2025S 마이크로프로세서 응용
(Microprocessor Application)

Team Project Report

| 팀원 김동준(20211513 IT 융합)
안덕찬(20202978 IT 융합)
우상욱(20203023 전자전공)
이장근 (20203496 IT 융합)

<목 차>

Ⅰ. 프로젝트 개요

Ⅰ-1. 프로젝트 주제

본 프로젝트는 32-bit RGBA 형식의 이미지를 ARM 어셈블리 언어를 이용하여 다양한 방식으로 변환하고, 이를 최적화하여 성능 향상을 도모하는 것을 목표로 한다. RGBA 포맷의 이미지는 각 픽셀이 Red, Green, Blue, Alpha 채널로 구성되어 있으며, 각 채널은 8-bit 의 정보를 담고 있어 총 32-bit, 즉 4 바이트로 구성된다.
본 과제에서는 다음과 같은 세 가지 서브 프로젝트로 구성된 이미지 처리 작업을 수행한다.

- Red Pixel Count: 이미지에서 Red 값이 128 이상인 픽셀의 개수를 계산하는 함수 구현
- RGB Negative 변환: RGB 값을 각각 255 에서 뺀 색상 반전 이미지를 생성하는 함수 구현
- Grayscale 변환: RGB 값을 가중합(3R + 6G + B)을 이용하여 16-bit Grayscale 이미지로 변환하는 함수 구현

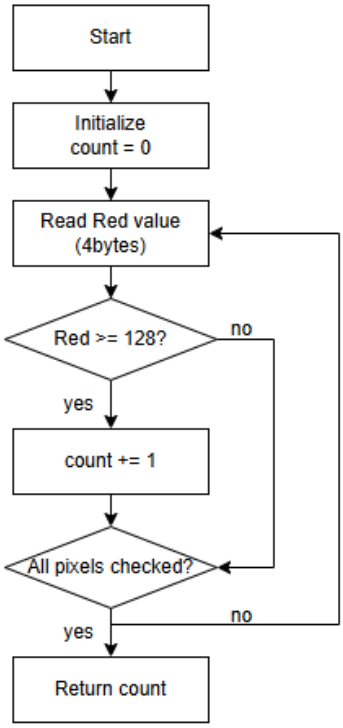
Ⅰ-2. 진행 방향

각 변환 함수는 먼저 C 언어로 구현하여 기능의 정확성을 검증하고, Keil uVision 의 시뮬레이션 환경을 통해 함수 수행 전후의 메모리 상태를 직접 확인하였다. countRed, convertReverse, convertGray 등의 함수는 메인 함수에서 연속적으로 호출되며, 각 결과는 별도의 메모리 블록에 저장되도록 설계하였다. 이를 통해 동일한 입력 데이터에 대해 각 변환 알고리즘이 올바르게 동작하는지를 시각적으로 확인할 수 있었다. 성능 평가는 performance analyzer 이후 성능 최적화를 위해 ARM 어셈블리 언어 기반으로 동일 기능을 재구현하였으며, 이 과정에서 Memory Relocation 을 도입하였다. 기존 32-bit RGBA 포맷에서 Alpha 채널을 제외하고, R, G, B 채널을 각각 연속된 메모리 공간에 분리 저장함으로써 Planar 메모리 구조를 구성하였다. 이 구조는 ARM 코드 내에서 연산 시 불필요한 메모리 접근을 줄이고, 캐시 활용도를 높이는 데 기여하였다.
또한, 최적화 이전과 이후의 함수 실행 시간, 명령어 개수를 Performance Analyzer 를 통해 비교 분석하였으며, 추가적인 논리 수정과 ARM 명령어 단위 수준에서의 추가 최적화도 시도하였다. 최종적으로는 기존 C 기반 결과와 ARM 최적화 버전의 출력 일치 여부를 검증하여, 코드 정확성과 성능 향상 효과를 종합적으로 평가하였다.

Ⅱ. 이미지 변환 함수의 동작 방식 및 Block Diagram

Ⅱ-1. Red pixel count

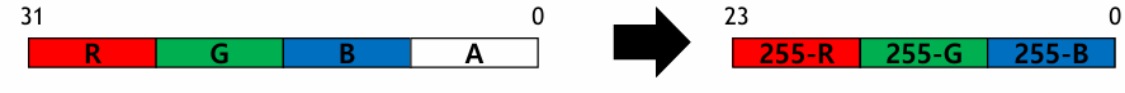
Red Pixel Count 변환은 이미지 내 픽셀 중 Red 채널 값이 128 이상인 픽셀의 개수를 계산하는 작업이다. RGBA 포맷에서 각 픽셀은 총 4 바이트로 구성되며, [R][G][B][A] 순서로 저장되어 있다. 이 중 Red 값은 각 픽셀의 첫 번째 바이트(Offset 0)에 위치한다.
해당 함수는 이미지의 시작 주소에서부터 4 바이트 단위로 픽셀을 순차적으로 순회하며, 각 픽셀의 Red 값이 128 이상인지 비교한다. 조건을 만족할 경우 카운트를 하나 증가시키고, 모든 픽셀을 검사한 뒤 최종 결과를 반환한다.
다음은 블록 다이어그램으로 나타낸 순서도이다:



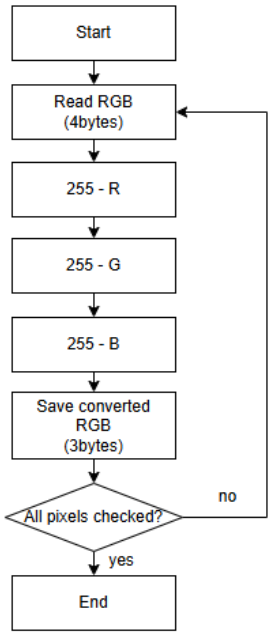
<그림 1. countRed 블록 다이어그램>

Ⅱ-2. RGB 색상반전

RGB Negative 변환은 컬러 이미지의 각 픽셀에 대해 색상을 반전시켜 새로운 시각 효과를 생성하는 작업이다. RGBA 포맷에서 한 픽셀은 Red, Green, Blue, Alpha 의 네 가지 채널로 구성되어 있으며, 각 채널은 1 바이트(8-bit)의 정보를 가진다. 이 중 Alpha 채널은 불투명도(opacity)를 나타내며 색상 자체에는 영향을 주지 않기 때문에 본 연산에서는 제외된다.



변환 방식은 간단하다. 각 채널(R, G, B)의 값을 각각 (255 - 원래 값)으로 계산하여 색상을 반전시킨다. 예를 들어, R 값이 100 이라면 반전된 R 은 155 가 된다. 이를 모든 픽셀에 대해 반복 수행함으로써 전체 이미지의 색상이 반전된 결과를 얻게 된다.
이미지 데이터에 순차적으로 접근하여 R, G, B 값을 각각 추출하고, 반전 연산 후 결과 메모리 공간에 저장한다. Alpha 값은 건너뛴다. 결과는 24-bit RGB 포맷으로 저장된다.
다음은 블록 다이어그램으로 나타낸 순서도이다:



<그림 2. 색상 반전 블록 다이어그램>

Ⅱ-3. Grayscale

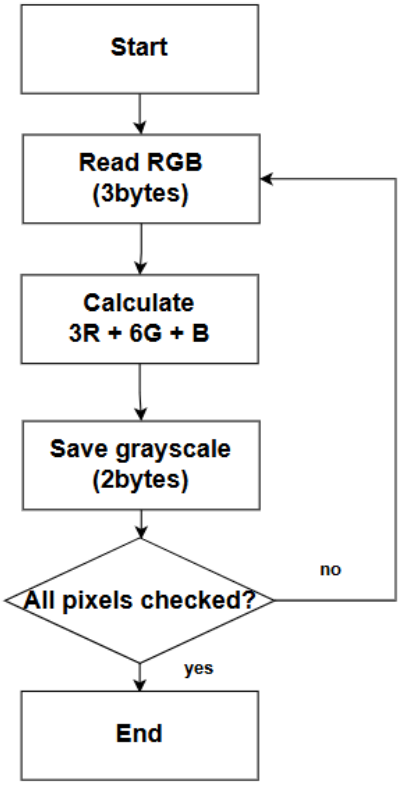
RGBA 포맷에서 Red, Green, Blue 값은 각각 8-bit 이지만, 이를 단순 평균으로 계산하면 사람이 인식하는 명도 특성을 반영하기 어렵다. 따라서 본 프로젝트에서는 가중합 방식을 적용해 Grayscale 값을 생성하였다. 사용된 공식은 다음과 같다:

$$Gray = 3 \times R + 6 \times G + 1 \times B$$

이 가중치는 사람 눈이 Green 에 가장 민감하고, Red, Blue 순으로 민감도가 낮다는 점을 반영한 것이다. 따라서 Green 의 영향력이 가장 크고, Blue 는 가장 작다.

RGBA 데이터를 순차적으로 읽고 R, G, B 값을 추출한 뒤, 위의 공식을 적용하고, 변환된 결과는 16-bit 크기로 별도의 메모리 공간에 저장된다. (공식에 따라 Alpha 채널은 무시된다.)

다음은 블록 다이어그램으로 나타낸 순서도이다:



<그림 3. Grayscale 블록 다이어그램>

Ⅲ. 이미지 변환 함수의 코드 및 실행결과

Ⅲ-1. 메인 함수

C 언어 함수로 구현된 세 가지 이미지 변환 함수는 main() 함수 내에서 순차적으로 호출되며, 각각의 출력 결과는 메모리 상의 서로 다른 위치에 저장된다. 이를 통해 함수 간 결과가 겹치지 않도록 하며, 시뮬레이션 상에서 결과를 개별적으로 확인할 수 있도록 설계되었다.

함수 호출 순서 및 메모리 흐름은 다음과 같다:

함수명	설명	결과 저장 주소
convertGray	32-bit RGBA → 16-bit Grayscale 변환	0x30020000 (C) 0x30030000 (ASM)
convertReverse	32-bit RGBA → 24-bit RGB Negative 변환	0x30040000 (C) 0x30050000 (ASM)
countRed	Red 값 ≥ 128 픽셀 수 계산	0x30010000 (C) 0x30010004 (ASM)

find_IDAT() 함수를 통해 이미지의 png 'IDAT' chunk 를 찾아 Pixel data 의 시작 주소를 찾은 뒤, 해당 위치(0x40000000 + offset)부터 9,600 개의 픽셀 데이터를 대상으로 세 가지 함수가 각각 실행된다. 이 과정은 단일 루프가 아닌 각 함수 별 독립적으로 수행되며, 함수 호출 전후의 메모리 상태는 memory map 을 통해 시각적으로 확인이 가능하다. 또한, 일일이 전체를 확인하는 데에 어려움이 있으므로, 추가적인 isSame 함수를 사용하여 같은 지 검증했다. C 코드 기반의 main() 함수 핵심 흐름은 아래와 같다.

지정한 헤더파일을 이용해 실험이 편의성을 확보하였다. 다음은 헤더를 불러온 코드이다:

```c
#include <stdint.h>
#include <stdlib.h>
#include "find_IDAT.h"
#include "convertGray.h"
#include "isSame.h"
extern void _sys_exit(int return_code);
```

```
extern  int countRedAsm(uint8_t* origin, int size);
extern  void convertGrayAsm(uint8_t* origin, uint16_t* toSave, int size);
extern  void convertReverseAsm(uint8_t* origin, uint8_t* toSave, int size);
```

<코드 1. 메인 함수의 header file>

다음은 메인함수의 코드이다:

```
int main(void) {
    const int PIXEL_COUNT = 9600;
    uint8_t* pixel_start = find_IDAT( (uint8_t*)0x40000000 ) + 4;
    int* count = (int*)0x30010000;

    uint16_t* grayTarget = (uint16_t*)0x30020000;   // ~0x30024B00
    uint16_t* grayTarget2 = (uint16_t*)0x30030000;   // ~0x30024B00

    uint8_t* reverseTarget = (uint8_t*)0x30040000;
    uint8_t* reverseTarget2 = (uint8_t*)0x30050000;
    uint8_t* isSameReturn = (uint8_t*)(count + 2);

    convertGray(pixel_start, grayTarget, PIXEL_COUNT);
    convertGrayAsm(pixel_start, grayTarget2, PIXEL_COUNT);
    *isSameReturn = isSame_uint16(grayTarget, grayTarget2, PIXEL_COUNT);

    convertReverse(pixel_start, reverseTarget, PIXEL_COUNT);
    convertReverseAsm(pixel_start, reverseTarget2, PIXEL_COUNT);
    *(isSameReturn + 4) = isSame_uint8(reverseTarget, reverseTarget2, PIXEL_COUNT*3);

    *count = countRed(pixel_start, PIXEL_COUNT);
    *(count + 1) = countRedAsm(pixel_start, PIXEL_COUNT);
    _sys_exit(0);
}
```

<코드 2. 메인함수의 C 코드>

메인 함수는 바뀐 값을 저장한 Target 주소를 지정해 실험 전후 비교를 용이하게 하였다. Target1 과 Target2 는 각각 C 언어 코드와 어셈블리 코드의 타겟 저장 주소이다.

Ⅲ-2. Red pixel count

아래는 c 언어로 작성한 구현 코드이다

```c
int countRed(uint8_t* origin, int size) {
    int cnt = 0;
    for (int i = 0; i < size; i++) {
        cnt += (*origin) >= 128 ? 1 : 0;
        origin += 4;
    }
    return cnt;
}
```

<코드 3. c 언어 코드>

C 언어 구현에서는 origin 포인터를 통해 RGBA 데이터를 순차적으로 접근하며, 각 픽셀의 Red 값을 추출하고 해당 값이 128 이상인 경우 누적 카운트를 증가시킨다. 이때 origin 포인터는 한 픽셀마다 4 바이트씩 이동하며, Alpha 채널은 연산에 포함되지 않는다.

다음은 어셈블리 언어로 작성한 구현 코드이다.

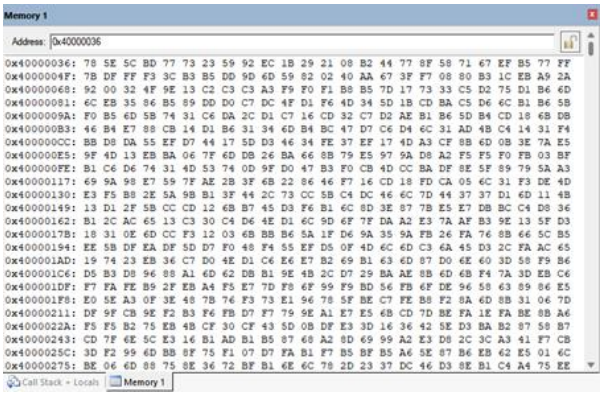
```
AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  countRedAsm

    ; int countRedAsm(unsigned char* baseAddress, int maxCount)
    ;   r0 = baseAddress
    ;   r1 = maxCount
    ; return: int (r0)
    ;   r2 = cnt
    ;   r3 = buf

countRedAsm
    MOV     r2, #0
```

```
CRA_L1
    LDRB    r3, [r0], #4
    CMP r3, #128
    ADDGE   r2, r2, #1
    SUBS    r1, r1, #1
    BGT     CRA_L1
    MOV r0, r2
    BX lr

END
```

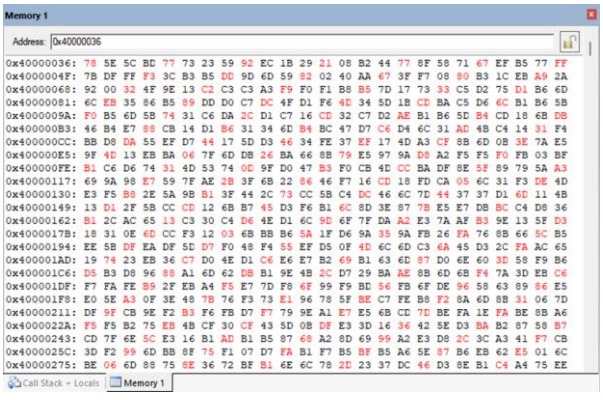
<코드 4. 어셈블리 코드>

countRedAsm 함수는 원본 이미지 주소(r0)에서 R 값을 하나씩 불러온 후, 128 과 비교하여 128 보다 크면 r2(cnt)에 1 을 더하여 누적하는 방식으로 동작한다. 픽셀 하나가 처리될 때마다 반복 횟수(r1)를 감소시키고, 0 보다 큰 경우 루프를 계속 진행한다. 이 구조는 픽셀 단위로 Red 픽셀 값이 128 보다 큰 경우를 카운트하는 과정을 반복하며, C 언어로 작성한 동일한 논리를 어셈블리 수준에서 직접 구현한 것이다.

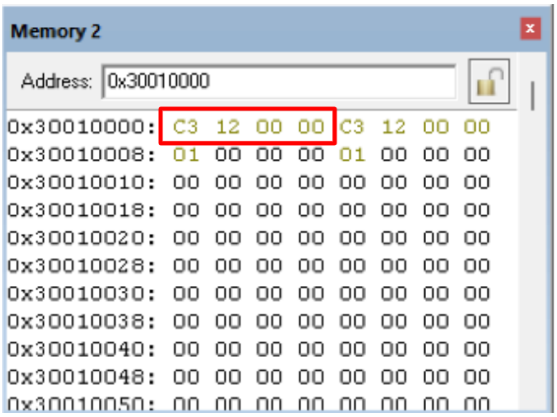
다음은 실행결과이다.



[실행 전 캡쳐]



[실행 후 캡쳐]



[0X30010000 에 저장된 값]

픽셀의 Red 값이 128 이상인 픽셀 개수를
0x30010000 에 저장했음으로,
0x12C3(4803 개)이 9600 개 픽셀 중 RED 픽셀이
128 개 이상인 개수이다.

Ⅲ-3. RGB 색상반전

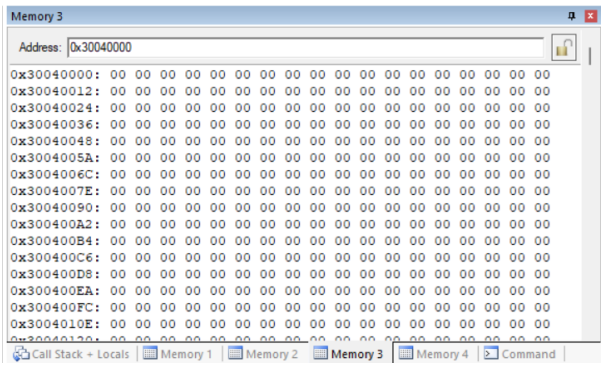
다음은 C 언어로 작성된 구현 코드이다:

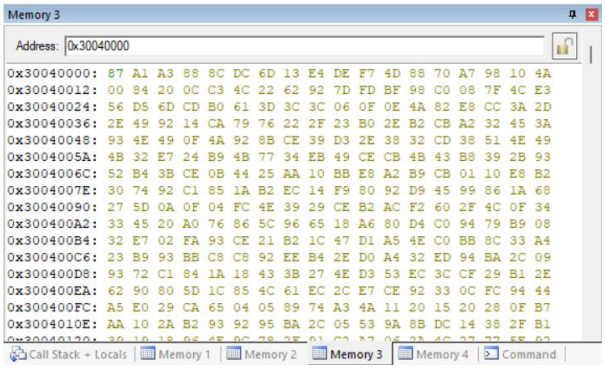
```
void convertReverse(uint8_t* origin, uint8_t* toSave, int size) {
    for (int i = 0; i < size; i++) {
        // *(toSave++) = 0;
        *(toSave++) = 255 - *(origin++);
        *(toSave++) = 255 - *(origin++);
        *(toSave++) = 255 - *(origin++);
        (origin++);
    }
}
```

<코드 5. convertReverse C 코드>

C 언어 구현에서는 각 픽셀에 대해 R, G, B 값을 순차적으로 읽고 각각에 대해 반전 연산을 수행한 뒤, 3 바이트 크기의 새로운 메모리 공간에 저장한다. 원본 포인터(origin)는 4 바이트씩 이동하며, 결과 포인터(toSave)는 반전된 R, G, B 값을 연속적으로 저장한다.

다음은 실행 결과이다:





실행 전 메모리맵	**실행 후 메모리맵**

C 코드에서 target 주소를 0x30040000 으로 지정한 뒤 실행한 결과, 색상 반전된 값이 해당 주소에 정상적으로 저장된 것을 확인할 수 있었다. 특히 시작 부분이 78 A1 A3 으로 나타나며, 이는 기존 픽셀값에서 255 를 뺀 결과와 일치하여 색상 반전 로직이 정확히 동작했음을 확인할 수 있다.

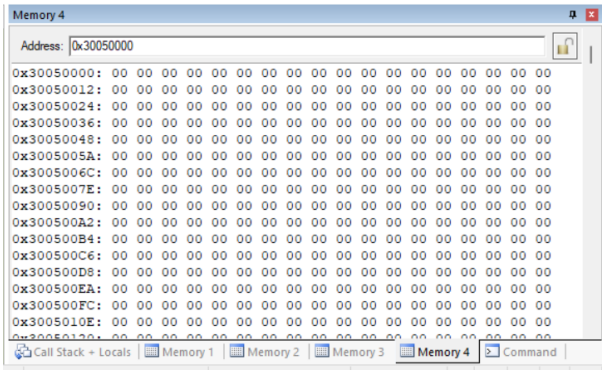
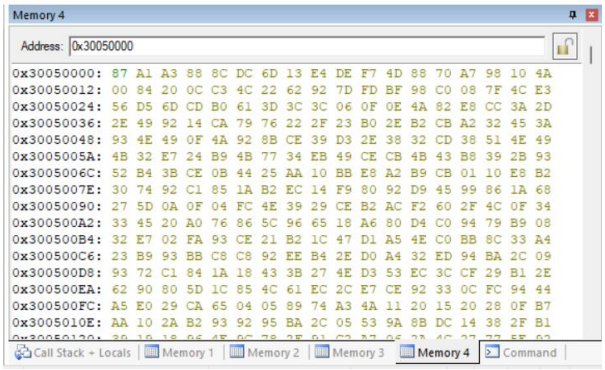
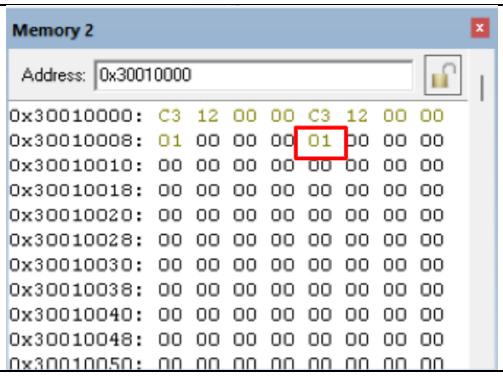
다음은 어셈블리 언어로 작성한 구현 코드이다:

```
AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  convertReverseAsm
    ;   void convertReverseAsm(uint8_t* origin, uint8_t* toSave, int size) {
    ;
    ;   r0 = baseAddress
    ;   r1 = targetAddress
    ;   r2 = maxCount
    ;   r3 = converted
    ;
    ;   r4 = buf

convertReverseAsm
    STMFD sp!, {r4, lr}
CRA_L1
    ; r4 = r
    LDRB    r4, [r0], #1
    RSB     r3, r4, #255
    STRB    r3, [r1], #1
    ; r4 = g
    LDRB    r4, [r0], #1
    RSB     r3, r4, #255
    STRB    r3, [r1], #1
    ; r4 = b
    LDRB    r4, [r0], #2
    RSB     r3, r4, #255
    STRB    r3, [r1], #1

    SUBS    r2, r2, #1
    BGT     CRA_L1
    ;BX lr;
    LDMFD sp!, {r4, pc}
END
```

<코드 6. convertReverse 어셈블리 코드>

`convertReverseAsm` 함수는 원본 이미지 주소(`r0`)에서 RGB 값을 하나씩 불러온 후, 각 채널에 대해 255 에서 해당 값을 뺀 결과를 계산하여(`RSB` 명령어) 저장 주소(`r1`)에 기록한다. R 과 G 채널은 각각 1 바이트씩 순차적으로 처리되며, B 채널은 `LDRB` 명령어의 오프셋으로 2 바이트를 건너뛰며 로드되는 방식으로 처리된다. 픽셀 하나가 처리될 때마다 반복 횟수(`r2`)를 감소시키고, 0 보다 큰 경우 루프를 계속 진행한다. 이 구조는 픽셀 단위로 색상 반전을 반복하며, C 언어 구현과 동일한 논리를 어셈블리 수준에서 직접 구현한 것이다. 마지막에는 `sp` 에 저장된 레지스터 상태를 복구하고 `pc` 로 복귀하여 함수 호출을 마친다.

다음은 실행 결과이다:

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>실행 전 메모리맵</td><td>실행 후 메모리맵</td></tr>
<tr><td> 
</td><td></td></tr>
<tr><td>bool isSame 실행 결과</td><td></td></tr>
</table>

어셈블리 코드에서는 target 주소를 `0x30050000` 으로 지정하여 실행하였고, 그 결과 색상 반전된 값이 해당 주소에 정상적으로 저장된 것을 확인할 수 있었다. 또한, C 언어로 구현했을 때와 동일한 방식으로 동작함을 통해 기능이 정확히 일치함을 검증할 수 있었다.

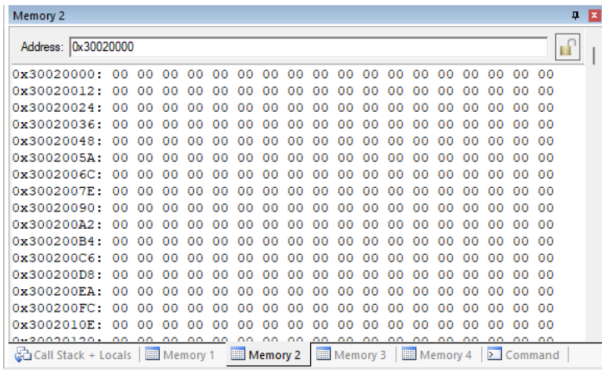
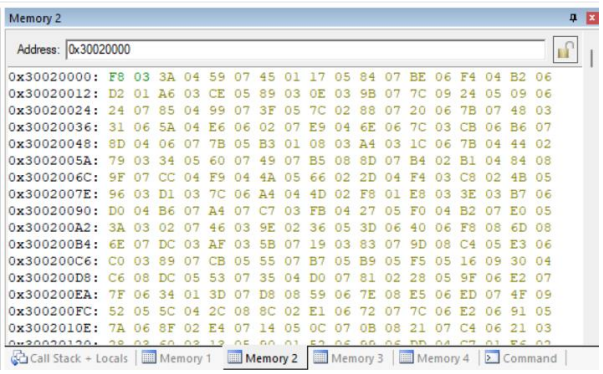
Ⅲ-4. Grayscale

아래는 C 언어 기반의 Grayscale 변환 함수 구현이다:

```c
void convertGray(uint8_t* origin, uint16_t* toSave, int size) {
    uint8_t r, g, b;

    for (int i = 0; i < size; i++) {
        r = *(origin++);
        g = *(origin++);
        b = *(origin++);
        (origin++); // Skip alpha channel

        *toSave = 3 * (uint16_t)r + 6 * (uint16_t)g + (uint16_t)b;
        toSave++;
    }
}
```

<코드 7. convertGray C 코드>

C 언어 구현에서는 RGBA 포맷으로 저장된 원본 데이터에서 R, G, B 값을 차례로 추출하고, 위 공식을 적용하여 Grayscale 값을 계산한 뒤 16-bit 크기의 메모리 공간에 저장한다. Alpha

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>[실행 전 메모리맵]</td>
<td>[실행 후 메모리맵]</td>
</tr>
</table>

채널은 연산에 사용되지 않으며, 4 번째 바이트는 무시된다.

다음은 실행 결과이다:C 코드에서 실행한 결과, Grayscale 변환 결과가 해당 주소에 정상적으로 저장된 것을 확인할 수 있었다. 변환에는 Gray = 3×R + 6×G + B 의 가중합 방식을 사용하였으며, 원래 RGB 값인 78 5E 5C 계산된 Grayscale 값은 0x03F8 로, 실제 메모리에서도 해당 값이 저장된 것을 통해 변환 로직이 정확하게 동작했음을 확인할 수 있었다.

아래는 어셈블리 언어 기반의 Grayscale 변환 함수 구현이다:

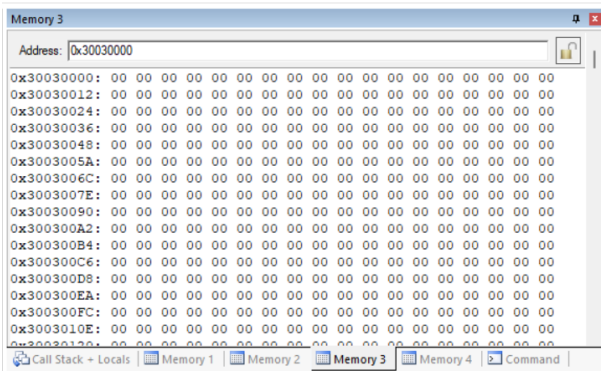
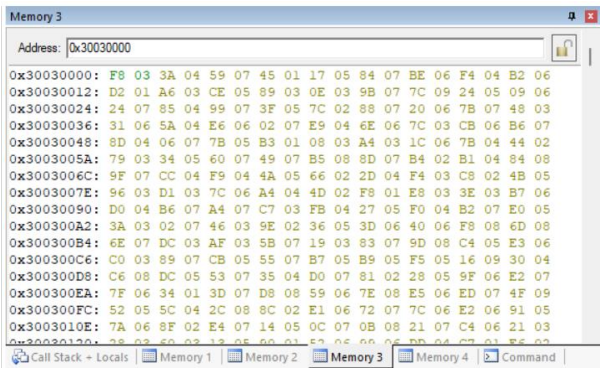
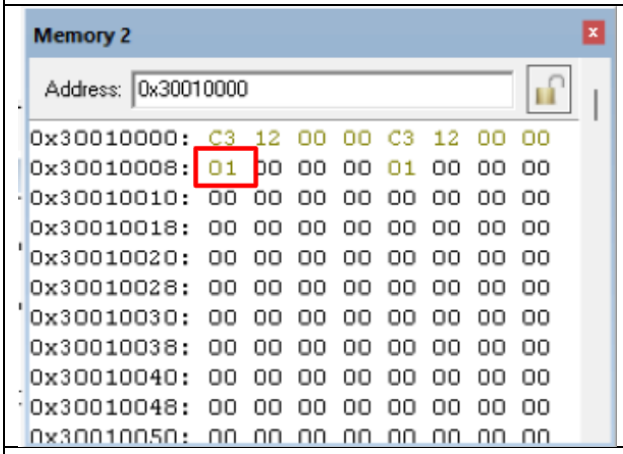
```
AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  convertGrayAsm
    ;   void convertGray(uint8_t* origin, uint16_t* toSave, int size) {
    ;   *toSave = 3 * (uint16_t)r + 6 * (uint16_t)g + (uint16_t)b
    ;   r0 = baseAddress
    ;   r1 = targetAddress
    ;   r2 = maxCount
    ;   r3 = converted
    ;
    ;   r4 = buf
convertGrayAsm
    STMFD sp!, {r4, lr}
CGA_L1
    ; r3 = r * 3
    LDRB    r3, [r0], #1
    ADD     r3, r3, r3, LSL #1
    ; r3 += g * 6
    LDRB    r4, [r0], #1
    ADD     r4, r4, r4, LSL #1
    ADD     r3, r3, r4, LSL #1
    ; r3 += b
    LDRB    r4, [r0], #2
    ADD     r3, r3, r4
    ; [r1] = r3
    STRH    r3, [r1], #2

    SUBS    r2, r2, #1
    BGT     CGA_L1
    LDMFD sp!, {r4, pc}
```

```
    ; BX lr;
END
```

<코드 8. convertGray 어셈블리 코드>

해당 어셈블리 코드는 RGB 데이터를 Grayscale 값으로 변환하여 저장하는 기능을 수행한다. convertGrayAsm 함수는 원본 이미지 주소(r0)에서 R, G, B 값을 순차적으로 불러온 뒤, 가중합 방식(Gray = 3×R + 6×G + B)에 따라 Grayscale 값을 계산한다. R 값에 대해 3×R 은 ADD r3, r3, r3, LSL #1 을 통해 효율적으로 처리하며, G 값에 대해서는 6×G = 2×(G + 2×G) 형태로 계산하여 누적한다. 마지막으로 B 값을 더한 후, 결과를 저장한다. 반복 횟수(r2)가 0 이 될 때까지 루프를 돌며 픽셀 단위로 변환이 수행되며, 전체 논리 흐름은 C 코드와 비슷하다.

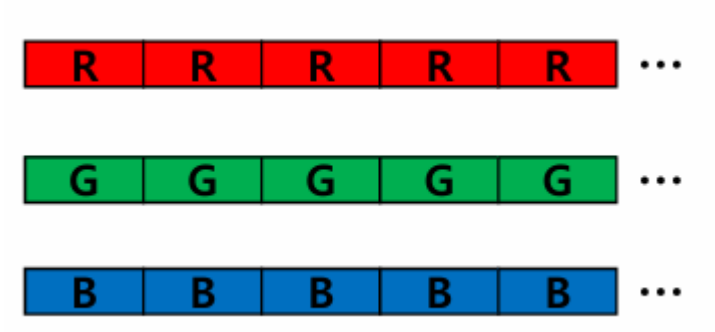
다음은 실행 결과이다:

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>[실행 전 메모리맵]</td><td>[실행 후 메모리맵]</td></tr>
<tr><td> 
</td><td></td></tr>
<tr><td>bool isSame 실행 결과</td><td></td></tr>
</table>

어셈블리 코드에서 실행한 결과 역시 Grayscale 변환된 값이 해당 주소에 정상적으로 저장된 것을 확인할 수 있었다. C 언어로 구현했을 때와 동일한 방식으로 동작함을 통해 기능이 정확히 일치함을 검증할 수 있었다.

IV. 메모리 재정렬

본 프로젝트에서는 이미지 처리 성능을 향상시키기 위해 기존의 32-bit RGBA 포맷에서 A(alpha) 채널을 제거하고, R, G, B 채널만을 별도의 메모리 공간에 각각 저장하는 방식의 Memory Relocation 기법을 적용하였다. 기존 RGBA 구조는 각 픽셀이 4 바이트 단위로 [R][G][B][A] 순서로 저장되며, 이로 인해 특정 채널 값에 접근하기 위해서는 4 바이트마다 건너뛰는 연산이 반복되어야 했다. 또한 pixel 의 시작이 word 단위 alignment 가 되어있지 않아서 word 단위 load 가 불가능하며 한 픽셀의 정보를 load 한번으로 가져올 수 없다. Memory Relocation 을 적용한 후에는 R, G, B 채널의 값들이 각각 독립된 배열에 연속적으로 저장되며, Alpha 채널은 제거된다. 예를 들어, Red 채널의 경우 [R0][R1][R2]... 형태로 순차적으로 저장되며, Green 과 Blue 도 동일한 방식으로 분리된다. 이러한 Planar 형태의 구조는 반복문을 통한 메모리 접근이 단순화되며, 채널별 데이터를 처리하는 과정에서 **주소** 연산 없이 연속 접근이 가능해진다.
또한, 채널별 데이터가 메모리에 연속적으로 배치됨으로써 캐시 효율성이 증가하고, 전체 이미지 처리 루틴의 메모리 병목이 완화되는 효과도 나타난다. 특히 연산에 불필요한 Alpha 채널을 제거함으로써 메모리 낭비가 줄어들고, 처리 대상이 명확해지는 구조적 장점도 있다.
결과적으로, 본 프로젝트의 Memory Relocation 기법은 단순한 데이터 재배치를 넘어, ARM 기반 연산 최적화와 성능 향상에 핵심적인 역할을 하였으며, 이후 구현된 Grayscale 변환, 색상 반전, Red pixel count 등의 연산에서도 효과적으로 활용되었다



IV-1. Main 함수

함수명	설명	결과 저장 주소
convertGray	8bit r, g, b → 16-bit Grayscale 변환	0x30020000 (C) 0x30030000 (ASM)
rRev, gRev, bRev	8bit r, g, b → 8bit Negative 변환	0x30040000 (C) 0x30050000 (ASM)
countRed	Red 값 ≥ 128 픽셀 수 계산	0x30010000 (C) 0x30010004 (ASM)
r, g, b	32bit RGBA → 8bit R, G, B reallocate	0x30000000

```c
#include <stdint.h>
#include <stdlib.h>

#include "find_IDAT.h"
#include "convertGray.h"
#include "reAllocation.h"
#include "isSame.h"

extern  void _sys_exit(int return_code);
extern  void convertReverseRelAsm(uint8_t* origin, uint8_t* toSave, int size);
extern  int countRedRelAsm(uint8_t* origin, int size);
extern  void convertGrayRelAsm(uint16_t* toSave, uint8_t* targetR, uint8_t* targetG, uint8_t*
targetB, int size);

int main(void) {
    const int PIXEL_COUNT = 9600;

    uint8_t* pixel_start = find_IDAT( (uint8_t*)0x40000000 ) + 4;

    uint8_t* r = (uint8_t*)0x30000000;
    uint8_t* g = r + PIXEL_COUNT;   // 0x30002580
    uint8_t* b = g + PIXEL_COUNT;   // 0x30004B00
```

```
    int* count = (int*)0x30010000;

    uint16_t* grayTarget = (uint16_t*)0x30020000;   // ~0x30024B00
    uint16_t* grayTarget2 = (uint16_t*)0x30030000;   // ~0x30024B00

    uint8_t* rRev = (uint8_t*)0x30040000;
    uint8_t* gRev = rRev + PIXEL_COUNT; // 0x30042580
    uint8_t* bRev = gRev + PIXEL_COUNT; // 0x30044B00

    uint8_t* rRev2 = (uint8_t*)0x30050000;
    uint8_t* gRev2 = rRev2 + PIXEL_COUNT; // 0x30052580
    uint8_t* bRev2 = gRev2 + PIXEL_COUNT; // 0x30054B00

    uint8_t* isSameReturn = (uint8_t*)(count + 2);

    // reallocate data
    reAllocation(pixel_start, r, g, b, PIXEL_COUNT);

    convertGrayRel(grayTarget, r, g, b, PIXEL_COUNT);
    convertGrayRelAsm(grayTarget2, r, g, b, PIXEL_COUNT);
    *isSameReturn = isSame_uint16(grayTarget, grayTarget2, PIXEL_COUNT);

    convertReverseRel(r, rRev, PIXEL_COUNT*3);
    convertReverseRelAsm(r, rRev2, PIXEL_COUNT*3);
    *(isSameReturn+4) = isSame_uint8(rRev, rRev2, PIXEL_COUNT*3);

    *count = countRedRel(r, PIXEL_COUNT);
    *(count + 1) = countRedRelAsm(r, PIXEL_COUNT);
    _sys_exit(0);
}
```

<코드 9. 메모리 재정렬 후 main.c>

Ⅳ-2. 메모리 재정렬 후의 Red pixel count

아래는 메모리 재정렬 후 C 언어 기반의 핵심 코드이다:

```c
int countRedRel(uint8_t* origin, int size) {
    int cnt = 0;
    for (int i = 0; i < size; i++) {
        cnt += *(origin++) > 127 ? 1 : 0;
    }
    return cnt;
}
```

<코드 10. 메모리 재정렬 후 countRed c 언어 코드>

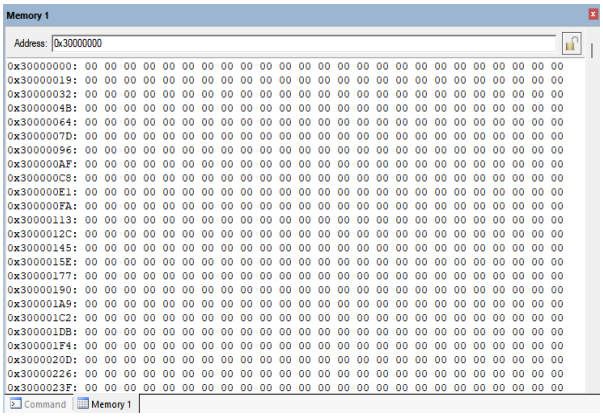
메모리 재정렬을 통해 C 언어 구현에서는 origin 포인터를 통해 RGBA 데이터를 순차적으로 접근하며, 각 픽셀의 Red 값을 추출하고 해당 값이 128 이상인 경우 누적 카운트를 증가시킨다. 이때 origin 포인터는 한 픽셀마다 1 바이트씩 이동하며, Alpha 채널은 연산에 포함되지 않는다.

아래는 메모리 재정렬 후 어셈블리 언어 기반의 핵심 코드이다:

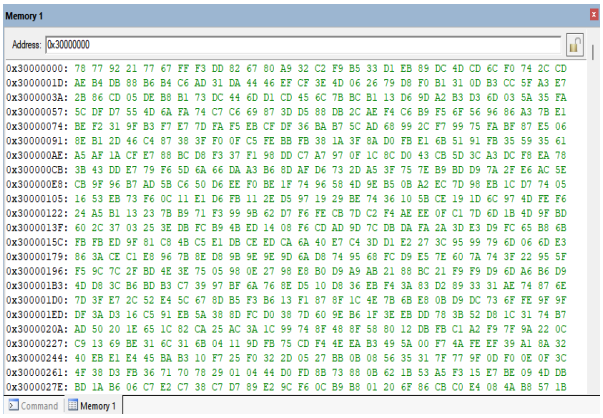
```
      AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  countRedRelAsmOr

    ; int countRedRelAsmOr(uint8_t* origin, int size)
    ;   r0 = baseAddress
    ;   r1 = maxCount
    ; return: int (r0)
    ;   r2 = cnt
    ;   r3 = buf

countRedRelAsmOr
    MOV     r2, #0
L1_cnt
    LDRB    r3, [r0], #1
    CMP     r3, #128
    ADDGE   r2, r2, #1
    SUBS    r1, r1, #1
```

```
    BGT     L1_cnt
    MOV     r0, r2
    BX lr


END
```

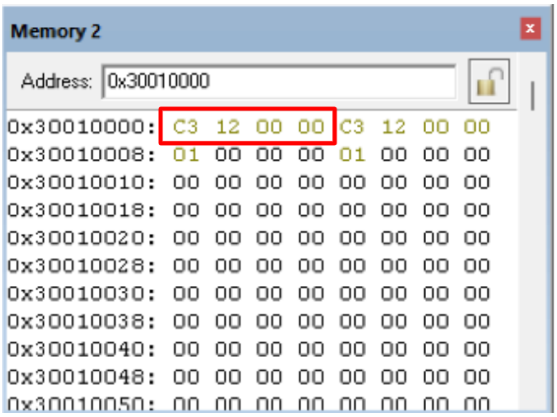
<코드 11. 메모리 재정렬 후 CountRed 어셈블리 코드>



[실행 전 메모리 맵]



[실행 후 메모리 맵]



[0X30010000 에 저장된 값]

픽셀의 Red 값이 128 이상인 픽셀 개수를
0x30010000 에 저장했음으로,
0x12C3(4803 개)이 9600 개 픽셀 중 RED 픽셀이
128 개 이상인 개수이다.

Ⅳ-2. 메모리 재정렬 후의 RGB 색상반전

아래는 메모리 재정렬 후의 convertReverse C 함수 구현이다:

```
void convertReverseRel(uint8_t* origin, uint8_t* toSave, int size) {
    for (int i = 0; i < size; i++) {
        *(toSave++) = 255 - *(origin++);
    }
}
```

<코드 12. 메모리 재정렬 후 convert Reverse c 언어 코드>

해당 C 코드는 Planar 구조로 재정렬된 RGB 데이터를 입력으로 받아, 각 바이트에 대해 단순 색상 반전을 수행하는 함수이다. convertReverseRel 함수는 RGB 데이터가 RRRR..., GGGG..., BBBB... 형태로 메모리에 연속 저장되어 있다는 전제를 기반으로 하며, 전체 데이터 크기만큼 반복문을 실행하여 각 바이트에 대해 255 - 원본 값 연산을 수행한 뒤 결과를 출력 포인터(toSave)에 저장한다.

아래는 메모리 재정렬 후의 convertReverse 어셈블리 함수 구현이다:

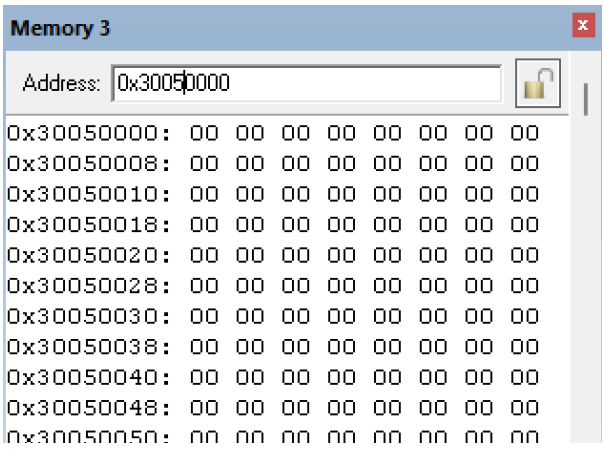
```
      AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  convertReverseRelAsmOr

    ;   void convertReverseRelAsmOr(uint8_t* origin, uint8_t* toSave, int size) {
    ;
    ;   r0 = baseAddress
    ;   r1 = targetAddress
    ;   r2 = maxCount
    ;   r3 = converted
    ;
    ;   r4 = buf

convertReverseRelAsmOr
    STMFD sp!, {r4, lr}
CRA_L1
    ; r4 = r
    LDRB    r4, [r0], #1
```

```
    MVN     r3, r4
    STRB    r3, [r1], #1

    SUBS    r2, r2, #1
    BGT     CRA_L1
    ;BX lr;
    LDMFD sp!, {r4, pc}

END
```

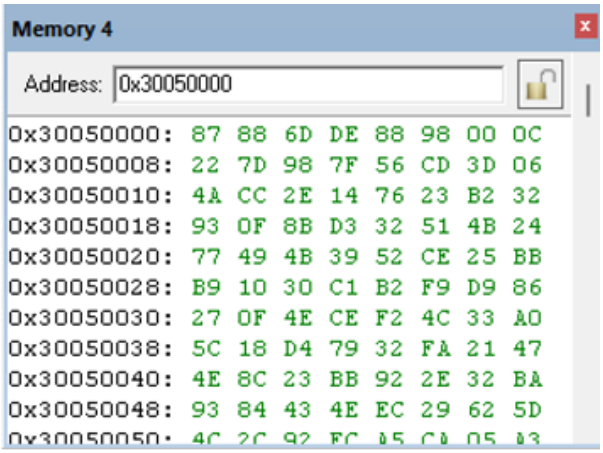
<코드 13. 메모리 재정렬 후의 convert Reverse 어셈블리 코드>

위 어셈블리 코드는 메모리 상의 RGB 데이터를 Planar 구조로 재정렬한 뒤, 각 채널 데이터를 순차적으로 접근하여 색상 반전을 수행하는 단순 구현이다. convertReverseRelAsmOr 함수는 재정렬된 메모리(RRRR..., GGGG..., BBBB...)를 기반으로 하며, 입력된 원본 주소(r0)로부터 각 채널 값을 1 바이트씩 읽어들여, MVN 명령어를 통해 비트를 반전시킨 후 대상 주소(r1)에 저장하는 구조이다. 각 루프에서는 RGB 중 하나의 채널 값만 처리되며, 반복문은 전체 바이트 수(r2)만큼 수행된다.

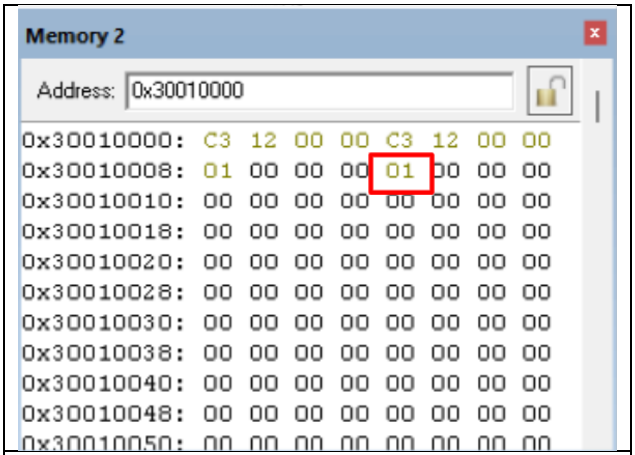
이 구현은 재정렬된 메모리 구조 덕분에 각 색상 채널을 독립적으로 처리할 수 있어 코드의 복잡도를 낮추면서도 기능 구현을 명확하게 할 수 있는 장점이 있다. 단순 루프 기반 처리로 고속 최적화는 적용되지 않았지만, 구조적으로 직관적이고, C 언어 기반의 색상 반전 로직과도 기능적으로 동일한 결과를 출력함을 통해 정상 동작을 확인할 수 있었다.



[실행 전 메모리맵]



[실행 후 메모리맵]



bool isSame 실행 결과 (reverse)

앞선 장의 메모리 재정렬 이전의 실행 결과와 동일한 결과값을 저장하는 것을 확인할 수 있었다. 재정렬 전후의 성능은 아래에서 performance analyzer 를 통해 한번에 비교 분석할 예정이다.

IV-3. 메모리 재정렬 후의 Grayscale

아래는 메모리 재정렬 후 C 언어 기반의 핵심 코드이다:

```
void convertGrayRel(uint16_t* toSave, uint8_t* targetR, uint8_t* targetG, uint8_t* targetB,
int size) {
    uint8_t r, g, b;
    for (int i = 0; i < size; i++) {
        // get color
        r = *(targetR++);
        g = *(targetG++);
        b = *(targetB++);
        // writeback
        *toSave++ = 3 * (uint16_t)r + 6 * (uint16_t)g + (uint16_t)b;
    }
}
```

<코드 14. 메모리 재정렬 후의 convert GrayScale C 언어 코드>

해당 C 코드는 Planar 구조로 재정렬된 RGB 데이터를 입력으로 받아, 각 바이트에 대해 단순 색상 반전을 수행하는 함수이다. convertGrayeRel 함수는 RGB 데이터가 RRRR..., GGGG..., BBBB... 형태로 메모리에 연속 저장되어 있다는 전제를 기반으로 하며, 전체 데이터 크기만큼 반복문을 실행하여 각 바이트에 대해 3×R + 6×G + B 연산을 수행한 뒤 결과를 출력 포인터(toSave)에 저장한다.

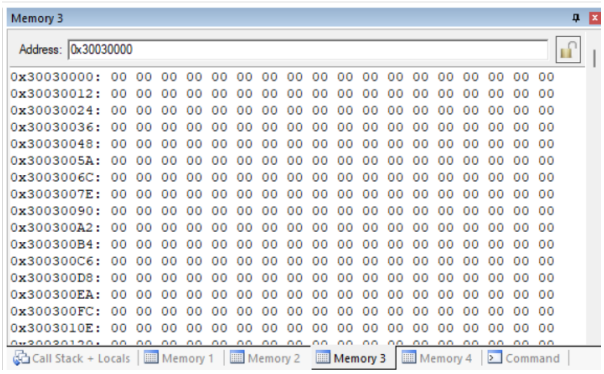
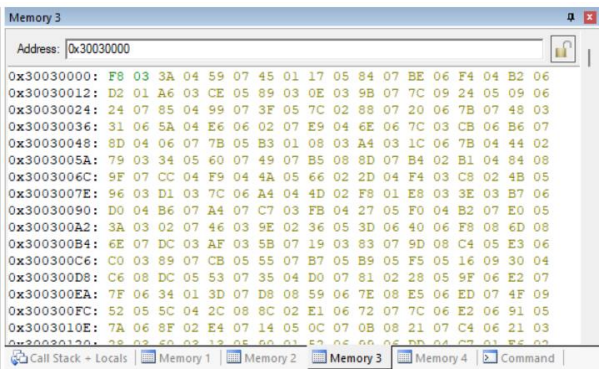
아래는 메모리 재정렬 후 어셈블리 언어 기반의 핵심 코드이다:

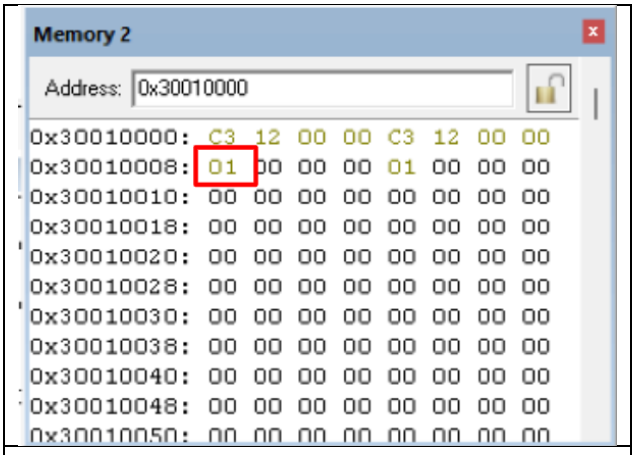
```
AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  convertGrayRelAsmOr

     ;    void convertGrayRelAsmOr(uint16_t* toSave, uint8_t* targetR, uint8_t* targetG,
uint8_t* targetB, int size)
         ;   *toSave = 3 * (uint16_t)r + 6 * (uint16_t)g + (uint16_t)b
    ;   r0 = toSave
    ;   r1 = targetR
    ;   r2 = targetG
    ;   r3 = targetB
    ;   r4 = size
    ;   r5 = r
    ;   r6 = g
    ;   r7 = b
    ;   r8 = buf1
    ;   r9 = buf2

convertGrayRelAsmOr
    STMFD   sp!, {r4-r9, lr}
    LDR     r4, [sp, #28]        ; r4 ← size

L1_gray
    LDRB  r5, [r1], #1
    LDRB  r6, [r2], #1
    LDRB  r7, [r3], #1
    MOV   r8, r5
    ADD   r8, r8, r5, LSL #1
    MOV   r9, r6, LSL #1
```

```
    ADD   r9, r9, r6, LSL #2
    ADD   r8, r8, r9
    ADD   r8, r8, r7
    STRH  r8, [r0], #2
    SUBS  r4, r4, #1
    BGT   L1_gray
    LDMFD sp!, {r4-r9, pc}
```

<코드 15. 메모리 재정렬 후의 convert GrayScale 어셈블리 코드>

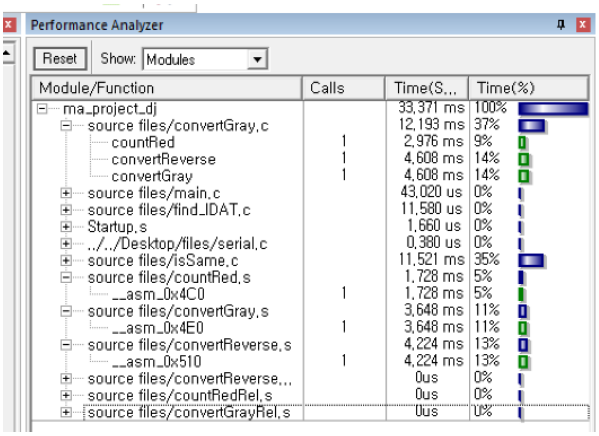
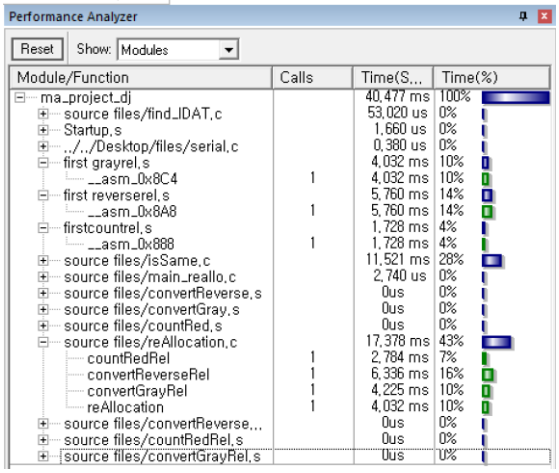
해당 어셈블리 코드는 메모리 상의 RGB 데이터를 Planar 구조로 재정렬한 뒤, 각 채널 데이터를 순차적으로 접근하여 RGB 데이터를 Grayscale 값으로 변환하여 저장하는 기능을 수행한다. convertGrayAsmOr 함수는 메모리(RRRR..., GGGG..., BBBB...)를 기반으로 하며, 입력된 원본 주소(r0)로부터 각 채널 값을 1 바이트씩 읽어 들여, 각 루프에서는 RGB 중 하나의 채널 값만 처리되며, 반복문은 전체 바이트 수(r4)만큼 수행된다. 값에 대해 3×R 은 ADD r5, r5, r5, LSL #1 을 통해 효율적으로 처리하며, G 값에 대해서는 6×G = (2×G) +(4×G) 형태로 계산하여 누적한다. 마지막으로 B 값을 더한 후, 결과를 저장한다. 반복 횟수(r4)가 0 이 될 때까지 루프를 돌며 픽셀 단위로 변환이 수행되며, 전체 논리 흐름은 C 코드와 비슷하다.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>[실행 전 메모리맵]</td>
<td>[실행 후 메모리맵]</td>
</tr>
</table>



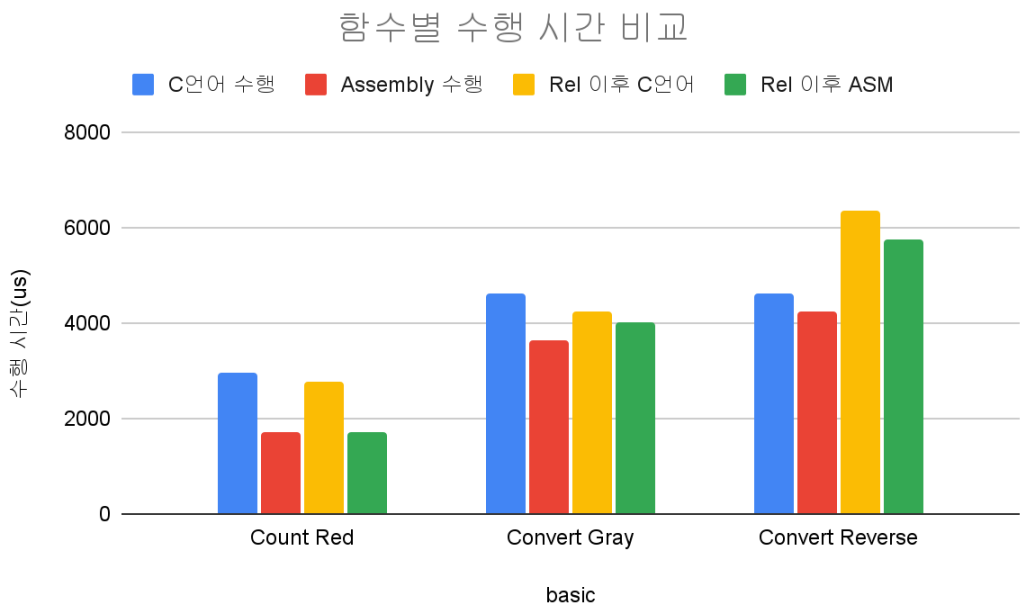
bool isSame 실행 결과 (Gray)

Ⅳ-4. Performance Analyzer 를 통한 정렬 전후 성능 비교

기존 Performance Analyzer 캡처	relocation Performance Analyzer 캡처
	

(us)	C 언어 수행	Assembly 수행	Rel 이후 C 언어	Rel 이후 ASM
Count Red	2976	1728	2784	1728
Convert Gray	4608	3648	4225	4032
Convert Reverse	4608	4224	6336	5760

<표: 주요 변환 함수의 수행 시간 정리>



<도표: 주요 변환 함수의 수행 시간 비교>

단순 Relocation 이후 추가적인 최적화 없이는 유의미한 속도 향상은 보이지 않았다. relocation 이후의 data 의 특성을 이용해 추가적인 최적화를 위해 분석이 필요함을 알 수 있다.

Ⅴ. ARM code level 최적화

Ⅴ-1. 최적화 논리

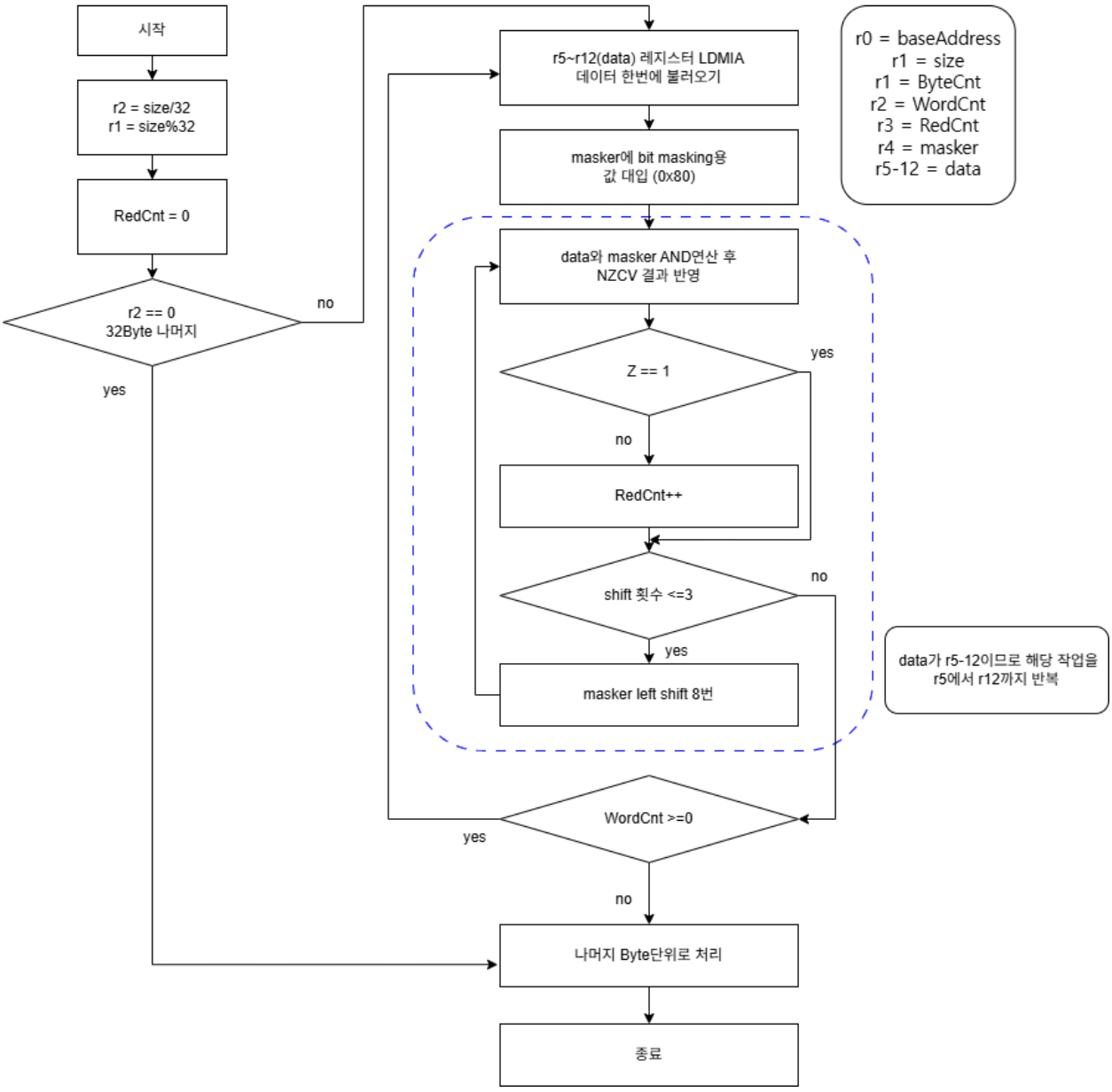
Arm code level 에서 추가적인 수행시간 향상을 위한 논리는 다음과 같다.

- 잦은 메모리 접근(LDR, STR)은 data bus 점유로 다음 instruction 을 느리게 만듦
    - Word, Half word, Byte 단위 Load, Store 는 모두 같은 사이클 점유
    - LDM, STM 을 이용하면 Cycle 절약 가능.
    - LDM, STM 사용시 AHP Bus 에 특성을 활용하여 32Byte 단위로 Burst 진행
    - 여러 Byte 단위(이하 Block 단위)로 Load, Store 하여 한 번에 많은 양을 block 단위로 처리
    - Block 단위에 맞지 않는 size 의 경우는 일반적인 Byte 단위로 처리
- 많은 Loop 횟수는 잦은 Branch 로 인해 cycle 손해
    - 고정된 횟수의 반복은 횟수 반복문이 아닌 정적인 명령으로 설계 (Loop Unrolling)
    - Block 단위 가속 설계를 통한 Loop 횟수 절감 (Loop 횟수 = (Size / Block) + (Size % Block))
- Binary 성질 이용
    - 255(0xFF) - X[7:0] = ~X[7:0] 성질을 이용하여 not 연산을 활용하여 실행속도를 높인다.
    - SUB 명령어를 쓰지 않고 MVN 명령어로 보다 빠르게 처리할 수 있다.
- Bit masking 연산 활용
    - if (X >= 128), X[7] = 1
    - 위와 같은 아이디어를 활용하여 (X[7:0] & 0X80)의 값이 1 이면 128 보다 크고 0 이면 128 보다 크다는 것을 알 수 있다.

위의 성질을 이용하여 LDM 명령어로 fetch, decode 과정을 최소화하여 한번에 많은 정보를 가져올 수 있게 하고 4Byte 를 하나의 Register 에서 연산하여 memory relocation 의 장점을 최대화하였다.

Ⅴ-2. Block Diagram

Red pixel counter 최적화 순서도



이번 연산은 총 32 Byte 단위로 연산하기 때문에 32Byte 로 나누어 떨어지지 않는 값들은 따로 처리해주어야 한다. (왼쪽 상단 로직)

앞서 설명한 bit masking 의 원리를 이용하여 작업을 수행하였다.

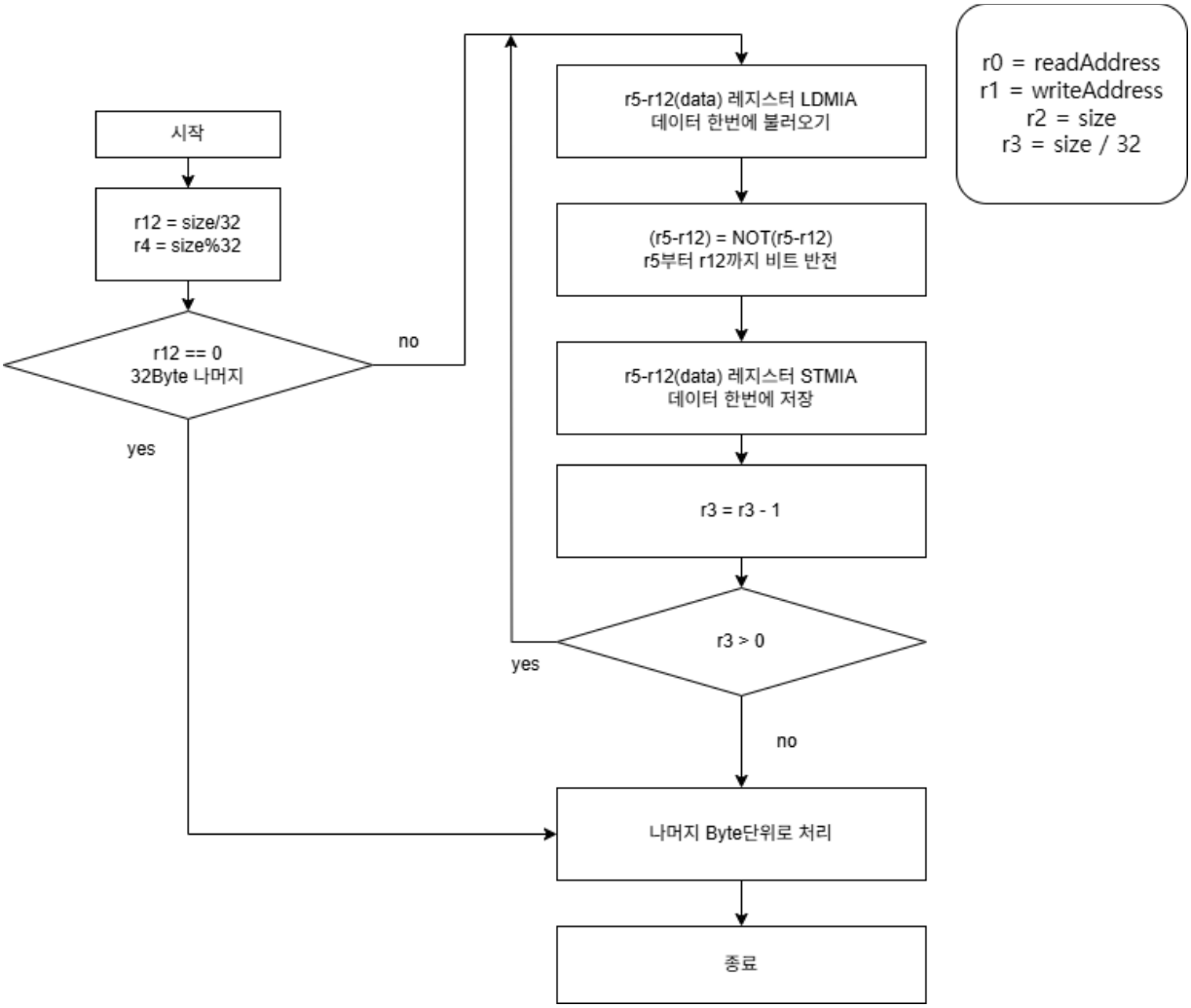
3	2	1	0
1000_0000	1000_0110	000_1100	1100_0100
0001_0000	0100_0110	000_1100	1110_0100

&

0000_0000	0000_0000	000_0000	1000_0000

4byte register 를 &연산 및 shift 연산을 활용하여 값이 128 보다 큰 red pixels 의 수를 셀 수 있다.

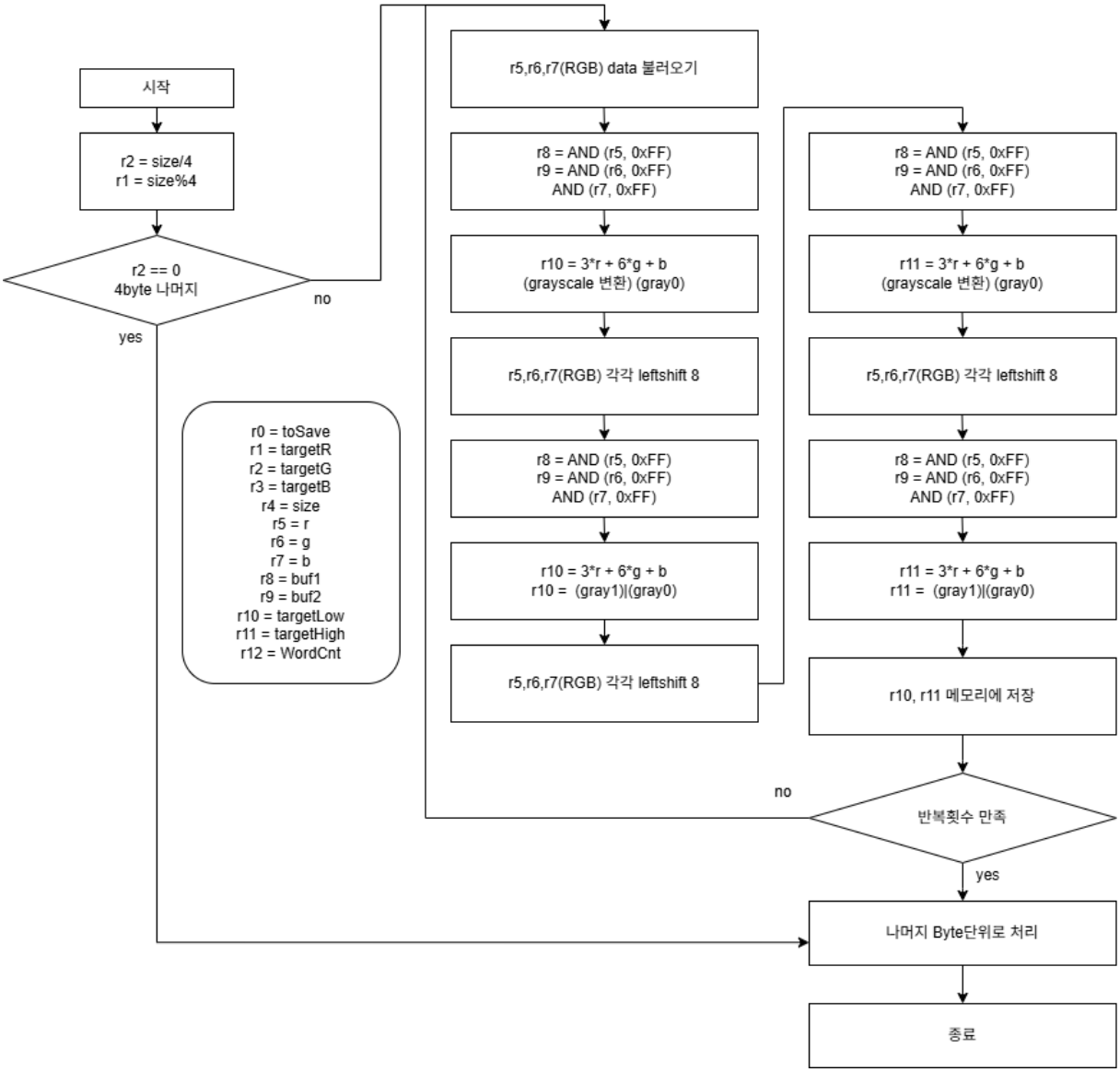
이미지 반전 최적화 순서도



마찬가지로 이번 연산은 총 32 Byte 단위로 연산하기 때문에 32Byte 로 나누어 떨어지지 않는 값들은 따로 처리해주어야 한다. (왼쪽 상단 로직)

앞서 설명한 Binary 연산의 성질을 이용하여 작업을 수행하였다.

gray convert 최적화 순서도



이번 연산은 총 4 Byte 단위로 연산하기 때문에 4Byte 로 나누어 떨어지지 않는 값들은 따로 처리해주어야 한다. (왼쪽 상단 로직)

grayscale 의 결과값은 16bit 으로 나오므로 한 레지스터에 2 개의 정보를 넣을 수 있다. 따라서 r10 에 두 개의 grayscale, r11 의 두개의 grayscale 총 4 개의 grayscale 값을 메모리에 한 번에 보낼 수 있다.

V-3. 최적화 된 Red Pixel Count

```
AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  countRedRelAsm

    ; int countRedRelAsm(uint8_t* origin, int size)
    ;   r0 = baseAddress
    ;   r1 = size
    ;   return: int (r0)
    ;   r1 = ByteCnt
    ;   r2 = WordCnt
    ;   r3 = RedCnt
    ;   r4 = masker
    ;   r5-12 = data

countRedRelAsm
    STMFD   sp!, {r4-r12, lr}

    MOV     r2, r1, LSR #5       ; r2 = size / 32
    SUB     r1, r1, r2, LSL #5   ; r1 = size % 32
    MOV     r3, #0;

    ; skip CNT_BLOCK_LOOP
    CMP     r2, #0
    BEQ     CNT_BYTE_LOOP

CNT_BLOCK_LOOP
    LDMIA       r0!, {r5-r12} ; info variable

    ; ========== word_0 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r5, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
```

```
    MOV     r4, r4, LSL #8
    TST     r5, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r5, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r5, r4
    ADDNE   r3, r3, #1

    ; ========= word_1 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r6, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
    MOV     r4, r4, LSL #8
    TST     r6, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r6, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r6, r4
    ADDNE   r3, r3, #1

    ; ========= word_2 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r7, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
```

```
    MOV     r4, r4, LSL #8
    TST     r7, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r7, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r7, r4
    ADDNE   r3, r3, #1

    ; ========= word_3 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r8, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
    MOV     r4, r4, LSL #8
    TST     r8, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r8, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r8, r4
    ADDNE   r3, r3, #1

    ; ========= word_4 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r9, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
```

```
    MOV     r4, r4, LSL #8
    TST     r9, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r9, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r9, r4
    ADDNE   r3, r3, #1

    ; ========= word_5 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r10, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
    MOV     r4, r4, LSL #8
    TST     r10, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r10, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r10, r4
    ADDNE   r3, r3, #1

    ; ========= word_6 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r11, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
```

```
    MOV     r4, r4, LSL #8
    TST     r11, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r11, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r11, r4
    ADDNE   r3, r3, #1

    ; ========= word_7 ==========
    ; pixel 0
    MOV     r4, #128
    TST     r12, r4; masking if is 0 then <128
    ADDNE   r3, r3, #1; sum++; when Z bit not 1
    ; pixel 1
    MOV     r4, r4, LSL #8
    TST     r12, r4
    ADDNE   r3, r3, #1
    ; pixel 2
    MOV     r4, r4, LSL #8
    TST     r12, r4
    ADDNE   r3, r3, #1
    ; pixel 3
    MOV     r4, r4, LSL #8
    TST     r12, r4
    ADDNE   r3, r3, #1

    ; check loop condition
    SUBS    r2, r2, #1
    BGT     CNT_BLOCK_LOOP

CNT_BYTE_LOOP
    ; skip CNT_BYTE_LOOP_BODY
```

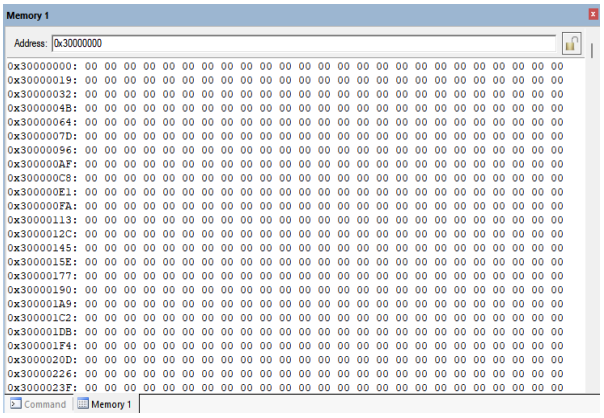
```
    CMP     r1, #0
    BEQ     EXIT

CNT_BYTE_LOOP_BODY
    LDRB    r5, [r0], #1
    CMP     r5, #128
    ADDGE   r3, r3, #1
    SUBS    r1, r1, #1
    BGT     CNT_BYTE_LOOP_BODY

EXIT
    MOV     r0, r3
    LDMFD   sp!, {r4-r12, pc}
```

<코드 16. 최적화된 convert Reverse 어셈블리 코드>

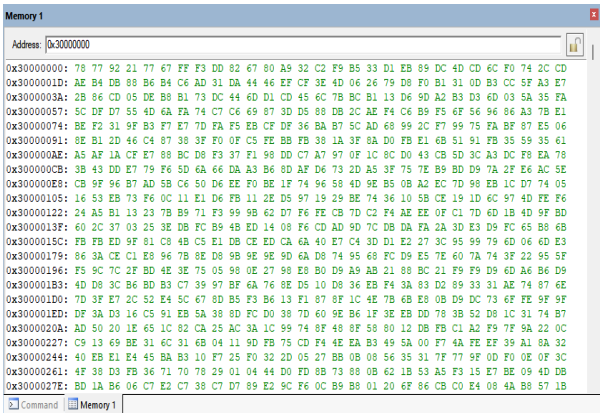
해당 어셈블리 코드는 RGB 이미지 데이터를 채널 재정렬(Planar 구조)된 상태를 전제로 하여, 향상된 성능의 red pixel count 를 수행하도록 최적화된 구조로 작성되었다.

countRedRelAsm 함수는 입력된 원본 포인터(r0), 처리할 픽셀의 크기(r1)를 인자로 받아 동작한다. r5 부터 r12 레지스터에 R 값들을 4byte * 8(총 32byte)를 차례로 처리한다. RRRR 각각 8bit 씩 비교해야 하므로 bit masking 으로 하위 8bit 을 추출하고 연산 후 shift 하여 다음 8bit 을 연산한다. shift 3 번은 고정이므로 성능향상을 위해 분기를 사용하지 않고 작성하였다. 남은 바이트는 바이트 단위 루프를 통해 동일한 방식으로 처리하며, 마지막에는 스택에 저장해 두었던 레지스터 상태를 복원하고 함수 실행을 종료한다.

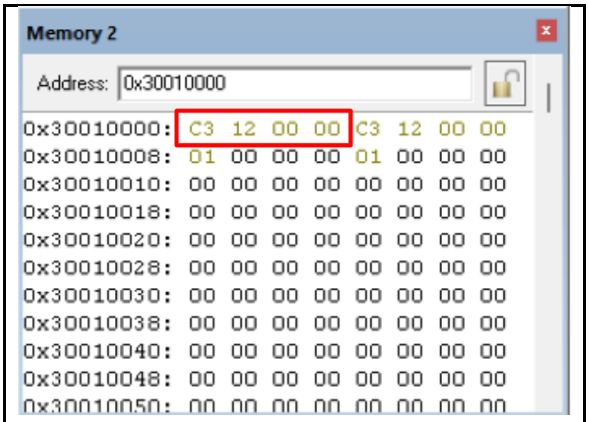
다음은 실행 결과이다.



[실행 전 메모리 맵]



[실행 후 메모리 맵]



[0X30010000 에 저장된 값]

V-3. 최적화 된 RGB 색상반전

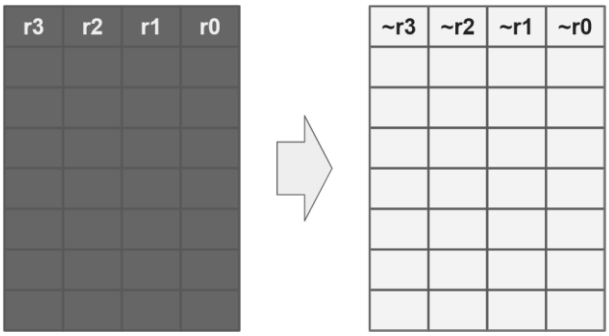
아래는 최적화 된 convertReverse 함수 구현이다:

```
AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  convertReverseRelAsm
    ;   void convertReverseRelAsm(uint8_t* origin, uint8_t* toSave, int size)
    ;   r0 = origin
    ;   r1 = toSave
    ;   r2 = size
    ;
    ;   r3 = size / 32; L1
    ;   r2 = mod(size % 32); L2

convertReverseRelAsm
    STMFD   sp!, {r4-r12, lr}

    MOV     r3, r2, LSR #5       ; r3 = size / 32
    SUB     r2, r2, r3, LSL #5   ; r2 = size % 32

    ; skip REV_BLOCK_LOOP
    CMP     r3, #0
    BEQ     REV_BYTE_LOOP

REV_BLOCK_LOOP
    ; invert block
    LDMIA   r0!, {r5-r12}
    MVN     r5, r5
    MVN     r6, r6
    MVN     r7, r7
    MVN     r8, r8
    MVN     r9, r9
    MVN     r10, r10
    MVN     r11, r11
    MVN     r12, r12
    STMIA   r1!, {r5-r12}
```

```
    ; check loop condition
    SUBS    r3, r3, #1
    BGT     REV_BLOCK_LOOP

REV_BYTE_LOOP
    ; skip REV_BYTE_LOOP_BODY
    CMP     r2, #0
    BEQ     REV_EXIT

REV_BYTE_LOOP_BODY
    LDRB    r3, [r0], #1
    MVN     r3, r3
    STRB    r3, [r1], #1
    SUBS    r2, r2, #1
    BGT     REV_BYTE_LOOP_BODY

REV_EXIT
    LDMFD   sp!, {r4-r12, pc}
```

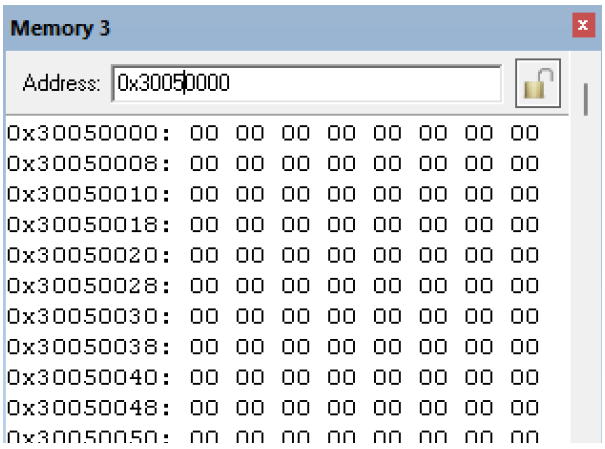
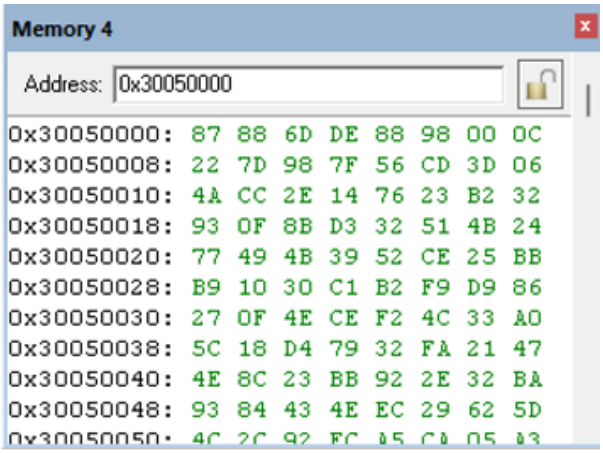
<코드 17. 최적화된 convertReverse 어셈블리 코드>

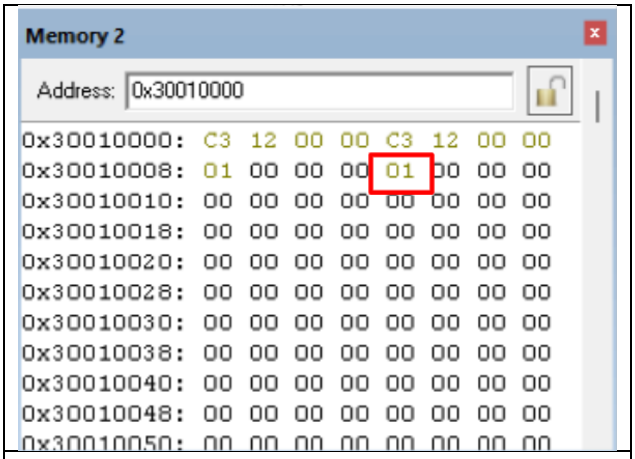
해당 어셈블리 코드는 RGB 이미지 데이터를 채널 재정렬(Planar 구조)된 상태를 전제로 하여, 향상된 성능의 색상 반전을 수행하도록 최적화된 구조로 작성되었다. convertReverseRelAsm 함수는 입력된 원본 포인터(r0), 저장할 대상 포인터(r1), 그리고 전체 데이터 크기(r2)를 인자로 받아 동작하며, 전체 데이터를 32 바이트(=8 워드) 단위로 묶어 처리하는 블록 단위 연산과 남은 바이트를 개별 처리하는 방식으로 구성되어 있다.



특히 이 구조는 RGB 가 RRRR..., GGGG..., BBBB...처럼 연속적으로 구성된 Planar 형식으로 재정렬되어 있기 때문에, 메모리 접근 효율을 극대화할 수 있다. LDMIA 명령어로 연속된 8 워드를 한 번에 로드하고, 각 워드에 대해 MVN 명령어를 통해 word 단위로 색상 반전을 수행한 뒤, STMIA 명령어로 일괄 저장함으로써 하나의 루프에서 32 바이트를 처리할 수 있다. 이러한 방식은 LDRB → RSB → STRB 방식식의 연산보다 훨씬 적은 명령어 수와 execute 단계로 동일한 결과를 구현하여, 반복 횟수와 처리 시간을 크게 줄일 수 있도록 설계하였다.

남은 바이트는 바이트 단위 루프를 통해 동일한 방식으로 처리하며, 마지막에는 스택에 저장해두었던 레지스터 상태를 복원하고 함수 실행을 종료한다. 이 최적화 기법은 ARM 아키텍처의 메모리 정렬 특성을 적극 활용하며, Planar 형식 재정렬과 병렬 처리를 결합함으로써 기능적 정확성과 함께 전반적인 연산 성능 향상을 동시에 달성하였다. 최적화 코드 실행 결과 이전의 convertReverse 코드와 동일한 결과값을 저장하는 것을 확인할 수 있었다. 최적화 전후의 성능 비교는 다음 장에서 전체 코드에 대해서 비교 분석할 예정이다.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>[실행 전 메모리맵]</td>
<td>[실행 후 메모리맵]</td>
</tr>
</table>



bool isSame 실행 결과 (reverse)

V-4. 최적화된 Grayscale

아래는 최적화된 convertGray 함수 구현이다:

```
    AREA    CODE, READONLY, CODE
    ENTRY
    EXPORT  convertGrayRelAsm
    ;   void convertGrayRelAsm(uint16_t* toSave, uint8_t* targetR, uint8_t* targetG, uint8_t*
targetB, int size)
    ;   *toSave = 3 * (uint16_t)r + 6 * (uint16_t)g + (uint16_t)b
    ;   r0 = toSave
    ;   r1 = targetR
    ;   r2 = targetG
    ;   r3 = targetB
    ;   r4 = size
    ;   r5 = r
    ;   r6 = g
    ;   r7 = b
    ;   r8 = buf1
    ;   r9 = buf2
    ;   r10 = targetLow
    ;   r11 = targetHigh
    ;   r12 = WordCnt
```

```
convertGrayRelAsm
    STMFD   sp!, {r4-r12, lr}
    LDR     r4, [sp, #40]        ; r4 ← size

    MOV     r12, r4, LSR #2      ; r12 = size / 4
    SUB     r4, r4, r12, LSL #2  ; r2 = size % 4

    ; skip GRAY_BLOCK_LOOP
    CMP   r12, #0
    BEQ   GRAY_BYTE_LOOP

GRAY_BLOCK_LOOP
    ; R, G, B block load (4 bytes each)
    LDR   r5, [r1], #4         ; r5 = R0 R1 R2 R3
    LDR   r6, [r2], #4         ; r6 = G0 G1 G2 G3
    LDR   r7, [r3], #4         ; r7 = B0 B1 B2 B3

    ; --- Pixel 0 ---
    AND   r8, r5, #0xFF
    ADD   r8, r8, r8, LSL #1         ; R × 3

    AND   r9, r6, #0xFF
    ADD   r9, r9, r9, LSL #1
    ADD   r8, r8, r9, LSL #1         ; G × 6

    AND   r9, r7, #0xFF
    ADD   r8, r8, r9                 ; gray0

    MOV   r10, r8        ; gray0

    ; --- Pixel 1 ---
    MOV   r8, r5, ROR #8
    AND   r8, r8, #0xFF
    ADD   r8, r8, r8, LSL #1         ; R × 3

    MOV   r9, r6, ROR #8
```

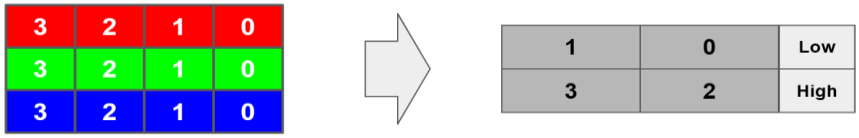
```
    AND   r9, r9, #0xFF
    ADD   r9, r9, r9, LSL #1
    ADD   r8, r8, r9, LSL #1         ; G × 6

    MOV   r9, r7, ROR #8
    AND   r9, r9, #0xFF
    ADD   r8, r8, r9                 ; gray1

    LSL   r8, r8, #16
    ORR   r10, r10, r8               ; gray1<<16 ¦ gray0

    ; --- Pixel 2 ---
    MOV   r8, r5, ROR #16
    AND   r8, r8, #0xFF
    ADD   r8, r8, r8, LSL #1   ; R × 3

    MOV   r9, r6, ROR #16
    AND   r9, r9, #0xFF
    ADD   r9, r9, r9, LSL #1
    ADD   r8, r8, r9, LSL #1   ; G × 6

    MOV   r9, r7, ROR #16
    AND   r9, r9, #0xFF
    ADD   r8, r8, r9         ; gray2

    MOV   r11, r8        ; gray2

    ; --- Pixel 3 ---
    MOV   r8, r5, ROR #24
    AND   r8, r8, #0xFF
    ADD   r8, r8, r8, LSL #1      ; R × 3

    MOV   r9, r6, ROR #24
    AND   r9, r9, #0xFF
    ADD   r9, r9, r9, LSL #1
    ADD   r8, r8, r9, LSL #1     ; G × 6
```

```
    MOV   r9, r7, ROR #24
    AND   r9, r9, #0xFF
    ADD   r8, r8, r9           ; gray3

    LSL   r8, r8, #16
    ORR   r11, r11, r8         ; gray3<<16 | gray2

    STMIA r0!, {r10, r11}      ; 저장: gray0~3

    SUBS  r12, r12, #1
    BGT   GRAY_BLOCK_LOOP

GRAY_BYTE_LOOP
    CMP   r4, #0
    BEQ   GRAY_EXIT

GRAY_BYTE_LOOP_BODY
    LDRB  r5, [r1], #1
    LDRB  r6, [r2], #1
    LDRB  r7, [r3], #1
    MOV   r8, r5
    ADD   r8, r8, r5, LSL #1
    MOV   r9, r6, LSL #1
    ADD   r9, r9, r6, LSL #2
    ADD   r8, r8, r9
    ADD   r8, r8, r7
    STRH  r8, [r0], #2
    SUBS  r12, r12, #1
    BGT   GRAY_BYTE_LOOP_BODY

GRAY_EXIT
    LDMFD sp!, {r4-r12, pc}
```

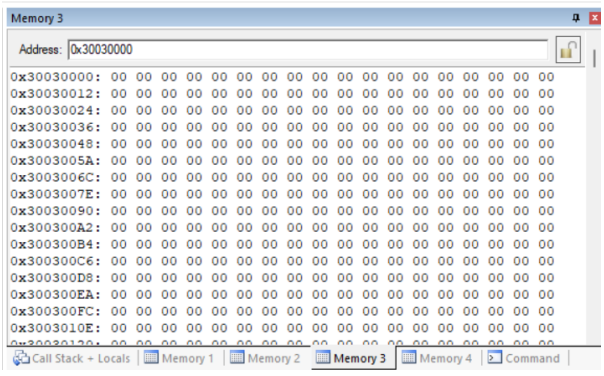
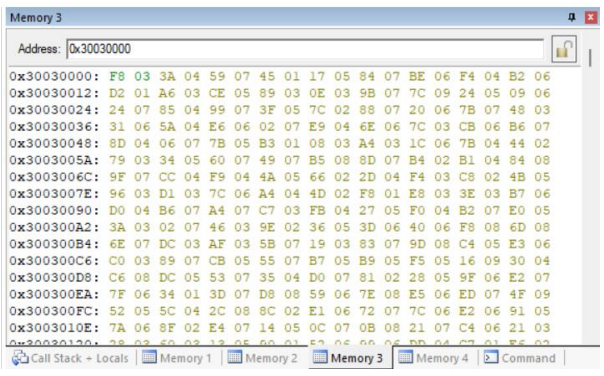
<코드 18. 최적화된 convertGray 어셈블리 코드>

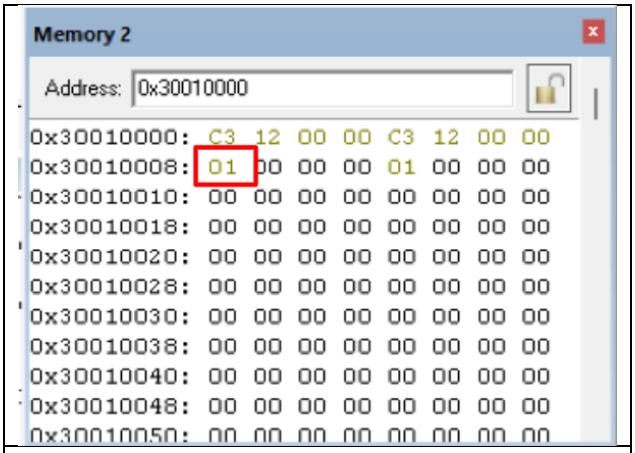
해당 어셈블리 코드는 Planar 형식으로 재정렬된 RGB 데이터를 입력으로 받아, Gray = 3×R + 6×G + B 의 가중합 방식에 따라 Grayscale 변환을 수행하는 함수이다. convertGrayRelAsm 함수는 각 채널(R, G, B)의 포인터(r1, r2, r3)를 독립적으로 받아 각각의 데이터를 접근하며, 성능 최적화를 위해 4 픽셀씩 처리하는 블록 처리 구조와 남은 데이터를 위한 바이트 처리 루프로 구성되어 있다.

블록 처리 루프에서는 R, G, B 채널 각각에서 4 바이트를 LDR 명령어로 로드한 후, 각 픽셀의 R, G, B 값을 비트 마스킹(AND)과 순환 이동(ROR)을 통해 분리한다. 분리된 각 값에 대해 3×R + 6×G + B 계산이 수행되며, 계산된 Grayscale 값은 16 비트로 저장된다. 두 개의 Grayscale 값을 하나의 32 비트 워드로 결합한 뒤(gray1 << 16 | gray0, gray3 << 16 | gray2), STMIA 명령어를 통해 출력 주소에 저장함으로써 4 픽셀당 2 번의 저장으로 효율적인 데이터 기록이 가능하다.



루프가 끝난 뒤 남은 픽셀 수(size % 4)에 대해서는 바이트 단위 루프에서 동일한 연산을 수행한다. 이 구조는 채널 재정렬로 인해 메모리 접근 효율이 높아졌으며, 동시에 비트 연산과 레지스터 조합을 통해 연산 속도를 최적화하였다. 결과적으로 실행 단계의 수를 줄이면서도 C 코드와 동일한 정확한 Grayscale 변환 결과를 출력함을 확인할 수 있었고, 이는 Planar 구조와 병렬 계산의 결합을 통한 최적화된 구현 사례로 볼 수 있다. 최적화 코드 실행 결과 이전의 convertGray 코드와 동일한 결과값을 저장하는 것을 확인할 수 있었다. 최적화 전후의 성능 비교는 다음 장에서 전체 코드에 대해서 비교 분석할 예정이다.

	
[실행 전 메모리맵]	**[실행 후 메모리맵]**

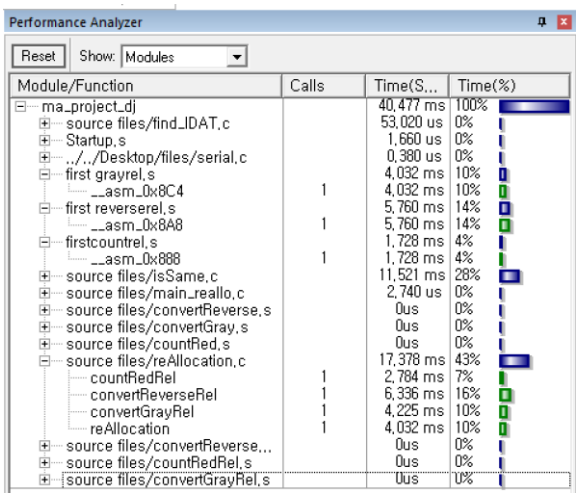
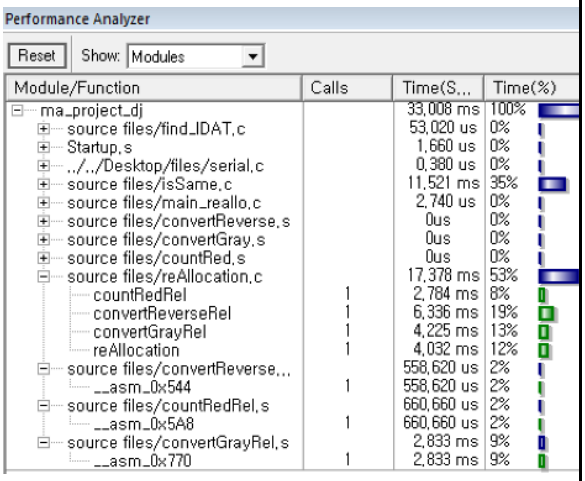


bool isSame 실행 결과 (Gray)

VI. 최적화 전후 비교

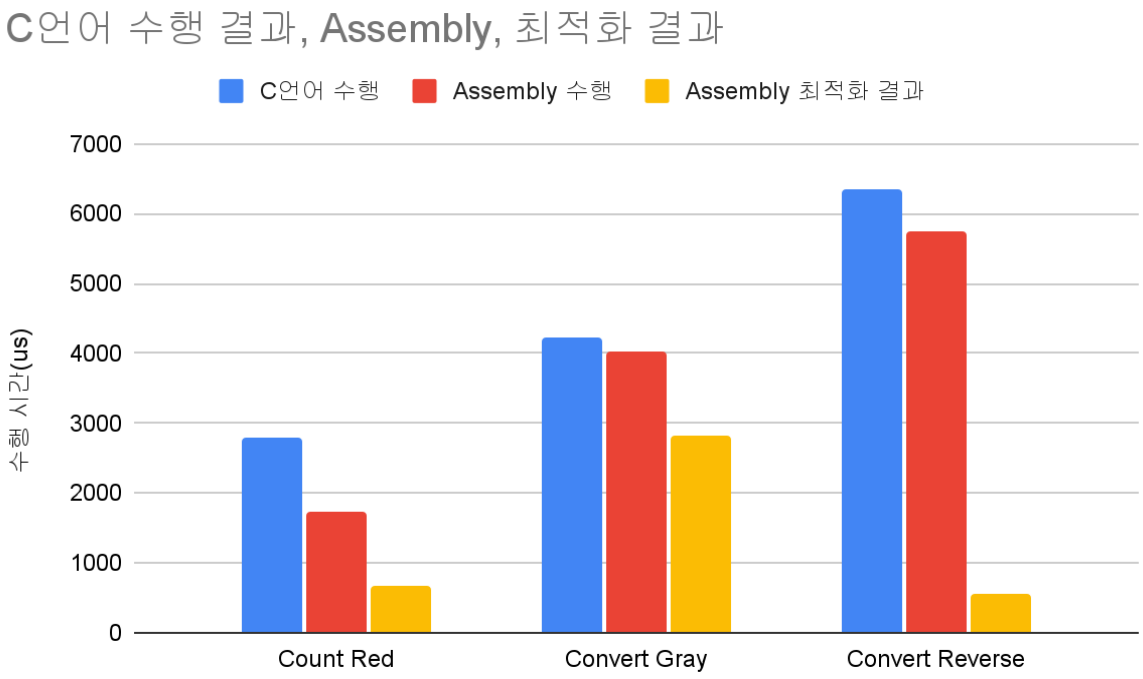
VI- 1. 최종 최적화 전후 성능 비교

위와 같이 ARM 코드 및 추가적인 논리를 이용한 최적화를 구현하였다. 다음은 수행 시간 결과이다.

최적화 이전 Performance analyzer 캡처	최적화 이후 Performance analyzer 캡처
	

(us)	C 언어 수행	Assembly 수행	Assembly 최적화 결과	C 대비 수행 시간
Count Red	2784	1728	660.66	23.73%
Convert Gray	4225	4032	2833	67.05%
Convert Reverse	6336	5760	558.62	8.82%

<표: 주요 변환 함수의 수행 시간 정리>



<도표: 주요 변환 함수의 수행 시간 비교>

Ⅶ. 결론

본 프로젝트에서는 32-bit RGBA 형식의 이미지 데이터를 기반으로 Red pixel count, RGB 색상 반전, Grayscale 변환의 세 가지 기능을 C 언어와 ARM 어셈블리 코드로 구현하고, 성능 최적화를 통해 효율적인 연산 구조를 도출하였다. 초기에는 픽셀 단위의 단순 반복 연산 구조로 기능의 정확성을 검증한 뒤, 이후 Planar 구조의 메모리 재정렬(Memory Relocation)을 적용함으로써 데이터 접근의 단순화와 성능 향상의 기반을 마련하였다.

재정렬된 메모리 구조를 기반으로 한 최적화 어셈블리 구현에서는 LDMIA/STMIA 명령어를 활용한 블록 단위 연산과 loop unrolling, 비트 마스킹, MVN 명령어 최적화 등 다양한 저수준 기법이 적용되었으며, 그 결과 수행 시간이 크게 단축되었다. 예를 들어, Count Red 연산의 경우 최적화 전 1728us 에서 최적화 후 660us 수준으로 감소하였고, Convert Reverse 또한 558us 수준까지 개선되는 등 성능 향상이 수치적으로 확인되었다.

또한, 각 어셈블리 함수는 C 언어로 구현된 동일 기능과 메모리 출력 결과가 완벽히 일치함을 통해 기능적 정확성 역시 확보할 수 있었다. 실험 전후의 메모리 상태 비교 및 isSame 함수 검증을 통해 모든 변환 함수가 의도대로 동작함을 확인하였다.

결론적으로, 본 프로젝트는 단순한 기능 구현을 넘어 ARM 아키텍처에 적합한 메모리 구조와 명령어 최적화를 설계에 반영함으로써, 시스템 수준에서 연산 성능을 향상시키는 방법론을 실험적으로 증명하였다.

VIII. 문제 해결 과정

1. **이미지 Save/Load 과정에서의 문제**
    - ○ Keil 환경에서 이미지 파일을 메모리에 로드할 때 Save/Load 명령어에 대한 이해 부족으로 인해 데이터가 정확히 메모리에 배치되지 않는 오류 발생.
2. **함수 호출 시 파라미터 전달 오류**
    - ○ C 언어의 main() 함수에서 최적화된 어셈블리 함수를 호출할 때, 파라미터가 5 개 이상일 경우 예상대로 동작하지 않는 현상 발생.
3. **변수 오염 및 디버깅 어려움**
    - ○ ARM 코드 디버깅 중 함수 호출 이후 변수 값이 비정상적으로 변경됨. 이는 C 언어로 구현 시 **Heap, Code, Stack 영역에 대한 이해 부족**으로 인해 메모리 간섭이 발생한 것으로 추정됨.

해결 방법

1. **정확한 메모리 주소 설정**
    - ○ Keil 의 Debug 모드에서 이미지 로드 주소와 실행 시작 주소를 명확하게 지정하여 이미지 데이터가 메모리상 정확한 위치에 로드되도록 수정함.
2. **코드 및 데이터 세그먼트 분리**
    - ○ C 기반 project 생성시, data 를 load 한 영역과 임의로 지정한 포인터, 변수 영역이 서로를 침범하여 data 가 오염되는 문제가 발생했었음. scatter 파일상에서 이 영역을 명확히 분리하여 Heap/Stack 간섭 문제를 해결. 변수 오염을 방지함.
3. **Stack 기반 파라미터 전달 방식 적용**
    - ○ relocation 이후 어셈블리 함수에서 5 개 이상의 파라미터가 필요한데, APCS 에 따라 r0-r3 4 개 만의 레지스터만으로는 전달이 불가능했음. 따라서 의도된 동작을 하지 못했는데, APCS 표준에 대해서 이해하여 Stack 을 통해 인자를 전달받도록 수정.

IX. 프로젝트 수행일지

이름	김동준(조장)	안덕찬	우상욱	이장근
모임일자	수행내용			
5/8	- 프로젝트개요에 대한 이해 - Keil 설치 및 기초 사용법 스터디 - Debug 모드에서의 메모리 Load 방법 학습			
5/15	- 각자 구현한 과제 1, 2, 3 번 코드 리뷰 - 코드별 성능 비교 분석 및 개선 방향 토의			
5/22	- 메모리 relocation 을 포함한 코드 성능 비교 및 그래프화 - 최적화 전후의 성능 변화 분석			
5/25	- 보고서 및 발표자료 최종 작성 - 프로젝트 결과 발표 준비			
6/5	- 보고서 점검 및 ppt 제작 - 발표 영상 녹화			