

Assignment

Introduction

The assignment is split into three tasks. In each the aim is to design and implement a program in ARM assembly which manipulates an image in a specified way. The image is stored as an array of word (32-bit) sized values, each element being a pixel of the image. The red, green and blue components of the pixel are stored in bytes 0,1 and 2 of the pixel respectively. Byte 3 is not used and is always 0.

A number of extra subroutines are used in order to allow the manipulation subroutine to work. In each of the programs, the start address of the pixel array is stored in R4.

Part 1- Brightness and contrast

Aim- design a program in ARM assembly language to modify the brightness and contrast of an image stored in memory.

Altering the brightness of a digital image is equivalent to changing the value of the colour channels of each pixel by a constant amount. In altering the contrast, brighter colours should be augmented and darker colours diminished, proportion to the desired change. This is achieved by multiplying each colour channel of each pixel by a constant amount. If the value is greater than one, the contrast increases, less than one and the contrast decreases.

By expressing the above mathematically, we can use the resulting equations in our program:

$$R' = \left(\frac{R * \alpha}{16} \right) + \beta$$

$$G' = \left(\frac{G * \alpha}{16} \right) + \beta$$

$$B' = \left(\frac{B * \alpha}{16} \right) + \beta$$

The first step I took was to determine the general algorithm the program would use, expressed below in pseudo-code:

For pixel in pixelArray:

 Load pixel's colour channels to individual registers (R,G,B)

 For each colour channel:

 colour=colour*contrast

 colour = colour/16

 colour += brightness

 if colour > 255:

 colour=255

 if colour < 0:

 colour = 0

 load modified colour channels back to pixel

Main program

```
28
29     MUL R7, R5,R6      ;R7 = N = W*H = NUMBER OF PIXELZ
30     LDR R9, =0         ;I=0, INITIALISE PIXEL INDEX
31
32
33     LDR R0, =16 ;contrast increase = 20
34     LDR R1, =10 ;brightness inc = 50
35
36
37 for
38     CMP R9, R7          ;if last pixel has been modified
39     BEQ endfor          ;stop for loop
40     BL changePixel      ;this method makes the required change of brightness and contrast for each pixel
41     ADD R9, R9, #1      ;increment the pixel index
42     B for
43 endfor
44
45     BL putPic           ; re-display the updated image
46
47 stop     B     stop
```

The above shows the main section of the program, which takes as inputs the brightness and contrast values (lines 33-34). The image width and height and their product are stored. Width * Height gives the total number of pixels, N, and this is used to construct a for loop which terminates when N iterations have occurred(37-43). R9 is set to 0 and represents the current pixel index.

Within the for loop, having the current pixel index in R9 and the image start address in R4, the program branches to the changePixel subroutine, which will apply the contrast and brightness changes to the pixel.

```
62 changePixel; (A(R4), I(R9), a(R0), b(R1) ), R3 = i
63     STMFD SP!, {R4-R8,LR}
64     ;STMFD SP!, {LR}
65     LDR R3, =0          ;i=0, the byte index
66 for_cp
67     LDR R8, =4
68     MUL R10,R9 ,R8       ;R10 = I*4
69     ADD R10,R10,R3       ;R10 = spare2 = I*4 + i
70
71     LDRB R11, [R4, R10] ;LOAD [A + I*4 + i] load primary colour byte (r/g/b)
72
73     MOV R10,R11
74     MUL R11, R10, R0      ;multiply colour value by contrast scale
75
```

The first section of change pixel saves the state of the registers before the branch to the stack, and then enters a for loop. The for loop will iterate three times, once for each colour channel in the pixel. Each iteration loads the next byte from the pixel, manipulates it appropriately, then stores it back at the correct address.

R3 is used as an index (i) to access the bytes of the pixel, as well as a count. It is incremented after each iteration. Line 71 implements the following pseudocode:

LOAD to R11 the BYTE at memory.[A + pixel_Index + byte_index]

This is achieved without changing the value in the start address register by ()mode (see lines 67-71).

The program now moves on to applying the equation s.

```

73     MOV R10,R11
74     MUL R11, R10, R0      ;multiply colour value by contrast scale
75
76     ;BL divide_by_16;(C(R11))    ;divide product by 16
77
78     MOV R10, R11          ;spare2 = C = coulour*16
79     LDR R11,=0            ;C = storage = 0
80 whileDiv16
81     CMP R10, #16
82     BLT endWhileDiv16
83     ADD R11, R11, #1      ;C++
84     SUB R10, R10, #16     ;spare2-16
85     B whileDiv16
86 endWhileDiv16
87     ;result of division is now in C, the colour storage = colour*a/16

```

First the contrast is altered. The loaded colour channel byte is multiplied by the contrast scale . Lines 78-86 divide that product by 16, using the standard division method we have been using in the course (repeated subtraction), leaving the quotient of the division in place of the colour byte.

```

88
89     ADD R11,R11,R1        ;C = STORAGE = C + BRIGHTNESS
90
91
92     CMP R11, #0           ;Ensure colour value is within
93     LDRLT R11, =0         ;range 0-255
94     CMP R11, #255
95     LDRGT R11, =255

```

Changing the brightness is a much simpler operation, we simply add the specified brightness value to the colour channel (89).

Since the the colour channels of the pixel are 8-bit values, they cannot be allowed to be outside the range 0-255. Lines 92-95 compare the value to the immediate values 0 and 255, and if it is greater the appropriate changes are applied.

```

97     LDR R10, = 4
98     MUL R8, R9, R10       ; R8 = I*4
99     ADD R8, R8, R3        ; R8 = I*4 + i
100    STRB R11, [R4, R8]    ;store modified colour back in memory
101
102    ADD R3, R3, #1        ;i++ //move on to next colour value
103    CMP R3,#3
104    BEQ endfor_cp
105    B for_cp
106 endfor_cp
107    ;LDMFD SP!, {PC}
108    STMFD SP!, {R4-R8,PC}

```

Lastly, the modified byte is stored back in its address using the same addressing method that was used to load the byte originally ($\text{address} = A + 4 \cdot I + i$). The loop count / byte index R3 is incremented and the loop branches back to its start, unless 3 bytes have already been modified (line 104) in which case the “for colour channel” loop breaks, and subsequently we branch back outside the subroutine to move on to the next pixel .

Testing and debugging section 1

The program did not work the first time I ran it, it returned the following image:



I followed the program's execution in the uvision debugger and found some misplaced labels and registers which were causing unending loops. After correcting the errors, I tested the program for various values of brightness and contrast:

Brightness	Contrast	Image
0	16	
20	32	
-10	10	

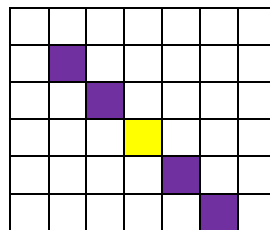
The program works as expected, leaving the image unchanged for contrast=16 and brightness =0. Higher values augment the brightness and contrast, lower values diminish them.

Part 2 – Motion Blur

Aim- design a program in ARM assembly language to apply a motion blur effect to an image stored in memory.

The blurring of any digital image always requires, for each pixel, some sampling of surrounding pixels. Motion blurring is blurring in a particular direction, so we want to sample the pixels in a line parallel to the direction of motion/ direction of blurring, and centred on the current pixel.

The intensity of the motion blur is determined by the length of this line (number of pixels sampled), which is called the radius of the blur. Visualising the blurring process with the pixels arranged in a grid, below is a diagram of the sampling which must take place in action:



The yellow pixel is the pixel that is currently being modified. Here, the blur is diagonal in direction, and has radius 5.

The value of the modified pixel will be the average of the sampled pixels. The assembly program will therefore carry out the sampling process for each pixel in the image, sum up each sampled value, and divide the total by the number of pixels sampled (note that this is not always the radius, eg: in the case where the pixel coordinates – radius < 0 or pixel coordinates >= image width/height).

For convenience in my program, I define the radius to be not the number of pixels sampled but, similar to the radius of a circle,

After understanding how the blur effect worked, the next step I took was to determine the general algorithm the program would use, expressed below in pseudo-code:

For pixel in image:

```
    get the pixel coordinates, X and Y //uses the current pixel index I
                                       //width and height to determine X,Y
    for(radius, radius >0, radius - -): //Samples the radius descending pixels
        if( !(X>=width or Y>=Height)):
            X++; Y++
            pixelsSampled ++ //counts pixels actually sampled
            storePixelToStack(X,Y) //function pushes the pixel with
                                   //coordinates X,Y to stack
    for(i=radius, i >0, i - -): //Samples the radius ascending pixels
        if( !(X<0 or Y<0)):
            X--;Y--
            pixelsSampled ++ //counts pixels actually sampled
            storePixelToStack(X,Y) //function pushes the pixel with
                                   //coordinates X,Y to stack

    byte R =0
    byte G =0
    byte B =0

    for pixels in stack: //amount to pop determined by the count
        pixelsSampled
        pop pixel from stack
        R+=pixel[R]
        G+=pixel[G]
        B+=pixel[B]
    R = R/pixelsSampled
    G = G/pixelsSampled
    B = B/pixelsSampled
    imagePixel[R] = R
    imagePixel[G] = G
    imagePixel[B] = B
```

```

22      ;R0 = spare3
23      LDR R1, =5      ;R1 = radius
24      MUL R2, R6, R5   ;R2 = N = number of pixels
25      LDR R3, =0      ;R3 = I: initialise
26
27      ;R7 = X
28      ;R8 = Y
29      ;R9 = I'
30      ;R10= n
31      ;R11= spare
32      ;R12= spare2
33
34      forPixelI
35
36      CMP R3,R2        ;COMPARE I and N
37      BEQ endForPixelI
38      BL pixelToRadiusSample; (A, I, W, H, radius)
39      ADD R3,R3,#1
40      B forPixelI
41      endForPixelI
42      BL putPic        ; re-display the updated image
43      stop B stop
44

```

The program begins with a main section, which initialises the appropriate registers, then enters a for loop, which iterates through each pixel in the image.

In the for loop, we branch to the pixelToRadiusSample subroutine, which does all the work in altering the pixel.

Before describing how pixelToRadiusSample works, I'll outline the subroutines it uses.

pixelCoordinates:

This subroutine takes a pixel index I, the width W, height H of the image, and returns the coordinates of the index, with (0,0) being the first pixel.

It calculates X and Y using the fact that the X coordinate of any index is the remainder of the division of the index by the image width, and Y is the quotient.

So all the subroutine has to do is perform this division (by repeated subtraction).

```

211      pixelCoordinates; (I, W, H) returns coordinates
212
213      STMFD SP!, {LR}
214
215      MOV R7, R3
216      LDR R8, =0
217
218      while
219          CMP R7, R6
220          BLT endwhile
221          ADD R8,R8,#1
222          SUB R7,R7,R6
223          B while
224      endwhile
225
226      LDMFD SP!, {PC}
227      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
228

```

CoordinatesToIndex:

This subroutine is the inverse of the above, taking the coordinates of a pixel and returning its index in the overall pixel array.

$$Index = X + Width * Y$$

```

229      coordinatesToIndex; (X, Y, W, H) returns index
230
231      STMFD SP!, {LR}
232
233      MUL R9, R6, R8
234      ADD R9, R9, R7
235      LDMFD SP!, {PC}
236

```

PixelToRadiusSample:

PixelToRadiusSample implements the pseudo-code shown above.

```
45  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
46  pixelToRadiusSample; (A,I,W,H,radius)
47      STMFD SP!, {LR}
48
49
50      BL pixelCoordinates; (I,W,H) returns
51      ;STORE CENTER PIXEL IN STACK
52      BL coordinatesToIndex
53      LDR R0, =4
54      MUL R12, R9, R0
55      LDR R0, [R4, R12]
56      STMFD SP!, {R0}
57
58      LDR R10, =1
59
60      LDR R11, =1
```

First, it calls pixelCoordinates to load the coordinates of the current pixel. These coordinates make navigating the image's pixel array and sampling different pixels much easier.

The current main pixel is loaded (lines 52-55) and pushed to the stack (because we always want to sample at least one pixel, ie: when effect intensity = 0) .

Next is an important section: storing the other pixels to the stack. The pixels we want to store are in a diagonal line from the current main pixel, but both above and below it.

This problem is split into two. One for loop pushes the pixels below (forDescendingPixels), and one pushes the pixels above (forAscendingPixels).

Since they both work the same way, I will explain only the descending pixels section.

The count of this for loop is the radius (which I define as the number of pixels above the center pixel to be sampled). In each iteration, the X,Y coordinates are simply incremented, and then represent the pixel diagonally down from the centre pixel.

```
62  ;store descending pixels to stack
63  forDescendingPixels
64      CMP R11, R1
65      BEQ endForDescendingPixels
66
67      ; INCREMENT X,Y
68      ADD R7, #1
69      ADD R8, #1
70
71      ;OUT OF BOUNDS TEST
72      CMP R7, R6
73      BGE outOfBounds
74      CMP R8, R5
75      BGE outOfBounds
76
77      ;LOAD PIXEL, STORE IT TO STACK, N++
78      ADD R10, R10, #1
79      BL coordinatesToIndex; (X,Y,W,H) returns 1
80
81      LDR R0, =4
82      MUL R12, R9, R0
83      LDR R0, [R4, R12]
84      STMFD SP!, {R0} ;pop word-sized pixel to
85
86  outOfBounds
87
88      ;COUNT++
89      ADD R11, R11, #1
90      B forDescendingPixels
91  endForDescendingPixels
```

Before storing a pixel, we must be sure that we don't try to sample a section of memory outside the image array. This is equivalent to X or Y being either less than zero, or greater than or

equal to the width and height (respectively). It is simple to test this (lines 72-75 only test for greater than width/height, because we are sampling the descending pixels).

If the pixel (X,Y) to be sampled is not actually within the array, the program branches past the storage section.

Otherwise, the pixels stored count (n) is incremented, the X,Y coordinates are used with coordinatesToIndex to get the actual pixel index I, which is then used to load the pixel from the array (lines 78-83). The 32-bit pixel is then simply pushed to the stack.

The loop count is then incremented and the for loop continues, until the count == radius.

The same process is repeated for the ascending pixels, after which we have the diagonal pixels stored on the stack, as well as a count of how many were stored.

```
125 ;PIXEL HAVE BEEN SAMPLED, CODE BELOW SUMS THEIR RGB BYTES, AVERAGES THEM, AND STORES MODIFIED PIXEL
126     LDR R11,=0 ;total R
127     LDR R12,=0 ;total G
128     LDR R0,=0 ;total B
129
130     LDR R9,=0
131
132 ;add stacked RGB vvalues to R, G, B
133 forStackedPixels
134     CMP R9, R10 ;while(COUNT<n) ie, stack hasn't been emptied of pixels
135     BEQ endForStackedPixels
136
137     LDRB R7, [SP]
138     ADD R11,R11,R7 ;add the red byte of the word in the stack to the red sum
139
140     LDRB R7, [SP,#1]! ;HERE, spare, spare2, spare3 are R,G,B values respectively
141     ADD R12,R12,R7 ;The store is the register which held X, R7
142
143     LDRB R7, [SP,#1]!
144     ADD R0,R0,R7 ;add the blue byte to the B register (R0)
145
146     ADD SP,SP,#2
147
148     ADD R9, R9,#1 ;COUNT++
149     B forStackedPixels
150 endForStackedPixels
```

With the correct pixels stored, the program next needs to sum them up and average them. However it cannot simply add the raw pixel values together, because the pixels are composed of colour channels, 3 8-bit values representing red, green and blue, which combine to produce the overall colour.

Therefore we must separate the bytes from each pixel, average the *bytes*, then recombine these into the modified pixel.

From lines 126-150, a for loop iterates through each pixel that has been stored in the stack (using R9 = n as the number of iterations), and uses the LDRB instruction to load the individual bytes from the stack. As each byte is loaded, it is added to one of three registers which have been initialised to 0, which store the sum of the R,G,B colour channels.

The stack pointer is incremented appropriately so that it points to the next word, and the for loop continues.

```

151
152     MOV R7, R11      ;move sum R to R7
153     BL divideColour; (R7, n); returns R/n in R9
154     MOV R11, R9
155
156     MOV R7, R12
157     BL divideColour; (R7, n); returns G/n in R9
158     MOV R12, R9
159
160     MOV R7, R0
161     BL divideColour; (R7, n); returns B/n in R9
162     MOV R0, R9
163
164     ;COLOUR CHECKER
165     CMP R0, #0        ;Ensure colour value is within
166     LDRLT R0, =0      ;range 0-255
167     CMP R0, #255
168     LDRGT R0, =255
169
170     CMP R11, #0       ;Ensure colour value is within
171     LDRLT R11, =0     ;range 0-255
172     CMP R11, #255
173     LDRGT R11, =255
174
175     CMP R12, #0       ;Ensure colour value is within
176     LDRLT R12, =0     ;range 0-255
177     CMP R12, #255
178     LDRGT R12, =255
179

```

Finally, the summed colour channels are divided by the number of pixels stored, using the divideColour subroutine, and if statements are employed to ensure the colour channel values are within 8 bits.

The R,G,B bytes are stored consecutively to memory, replacing the original pixel's colour channels with the modified values.

The subroutine then branches back to main, where the process is repeated for the next pixel in the array.

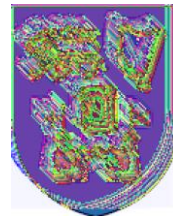
```

180
181     LDR R7, =4
182     MUL R9, R3, R7    ;I changed R12
183
184     ADD R9, R9, #1
185     STRB R0, [R4,R9]  ;STORE
186
187     ADD R9, R9, #1
188     STRB R12, [R4,R9]
189
190     ADD R9, R9, #1
191     STRB R11, [R4,R9]
192
193     ;end of subroutine
194
195     LDMFD SP!, {PC}
196     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;




```

Testing and debugging section 2

Debugging this program took quite long, there were many mistakes from registers in the wrong place to branching to the wrong labels. I know that it can take a long time to find one of these small errors, which is why I always try to test each individual section of the code and be as meticulous as possible (yet they still creep in). However at least some of the images produced were interesting!



I followed the program's execution in the uvision debugger to find the errors, looking out for unending loops. Once those had been corrected, the remaining problems were found by predicting what the values in the registers should be for each step, then testing to see if that was the case. The final program gave the following results:

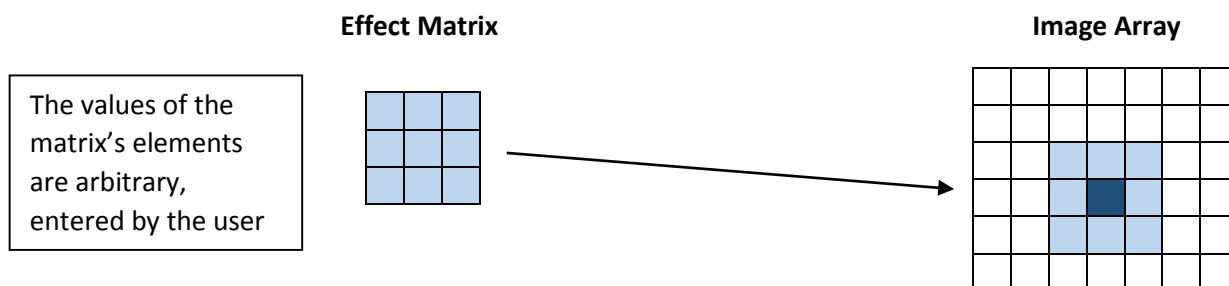
Radius	Image
0	
5	
10	

Part 3 – Bonus Effect

Aim- design a program in ARM assembly language to apply a chosen certain effect to an image stored in memory.

Many image effects can be represented by a convolution matrix. This is a square matrix whose elements are determined by the effect type. To modify a pixel, the convolution matrix is overlain on the image with the central element of the matrix over the current pixel. The modified pixel is generated by multiplying each element of the matrix by the corresponding image pixel, then taking the average of those values.

Visualising the process with the pixels arranged in a grid, below is a diagram of how the matrix is applied to a pixel:



The dark blue pixel on the right is the pixel that is currently being modified. The light blue values of the matrix on the left could be any integer.

Applying this multiplication method to each pixel of an image will give a certain effect, dependant on the elements of the effect matrix.

For example, the identity matrix is 0,0,0 ; 0,1,0 ; 0,0,0. It leaves the image unchanged (when division is turned off) because the sum carried out is $0+0+0+0+ (\text{centerPixel}) +0+0+0+0 = \text{centerPixel}$.

Implementing this in assembly language would create a versatile program which can apply many different effects with the same code, simply by changing the input matrix values.

While it is possible to do this with a 5-by-5 matrix or larger, I chose to use a 3-by-3 matrix in this program for simplicity.

Below is the pseudo-code I used for the program:

```
Int[] matrix=new int[]
For pixel in image:
    (X,Y) = Pixel[I].getCoordinates(I,W,H);
    X-
    Y++
    Col=0
    Row=0
    element=0
    n=0
    while(row<3):
        while(col<3):
            if(!(X<0 or Y<0 or X>=width or Y>=height)):
                n++
                stack.push( multiply(pixel,element))
            col++
            X++
        row++
        Y++
        X=X-3
    R=0,G=0,B=0
    for pixel in range(0,n):
        pix = stack.pop()
        R+=pix.R
        G+=pix.G
        B+=pix.B
    If(division):
        R=R/n
        G=G/n
        B=B/n
    keepByteSize(R)
    keepByteSize(G)
    keepbyteSize(B)

    pixel = new Pixel(R,G,B)
```

The program begins the same as the others in the assignment, initialising the variables and entering a “for each pixel” loop. The convolution matrix is input in memory at the READWRITE area at the bottom of the assembly file.

Within the for loop, the program branches to the main subroutine modifyPixel, which applies the convolution matrix to the pixel.

```

61  modifyPixel
62      STMFD SP!, {R4-R8,LR}
63
64      MOV R0,R8          ;copy I to R0 for function
65      BL indexToCoordinates; (r7=I,r6=W,r5=H)=>R11=X,R12=Y
66
67      SUB R11,#1         ;X--
68      SUB R12,#1         ;Y--

```

The first step that modifyPixel performs is to get the pixel coordinates from the pixel index, using the subroutine indexToCoordinates (see part 2-motion blur: indexToCoordinates).

X and Y are decremented so that the first pixel we modify corresponds to the first element of the convolution matrix.

```

77  forRow
78      CMP R3,#3
79      BEQ endForRow
80      forColumn
81          CMP R2,#3      ;WHILE
82          BEQ endForColumn
83          ;TEST IF PIXEL IS
84          ;IF TRUE, DON'T ST
85          CMP R11,#0
86          BLT outOfBounds
87          CMP R12,#0
88          BLT outOfBounds
89          CMP R11,R6
90          BGE outOfBounds
91          CMP R12,R5
92          BGE outOfBounds
93
94          ADD R9,#1      ;n++
95          BL pixelXelement;{
96
97      outOfBounds
98          ADD R10,#1      ;e++
99          ADD R11,#1      ;X++
100         ADD R2,#1        ;column
101         B forColumn
102     endForColumn
103         SUB R2,#3         ;col =
104         SUB R11,#3        ;X=X-3
105         ADD R12,#1        ;Y++
106         ADD R3,#1         ;row i
107         B forRow
108     endForRow

```

From there the program iterates through the pixels that are to be sampled. The forColumn loop moves across the array until X has been incremented 3 times (the width of the convolution matrix).

Once each pixel and element have been multiplied in a row, the for loop is exited and we move to the next row by incrementing Y within the forRow loop.

This process gives us 9 modified pixels stored in the stack, unless the outOfBounds condition is met. This only occurs for pixels on the edges of the image, where we don't want to sample pixels which are outside the pixel array. For this reason, R9 records the number of pixels actually stored to the stack.

Each iteration here uses the pixelXelement subroutine to correctly multiply the pixel by the corresponding element.

```

230      ;LOAD MATRIX ELEMENT
231      LDR R8,=#4
232      MOV R9,R10
233      MUL R10,R9,R8 ;R10=E*4
234      LDR R9, =MATRIX
235      LDR R10, [R9,R10] ;R10 = e

```

pixelXelement works by loading the matrix element from memory using e, the element index count which is incremented in each iteration of the forColumn loop.

The input coordinates X, Y are then converted to the corresponding pixel index using coordinatesToIndex, and this is used with the image start address to load the pixel from memory.

```

237      ;LOAD PIXEL[I']
238      BL coordinatesToIndex
239      MOV R9, R0
240      MUL R0, R9, R8 ; R0 = I'*4
241      LDR R9,[R4,R0] ; R9 = PIXEL

```

```

243      ;LOAD RGB BYTES TO R1,R2,R3 RESPECTIVELY
244      AND R1, R9, #0X000000FF
245      AND R2, R9, #0X0000FF00
246      AND R3, R9, #0X00FF0000
247
248      LSR R2, R2, #8 ;B=BYTE SIZE
249      LSR R3, R3, #16 ;C=BYTE SIZE
250
251      MOV R0,R1
252      MUL R1,R0,R10 ;R*e
253
254      MOV R0,R2
255      MUL R2,R0,R10 ;G*e
256
257      MOV R0,R3
258      MUL R3,R0,R10 ;B*e
259
260      BL keepRGBInRange
261

```

The pixel's colour channels are split into 3 registers using a mask with the AND function. The Green and Blue channels are converted to 8-bit values using the barrel shifter/LSR function.

The individual colour channels are now be multiplied by the matrix element.

A keepRGBInRange subroutine is used to keep R,G and B byte-sized.

The colour channels are then recombined into a 24-bit value, which is pushed to the stack for use later on in the program.

```

111     LDR R1, =0 ;R=0
112     LDR R2, =0 ;G=0
113     LDR R3, =0 ;B=0
114
115     LDR R12, =DIVISION
116     STRB R9, [R12,#1] ;
117     ;FOR THE STACKED PIXELS,
118     ;ADD THE MODIFIED R,G,B V
119 forStackedPixels
120     CMP R9, #0
121     BEQ endForStackedPixels
122     SUB R9, #1
123     LDMFD SP!, {R0}
124     BL popAndAdd ;pop next
125     B forStackedPixels
126 endForStackedPixels
127

```

After the forRow loop has stored all the adjusted pixels to the stack, the program sums them up and, if division is turned on, gets their average (division is turned on by a Boolean value stored in memory).

The number of pixels stored to the stack is stored to a space reserved for it in memory, in case division is turned on and it needs to be used later on.

A forStackedPixels loop then branches to the popAndAdd function n times.

popAndAdd pops a pixel from the stack, and adds its individual colour channels to the registers R,G,B, which the program uses here to store the summations of the colours.

If division is turned on, the program branches to the divideRGBbyn subroutine, which returns the averages of the modified colour channels (using repeated subtraction).

At the end, we have 3 registers holding the R,G,B colour channels of the pixel modified by the convolution matrix. The original pixel's channels are replaced with these bytes, and the subroutine branches back to main, where we move on to the next pixel.

```

143     ;REPLACE ORIGINAL PIXEL WITH THE MODIFIED VALUES
144     ;BL coordinatesToIndex;(should take R11,R12,R5,R6 as inputs)=>'I' in R0
145     LDR R11, =4
146     MUL R0, R11, R8 ;R0=I*4
147     ADD R0, R0, R4 ;R0 = A+I*4
148     ;STORES MODIFIED RGBs IN PIXELS[I]
149     STRB R1, [R0]
150     STRB R2, [R0,#1]
151     STRB R3, [R0,#2]

```






Testing and debugging section 3

This program didn't take as long as the others to debug. When writing the code I tried to be as diligent as possible in preventing errors and it seems the effort paid off. As well as this, I was able to recycle a lot of the code from the previous 2 sections for use in this.

The convolution matrix program has 3 inputs, stored in memory:

- The matrix
- Division on/off
- Iterations of program

While I wrote the program with blurring in mind (such as box or motion blur) playing around with different values of the inputs shows a range of effects that the program can produce:

MATRIX	DIVISION	ITERATIONS	Image	Comment
<pre>0 0 0 0 1 0 0 0 0</pre>	0	1		The identity convolution matrix leaves the image unchanged. Each pixel is replaced with itself.
<pre>1 1 1 1 1 1 1 1 1</pre>	1	1		<p>The blurring effect works perfectly, but unfortunately its intensity cannot be increased in the program due to the constant size of the matrix.</p> <p>Here, each pixel is the average of its neighbours + itself.</p> <p>Note that division is turned on.</p>
<pre>0 0 0 0 2 0 0 0 0</pre>	0	1		The program can also increase the contrast, however it provides less control over the increase (contrast can only be doubled, tripled, etc).
<pre>0 0 0 0 0 0 0 1 0</pre>	0	20		<p>This matrix replaces each pixel with the value of the one below it, creating an upwards translation effect.</p> <p>It was for these type of effects that I added the iterations variable/loop, as each complete cycle of the matrix over the image only translates it by 1 pixel.</p>

<div data-bbox="124 264 236 365"><div>0 0 0</div><div>0 0 0</div><div>0 0 1</div></div>	<div data-bbox="352 309 376 342">0</div>	<div data-bbox="539 309 584 342">20</div>	<div data-bbox="836 190 1008 405"></div>	<div data-bbox="1195 197 1493 472"><p>Similar to the previous matrix, this matrix causes each pixel to be replaced with the one to its immediate bottom right. This creates an upward diagonal translation.</p></div>
<div data-bbox="124 658 236 759"><div>1 0 0</div><div>0 1 0</div><div>0 0 1</div></div>	<div data-bbox="352 703 376 736">1</div>	<div data-bbox="549 703 572 736">1</div>	<div data-bbox="679 510 1171 1122"></div> <div data-bbox="679 1160 1171 1771"></div>	<div data-bbox="1195 483 1493 1406"><p>The program can also create a motion blur, however it is far from perfect. Since in this program there is no way to not include the 0 elements in the calculation, the average is brought down by those, causing unwanted darkening. As well as this, the radius cannot be greater than 3, due to the constant size of the matrix. Combined, these factors make the motion blur barely perceptible (but it's there, I promise).</p><p>Below the first image is a copy brightened with external software.</p></div>