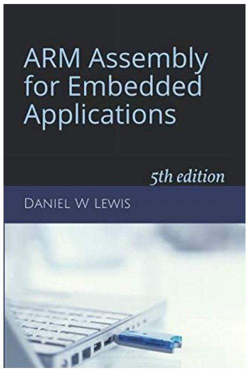


Programming Lab 11A

Arithmetic with Reals

Topics: Alternative representations of reals; floating-point hardware & emulation, Q16 fixed-point, posit emulation.

Prerequisite Reading: Chapters 1-11
Revised: June 7, 2021

⇩ Click to download Lab11A-Main.c

⇩ Click to download real-libs.zip

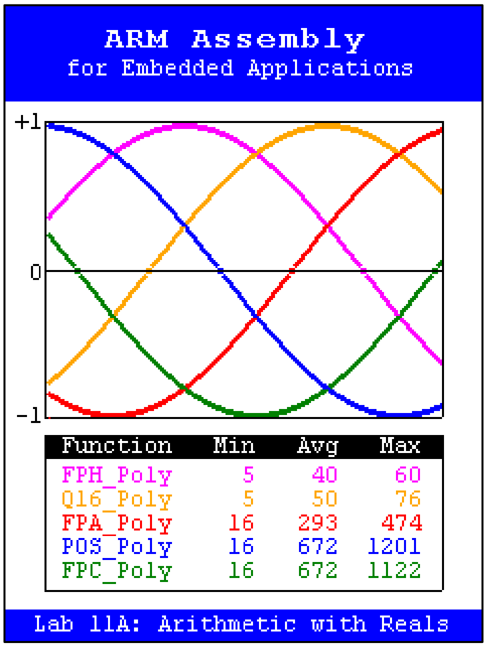
Background: Real numbers may be represented in floating-point[1], fixed-point[2], or posit[3] format. Arithmetic using these representations may be implemented in hardware, or in software written in either a high-level language or assembly. This lab explores the relative performance of these alternatives by evaluating Taylor series approximations of the sine function – i.e., polynomials with coefficient $(a_0, a_1, a_2, \cdots)$ chosen to produce sin(x):

$$poly(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_{n-1}x^{n-1}$$

The polynomial is most efficiently evaluated using Horner's method[4], working backwards from $a_{n-1}$to a_0:

$$poly(x) = (((0)x + a_{n-1})x + a_{n-2})x + \cdots + a_0$$

Assignment: The main program will compile and run without writing any assembly. However, your task is to create equivalent replacements in assembly language for the following five functions found in the C main program. The original C versions have been defined as "weak" so that the linker will automatically replace them in the executable image by those you create in assembly; you do not need to remove the C versions. This allows you to create and test your assembly language functions one at a time. The five function prototypes share a common format, but the ***function-name*** and ***data-type*** vary. *(Note: The code for each of the last three functions should be almost identical.)*



Function	Min	Avg	Max
FPH_Poly	5	40	60
Q16_Poly	5	50	76
FPA_Poly	16	293	474
POS_Poly	16	672	1201
FPC_Poly	16	672	1122

Lab 11A: Arithmetic with Reals

data-type ***function-name***`(`***data-type*** `x,` ***data-type*** `a[], int32_t n) ;`

function-name	***data-type***	***same as***	***Implement this function in assembly using …***
`FPH_Poly`	`float`	`float`	*floating-point addition & multiply instructions*
`Q16_Poly`	`Q16`	`int32_t`	*integer addition & multiply instructions*
`FPA_Poly`	`float32_t`	`int32_t`	*ASM library functions* `qfp_fadd` *&* `qfp_fmul`
`FPC_Poly`	`float32_t`	`int32_t`	*C library functions* `AddFloats` *&* `MulFloats`
`POS_Poly`	`posit32_t`	`int32_t`	*C library functions* `AddPosits` *&* `MulPosits`

Download the main program and `real-libs.zip`. Inside the `zip` are the ***library files*** `lib1-float.s`, `lib2-float.c`, and `lib3-posit.c` to be extracted into your `src` directory together with the C main program. The program calls each of your polynomial functions with an array of n coefficients chosen to approximate the *sine* function at an angle x expressed in radians. Since the approximation of the sine function requires fewer terms near $x = 0$ for the same accuracy, the main program varies n from 0 to 8 to reduce average execution time. The values returned by your polynomial functions are used to display moving sine waves. If your code is correct, the display should look like the image above although with possibly different cycle counts. Error messages (if any) will appear as white text on a red background.

[1] https://en.wikipedia.org/wiki/Floating-point_arithmetic
[2] https://en.wikipedia.org/wiki/Fixed-point_arithmetic
[3] https://en.wikipedia.org/wiki/Unum_(number_format)
[4] https://en.wikipedia.org/wiki/Horner%27s_method