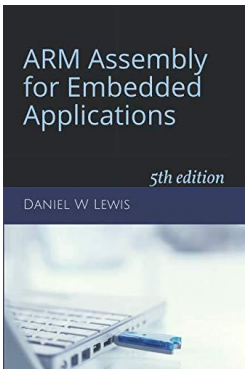


Programming Lab #1

Binary Number Systems

Prerequisite Reading: Chapters 1-2 & Appendix B
Revised: April 12, 2019

PART 1:

1. Download and unzip the ZIP file containing the workspace and sample program from here. Build and test the sample program that's already in the **src** subdirectory of the workspace as follows:
2. **Windows Only:** Create a shortcut on your desktop to the file named **setup.bat** that is located in your workspace folder. Double-click on the shortcut. This will open a command line window with the root of the workspace folder as the current directory.
3. **Linux and OS X Only:** Open a terminal window and change the current directory to the root of the workspace directory.
4. Enter the command "**make**". This will run the compiler, the assembler and the linker, producing the final program in a file named **output.bin**.
5. Connect the single board computer to the desktop computer using a mini-USB cable. A window will open as if you had inserted a USB thumb drive. Into that window, copy the file **output.bin** from the root directory of your workspace. The board will automatically load and execute your program.

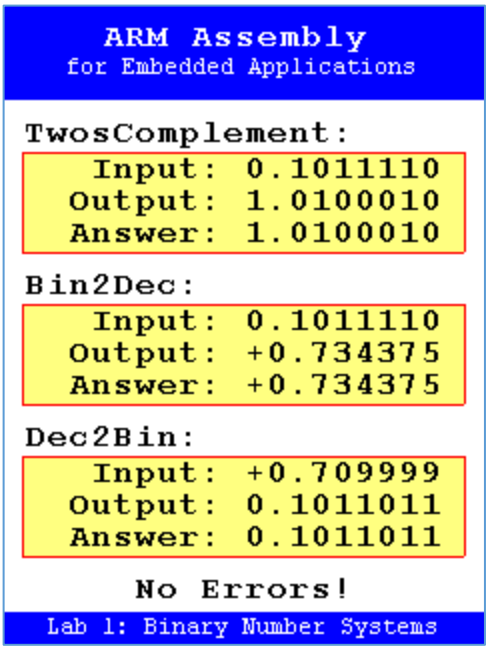
PART 2:

1. Delete any existing files in the **src** and **obj** subdirectories of your workspace folder.
2. Download the C main program for Lab1 from here and store it in the **src** subdirectory of your workspace folder.
3. Use your favorite text editor (not a word processor) to create a second C source code file in the **src** subdirectory that implements the three functions shown below. Do not use filenames containing spaces or filename extensions with uppercase letters. Each array parameter holds an 8-bit binary number $b_7.b_6b_5b_4b_3b_2b_1b_0$ in the range $-1.0 \le x < +1.0$, where (for example) bin[7] = b_7 and bin[0] = b_0. Note that `Dec2Bin` should produce a rounded result.

```
void  TwosComplement(const int input[8], int output[8]) ;
float Bin2Dec(const int bin[8]) ;
void  Dec2Bin(const float x, int bin[8]) ;
```

4. Repeat steps 2-5 of Part 1 to build and download your program to the board.

RUNNING THE PROGRAM:



When the program runs, it will cycle through a long sequence of tests, displaying various input and output values for each of your functions, and what the correct answer should be. If there is an error in one of your functions, the program will pause, display your output in white text on a red background, and wait for you to press the blue pushbutton to proceed. If no errors occur after going through all the tests, the program will display the message "No Errors!" at the bottom of the screen.

HINTS:

You may be tempted to use the C library `pow()` function in `Bin2Dec` to compute the terms of a polynomial. Although that certainly works, 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 actually becomes four function calls! The `pow` function requires its input parameters and the return value to be of type double. But when converting number representations, you need to compute the integer 2^k, where `2` and `k` are also integers. 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 requires the compiler to call library functions to convert back and forth between floats and ints. For example, the assignment statement "`x2n = pow(x, n);`" effectively becomes:

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

A BETTER WAY TO CALCULATE THE POLYNOMIAL:

Consider an 8-bit binary signed integer, 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:

$$polynomial \leftarrow -b_7$$
$$for\ i = 6\ down\ to\ 0: polynomial \leftarrow 2 \times polynomial + b_i$$

Function `Bin2Dec`: The polynomial gives you an integer in the range -128 to +127; divide it by 128 to get the floating-point value to return. **Function `Dec2Bin`**: Start by multiplying the floating-point input parameter by 128 and then convert the resulting integer into an array of bits.