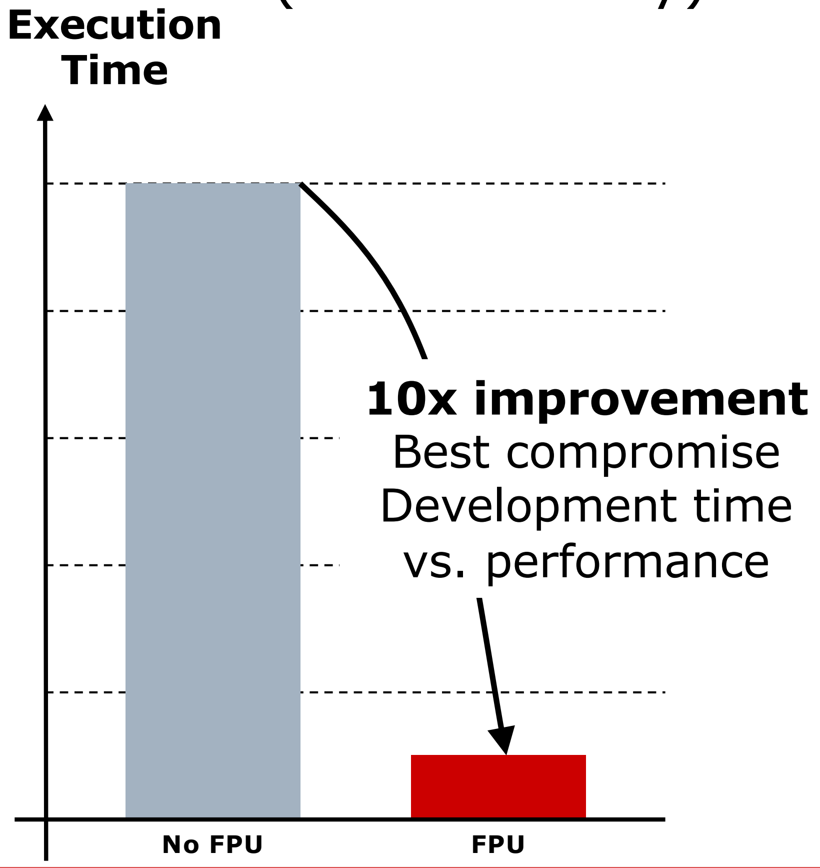
Benefits of FPUs

- **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)

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
# }
```

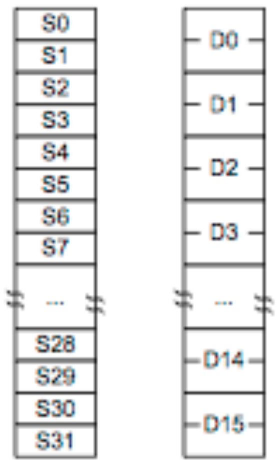
1 Assembly instruction

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…

Cortex M4: Relation to IEEE 754

- Full Compliance mode
  - Process operations according to IEEE 754
- Alternative Half-Precision format
  - $(-1)^s \times (1 + \Sigma(N_i . 2^{-i})) \times 2^{16}$, 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 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

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

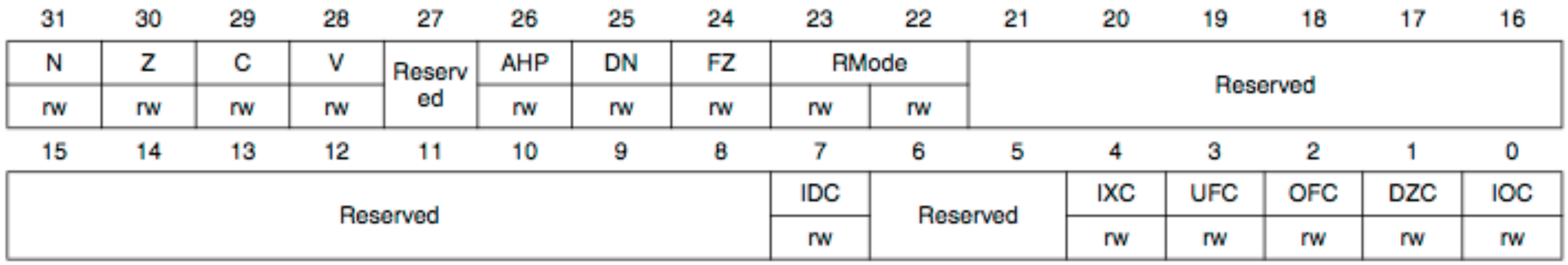
- **Floating-Point Context Control Register**
  - Indicates the context when the FP stack frame has been allocated
  - Context preservation setting
- **Floating-Point Context Address Register**
  - Points to the stack location reserved for S0

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;
```

Embedded C (more on MyCourses)

- C preprocessor
  - #define, #ifndef, #if, #ifdef, #else …etc.
    - #define specifies flags for conditional compilation
    - All remaining preprocessor statements initiate conditional compilation
      - Example: #ifdef compiles a block of code if some condition is defined in the #define statement
  - #ifndef is used to prevent multiple includes
    - Use #ifndef if you want to include a new definition
  - Widely used in Firmware (embedded SW) drivers

C Preprocessor Examples

Inline macro functions:

```
#define MIN(n,m) (((n) < (m)) ? (n) : (m))

#define MAX(n,m) (((n) < (m)) ? (m) : (n))

#define ABS(n) ((n < 0) ? -(n) : (n))
```

Macro used to set LCD control (## provides the way to concatenate actual arguments during macro expansion)

```
#define SET_VAL(x) LCD_Settings.P##x
```

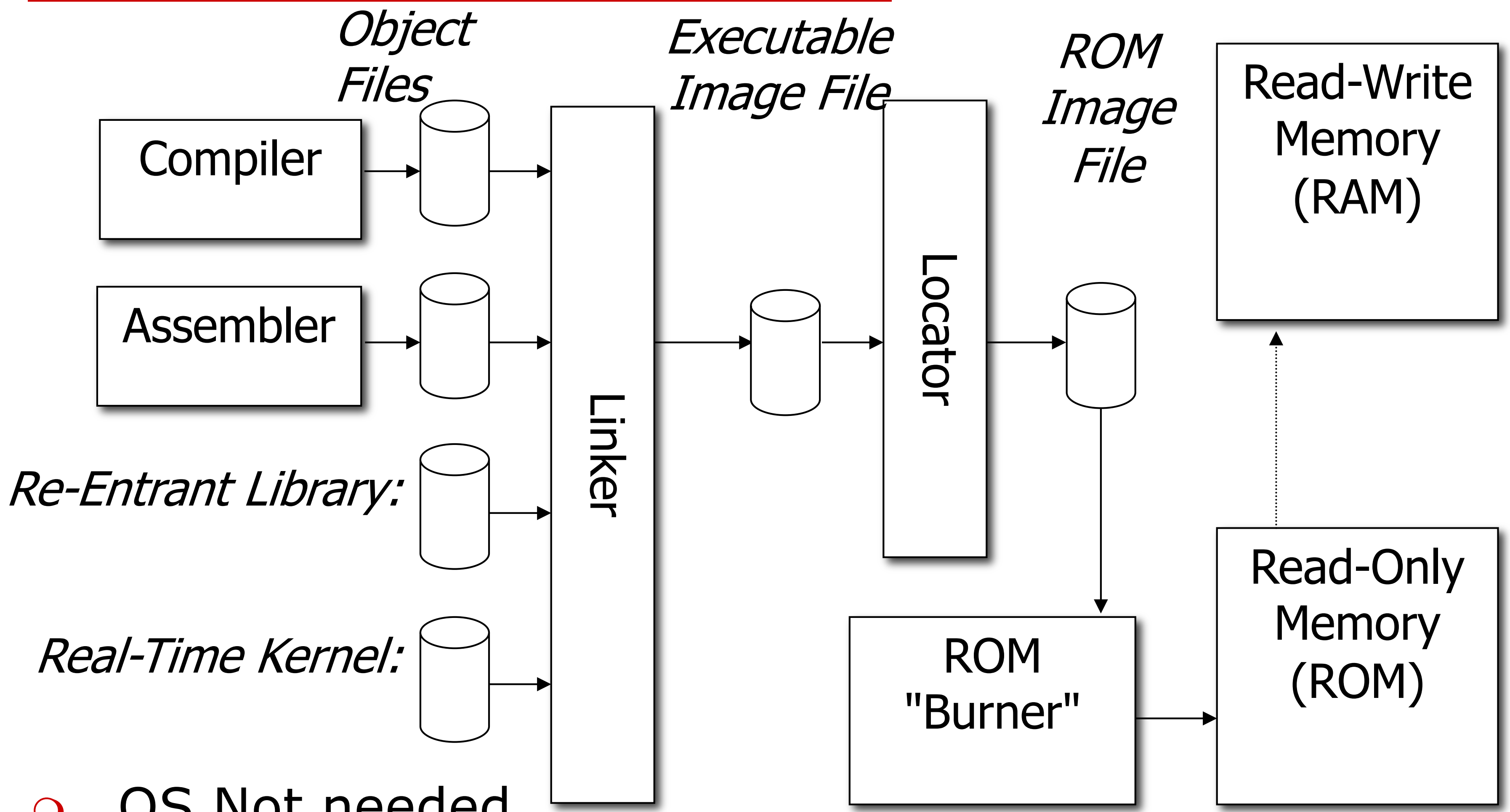
Nested macro definitions

```
#define SET(x, val) SET_VAL(x) = val
#define DEF_SET(x) SET(x, DS_P##x)
```

Global variables

- Distinguish global variables from local by choosing appropriate naming convention
  - Example: RX_Buffer_Gbl
  - Stick to your convention throughout the program
- Use them as Software flags
  - Example: PACKET_RECEIVED – use capitals
- Have them all in ONE place
- With your SW tool, global variables are easy to observe during debug (“watch variables”)

Embedded Build and Load



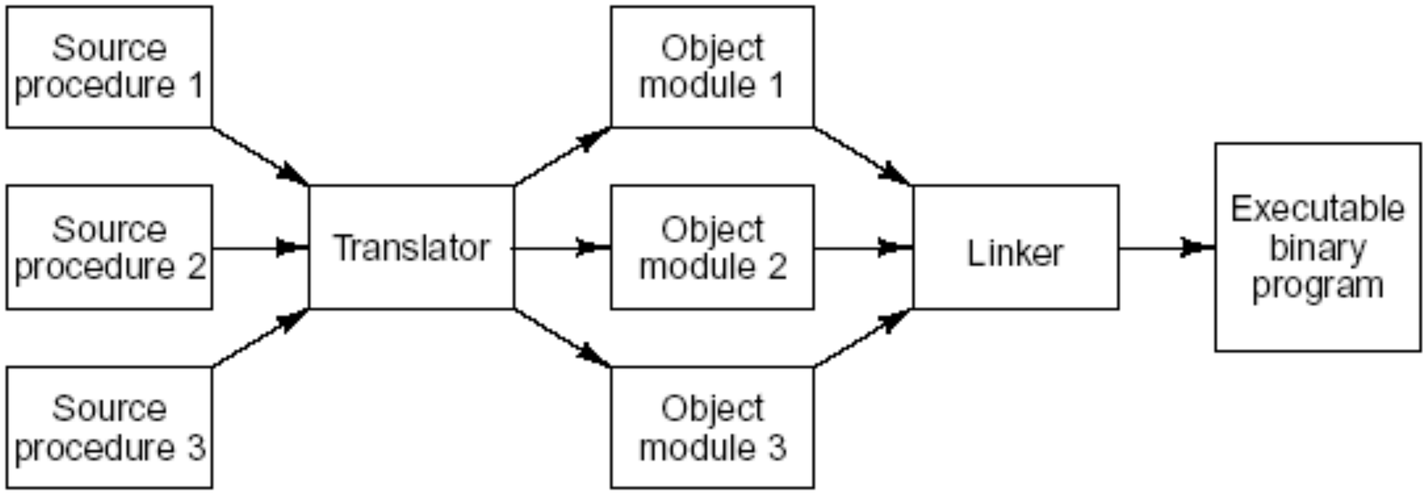
- OS Not needed

Linker

- Each procedure in assembly is translated individually and stored in the translated output disk
  - Once all the procedures are translated they have to be put together, i.e., linked, before the program can run
- When assembly source file is assembled by an assembler and C source file is compiled by C compiler, then the resulting two object files can be linked together by a software called **linker** to form the final executable
- Steps in program translation
  - Compilation (C programs) or assembly (assembly programs) of the source procedures
    - Task performed by compiler (assembler)
  - Linking of object modules
    - Done by linker

Modules - Basic Concepts

- Linking multiple files
  - Good SW design techniques
- Principle: have standardized modules
  - Some overhead – module structure

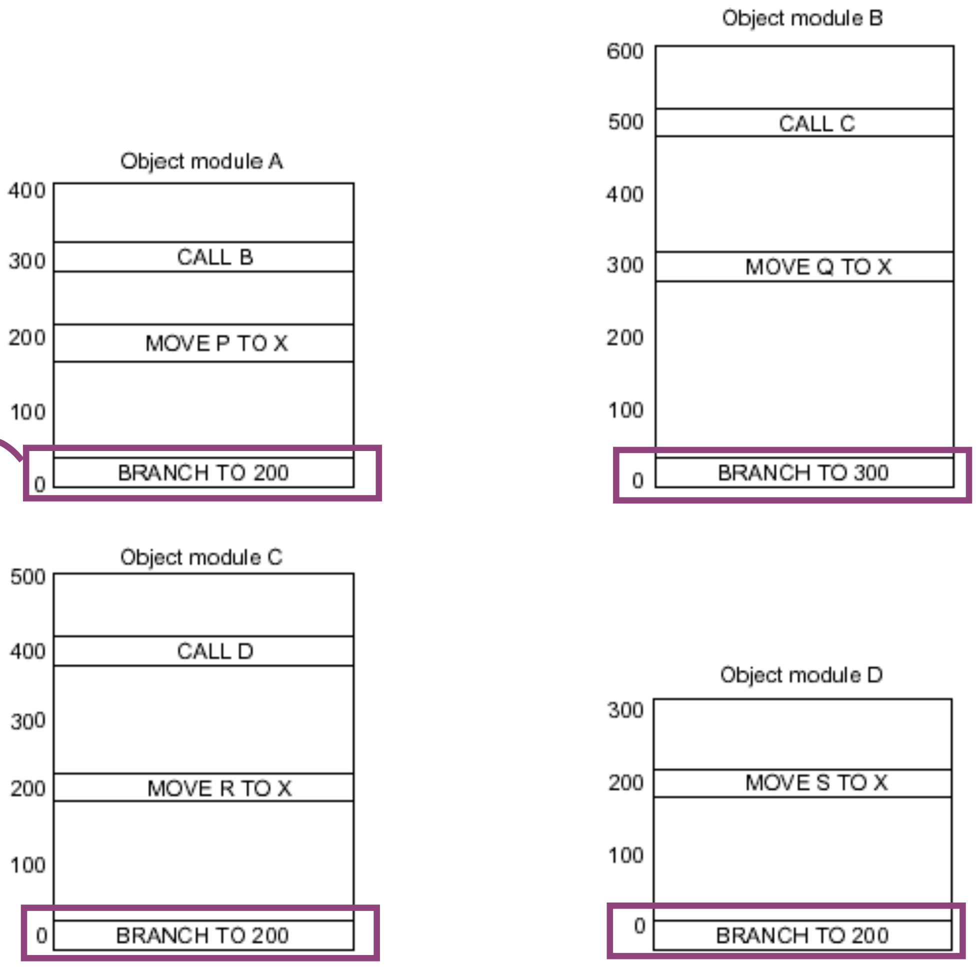


Benefits of Individual Procedure Compilation

- Assembly files can be written using any syntax and assembler available
- All the assembly code exists in a separate file, which is comfortable if any changes are required in the assembly code
  - If compiler/assembler were to translate a series of source procedures directly into a machine language, then any changes in the original source file would require retranslation of all the source file into machine language
  - When each procedure is translated into a separate file, then upon changes to the source file of this procedure it is necessary to retranslate only the source file of the procedure, while all the remaining code stays the same
    - Relinking of a retranslated file to the rest of the code is required

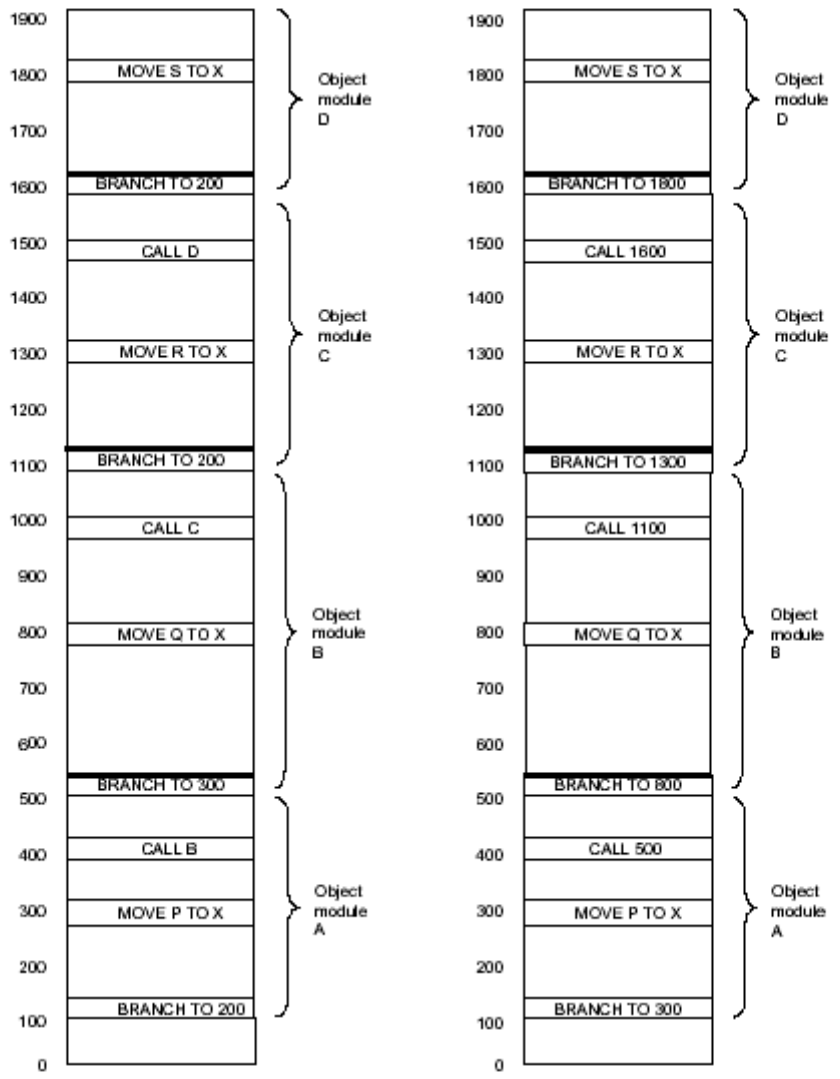
Module - Location in Virtual Memory

- At the beginning of assembly translation instruction counter is set to 0
  - Object module will be located at (virtual) address 0 during execution
- Example: 4 modules each beginning with BRANCH



Module "Composition"

- Linker brings all objects into main memory
  - By that the exact image of executable program's virtual address space inside the linker is formed
    - Typically few initial memory locations are reserved for various communications with operating system and is not accessible for program data
      - In the example the starting memory location is 100



Relocation Problem - Problems with Address Changes

- The created image of executable binary program has wrong reference addresses for branch instructions
  - Addresses were created for modules, whose starting instruction were all located at address 0 of virtual memory
    - That means that the branch addresses of each of the four modules in the example were calculated w.r.t. address 0 of the first object instruction
    - Upon changing the initial address of each of the object instructions, the addresses pointed by the branch instructions must be recalculated
- **Relocation problem** happens as each object module represents a separate address space

Solving Relocation Problem

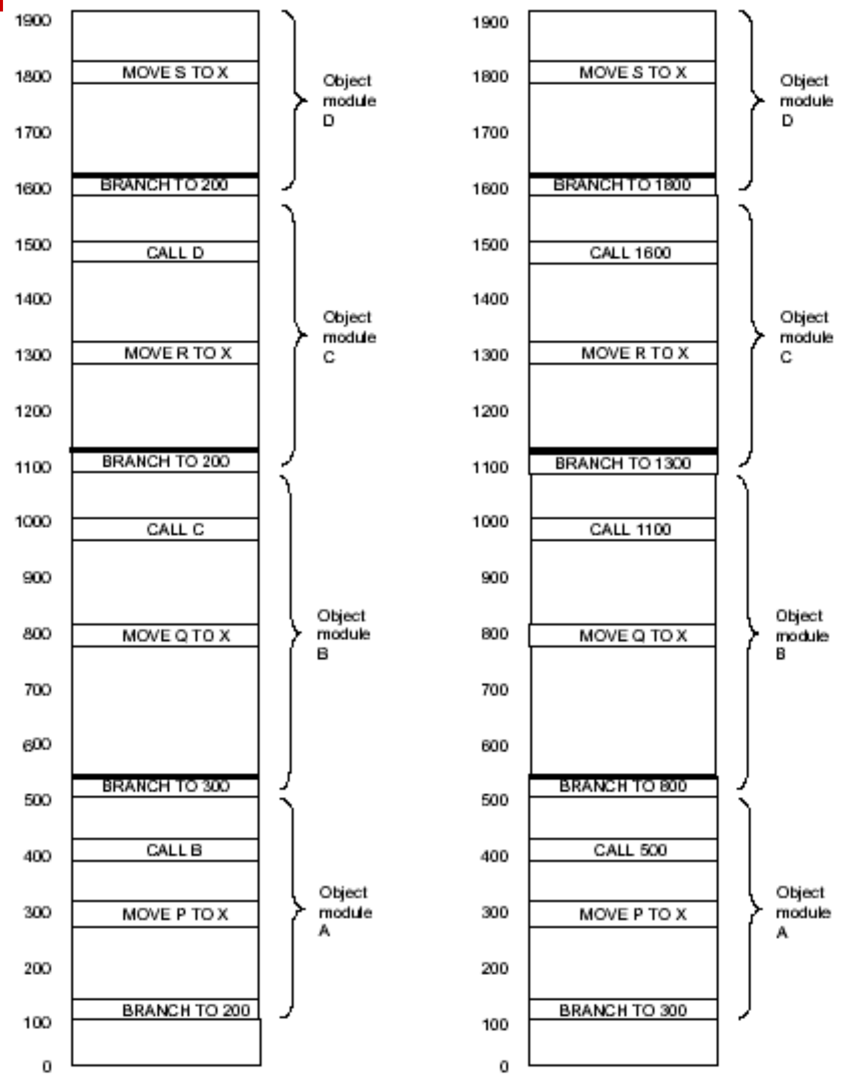
- Linker solves the relocation problem by:
  - Getting an exact estimate of the sizes of all object modules
  - Assigning the starting address to the first module
  - Assigning the right reference addresses for branch and jump procedures inside each module based on the starting address and the size of each module
- The actual steps taken by linker can be summarized as:
  - Construction of a table of all object modules and their length
  - Assignment of a starting address to each object module based on the above table
  - Addition of the relocation constant to all instructions, which reference memory
    - Relocation constant is equal to the starting address of its module
  - Finding of all instructions that reference other procedures and inserting the address of these procedures in place

Example: Object Module Table

- The object module table for the modules to the right

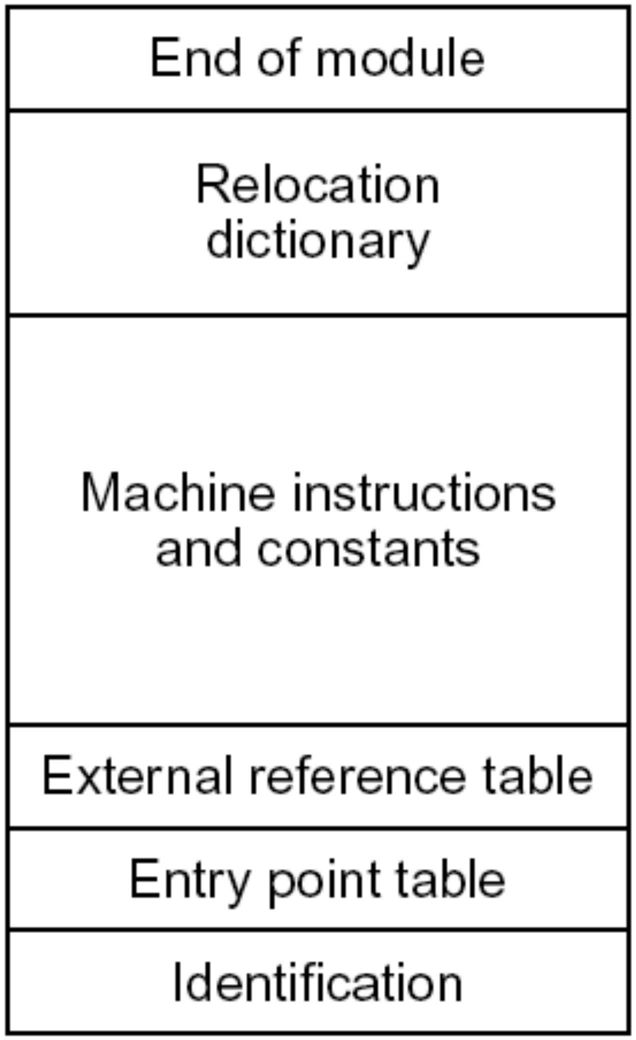
Module	Length	Starting Address
A	400	100
B	600	500
C	500	1100
D	300	1600

- Most linkers require two passes
  - During the first pass the linker reads all the object modules and builds up a table of module names and length, and a global symbol table consisting of all entry points and external references
  - During the second pass the object modules are read, relocated and linked one module at a time



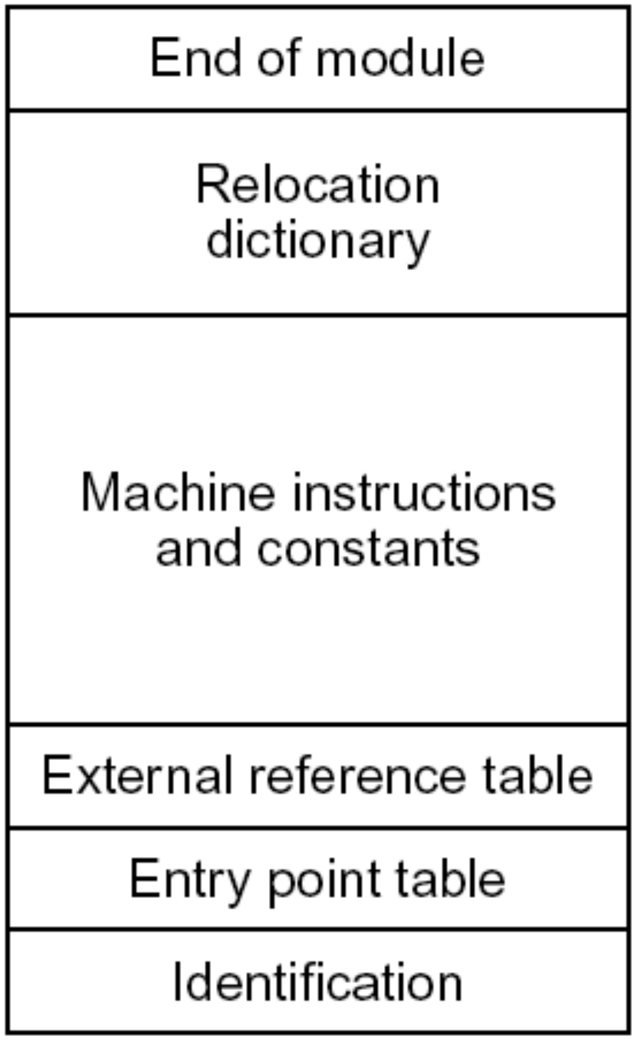
Object Modules

- Six different components
  - First part: name of the module and likner information such as length of various parts of module
  - Second part: list (and value) of symbols defined in this module subject to references by other modules
  - Third part: list of symbols that are used in the module but are defined in other modules (including list of all machine instructions used with each symbol)
  - Fourth part: assembled code and constants
    - The only part which is actually loaded into memory to be executed
    - Other parts will be used only by linker and be discarded before the execution begins
  - Fifth part: relocation dictionary
    - Instructions in each module which contain memory addresses must have a relocation constant added when put all modules are put together
  - Sixth part: end-of-module mark



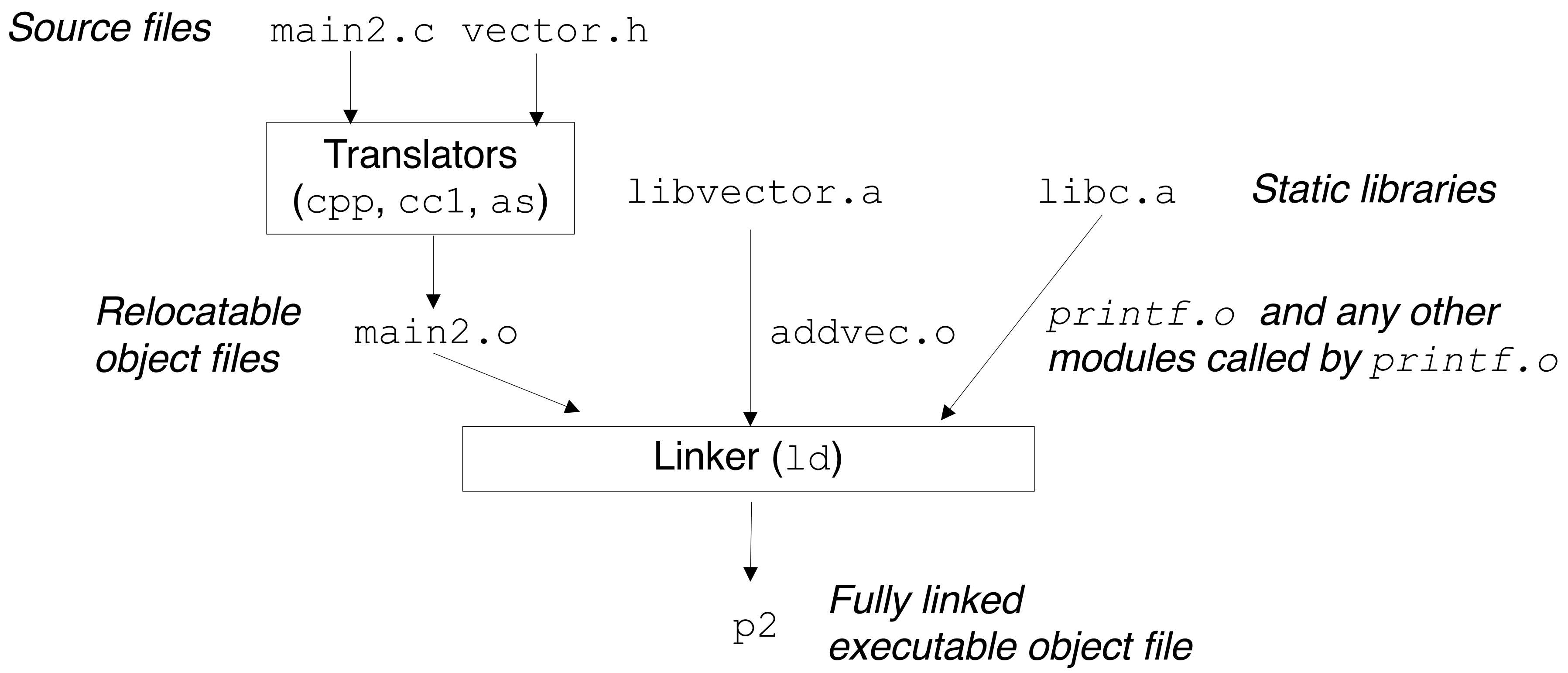
Object Modules

- Six different components
- Module standards
  - Unix
    - ELF
    - Shared Objects (.so files)
    - .o files
  - Windows
    - DLL
    - .obj



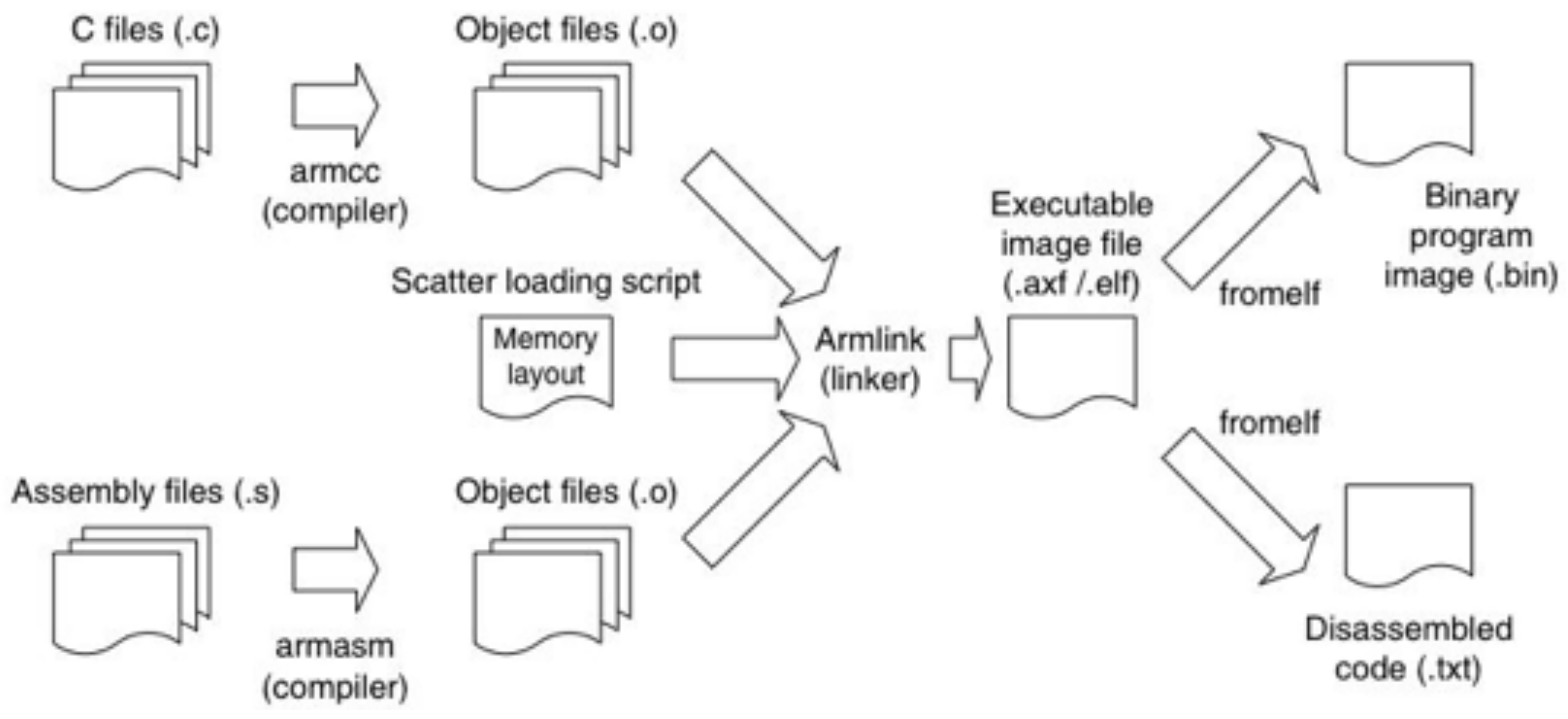
Linking Together

- Static Library - Unix/Linux



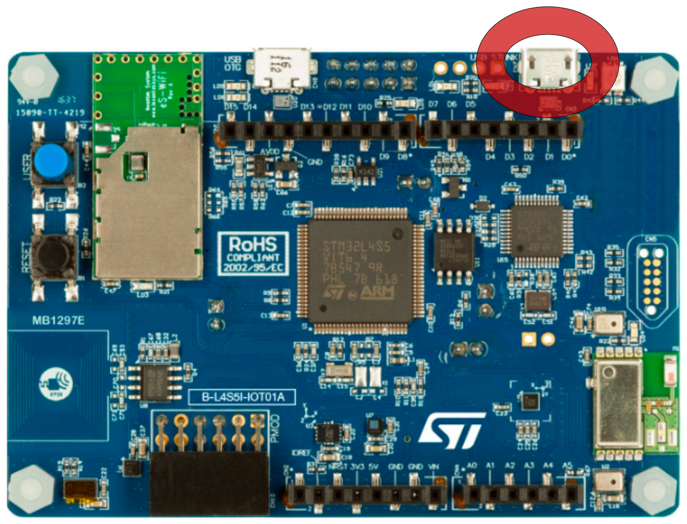
ARM Tools Flow: Recap

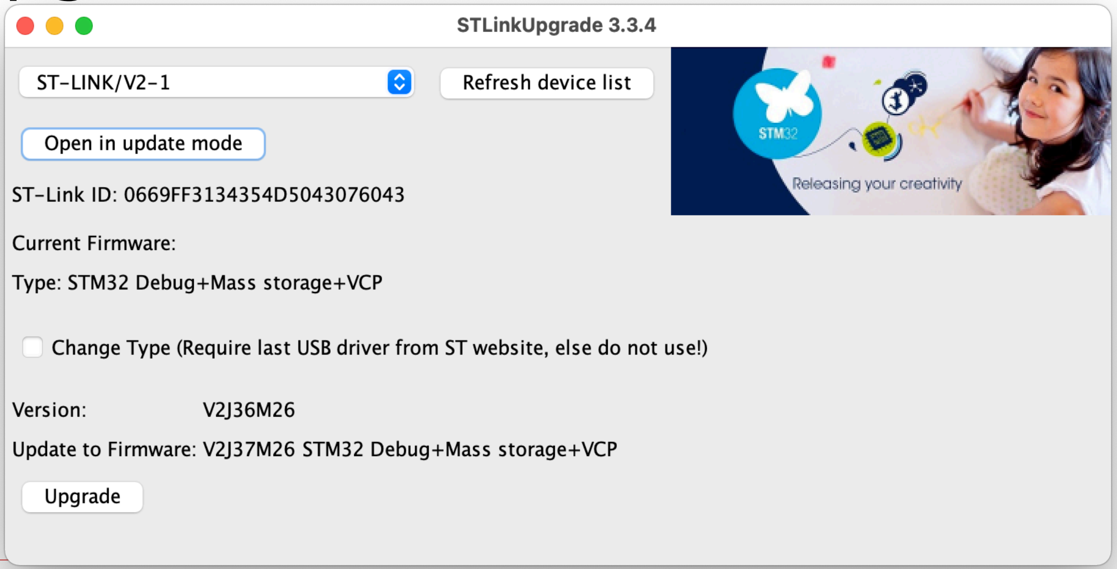
- ARM Standard using ARM Tools



STM32CubeIDE+Board Hints

- Integrated: single download and install
  - On Windows might need STLink tool, driver
- Checking STLink (connect the board!)
  - Help->ST-LINK Upgrade





ST-Link ID: 0669FF3134354D5043076043

Current Firmware:

Type: STM32 Debug+Mass storage+VCP

Change Type (Require last USB driver from ST website, else do not use!)

Version: V2J36M26

Update to Firmware: V2J37M26 STM32 Debug+Mass storage+VCP

FP Compilation Options

Access at:

Project->

Properties

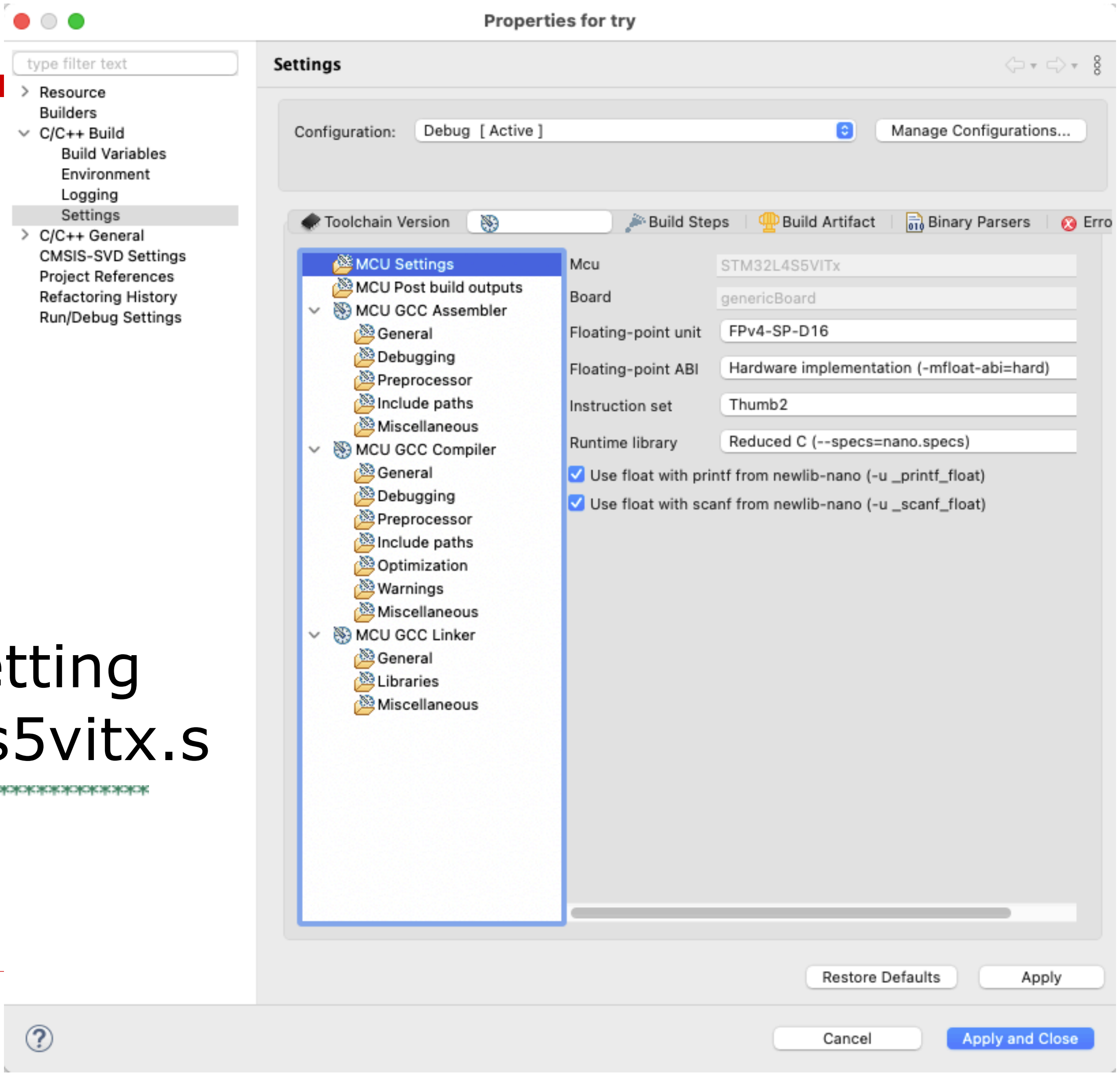
(C/C++ Build

-> Settings)

Check assembler setting in startup_stm32l4s5vitx.s

```
*******************************************
*/

.syntax unified
  .cpu cortex-m4
  //.fpu softvfp
  .thumb
```



Properties for try

type filter text

- Resource
- Builders
- C/C++ Build
  - Build Variables
  - Environment
  - Logging
  - Settings
- C/C++ General
- CMSIS-SVD Settings
- Project References
- Refactoring History
- Run/Debug Settings

Settings

Configuration: Debug [Active] — Manage Configurations...

Toolchain Version | Build Steps | Build Artifact | Binary Parsers | Erro

- MCU Settings
- MCU Post build outputs
- MCU GCC Assembler
  - General
  - Debugging
  - Preprocessor
  - Include paths
  - Miscellaneous
- MCU GCC Compiler
  - General
  - Debugging
  - Preprocessor
  - Include paths
  - Optimization
  - Warnings
  - Miscellaneous
- MCU GCC Linker
  - General
  - Libraries
  - Miscellaneous

Mcu	STM32L4S5VITx
Board	genericBoard
Floating-point unit	FPv4-SP-D16
Floating-point ABI	Hardware implementation (-mfloat-abi=hard)
Instruction set	Thumb2
Runtime library	Reduced C (--specs=nano.specs)

☑ Use float with printf from newlib-nano (-u _printf_float)

☑ Use float with scanf from newlib-nano (-u _scanf_float)

Restore Defaults | Apply

Cancel | Apply and Close