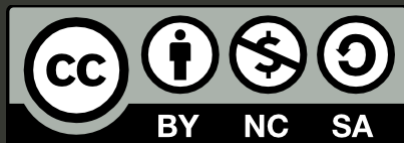


# ENSC254 – Floating Point Computation

Updated July 2021

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/4.0/>. Please share all edits and derivatives with the below authors.

© 2017 -- Fabio Campi and Craig Scratchley  
School of Engineering Science  
Simon Fraser University  
Burnaby, BC, Canada



# Floating Point Operations

- Integer computation is the choice of reference for most signal processing applications (Sound / Video / Telecommunications) and embedded processors have been mostly focusing on integers
- *Floating point numbers come into play when we are interested in a large range of values*
- Traditionally, floating point algorithms have been manually ported to the integer domain for embedded systems. BUT, as the complexity of embedded systems and their performance increase exponentially, floating point computations are now increasingly common
- Floating point computation can be applied with two strategies
  - HARDWARE IMPLEMENTATION
  - EMULATION

# IEEE 754 - 2008

- FP calculation can be performed in hardware in different ways and precisions, that affect significantly the chip area and power consumption
- In order to clarify what expectations a customer should have from a computer, *IEEE defined a reference standard document, 754-2008*
- This standard specifies interchange and arithmetic formats and methods for floating-point arithmetic in computer environments.
- It specifies exception conditions and their default handling.
- An implementation of a floating-point system conforming to this standard **may be realized entirely in software/firmware, entirely in hardware, or in any combination of software/firmware and hardware.**

# Floating Point in C environment

- IEEE 754-2008 defines 4 binary FP formats: 16, 32, 64 and 128 bits (*half precision, single precision, double precision and quad precision*)
- C, C++ and Java use the following type formats:
  - 32-bit (FLOAT)
  - 64-bit (DOUBLE)
- ARM supports the 16-bit format in storage, and 32-bit in computation [but one needs to activate such support with a compiler flag]

# Floating point Support in ARM

## FLOATING POINT

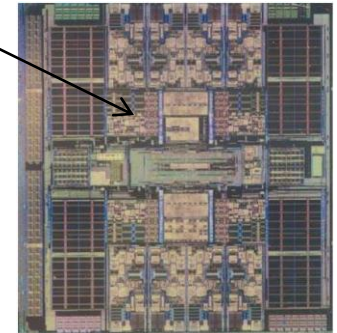
Manual  
Porting



Software  
Libraries



Hardware

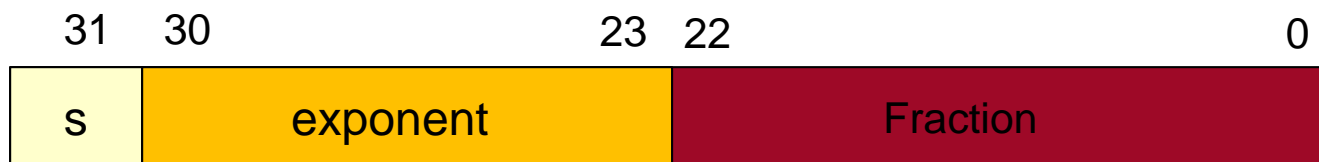


- Support of FP on ARM has followed the market of embedded systems:
  - The very successful ARM7TDMI architecture, does not have any hardware Floating Point support
  - As the need for floating point applications emerged from MATLAB into embedded systems, users started to make manual porting by hand
  - Soon, the cost of porting became unbearable, and to avoid losing market share ARM started to work on emulation libraries, which would efficiently translate software FP operations into integer HW operations using the GP registers as operands
  - As FP applications became increasingly common in embedded systems and more transistors became available, a HW FPU starts appearing in ARM9 and ARM11

# FP Numbers representation

Floating point numbers are represented in HW according to the following equation:

$$F = (-1)^s * 2^{exp-bias} * 1.f$$



- bias=Fixed value depending on the format: 127 for single precision
- exp ranges from 1 to 254, and the representable EXP from -126 to 127
- exp= 0 and exp=255 have a special meaning. A number is not represented for exp=255
- The part 1.f is called the “*Significand*” and MUST be by definition between 1 and 2

See Pyeatt textbook Section 8.5 or Hohl textbook page 180 for details for analogous 16 and 64 bit formats

# Example from Hohl textbook (Page 181)

## **Describe the single-precision representation of 6.5:**

S=0 (Positive number)

We need to find a representation of 6.5 between 1 and 2 to comply to the format of the significand [1.f] in the previous slide

- $6.5/2 = 3.25$
- $6.5/4 = 1.625$

$$\Rightarrow 6.5 = (-1)^0 * 2^2 * 1.625 \Rightarrow s = 0, \text{exp} - \text{bias} = 2 \Rightarrow \text{exp} = 129 = 0x81$$

$$\Rightarrow 0.625 = \frac{1}{2} + \frac{1}{8} \Rightarrow f = 101$$

$$\text{Result} = 0b0|100\ 0000\ 1|101\ 0000\ 0000\ 0000\ 0000\ 0000 = 0x40D00000$$

# Example from Hohl Textbook

**Describe the single-precision representation of -0.4375:**





# Example from Hohl Textbook

## **Describe the single-precision representation of -0.4375 :**

S=1 (Negative number)

We need to find a representation of 0.4375 between 1 and 2 to comply to the format of the significand [1.f] in the previous slide

- $0.4375/2^{-1}=0.875$
- $0.4375/2^{-2}=1.75 \quad \Rightarrow \text{exp-bias}=-2$

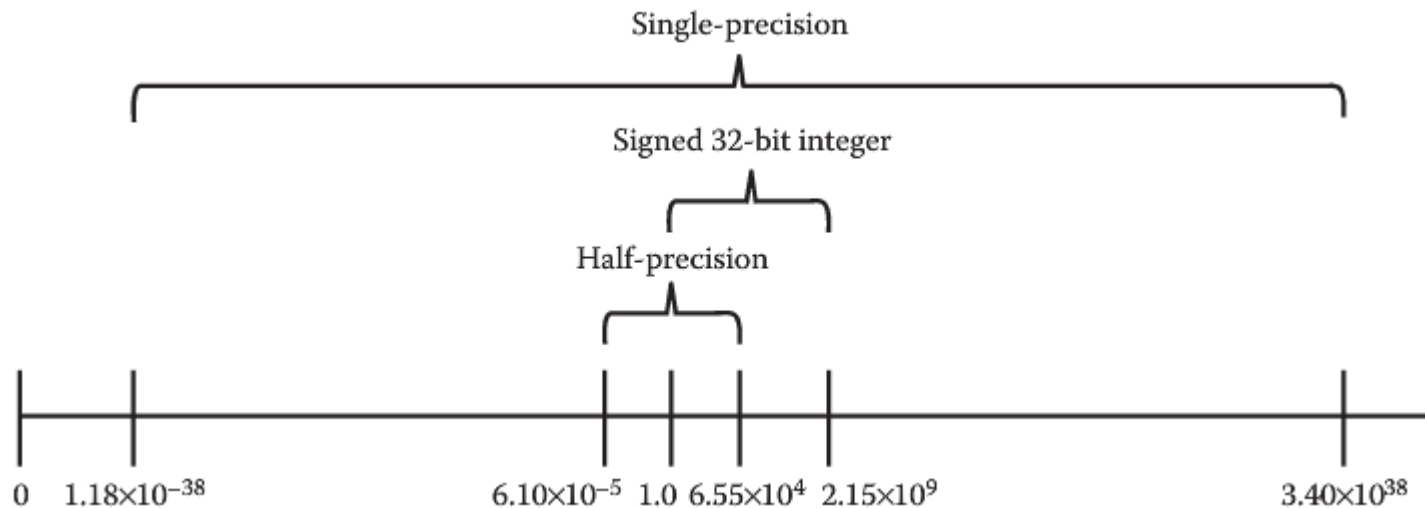
$$\Rightarrow -0.4375 = (-1)^1 * 2^{-2} * 1.75 \Rightarrow s = 1, \text{exp} - \text{bias} = -2 \Rightarrow \text{exp} = 125 = 0x7d$$

$$\Rightarrow 0.75 = \frac{1}{2} + \frac{1}{4} \Rightarrow f = 11$$

Result=0b1|011 1110 1|110 0000 0000 0000 0000 = 0xBEE00000

# Range of FP Numbers

- Single-precision Floating point numbers cover a much wider range comparing with integers, but they are represented with the same number of bits (32)
- This means that they can be *less accurate* for some integer numbers: for example, there can be a higher distance between one element and the following element, leading to possible ERRORS in the REPRESENTATION

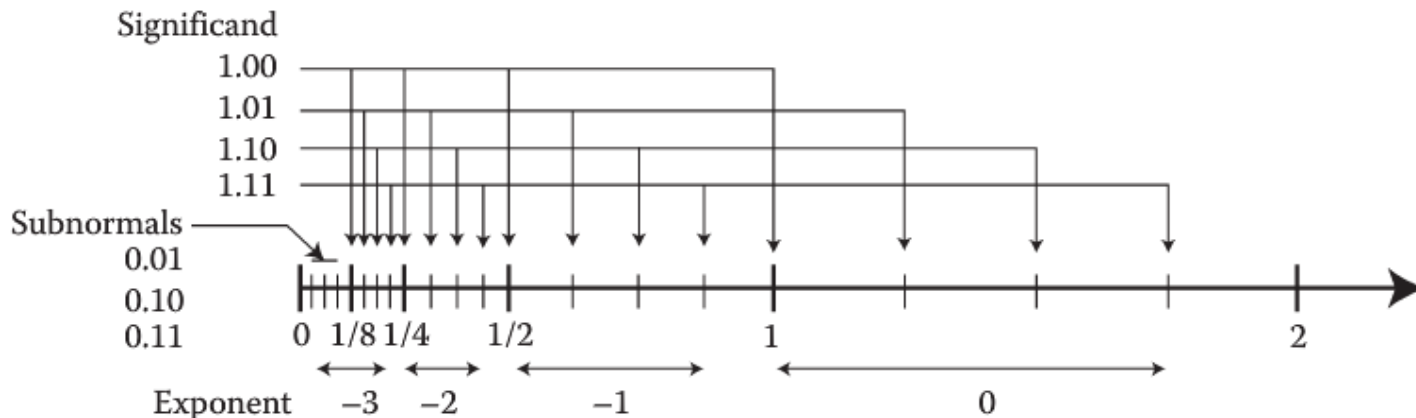


# Accuracy of Floating Point Numbers

- One important thing to notice is that the FP representation imposes limitation on the “*Precision*” of FP numbers

$$F = (-1)^s * 2^{exp-bias} * 1.f$$

- The precision of a binary representation is the distance between two successive numbers:** If the number is defined by the product of its binary representation times an exponent, then the distance between two consecutive FP Numbers depends on their exponent:
  - The larger the exponent, the larger would be the distance between two consecutive representations, so the smaller would be its precision



# De-normal / Subnormal Numbers

- A NORMAL FP encoding is an encoding whose significand is assumed to have a 1 value in front, such as the case we just introduced.
- The smallest positive number available with FP representation is

$$F = 2^{-126} * 1.0 = 1.18 * 10^{-38}$$

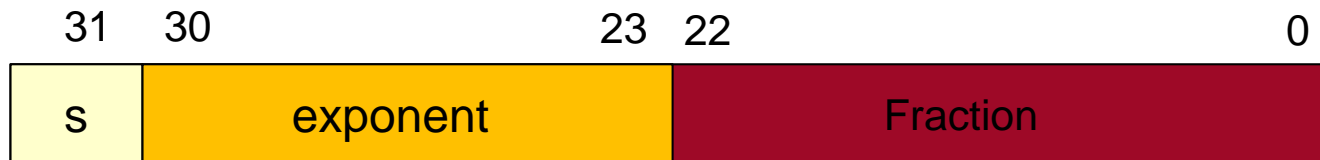
If we want to represent even smaller numbers we can use De-Normalized numbers (a.k.a. Sub-Normal), that are defined by the formula

$$F = (-1)^s * 2^{-126} * 0.f$$

In this case we can add  $2^{23}$  more numbers to the list of representable floats. Sub-normal numbers are indicated by an unbiased exponent of zero, that is all numbers with  $\text{exp}=0$  are considered de-normalized.

# Zero and Infinity

- FP representation includes two zero (+/-0) and two infinity values (+/- $\infty$ )  
Infinity is considered as a mathematical concept, and NOT as the maximum representable value!
- 0 is represented as a number with exponent of 0 and significand of 0
- $\infty$  is represented as a number with an exponent of all 1s, and significand of 0



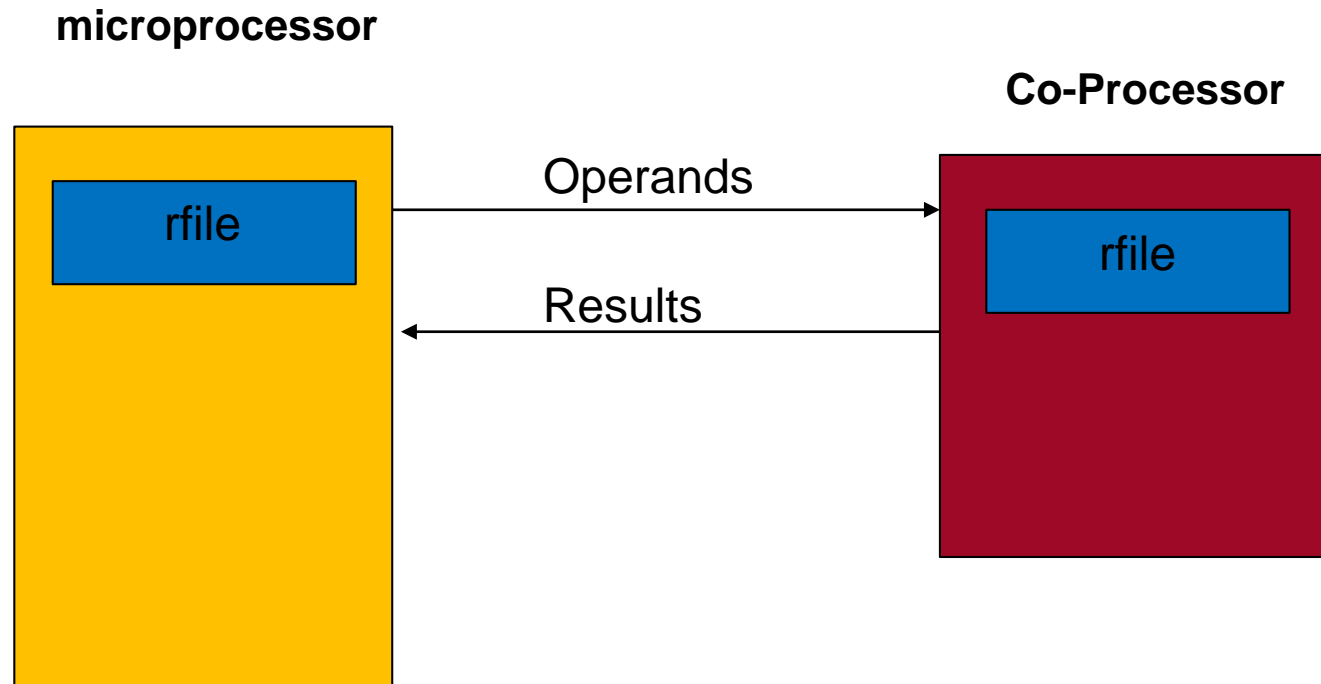
# Not a Number (NaN)

- NaN can be used to describe different configurations:
  - Result in the presence of unexpected conditions during computation (in this case the condition is specified in the significand – we call this a “*signalling*” NaN )
  - Default value for registers not initialized (“*quiet*” NaN)
- IEEE 754-2008 imposes that the result of any operation involving a NaN is a NaN
- NaN are encoded with exponent of all ‘1’s and non zero significand

# Implementation of FP Operations in ARM

- ARM7TDMI has no Hardware FP support. FP calculations are computed using software emulation and FP numbers are mapped over integer registers
- Recent architectures from ARM such as the later Cortex models allow the use of a Hardware FP calculation
- ***FP operations (Sum, Subtraction, Multiplication, Division) tend to be very demanding from a hardware point of view, so they COULD NOT POSSIBLY FIT in the processor pipeline***
- For this reason, historically, in most processor architectures, FP calculations are performed by a CO-PROCESSOR

# Co-Processor Computation

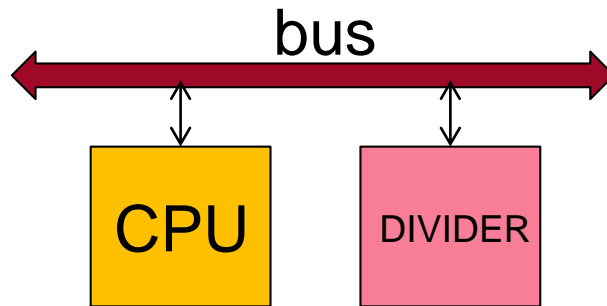


- Performing FP operation IN the main processor pipeline will require long operations, that reserve for several cycles processor registers and disrupt the program flow
- It is instead convenient to reserve a specific co-processor for FP operations, with an independent register file.



# Coprocessor vs Memory mapped peripheral

- There are two ways to map a computation peripheral on to a processor architecture: *Memory Mapped Peripheral* or *Co-Processor*
- *Ex: Suppose we want to use a Hardware divider [note an integer divider is almost never present in Embedded processors, while a FP divider is much more common]:*
  - *We can map the divider as a peripheral on the BUS:*



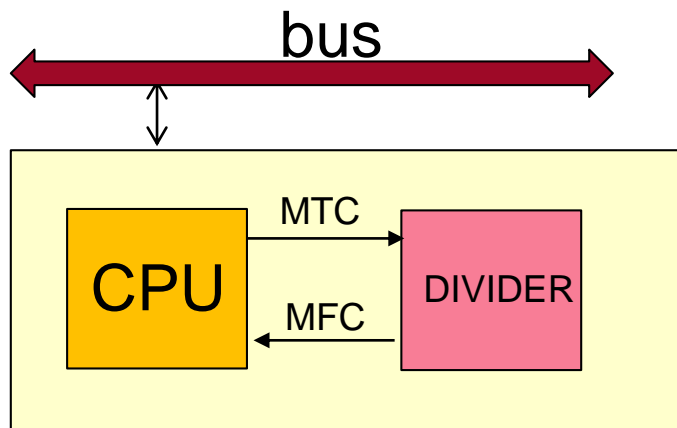
$z=x/y;$

```
LDR R4,=base_addr_divider
STR  R0, [r4]
STR  R1, [r4,#4]
[....wait necessary time ....]
LDR  R2,[r4,#8]
```

- Memory mapped peripherals are *loosely coupled* to the CPU. They have NO IMPACT on the CPU architecture and ISA, that sees the divider as a part of the memory. But memory accesses can be demanding in terms of cycle time

## Coprocessor vs Memory mapped peripheral (2)

- Alternatively, we can map the divider as co-processor. ***In this configuration, the coprocessor is a separate datapath, with independent pipeline***, but there are specific instructions in the ISA moving data back and forth from the coprocessor and starting computations. The co-processor is *tightly coupled* to the CPU



$$z=x/y;$$

```
MTC CR0, r4
MTC CR1, r5
COP.div CR2,CR0,CR1
[....wait necessary time ....]
MFC CR2,r2
```

MTC=Move to Coprocessor

MFC=Move from Coprocessor

- In this case, it is customary for the coprocessor to have an internal register file to store temporary values internally minimizing transfers to/from the main CPU

# Floating Points Units

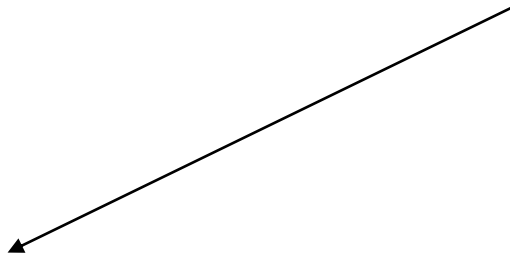
- Many processor architectures (e.g. MIPS) use a Floating Point coprocessor
- The case of the ARM Cortex, its Floating Point Unit is very similar: Floating point operations (where available) are deployed on a separate data path that is tightly coupled to the main CPU
- The FPU is implemented as a coprocessor: the FPU has an independent Register file composed of 32 registers of 32 bit each.
  - 1 FP register stores one single precision FP number
  - 2 FP registers can be used to represent a double precision FP number
  - The upper/lower 16 bits of a FP register can be used to store a half-precision number
- The ARM FPU also has two control FP registers: FPCSR (Floating Point Control and Status Register) and CPACR (CoProcessor Access Control Register)

# Cortex FP Registers

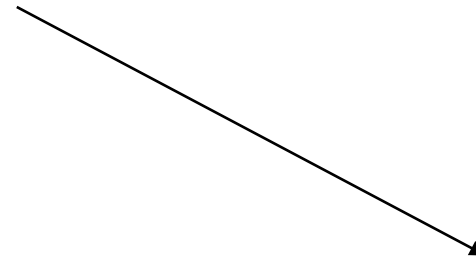
- The 32 GP registers are “flat”: there is no specific usage such as in the case of integer registers, they can all be used interchangeably. *The mnemonics s# (single precision) or d# (double precision) are used instead of r# to indicate floating point registers*
- FPCSR (Floating Point Status and Control Register) is the equivalent of the integer CPSR, and stores operation information:
- CPACR (Coprocessor Access Control Register) holds information about access permission to the available coprocessor slots (0-15 in Cortex). The FPU is at slot 10 (Single precision) or 11 (Double precision)
  - Note: Physically the FPU is one coprocessor, but since the slot number is added to any coprocessor instruction, the different slot number is used to specify to the FPU the type of operation expected

# Loading FP Registers

- Floating Point numbers can be loaded on the FPU registers in two ways



Directly from memory, where they are stored in FP format

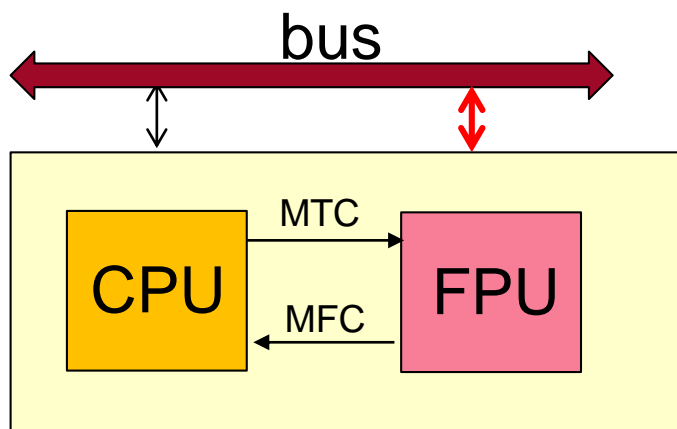


From an ARM GP register. In this case, the FPU will automatically convert them from the integer to FP format (possibly losing information)

# Loading/Storing data to/from the FPU

VLDR / VSTR.32, S#, [addressing Mode]

VLDR / VSTR.64, D#, [addressing Mode]



- Compared to most other processor architectures, ARM is peculiar because it allows a coprocessor to load data directly from memory.
- Most processors load data from memory to a GP register, and then from GP register to FP registers
- Note that the registers used to address memory are in the CPU and \*NOT\* in the FPU !!!

# Moving FP data between GP and FP Registers

- Data transfers between GP and FP registers are implemented by the move to/from coprocessor instructions

VMOV.f32 S#, R#

VMOV.f32 R#, S#

VMOV.f32 S#, S#

VMOV.f32 S#, immed

- NOTE: VMOV, VLDR, VSTR as well as all other floating point operations are part of the ARM ISA, so they support all conditional execution suffixes

# Double Precision FP move

- In order to support double precision arithmetic, we can transfer TWO registers at the same time. Note: The GP registers can be independent, but the FP registers must be consecutive

VMOV S#,S#,R#,R#

VMOV R#,R#,S#,S#

VMOV D#, R#,R#

VMOV R#,R#,D#

- Note that the last two are not independent operation, but a different way (aliasing) of writing the same operation: they correspond to the same machine code



# Floating Point Processing Instructions

- All FP Instructions have a similar format:

V<operation>{cond}.F32 dest src1 src2

The most important operations supported by the ARM FPU are

ABS / NEG / ADD / SUB / MUL / MLA / MLS / CMP / DIV / SQRT

# Format Conversion Instructions

- An additional set of instructions is used to convert numbers between integer and floating point or between different floating point precision formats
- Note that normally floats will remain in the same format all through computation, so these instruction would normally not be utilized, UNLESS there is a specific type casting in the c code

```
int a;  
float b;  
main()  
{ b=(float) a;}
```

Instruction for format conversion:

VCVTB, VCVTT, VCVT

# Tutorial 1: Disassembling a simple FPU Code

C Code:

*float a,b=2.3,c=3.4;*

*main()*

*{a=b+c;}*

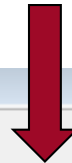


## Cortex M4 Version

Disassembly				
3: {a=b+c;}				
0x000003B0	4806	LDR	r0, [pc, #24]	; @0x000003CC
0x000003B2	ED900A00	VLDR	s0, [r0, #0x00]	
0x000003B6	4806	LDR	r0, [pc, #24]	; @0x000003D0
0x000003B8	EDD00A00	VLDR	s1, [r0, #0x00]	
0x000003BC	EE300A20	VADD.F32	s0, s0, s1	
0x000003C0	4804	LDR	r0, [pc, #16]	; @0x000003D4
0x000003C2	ED800A00	VSTR	s0, [r0, #0x00]	

Two's complement representation of the input floats

Memory 1	
Address:	0x10000000
0x10000000:	00000000 40133333 4059999A 00000000 00000000 00000000 00000000
0x1000001C:	00000000 00000000 00000000 00000000 00000000 00000000 00000000



## ARM7TDMI Version

Disassembly				
3: {a=c+b;}				
0x000001C4	E92D4010	STMDB	R13!, {R4, R14}	
0x000001C8	E59F0020	LDR	R0, [PC, #0x0020]	
0x000001CC	E5901000	LDR	R1, [R0]	
0x000001D0	E59F001C	LDR	R0, [PC, #0x001C]	
0x000001D4	E5900000	LDR	R0, [R0]	
0x000001D8	EB000007	BL	__aeabi_fadd(0x000001FC)	
0x000001DC	E59F1014	LDR	R1, [PC, #0x0014]	
0x000001E0	E5810000	STR	R0, [R1]	

Emulation  
Function

# Tutorial 1: Profiling Information

Two cycles per instruction

0.167 us	0x000003B0	4806	LDR	r0, [pc, #24] ; @0x000003CC
0.167 us	0x000003B2	ED900A00	VLDR	s0, [r0, #0x00]
0.167 us	0x000003B6	4806	LDR	r0, [pc, #24] ; @0x000003D0
0.167 us	0x000003B8	EDD00A00	VLDR	s1, [r0, #0x00]
0.083 us	0x000003BC	EE300A20	VADD.F32	s0, s0, s1
0.167 us	0x000003C0	4804	LDR	r0, [pc, #16] ; @0x000003D4
0.167 us	0x000003C2	ED800A00	VSTR	s0, [r0, #0x00]

Single cycle

ARM Cortex M4 (Using FPU): Total Time 13 cycles


# Tutorial 2: Disassembling an FP Division

C Code:

```
float a,b=2.3,c=3.4;
```

```
main()
```

```
{a=c/b;}
```



0.167 us	0x000003B0	4806	LDR	r0, [pc, #24] ; @0x000003CC
0.167 us	0x000003B2	ED900A00	VLDR	s0, [r0, #0x00]
0.167 us	0x000003B6	4806	LDR	r0, [pc, #24] ; @0x000003D0
0.167 us	0x000003B8	EDD00A00	VLDR	s1, [r0, #0x00]
1.167 us	0x000003BC	EE801A20	VDIV.F32	s2, s0, s1
0.167 us	0x000003C0	4804	LDR	r0, [pc, #16] ; @0x000003D4
0.167 us	0x000003C2	ED801A00	VSTR	s2, [r0, #0x00]

Profiling: Div=14 cycles, all other operations 2 cycles, total 26 cycles

ARM Cortex M4 (Using FPU): Total Time 26 cycles

Note: For your reference, a SP mul operation takes one cycle, a MLA 3 cycles