Embedded system 에서 제한된 자원을 이용하여 효율적인 ARM 코드 작성

마이크로프로세서응용
PROJECT

전자정보공학부 (IT 융합 전공)

20160458 김지우
20160459 김초원
20160521 유영미
20160589 조윤지

목차

1. **시스템 개요 (3p)**

    1-1 동작원리 (3p)

    1-2 알고리즘(4p)

    1-3 역할분담(6p)

2. **각 함수 별 설명 및 블록 다이어그램(7p)**

    2-1 순수 C 코드(7p)

    2-2 어셈블리 코드(15p)

3. **검증 (37p)**

    3-1 C code (37p)

    3-2 assembly(56p)

4. **프로젝트 진행 과정**

    4-1 프로젝트 일정 계획

    4-2 프로젝트 진행 일지

1. 시스템 개요

1-1 동작 원리

1920*1080 사이즈의 이미지의 data 를 저장한 바이너리 파일을 CNN 연산을 사용하여 연산 처리하고 최종 이미지 결과를 확인한다.

1 단계 : 이미지 파일의 바이너리 **data 를 메모리 공간으로 불러온다**.
2 단계 : 이미지의 픽셀 하나는 RGBA 총 32bit=4Byte 에 저장된다. 이 데이터를 R, G, B, A 로 나눈 후 가장 큰 data 와 가장 작은 data 의 평균값을 해당 픽셀안에 저장한다. 이 단계를 실행하면 3 가지 색상이 한가지가 된다. =>**Grayscale**
3 단계 : 위의 단계를 진행한 data 와 커널을 **Convolution** 한다.
4 단계 : **max pooling** 을 진행한다. 실행 시 1920*1080 사이즈의 이미지는 960*540 사이즈로 변환된다

○참고 사이트

-딥러닝 이해 :

https://hamait.tistory.com/535

http://taewan.kim/post/cnn/

-이미지 파일 불러오기

https://sacstory.tistory.com/entry/바이너리-읽기(이미지 data 를 16 진수 data 로 불러오는 것을 비쥬얼 스튜디어로 가져와 실행하여 0x40000000 에 저장된 메모리와 비교하여 기본 개념을 이해해봄)

-Convolution 과 padding 관련

http://realheart.egloos.com/2193436

https://stackoverflow.com/questions/3982439/fast-2d-convolution-for-dsp

https://umbum.tistory.com/223

https://blog.naver.com/stelch/221497552593

https://kr.mathworks.com/help/matlab/ref/conv.html

https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/duz1504034427736-1.html

https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/hvq1504034428018.html

C 코드 최적화를 위한 참고

https://m.blog.naver.com/knix008/220668672529

1-2 알고리즘

1-2-1 C 코드 알고리즘

1 단계 load Image

: 이미지 불러오기 => main 함수에서 이미지 불러오는 함수의 전달인자로 이미지 데이터를 불러올 배열의 주소를 받아온다. for 문을 통해 1920*1080 개의 data 를 배열에 불러온다.

2 단계 Grayscale 함수

: padding=> 패딩을 위해 행과 열에 2 를 더한 1922*1082 사이즈의 배열을 main 에서 선언하여 Grayscale 함수의 전달인자로 받아온다. 원본 이미지 data 를 저장하고 있는 배열을 char 로 선언하고 열의 1920*4 로 확장하여 전달인자로 가져온다. 이는 RGB 에서 R, G, B, A 에 각각 접근 가능하도록 한다. Grayscale 배열의 가장 자리에 0 을 집어넣어준다.

Grayscale => 배열의 (1,1)좌표부터 RGBA 에서 가장 큰 값과 작은 값의 평균을 계산하여 저장한다. 계산은 1920*1080 번 실행한다

3 단계 convolution 함수

: convolution=> main 함수로부터 해당 함수 결과를 저장할 1920*1080 배열과 Grayscale 결과 배열의 주소를 전달받는다. kernel W 배열은 해당 함수내에서 선언한다. 빠른 계산을 위해 1024 를 값 하여 W 배열을 선언한다. 4 중 for 문을 이용하여 Grayscale 한 배열과 W 를 convolution 한 후 새로운 배열에 저장한다. 계산 값이 음수가 나오면 0 으로 바꿔 배열 안에 저장한다.

4 단계 max pooling 함수

: max pooling =>: main 함수로부터 convolution 결과 배열의 주소와 최종 결과, max pooling 결과를 저장할 배열의 주소를 전달받는다. 배열에 저장된 data 를 2*2 사이즈로 잡고 4 개의 data 중에서 가장 큰 값을 결과 배열에 저장한다. 이때 1024 를 나눠 저장하도록 한다.

1-2-2 어셈블러 코드 알고리즘

1 단계 Relocation

: 메모리에 연속적으로 저장되어 있는 원본 이미지 data 를 relocation 하여 1080*1920 의 2 차원 배열과 같은 모습으로 메모리에 저장한다. 2 차원 배열로 생각하면 0 행의 data 는 0x40000000 부터 1920 개의 data 가 저장된다. 그 후 128*4 번지 후부터 data 를 저장한다. 즉, 0x40000000 + 2048*4*N 한 주소(0x2000)가 N 행의 시작주소가 된다. 기존 메모리의 마지막 주소부터 4 개씩 데이터를 가져와 옮긴다.

2 단계 Grayscale

: convolution 시 행과 열의 사이즈가 2 만큼 추가된 padding 한 Grayscale 배열을 사용하도록 하기 위해 Grayscale 배열 생성시 각 가장자리에 0 을 넣어주고 1, 1 에 해당하는 주소부터 L 값을 저장하도록 한다.

cycle 을 줄이기 위해 4B 짜리 DATA 4 개를 한번에 가져온다. 먼저 한 개의 4B data 를 R, G, B, A 를 나누기 위해 SHIFT 연산을 진행한다. 이때 A data 는 사용하지 않으므로 SHIFT 3 번만 진행한다. 즉, 4B 에서 뽑아낸 1B 짜리 DATA 3 개를 4B 공간에 저장한다. 이 3 개의 값들을 비교하여 가장 큰 값과 작은 값을 찾아내어 평균 값을 저장한다. 위의 과정을 4 번 반복하여 총 4 개의 L 값을 구해 메모리에 저장한다. 이때 각 이미지의 행의 첫번째 픽셀이 2^n 의 배수로 시작하는 주소에 저장되도록 해준다.

3 단계 convolution

: w 배열은 값이 소수 형태이므로 1024 를 곱한 값에 임의로 소수점 아래 값을 없앤 값을 사용한다. W 배열에 중복되는 값이 있으므로 총 4 개의 레지스터에 해당 값을 저장한다.

Grayscale 배열의 data 를 3 개씩 가져와 w 와 convolution 처리->다음 행 3 개 가져와 w 와 convolution 처리 -> 그 다음 행 가져와 w 와 convolution 처리

위의 과정을 loop 로 반복하며 각 이미지의 행의 첫번째 픽셀이 2^n 의 배수로 시작하는 주소에 저장되도록 해주기 위해 각 행은 1920 개의 data 가 저장되도록 한다.

4 단계 max pooling

: 2X2 사이즈로 데이터를 가져오므로 0 번 행에 해당하는 4 개의 DATA 를 가져오고 1 번 행의 DATA 를 가져와 2X2 배열을 2 개 만들어낸다. 각 배열의 max 값을 각 레지스터에 저장한다. 다시 한번 위의 과정을 진행하면 총 4 개의 MAX 값을 만들 수 있다. 이 4 개의 DATA 를 한번에 메모리에 저장한다. 이때, 해당 값은 1024 을 곱한 배열 w 를 이용하여 구한 값이므로 SHIFT 연산을 하여 1024 로 나누어 준다. 위의 과정을 해당 이미지가 960*540 사이즈가 되도록 반복한다.

1-3 역할 분담

김 지우: C 함수 1 단계 작성, 어셈블러 3, 4 단계 작성. 작성된 코드 최적화 , 보고서 작성 - assembly 함수 설명

김 초원: C 함수 3 단계 작성, 어셈블러 3 단계 작성, 보고서 작성 - 블록 다이어그램(제작 및 설명), 프로젝트 진행상황

유 영미: 어셈블러 1 단계 작성, 보고서 작성-1 시스템 개요, 블록 다이어그램(제작 및 설명)

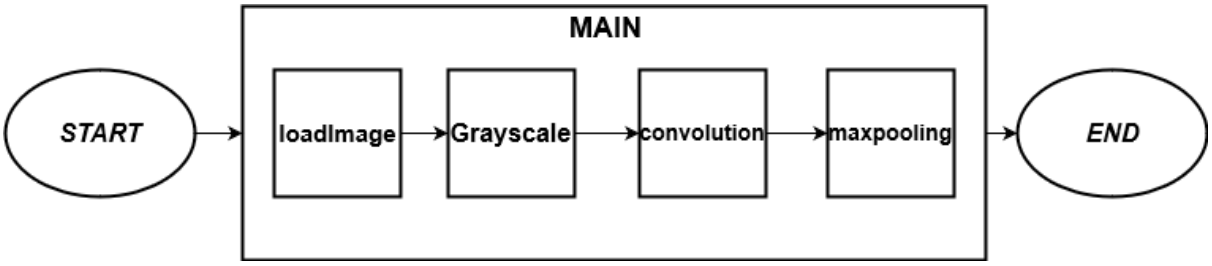
조 윤지: C 함수 2 단계 작성, 어셈블러 3 단계 작성, 보고서 작성 - 성능.

★모든 코드는 위의 작성자가 알고리즘, 기본 틀을 구현하였으며 오류 해결 및 코드 수정, 코드 최적화와 결과 비교와 확인, 검증들은 모든 조원이 함께 진행함.

2. 각 함수 별 설명 및 다이어그램

2-1 순수 C 코드

2-1-1 전체 블록 다이어그램



Main 함수 안에서 4 개의 함수 호출을 한다.

Load Image 는 메모리에 저장된 이미지를 불러와서 배열에 저장한다.

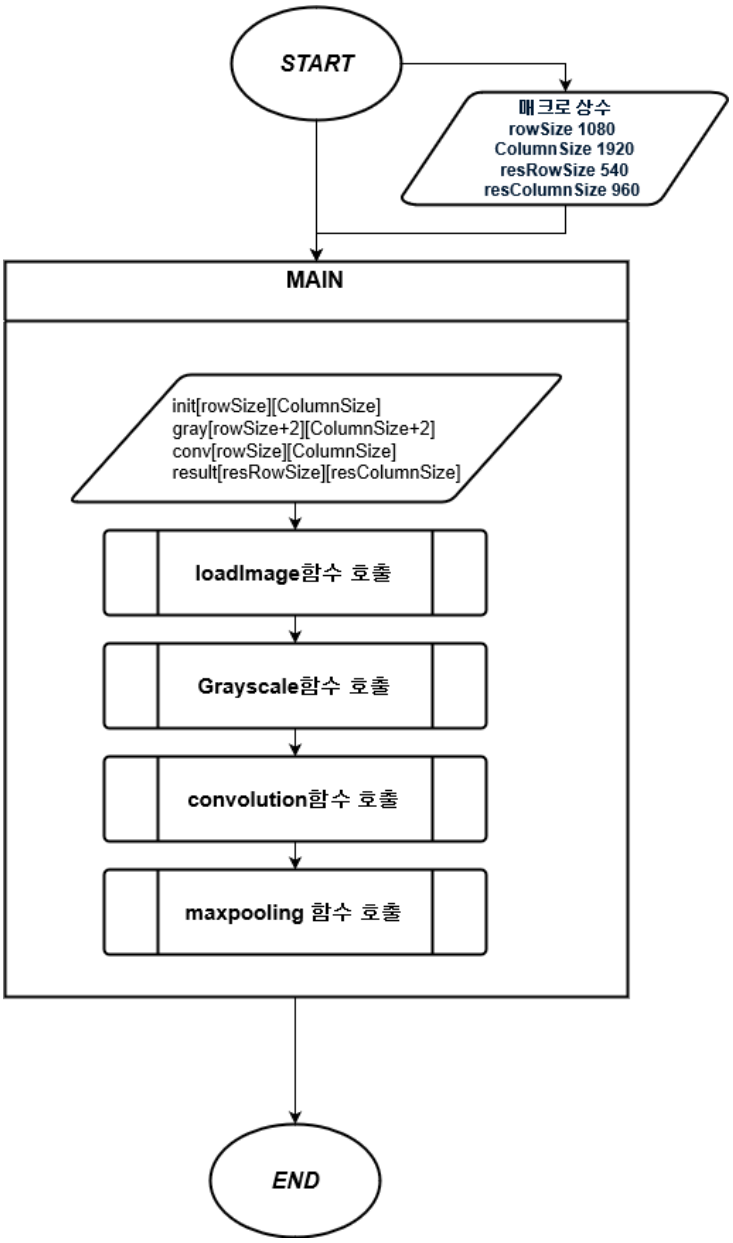
Grayscale 은 변환식을 이용해 8bit 이미지로 변환한다.

Convolution 은 변환식으로 바꾼 값에 대해 convolution 연산을 수행한다.

Max pooling 은 2X2 배열 중 가장 큰 값을 추출하는 연산을 수행한다.

추가적으로 [100,100], [100,200], [200,100], [200,200] 위치의 픽셀 값을 프린트하기 위해 PrintDecimal 으로 각 배열에 있는 값들을 출력하였다.

2-1-2 main 함수



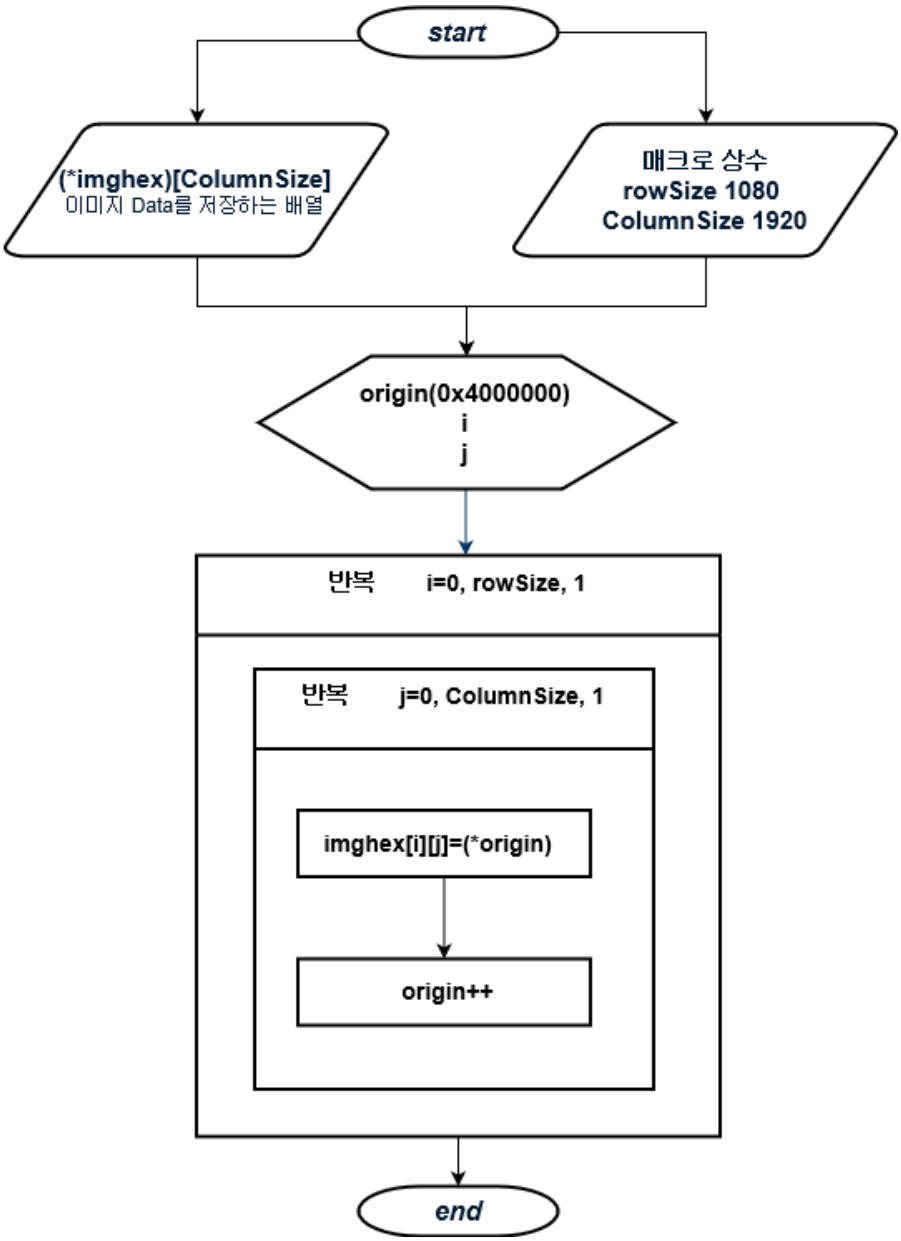
이미지 배열의 행과 열 크기는 매크로 상수를 통해 선언하였다. 매크로 상수를 이용하는 것이 변수를 사용하는 것보다 처리속도가 빠르다.

- 매크로 상수 의미

Row Size	Column Size	Res Row Size	Res Column Size
이미지 배열의 행	이미지 배열의 열	최종 배열의 행	최종 배열의 열
1080	1920	540	960

4 개의 함수 load Image, Grayscale, convolution, max pooling 를 main 에서 호출하고 실행완료 후 종료한다.

2-1-2 Load Image 함수



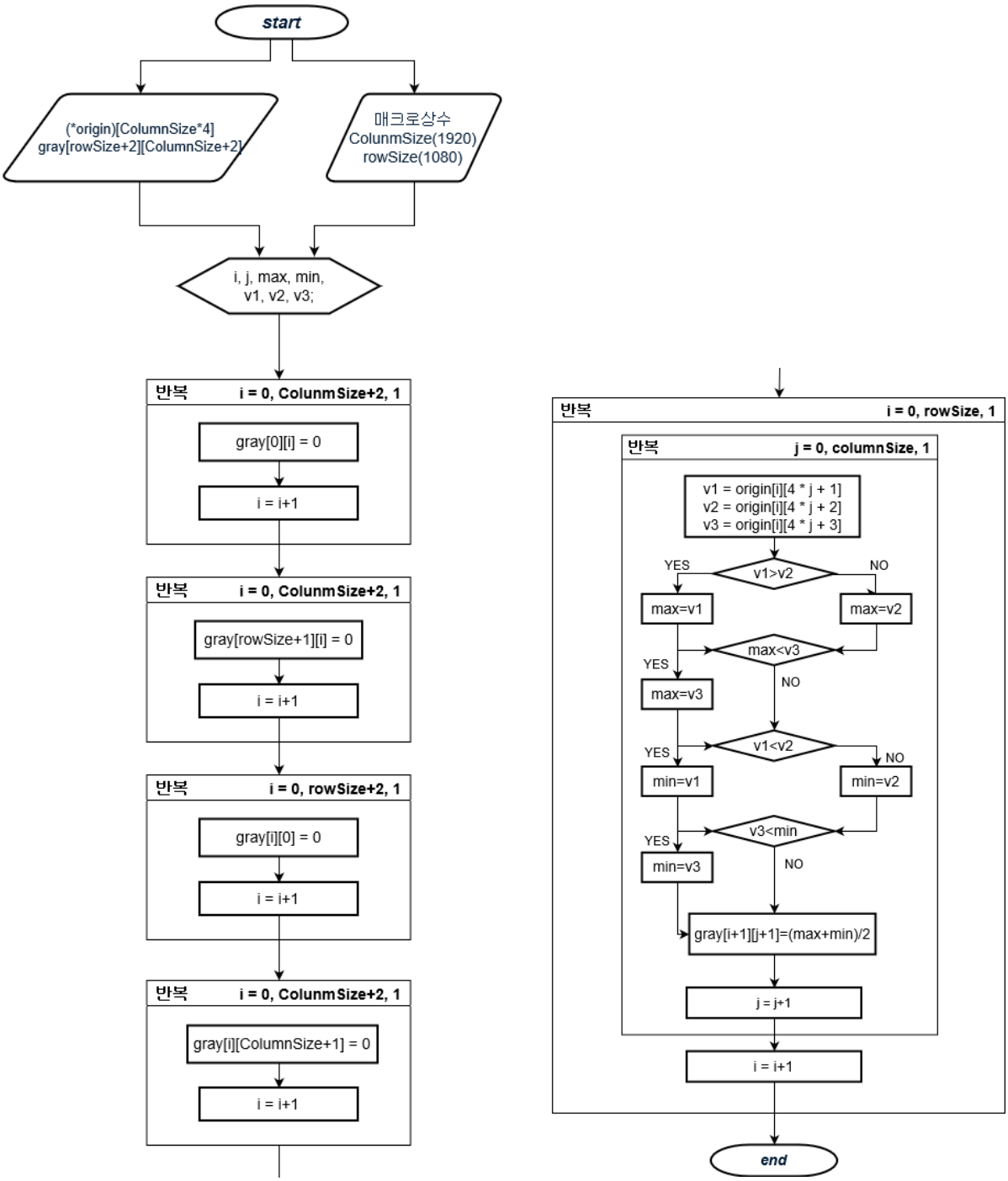
사용되는 매크로 상수는 row Size(1080)와 Column Size(1920)이다.

Origin 변수는 메모리에 저장된 이미지 값을 다른 배열에 저장시킬 시작주소이다.

반복문을 통해 메모리에 저장된 이미지 값을 origin 주소부터 imghex 배열에 저장한다.

이는 이미지 배열의 크기만큼 반복한다.

2-1-3 Grayscale 함수



사용되는 매크로 상수는 row Size(1080)와 Column Size(1920)이다.

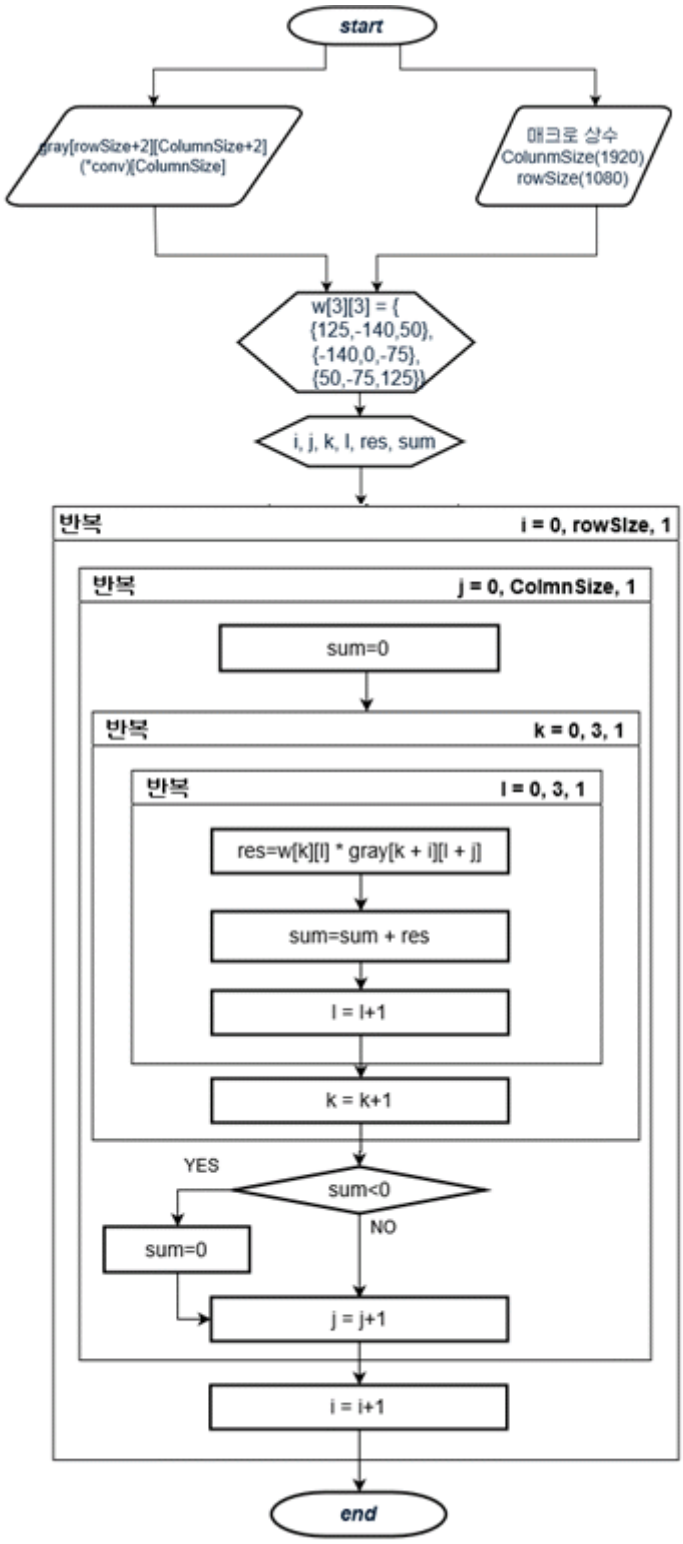
Max 는 값들을 비교해서 가장 큰 값을 저장한 변수이고 min 은 가장 작은 값을 저장한 변수이다.

gray 배열은 이미지크기보다 각각 +2 한 배열의 크기를 갖는다. 이는 padding 을 위한 것이다.

반복문 4 개는 padding 의 과정이다. 순서대로 첫번째 행, 마지막 행, 첫번째 열, 마지막 열에 모두 0 을 넣어준다.

RGB 를 비교하기 위해서 변수를 3 개로 선언하였다. v1, v2, v3 변수들을 비교하여 가장 큰 값(max)와 가장 작은 값(min)을 찾는다. L = $\frac{\max + min}{2}$ 의 결과값들은 gray 배열의 [i+1] [j+1]부터 저장한다. 이는 이미지 배열의 크기만큼 반복한다.

2-1-4 convolution 함수



사용되는 매크로 상수는 row Size(1080)와 Column Size(1920)이다.

Convolution 연산을 위해 w 배열(3 X 3)의 선언 및 초기화를 미리 해주었다.

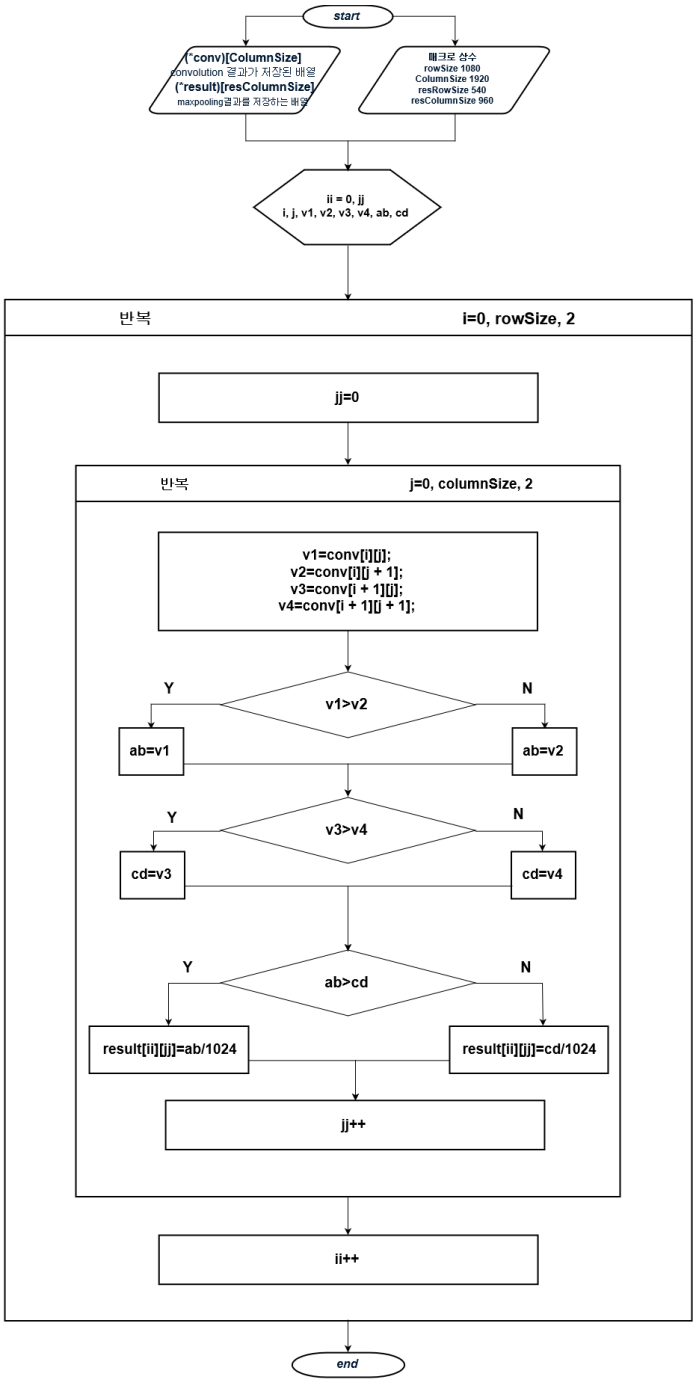
```
w[3][3] = {
{0.125,-0.14,0.05},
{-0.14,0,-0.075},
{0.05,-0.075,0.125}}
```
→
```
w[3][3] = {
{125,-140,50},
{-140,0,-75},
{50,-75,125}}
```

원래 w 배열은 double 형태이지만 X1000 을 해줘서 int 형으로 바꿔서 초기화해주었다. 그 이유는 원래대로 gray 배열이 int 형이고 w 배열이 double 형이었을 때 int 형과 double 형을 계산하면 자동형변환을 하는 시간과 메모리 문제로 인해 너무 많은 시간이 걸린다. 실제로 실행해봤을 때 거의 1 시간이 넘도록 끝나지 않았다.

Convolution 연산은 gray 배열(grayscale 과 padding 까지 완료한 배열)과 w 배열을 각각 곱해주고 그 값들을 모두 더한 후 conv 배열에 저장한다.

RELU activation function 은 결과값이 음수이면 모두 0 처리해주는 것이다. 이 함수는 if 문을 통해 sum<0 이면 0 으로 바꿔주었다. 이는 이미지 배열의 크기만큼 반복한다.

2-1-5 Max pooling 함수



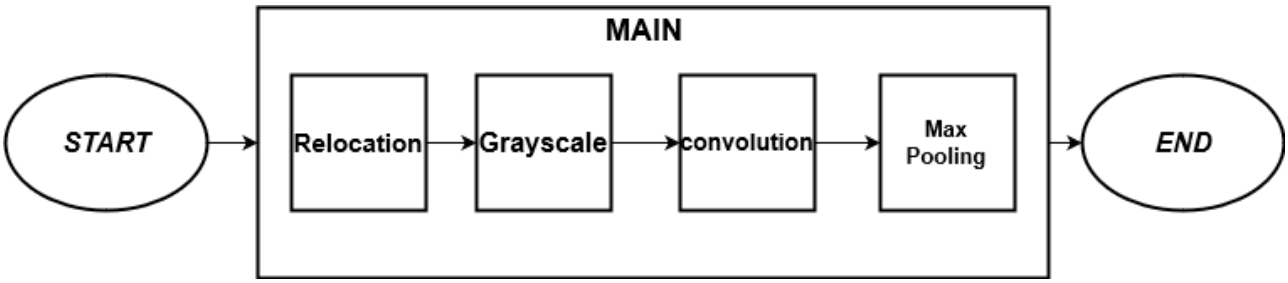
사용되는 매크로 상수는 row Size(1080)와 Column Size(1920)이다.

2 x 2 배열만큼의 값들을 비교해야 하므로 변수는 4 개를 선언하였다. (v1, v2, v3, v4)

ab 와 cd 는 2 개의 값들 중 가장 큰 값을 저장한 변수이다. 이 둘을 비교하여 가장 큰 값은 result 배열에 저장한다. 이는 이미지 배열의 크기만큼 반복한다.

2-2 어셈블리 코드

2-2-1 전체 블록 다이어그램



Main 함수 안에서 4 개의 함수 호출을 한다.

Relocation 은 메모리에 저장된 이미지를 불러와서 배열에 저장한다.

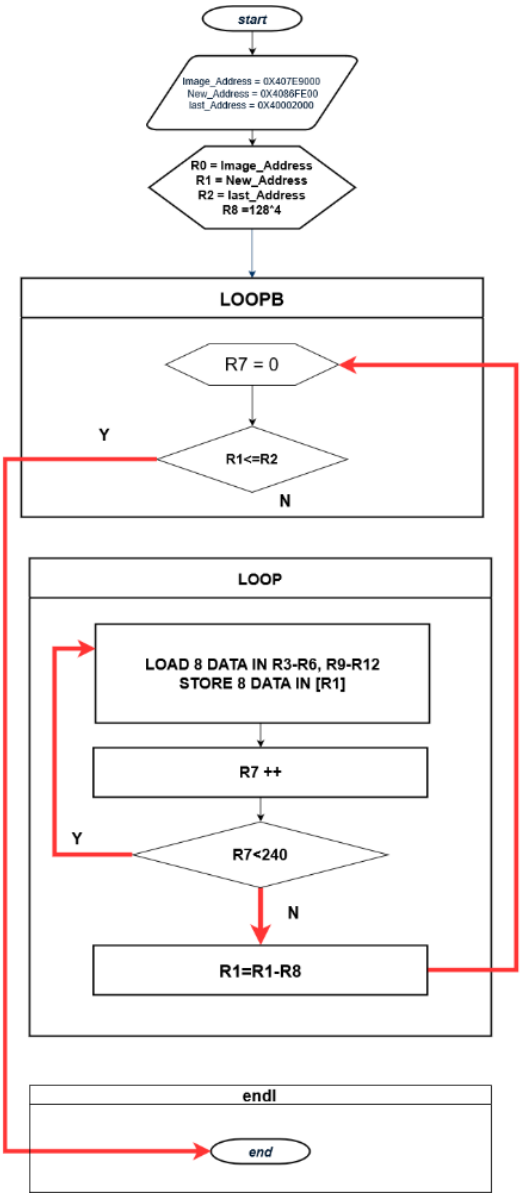
Grayscale 은 변환식을 이용해 8bit 이미지로 변환한다.

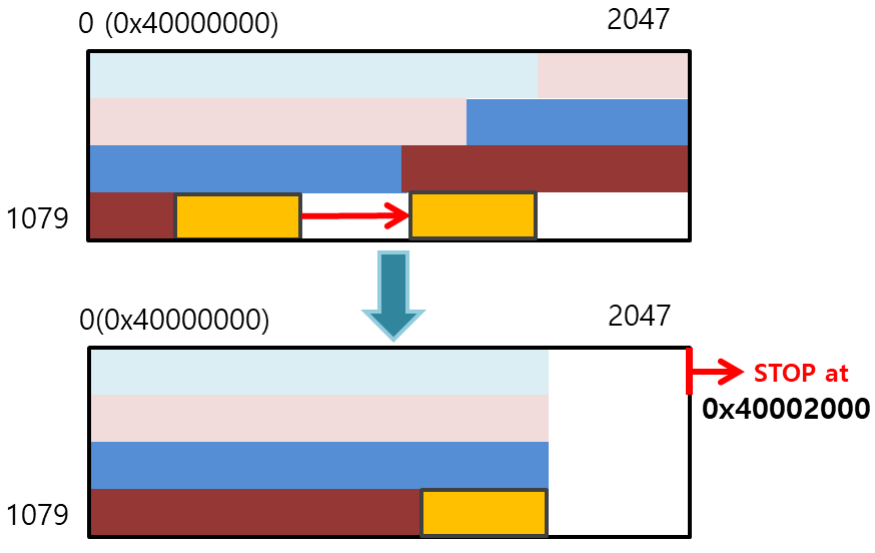
Convolution 은 변환식으로 바꾼 값에 대해 convolution 연산을 수행한다.

Max pooling 은 2X2 배열 중 가장 큰 값을 추출하는 연산을 수행한다.

추가적으로 [100,100], [100,200], [200,100], [200,200] 위치의 픽셀 값을 프린트하기 위해 각각의 메모리 주소에 있는 값을 PrintDecimal 으로 각 배열에 있는 값들을 출력하였다.

2-2-2 Relocation 함수





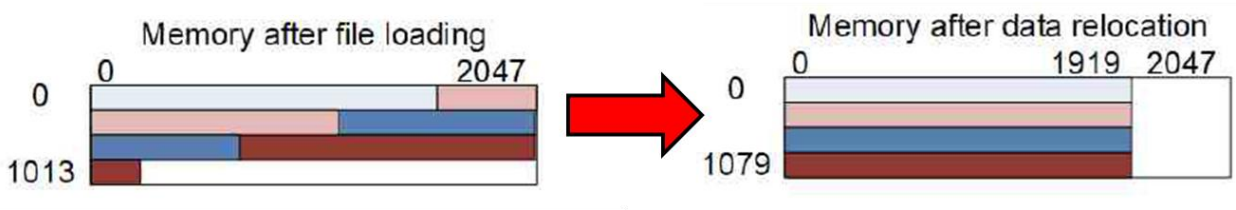
설명

Relocation 의 assembly 코드 원리는 위의 그림과 같다. 본래 연속적으로 저장된 이미지의 **끝부분부터** 32Byte 씩 (4 바이트 8 개) 가져와 Relocation 위치에 넣는다. (Original image) 끝부분부터 가져와 (Relocation image) 끝부분부터 넣기 때문에 첫 부분에 저장된 1920*4Byte 는 건드릴 필요가 없다. 그러므로 Relocation 위치가 0x40002000 (처음 2048*4 가 되는 부분)가 되었을 때는 더 이상 재배치 하지 않고, 함수를 리턴하게 된다.

```
AREA    mem_Relocation, CODE, READONLY
   ;EQU       &0
;   SWI_Exit       EQU       &11

   EXPORT Relocation

   ENTRY
Relocation
```

```
   LDR R0, Image_Address    ;r0=0X407E9000(last address of original image)
```

r0 는 original image pointer 역할을 맡는다. Image (hex file)은 0x40000000 부터 0x407E9000 까지 로딩되어 있다. r0 에는 로딩된 Original image 의 끝 주소를 넣는다. (0x40000000+0x7E9000), 이 때 0x7E9000 은 1920*1080*4 에 해당하는 값이다. 이 곳부터 Original Image 를 읽어온다.




```
   LDR R1, New_Address       ;r1= 0X4086FE00(last address of relocated image)
```

r1 은 relocation image pointer 역할을 맡는다. 왼쪽 그림은 이미지를 로딩했을 때 메모리 상태이다. (메모리에 연속적으로 할당) Relocation 한 이후에 메모리 상태는 오른쪽 그림과 같아진다. r1 에는 재배치된 이미지의 끝 주소를 넣는다. 해당 주소는 0x40000000 + 0x86FE00 으로, 0x86FE00 은 2048*__1079__*4+1920*4 에 해당하는 값이다. 이곳부터 Relocation Image 를 저장한다.

```
   LDR R2, last_Address
```

r2 는 relocation pointer (r1)가 첫 번째 줄에 도달했을 때, 바로 return 하기 위해 첫 번째 줄이 끝나는 주소(0x40002000)를 저장해두는 역할을 한다.

```
   MOV R8, #128<<2          ;r8=128 is  movw-jmp size
```

R8 에는 128*4 만큼의 정수를 저장한다. 이미지를 재배치 할 때, 빈 공간이 128*4Byte 만큼 남게 된다. 그 곳을 건너 뛸 때 사용한다.

```
LOOPB                    ;loop count reset
   MOV R7, #0             ;r7=480 is move-set count
```

R7 은 count 변수 i 의 역할을 맡기 때문에 0 으로 초기화한다.

```
   CMP R1,R2       ;if r1<0x40002000
```

relocation 포인터인 r1 이 0x40002000 (첫 번째 줄) 끝에 도달했는지 물어 본다. 끝에 도달했거나, 그것보다 작으면 (즉, 첫 번째 줄 안에 도달했으면) **endl** 로 이동한다.

```
   BLE endl            ;end


   ;else goto loop

LOOP
   ;LDMDB R0!,{R3-R6}
   ;LDMDB R0!,{R9-R12}
   LDMDB R0!,{R3-R6,R9-R12}       ;get 8data (RGBA x4=16B)
```

최적화 후, 네 개씩 Load 하던 코드는 없애고, 한 번에 8 개씩 Load 한다. 끝에서부터 Load 하기 때문에 DB 를 사용했다.

```
   ;STMFD R1!, {r3-r6}
   ;STMFD R1!, {r9-r12}
   STMFD R1!, {R3-R6,R9-R12}        ;store 8data
```

최적화 후, 네 개씩 저장하지 않고, 8 개씩 재배치한다. 끝에서부터 저장하기 때문에 FD 를 사용했다.

```
   ADD R7,R7,#1          ;r7--;
   CMP R7,#240          ;if (r7<480)
   BLT LOOP            ;goto loop
```

r7 을 하나 증가시키고, 240 과 비교한다.1920*1080 픽셀의 한 줄에는 1920*4Byte 가 저장되어 있다. 4Byte 8 개씩을 Load, Store 하기 때문에 한 줄에서 240 번 Loop 를 돌게 된다. 240 보다 작으면 LOOP 를 계속 돌도록 만든다.

```
   ;else (r7>=480) == data copy end
   SUB R1,R1,R8         ;r7-=128
   B LOOPB
```

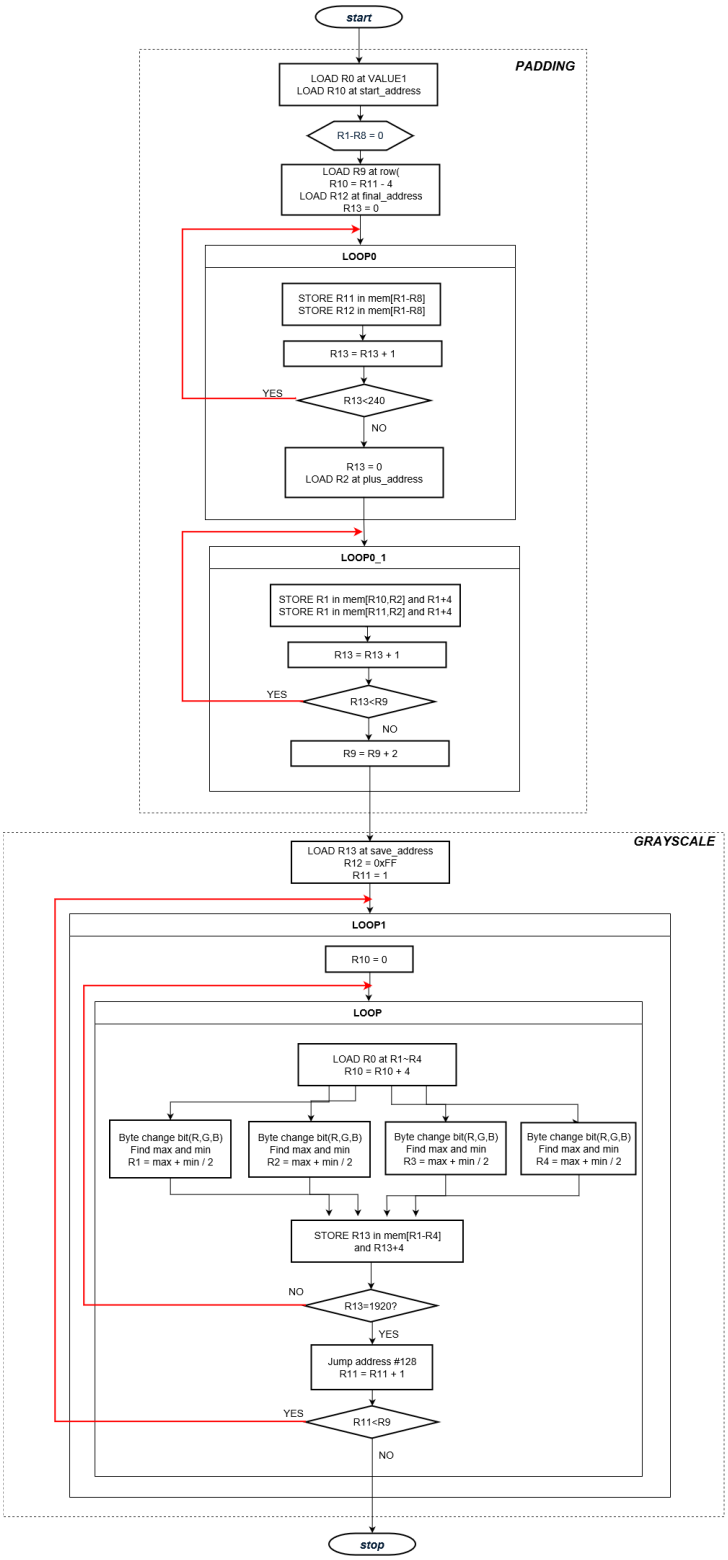
한 줄이 끝났으면, R1 에서 128*4 만큼을 뺀다. Relocation 공백 부분을 건너뛰는 것이다. 건너 뛰고 LOOP 를 다시 돌기 시작하는데, LOOPB 로 이동하여 R7 을 0 으로 다시 만드는 과정을 시행하게 된다.

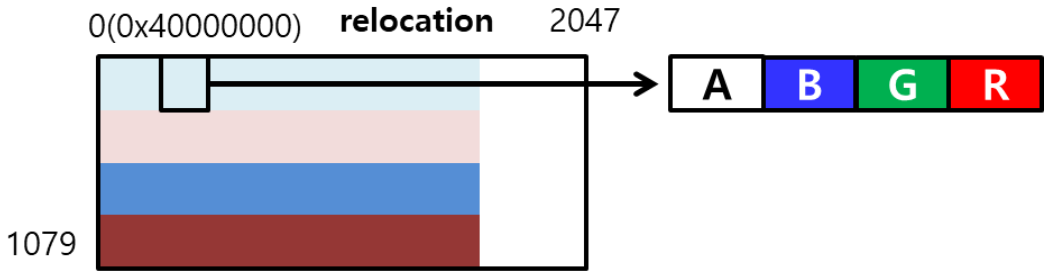
```
endl
   BX lr
```

완전히 끝난 경우, BX LR 로 자신을 부른 본래 main 함수로 돌아간다.

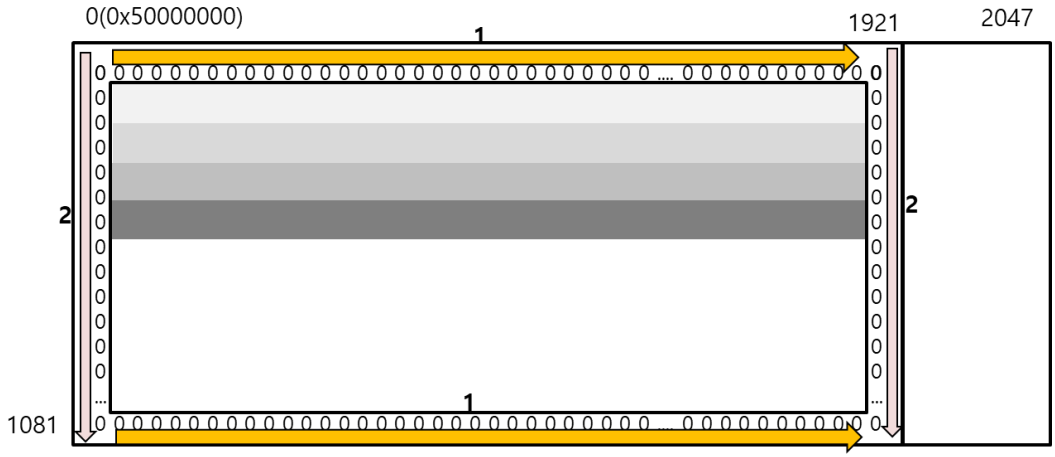
```
 Image_Address DCD 0X407E9000
New_Address DCD 0X4086FE00
last_Address DCD  0X40002000

   END
```

2-2-3 Grayscale 함수





GrayScale 의 assembly 코드 원리는 위의 그림과 같다. 이미지 로딩 후 4Byte(32bit)를 가져오면, Little Endian 이기 때문에 (낮은 주소) A,B,G,R (높은 주소)로 저장이 되어있다. 먼저 4Byte 를 가져와 shift, AND 과정을 거쳐 R,G,B 를 따로 추출하여 저장하고, Max, Min 값을 계산하여 L 값을 추출하고, gray[][] 배열에 저장한다.



gray[][]가 메모리에 어떻게 들어가있는지 나타낸 것이다. grayscale assembly code 에서는 다음 convolution 을 위해 가장자리를 0 으로 padding 처리를 해준다. padding 을 저장해야 하기 때문에, 한 줄당 2*4Byte 를 더 많이 차지하며, 총 1082 줄을 차지하게 된다. 그래서 공백을 제외하고 **1922*1082***4Byte 공간에 데이터가 들어있다. 처음 padding 에서는 맨 윗줄과 맨 아랫줄을 0 으로 만든다. (각 줄의 첫 번째, 마지막 방은 그 다음 padding 과정에서 0 으로 만들 것이기 때문에 두 번째 방부터 시작한다.) 그 다음 과정의 padding 에서는 옆면 padding 을 실행한다. 첫 줄부터 마지막 줄까지 모두 0 으로 만든다.

```
AREA gray, CODE,READONLY
      EXPORT Grayscale

   ENTRY
   ;0x50871e00 <-final grayscale
   ;0x50873dfc <-real final
Grayscale

;****padding******
   LDR r0,VALUE1 ;address
```

```
LDR r11,start_address
```

r0 에는 relocation 한 결과가 저장된 메모리 주소가 들어간다. relocation 배열 pointer 역할을 한다. r0 에는 0x40000000 의 값이 저장된다. r10 에는 grayscale 배열의 첫 주소가 들어가게 된다. grayscale 배열의 pointer 역할을 한다. r10 에는 **0x50000004** 의 값이 저장된다. grayscale 배열은 0x50000000 부터 시작이지만, 처음 padding 과정에서는 (윗줄 , 아랫줄에 대한 패딩) 두 번째 방부터 0 으로 초기화하기 때문에 0x50000004 가 들어가게 된다.

```
MOV r1,#0 ;r1=0 for initialize 0 (when STM, use r1~r8)
MOV r2,#0 ;r2=0
MOV r3,#0 ;r3=0
MOV r4,#0 ;r4=0
MOV r5,#0 ;r5=0
MOV r6,#0 ;r6=0
MOV r7,#0 ;r7=0
MOV r8,#0 ;r8=0
```

r1-r8 까지 모두 0 으로 초기화 한다. 이 값들은 padding 을 위해 사용 된다.

```
LDR r9,row ;1082
SUB r10, r11, #4 ;r11=r10 for loop0_1, 0x50000000
```

r9 에는 옆면의 padding 을 처리하기 위해 1082 을 저장한다. 또한 r10 에는 좌, 우 패딩을 동시에 처리하기 위해 0x50000000 의 값을 저장해야 한다. 그러므로 r11 에서 4 를 뺀 값을 저장하게 된다.

```
LDR r12,final_address ;final line 00000000....000
```

r12 에는 맨 밑줄 패딩을 위해 맨 밑줄 시작 주소를 저장한다. 시작 주소는 0x50872004 이다. 이도 역시 마지막 줄의 두 번째 방부터 0 으로 초기화하기 때문에 0x50872004 라는 값이 들어가게 된다.

```
MOV r13,#0 ;counter
```

패딩을 시작하기 전에 r13 을 0 으로 초기화 한다.

```
loop0
STM r11!,{r1,r2,r3,r4,r5,r6,r7,r8} ;first row 0000...00
STM r12!,{r1,r2,r3,r4,r5,r6,r7,r8} ;final row 0000...00
```

맨 윗줄과 맨 아랫줄에 8 개씩 0 으로 초기화시킨다.

```
ADD r13,r13,#1 ;counter=counter+1
CMP r13,#240 ;
BLT loop0
```

counter 를 1 증가시킨 다음, 240 번 돌았다면 (1920*4Byte 를 모두 padding 처리 했다면) 다음으로 넘어가고 그렇지 않다면 다시 padding 처리를 하러 loop0 으로 돌아간다.

```
MOV r13,#0 ;counter reset
LDR r2,plus_address
```

처음 패딩이 끝나고 (위, 아랫줄 패딩), 다음 패딩(옆면 패딩)으로 넘어가야 한다. counter 역할을 하는 r13 을 0 으로 리셋한다. 그리고 옆면 패딩 시, 계속 다음 줄로 넘어가야 하기 때문에 2048*4 을 더해야 한다. r2 에 그 값을 저장한다. (=0x2000=2048*4)

```
loop0_1
   STR r1,[r10,r2]!
   STR r1,[r11,r2]!
```

양 쪽 옆면에 0 을 저장하고 다음 줄로 넘어간다.

```
   ADD r13,r13,#1
   CMP r13,r9
```

r13(counter)에 1 을 더하고, counter 가 1082 가 되면 padding 을 끝낸다. 1082 보다 작다면, 다시 옆면 패딩을 하기 위해 loop0_1 로 이동한다.

```
   BLT loop0_1

;**********grayscale*******************
    SUB r9, r9, #2 ; 1080
```

r9 을 1080 으로 만든다.

```
    LDR r13,save_address
   MOV r12,#0xff ;for and (0x000000ff)
   MOV r11,#1 ;counter2 (cheak 1080)
```

r13 은 처음 데이터가 저장되는 주소이다. 0x50000000 에서 padding 을 한 첫 줄과, 다음 줄의 첫 방을 띄우면 2048*4+1*4 를 더해야 한다. 그러므로 0x50002004 가 저장된다. r11 은 1080 을 체크하기 위한 counter 이고, 1 로 초기화한다. r12 는 0x000000ff 가 저장되는데, 이는 R,G,B 를 AND 연산을 하여 추출해 내기 위함이다.

```
loop1

   MOV r10,#0 ;counter (cheak 1920)
```

한 줄에 1920*4Byte 가 있고, 한 줄 모든 데이터에 대해 grayscale 을 시행한다. r10 은 1920 을 체크하는 counter 이다.

```
loop
   LDM r0!,{r1,r2,r3,r4} ;load data
   ADD r10,r10,#4 ;r10=r10+4 (compare with 1920 later)
```

r1-r4 에 4 개의 pixel 을 저장하고, 4 개를 읽었으므로 r10 에는 4 를 더한다.

```
;**************r1**************
   AND r5,r12,r1,LSR #8
   AND r6,r12,r1,LSR #16
   AND r7,r12,r1,LSR #24
```

r12 에는 0x000000ff 가 저장 되어있고, r1 에는 0xRRGGBBAA 가 저장되어 있다. A 는 버리고, R, G, B, A 만 쓴다. r1 을 right shift 8 하면 0x00RRGGBB 가 되고, r12(0xff)와 AND 연산을 하면 B 만 남게 된다. r5 에는 B 가 저장된다. 같은 방법을 써서 r6 은 G 가, r7 은 R 이 저장되게 된다.

```
CMP r5,r6 ;compare
```

먼저 r5 와 r6 을 비교한다.

```
MOVGE r8,r5
MOVLT r8,r6
MOVLT r6,r5
```

만약 r5 가 크다면 r8 에 r5 를, r6 이 크다면 r8 에 r6 을 넣는다. 그리고 r6 이 크다면, r6 에 r5 를 저장한다. 이렇게 하면 r5,r6 중 **큰 값은 r8 에, 작은 값은 r6 에** 저장된다.

```
CMP r7,r8
MOVGE r8,r7
```

r7 과 r8 을 비교한다. r8 은 r5, r6 중 큰 값이다. r7 이 크다면 r8 에 r7 을 저장한다. 이렇게 되면 r8 에는 r5,r6,r7 (RGB)중 가장 큰 값이 들어가게 된다.

```
CMP r6,r7
MOVGE r6,r7
```

r6 과 r7 을 비교한다. r6 은 r5, r6 중 작은 값이 저장 되어있다. r6 이 크다면, r7 이 작다는 것이므로 r6 에 r7 을 저장한다. 이러면 r6 에 r5,r6,r7(RGB)중 가장 작은 값이 들어가게 된다.

```
ADD r1,r8,r6 ;MAX+MIN
MOV r1,r1,LSR #1 ;L=(MAX+MIN)/2
```

r1 에 (MAX(r8)+MIN(r6))/2 를 저장한다.

```
;**************r2**************

AND r5,r12,r2,LSR #8
AND r6,r12,r2,LSR #16
AND r7,r12,r2,LSR #24

CMP r5,r6 ;compare R,G
MOVGE r8,r5
MOVLT r8,r6
MOVLT r6,r5
CMP r7,r8
MOVGE r8,r7
CMP r6,r7
MOVGE r6,r7
ADD r2,r8,r6 ;MAX+MIN
MOV r2,r2,LSR #1 ;L=(MAX+MIN)/2
```

같은 방법으로, r2 에 (MAX+MIN)/2 를 저장한다.

```
;**************r3**************

   AND r5,r12,r3,LSR #8
   AND r6,r12,r3,LSR #16
   AND r7,r12,r3,LSR #24

   CMP r5,r6 ;compare R,G
   MOVGE r8,r5
   MOVLT r8,r6
   MOVLT r6,r5
   CMP r7,r8
   MOVGE r8,r7
   CMP r6,r7
   MOVGE r6,r7
   ADD r3,r8,r6 ;MAX+MIN
   MOV r3,r3,LSR #1 ;L=(MAX+MIN)/2
```

같은 방법으로, r3 에 (MAX+MIN)/2 를 저장한다.

```
;**************r4**************

   AND r5,r12,r4,LSR #8
   AND r6,r12,r4,LSR #16
   AND r7,r12,r4,LSR #24

   CMP r5,r6 ;compare R,G
   MOVGE r8,r5
   MOVLT r8,r6
   MOVLT r6,r5
   CMP r7,r8
   MOVGE r8,r7
   CMP r6,r7
   MOVGE r6,r7
   ADD r4,r8,r6 ;MAX+MIN
   MOV r4,r4,LSR #1 ;L=(MAX+MIN)/2
```

같은 방법으로, r4 에 (MAX+MIN)/2 를 저장한다.

```
;*******************************
   STM r13!,{r1,r2,r3,r4} ;store
```

r13 (gray[][] save_address pointer)에 r1-r4 의 계산한 값을 저장한다.

```
  CMP r10,#1920
  BNE loop
```

1920 과 r10(1920 까지 가는 counter)를 비교하여, 1920 이 아니라면 계속해서 Loop 를 돈다.

```
  ADD r13,r13,#128<<2 ;blank space
  ADD r0,r0,#128<<2 ;blank space
```

만약 한 줄이 넘어갔다면, (1920 번 grayscale 처리를 했다면) 공백 처리를 해주어야 하기 때문에 128*4Byte 만큼 건너뛴다.

```
  ADD r11,r11,#1 ;r11=r11+1 (compare with 1080 later)
  CMP r11,r9 ;compare with 1080
```

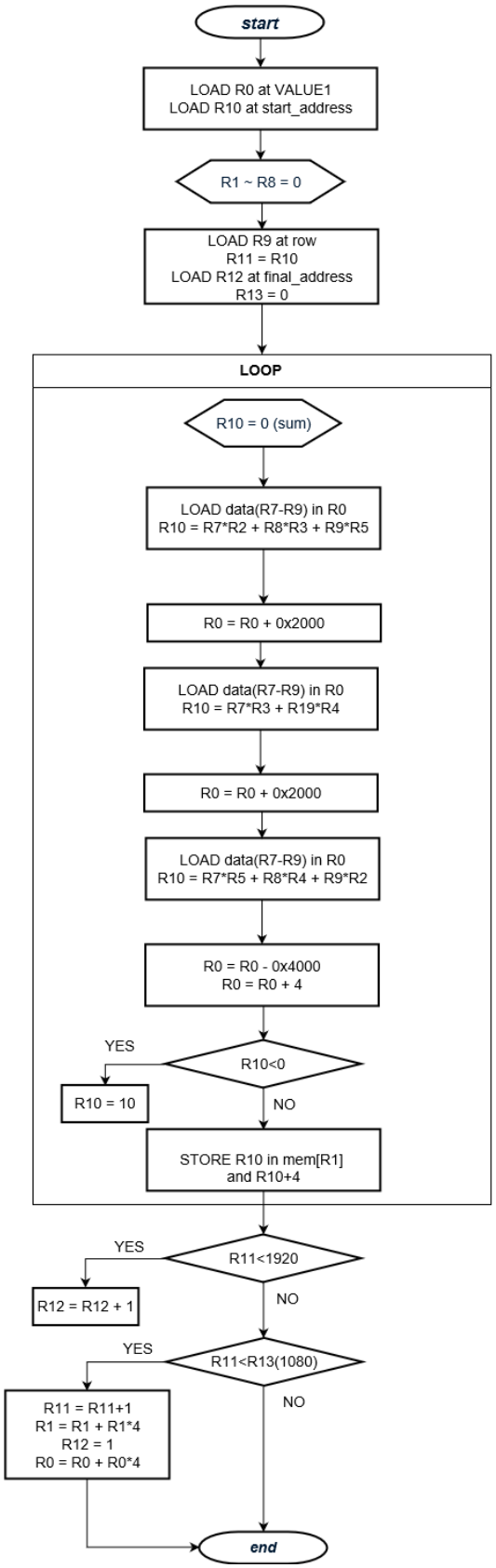
이제, r11 을 1 만큼 증가시키고, r11 과 1080 을 비교하고, 1080 보다 적거나 같으면 LOOP 를 돈다.

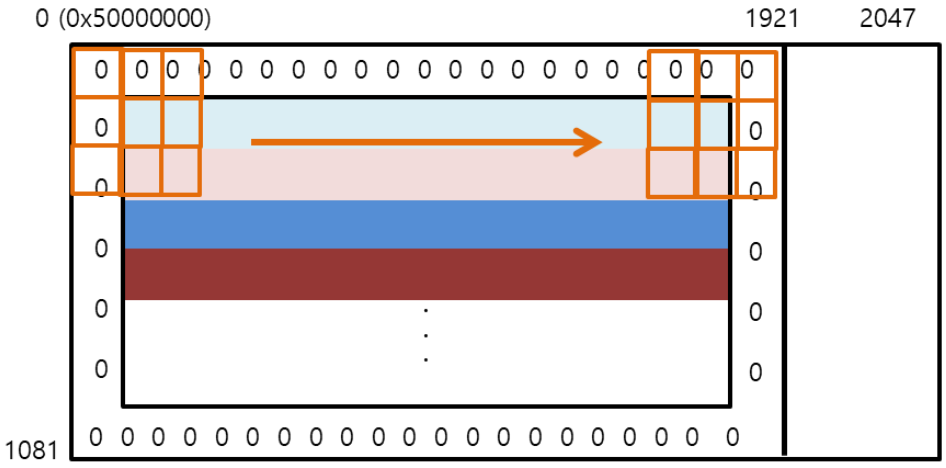
```
  BLE loop1

  BX lr
```

모두 끝났다면, main 함수로 돌아가게 된다.

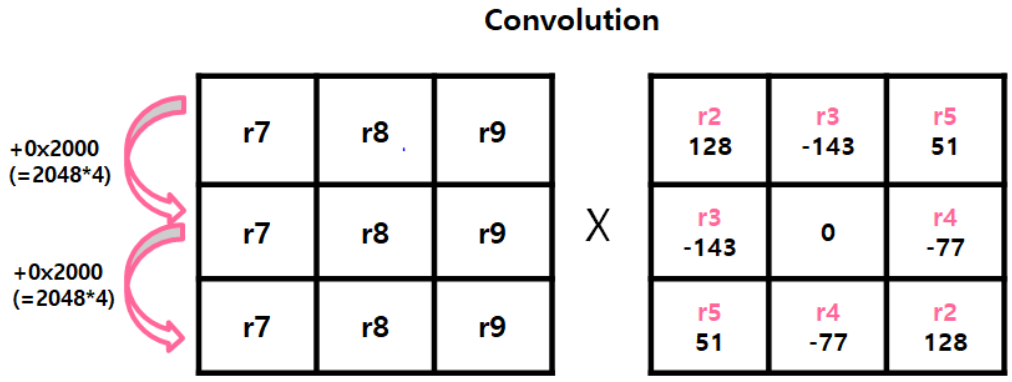
```
VALUE1 DCD   &40000000
row DCD &0000043A ;1082
save_address DCD &50002004
start_address DCD &50000004
final_address DCD &50872004
count1 DCD &281 ;1923/3=641
plus_address DCD &2000

  END
```

2-2-4 Convolution 함수





Relocation 의 assembly 코드 원리는 위의 그림과 같다. GrayScale 을 마친 배열은 0x50000000 에 저장된다. Convolution 을 할 때, 가장자리 셀에 대해서는 0 padding 처리를 해줘야 한다. 그래서 위의 grayscale code 에서 가장자리 부분에 0 padding 처리를 해준 것이다. 가장자리 패딩 처리를 미리 해주었기 때문에, Convolution code 에서는 연산만 반복적으로 하게 된다.



한 LOOP 에서 첫 번째 줄 3 개를 가져오고, w 배열의 첫 번째 줄 **값이 들어있는 레지스터**들과 연산, 두 번째 줄 3 개를 가져와서 w 배열의 두 번째 줄 값이 들어있는 레지스터들과 연산, 세 번째 줄 3 개를 갖고 와서 w 배열의 세 번째 줄 값이 들어있는 레지스터들과 각각 연산을 하게 된다. w 배열 값이 겹치는 것이 있다 보니 레지스터 4 개로 처리가 가능했다. 즉, register 에 w 배열에 들어있는 값들을 미리 저장해놓고 연산한다.

0.125	-0.14	0.05
-0.14	0	-0.075
0.05	-0.075	0.125

r2 128	r3 -143	r5 51
r3 -143	0	r4 -77
r5 51	r4 -77	r2 128

첫 번째 그림은 프로젝트에 주어진 W 배열이다. 두 번째 그림은 W 배열을 C code 및 assembly 에서 변형하여 쓴 것으로, 실제 W 배열 값에 1024 를 곱한 Integer 값이다. 우리가 짠 c 와 assembly code 는 1024 를 곱한 정수 w 배열로 Convolution 을 진행하고, 최종 결과를 1024 로 나누고 저장하는 형식이었다. 1000 을 곱하고 나눈 것보다 1024 가 훨씬 빠르게 진행되어서 1024 를 곱한 값을 쓰기로 하였다. (c, assembly 둘 다 1024 가 빨랐다.)

```
 AREA convolution, CODE, READONLY
      EXPORT Convolution

   ENTRY

Convolution

; r0 = gray [][]
; r1 = conv [][]
;
; w= {
;    {125,-140,50},
;   {-140,0,-75},
;   {50,-75,125}
;   }
;
;
; {128, -143, -77, 51} it is *1024!
; so, we use r2=128, r3=-143, r4=-77, r5=51
; r6 -> res
; r7, r8, r9 -> ldm register
; r10 : sum
; r11 : i
; r12 : j
; r13 : 1080

   LDR r0, VALUEG;
```

r0 는 grayscale 과 padding 처리를 마친 주소가 들어간다. gray[][] pointer 이다. 0x50000000 이 들어간다.

```
   LDR r1, VALUEC;
```

r1 은 convolution 한 결과를 저장할 conv[][] 주소가 들어간다. conv[][] pointer 이다. 0x60000000 이 들어간다.

```
   LDR r13, VALUEE;
```

r13 은 1080(=0x438)이 들어간다. 1080 은 상수로 표현이 되지 않기 때문에 레지스터에 저장해놓고 값을 비교하게 된다. LOOP 의 종단점을 찾기 위해 필요한 값이다.

```
MOV r2, #128;
MOV r3, #-143;
MOV r4, #-77;
MOV r5, #51;
```
W 의 값들이 겹치는 값들이 있다 보니, 중복된 값을 제거하고 나면 4 개의 레지스터만으로 W 배열을 표현할 수 있다. 각각의 값들을 r2-r5 에 저장한다.

```
MOV r11, #1;
MOV r12, #1;
```
r11 은 i(바깥쪽 루프), r12 는 j(안쪽 루프)를 의미한다. 이 어셈블리에서는 LOOP 가 하나 뿐이지만, 루프 하나로도 루프 두 개의 효과를 낸다.

LOOP
```
MOV r10, #0; sum=0
```
sum 을 먼저 0 으로 초기화시킨다.
```
LDM r0, {r7-r9};
```
padding 처리가 된 gray[][] 주소로 가서 4Byte 데이터 3 개를 가지고 온다. (auto-indexing 은 하지 않는다. 다음 줄의 데이터 3 개를 가지고 와야 하기 때문이다.)

```
MUL r6, r7, r2; no constant... no same register..
ADD r10, r10,r6;
MUL r6 ,r8, r3;
ADD r10, r10, r6;
MUL r6,r9, r5;
ADD r10, r10, r6;
```
{r7,r8,r9}X{128, -143, 51}(W 첫 번째 줄)를 sum 에다가 저장하는 코드이다.

```
ADD r0, r0, #0x2000;
```
다음 줄로 넘어가기 위해 0x2000(=2048*4)를 더한다.
```
LDM r0, {r7-r9};
MUL r6, r7, r3;
ADD r10, r10, r6;
; finally r8 * 0 is zero. so, we won't calculate
MUL r6 ,r9, r4;
ADD r10, r10, r6;
```
다음 줄의 데이터 세 개를 r7-r9 에 로드한 뒤, {r7,r8,r9}X{-143, 0, -77}(W 두 번째 줄)를 sum 에다가 저장한다. (auto-indexing 은 하지 않는다. 다음 줄의 데이터 3 개를 가지고 와야 하기 때문이다.) 여기서, 0 은 굳이 계산하지 않는다. 곱해도 0 이기 때문에 계산도, sum 에 저장도 하지 않는다.

```
ADD r0, r0, #0x2000;
```
다음 줄로 넘어가기 위해 0x2000(=2048*4)를 더한다.
```
LDM r0, {r7-r9};
```

```
MUL r6, r7, r5;
ADD r10, r10, r6;
MUL r6 ,r8, r4;
ADD r10, r10, r6;
MUL r6 ,r9, r2;
ADD r10, r10, r6;
```

마지막 줄의 데이터 세 개를 r7-r9 에 로드한 뒤, {r7,r8,r9}X{51, -77, 128}(W 세 번째 줄)를 sum 에다가 저장한다.

```
SUB r0, r0, #0x4000;
```

첫 번째 줄로 되돌아가기 위해 0x4000(=2048*4*2)를 빼준다.

```
ADD r0, r0, #4;
```

한 칸(4Byte)를 옆으로 옮긴다.

```
CMP r10, #0; sum <0
MOVLT r10, #0; sum=0
```

sum 이 0 보다 작은 경우, sum 에다 0 을 넣어준다.

```
; MOV r10, r10, LSR #10; sum=sum/1024 we don't do it
STR r10, [r1], #4 ; result stored at conv[][], post-index
```

sum 을 conv[][]에 저장한다. 저장하고 나서는 post-index 를 이용해 4Byte 증가시킨다. 원래는 sum 을 1024 로 나누고 그 값을 저장했으나, maxpooling 에서 저장하는 것이 효율적이라고 생각하여 1024 로 나누는 부분을 maxpooling 으로 옮겼다.

```
CMP r12, #1920; j vs 1920
ADDLT r12, r12, #1; j++
BLT LOOP;
```

r12(j)가 1920 보다 작으면 j 를 하나 더 늘리고 LOOP 를 다시 돈다. (총 1920*1080 픽셀에 대해 모두 convolution 처리를 해야 하기 때문에 j 는 1~1920 까지 가게 된다.)

```
CMP r11, r13; i vs 1080
ADDLT r11, r11, #1; i++
ADDLT r1, r1, #128<<2 ; no padding
MOVLT r12, #1; j make 1
ADDLT r0, r0, #128<<2 ; yes padding
BLT LOOP;
```

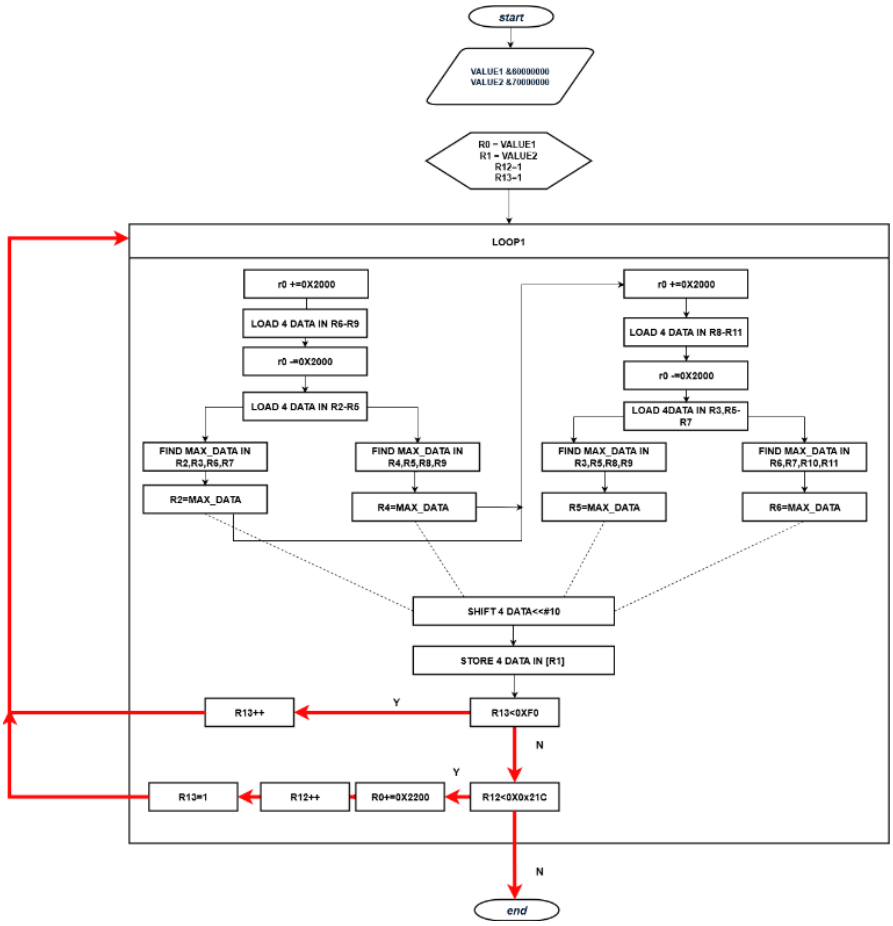
r11(i)가 1080 보다 작으면 i 를 하나 더 늘리고 LOOP 를 돌게 되는데, 이 경우 줄이 하나 바뀌어야 하므로 다음 줄로 넘어가기 위해 돌기 전에 j 를 1 로 만들고, gray pointer (r0)와 conv pointer(r1)를 128*4Byte 만큼 건너뛰도록 한다. grayscale 후 결과인 gray[][]도 연속적인 메모리에 저장이 된 것이 아니기 때문에 건너뛰어주어야 하고, conv[][] 또한 계속해서 연속적인 메모리에 할당하지 않으므로 건너뛰어야 한다.

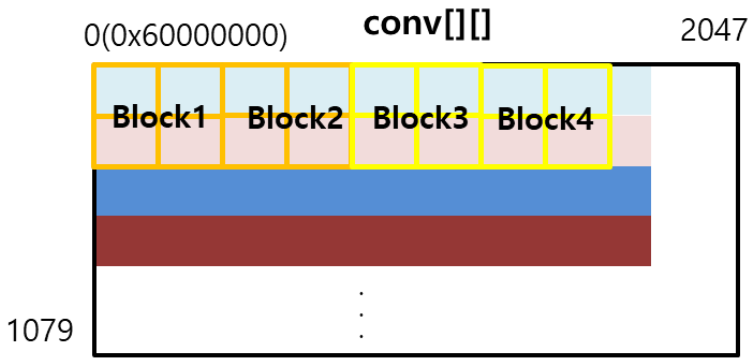
```
BX LR;
```

모든 코드가 끝이 나면, main 함수로 돌아가게 된다.

```
VALUEG   DCD 0x50000000
VALUEC   DCD 0x60000000
VALUEE   DCD 0x438 ;1080
   END
```

2-2-5 Max Pooling 함수





MaxPooling 의 assembly 코드 원리는 위의 그림과 같다. convolution 결과를 저장한 conv[][]에서 윗줄 두 개, 아랫줄 두 개로 이루어진 Block 에서 max 값 하나를 찾아 result[][]에 저장한다. 이 코드에서는 한 LOOP 에 **block 4 개**를 계산하여 결과 4 개를 한번에 Store 한다.

r2	r3	r4	r5	r3	r5	r6	r7
r6	r7	r8	r9	r8	r9	r10	r11

그림과 같이 윗줄 4 개, 아랫줄 4 개를 한꺼번에 로드하여 계산하고(총 block 2 개의 분량), 그 다음 윗줄 4 개, 아랫줄 4 개를 한꺼번에 로드하여 계산하는 식이다.

```
  AREA    maxpooling, CODE, READONLY

    EXPORT MaxPooling

  ENTRY

MaxPooling

; r0 = conv address
; r1 = result address
; {r2-r5}
; {r6-r9}, r2 max, r4 max

; {r3, r5, r6, r7}
; {r8-r11}, r5 max, r6 max
; r12 i
; r13 j


  LDR r0,VALUE1 ; conv
  LDR r1, VALUE2 ; result
```

r0 은 convolution 결과를 저장한 conv[][]의 pointer 역할을 한다. 0x60000000 의 값이 저장된다. r1 은 maxpooling 후 최종 결과를 저장할 result[][]의 pointer 역할을 한다. 0x70000000 의 값이 저장된다.

```
  MOV r12, #1;
  MOV r13, #1;
```

r12 는 i 의 역할을, r13 은 j 의 역할을 한다. 각각을 1 로 초기화시킨다.

```
LOOP1
  ;first, second block
  ADD r0, r0, #0x2000;
  LDMIA r0,{r6-r9}
```

먼저 r0(conv[][])에 0x2000(=2048*4)를 더한다. 이러면 다음 줄로 넘어가게 되는데, 이 때 r6-r9 에 다음 줄의 데이터 4 개를 가지고 온다. (이 때는 auto-indexing 을 하지 않는다.)

```
  SUB r0, r0, #0x2000;
  LDMIA r0!, {r2-r5};
```

다시 r0 에 0x2000 을 빼고 원래 위치로 돌아와 **auto-indexing** 을 하며 r2-r5 에 4 개의 데이터를 가지고 온다. auto-indexing 을 해주었으므로 다음 블록으로 포인터가 넘어가있을 것이다. 이렇게 첫 번째 블록과 두 번째 블록의 데이터 총 8 개를 모두 가지고 왔다.

```
;first block
CMP r2, r3;
MOVLT r2, r3; r2<r3, then make r2 big thing
```

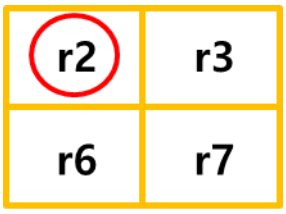
r2, r3 를 비교하여 만약 r2 가 작다면 r3 를 r2 에 저장한다. r2 에는 r2, r3 중 큰 값이 저장되어있게 된다.

```
CMP r6, r7;
MOVLT r6, r7; r6<r7, then make r6 big thing
```

r6, r7 을 비교하여 만약 r6 이 작다면 r7 을 r6 에 저장한다. r6 에는 r6, r7 중 큰 값이 저장되어있게 된다.

```
CMP r2, r6;
MOVLT r2, r6; first block max stored at r2
```

r2 에는 r2, r3 중 큰 값이, r6 에는 r6,r7 중 큰 값이 들어있게 된다. r2, r6 를 비교하면 첫 번째 블록에서 가장 큰 값이 나오게 된다. r2 가 작다면, r6 를 r2 에 저장한다. 첫 번째 블록에서 가장 큰 값은 r2 에 저장되게 된다.



첫 번째 블록에서 가장 큰 값은 r2 에 저장된다.

```
;second block
CMP r4, r5;
MOVLT r4, r5; r4<r5, then make r4 big thing
```

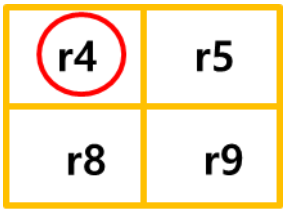
r4, r5 를 비교하여 만약 r4 가 작다면 r5 를 r4 에 저장한다. r4 에는 r4, r5 중 큰 값이 저장되어있게 된다.

```
CMP r8, r9;
MOVLT r8, r9; r8<r9, then make r8 big thing
```

r8, r9 를 비교하여 만약 r8 이 작다면 r9 를 r8 에 저장한다. r8 에는 r8, r9 중 큰 값이 저장되어있게 된다.

```
CMP r4, r8;
MOVLT r4, r8; second block max stored at r4
```

r4 에는 r4, r5 중 큰 값이, r8 에는 r8, r9 중 큰 값이 들어있게 된다. r4, r8 를 비교하면 두 번째 블록에서 가장 큰 값이 나오게 된다. r4 가 작다면, r8 을 r4 에 저장한다. 두 번째 블록에서 가장 큰 값은 r4 에 저장되게 된다.



두 번째 블록에서 가장 큰 값은 r4 에 저장된다.

```
;thrid, fourth block
ADD r0, r0, #0x2000; 2048*4!
LDMIA r0, {r8-r11}; is it right?
SUB r0, r0, #0x2000;
LDMIA r0!, {r3, r5-r7};
```

마찬가지로 r0 에 0x2000 을 더하여 r8-r11 에 4 개의 데이터를 가지고 오고, r0 에 0x2000 을 빼고 원래 위치로 돌아와 **auto-indexing** 을 하며 r3, r5-r7 에 4 개의 데이터를 가지고 온다. auto-indexing 을 해주었으므로 다음 블록으로 포인터가 넘어가있을 것이다. 이렇게 세 번째 블록과 네 번째 블록의 데이터 총 8 개를 모두 가지고 왔다. (r2, r4 은 첫 번째, 두 번째 블록의 max 값이 들어있으므로 사용하지 않는다.)

```
;thrid block
CMP r3, r5;
MOVGE r5, r3; *r5<r3, then make r5 big thing
```

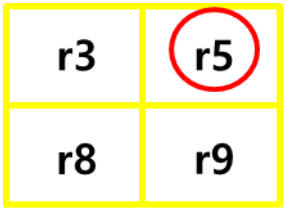
r3, r5 를 비교하는데, 여기서는 조금 다르게 만약 **r3 이 크다면 r3 를 r5 에 저장**한다. r5 에는 r3, r5 중 큰 값이 저장되어있게 된다. 여기서 r5 에 큰 값을 저장하는 이유는, STM 을 할 때, 레지스터가 큰 것이 큰 주소에 들어가기 때문이다. 그러므로 r4 보다 큰 r5 에 저장할 수 밖에 없었다.

```
CMP r8, r9;
MOVLT r8, r9; r8<r9, then make r8 big thing
```

r8 에는 r8, r9 중 큰 값이 들어가있게 된다.

```
CMP r5, r8;
MOVLT r5, r8; first block max stored at r5
```

r5 에는 r3, r5 중 큰 값이 들어있고, r8 에는 r8, r9 중 큰 값이 들어있다. r5 가 작다면, r8 을 r5 에 넣는다. r5 는 세 번째 블록에서 가장 큰 값이 들어있게 된다.

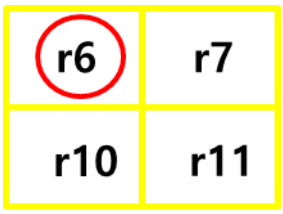


세 번째 블록에서 가장 큰 값은 r5 에 저장된다.

```
;fourth block
CMP r6, r7;
MOVLT r6, r7; r4<r5, then make r6 big thing
```

```
CMP r10, r11;
MOVLT r10, r11; r10<r11, then make r10 big thing

CMP r6, r10;
MOVLT r6, r10; second block max stored at r6
```

똑같이 네 번째 블록에 r6, r7, r10, r11 을 비교하여 가장 큰 값을 r6 에 저장한다.



네 번째 블록에서 가장 큰 값은 r4 에 저장된다.

```
MOV r2, r2, LSR #10; sum=sum/1024
MOV r4, r4, LSR #10; sum=sum/1024
MOV r5, r5, LSR #10; sum=sum/1024
MOV r6, r6, LSR #10; sum=sum/1024
```

r2, r3, r4, r5 (각각 첫 번째, 두 번째, 세 번째, 네 번째 블록의 max 값)을 1024 로 나눈다. convolution code 에서 W 가 본래 w 배열에 1024 를 곱해 나온 정수였기 때문에, 이 곳에서 1024 로 나누게 된다.

```
STMIA r1!, {r2, r4-r6}; stored at result[][]
```

그리고 그 값들을 result 배열에 저장한다. 이 때, auto-indexing 을 이용한다. 최종 결과인 **result 배열은 연속적인 메모리로 저장**되게 된다. 결과를 viewer 로 확인하기 위해서는 연속적인 메모리에 저장되어 있어야 하기 때문이다.

```
CMP r13, #0xF0; r13(j) compare 1920/8=240
ADDLT r13, r13, #1; j++
BLT LOOP1;
```

r13 (j)를 240 과 비교한다. 한 줄에서 4Byte 데이터를 8 개씩 가져오기 때문에, 한 줄당 240 번 돌면 되기 때문이다. 만약 240 보다 작다면, r13 을 1 증가시키고 LOOP 로 돌아간다.

```
;if r13==240, then i++
CMP r12, #0x21C; r12(i) compare 1080/2=540
ADDLT r0, r0, #0x2200; 128*4 + 2048*4
ADDLT r12, r12, #1;
MOVLT r13, #1;
BLT LOOP1;
```

한 줄 Maxpooling 을 끝냈다면, 줄을 옮길 차례다. conv[][]가 연속적인 메모리에 저장되어있지 않기 때문에 공백을 뛰어넘기 위해 128*4 를 더해야 한다. 거기다, maxpooling 은 두 줄에 대해서 처리하기 때문에

2048*4 (한 줄)을 더 뛰어넘어야 한다. 총 128*4+2048*4 에 해당하는 0x2200 을 conv pointer(r0)에 더하게 된다. 또한 i 에 해당하는 r12 도 하나 증가시키고, j 에 해당하는 r13 은 1 로 초기화시킨 뒤 LOOP 를 다시 돌게 된다. 만약, 두 줄씩 540 번을 실행해 r12 (=i)가 540 이 된다면, 1080 줄을 돌았다는 뜻이므로 모든 블록에 대한 Maxpooling 처리가 끝나게 된다.

Maxpooling 의 최종 결과는 **960*540***4Byte 의 크기를 가지게 된다.

```
 BX LR;
```

만약 모든 처리가 끝났으면, main 함수로 돌아간다.

```
VALUE1 DCD      &60000000 ; conv[][] address
VALUE2 DCD      &70000000 ; result[][] address

  END
```

3. 검증

3-1 순수 C 코드

3-1-1 Loadimage : 이미지 저장된 주소 가져오는 함수

```
void loadImage(int (*imghex)[1920]) {
int * a;
int i=0;
int j;
a=0x40000000;
for(i=0; i<1080;i++) {
    for(j=0; j<1920;j++) {
        imghex[i][j]=(*a);
        a++;
        }
  }
}
```

3-1-2 Grayscale : RGB 파일을 grayscale 로 바꿔주는 함수

1) 초안

```
void Grayscale(char a[1080][7680],int b[1080][1920]) {
    int i,j,k;
    int max, min;
    for (i = 0; i < 1080; i++) {
        for (j = 0; j < 1920; j++) {
            max = 0;
            min = 255;
            for (k = 0; k < 2; k++) {
                max = Max(a[i][4 * j + k], a[i][4 * j + k + 1], max);
                min = Min(a[i][4 * j + k], a[i][4 * j + k + 1], min);
                b[i][j] = (max + min) / 2;
                }
            }
      }
}
int Max(int a, int b, int max) {
    if (a >= b) {
      if (a > max) {
        max = a;
      }
    }
    else {
```

```
        if (b > max) {
            max = b;
        }
    }
    return max;
}
int Min(int a, int b, int min) {
    if (a <= b) {
        if (a < min) {
            min = a;
        }
    }
    else {
        if (b < min) {
            min = b;
        }
    }
    return min;
}
```

2) Grayscale+Padding : convolution 에서 padding 을 진행하면서 다시 값을 가져와서 저장하는 것 보다 grayscale 단계에서 L 값을 저장 할 때 padding 을 하는 것이 더 효율적일 것이라 판단하여 padding 을 추가하였다.

Padding

① 방법 1 : 1082x1922 배열 선언 할 때 전체를 0 으로 초기화 하기

```
int imghex[1082][1922]={0,};
```

② 방법 2 : 1082x1922 배열의 가장자리만 0을 넣기

```
for(i=0; i<1922 ;i++)
        imghex[0][i]=0;
    for(i=0; i<1922 ;i++)
        imghex[1081][i]=0;
    for(i=0; i<1082 ;i++)
        imghex[i][0]=0;
    for(i=0; i<1082 ;i++)
        imghex[i][1921]=0;
```

3) Little Endian 고려함 : Little Endian 때문에 RGBA 가 아닌 ABGR 순으로 저장되는 것을 고려하여 Max, Min 에 들어가는 파라미터를 수정했다

```
max = Max(a[i][4 * j + k+1], a[i][4 * j + k + 2], max);
min = Min(a[i][4 * j + k+1], a[i][4 * j + k + 2], min);
```

3-1-3 Convolution

1) 초안 : 3 시간이 지나도 실행 완료가 되지 않았다.

```
void convolution(int b[1082][1922],int d[1080][1920]) {
	int i, j, k, l; //b array is grayscale array
	double res, sum;
	double w[3][3] = { { 0.125, -0.14, 0.05 },{ -0.14, 0, -0.075 },{ 0.05, -0.075, 0.125 } };
	//convolution
	for (i = 0; i<1080; i++) {
		for (j = 0; j<1920; j++) {
			sum = 0;
			for (k = 0; k<3; k++) {
				for (l = 0; l<3; l++) {
					res = w[k][l] * b[k + i][l + j];
					sum += res;
				}
			}
			if (sum < 0) sum = 0;
			d[i][j] = sum;
		}
		int num=1;
		printDecimal(1);
		if(num==100) { sendchar('₩n');}
		 num=num+1;
	}
}
```

2) W double->int : w 값들에 1000 을 곱하여 int 형으로 만들어서 convolution 을 계산하고 마지막에 sum 을 결과 배열에 저장할 때 1000 을 나눠줬다. 시간이 비약적으로 줄었다.

```
int i, j, k, l, res, sum;
int w[3][3]={ {125,-140,50}, {-140,0,-75}, {50,-75,125} };
d[i][j] = sum/1000;
```

3-1-4 Maxpooling

```
void maxpooling(int (*a)[1920], int (*b)[960]) {
	int ii = 0, jj = 0;
	int i = 0, j = 0;
	for (i = 0; i < 1080; i = i + 2) {
		jj = 0;
		for (j = 0; j < 1920; j = j + 2) {
			b[ii][jj] = max(a[i][j], a[i][j + 1], a[i + 1][j], a[i + 1][j + 1]);
			jj++;
		}
		ii++;
	}
}

int max(int a, int b, int c, int d) {
	int ab;
	int cd;
	if (a > b) { ab = a; }
	else { ab = b; }
	if (c > d) { cd = c; }
	else { cd = d; }
	if (ab > cd)
	return ab;
	else return cd;
}
```

3-1-5 최종 코드 및 결과 시간

1) grayscale, maxpooling 의 max 함수 통합 : grayscale 의 함수를 통해 maxpooling 의 최댓값 계산을 했다.

```
#include "base.h"
#define bSizeRow 1080+2
#define bSizeColumn 1920+2
#define dSizeRow 1080*4
#define dSizeColumn 1920*4

void loadImage(int (*imghex)[1920]);
void Grayscale(char a[1080][7680],int b[1082][1922]);
int Max(int a, int b, int max);
int Min(int a, int b, int max);
void convolution(int b[1082][1922],int d[1080][1920]);
void maxpooling(int (*a)[1920], int (*b)[960]);

int main(void) {
```

```
	int init[1080][1920];
	int b[1082][1922];
	int d[1080][1920];
	int result[540][960];
	loadImage(init);
	Grayscale ((char (*) [7680])init, b);
	convolution(b,d);
	maxpooling(d,result);
	_sys_exit(0);
}

void loadImage(int (*imghex)[1920]){
	int * a;
	int i=0;
	int j;
	a=0x40000000
	for(i=0; i<1080;i++){
		for(j=0; j<1920;j++){
		imghex[i][j]=(*a);
		a++;
		}
	}
}

void Grayscale(char a[1080][7680],int b[1082][1922]){
	int i,j,k;
	int max, min;
	for(i=0; i<1922 ;i++) b[0][i]=0;
	for(i=0; i<1922 ;i++) b[1081][i]=0;
	for(i=0; i<1082 ;i++) b[i][0]=0;
	for(i=0; i<1082 ;i++) b[i][1921]=0;
	for (i = 0; i < 1080; i++) {
		for (j = 0; j < 1920; j++) {
			max = 0;
			min = 255;
			for (k = 0; k < 2; k++) {
				max = Max(a[i][4 * j + k+1], a[i][4 * j + k + 2], max);
				min = Min(a[i][4 * j + k+1], a[i][4 * j + k + 2], min);
				b[i+1][j+1] = (max + min) / 2;
			}
		}
	}
}

int Max(int a, int b, int max) {
	if (a >= b) {
		if (a > max) { max = a; }
```

```
	}
	else {
		if (b > max) { max = b; }
	}
	return max;
}

int Min(int a, int b, int min) {
	if (a <= b)
	{
		if (a < min) { min = a; }
	}
	else {
		if (b < min) { min = b; }
	}
	return min;
}

void convolution(int b[1082][1922], int (*d)[1920]) {
	int i, j, k, l, res, sum;
	int w[3][3]={{125,-140,50},{-140,0,-75},{50,-75,125}};
	for (i = 0; i<1080; i++) {
		for (j = 0; j<1920; j++) {
			sum = 0;
			for (k = 0; k<3; k++) {
				for (l = 0; l<3; l++) {
					res = w[k][l] * b[k + i][l + j];
					sum += res;
				}
			}
			if (sum < 0)
			sum = 0;
			d[i][j] = sum/1000;
		}
	}
}

void maxpooling(int (*a)[1920], int (*b)[960]) {
	int ii = 0, jj = 0;
	int i = 0, j = 0;
	for (i = 0; i < 1080; i = i + 2) {
		jj = 0;
		for (j = 0; j < 1920; j = j + 2) {
			int max=0;
			max=Max(a[i][j],a[i][j+1],max);
			max=Max(a[i+1][j],a[i+1][j+1],max);
			b[ii][jj] = max;
```

```
            jj++;
        }
        ii++;
    }
}
```

Module/Function	Calls	Time(Sec)	Time(%)
pr2		32.003 s	100%

2) grayscale, maxpooling 의 max 함수 미통합 : 각자의 max 함수를 이용하여 max 값을 계산하였다. (각 함수를 따로 만들었을 때의 코드를 그대로 붙였다.)

Module/Function	Calls	Time(Sec)	Time(%)
pr2		31.988 s	100%

3) grayscale, maxpooling 의 max 함수 미통합 : grayscale 의 max, min 함수를 2 개중 최대, 최소를 찾는 것에서 3 개중 최대,최소를 찾는 것으로 바꾸었다.

```
int Max(int a, int b, int c) {
    int max;
    if (a >= b) { max = a; }
    else { max = b; }
    if (c>max){ max=c; }
    return max;
}
int Min(int a, int b, int c) {
    int min=255;
    If (a <= b) { min = a; }
    else { min = b; }
    if(c<min){ min=c; }
    return min;
}
```

Grayscale 내에서 max,min 함수 호출할 때,

```
max = Max(a[i][4 * j + 1], a[i][4 * j + 2],a[i][4*j+3]);
min = Min(a[i][4 * j + 1], a[i][4 * j + 2],a[i][4*j+3]);
```

Module/Function	Calls	Time(Sec)	Time(%)
pr2		28.567 s	100%

3) max, min 함수 따로 만들지 않고 grayscale, maxpooling 내에서 최대,최솟값 구하기

: max, min 함수들을 재사용할 일이 없어 함수 내에서 진행했다. 변수들을 통일시키고 정리했다.

```
#include "base.h"
#define rowSize 1080
#define ColumnSize 1920
#define resRowSize 540
#define resColumnSize 960

void loadImage(int (*imghex)[ColumnSize]);
void Grayscale(char (*origin)[ColumnSize*4],int gray[rowSize+2][ColumnSize+2]);
void convolution(int gray[rowSize+2][ColumnSize+2],int conv[rowSize][ColumnSize]);
void maxpooling(int (*conv)[ColumnSize], int (*result)[resColumnSize]);
int maxFormaxpooling(int a, int b, int c, int d);

int main(void) {
	int init[rowSize][ColumnSize];
	int gray[rowSize+2][ColumnSize+2];
	int conv[rowSize][ColumnSize];
	int result[resRowSize][resColumnSize];
	loadImage(init);
	Grayscale ((char (*) [ColumnSize*4])init, gray);
	convolution(gray,conv);
	maxpooling(conv,result);
	_sys_exit(0);
}

void loadImage(int (*imghex)[ColumnSize]) {
	int *origin;
	int i,j;
	origin=0x40000000;
	for(i=0; i<rowSize ;i++){
		for(j=0; j<ColumnSize ;j++){
			imghex[i][j]=(*origin);
			origin++;
		}
	}
}

void Grayscale(char (*origin)[ColumnSize*4],int gray[rowSize+2][ColumnSize+2]) {
	int i, j, max, min, v1, v2, v3;
  //padding for convolution
	for(i=0; i<ColumnSize+2; i++)  gray[0][i]=0;
	for(i=0; i<ColumnSize+2; i++)  gray[rowSize+1][i]=0;
	for(i=0; i<rowSize+2; i++)  gray[i][0]=0;
	for(i=0; i<rowSize+2; i++)  gray[i][ColumnSize+1]=0;
```

```
   //Save L(grayscale)
      for (i = 0; i < rowSize; i++) {
            for (j = 0; j < ColumnSize; j++) {
                  v1=origin[i][4 * j + 1];
                  v2=origin[i][4 * j + 2];
                  v3=origin[i][4*j+3];
                  if(v1>v2) max=v1;
                  else max=v2;
                  if(max<v3) max=v3;
                  if(v1<v2) min=v1;
                  else min=v2;
                  if(v3<min) min=v3;
                  gray[i+1][j+1] = (max + min) / 2;
            }
      }
}

void convolution(int gray[rowSize+2][ColumnSize+2], int (*conv)[ColumnSize]) {
      int i, j, k, l, res, sum;
   // w array is float type but we change int type for quick speed
      int w[3][3]={{125,-140,50}, {-140,0,-75}, {50,-75,125}};
   //convolution
      for (i = 0; i<rowSize; i++) {
            for (j = 0; j<ColumnSize; j++) {
                  sum = 0;
                  for (k = 0; k<3; k++) {
                        for (l = 0; l<3; l++) {
                              res = w[k][l] * gray[k + i][l + j];
                              sum += res;
                        }
                  }
            if (sum < 0) sum = 0;
            conv[i][j] = sum/1000;
            }
      }
}

void maxpooling(int (*conv)[ColumnSize], int (*result)[resColumnSize]) {
      int ii = 0, jj;
      int i, j, v1, v2, v3, v4, ab, cd;
      for (i = 0; i < rowSize; i = i + 2) {
            jj = 0;
            for (j = 0; j < ColumnSize; j = j + 2) {
                  v1=conv[i][j];
                  v2=conv[i][j + 1];
                  v3=conv[i + 1][j];
                  v4=conv[i + 1][j + 1];
```

```
                if (v1 > v2) ab = v1;
                else ab = v2;
                if (v3 > v4) cd = v3;
                else cd = v4;
                if (ab > cd) result[ii][jj]=ab;
                else result[ii][jj]=cd;
                jj++;
            }
        ii++;
        }
}
```

Module/Function	Calls	Time(Sec)	Time(%)
pr2		26.903 s	99%

위의 3 개 방법 중 3 번째 방법이 약 27 초로 가장 빠르게 실행됐다.

Module/Function	Calls	Time(Sec)	Time(%)
pr2		5.385 s	99%

.uvopt 파일 등을 포함한 불필요한 파일들을 삭제하고 실행해 보았더니 실행시간이 약 27 초에서 약 5.3 초로 감소하였다. 이유는 잘 모르겠으나 실행시간이 현저하게 감소하여 메모리에 들어간 값을 다 확인해 보았는데 잘 들어가 있었고 결과도 파일을 지우기 전과 같았다.

4) 1000 으로 나누는 것을 convolution 이 아닌 maxpooling 에서 하기

: convolution 에서 값 저장할 때 1000 으로 나누는 것 보다 maxpooling 이후 값을 저장할 때 1000 으로 나누는 것이 연산 횟수가 적을 것이라 판단하였다.

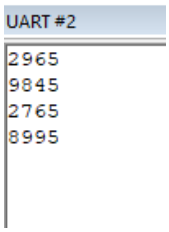
Convolution 함수에서 conv[i][j] = sum/1000 -> conv[i][j]=sum 으로 바꿨고

Maxpooling 함수에서 if (ab > cd) result[ii][jj]=ab; -> if(ab>cd) result[ii][jj]=ab/1000;

else result[ii][jj]=cd; -> else result[ii][jj]=cd/1000;으로 바꿨다..

Module/Function	Calls	Time(Sec)	Time(%)
pr2		3.087 s	100%

실행시간이 3.087 초로 많이 빨라졌다.



maxpooling 에서도, convolution 에서도 1000 으로 나누지 않았을 때 결과 창

이 결과를 보면 계산이 잘 되고 있다는 것을 볼 수 있다.

차례대로 2.965, 9.845, 2.765, 8.995 이다.

5) W 를 1000 이 아닌 1024 를 곱하여 결과값 저장할 때 1024 로 나누기

: 1000 을 곱하고 나누는 연산보다 1024 를 곱하고 나누는 것이 실행 속도가 빠를 것이라 판단하였다.
w[3][3]={{128,-143,51}, {-143,0,-77}, {51,-77,128}}로 계산한다.

Maxpooling 함수에서 if (ab > cd) result[ii][jj]=ab/1000; -> if(ab>cd) result[ii][jj]=ab/1024;

else result[ii][jj]=cd/1000; -> else result[ii][jj]=cd/1024; 로 바꿨다..

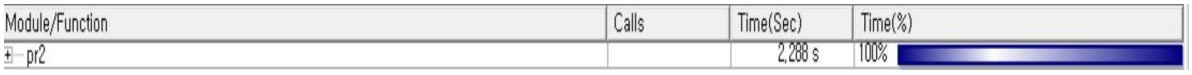
Printdecimal 을 포함하지 않은 code 에서는

실행시간이 2.284 초이고

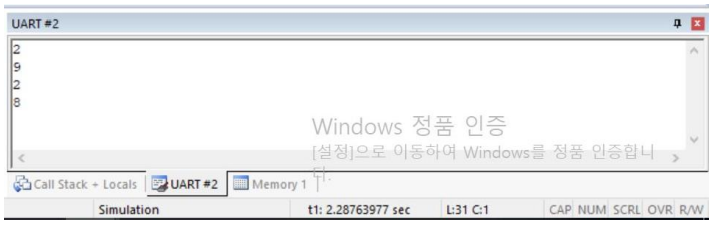
Printdecimal 로 출력 하는 code 는

```
printDecimal(result[300][100]); sendchar('₩n');
printDecimal(result[300][300]); sendchar('₩n');
printDecimal(result[400][300]); sendchar('₩n');
printDecimal(result[300][800]);
```

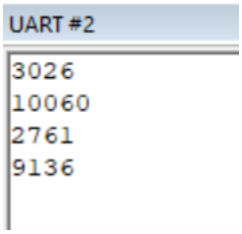
를 코드 마지막에 추가시켜주면 된다.



이 코드의 실행시간은 2.288 초였다.



printdecimal 로 출력한 결과이다.



maxpooling 에서도, convolution 에서도 1024 로 나누지 않았을 때 결과 창

W 에 1024 를 곱하면서 소수부분을 반올림 시키면서 약간의 오차가 발생하였으나 결과 사진을 보는 데에는 문제가 없다.

차례대로 2.955, 9.824, 2.696, 8.922 이다.

6) 2.965 출력을 위한 코드

```
void maxpooling(int (*conv)[ColumnSize], int (*result)[resColumnSize],int (*result2)[resColumnSize]);
int main(void) {
		int init[rowSize][ColumnSize];
		int gray[rowSize+2][ColumnSize+2];
		int conv[rowSize][ColumnSize];
		int result[resRowSize][resColumnSize];
		int result2[resRowSize][resColumnSize];
		loadImage(init);
		Grayscale ((char (*) [ColumnSize*4])init, gray);
		convolution(gray,conv);
		maxpooling(conv,result,result2);
		 int a;
		  printDecimal(result[300][100]);
		 sendchar('.');
		 a=result2[300][100]-result[300][100]*1000;
		 printDecimal(a);
		 sendchar('₩n');

		  printDecimal(result[300][300]);
		  sendchar('.');
		 a=result2[300][300]-result[300][300]*1000;
		printDecimal(a);
		sendchar('₩n');

		 printDecimal(result[400][300]);
		  sendchar('.');
		 a=result2[400][300]-result[400][300]*1000;
		  printDecimal(a);
		 sendchar('₩n');

		printDecimal(result[300][800]);
		  sendchar('.');
		 a=result2[300][800]-result[300][800]*1000;
```

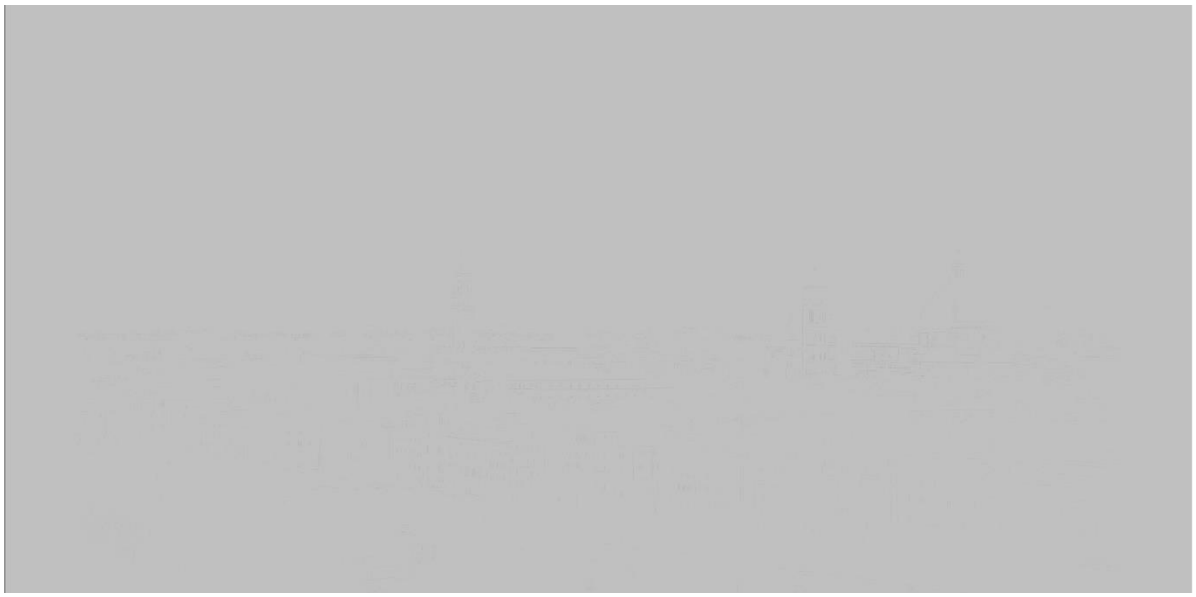
```
            printDecimal(a);
              sendchar('₩n');

        _sys_exit(0);
}
void maxpooling(int (*conv)[ColumnSize], int (*result)[resColumnSize],int (*result2)[resColumnSize])
 {
        int ii = 0, jj;
        int i, j, v1, v2, v3, v4, ab, cd;
        for (i = 0; i < rowSize; i = i + 2) {
                jj = 0;
                for (j = 0; j < ColumnSize; j = j + 2) {
                        v1=conv[i][j];
                        v2=conv[i][j + 1];
                        v3=conv[i + 1][j];
                        v4=conv[i + 1][j + 1];
                        if (v1 > v2) ab = v1;
                        else ab = v2;
                        if (v3 > v4) cd = v3;
                        else cd = v4;
                        if (ab > cd)
                        {
                          result[ii][jj]=ab/1000;
                                result2[ii][jj]=ab;
                        }
                        else
                        {
                                 result[ii][jj]=cd/1000;
                                result2[ii][jj]=cd;
                        }
                        jj++;
                }
                ii++;
        }
}
```

결과

```
UART #2
2.965
9.845
2.765
8.995
```

3-1-6 결과 이미지 출력

1) Convolution 한 이후 출력



2) maxpooling까지 한 최종 출력

3) w[3][3]={{125,-140,150}, {-140,0,-75}, {50,-75,125}} 일 때 결과 이미지

: 처음 코드 짤 때, {0.125,-0.14,0.05}를 {0.125,-0.14,0.15}로 봐서 잘못 실행했을 때의 결과 이미지이다.

① grayscale로 바꾼 후 출력

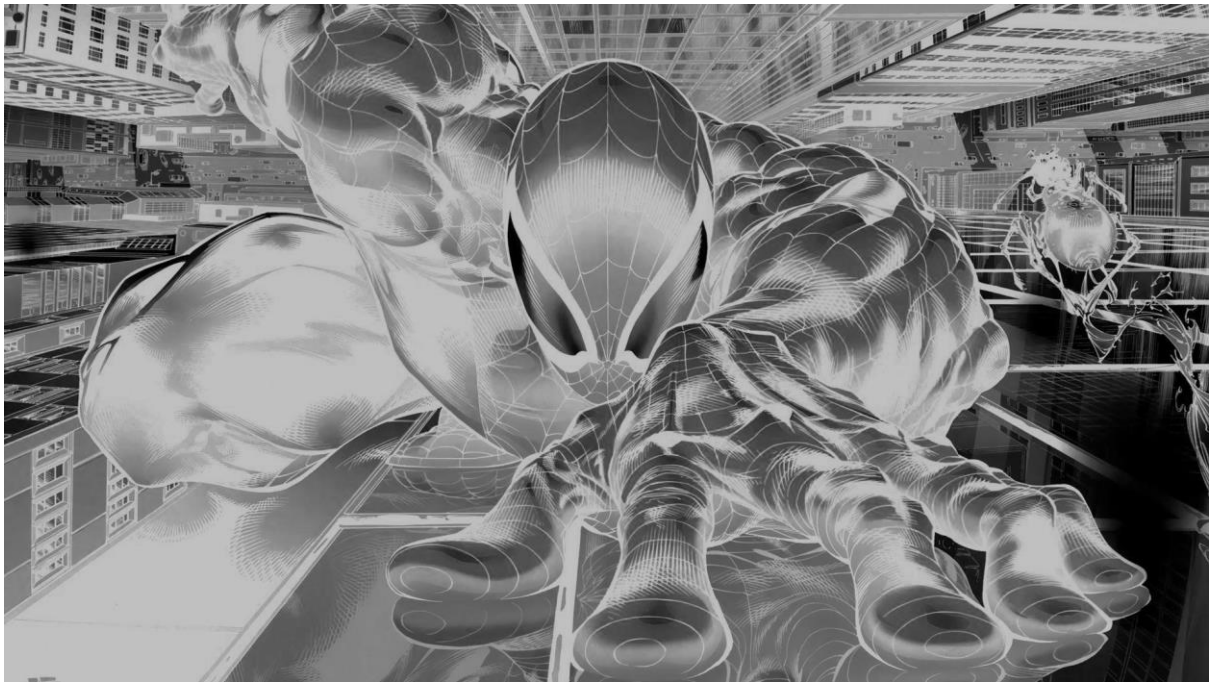
② convolution 한 이후 출력

③ maxpooling 까지 한 최종 출력

3-1-7 다른 사진 넣었을 때

1) 원본

2) grayscale로 바꾼 후 출력



3) maxpooling까지 한 최종 출력

4) w[3][3]={{125,-140,150}, {-140,0,-75}, {50,-75,125}} 일 때 결과 이미지

: w[3][3]={{125,-140,50}, {-140,0,-75}, {50,-75,125}}일 때 결과 이미지가 잘 보이지 않아서 w[3][3]={{125,-140,150}, {-140,0,-75}, {50,-75,125}}로 바꿔서 결과가 더 선명하게 나오게 하여 보았다. 다른 이미지에서도 잘 동작하는 것을 확인 할 수 있었다.

3-2. 어셈블리 코드

3-2-1 main

```
#include "base.h"
extern void Relocation(void);
extern void Grayscale(void);
extern void Convolution(void);
extern void MaxPooling(void);
int main(void)
{
	Relocation();
	Grayscale();
	Convolution();
	MaxPooling();
	int * result;
	result=0x70119590;  printDecimal((*result)); sendchar('₩n');
	result=0x701198B0;  printDecimal((*result)); sendchar('₩n');
	result=0x701774B0;  printDecimal((*result)); sendchar('₩n');
	result=0x7011A080;  printDecimal((*result));
	_sys_exit(0);
}
```

3-2-2 relocation

1) 초안

```
	AREA memRelocation, CODE, READONLY
	EXPORT Relocation
	ENTRY
Relocation
	LDR R0, Image_Address
	LDR R1, New_Address
	LDR R2, last_Address
	MOV R8, #128<<2

LOOPB
	MOV R7, #0
	CMP R0,R2
	BLE endl
LOOP
  LDMDB R0!,{R3-R6}
  STMFD R1!, {r3-r6}
  ADD R7,R7,#1
  CMP R7,#480
```

```
   BLT LOOP
   SUB R1,R1,R8
   B LOOPB
endl
   BX lr
Image_Address DCD &407E9000
New_Address DCD &4086FE00
last_Address DCD &40002000
END
```

2) 최적화

▶mem_Relocation 최적화 시간비교

LDMDB R0!,{R9-R12} ;4개씩 2번 가져오기 0.14초
LDMDB R0!,{R3-R6,R9-R12} ;8개씩 가져오기 0.12초
8 개씩 가져오는 것이 더 빠르다.

```
LOOP
   LDMDB R0!,{R3-R6,R9-R12}
   STMFD R1!, {R3-R6,R9-R12}
   ADD R7,R7,#1
   CMP R7,#240
   BLT LOOP
   SUB R1,R1,R8
   B LOOPB
```

3) 1920x1080 형태로 잘 저장됐는지 확인하기 위한 코드

: 열을 2048에서 1920으로 바꾸기 때문에 한 행의 1920열 이후 2048-1920 만큼이 비게 된다.

(메모리 주소 측면에서 보면 (2048-1920)*4 만큼 빈다.)

이 빈 곳을 특정 수로 특정해주지 않으면 relocation 하기 전에 그 주소에 저장되어있던 픽셀 정보가 들어 있어 값이 우리가 원하는 대로 잘 저장되었는지 확인이 어렵다. 그렇기에 빈 공간을 0으로 넣어주는 코드를 추가하여 실행이 잘 되었는지 여부를 테스트 할 수 있게 했다.

```
   AREA memRelocation, CODE, READONLY
   EXPORT Relocation
   ENTRY
Relocation
   LDR R0, Image_Address
   LDR R1, New_Address
```

```
    MOV R9, #0
    LDR R10,VALUE
LOOPB
    MOV R7, #0
    MOV R8, #0
LOOP
    LDMDB R0!,{R3-R6}
    STMFD R1!, {r3-r6}
    ADD R7,R7,#1
    CMP R7,#480
    BLT LOOP
    CMP R0,R10
    BLE endl
INZERO
    ADD R8,R8,#1
    CMP R8,#129
    BEQ LOOPB
    SUB R1,R1,#4
    STR R9,[R1] ;*r1 =0
    B INZERO
endl
    BX lr
Image_Address DCD 0X407E9000
New_Address DCD 0X4086FE00
VALUE DCD 0x40000000

END
```

3-2-3 Grayscale

1) 초안

```
    AREA memRelocation, CODE, READONLY
    EXPORT Relocation
    ENTRY
Grayscale
    LDR r0,VALUE1
    LDR r13,save_address
    MOV r12,#0xff
    MOV r11,#1

loop1
    MOV r10,#0
loop
    LDM r0!,{r1,r2,r3,r4}
    ADD r10,r10,#4
```

```
;**************r1**************
        AND r5,r12,r1,LSR #8
        AND r6,r12,r1,LSR #16
        AND r7,r12,r1,LSR #24
        CMP r5,r6
        BLT less_r5_1
        MOV r8,r5
        MOV r9,r6
        BL comp1

less_r5_1
        MOV r8,r6
        MOV r9,r5

comp1
        CMP r8,r7
        MOVLT r8,r7
        CMP r9,r7
        MOVGT r9,r7
        ADD r1,r8,r9
        MOV r1,r1,LSR #1

;**************r2**************
        AND r5,r12,r2,LSR #8
        AND r6,r12,r2,LSR #16
        AND r7,r12,r2,LSR #24
        CMP r5,r6
        BLT less_r5_2
        MOV r8,r5
        MOV r9,r6
        BL comp2

less_r5_2
        MOV r8,r6
        MOV r9,r5

comp2
        CMP r8,r7
        MOVLT r8,r7
        CMP r9,r7
        MOVGT r9,r7
        ADD r2,r8,r9
        MOV r2,r2,LSR #1


;**************r3**************
```

```
        AND r5,r12,r3,LSR #8
        AND r6,r12,r3,LSR #16
        AND r7,r12,r3,LSR #24
        CMP r5,r6
        BLT less_r5_3
        MOV r8,r5
        MOV r9,r6
        BL comp3

less_r5_3
        MOV r8,r6
        MOV r9,r5

comp3
        CMP r8,r7
        MOVLT r8,r7
        CMP r9,r7
        MOVGT r9,r7
        ADD r3,r8,r9
        MOV r3,r3,LSR #1

;**************r4**************
        AND r5,r12,r4,LSR #8
        AND r6,r12,r4,LSR #16
        AND r7,r12,r4,LSR #24
        CMP r5,r6
        BLT less_r5_4
        MOV r8,r5
        MOV r9,r6
        BL comp4

less_r5_4
        MOV r8,r6
        MOV r9,r5

comp4
        CMP r8,r7
        MOVLT r8,r7
        CMP r9,r7
        MOVGT r9,r7
        ADD r4,r8,r9
        MOV r4,r4,LSR #1
;*******************************
        STM r13!,{r1,r2,r3,r4}
        LDR r1,row
        CMP r10,#1920
        BNE loop
```

```
        ADD r13,r13,#128<<2
        ADD r0,r0,#128<<2
        ADD r11,r11,#1
        CMP r11,r1
        BLE loop1
        BX r14

VALUE1 DCD       &40000000
row DCD &00000438
save_address DCD &60000000
        END
```

2) 최적화

: branch 사용을 줄이고, r5,r6,r7,r8 4 개의 레지스터로만 r,g,b 비교하여 max,min 값 저장하였다.

```
            Loop 들어가기 전에 r9에 1080을 넣어준다 LDR r9,row
;**************r1**************
            AND r5,r12,r1,LSR #8
            AND r6,r12,r1,LSR #16
             AND r7,r12,r1,LSR #24

            CMP r5,r6
            MOVGE r8,r5
            MOVLT r8,r6
            MOVLT r6,r5
            CMP r7,r8
            MOVGE r8,r7
            CMP r6,r7
            MOVGE r6,r7
            ADD r1,r8,r6 ;MAX+MIN
            MOV r1,r1,LSR #1
;*********************************
            CMP r11,r9
```

3) padding+grayscale

: assembly 에서도 padding 을 grayscale 단계에서 하는 것이 더 효율적일 것 같아 grayscale 에서 진행하였다.

```
;**********padding********
   LDR r0,VALUE1
   LDR r11,start_address
   MOV r1,#0
```

```
    MOV r2,#0
    MOV r3,#0
    MOV r4,#0
    MOV r5,#0
    MOV r6,#0
    MOV r7,#0
    MOV r8,#0 ;

    LDR r9,row
    SUB r10,r11,#4
    LDR r12,final_address
    MOV r13,#0
loop0
    STM r11!,{r1,r2,r3,r4,r5,r6,r7,r8}
    STM r12!,{r1,r2,r3,r4,r5,r6,r7,r8}
    ADD r13,r13,#1
    CMP r13,#240
    BLT loop0

    MOV r13,#0
    LDR r2,plus_address
loop0_1
    STR r1,[r10,r2]!
    STR r1,[r11,r2]!
    ADD r13,r13,#1
    CMP r13,r9
    BLT loop0_1

    ;**********grayscale******************
    SUB r9,r9, #2
.....
VALUE1 DCD    &40000000
row DCD &0000043A ;1082
save_address DCD &50002004
start_address DCD &50000004
final_address DCD &50872004
count1 DCD &281 ;1923/3=641
plus_address DCD &2000

    END
```

3-2-4 Convolution

1) 초안

: w 배열이 c 에서 assembly 로 넘어가지 않았고, 레지스터가 부족하여 실행이 불가능하였다. 또한 1000 으로 나누는 연산을 제대로 구현하지 못하였다.

```
        AREA convolution, CODE, READONLY
        ENTRY
        EXPORT Convolution
Convolution
        LDR r0, VALUE2
        LDR r4, VALUE1
        MOV r13, #0
        B Loopj
Loopi
        CMP r10, #1920
        BEQ B0
        ADD r10, r10, #1
B0
        CMP r13, #2048
        BEQ return
        ADD r13, r13, #1
        LDR r4, [r0]
        MOV r11, #0
        MOV r12, #0
Loopj
        ADD r12, r12, #1
        LDM r4!,{r1,r2,r3}
        LDM r0!,{r5,r6,r7}
        MUL r8,r1,r5
        ADD r11, r8, r11
        MUL r8,r2,r6
        ADD r11, r8, r11
        MUL r8,r3,r7
        ADD r11, r8, r11 ; r11
        CMP r11, #0
        MOVLT r11, #0
        ;/1024 - (8)-(16)
        MOV r1, r11, LSR #10
        SUB r1, r1, r11, LSR #3
        SUB r1, r1, r11, LSR #4
        LDR r9, VALUE3
        STR r1, [r9], #4
        CMP r12, #3
        BGE Loopi
```

```
        BLT Loopj
return  BX LR

VALUE1 DCD 0x50000000
;VALUE2 DCD 0x20000000
VALUE3 DCD 0x60000000

        END
```

2) 코드 전면 수정

: 처음에 짰던 코드의 문제점을 인식하고 코드를 다시 새로 짰다. w 배열 값 자체를 레지스터에 저장하여 계산하였다. 이 때, 1000 으로 나누는 계산을 하지 못하여 w 배열에 1024 를 곱한 값을 레지스터에 저장하여 모든 계산 이후 최종적으로 배열에 저장할 때는 LSR 을 사용하여 1024 를 나누어 저장하였다.

```
        AREA convolution, CODE, READONLY
        EXPORT Convolution2
      ENTRY
Convolution2

     MOV r2, #-128;
        MOV r3, #-143
        MOV r4, #150
        MOV r5, #-75
        MOV r6, #50
LOOP
        MOV r10, #0; sum=0
        LDM r0, {r7-r9}
        MUL r6, r7, r2
        ADD r10, r10,r6
        MUL r6 ,r8, r3
        ADD r10, r10, r6
        MUL r6,r9, r4
        ADD r10, r10, r6
        LDM r0, {r7-r9}
        MUL r6, r7, r3
        ADD r10, r10, r6
        MUL r6 ,r9, r5
        ADD r10, r10, r6
        MOV r6, #50
        LDM r0, {r7-r9}
        MUL r2, r7, r6
        ADD r10, r10, r2
        MUL r2 ,r8, r5
        ADD r10, r10, r2
```

```
MUL r2 ,r9, r2
ADD r10, r10, r2
MOV r2, #125
CMP r10, #0; sum <0
MOVLT r10, #0; sum=0
MOV r10, r10, LSR #10
STR r10, [r1], #4
CMP r12, #1920
ADDLT r12, r12, #1
BLT LOOP
ADD r11, r11, #1
CMP r11, r13
MOVLT r12, #1
BLT LOOP
BX LR
END
```

3) 최종

: 중간 단계에서 발생했던 주소계산 등으로 인한 오류를 해결했다.

```
AREA convolution, CODE, READONLY
EXPORT Convolution
ENTRY

Convolution
LDR r0, VALUEG
LDR r1, VALUEC
LDR r13, VALUEE
MOV r2, #128
MOV r3, #-143
MOV r4, #-77
MOV r5, #51
MOV r11, #1
MOV r12, #1

LOOP
MOV r10, #0; sum=0
LDM r0, {r7-r9}
MUL r6, r7, r2
ADD r10, r10,r6
MUL r6 ,r8, r3
ADD r10, r10, r6
MUL r6,r9, r5
ADD r10, r10, r6
ADD r0, r0, #0x2000
LDM r0, {r7-r9}
```

```
        MUL r6, r7, r3
        ADD r10, r10, r6
        ; finally r8 * 0 is zero. so, we won't calculate
        MUL r6 ,r9, r4
        ADD r10, r10, r6
        ADD r0, r0, #0x2000
        LDM r0, {r7-r9}
        MUL r6, r7, r5
        ADD r10, r10, r6
        MUL r6 ,r8, r4
        ADD r10, r10, r6
        MUL r6 ,r9, r2
        ADD r10, r10, r6
        SUB r0, r0, #0x4000
        ADD r0, r0, #4
        CMP r10, #0
        MOVLT r10, #0
        MOV r10, r10, LSR #10
        STR r10, [r1], #4
        CMP r12, #1920
        ADDLT r12, r12, #1
        BLT LOOP
        CMP r11, r13
        ADDLT r11, r11, #1
        ADDLT r1, r1, #128<<2
        MOVLT r12, #1
        ADDLT r0, r0, #128<<2
        BLT LOOP
        BX LR
VALUEG DCD 0x50000000
VALUEC DCD 0x60000000
VALUEE DCD 0x438;1080
END
```

3-2-5 Maxpooling

1) 초안

```
        AREA maxpooling, CODE, READONLY
        EXPORT MaxPooling
        ENTRY
MaxPooling
        LDR r0,VALUE1
        LDR r1, VALUE2
        MOV r12, #1
```

```
        MOV r13, #1
LOOP1
        ADD r0, r0, #2048
        LDMIA r0,{r6-r9}
        SUB r0, r0, #2048
        LDMIA r0!, {r2-r5}

        CMP r2, r3
        MOVLT r2, r3
        CMP r6, r7
        MOVLT r6, r7
        CMP r2, r6
        MOVLT r2, r6

        CMP r4, r5
        MOVLT r4, r5
        CMP r8, r9
        MOVLT r8, r9
        CMP r4, r8
        MOVLT r4, r8

        ADD r0, r0, #2048
        LDMIA r0, {r8-r11}
        SUB r0, r0, #2048
        LDMIA r0!, {r3, r5-r7}

        CMP r3, r5
        MOVGE r5, r3
        CMP r8, r9
        MOVLT r8, r9
        CMP r5, r8
        MOVLT r5, r8

        CMP r6, r7
        MOVLT r6, r7
        CMP r10, r11
        MOVLT r10, r11
        CMP r6, r10
        MOVLT r6, r10
        STMIA r1!, {r2, r4-r6}
        CMP r13, #0xF0
        ADDLT r13, r13, #1
        BLT LOOP1

        CMP r12, #0x21C
        ADDLT r0, r0, #128<<2
        ADDLT r12, r12, #1
```

```
        MOVLT r13, #1
        BLT LOOP1
        MOV PC, LR
VALUE1 DCD &60000000
VALUE2 DCD &70000000
        END
```

2) 최종

: 위 코드에서 주소 계산이 잘못되었음을 알았다.

값이 4 바이트로 들어가기 때문에 주소 계산할 때 4 를 곱해야 한다.

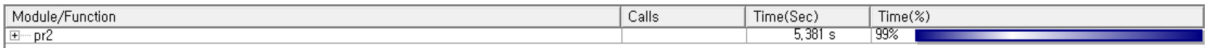
#2048 -> 2048*4 = 8192 = 0x2000

Maxpooling 할 때 한 번 읽을 때 두 줄을 읽어오기 때문에 다음 루프를 돌 때 두 줄을 넘어가야 한다.

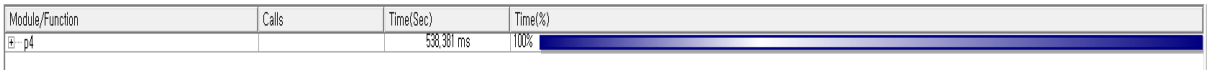
#128<<2 -> 128*4+2048*4

3-2-6 최종 코드 및 결과 시간

1) 위의 최종 코드들로 결과 냈을 때



Assembly code 결과 시간은 약 5.38 초가 나왔다. c 로만 짠 code 보다 훨씬 빨라진 것을 볼 수 있다.



C code 결과 볼 때와 마찬가지로 폴더 내의 불필요한 파일들을 삭제하고 실행해 보았더니 실행시간이 초로 감소하였다. 결과는 같았다.

2) 최종 최적화

: C 에서처럼 1024 를 나누는 과정을 convolution 에서 하는 것 보다 maxpooling 에서 하는 것이 훨씬 연산 횟수가 적어 빠를 것이라 판단했다

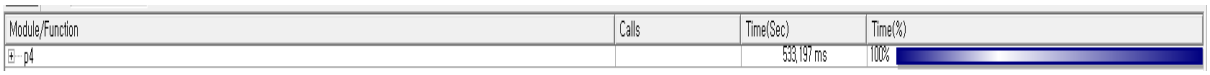
Convolution code에서 MOV r10, r10, LSR #10을 MOV r10,r10으로 바꾸고 Maxpooling code에서 STMIA r1!, {r2, r4-r6}으로 값 저장하기 전에

```
MOV r2, r2, LSR #10; sum=sum/1024
MOV r4, r4, LSR #10; sum=sum/1024
```

MOV r5, r5, LSR #10; sum=sum/1024

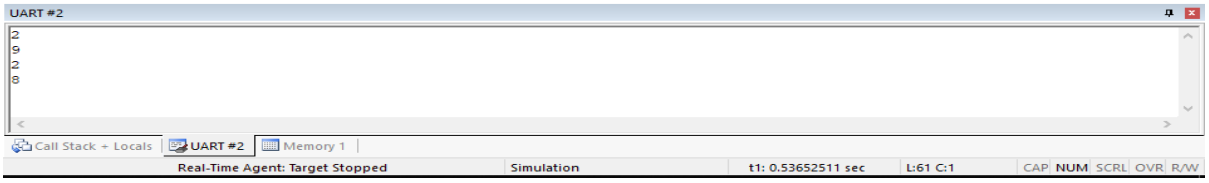
MOV r6, r6, LSR #10; sum=sum/1024 이 코드를 넣어주었다

그 결과 실행 시간이 printdecimal 이 없는 코드에서는

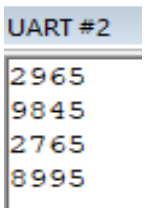


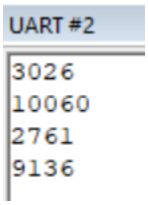
약 0.5332 가 나왔고.

Printdecimal 을 넣어 지정된 값을 출력하는 코드에서는



으로 약 0.5365 초가 되었다.





assembly 에서도 c 와 같은 방법으로 이와 같은 결과를 확인 할 수 있다.

3-2-7 결과 이미지 출력

1) 결과 출력

3-2-8 다른 사진 넣었을 때

C 에서 했던 것과 같은 스파이더맨 사진을 넣었을 때의 결과 사진이다.

4. 프로젝트 진행 과정

4-1 프로젝트 일정 계획

일	월	화	수	목	금	토
	20	21	22	23	24	25
	Image Load 이해 및 CNN 이해, 코드 틀 작성		c coding			c code 완성 및 결과 도출, assembly 틀 작성
26	27	28	29	30	31	
assembly coding 및 최적화, 결과 도출				보고서 작성		

4-2 프로젝트 진행 일지

✓ **5월 14일**

프로젝트 시작. CNN에 대해 기본 개념 공부했다.

→ 질문 : OpenCV를 이용하여 이미지를 keil에 불러와도 되는지.

W배열이 무엇을 의미하는지?

✓ **5월 20일** 17:00 ~ 21:30 (카페 – 투썸)

PDF를 설명을 보고 keil에 이미지를 불러왔다.

: RGB 이미지 파일로 실행 -> RGBA파일로 수정됨 -> RGBA 이미지 파일로 실행 완료

assembly code 알고리즘을 만들기 시작했다.

메모리(0x40000000부터)에 저장된 값 불러오는 방법에 대한 의논 :

4byte씩 저장되어 있으므로 r0 ~ r3를 이용한다.

1. right shift 이용

r0에 저장된 값은 각각 shift해서 저장(r4~r7)

AND 0x00000011을 통해 마지막에 있는 값만 남기기 -> r4에 저장한다.

이런 방법으로 r4, r5, r6, r7에 값 저장하고 비교한다.

2. left shift 이용

r0에 저장된 값을 두 번 left shift하여 r4에 저장

r0에 저장된 값을 네 번 left shift하여 r5에 저장

이런 방법으로 r4, r5, r6, r7에 값을 저장하고 비교한다.

L을 계산 후 r9 := r0&r1&r2&r3 저장 : 총 r9~r12에 저장한다.(STM을 이용해 한꺼번에 저장)

→ 질문 : C code에서 이미지를 불러올 때 FILE I/o를 이용하는 것이 아니라면 C code에서 어떻게 이미지를 바로 받아 올 수 있는 건지

RGBA 32bit로 되어있는 이미지에서 A를 사용하지 않는다면 RGB만 우리가 꺼내서 사용하는 것인지

✓ **5월 21일** 17:00 ~ 21:30 (카페 – 투썸)

c code를 짜기 시작했다.

1번 이미지 load

이미지를 메모리에 저장할 때 assembly를 통해 load한 후에 main 함수에서 나머지 함수를 호출하는 방법으로 하기로 결정했다.

2번 grayscale 알고리즘 및 간단한 코드 작성했다.

convolution의 padding 방법에 대해 의논 :

1. 배열을 선언할 때 본래 grayscale 배열크기보다 행 열이 +2 만큼 증가된 배열을 선언 한 후 grayscale을 [1,1]부터 [1081,1921]까지 가장자리는 비워두고 저장시키고, 그 가장자리가 0이 되도록 하는 방법
2. grayscale을 더 큰 배열에 넣으려면 grayscale 배열을 다시 불러와서 새로운 배열에 넣어야 하기 때문에 계산 할 때 위치가 가장자리인지 확인 하고, 값이 없는 부분을 0으로 계산하는 방법

=> 계산 위치가 가장자리인지 확인 하고 값이 없는 부분을 0으로 두고 계산하는 것은 if문이 너무 많이 필요하고 복잡하기 때문에 1번 방법 선택했다.

3번 convolution 알고리즘 및 간단한 코드 작성했다.

convolution의 계산은 4중 for문을 이용하여 작성했다.

4번 max pooling 알고리즘 및 간단한 코드 작성했다.

→ 질문 : grayscale 후 저장되는 방식이 4byte(int형)인지 1byte(char형)인지

✓ 5월 23일 18:00 ~ 22:00 (형남공학관 – 6~7층 쉼터)

2번 grayscale 3번 convolution 4번 max pooling 코드 설계를 확장시켰다.

코드들이 정상적으로 돌아가는지 확인하기 위해 visual studio를 통해 각 함수 코드 실행시킨 후 정상적으로 실행하면 keil을 통해 실행을 시도했다.

함수를 기능별로 나눠서 코드를 작성하기로 하기로 했다.

grayscale 결과는 4byte(int형)으로 저장하기로 결정했다.

→ 순수 C언어로만 코드를 작성할 때, 다른 헤더들도 사용할 수 있는지

✓ **5월 24일** 13:30 ~ 20:00 (카페 – 핀벨(숙대입구))

1번 코드 작성 완료 : 이미지 파일의 값 load

assembly를 통해 이미지 load하는 방법을 사용하려 했으나 main에서 c를 통해 memory 접근 방법을 알아내었다.

2번 grayscale 수정 & 통합완료 : grayscale 함수를 완성했고, 의논 결과 convolution에서 padding을 하려면 우리가 선택한 방법(1번)을 쓰기 위해서는 grayscale배열의 값을 다른 배열에 다시 저장해야하는 과정이 추가적으로 필요하므로 grayscale 연산 후 바로 [1082][1922]배열에 값을 넣기로 하면서 convolution에서 하는 padding을 grayscale 함수 내에서 하기로 정했다.

padding방법 의논 :

1. [1082][1922] 배열 선언 시 0으로 초기화 하여 grayscale 값을 넣기만 하면 되는 방법
2. [1082][1922] 배열 선언 후 0이 되어야 하는 가장자리 부분만 0으로 초기화 하는 방법

=> 직접 두 방법 다 실행 해 본 결과, 2번 방법이 더 빨라서 2번 방법을 쓰기로 결정했다.

3번 convolution 수정 & 통합 : convolution 계산 결과 값이 음수이면 0으로 만들어 준다.

1. for문 안에서 음수인지 확인하는 방법

2. convolution 계산 과정이 끝난 후 새로운 for문을 통해 음수인지 확인하는 방법

각 함수의 실행 후 memory에 정상적인 값이 들어가 있는지 확인하면서 실행했다. 하지만convolution까지 실행한 결과 실행이 너무 오래 걸렸다.

✓ **5월 25일** 13:30 ~ 20:00 (카페 – dz, 미소콩(숙대입구))

3번 convolution 수정 & 통합 : convolution 계산을 2과정으로 나눠봤으나 더 오래 걸렸다. w배열을 double 형에서 int형으로 바꿔주기 위해서 배열 각각의 값들을 x 1000을 해서 실행해본 결과 시간이 훨씬 줄었다.

4번 max pooling 수정 & 통합 완료 : maxpooling 함수에서도 max함수를 따로 만들어 사용했었으나, 따로 쓰이는 곳이 없고, 오직 maxpooling 함수에서만 4개의 데이터를 비교하는 max 함수를 사용하여 재사용성이 없기 때문에, 한 코드로 합치는 것이 낫다고 생각했다. max 함수를 따로 만들지 않고 maxpooling안에서 max값을 찾는 작업을 하니, 시간이 더 빨라졌다. 함수가 그렇게 복잡하거나 길어지지도 않았다.

최적화 : 유지보수를 위해서는 함수를 기능별로 여러 개 만드는 것이 좋지만, 속도만을 위한다면 함수를 최대한 만들지 않는 것이 좋다고 판단했다. 따라서 max함수와 min함수를 만들어서 실행한 결과와 만들지 않고 바로 함수 안에서 코드로 풀어서 쓴 결과와 시간이 약 2초정도 차이 난다.

maxpooling까지 실행한 최종 결과가 convolution함수까지만 실행한 결과보다 크기는 1/4만큼 줄고 더 선명해졌다

순수 c코드는 완성하였다.

→ 질문 : 최종결과 이미지가 960*540인데 viewer를 통해 이미지 변환이 안되는 이유?(메모리범위주소 잘못 설정)

완성된 c코드를 보고 assembly로 알고리즘을 만들고 코드 작성을 하기 시작했다.

속도 향상을 위해 stack을 이용하기보다는 최대한 register들을 활용하기로 했다.

assembly code에서 padding에 대한 의논 :

1. 처음부터 0으로 초기화? vs 2. 가장자리에만 넣어주기(주소계산필요)

모든 메모리를 0으로 만들어서 초기화 하는 과정(한번에 8개씩 저장하는 방법 사용)과 C에서와 마찬가지로 가장자리의 주소만 계산하여 그 부분만 0을 넣어주는 과정 중 무엇이 더 빠를지 고민함

=> branch가 많은 것 보다 STM이 많은 것이 시간 소모가 더 클 것 같아 2번을 선택했다.

→ 질문 : 굳이 메모리를 2048*1080으로 해야 하는지?(비효율적이 아닌가)

✓ **5월 28일** 17:30 ~ 21:30 (카페 – 투썸)

1번 relocation 코드 완성 및 실행 + 최적화 진행

2번 grayscale 코드 완성 및 실행 해보기 + 최적화 : branch를 많이 이용한 직관적인 코드로 우선 완성 한 후 최대한 branch를 덜 사용하도록 최적화를 하였다.

3번 convolution 코드 작성 및 실행 : 일단 최적화의 방법을 생각하지 않고 convolution 계산이 작동이 되도록 코드를 짜고 실행하였다.

4번 max pooling 코드 작성 및 실행 : 우리가 사용한 최적화 방법은 Branch 횟수를 줄이고 cycle을 많이 차지하는 LDR/STR를 덜 쓰는 것이다. 설계 전 방법은 2가지였는데, 하나는 한 Block씩 로드하여 총 4번 실행해 4개를 한 번에 저장하는 것이었고, 하나는 두 블록을 로드하여 총 2번 실행해 2개를 한 번에 저장하는 일을 2번 하는 것이었다. 브랜치와 LDM, STM 등을 고려한 cycle을 계산해본 결과, 두 번째 방법이 훨씬 빠르다는 것을 알게 되었고, 두 번째 방법에서 좀 더 발전한 방법으로 코드를 작성하고 실행해보았다.

✓ **5월 29일** 13:00 ~ 22:30 (카페 – 스타벅스)

2번 grayscale 코드 완성 : padding에서 발생했던 주소 오류를 해결하여 완성 한 후 grayscale 값이 잘 들어가는 것도 확인하였다.

3번 convolution 코드 완성 : 최적화를 위해 branch사용 횟수를 줄이고, w배열을 직접 레지스터에 저장하는 코드로 수정하고 grayscale 코드와 통합하여 정상적으로 작동하는지 확인했다.

4번 maxpooling 코드 완성 : register에 대해 좀 더 생각을 해보니 네 개의 블록을 한 번에 로드할 수 있었다. 즉, 전날 생각했던 첫 번째 방법과 두 번째 방법을 합하여, 4개의 블록을 로드하고 4개의 값을 한 번에 저장하는 방법을 모색하여 코드를 완성하였다. convolution까지 통합 한 후 maxpooling도 통합했으나 값이 정상적으로 나오지 않았다. 주소 계산이 잘못 되어 있는 것을 발견 해 수정 후에 정상적으로 작동하는 것을 확인했다.

c코드를 참고하던 중 w배열의 [0,3]이 0.05가 아닌 0.15로 되어 있는 오류를 발견하였다. 이후 정상적으로

수정하여 c결과 이미지 파일을 다시 출력하였다.

✓ **5월 30일** 18:00 ~ 22:30 (형남공학관 – 5~6층 쉼터)

통합된 코드 검토 및 코드를 토대로 역할 분담을 하여 보고서를 작성하기 시작하였고 그 과정에서 코드 오류를 발견하여 수정했다.

지정된 곳의 값을 출력하는 과정에서 소수를 printdecimal로 표현하는 방법에 대해 고민하다가 convolution에서 다시 1000 or 1024로 곱하는 과정을 생략하고 1000의자리까지 출력하여 소수점 셋째 자리 수까지 확인하였다.

keil 프로젝트 폴더 내의 불필요한 파일들을 삭제 후 다시 실행해 보았을 때, 실행 속도가 확연히 줄어든 것을 발견하였다.

✓ **5월 31일** 11:00 ~ 22:30 (숭실마루)

convolution에서 w에 곱해주었던 가중치를 나누는 것 보다 maxpooling하고 값을 저장 할 때 나눠주는 것이 연산 횟수가 적어져 총 실행 시간이 줄어드는 것을 발견하였다.

C code에서도 w에 1024를 곱해주고 연산 이후 저장할 때 나눠주는 것이 1000의 경우보다 실행시간이 짧은 것을 발견하였다.

전반적인 설계보고서 작성을 완료하였다.