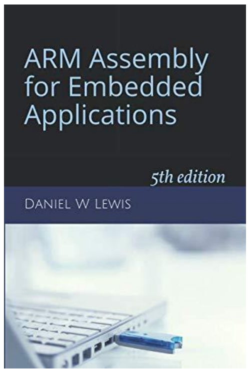


Programming Lab 12B

Image Processing

Click to download Lab12B-Main.c

Topics: SIMD processing, saturating arithmetic, composite data types and inline code.

Prerequisite Reading: Chapters 1-12 (and 14 for part 2)
Revised: January 20, 2021

Background: Digital images are comprised of millions of tiny dots (pixels), each represented by a minimum of three bytes of data – one each for the red, green and blue (RGB) components of its color. Processing such images therefore involves the manipulation of millions of bytes of data, requiring image processing software to use any means necessary to reduce execution time. This is particularly true for video, where processing must keep up with the refresh rate of the display. This lab will explore the use of SIMD instructions as a means of accelerating processing of the RGB color information of a single static image.

Assignment - Part 1: 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 two 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.

```
void SIMD_USatAdd(uint8_t bytes[], uint32_t count, uint8_t amount) ;
void SIMD_USatSub(uint8_t bytes[], uint32_t count, uint8_t amount) ;
```

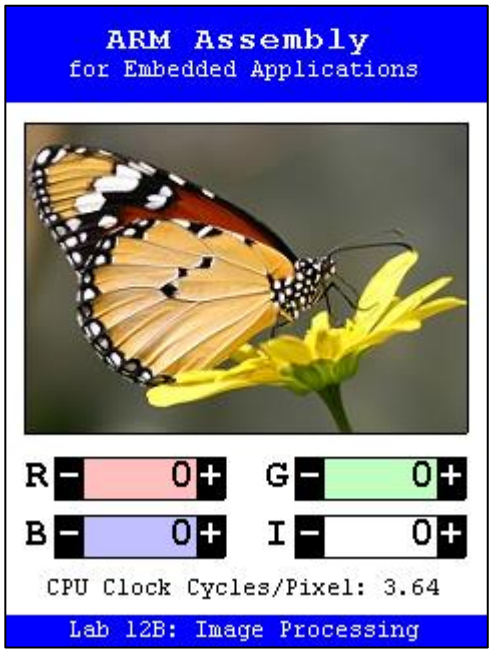
The two functions are almost identical except that one adds a constant to every byte in an array, and the other subtracts a constant from every byte. The addition and subtraction operations are used to increase or decrease the 8-bit unsigned red, green and blue intensity components of the pixels in one row of an image, so unsigned saturating addition and subtraction must be used to limit each result to the range 0-255.

These two functions are similar to function `SIMD_USatAdd` found in Chapter 12 of the text. However, the loop in that version only processes 4 bytes per iteration while each row of our image contains more than 600 bytes. Every iteration of the loop must check how many bytes remain to be processed and branch from the bottom to the top of the loop. To reduce this overhead and improve performance, design your functions to process 40 bytes per iteration instead of four, thus reducing the number of iterations by an order of magnitude. The total number of bytes in each row is only guaranteed to be a multiple of four, so you will need a "cleanup" loop to process up to 36 bytes of additional RGB data not processed by the main loop.

Processing *more* than 40 bytes per iteration decreases the time per pixel in the main loop, but it also increases the (shorter) execution time of the cleanup loop. Although it's not usually a significant issue, note that increasing the number of bytes processed in one iteration of the main loop inherently limits the minimum number of bytes per row and thus the minimum width of the image.

Assignment - Part 2 (Optional): Rewrite the C functions `Saturate`, `Between`, `AssemblePixel`, `UnpackRGB` and `PackPXL` that appear at the beginning of the main program as inline functions encapsulating inline assembly. Use IT blocks in functions Saturate and Between.

Test your code using the main program. Press the reset (black) pushbutton to start the program. Touch the "`+`" or "`-`" characters to increase or decrease the red (**`R`**), green (**`G`**), blue (**`B`**) components by the amount displayed. The intensity (**`I`**) value is added to all three of the RGB adjustments when the image is displayed. Touch the image itself to reset all values to zero. Press the blue pushbutton to invert the image.


ARM Assembly
for Embedded Applications
R - 0 +
G - 0 +
B - 0 +
I - 0 +
CPU Clock Cycles/Pixel: 3.64
Lab 12B: Image Processing