Lab #6 – Conditional Execution

Goals:

Write programs on the Discovery Board forming loops and conditional structures
Further develop the use of the Discovery Board graphics display
Learn the .equ assembler directive

Reference: 2^0=1 2^1=2 2^2=4 2^3=8 2^4=16 2^5=32 2^6=64 2^7=128 2^8=256 2^9=512 2^{10}=1024 2^{11}=2048 2^{12}=4096 2^{13}=8192 2^{14}=16384 2^{15}=32768 2^{16}=65536 …

For each step enter the necessary responses into the Canvas assignment text entry tool to let the TA follow your work.

Remember the 'SVC #0' debugging instruction available with your Discover Board library! That can be inserted in your assembly code any time you need to clarify exactly what is in the registers!

Refer to the Weeks 5 and 6 lecture slides for the following:

Part 1: Download the template main C program Lab06-DrawMain.c and the template assembly source code Lab06-draw.s from the Canvas files area, and in the LabCode folder. Note that the main program simply calls a function.

In Lab 5 you learned two useful assembler directives, `.rept` and `.endr`, to delineate a block of assembly instructions to be repeated in the source code. In this lab let's learn about another very useful directive, `.equ`. This directive stands for "equate", and simply defines a symbol to be equal to some constant value. Look at the top lines of Lab06-draw.s and you will see a block of lines in the form

```
.equ COLOR_BLUE,    0xFF0000FF
...
.equ COLOR_ORANGE,  0xFFFFA500
.equ DISPLAY,       0xD0000000
```

These lines simply define constant values to be equated to various symbolic names. You'll recognize the first several lines as defining a 32-bit hex value that corresponds to a given color pixel on the Discovery Board display. After these definitions are processed by the assembler, you may simply refer to the easy-to-remember symbolic color name, for example, `COLOR_BLUE`, in your code without having to look up exactly what 8-digit hex value is needed. For example, you could load register R0 with the pixel value for a green pixel with the pseudo-instruction

```
LDR  R0, =COLOR_GREEN
```

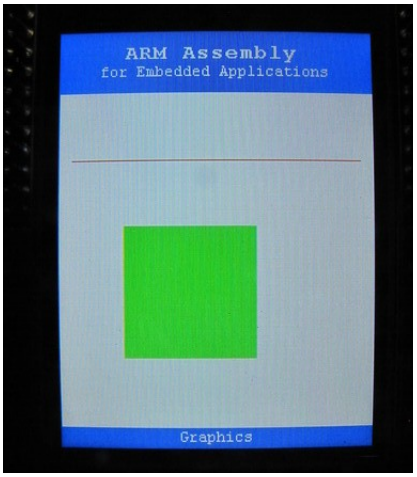
Similarly, you can make use of the very last .equ equate line to load R1, for example, with the base address of the display memory array:

```
LDR  R1, =DISPLAY
```

without having to remember exactly what numerical address is needed.

a) Write assembly code in the function to draw a horizontal line across the display of a color of your choosing. Draw it on row 100, from columns 10 through 229, inclusive. (Since this is in the white

background region of the display, obviously choosing a drawing color other than white would be a good idea!) Write your code to include a loop over the column number from 10 to 229 using the conditional instructions on the processor. You can follow the example loop coding shown on lecture slide 13 if you like. Inside the loop compute the pixel index from the row and column indices, and then use the `[Rn, Rm, LSL #<n>]` addressing mode in an STR instruction to store your chosen pixel word to the display memory at the proper address. Your line should look like the red line in the photo below, except in whatever color you choose. (Ignore the filled color rectangular area, as we will get to that in Lab #7 once we cover how to use the push-pop stack.) Write your code to increment the column coordinate of the pixel to be written as a programmed loop using conditional assembly code. No `.rept` and `.endr` repeat blocks allowed in this lab!



b) Now add to your function the code for another loop that will draw a vertical line of a different color, from rows 50 through 299, inclusive, in column 25. Upload your code and a cell phone photo of your display screen showing the two drawn lines. Again, write your code to increment the row coordinate of the pixel to be written as a programmed loop using conditional assembly code. No `.rept` and `.endr` repeat blocks allowed in this lab!

Part 2: Download and edit the template source code files Lab06-ConvertMain.c and Lab06-convert.s. Your programming task is to complete the two assembly functions so that they convert an integer in the range of 0-15 to a hexadecimal ASCII character suitable for printing, and the reverse conversion, from an ASCII character to an integer, for an input routine. Note that the main C program will call both conversion functions over their range of input integers or characters.

a) Use the coding outline shown in the lecture slide 22 for the conversion from ASCII to integer, using an efficient If-Then block. There should be both ‘then’ and ‘else’ conditions in the IT block to implement two different SUB instructions. Then complete the integer to ASCII function in the same way but with ADD instructions.

Run your code and verify that when called by the main program the two functions return the correct conversions. Upload your code to Canvas.

b) Now rewrite the code for the two functions so that the If-Then block in each function has only a single ‘then’ condition. This involves using one *unconditional* ADD or SUB, and an additional ADD or SUB according to a conditional test.

Run your code and verify that when called by the main program the two functions return the correct conversions. Upload your code to Canvas.