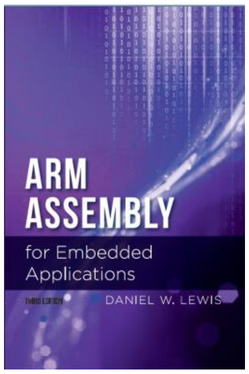


Programming Lab #1

Binary Number Systems

Prerequisite Reading: Chapters 1-2 & Appendix E
Revised: October 10, 2017

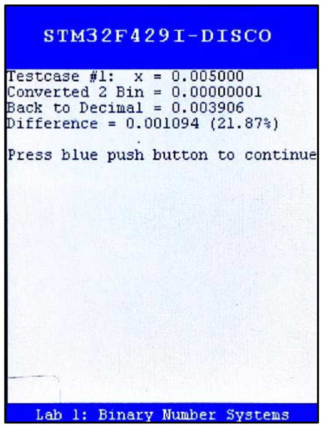
Part 1:

Download and unzip the ZIP file containing the sample program of Listing E-1 on page 290 (and all the additional files required) from here. Build and test the program following the instructions in Appendix E.

Part 2:

1. Download the C main program for Lab1 from here.
2. Edit the source code to complete the two functions Dec2Bin and Bin2Dec. Do not modify their parameter list or return values, or any other part of the program. Note that Dec2Bin should produce a rounded result.
3. Compile your program using the EmBitz IDE. (Use the Lab1Main.c file to replace the main.c file of Part 1 and recompile.)
4. Download your program to the ARM board and test your code. If your code is correct, your results should match the values below.

Testcase	x	binary	decimal	difference	percent
1	0.005000	0.00000001	0.003906	0.001094	21.87%
2	0.010000	0.00000011	0.011719	0.001719	17.19%
3	0.050000	0.00001101	0.050781	0.000781	1.56%
4	0.100000	0.00011010	0.101562	0.001562	1.56%
5	0.300000	0.01001101	0.300781	0.000781	0.26%
6	0.700000	0.10110011	0.699219	0.000781	0.11%
7	0.900000	0.11100110	0.898438	0.001562	0.17%



```
STM32F429I-DISCO

Testcase #1:  x = 0.005000
Converted 2 Bin = 0.00000001
Back to Decimal = 0.003906
Difference = 0.001094 (21.87%)

Press blue push button to continue

Lab 1: Binary Number Systems
```

You may be tempted to use the C library `pow()` function. While using the `pow` function can certainly work, there is a better way to do it.

Whenever you can avoid calling a library function your code will occupy less memory (use fewer instructions) and usually run faster – both of which are important in embedded applications because you should always be trying to squeeze the most performance you can out of a typically inexpensive processor.

But it's much worse! It's not just one function call to pow, it's actually becomes four function calls! The input parameters and the return value of the pow function must be of type double. But when used to convert number representations, the input parameters and return value are of type int. You can certainly give the pow function integer parameters, and you can certainly store the value that it returns into an integer variable, but doing so will cause the compiler to call library functions that convert the integer input parameters to doubles before calling pow, and will call another library function to convert the return value from a double to an int.

For example, this line of C code where all the variables are ints:

```
x2n = pow(x, n) ;
```

causes the compiler to produce the following sequence of instructions:

```
080001C8 ldr r0, [r5, #0]
080001CA bl 0x8004ae4 <__floatsidf>
080001CE strd r0, r1, [sp]
080001D2 ldr r0, [r4, #0]
080001D4 bl 0x8004ae4 <__floatsidf>
080001D8 vldr d1, [sp]
080001DC vmov d0, r0, r1
080001E0 bl 0x8000218 <pow>
080001E4 vmov r0, r1, d0
080001E8 bl 0x8005110 <__fixdfsi>
080001EC ldr r2, [r5, #0]
080001EE str r0, [r6, #0]
```

Note the calls to `__floatsidf` to convert the parameters from int to float, and the call to `__fixdfsi` to convert the float result to an int.

So here's a better solution:

Consider an 8-bit binary number, represented as $b_7b_6b_5b_4b_3b_2b_1b_0$, where the b's are the 1's and 0's of the number. The corresponding polynomial would be:

$$polynomial = 2^7b_7 + 2^6b_6 + 2^5b_5 + 2^4b_4 + 2^3b_3 + 2^2b_2 + 2^1b_1 + 2^0b_0$$

But note that you can rewrite this as:

$$polynomial = b_0 + 2(b_1 + 2(b_2 + 2(b_3 + 2(b_4 + 2(b_5 + 2(b_6 + 2b_7))))))$$

Which can be computed using a simple loop:

```
$polynomial \leftarrow 0$
$for\ i = 7\ down\ to\ 0: polynomial \leftarrow 2 \times polynomial + b_i$
```