

---

# Microprocessor Application

Project – ARM Code Optimization for Image Converting  
with Keil MDK tool



---

학과	소프트웨어학부	이름(학번)	소희연 (20220221)
학과	전자정보공학부 IT융합전공	이름(학번)	신주환 (20212979)
학과	글로벌미디어학부	이름(학번)	유진 (20213596)

---

## 1. 프로젝트 개요

본 프로젝트는 ARM 기반 마이크로프로세서 환경에서 동작하는 이미지 변환 기능을 직접 구현하고, 성능 최적화를 수행하는 것을 목표로 한다. 저수준 프로그래밍 언어인 ARM 어셈블리를 활용하여 이미지 데이터를 처리하고, 연산 효율 및 메모리 구조에 대한 이해를 기반으로 코드 수준에서의 최적화를 실현한다.

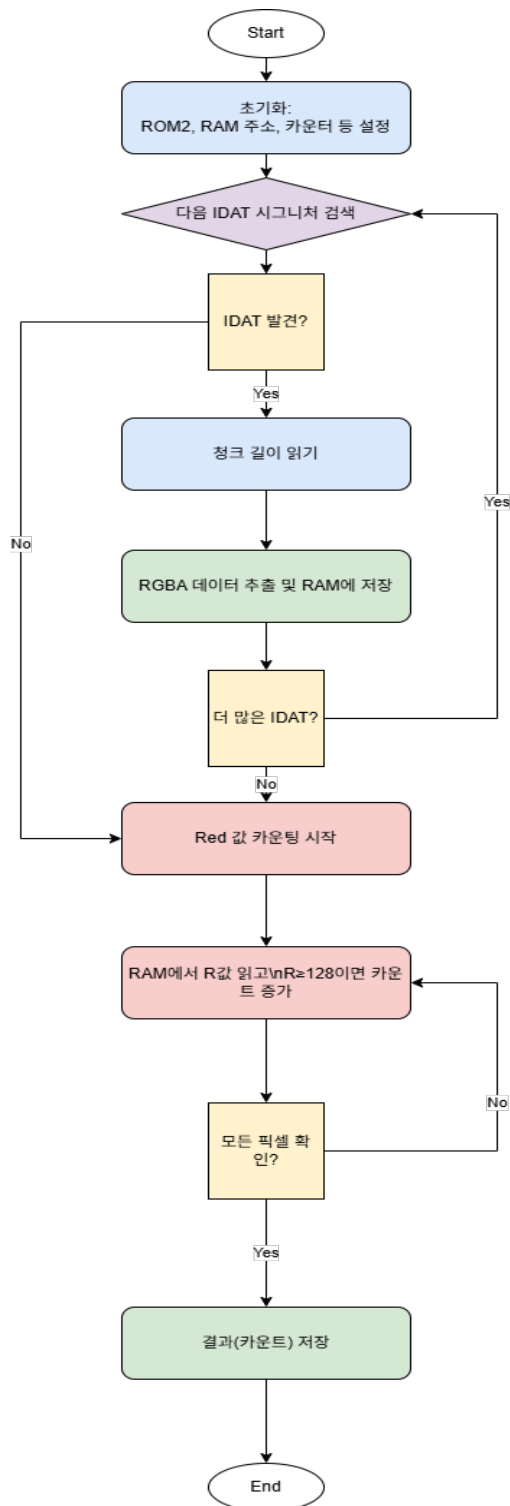
우리는 총 세 가지의 주요 기능을 구현하였다. 첫째, RGBA 이미지에서 Red 값이 128 이상인 픽셀의 개수를 계산하는 Red Pixel Count 기능이다. 둘째, 각 픽셀의 RGB 값을 반전시켜 부정 이미지를 생성하는 Image Negative 기능이며, 셋째는 RGB를 조합하여 16-bit 단일 명암값으로 변환하는 Grayscale 변환 기능이다. 이들 기능은 모두 32-bit RGBA 포맷의 이미지 데이터를 기반으로 동작하며, 실험은 총 9,600개의 픽셀을 대상으로 수행되었다.

기본적인 기능 구현 이후에는 성능 향상을 위한 최적화 작업을 병행하였다. 먼저, Alpha 채널을 제외한 RGB 데이터를 별도로 메모리에 재배치하는 Memory Relocation 기법을 적용하였고, 이를 통해 루프 내 메모리 접근 효율을 높였다. 이후 반복문 구조 개선 및 명령어 수 최소화를 위한 Instruction-level 최적화를 진행하였다. 또한 최적화된 코드가 기존 코드와 동일한 결과를 생성하는지 시뮬레이션과 메모리 맵 비교를 통해 검증하였다.

프로젝트는 단순한 기능 구현을 넘어서 코드의 효율성, 메모리 활용 구조, 연산 성능에 대한 고려를 포함하며, 실험 결과와 성능 비교를 바탕으로 ARM 프로그래밍에서의 최적화 전략을 실제로 적용해 보는 실습 중심의 작업으로 구성되었다. 특히 각 기능별로 기존 코드와 최적화된 코드 간의 실행 시간, 명령어 수 등을 정량적으로 비교함으로써, 하드웨어 자원이 제한된 환경에서 얼마나 효과적으로 연산을 수행할 수 있는지를 직접 검증하였다. 또한 실습 과정에서 단순히 결과가 나오는지 확인하는 것을 넘어, 어떤 방식의 메모리 구조가 연산 병목을 줄이는 데 유리한지, 어떤 명령어 조합이 효율적인지를 분석하며 코드 수준에서의 개선 방향을 도출해냈다. 이러한 과정을 통해 단순한 코드 작성 능력을 넘어서, 시스템 전체의 성능을 고려한 설계와 구현, 그리고 디버깅 및 검증까지 아우르는 실질적인 임베디드 소프트웨어 개발 능력을 체계적으로 학습할 수 있었다.

## 2. 이미지 변환 함수 설명

### 2.1 Function #1: RGBA → Pixel Number

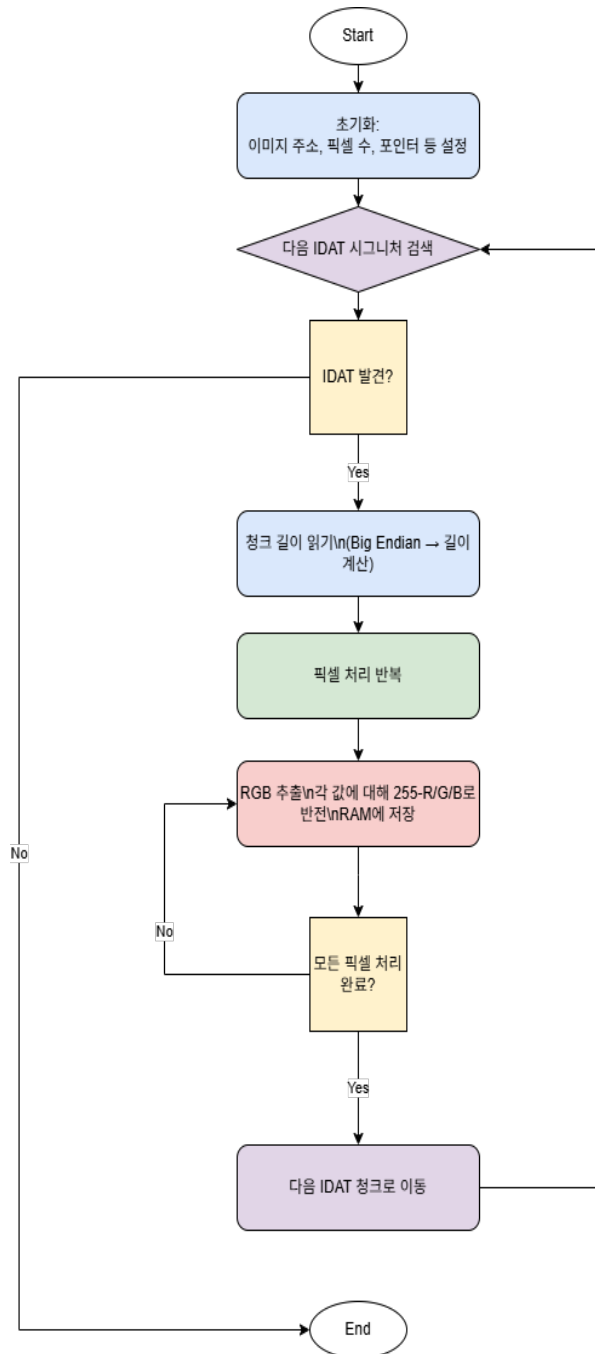


프로그램이 시작되면 필요한 변수와 레지스터를 초기화한다. 그 다음 메모리 영역에서 PNG 이미지의 IDAT 시그니처를 찾기 시작한다. 이를 발견하게 되면, 해당 청크의 데이터 길이를 읽어들이고, 그 길이만큼 RGBA 데이터를 추출하여 지정된 메모리(RAM) 공간에 연속적으로 저장한다. 이 과정을 반복하여 더 많은 IDAT 청크가 존재하면 같은 방식으로 계속해서 RGBA 데이터를 추출하고 저장한다.

더 이상 IDAT 청크를 찾지 못하게 되면, 저장된 RGBA 데이터 중 R 값을 기준으로 조건에 맞는 픽셀의 개수를 세는 단계로 넘어간다. 이때 메모리에 저장된 각 픽셀의 R 값을 하나씩 읽어서, 특정 조건을 만족하는지 확인한다. 모든 픽셀에 대해 이 과정을 반복하며 조건을 만족하는 픽셀의 수를 카운트한다.

모든 픽셀의 확인이 끝나면, 최종적으로 카운트 된 결과 값을 지정된 메모리 위치에 저장하고, 프로그램은 종료 단계로 진입한다.

## 2.2 Function #2: RGBA → 24-bit RGB (Negative)



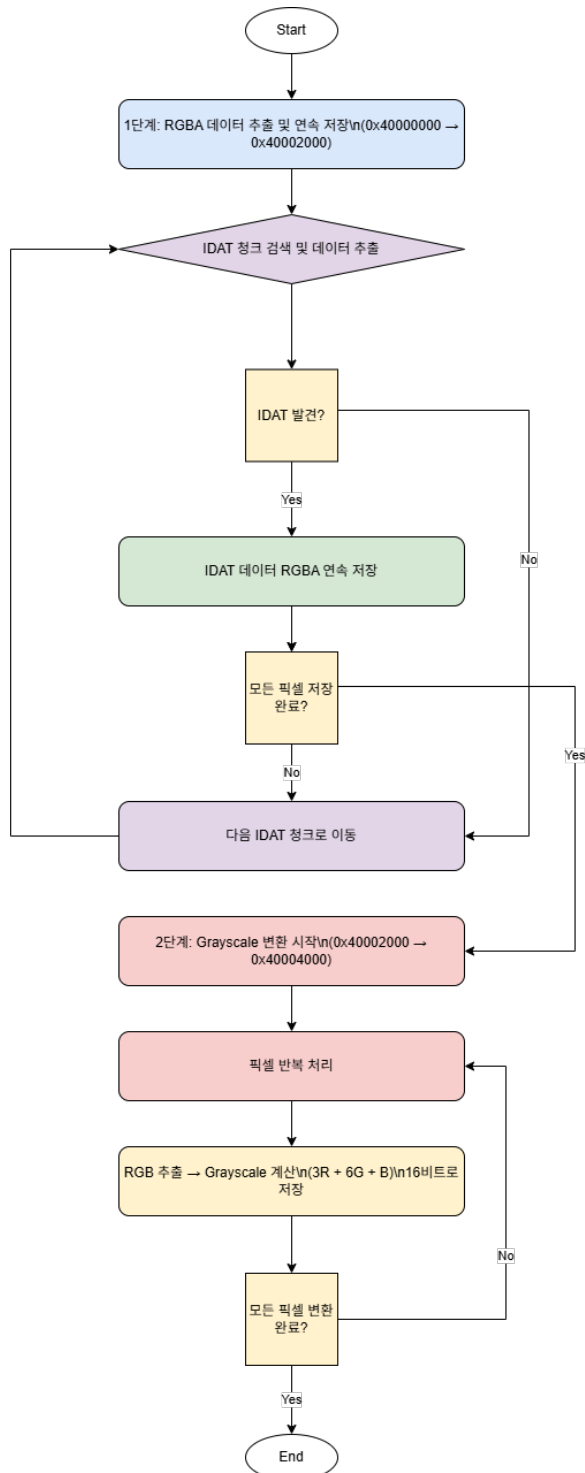
프로그램이 시작되면 먼저 이미지 데이터의 시작 주소, 처리할 픽셀 수, 그리고 데이터 처리를 위한 포인터 등 주요 레지스터들을 초기화한다. 이후 PNG 파일 내에서 IDAT 시그니처를 찾기 위해 메모리 영역을 순차적으로 탐색한다.

IDAT 시그니처를 발견하면, 해당 청크의 데이터 길이를 읽어오는데, 이때 데이터의 저장 방식에 맞춰 Big Endian 형식으로 길이를 계산한다. 이후 실제 픽셀 데이터 처리를 반복하게 된다. 각 픽셀마다 RGBA 데이터 중 RGB 값만을 추출하여 각의 값에 대해 255에서 해당 값을 빼는 방식으로 색상을 반전시킨다. 이렇게 변환된 RGB 값은 지정된 RAM 영역에 저장된다.

모든 픽셀에 대해 처리가 끝났는지 확인하고, 아직 남아 있다면 같은 과정을 반복한다. 만약 한 IDAT 청크의 픽셀 처리가 모두 끝났다면, CRC 4바이트를 건너뛰고 다음 IDAT 청크를 찾아 다시 같은 작업을 진행한다.

이 과정을 통해 모든 IDAT 청크의 픽셀 처리가 완료되면, 프로그램은 종료 단계로 진입한다.

## 2.3 Function #3: RGBA → 16-bit Grayscale



프로그램이 시작되면 먼저 PNG 이미지 데이터에서 RGBA 정보를 추출하여 연속적으로 저장하는 작업을 수행한다. 이 과정에서는 IDAT 청크를 차례로 검색하여, 각 청크에서 RGBA 데이터를 읽어 지정된 메모리 공간에 차례대로 저장한다. 모든 픽셀의 RGBA 데이터가 저장되면 다음 단계로 넘어가게 된다. 만약 아직 모든 픽셀이 저장되지 않았다면, 다음 IDAT 청크를 찾아 같은 과정을 반복한다.

이후 두 번째 단계에서는, 앞서 저장한 RGBA 데이터에서 RGB 값을 추출하여 Grayscale 변환을 진행한다. 변환은 각 픽셀의 R, G, B 값을 이용해 Grayscale 공식( $3R + 6G + B$ )을 적용하는 방식으로 이루어진다. 이렇게 계산된 결과값은 16비트 단위로 새로운 메모리 공간에 저장된다. 모든 픽셀의 변환이 끝나면 프로그램은 종료된다.

### 3. 기본 ARM code 구현

이미지 데이터를 처리하는 세 가지 ARM 코드 함수(① Red 픽셀 개수 카운트, ② 색 반전, ③ Grayscale 변환)를 구현하였다. 이들 세 함수는 모두 hex파일에서 IDAT 청크를 탐색하여 RGBA 픽셀 데이터를 추출하고, 이를 같은 메모리에 저장하는 동일한 전처리 과정을 포함한다. 따라서 공통적인 기능을 수행하는 부분을 먼저 설명한 후, 각 함수에서 구현된 핵심 처리 함수에 대해 개별적으로 서술하고자 한다.<sup>1</sup>

#### 3-1. 전처리

##### ① IDAT 청크 탐색

```
search_next_idat
    CMP    R0, R9                ; 메모리 범위 검사
    BGT    start_counting
    CMP    R1, R6                ; 최대 픽셀 도달 여부 확인
    BGE    start_counting

    ; IDAT Chunk 검색
    LDRB    R2, [R0], #1
    CMP     R2, #'I'
    BNE     search_next_idat
    LDRB    R2, [R0], #1
    CMP     R2, #'D'
    BNE     search_next_idat
    LDRB    R2, [R0], #1
    CMP     R2, #'A'
    BNE     search_next_idat
    LDRB    R2, [R0], #1
    CMP     R2, #'T'
    BNE     search_next_idat
```

<sup>1</sup> 모든 함수에 대한 코드 전문은 해당 보고서 말미 '부록' 목차에 주석과 함께 삽입하였으며, 본문에서는 함수의 핵심 연산 부분에 초점을 맞추어 설명한다. 더하여, 세 함수의 IDAT 청크 탐색 및 메모리 탑재 과정은 구현 코드만 일부 다를 뿐 핵심 로직과 기능은 모두 같기 때문에 이 파트에서는 대표적으로 Function #1의 코드를 설명하였으며, 다른 함수에서 해당 부분을 구현한 코드는 부록 목차에서 주석으로 설명하였다.

```

; IDAT 발견
; Chunk 길이(4 바이트) 읽기
SUB    R10, R0, #8           ; R0 는 'T' 뒤 1 바이트, IDAT 앞 4 바이트는 길이
LDRB   R2, [R10], #1        ; byte 0 (LSB)
LDRB   R3, [R10], #1        ; byte 1
LDRB   R4, [R10], #1        ; byte 2
LDRB   R5, [R10], #1        ; byte 3 (MSB)

; Little-Endian 변환
MOV     R11, R5, LSL #24
ORR     R11, R11, R4, LSL #16
ORR     R11, R11, R3, LSL #8
ORR     R11, R11, R2

; 데이터 시작 주소는 IDAT 시그니처 바로 뒤 (R0)
MOV     R10, R0
ADD     R11, R10, R11        ; 데이터 끝 주소

```

이 단계에서는 hex 파일에서 IDAT 청크의 시작 위치를 탐색한다. 실제 hex 파일에서는 RGBA 픽셀 데이터 IDAT Chunk 바로 뒤부터 위치하며, 파일 내에 여러 개의 IDAT 청크가 중복으로 존재하는 경우도 있기 때문에 이를 처리하는 과정이 반드시 필요하다. 상기 코드는 'R0'부터 'R9'까지의 메모리를 1바이트씩 순차적으로 읽으며 IDAT의 위치를 탐색하고, 이를 발견하면 해당 위치 앞 4바이트에 저장된 값을 읽어 해당 청크의 데이터 길이를 계산한다. 이를 통해 RGBA 데이터가 몇 바이트인지 파악하고, 이후 처리에 사용할 수 있도록 준비한다.

## ② RGBA 저장 및 다음 Chunk 이동

```

store_rgba
    CMP    R10, R11                ; 체크 데이터 끝 확인
    BGE    next_idat
    CMP    R1, R6                  ; 최대 픽셀 확인
    BGE    start_counting

    ; RGBA 4 바이트 처리
    LDRB    R2, [R10], #1          ; R
    LDRB    R3, [R10], #1          ; G
    LDRB    R4, [R10], #1          ; B
    LDRB    R5, [R10], #1          ; A
    ; 32 비트 RGBA 값으로 조합
    ORR     R2, R2, R3, LSL #8
    ORR     R2, R2, R4, LSL #16
    ORR     R2, R2, R5, LSL #24
    ; 정렬된 주소에 저장
    STR     R2, [R8], #4
    ADD     R1, R1, #1
    B       store_rgba

next_idat
    ; 다음 IDAT 검색 (IDAT 데이터 끝 + CRC 4 바이트)
    ADD     R0, R11, #4
    B       search_next_idat

```

해당 구간은 하나의 IDAT 청크에서 RGBA 데이터를 추출하고, 이를 'R8'에 저장한 뒤, 해당 Chunk의 끝에 도달하면 다음 IDAT 청크로 이동하는 과정을 수행한다. RGBA 데이터는 각각 1바이트씩 총 4바이트로 구성되어 있으며 프로그램은 이를 바이트 단위로 읽어 32비트 정수 형태로 조합한 뒤, 'R8' 주소에 순차적으로 저장한다.

또한 저장된 픽셀 수가 최대치(9600개)에 도달하면 저장 루프를 종료하고, 그렇지 않은 경우에는 다음 IDAT Chunk를 탐색하기 위해 CRC 영역(4바이트)을 건너뛰고 다시 탐색 루틴으로 분기하게 된다. 이 과정은 hex 파일 내에 여러 개의 IDAT 청크가 존재하는 경우에도 모든 데이터를 빠짐없이 처리할 수 있도록 설계된 구조이다.

위에서 설명한 ①, ② 과정을 통해 hex 파일 내부에서 RGBA 픽셀만을 뽑아 메모리에 저장하여 메인 기능을 수행할 준비를 마친 상태가 된다.



### 3-2. Function #1

Function#1은 이미지 데이터에서 RGBA 픽셀을 추출하고, Red 값이 128 이상인 픽셀 개수를 카운트하는 프로그램이다. 앞선 IDAT 청크 계산 및 메모리 탑재 과정을 제외한 함수 동작의 핵심 부분을 단계별로 설명한다.

#### (1) 코드 설명

##### ① 초기화

```
main    PROC
        ; 초기화
        LDR    R0, =0x40000000    ; 소스 데이터 시작 주소 (ROM2)
        LDR    R9, =0x4003FFFF    ; ROM2 끝 주소
        LDR    R8, =0x20000000    ; RGBA 저장 시작 주소 (RAM, 4 바이트 정렬)
        LDR    R12, =0x20009600   ; 결과 카운트 저장 주소
        MOV    R1, #0             ; 픽셀 카운터
        LDR    R6, =9600          ; 최대 픽셀 수
```

먼저, 이미지가 저장된 메모리 위치와 이미지 데이터의 크기, 그리고 결과를 저장할 RAM 주소를 설정한다. 그리고 몇 개의 카운터도 초기화해둔다

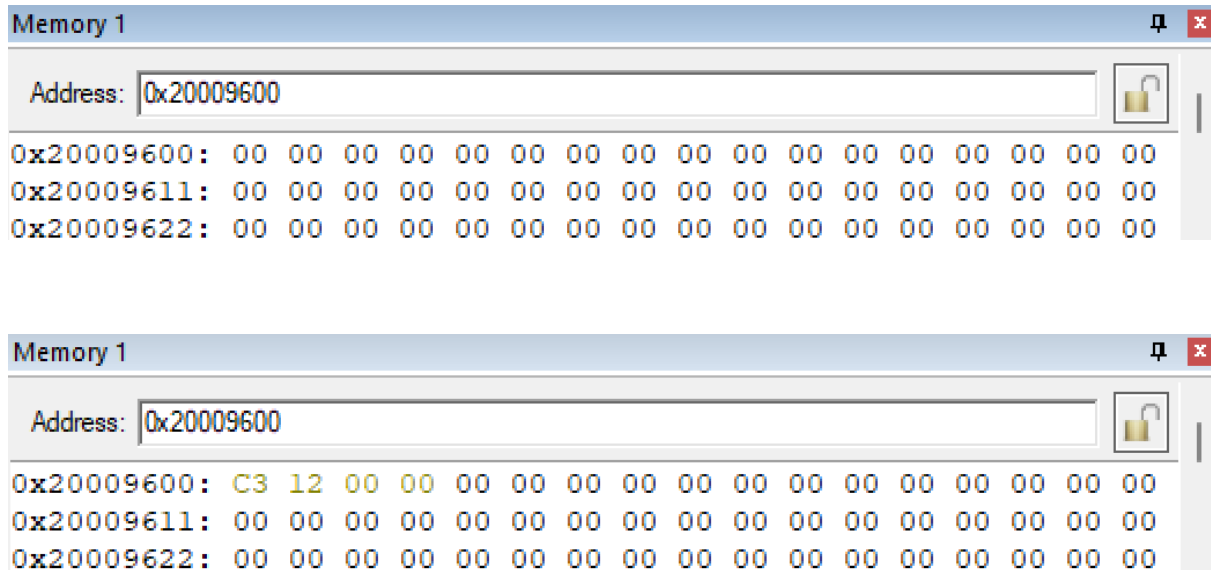
##### ② Red 값 카운팅(start\_counting&count\_loop)

```
start_counting
        LDR    R8, =0x20000000
        MOV    R1, #0
        MOV    R2, #0

count_loop
        CMP    R1, R6
        BGE    final_store
        LDRB   R3, [R8], #4
        CMP    R3, #128
        ADDGE  R2, R2, #1
        ADD    R1, R1, #1
        B      count_loop
```

모든 픽셀 데이터를 저장한 후 R 값이 128 이상인 픽셀의 개수를 센다. 저장된 픽셀 데이터를 4바이트씩 읽으면서, 각 픽셀의 R 값만 추출해서 128 이상인지 확인한다. 조건을 만족하면 카운터를 하나 올리고, 모든 픽셀을 검사할 때까지 계속 반복한다.

## (2) Simulation 결과 및 설명



Simulation 및 메모리 분석을 통해 프로그램이 의도한 대로 동작하는지 확인할 수 있었다. 실제 시뮬레이션 결과, 메모리 주소 0x20000000부터 RGBA 픽셀 데이터가 IDAT를 제외하고 순차적으로 저장된 것이 확인되었다. 각 픽셀은 4바이트 단위로 저장되며, 첫 픽셀의 원본 데이터가 R=0x77, G=0x43, B=0x23, A=0xFF이며 메모리에 정확히 0x77, 0x43, 0x23, 0xFF가 저장되어 있다. 이는 프로그램이 hex 파일 내에서 IDAT 청크를 정확히 찾아내고, 해당 청크 내의 픽셀 데이터를 올바르게 추출하여 저장하고 있음을 의미한다. 이후 저장된 픽셀 데이터를 기준으로, 각 픽셀의 Red 값이 128 이상인지 확인하는 과정이 진행되었고 최종적으로 Red 값이 128 이상인 픽셀의 개수가 메모리 주소 0x20009600에 저장된다. 시뮬레이션 결과, 예상대로 Red 값이 128 이상인 픽셀의 개수가 누적되어 저장된 것이 확인되었다. 결과 값은 00 00 12 C3개, 10진수로 4803개이다.

### 3-3. Function #2

Function #2는 PNG 이미지 내 IDAT 청크를 탐색한 뒤, RGBA 데이터를 추출하여 RGB 채널만 반전(RSB 연산)한 후 다시 RGBA 형식으로 연속 저장하는 작업을 수행한다. 이 과정은 SeparateRGBA\_Pack 함수에서 실행되며, 이후 Grayscale 처리와는 별도로 RGBA의 RGB 채널 반전 테스트 용도로 활용된다.

#### (1) 코드 설명

##### ① 초기화

```
SeparateRGBA_Pack
    LDR    R0, =0x40000000    ; PNG 이미지 시작 주소
    LDR    R1, =0x4003FFFF    ; PNG 이미지 끝 주소
    LDR    R2, =9600          ; 처리할 픽셀 수
    MOV    R3, #0             ; 픽셀 카운터

    LDR    R4, =0x20002000    ; RGBA 연속 저장 주소
```

먼저 R0에는 PNG 이미지의 시작 주소가, R1에는 탐색 종료 주소가 로드된다. R2에는 총 처리할 픽셀 수인 9600이 저장되며, R3는 루프 카운터로 초기화된다. 이후 R4에는 반전된 RGBA 데이터를 순차적으로 저장할 메모리 공간 주소(0x20002000)가 저장된다.

## ② 픽셀 처리 (RGB 반전)

```

ProcessLoop
    CMP    R3, R2
    BGE    EndProgram

    LDRB    R5, [R0]           ; R
    LDRB    R6, [R0, #1]      ; G
    LDRB    R7, [R0, #2]      ; B
    LDRB    R8, [R0, #3]      ; A

    RSB     R5, R5, #255       ; R 반전
    RSB     R6, R6, #255       ; G 반전
    RSB     R7, R7, #255       ; B 반전

    STRB     R5, [R4], #1      ; R 저장
    STRB     R6, [R4], #1      ; G 저장
    STRB     R7, [R4], #1      ; B 저장
    STRB     R8, [R4], #1      ; A 저장

    ADD     R0, R0, #4
    ADD     R3, R3, #1
    B       ProcessLoop

```

픽셀 처리 루프에서는 R0가 가리키는 주소에서 4바이트(R, G, B, A)를 순차적으로 읽는다. 각각의 R, G, B 값은  $255 - \text{값}$ 의 방식으로 반전되며, A 값은 그대로 유지된다. 반전된 RGB와 원본 A 값은 R4가 가리키는 메모리에 RGBA 순서로 연속 저장된다. 매 반복마다 R0는 4바이트, R4는 4바이트씩 증가하며, R3는 현재까지 처리한 픽셀 수를 기록하기 위해 증가한다.

### ③ IDAT 탐색 실패 시 처리 및 종료 루틴

```
Skip
    ADD    R0, R0, #1
    B      FindIDAT

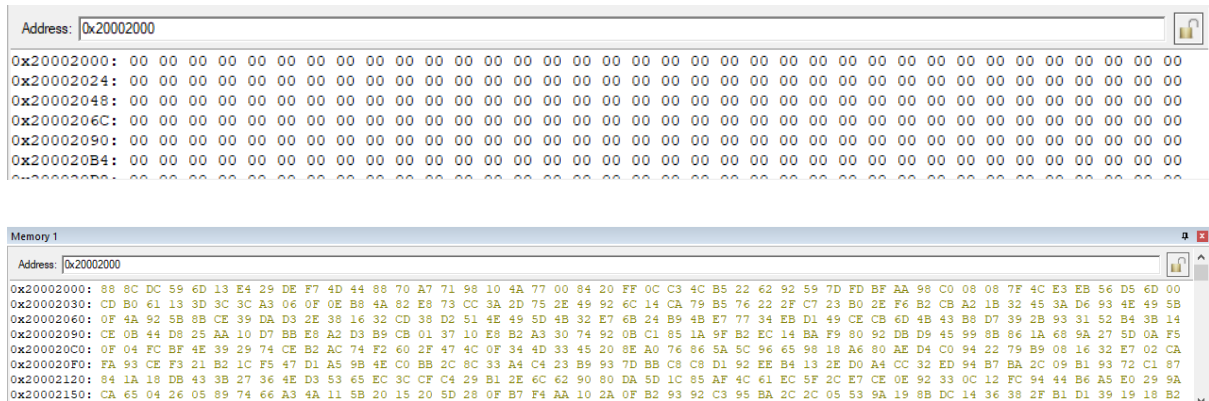
EndProgram
    B      EndProgram

END
```

IDAT 청크 탐색 과정에서 현재 주소(R0)에 있는 데이터가 'IDAT' 시그니처와 일치하지 않는 경우, Skip 레이블로 분기하여 R0 값을 1 증가시킨 뒤 다시 탐색 루틴(FindIDAT)으로 되돌아간다. 이 방식은 이미지 메모리 전체를 한 바이트 단위로 순차 탐색하여 IDAT 위치를 정확히 찾아내기 위한 예외 처리 구조이다.

탐색이 끝나거나 픽셀 데이터 처리가 완료된 경우에는 EndProgram 레이블로 분기된다. 이때는 프로그램 흐름이 B EndProgram 명령에 의해 무한 루프에 진입하며 함수 실행이 종료된다. 이는 후속 연산 없이 시뮬레이터 또는 테스트 환경에서 실행 완료를 명시적으로 나타내기 위한 종료 처리 방식이다.

## (2) Simulation 결과 및 설명



최종 반전된 RGBA 값은 메모리 주소 0x20002000부터 순차적으로 저장되도록 설계되었기 때문에, 시뮬레이션 결과에서 해당 주소의 내용을 확인함으로써 코드가 의도대로 작동했는지를 검증할 수 있다. hex 파일 상의 첫 번째 픽셀은 77 73 23이며, 이를 10진수로 변환하면 R=119, G=115, B=35가 된다. 이에 대한 반전 결과는 각각 R=136(0x88), G=140(0x8C), B=220(0xDC)이고, A는 그대로 255(0xFF)이므로 최종 저장값은 88 8C DC FF가 되어야 한다. 실제로 시뮬레이션 결과 메모리 주소 0x20002000에는 88 8C DC FF가 저장되어 있으며, 이는 해당 픽셀의 RGB 값에 대해 반전 연산이 정확히 수행되었음을 의미한다. 따라서 실제 메모리 덤프에 저장된 값 88 8C DC FF는 해당 픽셀의 반전 연산 결과와 정확히 일치하며, 코드가 정상적으로 동작했음을 확인할 수 있다.

### 3-4. Function #3

Function #3은 앞서 설명한 IDAT 탐색 및 RGBA 저장 과정을 수행하는 `extract\_rgba\_data` 함수 호출로 시작되며, 이후 핵심 처리 부분인 Grayscale 변환이 `calculate\_grayscale\_rgba`에서 수행된다. 따라서 이번 절에서는 Grayscale 계산 로직에 대해 집중적으로 설명한다.

#### (1) 코드 설명

##### ① 초기화

```
extract_rgba_data PROC
    LDR    R0, =0x40000000    ; 이미지 시작 주소
    LDR    R8, =0x40002000    ; RGBA 저장 주소
    LDR    R9, =0x4000FFFF    ; 이미지 끝 주소 한계
    MOV    R1, #0             ; 픽셀 카운터
    LDR    R6, =9600           ; 최대 9,600 픽셀
```

```
calculate_grayscale_rgba PROC
    LDR    R0, =0x40002000    ; RGBA 데이터 시작 주소
    LDR    R8, =0x40004000    ; Grayscale 결과 저장
    LDR    R6, =9600           ; 최대 9,600 픽셀
    MOV    R1, #0
```

각각의 함수 내부에서 연산에 필요한 주소와 루프 변수들을 독립적으로 초기화 한다. 먼저 호출되는 `extract\_rgba\_data` 함수에서는 RGBA 데이터를 `R8` 주소에 저장하며, 이후 `calculate\_grayscale\_rgba` 함수가 이를 순차적으로 읽어 Grayscale 계산을 수행한다.

## ② Grayscale 변환

```

process_rgba_pixels
    CMP     R1, R6
    BGE     calculate_done_rgba

    LDRB     R3, [R0], #1      ; R
    LDRB     R4, [R0], #1      ; G
    LDRB     R5, [R0], #1      ; B
    ADD     R0, R0, #1        ; A 는 그냥 스킵

    ; Grayscale 공식: 3*R + 6*G + B
    MOV     R7, R3
    ADD     R7, R7, R7, LSL #1 ; 3*R
    MOV     R12, R4
    ADD     R12, R12, R12, LSL #1 ; 3*G
    ADD     R12, R12, R12      ; 6*G
    ADD     R7, R7, R12
    ADD     R7, R7, R5

    STRH     R7, [R0], #2      ; 16 비트로 저장

    ADD     R1, R1, #1
    B       process_rgba_pixels

```

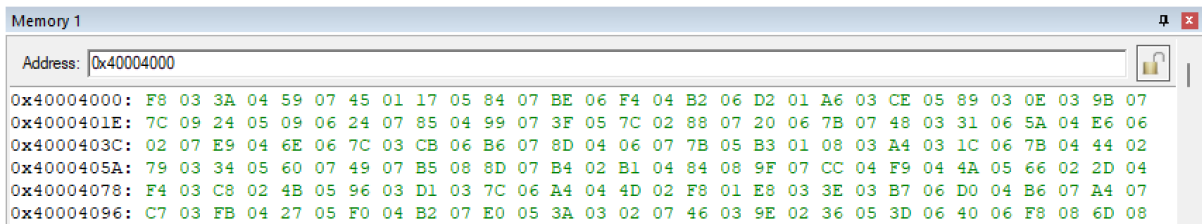
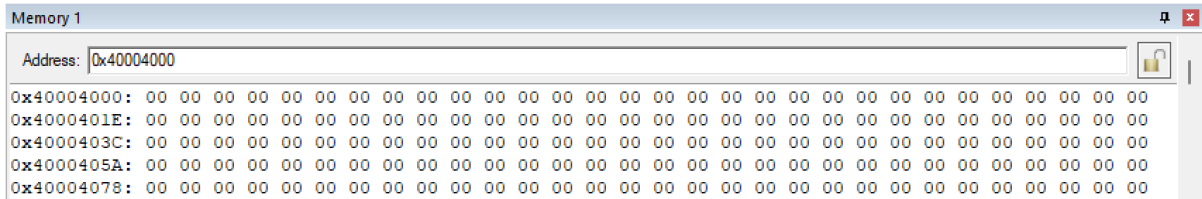
위 함수는 메모리에 저장된 RGBA 데이터를 순차적으로 읽어 각 픽셀의 Grayscale 값을 계산하고 그 결과를 16비트 형태로 저장한다. 루프를 통해 총 9600개의 픽셀을 처리하며, 각 반복에서 R, G, B 값을 차례로 읽고, A 값은 연산에 사용하지 않기 때문에 건너뛰는다.

Grayscale 변환은  $3 \times R + 6 \times G + B$  공식을 기반으로 수행되며, 코드에서는 곱셈 대신 시프트와 덧셈을 활용해 구현하였다.  $3 \times R$ 은  $R + (R \ll 1)$ 로 계산되고,  $6 \times G$ 는 먼저  $3 \times G = G + (G \ll 1)$ 을 만든 후 이를 다시 한 번 더하여  $6 \times G = 3 \times G + 3 \times G$ 로 계산된다.

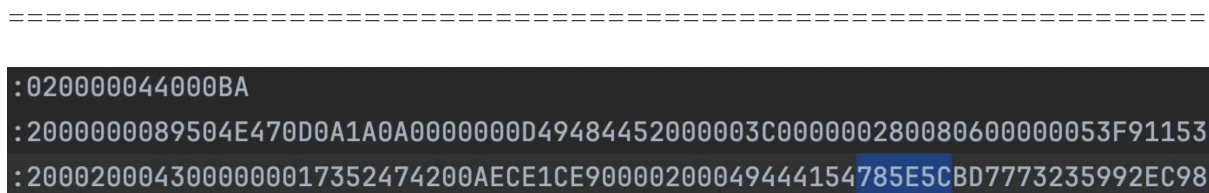
최종 계산된 Grayscale 값은 16비트(Half-word) 단위로 저장되며, 이후 다음 픽셀로 반복이 진행된다. 이 과정을 통해 RGBA 이미지가 Grayscale 이미지로 변환된다.



## (2) Simulation 결과 및 설명



최종 Grayscale 연산 결과는 '0x40004000'에 저장하도록 했기 때문에 hex 파일의 첫 번째 IDAT 이후 R, G, B 바이트와 실제 결과를 비교하여 실행 결과가 정확한지 확인해야 한다.



hex 파일에서 첫 번째 RGB 픽셀은 '78 5E 5C' 이고, 이 16진수를 10진수로 변환하면 R = 0x78 (120), G = 0x5E (94), B = 0x5C (92)가 된다.

Grayscale 변환식  $3 \times R + 6 \times G + B$ 에 위 10진수들을 대입하면 '1016'이라는 결과가 나오고 이는 10진수이기 때문에 메모리 주소 값으로 확인하려면 다시 16진수로 변환하는 과정이 필요하다. 결과적으로 16진수로 변환한 값 '0x03F8'를 상기 결과 메모리 저장 주소에서 확인하면 정확히 일치하는 것을 볼 수 있다.



## 4. Memory Relocation 기반 ARM code 최적화

이미지 데이터는 기본적으로 RGBA 형식으로 저장되어 있으나, 처리 과정에서 A는 활용되지 않는다. 그렇기 때문에 RGBA에서 A만 제외하고 RGB 값을 연속적으로 저장하는 방식과 R, G, B 값을 각각의 독립된 메모리 블록에 따로 저장하여 연산 접근을 단순화하는 구조를 세 함수에 적용하여 코드를 재구현 했으며, 아래에서는 이 두 가지 Memory Relocation 방식의 구조와 코드 구현에 대해 설명하고, 이후 각 함수 Function 1, 2, 3에 이를 적용했을 때의 특이점과 결과를 분석하고자 한다. <sup>2</sup>

### 4-1. Memory Relocation

#### (1) Relocation 1 - Alpha 제외하고 저장

```
extract_rgb_pixels
    CMP    R1, R6                ; 9600 개 초과 여부
    BGE    extract_done

    CMP    R10, R11              ; IDAT 청크 끝
    BGE    next_idat_search_step1

    ; RGBA 에서 RGB 만 추출하여 연속 저장
    LDRB   R3, [R10, #0]         ; Red
    LDRB   R4, [R10, #1]         ; Green
    LDRB   R5, [R10, #2]         ; Blue
    ; Alpha [R10, #3]는 무시

    STRB   R3, [R8], #1          ; R 저장
    STRB   R4, [R8], #1          ; G 저장
    STRB   R5, [R8], #1          ; B 저장

    ADD    R10, R10, #4          ; 다음 RGBA 픽셀
    ADD    R1, R1, #1            ; 픽셀 카운터 증가
    B      extract_rgb_pixels
```

상기 코드는 RGBA 데이터로부터 각 픽셀의 R, G, B 요소만을 추출하여 연속으로 저장하는

<sup>2</sup> 앞선 목차와 마찬가지로 세 함수의 R, G, B 메모리 탑재 과정은 구현 코드만 일부 다를 뿐 핵심 로직과 기능은 모두 같기 때문에 이 파트에서는 대표적으로 Function #3의 코드를 설명하였으며, 다른 함수에서 해당 부분을 구현한 코드는 부록 목차에서 주석으로 설명하였다.

과정을 구현한 것이다. 각 픽셀은 원래 4바이트로 구성되어 있으나, A값은 연산에 사용되지 않기 때문에 'LDRB' 명령어로 R, G, B만 읽고, 각각 'STRB' 명령어로 메모리에 저장한다.

(2) Relocation 2 - Alpha 제외, R, G, B 각각 따로 저장

```
extract_separate_pixels
    CMP     R1, R6
    BGE     extract_separate_done
    CMP     R10, R11
    BGE     next_idat_search_step1

    ; RGB 각각 분리해서 저장
    LDRB     R5, [R10, #0]      ; Red
    STRB     R5, [R2, R1]      ; R 저장

    LDRB     R7, [R10, #1]      ; Green
    STRB     R7, [R3, R1]      ; G 저장

    LDRB     R12, [R10, #2]     ; Blue
    STRB     R12, [R4, R1]     ; B 저장

    ADD     R10, R10, #4        ; 다음 RGBA 픽셀
    ADD     R1, R1, #1
    B       extract_separate_pixels
```

상기 코드는 RGBA 형식의 데이터를 순회하며, 각 픽셀의 R, G, B 값을 각각 독립된 메모리 배열에 분리 저장한다. Red, Green, Blue는 각각 R2, R3, R4로 시작하는 배열에 인덱스 기반으로 저장되며, Alpha 값은 사용되지 않기 때문에 건너뀐다. 이 방식은 이후 연산에서 각 색상 채널에 독립적으로 접근이 가능하도록 하여, 프로젝트 명세에서 요구하는 메인 연산을 효율적으로 수행할 수 있도록 한다.

## 4-2. Function #1

## (1) Relocation 1 - Alpha 제외하고 저장

## ① 코드 설명

```

main    PROC
        ; 초기화
        LDR    R0, =0x40000000
        LDR    R9, =0x4003FFFF
        LDR    R8, =0x20000000
        LDR    R12, =0x20007080
        MOV    R1, #0
        LDR    R6, =9600

start_counting
        ; 저장된 데이터에서 Red 값 카운팅 (예시: R 값만 읽어서 카운트)
        LDR    R8, =0x20000000    ; RGB 저장 시작 주소
        MOV    R1, #0              ; 루프 카운터
        MOV    R2, #0              ; Red 카운터

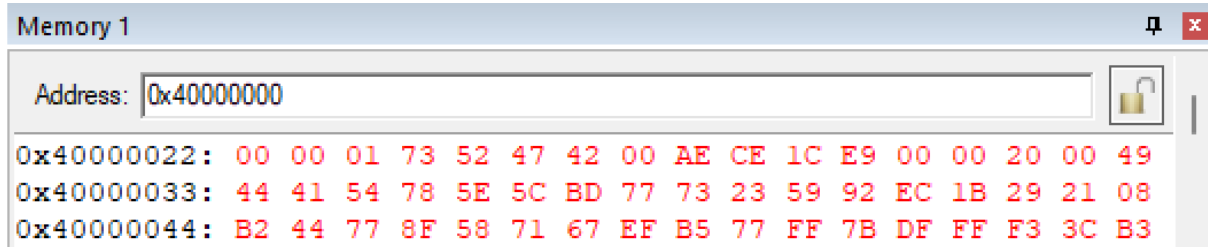
count_loop
        CMP    R1, R6              ; 9600 픽셀 확인
        BGE    final_store
        ; 3 바이트 단위로 R 값만 읽기
        LDRB   R3, [R8], #3        ; R 값만 읽음 (3 바이트 단위, 정렬 보장)
        CMP    R3, #128
        ADDGE  R2, R2, #1          ; 카운트 증가
        ADD    R1, R1, #1
        B      count_loop

```

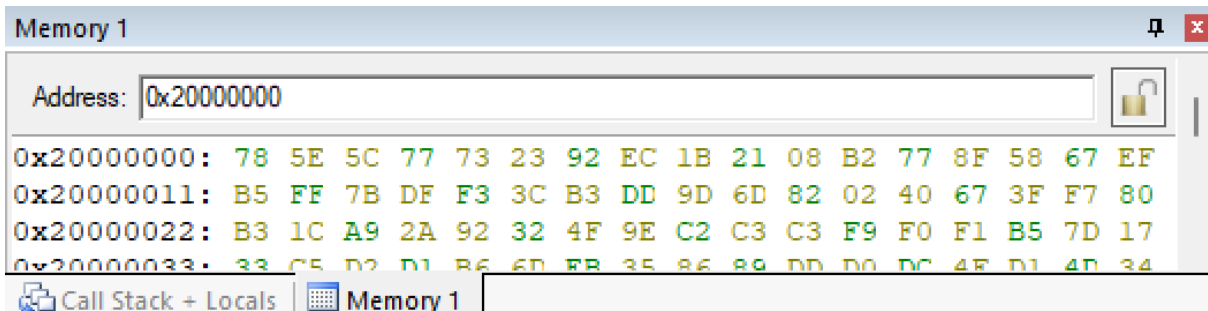
프로그램 실행 전 이미지 데이터 시작 주소를 0x40000000로 정하고 결과는 0x20007080 주소에 넣었다. 각 픽셀 RGBA값에서 A값은 무시하고 RGB값만 연속해서 RAM에 저장한다. 픽셀을 하나씩 처리하면서 청크 데이터가 끝날 때 까지 반복한다.

## ② Simulation 결과 및 설명

실행 전 데이터 상태



실행 후 데이터 상태 (IDAT를 제외하고 RGB만 저장된 모습)

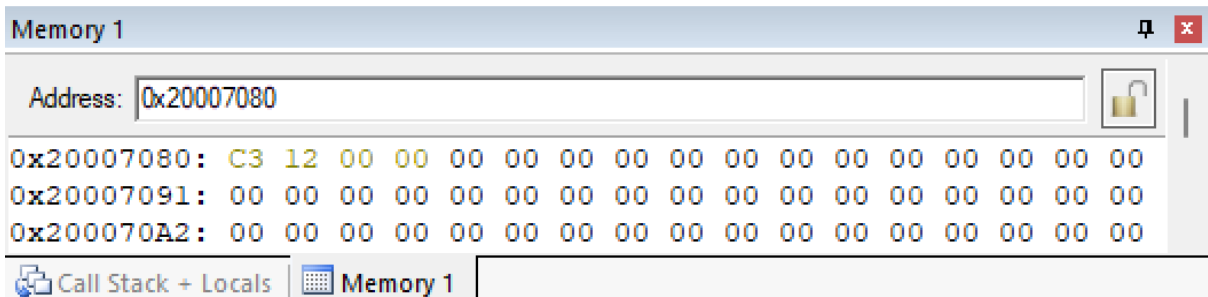


실행 후 결과값

Performance Analyzer

Reset Show: Modules

Module/Function	Calls	Time(Sec)	Time(%)
rgbrgb		35,581 ms	100%
rgbrgb.s		35,581 ms	100%
main	1	35,581 ms	100%



실행 결과를 살펴보면, RGBA 데이터를 4바이트 단위로 저장하고 계산했을 때와 달리, RGB만 저장하는 방식으로 바꾸자 원래는 34.621ms 걸리던 작업이, 35.581ms가 걸렸다. 이런 현상이 나타난 이유는 크게 두 가지로 볼 수 있다. 결과 값은 기본 arm code와 일치한다.

첫 번째 원인은 메모리 접근 횟수에 있다. ARM7 같은 아키텍처는 32비트, 즉 4바이트 정렬된 메모리 접근에 최적화되어 있다. 그래서 STR이나 LDR 명령어로 4바이트씩 한 번에 읽고 쓰는 것이 가장 빠르다. 하지만 코드에서 RGB만 저장할 때는 STRB 명령어로 1바이트씩, 그리고 3바이트 단위로 인터리브된 데이터를 다루게 되는데 이렇게 되면 내부적으로 버스 접근이 비효율적으로 일어나고, 추가적인 연산이나 정렬 오류가 발생할 수 있다. 결국 RGBA 저장 방식에서는 픽셀당 메모리 쓰기를 한 번만 하면 되지만, RGB 저장 방식에서는 픽셀당 메모리 쓰기를 3번 해야 해서, 메모리 접근 횟수가 늘어나게 된다.

두 번째 원인은 데이터 조합과 관련된 오버헤드이다. 3바이트 단위로 접근하면 내부적으로 4바이트씩 읽고 원하는 바이트만 추출하는 추가 연산이 요구된다. 또, 캐시 라인이나 버스 접근도 4바이트 단위로 동작하는데, 3바이트 단위로 계속 접근하면 오히려 오버헤드가 발생할 수 있다. RGBA 저장 방식에서는 4바이트를 한 번에 조합해서 저장하지만, RGB 저장 방식에서는 3바이트를 개별적으로 처리하고 저장해야 하니, 추가적인 연산이 들어간다. 거기에 패딩 바이트를 처리해야 하는 경우도 생길 수 있다.

이처럼, ARM 아키텍처에서는 4바이트 정렬된 메모리 접근이 훨씬 빠르고 효율적이다. 그래서 RGBA처럼 4바이트 단위로 데이터를 다루는 것이 RGB, 3바이트 단위로 다루는 것보다 실행 시간이 더 짧게 나온 것을 볼 수 있다.

## (2) Relocation 2 – Alpha 제외, R, G, B 각각 따로 저장

## ① 코드 설명

```

main PROC
; 초기화 - 레지스터 사용 최적화
LDR    R0, =0x40000000    ; 소스 데이터 시작 주소 (ROM2)
LDR    R8, =0x4003FFFF    ; ROM2 끝 주소 (256KB)
LDR    R1, =0x20000000    ; R 배열 시작 주소 (9600 바이트)
LDR    R2, =0x20002580    ; G 배열 시작 주소 (R 배열 + 9600)
LDR    R3, =0x20004B00    ; B 배열 시작 주소 (G 배열 + 9600)
LDR    R12, =0x20007080   ; 결과 저장 주소 (B 배열 + 9600)
MOV     R4, #9600          ; 최대 픽셀 수
MOV     R9, #0             ; Red 카운터 (>= 128)
MOV     R13, #0            ; 픽셀 카운터

```

```

process_idat_pixels_loop
CMP     R10, R11           ; 청크 데이터 끝 확인
BGE     next_idat_search
CMP     R13, R4            ; 최대 픽셀 확인
BGE     store_result

; RGBA 4 바이트 읽기
LDRB    R5, [R10], #1      ; R
LDRB    R6, [R10], #1      ; G
LDRB    R7, [R10], #1      ; B
ADD     R10, R10, #1       ; A 스킵

; R, G, B 저장
STRB    R5, [R1], #1       ; R 배열
STRB    R6, [R2], #1       ; G 배열
STRB    R7, [R3], #1       ; B 배열

; Red 값 검증
CMP     R5, #128
ADDGE   R9, R9, #1         ; 카운트 증가

ADD     R13, R13, #1       ; 픽셀 카운터 증가
B       process_idat_pixels_loop

```

먼저 hex 파일에 들어있는 RGB 값을 각각 추출하기 위해 R, G, B 값을 저장할 RAM 주소를 준비한다. 결과 값은 0x20007080에 저장된다. 픽셀 카운터와 Red 값 카운터도 0으로 초기화한다. IDAT 청크 계산 후에 RGBA 값을 읽고 R, G, B만 각각의 배열에 저장한다. 픽셀을 처리할 때마다 카운터가 증가하며, 최대 픽셀 수에 도달하면 멈춘다.



## ② Simulation 결과 및 설명


실행 전 데이터 상태

The image shows a screenshot of a debugger's 'Memory 1' window. At the top, there's a title bar 'Memory 1' with a search icon and a close button. Below the title bar, there's a text field labeled 'Address:' containing the value '0x40000000'. To the right of the text field is a lock icon. Below the text field, there's a memory dump showing three lines of memory data. Each line starts with an address followed by a colon and then 16 hexadecimal values. The addresses are 0x40000022, 0x40000033, and 0x40000044. The hexadecimal values are displayed in red text. The first line is 0x40000022: 00 00 01 73 52 47 42 00 AE CE 1C E9 00 00 20 00 49. The second line is 0x40000033: 44 41 54 78 5E 5C BD 77 73 23 59 92 EC 1B 29 21 08. The third line is 0x40000044: B2 44 77 8F 58 71 67 EF B5 77 FF 7B DF FF F3 3C B3.

실행 후 데이터 상태(상단부터 차례로 R, G, B 따로 저장한 것)

Memory 1

Address: 0x20000000



0x20000000: 78 77 92 21 77 67 FF F3 DD 82 67 80 A9 32 C2 F9 B5

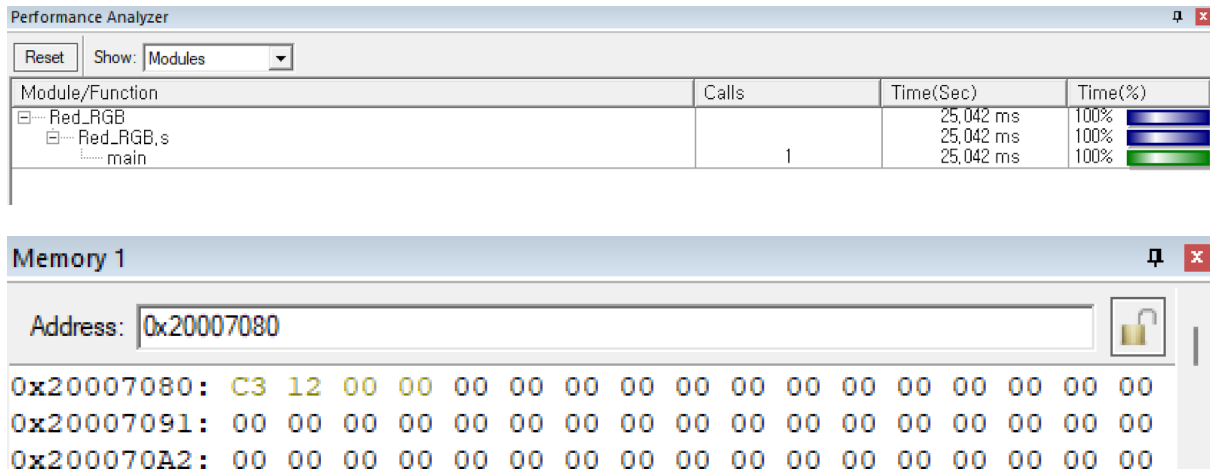
0x20000011: 33 D1 EB 89 DC 4D CD 6C F0 74 2C CD AE B4 DB 88 B6

0x20000022: B4 C6 AD 31 DA 44 46 EF CF 3E 4D 06 26 79 D8 F0 B1

The image shows a screenshot of a debugger's 'Memory' window. The title bar reads 'Memory 1'. Below the title bar, there is a text field labeled 'Address:' containing the value '0x20002580'. To the right of this field is a lock icon. The main area of the window displays a memory dump with three lines of data. Each line shows an address followed by a colon and then 16 hexadecimal bytes. The first line is '0x20002580: 5E 73 EC 08 8F EF 7B 3C 9D 02 3F B3 2A 4F C3 F0 7D'. The second line is '0x20002591: C5 B6 35 DD 4F 34 BA B1 B5 31 D1 32 B1 CD 46 CB 31'. The third line is '0x200025A2: BC D4 4B F4 55 17 34 17 8B 7A 13 7F BA E5 A2 FB C6'. At the bottom of the window, there is a tab bar with two tabs: 'Call Stack + Locals' and 'Memory 1', with 'Memory 1' being the active tab.

Memory 1																	
Address:		0x20004B00															
0x20004B00:		5C	23	1B	B2	58	B5	DF	B3	6D	40	F7	1C	92	9E	C3	F1 17
0x20004B11:		D2	6D	86	D0	D1	5D	C5	B6	6D	C6	C7	C7	B6	18	B4	14 34
0x20004B22:		47	6C	C4	BB	EF	5D	FE	4D	6D	E5	EB	6D	66	97	F5	03 D6

위와 같이 IDAT를 제외하고 RGBA 순서대로 78, 5E, 5C, BD ...이 저장되어 있는데 R 배열에는 0x78, G 배열에는 0x5E, B 배열에는 0x5C가 저장된 것을 확인할 수 있다.



시뮬레이션 결과, 저장된 결과 값이 기존의 결과 값과 일치함을 확인할 수 있었다. 기존 코드에서는 R 값만 추출 시 4바이트 단위 접근 후 R 값 분리가 요구 되었다.(LDRB + shift) 따라서 R, G, B를 각각의 배열에 저장하니 불필요한 연산이 줄어들었고 메모리 접근도 더 빨라졌다. 또한 기존에는 4바이트 단위로 접근했지만 1바이트 단위로 각 채널에 접근하면서도 배열이 연속적으로 배치되어 있어 효율이 더 좋아졌다. 결과 값은 기본 arm code와 일치한다.

### 4-3. Function #2

#### (1) Relocation 1 - Alpha 제외하고 저장

##### ① 코드 설명

```

ProcessLoop2
    CMP    R3, R2
    BGE    EndProgram

    LDRB    R7, [R0]           ; R
    LDRB    R8, [R0, #1]      ; G
    LDRB    R9, [R0, #2]      ; B

    RSB     R7, R7, #255       ; 반전: 255 - R
    RSB     R8, R8, #255       ; 반전: 255 - G
    RSB     R9, R9, #255       ; 반전: 255 - B

    STRB     R7, [R10], #1     ; R 저장
    STRB     R8, [R10], #1     ; G 저장
    STRB     R9, [R10], #1     ; B 저장

    ADD     R0, R0, #4         ; 다음 픽셀 (4 바이트)
    ADD     R3, R3, #1         ; 픽셀 카운터 증가
    B       ProcessLoop2
  
```

해당 코드는 RGBA 포맷의 픽셀 데이터 중에서 R, G, B만 읽고, 각 값을 반전(255에서 뺀)하여 메모리에 RGBRGB... 형태로 연속 저장하는 역할을 한다.

특히 이 연산에서는 A(Alpha) 채널을 명시적으로 건너뛰기 위해 ADD R0, R0, #4 명령을 사용하여 픽셀 단위(4바이트)를 한 번에 넘긴다. 따라서 별도의 A 스킵 로직 없이도 반복 루프마다 자동으로 A를 제외한 데이터에 접근할 수 있다.

즉, 이 방식은 연산 효율성과 메모리 정렬을 동시에 고려한 구조로, 이후 Grayscale 변환이나 이미지 출력 등의 후속 처리에 최적화되어 있다.



## (2) Relocation 2 – Alpha 제외, R, G, B 각각 따로 저장

## ① 코드 설명

```
ProcessLoop
    CMP    R3, R2
    BGE    EndProgram

    LDRB   R7, [R0]           ; R
    LDRB   R8, [R0, #1]      ; G
    LDRB   R9, [R0, #2]      ; B

    RSB    R7, R7, #255
    RSB    R8, R8, #255
    RSB    R9, R9, #255

    STRB   R7, [R4], #1
    STRB   R8, [R5], #1
    STRB   R9, [R6], #1

    ADD    R0, R0, #4
    ADD    R3, R3, #1
    B      ProcessLoop
```

이 코드는 RGBA 이미지 데이터에서 R, G, B 채널만 추출하여 각각 0x20000000, 0x20000800, 0x20001000에 분리 저장한다. 각 채널 값은 255 - 값 방식으로 반전 처리되며, Alpha 채널은 연산 및 저장에서 완전히 제외된다. 이러한 구조는 RGB 각 채널을 독립적으로 분석하거나 처리할 수 있도록 해주며, 예를 들어 특정 색상 성분만 따로 활용하거나 비교할 때 유용하다. 또한 루프마다 ADD R0, R0, #4를 통해 픽셀 단위로 이동하면서 A를 자동으로 건너뛰기 때문에 코드가 간단하고 메모리 접근도 효율적이다.



먼저, 메모리에서 추출된 첫 번째 픽셀의 R, G, B 값은 각각 다음과 같다. 0x20000000에 저장된 R 값은 0x88(136), 0x20000800의 G 값은 0xFF(255), 0x20001000의 B 값은 0x9E(158)이다. 이 값들은 반전된 결과이므로, 원래의 RGB는  $R = 255 - 136 = 119$ ,  $G = 255 - 255 = 0$ ,  $B = 255 - 158 = 97$ 이다. 즉, 원본 픽셀은 77 00 61이며, 이는 PNG 이미지의 IDAT 청크 이후 실제 픽셀 데이터와 일치해야 한다. 이후 메모리 값들을 보면, 각 채널이 1바이트 단위로 독립된 메모리 블록에 연속 저장되어 있으며, A 채널은 완전히 제외되었다. 또한, 모든 RGB 값은 정확하게 반전된 형태로 저장되고 있다.

결과적으로, SeparateRGB 함수는 R, G, B를 각각 반전 처리한 뒤 세 영역에 정확히 나눠 저장하고 있으며, 코드가 설계대로 올바르게 작동했음을 메모리 덤프를 통해 확인할 수 있다.

,

#### 4-4. Function #3

##### (1) Relocation 1 - Alpha 제외하고 저장

###### ① 코드 설명

```

calculate_grayscale PROC
    LDR    R0, =0x40002000    ; RGB 데이터 시작 주소
    LDR    R8, =0x40004000    ; Grayscale 결과 저장 주소
    LDR    R6, =9600          ; 최대 9,600 픽셀
    MOV    R1, #0             ; 픽셀 카운터

process_rgb_pixels
    CMP    R1, R6             ; 9600 개 처리했는지 확인
    BGE    calculate_done

    LDRB   R3, [R0], #1        ; Red
    LDRB   R4, [R0], #1        ; Green
    LDRB   R5, [R0], #1        ; Blue

    ; Grayscale 공식: 3*R + 6*G + B
    MOV    R7, R3
    ADD    R7, R7, R7, LSL #1  ; R7 = R + 2*R = 3*R
    MOV    R12, R4
    ADD    R12, R12, R12, LSL #1 ; R12 = G + 2*G = 3*G
    ADD    R12, R12, R12        ; 3*G + 3*G = 6*G
    ADD    R7, R7, R12
    ADD    R7, R7, R5

    STRH   R7, [R8], #2        ; 16 비트로 저장하고 주소 증가

    ADD    R1, R1, #1
    B      process_rgb_pixels

```

RGBA 데이터를 기반으로 연산하는 코드(목차 3-4)는 메모리 상에 RGBA를 모두 저장해놓고 A는 계산에 필요하지 않기 때문에 'ADD R0, R0, #1' 명령어로 스킵한 반면에 A를 제외하고 RGB만 저장한 메모리를 가지고 연산을 수행하면 처음부터 A가 제거된 3바이트 데이터이므로 별도의 스킵 명령어 없이 연속적으로 R, G, B를 읽을 수 있다. 이로 인해 후자의 방식이 메모리 접근 측면에서 더 효율적이다.



## ② 실행 결과

Memory 1	
Address: 0x40002000	
0x40002000:	00 00
0x4000201E:	00 00
0x4000203C:	00 00
0x4000205A:	00 00

Memory 1	
Address: 0x40002000	
0x40002000:	78 5E 5C 77 73 23 92 EC 1B 21 08 B2 77 8F 58 67 EF B5 FF 7B DF F3 3C B3 DD 9D 6D 82 02 40
0x4000201E:	67 3F F7 80 B3 1C A9 2A 92 32 4F 9E C2 C3 C3 F9 F0 F1 B5 7D 17 33 C5 D2 D1 B6 6D EB 35 86
0x4000203C:	89 DD D0 DC 4F D1 4D 34 5D CD BA C5 6C B1 B6 F0 B5 6D 74 31 C6 2C D1 C7 CD 32 C7 AE B1 B6
0x4000205A:	B4 CD 18 DB 46 B4 88 CB 14 B6 31 34 B4 BC 47 C6 D4 6C AD 4B C4 31 F4 BB DA 55 EF 44 17 5D
0x40002078:	46 34 FE EF 17 4D CF 8B 6D 3E 7A E5 4D 13 EB 06 7F 6D 26 BA 66 79 E5 97 D8 A2 F5 F0 FB 03
0x40002096:	B1 C6 D6 31 4D 53 0D 9F D0 B3 F0 CB CC BA DF 5F 89 79 A3 69 9A E7 59 7F 2B 3F 6B 86 46 F7

RGB 데이터를 저장한 주소 `0x40002000`을 보면 목차 3-4의 기본 코드와 다르게 3바이트 단위로 RGB 값이 연속적으로 저장되어 있는 것을 확인할 수 있다. 이는 기존 RGBA 구조에서 4바이트마다 Alpha 채널이 포함되던 방식과 달리, A값 없이 3바이트 단위로 데이터가 연속 저장되었음을 보여준다.

Memory 1	
Address: 0x40004000	
0x40004000:	00 00
0x4000401E:	00 00
0x4000403C:	00 00
0x4000405A:	00 00

Memory 1	
Address: 0x40004000	
0x40004000:	F8 03 3A 04 59 07 45 01 17 05 84 07 BE 06 F4 04 B2 06 D2 01 A6 03 CE 05 89 03 0E 03 9B 07
0x4000401E:	7C 09 24 05 09 06 24 07 85 04 99 07 3F 05 7C 02 88 07 20 06 7B 07 48 03 31 06 5A 04 E6 06
0x4000403C:	02 07 E9 04 6E 06 7C 03 CB 06 B6 07 8D 04 06 07 7B 05 B3 01 08 03 A4 03 1C 06 7B 04 44 02
0x4000405A:	79 03 34 05 60 07 49 07 B5 08 8D 07 B4 02 B1 04 84 08 9F 07 CC 04 F9 04 4A 05 66 02 2D 04
0x40004078:	F4 03 C8 02 4B 05 96 03 D1 03 7C 06 A4 04 4D 02 F8 01 E8 03 3E 03 B7 06 D0 04 B6 07 A4 07

Grayscale 연산 결과를 저장한 주소 `0x40004000`을 보면 Relocation을 적용하지 않은 기존 코드와 정확히 동일한 결과가 나온 것을 확인할 수 있다.

## (2) Relocation 2 – Alpha 제외, R, G, B 각각 따로 저장

## ① 코드 설명

```

calculate_from_separate PROC
    LDR    R2, =0x40001000    ; R 채널 배열 시작
    LDR    R3, =0x40004000    ; G 채널 배열 시작
    LDR    R4, =0x40007000    ; B 채널 배열 시작
    LDR    R8, =0x4000A000    ; Grayscale 결과 저장
    LDR    R6, =9600          ; 최대 9,600 픽셀
    MOV    R1, #0             ; 픽셀 카운터

process_separate_pixels
    CMP    R1, R6
    BGE    calculate_separate_done

    ; 각 채널 배열에서 값 읽기
    LDRB   R5, [R2, R1]       ; R 값
    LDRB   R7, [R3, R1]       ; G 값
    LDRB   R12, [R4, R1]      ; B 값

    ; Grayscale 공식: 3*R + 6*G + B
    ADD    R5, R5, R5, LSL #1  ; 3*R
    ADD    R7, R7, R7, LSL #1  ; 3*G
    ADD    R7, R7, R7          ; 6*G
    ADD    R5, R5, R7          ; 3*R + 6*G
    ADD    R5, R5, R12         ; 3*R + 6*G + B

    MOV    R10, R1, LSL #1     ; 오프셋 계산 (픽셀 인덱스 × 2)
    STRH   R5, [R8, R10]       ; Grayscale 저장

    ADD    R1, R1, #1
    B      process_separate_pixels

```

위 코드는 각각 따로 분리된 R, G, B 배열로부터 같은 인덱스의 값을 읽어온 후, 이를 바탕으로 Grayscale 값을 계산하여 저장하는 역할을 한다. 각 값은 0x40001000, 0x40004000, 0x40007000부터 시작하는 메모리 블록에 저장되어 있으며, 'LDRB [Rn, R1]' 명령어를 통해 해당 픽셀 인덱스의 값에 직접 접근한다.

계산된 Grayscale 값은 16비트 단위로 0x4000A000 주소부터 저장되며, 픽셀 인덱스 R1에 'LSL #1'을 적용하여 2바이트 단위 주소 오프셋을 계산한다. 이를 통해 각 픽셀의 결과가 정확한 위치에 순차 저장된다.

## ② 실행 결과

Memory 1	
Address: 0x40001000	
0x40001000:	00 00
0x4000101E:	00 00
0x4000103C:	00 00
0x4000105A:	00 00

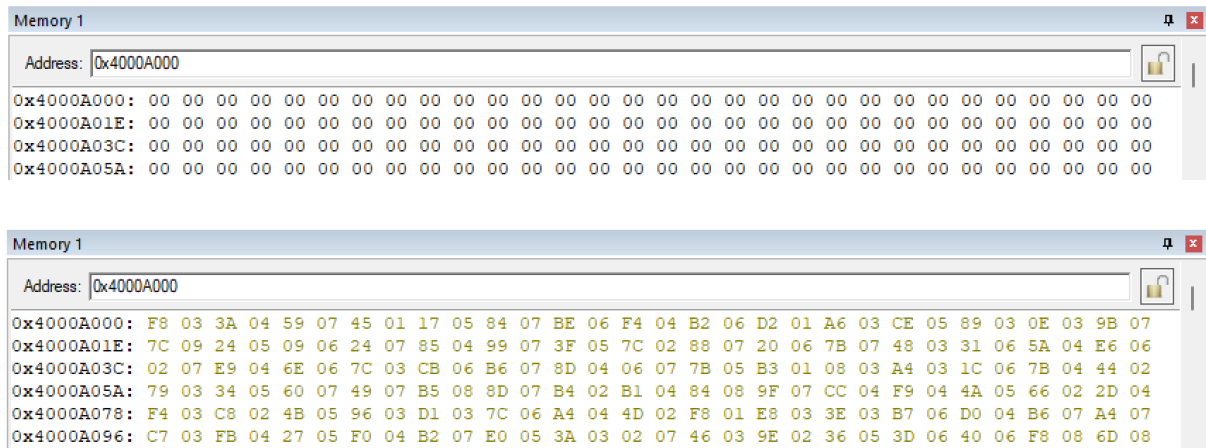
Memory 1	
Address: 0x40001000	
0x40001000:	78 77 92 21 77 67 FF F3 DD 82 67 80 A9 32 C2 F9 B5 33 D1 EB 89 DC 4D CD 6C F0 74 2C CD AE
0x4000101E:	B4 DB 88 B6 B4 C6 AD 31 DA 44 46 EF CF 3E 4D 06 26 79 D8 F0 B1 31 0D B3 CC 5F A3 E7 2B 86
0x4000103C:	CD 05 DE B8 B1 73 DC 44 6D D1 CD 45 6C 7B BC B1 13 D6 9D A2 B3 D3 6D 03 5A 35 FA 5C DF D7
0x4000105A:	55 4D 6A FA 74 C7 C6 69 87 3D D5 88 DB 2C AE F4 C6 B9 F5 6F 56 96 86 A3 7B E1 BE F2 31 9F
0x40001078:	B3 F7 E7 7D FA F5 EB CF DF 36 BA B7 5C AD 68 99 2C F7 99 75 FA BF 87 E5 06 8E B1 2D 46 C4
0x40001096:	87 38 3F F0 0F C5 FE BB FB 38 1A 3F 8A D0 FB E1 6B 51 91 FB 35 59 35 61 A5 AF 1A CF E7 88

Memory 1	
Address: 0x40004000	
0x40004000:	00 00
0x4000401E:	00 00
0x4000403C:	00 00
0x4000405A:	00 00

Memory 1	
Address: 0x40004000	
0x40004000:	5E 73 EC 08 8F EF 7B 3C 9D 02 3F B3 2A 4F C3 F0 7D C5 B6 35 DD 4F 34 BA B1 B5 31 D1 32 B1
0x4000401E:	CD 46 CB 31 BC D4 4B F4 55 17 34 17 8B 7A 13 7F BA E5 A2 FB C6 4D 9F F0 BA 89 69 59 3F 46
0x4000403C:	18 6C 4D 2E 3F CC 46 37 11 2F 12 D3 8D E5 C4 2C C3 4E 6F E3 9E 18 CC 6B 1F 9A 76 B5 EA F0
0x4000405A:	EF 6C 45 AC 23 D0 E6 B1 D0 58 B3 A1 B1 D7 8B 7A F7 2F E7 99 FB 58 E5 0F 76 96 C7 8A 06 CB
0x40004078:	F6 79 E5 BE BE F5 4B 43 E3 42 B2 CD E3 B1 A2 A2 3C CB 6D F1 B1 B5 B6 01 6D 36 6E 23 D3 A4
0x40004096:	88 1C 34 14 B1 61 61 31 18 EC 9E C5 71 8D F6 1A FF 7B 7B 35 A2 CF B6 F9 D1 53 97 D7 3E E5

Memory 1	
Address: 0x40007000	
0x40007000:	00 00
0x4000701E:	00 00
0x4000703C:	00 00
0x4000705A:	00 00

Memory 1	
Address: 0x40007000	
0x40007000:	5C 23 1B B2 58 B5 DF B3 6D 40 F7 1C 92 9E C3 F1 17 D2 6D 86 D0 D1 5D C5 B6 6D C6 C7 C7 B6
0x4000701E:	18 B4 14 34 47 6C C4 BB EF 5D FE 4D 6D E5 EB 6D 66 97 F5 03 D6 53 D0 CB DF 79 9A 7F 6B F7
0x4000703C:	FD 31 E3 5A 44 5B 6C 37 4B 5B 6B F6 3E E7 D8 AC 30 D1 7F 7A 13 31 F3 BB D6 FB 8B EE DF 48
0x4000705A:	D5 6D D3 65 EB 4E E7 63 6E F9 D8 6D 9E 29 6D 3D FA EB 7D F9 6F 63 E0 3E F3 78 FE 6D 7D 9E
0x40007078:	FB 9E 6B FA 8B B2 CF 5D 3D 5E 87 7F 16 B5 8D E3 A3 3D BB 07 F7 A6 EB 6C 88 72 6C 37 8E 75
0x40007096:	18 77 F1 4F 3F 3C 17 1E 4E 0F E2 E9 38 DA 29 5D 75 6F 89 4F 59 11 98 5B B9 4C 69 36 9E BA



Grayscale 연산 결과를 저장한 '0x4000A000'을 보면 Relocation을 적용하지 않은 기존 코드 및 Relocation 1의 결과와 정확히 동일한 연산 결과가 나온 것을 확인할 수 있다.

## 5. 명령어 변경 기반 ARM code 최적화

이번 목차에서는 이전 과정에서 수행했던 Relocation2 (Alpha 제외, R, G, B 각각 따로 저장) 코드를 명령어 변경에 기반하여 각 함수별로 한 번 더 최적화 하는 과정을 설명한다.

### 5-1. Function #1

#### ① 코드 설명

기존 코드:

```
; Read RGBA 4 bytes
LDRB    R5, [R10], #1      ; R
LDRB    R6, [R10], #1      ; G
LDRB    R7, [R10], #1      ; B
ADD     R10, R10, #1       ; Skip A

; Store R, G, B
STRB    R5, [R1], #1       ; R array
STRB    R6, [R2], #1       ; G array
STRB    R7, [R3], #1       ; B array

; Check Red value
CMP     R5, #128
ADDGE   R9, R9, #1         ; Increment count
```

최적화된 코드:

```
; Read 4 bytes (RGBA) at once
LDR     R5, [R10], #4

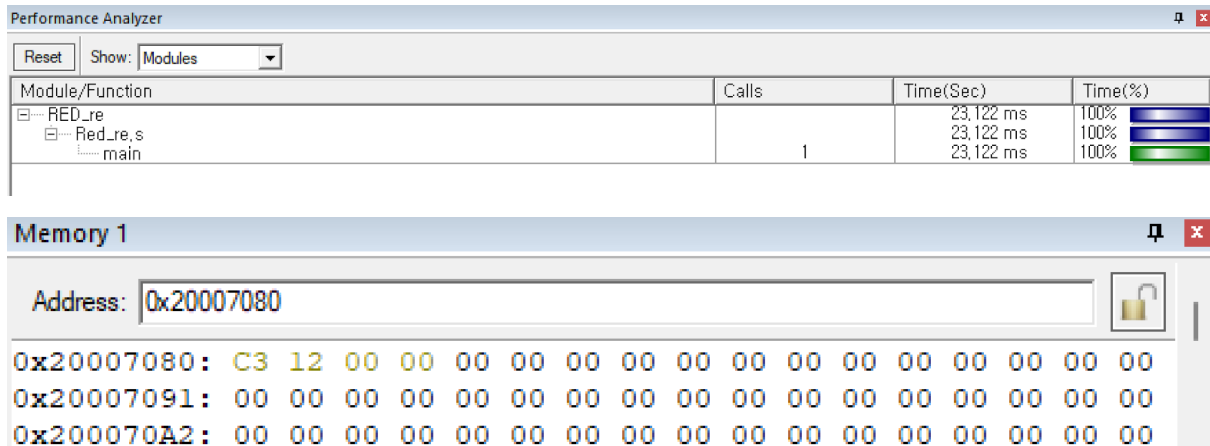
; Extract and store R, G, B
AND     R6, R5, #0xFF
AND     R7, R5, #0xFF00
MOV     R7, R7, LSR #8
AND     R14, R5, #0xFF0000
MOV     R14, R14, LSR #16

STRB    R6, [R1], #1
STRB    R7, [R2], #1
STRB    R14, [R3], #1

; Count Red values
CMP     R6, #128
ADDGE   R9, R9, #1

ADD     R13, R13, #1
B       process_idat_pixels_loop
```

## ② 실행 결과



최적화 작업에서 메모리 접근 횟수를 줄이는데 집중했다. 기존 코드에서는 픽셀 하나를 처리할 때마다 7 회의 메모리 접근이 필요했는데, 최적화된 코드는 이를 4 회로 줄였다. 느린 LDRB 명령어 4 회를 LDR 1 회로 대체하여 메모리 버스 사용량을 줄였다. 이로 인해 메모리 대역폭 사용이 감소했고 대기 시간의 감축으로 이어졌다. 메모리 접근이 줄어들면서 CPU 가 효율적으로 작업을 수행했다고 보여진다. ALU 연산은 증가했지만 AND 와 MOV 같은 연산은 메모리 접근보다 훨씬 빠르다. 그래서 전체적으로 연산 효율이 높아지고 파이프라인 활용도도 좋아졌다. 결과적으로 기존 34.621ms 걸리던 작업이 최적화 후에는 23.122ms 로 약 33%가 빨라졌다. 결과 값은 기본 arm code 와 일치한다.

## 5-2. Function #2

### ① 코드 설명

기존 코드:

```
; RGBA 중 R, G, B를 각각 개별적으로 로드
LDRB R7, [R0]      ; R
LDRB R8, [R0, #1]   ; G
LDRB R9, [R0, #2]   ; B

; 각각 반전
RSB R7, R7, #255
RSB R8, R8, #255
RSB R9, R9, #255

; 저장 (채널 분리 저장)
STRB R7, [R4], #1   ; R 저장
STRB R8, [R5], #1   ; G 저장
STRB R9, [R6], #1   ; B 저장

ADD R0, R0, #4      ; 다음 픽셀로 이동 (RGBA 4바이트)
```

최적화된 코드:

```
; RGBA 4바이트 전체를 한 번에 로드
LDR R7, [R0]        ; R7 = 0xBBGGRRAA

; R 채널 추출 및 반전
ANDS R8, R7, #0xFF   ; R8 = R7 & 0x000000FF
RSB R8, R8, #255

; G 채널 추출 및 반전
LSR R9, R7, #8       ; R9 = R7 >> 8
ANDS R9, R9, #0xFF
RSB R9, R9, #255

; B 채널 추출 및 반전
LSR R10, R7, #16      ; R10 = R7 >> 16
ANDS R10, R10, #0xFF
RSB R10, R10, #255

; 저장
STRB R8, [R4], #1     ; R
STRB R9, [R5], #1     ; G
STRB R10, [R6], #1    ; B

ADD R0, R0, #4        ; 다음 픽셀로 이동
```

해당 함수는 PNG 이미지 내 IDAT 청크를 탐색한 뒤 RGBA 데이터를 처리하여, R, G, B 값을 각각 지정된 메모리 주소에 분리 저장한다. 각 채널은 255 - 값 방식으로 반전되며, Alpha 채널은 무시된다.


기존 버전에서는 각 픽셀의 R, G, B 값을 LDRB 명령어로 각각 1 바이트씩 메모리에서 불러오고, 각각 반전 및 저장하는 구조였다. 메모리 접근이 한 픽셀당 세 번 발생하고, 연산도 채널별로 나뉘어 있어 루프당 명령어 수가 많았다.


최적화 버전에서는 LDR 명령어로 RGBA 4 바이트를 한 번에 로드한 뒤, 시프트와 마스크 연산을 통해 각 채널을 레지스터 내에서 분리한다. 이후 반전 연산과 저장까지 모두 레지스터 기반으로 처리되며, 메모리는 한 번만 접근한다. 루프는 4 바이트 단위로 이동하므로 Alpha 채널도 별도 처리 없이 자연스럽게 제외된다.


이처럼 메모리 접근을 줄이고 비트 연산으로 채널을 분리한 구조 덕분에 전체 명령어 수와 실행 시간이 감소하며, 성능이 실질적으로 향상되었다.





## ② 실행 결과


Address: 0x20000000		
0x20000000:	00 00	
0x20000016:	00 00	
0x2000002C:	00 00	
0x20000042:	00 00	
0x20000058:	00 00	

Address: 0x20000800		
0x20000800:	00 00	
0x20000816:	00 00	
0x2000082C:	00 00	
0x20000842:	00 00	
0x20000858:	00 00	

Address: 0x20001000		
0x20001000:	00 00	
0x20001016:	00 00	
0x2000102C:	00 00	
0x20001042:	00 00	
0x20001058:	00 00	

Address:	<input type="text" value="0x20000000"/>	
0x20000000:	88 6D DE 88 98 00 0C 22 7D 98 7F 56 CD 3D 06 4A CC 2E 14 76 23 B2	
0x20000016:	32 93 0F 8B D3 32 51 4B 24 77 49 4B 39 52 CE 25 BB B9 10 30 C1 B2	
0x2000002C:	F9 D9 86 27 0F 4E CE F2 4C 33 A0 5C 18 D4 79 32 FA 21 47 4E 8C 23	
0x20000042:	BB 92 2E 32 BA 93 84 43 4E EC 29 62 5D 4C 2C 92 FC A5 CA 05 A3 20	
0x20000058:	28 AA B2 95 05 8B 38 39 96 78 C2 2A 77 24 D3 51 0B 39 46 0A 90 A9	
0x2000006E:	69 79 5C 84 1E 41 0D CE 60 4C 08 18 82 05 0A 14 30 20 C9 45 48 A3	
0x20000084:	52 07 66 D3 00 66 03 0F 40 70 13 F0 51 4F D3 D0 3D 50 07 0E F0	

Address:	<input type="text" value="0x20000800"/>	
0x20000800:	FF B6 E5 B6 0B 26 67 1F B5 BF D4 C0 4B A2 03 C0 14 5F 9D 0E 26 FF	
0x20000816:	CA C5 73 A0 BC CD 01 33 E6 AA B5 31 FB A3 93 66 09 8C CB 2E D4 23	
0x2000082C:	6C 03 5D BD 8A DE 00 51 BA 05 90 FA 4B F2 3F 7E C9 8F 8C 11 62 B0	
0x20000842:	51 20 06 7E 58 1C B5 0D B5 25 A3 7E 13 99 6E CF A9 EF B2 C5 36 FB	
0x20000858:	37 B7 90 1C 47 B0 18 50 40 23 9C 42 F3 28 21 72 40 41 81 50 83 B8	
0x2000086E:	AD BB 8D 95 2C CD 4A DC B2 36 BE C5 AF 6A 5D 6E 83 5D 09 DE D7 43	
0x20000884:	0E 07 10 ED D0 0E D3 D0 65 D5 3D 36 E0 35 03 75 73 DD D0 00 35 35	

Address:	<input type="text" value="0x20001000"/>	
0x20001000:	9E F1 21 FF B6 56 DA 9D 1C 56 03 FA C2 10 43 FE CE 98 67 CD D1 1E	
0x20001016:	6D 48 AF AA 29 01 0F B6 A6 20 39 74 C5 BA 7E 3F 8E 78 B3 9F 0E 2E	
0x2000102C:	EF 63 10 45 6D CD 77 A8 AC E8 13 AA 4C E3 53 75 0B BF 13 EF 96 65	
0x20001042:	77 E4 27 17 11 B2 7E 2D CE 19 C6 4D 03 C5 B7 52 08 F3 FA F6 21 52	
0x20001058:	A6 A6 B5 FA 5D 7F B6 34 52 96 3D 28 E3 C4 CD 09 B3 BF AC 7C 74 9F	
0x2000106E:	7B 1C 38 BB 64 6D 87 C3 4E CB 10 8B 59 98 3A 28 7B 0A EB 68 90 5D	
0x20001084:	D0 E7 E6 2C ED 07 E0 00 00 CF 03 E0 00 10 C0 C0 64 DD 00 70 ED 00	

Module/Function	Calls	Time(Sec)	Time(%)
plz		23,110 ms	100%
red_pixel,s		23,110 ms	100%

사용된 PNG 이미지의 첫 번째 픽셀은 RGBA 기준으로 77 73 23 XX 이며, 이를 10 진수로 환산하면 R = 119, G = 115, B = 35 이다. 이 값은 각각 255 에서 뺀 방식으로 반전 처리되어 R = 136 (0x88), G = 140 (0x8C), B = 220 (0xDC)로 계산된다. Alpha 채널은 사용하지 않기 때문에 연산과 저장 대상에서 제외된다. 시뮬레이션 결과에 따르면, 최종 RGB 반전 값은 메모리 주소 0x20000000 부터 순차적으로 저장되며, 해당 주소에는 88 6D DE 88 98 00 등 반전된 값들이 정확히 기록되어 있는 것을 확인할 수 있다. 각 픽셀은 RGBRGB... 형식으로 나열되어 있으며, 실제로 첫 번째 세 바이트가 각각 반전된 R, G, B 값과 일치한다. 이로써 데이터 분리 및 반전 처리 로직이 정확히 수행되었음을 검증할 수 있다.

R 채널 데이터는 0x20000000, G 채널은 0x20000800, B 채널은 0x20001000 부터 시작되며, 각각의 주소 공간에 해당 채널 값만이 순서대로 저장되어 있다. 예를 들어, 0x20000000 에는 R 값 88, 6D, DE 등이 저장되고 있고, G 채널이 저장된 0x20000800 에는 F7, B6, E5 등 반전된 G 값들이 독립적으로 저장되어 있는 것을 확인할 수 있다. 마찬가지로 B 채널은 0x20001000 부터 시작되며, F1, A1, 5B 등이 순차적으로 기록되어 있다. 본 함수는 LDR 명령어를 통해 RGBA 4 바이트를 한 번에 불러오고, 비트 마스크 및 시프트 연산을 통해 R, G, B 를 추출하고 각각 반전한 뒤, 지정된 메모리 주소에 분리 저장하는 방식으로 최적화되어 있다. 기존 방식처럼 LDRB 를 3 회 반복하여 각 채널을 따로 불러오는 방식보다 메모리 접근 횟수가 현저히 줄었으며, 레지스터를 반복적으로 재사용하여 전체 명령어 수 또한 감소하였다. 루프는 4 바이트 단위로 순차 이동하므로 Alpha 채널은 자동으로 건너뛰게 되며, 코드 구조는 간결하고 실행 속도는 향상되었다.

결과적으로 실험을 통해 RGB 분리 및 반전 연산이 정확하게 수행되었고, 최적화 전략이 실제 메모리 결과에도 잘 반영되었음을 확인할 수 있었다.

### 5-3. Function #3

#### ① 코드 설명

기존 코드:

```
ADD    R5, R5, R5, LSL #1 ; 3*R
ADD    R7, R7, R7, LSL #1 ; 3*G
ADD    R7, R7, R7      ; 6*G
ADD    R5, R5, R7      ; 3*R + 6*G
ADD    R5, R5, R12     ; 3*R + 6*G + B
```

최적화 코드:

```
MLA    R11, R7, R10, R12 ; R11 = (G * 6) + B
MLA    R12, R5, R9, R11  ; R12 = (R * 3) + (G * 6 + B)
```

기존 코드에서는 Grayscale 값( $3 \times R + 6 \times G + B$ )을 계산하기 위해 각각의 곱셈을 비트 시프트와 덧셈 명령어로 나누어 수행하였다. 예를  $6 \times G$  연산은 'ADD R7, R7, R7, LSL #1'과 'ADD R7, R7, R7'라는 명령어를 사용하여  $6 \times G$ 를 한 번에 계산하지 않고 단계적으로 계산하는 방법을 이용했으며, 최종적으로 여러 단계의 ADD 연산을 통해 결과를 산출하였다. 이 방식은 명령어 수가 많아 루프당 처리 비용이 증가하는 단점이 있다.

최적화한 코드에서는 ARM의 'MLA' 명령어를 사용하여 기존 코드에서 5개의 명령어로 처리되던 것을 2개의 명령어로 압축하였다. 먼저 'MLA R11, R7, R10, R12'를 통해  $6 \times G + B$ 를 한 번에 계산하고, 이어서 'MLA R12, R5, R9, R11'을 통해  $3 \times R + (6 \times G + B)$ 를 계산한다. 이는 루프 내의 명령어 수를 효과적으로 감소시키는 방법이다.

## ② 실행 결과

Memory 1	
Address: 0x4000A000	
0x4000A000:	00 00
0x4000A01E:	00 00
0x4000A03C:	00 00
0x4000A05A:	00 00

Memory 1	
Address: 0x4000A000	
0x4000A000:	F8 03 3A 04 59 07 45 01 17 05 84 07 BE 06 F4 04 B2 06 D2 01 A6 03 CE 05 89 03 0E 03 9B 07
0x4000A01E:	7C 09 24 05 09 06 24 07 85 04 99 07 3F 05 7C 02 88 07 20 06 7B 07 48 03 31 06 5A 04 E6 06
0x4000A03C:	02 07 E9 04 6E 06 7C 03 CB 06 B6 07 8D 04 06 07 7B 05 B3 01 08 03 A4 03 1C 06 7B 04 44 02
0x4000A05A:	79 03 34 05 60 07 49 07 B5 08 8D 07 B4 02 B1 04 84 08 9F 07 CC 04 F9 04 4A 05 66 02 2D 04
0x4000A078:	F4 03 C8 02 4B 05 96 03 D1 03 7C 06 A4 04 4D 02 F8 01 E8 03 3E 03 B7 06 D0 04 B6 07 A4 07
0x4000A096:	C7 03 FB 04 27 05 F0 04 B2 07 E0 05 3A 03 02 07 46 03 9E 02 36 05 3D 06 40 06 F8 08 6D 08
0x4000A0B4:	6E 07 DC 03 AF 03 5B 07 19 03 83 07 9D 08 C4 05 E3 06 C0 03 89 07 CB 05 55 07 B7 05 B9 05
0x4000A0D2:	F5 05 16 09 30 04 C6 08 DC 05 53 07 35 04 D0 07 81 02 28 05 9F 06 E2 07 7F 06 34 01 3D 07
0x4000A0F0:	D8 08 59 06 7E 08 E5 06 ED 07 4F 09 52 05 5C 04 2C 08 8C 02 E1 06 72 07 7C 06 E2 06 91 05
0x4000A10E:	7A 06 8F 02 E4 07 14 05 0C 07 0B 08 21 07 C4 06 21 03 28 03 60 03 13 05 90 01 52 06 99 06
0x4000A12C:	DD 04 C7 01 E6 02 97 03 92 04 D1 04 57 05 75 03 CF 03 3F 06 E4 04 44 06 7C 04 98 06 DE 08

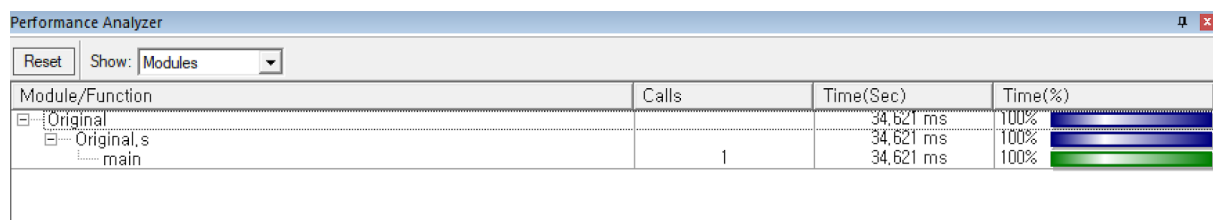
Grayscale 연산 결과는 '0x4000A000'부터 저장되며 각 픽셀마다 16비트 단위로 값이 저장된다. 기존 5개의 명령어를 2개로 압축시킨 최적화 코드 또한 이전 과정의 모든 코드들과 동일한 연산 결과가 나오는 것을 확인할 수 있다.

## 6. 성능 비교 분석

이번 챕터는 ARM 아키텍처에서 동작하는 이미지 변환 함수의 성능을 비교하는 내용을 담고 있다. 비교 대상은 기본 ARM 코드와 Relocation이 진행된 코드이다. 각 함수의 전체 수행 시간과 필요한 명령어 개수, 그리고 메모리 접근 효율성 등 여러 측면에서 분석을 진행할 예정이다.

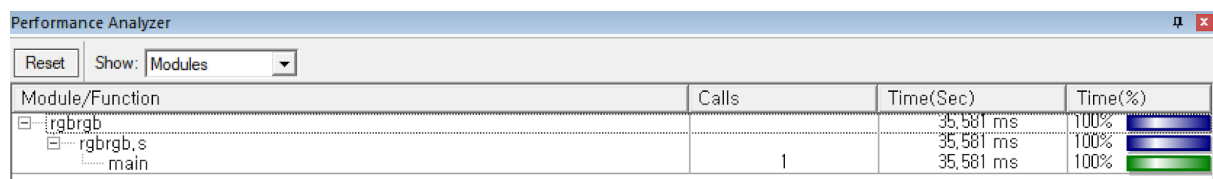
### 6-1. Function #1

#### ① 기본 코드



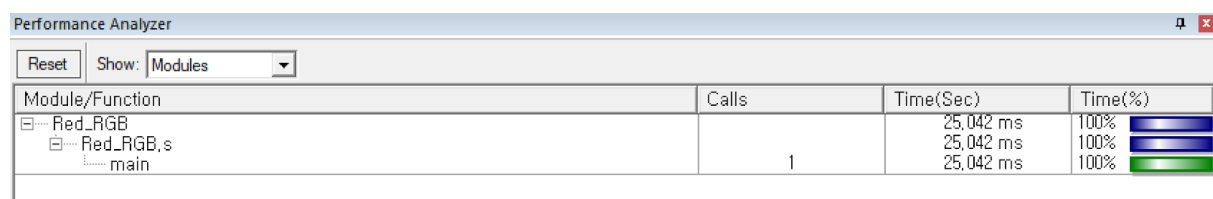
Module/Function	Calls	Time(Sec)	Time(%)
Original		34,621 ms	100%
Original.s		34,621 ms	100%
main	1	34,621 ms	100%

#### ② Relocation 1 - Alpha 제외하고 저장



Module/Function	Calls	Time(Sec)	Time(%)
rgbrgb		35,581 ms	100%
rgbrgb.s		35,581 ms	100%
main	1	35,581 ms	100%

#### ③ Relocation 2 - Alpha 제외, R, G, B 각각 따로 저장

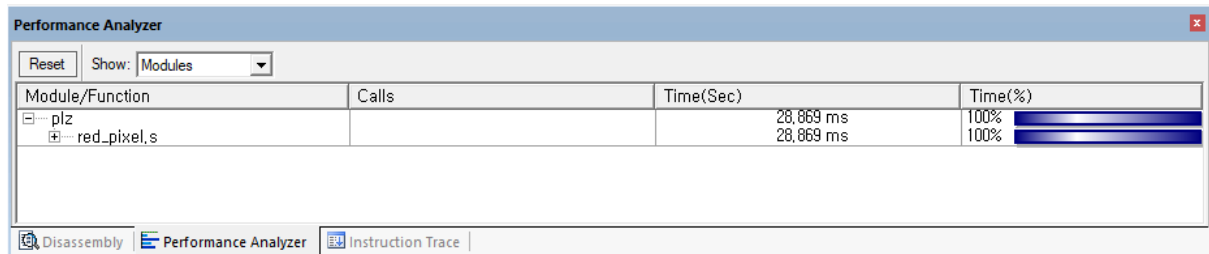


Module/Function	Calls	Time(Sec)	Time(%)
Red_RGB		25,042 ms	100%
Red_RGB.s		25,042 ms	100%
main	1	25,042 ms	100%

ARM 코드 3종류(기본, relocation 1, relocation 2)의 성능 비교를 위해 각각의 수행 시간, 명령어 개수 및 주요 특징을 분석했다. 기본 ARM 코드는 RGBA 픽셀 데이터를 4바이트 단위로 읽고 32비트 값으로 조합하여 저장하는 방식이다. 이 방식은 메모리에 두 번 접근해야 하므로 전체 수행 시간이 34.621 밀리초로 측정되었다. 두 번째 코드는 RGB 24비트 데이터를 추출하여 RAM에 인터리브 형태로 저장한 후, 저장된 데이터에서 Red 값을 카운트한다. 첫 번째 코드와 마찬가지로 메모리 접근이 두 번 발생하며, 수행시간은 35.581 밀리초로 측정되었다. 첫 번째 코드와 비교했을 때 데이터 저장 방식만 변경되었으나 수행 시간은 소폭 증가했다. 세 번째 코드는 R, G, B 데이터를 추출하여 각각 별도의 배열에 저장하면서 동시에 Red 값을 카운트하는 방식으로 구현되었다. 이 방식은 메모리 접근 횟수를 최소화하여 전체 수행 시간을 25.042 밀리초로 단축시켰다. 결론적으로, 세 가지 코드 중 세 번째 코드가 가장 효율적인 것으로 분석되었다. 메모리 접근 횟수를 줄이고 데이터 처리와 카운트 작업을 동시에 수행함으로써 전체 수행 시간을 단축시킨 점이 주요 개선 요인으로 판단된다.

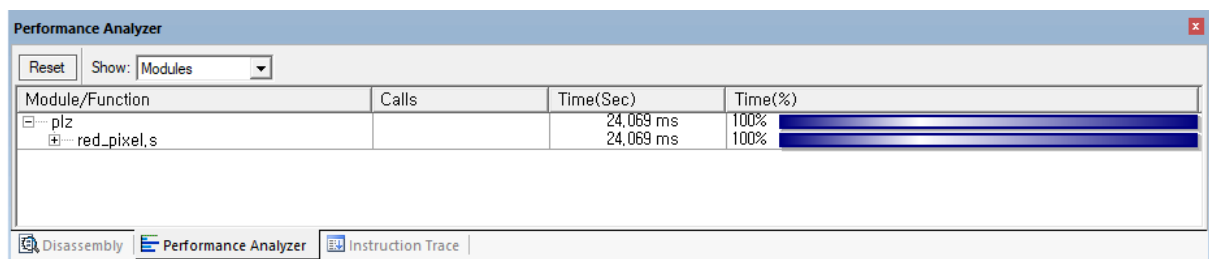
## 6-2. Function #2

### ① 기본 코드



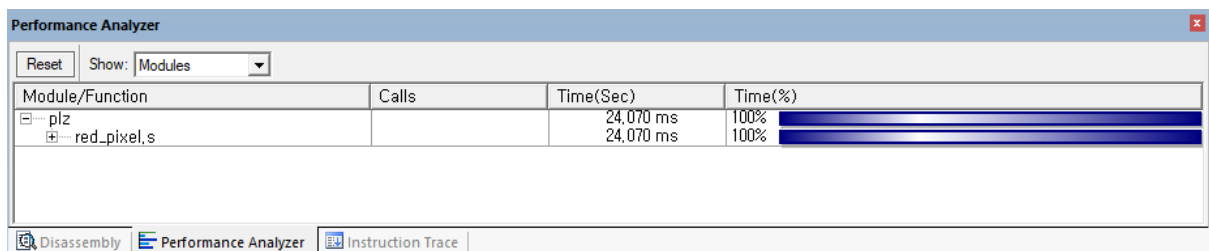
Module/Function	Calls	Time(Sec)	Time(%)
plz		28,869 ms	100%
red_pixel,s		28,869 ms	100%

### ② Relocation 1 - Alpha 제외하고 저장



Module/Function	Calls	Time(Sec)	Time(%)
plz		24,069 ms	100%
red_pixel,s		24,069 ms	100%

### ③ Relocation 2 - Alpha 제외, R, G, B 각각 따로 저장



Module/Function	Calls	Time(Sec)	Time(%)
plz		24,070 ms	100%
red_pixel,s		24,070 ms	100%

ARM 기반으로 구현된 이미지 처리 코드 3가지(기본 버전, Relocation 1, Relocation 2)는 각각 RGBA 데이터를 어떤 방식으로 다루고 저장하느냐에 따라 성능에 차이를 보인다.

기본 코드는 RGBA 픽셀 데이터를 4바이트 단위로 읽어들이며, R, G, B 채널 값은 반전하여 저장하고, Alpha 채널(A)은 가공 없이 그대로 포함한다. 이 방식은 네 채널 모두 처리하기 때문에 메모리 접근이 많고, 처리 시간이 가장 길게 측정되었다. 실제로 총 9600개의 픽셀을 처리하는 데 걸린 시간은 약 28.683밀리초였다.

Relocation 1에서는 Alpha 채널을 아예 무시하고, R/G/B 세 채널만 반전 처리한 뒤 RGBRGB... 식으로 연속된 메모리에 저장하는 방식이다. 채널 수가 줄어들면서 연산과 저장 횟수도 감소해, 전체적으로 보다 간결한 루프 구조가 가능해졌다. 이로 인해 실행 시간은 약 24.088밀리초로, 기존 방식보다 약 4.6ms가 줄었다. 성능 향상의 핵심은 Alpha 채널 제거에서 비롯된다.

Relocation 2는 Relocation 1과 동일하게 Alpha는 제외하지만, R, G, B 값을 각각의 독립된 메모리 영역에 따로 저장한다는 점에서 구조적으로 다르다. 예를 들어 R은 0x20000000부터, G는 0x20000800부터, B는 0x20001000부터 저장된다. 이 방식은 메모리상의 정리가 잘 되어 있어 이후 연산이나 분석에 유리하며, 성능 면에서도 Relocation 1과 거의 같은 수준인 24.070밀리초를 기록했다.

결론적으로, 전체 수행 시간에 가장 큰 영향을 준 요소는 Alpha 채널의 제거였으며, R/G/B 저장 방식의 차이(RGB 연속 저장 vs 분리 저장)는 성능에 큰 영향을 주지 않았다. 다만 구조적 명확성과 유지보수 측면에서는 Relocation 2 방식이 더 우수하다고 평가된다.



### 6-3. Function #3

성능 분석 지표로는 명령어 수와 Keil MDK의 Performance Analyzer 결과를 함께 활용하였다. 명령어 수 기준 분석에서는 프로그램 전체 명령어 수가 아닌, 실제 연산에서 반복적으로 수행되는 루프 영역을 중심으로 비교하였다. 전체 명령어 수는 초기 설정, IDAT 청크 탐색 등 공통적으로 반복되지 않는 구간이 포함되어 정확한 비교가 어렵기 때문이며, 루프 내부 명령어 수가 성능에 더 큰 영향을 미치므로 RGB(A) 저장 루프와 Grayscale 계산 루프를 중심으로 명령어 수를 카운트 하였다.

## ① 기본 코드

Performance Analyzer		
Reset	Show: Modules	
Module/Function	Calls	Time(Sec)
test		51.913 ms
subproject3_basic.s		51.913 ms
main	1	0.600 us
extract_rgba_data	1	27.911 ms
calculate_grayscale_rgba	1	24.001 ms

RGBA 저장 루프: 11개

Grayscale 계산 루프: 14개

## ② Relocation 1 - Alpha 제외하고 저장

Performance Analyzer		
Reset	Show: Modules	
Module/Function	Calls	Time(Sec)
test		46.153 ms
subproject3_relocation1.s		46.153 ms
main	1	0.600 us
extract_rgb_data	1	23.111 ms
calculate_grayscale	1	23.041 ms

RGB 저장 루프: 9개

Grayscale 계산 루프: 13개

## ③ Relocation 2 - Alpha 제외, R, G, B 각각 따로 저장

Performance Analyzer		
Reset	Show: Modules	
Module/Function	Calls	Time(Sec)
test		45.194 ms
subproject3_relocation2.s		45.194 ms
main	1	0.600 us
extract_separate_rgb	1	23.112 ms
calculate_from_separate	1	22.082 ms

RGB 각각 저장 루프: 9개

Grayscale 계산 루프: 12개

명령어 수 기준으로 보면 기본 코드는 RGBA의 4바이트를 모두 저장하기 때문에 저장 루프 내에 11개의 명령어가 반복되며, Grayscale 연산에는 총 14개의 명령어가 사용된다. 반면 Relocation 1은 Alpha를 제외한 RGB 값만 연속으로 저장하기 때문에 저장 루프에서 9개의 명령어로 줄어들고, Grayscale 연산도 A값이 필요 없으므로 간소화된 구조로 13개의 명령어가 사용된다. 가장 최적화된 구조인 Relocation 2에서는 R, G, B를 각각 다른 메모리에 저장하여 연산 중 간단한 메모리 접근을 가능하게 하며, Grayscale 계산 루프에서 명령어 수는 가장 적은 12개로 줄어든다.

실행 시간 측면에서 보면 기본 코드의 전체 수행 시간은 약 51.913ms이며, 저장과 계산이 각각 약 27.911ms, 24.001ms로 나뉜다. Relocation 1에서는 저장 루틴과 계산 루틴 모두 약간의 시간이 감소하여 총 46.153ms가 소요되며, Relocation 2는 가장 빠른 45.194ms로 측정되었다. 특히 Grayscale 계산이 22.082ms로 가장 짧아, 명령어 수 감소가 실제 실행 속도에도 긍정적인 영향을 준 것을 확인할 수 있다.

결과적으로 명령어 수 기준과 실제 측정된 실행 시간을 종합적으로 고려할 때, Relocation 2 방식이 가장 효율적인 구조임을 확인할 수 있다.

## 7. Trouble Shooting

프로젝트를 처음 진행할 때 가장 먼저 마주한 문제는 hex 파일의 데이터 구성을 정확히 이해하지 못한 데에서 비롯되었다. 초기에는 이미지 데이터를 구성하는 IDAT가 파일 내에 하나만 존재한다고 생각하고 단일 IDAT를 찾은 뒤 그 이후의 모든 데이터를 픽셀 데이터로 처리하는 방식으로 코드를 작성하였다. 그러나 hex 파일을 분석해보니 IDAT는 하나가 아니라 여러 개가 존재할 수 있으며, 이는 PNG 파일 포맷의 표준적인 구성 방식에 부합하는 것이었다.

```
[Header]
[Chunk 1] ← 예: IDAT (픽셀 데이터 일부)
[Chunk 2] ← 예: IDAT (픽셀 데이터 나머지)
```

PNG(Hex) 파일은 하나의 덩어리가 아닌 여러 개의 청크 구조로 구성된다. 프로젝트 명세에도 설명되어 있는 것처럼 IDAT 이후에 픽셀이 시작되는 것은 맞으나, 픽셀 데이터가 너무 많을 경우 위처럼 IDAT 하나에 전체 픽셀 정보를 다 담지 않고 여러 개의 IDAT 구조로 나눠 저장하는 것이다.

```
| Length (4 bytes) | IDAT (4 bytes) | Data (variable) | CRC (4 bytes) |
```

각 청크는 위 구조로 구성되는데, 여기서 CRC란 IDAT와 Data 전체에 대해 계산된 오류를 검출하는 코드이다. 따라서 하나의 IDAT 청크를 처리한 뒤, 그 청크의 마지막에 있는 CRC 4바이트를 건너뛰고, 다음 청크의 시작 주소로 이동하는 과정을 거쳐서 코드를 구성해야 한다.

초기 코드는 다음과 같다.

```
FindIDAT
    LDRB    R2, [R1]           ; R2 ← [R1] ('I'인지 확인)
    CMP     R2, #'I'
    BNE     NextCheck         ; 아니면 다음 주소 확인
    LDRB    R2, [R1, #1]       ; R2 ← [R1+1]
    CMP     R2, #'D'
    BNE     NextCheck
    LDRB    R2, [R1, #2]       ; R2 ← [R1+2]
    CMP     R2, #'A'
    BNE     NextCheck
    LDRB    R2, [R1, #3]       ; R2 ← [R1+3]
    CMP     R2, #'T'
    BNE     NextCheck

    ADD     R1, R1, #4         ; IDAT(4 바이트) 지나고 픽셀 시작 위치(0x78)
    B       StartLoop

NextCheck
    ADD     R1, R1, #1         ; 한 바이트 뒤로 이동해서 다시 검사
    B       FindIDAT
```

초기 버전의 코드는 'IDAT' 문자열을 메모리에서 단 한 번만 탐색한 후, 그 지점을 시작으로 픽셀 데이터를 9600개 연속해서 읽는 방식으로 설계하였다. 이 코드는 IDAT가 하나만 존재할 것이라는 전제를 두고 작성된 것이다. 그러나 실제 PNG 파일 구조상 IDAT 청크는 여러 개로 나뉘어 존재할 수 있기 때문에, 이러한 방식은 이후에 이어지는 다른 IDAT 청크의 데이터를 무시하게 되는 문제가 있었고, 상기 코드로 Keil MDK에서 실행했을 때 RGBA를 정상적으로 찾아내지 못하는 문제가 발생하였다. 결과적으로 일부 이미지 데이터가 누락되거나 잘못 처리될 수 있는 위험이 있었고, 이를 해결하기 위해 IDAT 청크를 반복적으로 탐색하도록 코드가 수정되었다.

수정한 코드는 다음과 같다.

```
search_next_idat
    CMP    R0, R9                ; memory range check
    BGT    store_result          ; search 끝

    ; search for 'IDAT'
    LDRB   R3, [R0]
    CMP    R3, #'I'
    BNE    skip_byte
    LDRB   R3, [R0, #1]
    CMP    R3, #'D'
    BNE    skip_byte
    LDRB   R3, [R0, #2]
    CMP    R3, #'A'
    BNE    skip_byte
    LDRB   R3, [R0, #3]
    CMP    R3, #'T'
    BNE    skip_byte

    ; IDAT chunk length (Little-endian)
    SUB    R7, R0, #4
    LDRB   R3, [R7]
    LDRB   R4, [R7, #1]
    LDRB   R5, [R7, #2]
    LDRB   R12, [R7, #3]
    MOV    R11, R12, LSL #24
    ORR    R11, R11, R5, LSL #16
    ORR    R11, R11, R4, LSL #8
    ORR    R11, R11, R3        ; R11 ← IDAT length

    ADD    R10, R0, #4          ; R10 ← IDAT data start
    ADD    R11, R10, R11        ; R11 ← IDAT data end
```

초기 코드는 파일 내에 IDAT 청크가 하나만 존재한다고 가정하고, 첫 번째 IDAT 청크를 찾은 뒤 바로 그 다음 바이트부터 픽셀 데이터를 순차적으로 읽는 방식으로 구성하였고, 픽셀을 잘 읽어들이지 못하는 문제가 발생했다. 이에 따라 수정된 코드에서는 메모리 끝 주소까지 반복적으로 IDAT 청크를 탐색하고, 각 청크의 크기를 Little-endian 방식으로 계산하여 그 범위 내에서만 픽셀을 처리한 뒤 CRC 4바이트를 건너뛰고 다음 청크를 찾아가는 구조로 변경하였다. 이를 통해 여러 개의 IDAT 청크가 있는 경우에도 모든 픽셀 데이터를 빠짐없이 처리할 수 있도록 개선하였으며, 프로그램 상에서도 픽셀을 정확히 인식하여 메모리에 탑재하는 것을 확인할 수 있었다.

## 8. 프로젝트 수행일지

이름	소희연(조장)	신주환	유진
모임일자	수행내용		
5/8	<ul style="list-style-type: none"> <li>• 각자의 진행 방향에 대한 역할 분담: 1, 2, 3번 문항의 난이도가 서로 다름. 각자 1, 2, 3번 문항을 모두 풀어본 후, 각자 어떻게 코드를 짰는지 공유하고 합치는 방식으로 결정.</li> <li>• 발표 및 PPT 제작 등은 추후 결정 예정.</li> <li>• 비대면 회의 진행.</li> <li>• 노선에 회의록 저장.</li> </ul> #) 17일까지 1번 문항 풀기		
5/17	<ul style="list-style-type: none"> <li>• IDAT 값 처리하는 과정에서 이슈 발생</li> <li>• IDAT 다음 값부터 카운트 필요, 첫 번째 IDAT 이전 HEADER INFO 제외, IDAT 값 반복적 제외 필요</li> <li>• 소희연님 코드에서 2번째 IDAT 값 이후 카운팅 이슈</li> <li>• HEX 파일 vs Keil 값 불일치</li> <li>• 이슈: IDAT 주소 값 하드코딩 여부</li> <li>• 결론: 다른 사진 적용 시에도 동작 가능한 코드 필요</li> </ul> #) 22일까지 IDAT 제외 코드 작성, 피드백		
5/22	<ul style="list-style-type: none"> <li>• Red Pixel Counting 값 불일치 문제 해결</li> <li>• 이슈: 처음에는 GPT가 카운트한 값과 팀원들의 계산 값이 서로 달랐음.</li> <li>• 해결: 코드를 다시 실행해 본 결과, 팀원 3명이 모두 동일한 값이 나옴.</li> <li>• 결론: 계산된 픽셀 값 최종 확정.</li> </ul> #)25일까지 2번 문항 풀기		

이름	소희연(조장)	신주환	유진
모임일자	수행내용		
5/25	<ul style="list-style-type: none"> <li>• 어려웠던 점: 유진님: 루프 2번 도는 오류 (A 값에 완료 플래그 추가로 해결)</li> <li>소희연님: MDK 경로 인식 오류 (재실행 후 해결)</li> <li>신주환님: 큰 어려움 없었음</li> <li>• 결론: 2번 문항 해결 완료</li> <li>• 3번 문제와 최적화 해야하는 문제도 있어서 31일에 3번 문제를 풀고 6월 1일에 최적화를 끝내기로 결정</li> </ul> #) 31일 까지 보고서, ppt, 발표 어떻게 분담할지 생각해오기로 함		
5/31	<ul style="list-style-type: none"> <li>• A값 빼고 RGB값 저장하는 코드와 기본 코드에서 메모리에 RGBA값을 안 불러 오는 것 같아서 수정하기로 결정</li> <li>• 최적화 됐는지 Performance Analyzer를 사용하고 실행 시간 측정하기로 함</li> </ul> #)1일까지 최적화 마치기로 결정, 최적화할 문제와 ppt, 발표에 대한 역할을 정함		
6/1	<ul style="list-style-type: none"> <li>• 각자 relocation한 코드를 실행해봤으나 Break point 없이는 실행 시간 측정이 불가능함을 인지→Break point하고 실행 시간 측정하기로 결정</li> <li>• 혹시 모르니 교수님께 Break point에 관한 조언을 얻기로 함</li> </ul> #)4일까지 보고서 완성하기로 결정		
6/7	<ul style="list-style-type: none"> <li>• 대대적인 수정과 피드백을 진행함</li> <li>• PPT, 발표, 보고서를 보면서 수정을 진행함</li> </ul>		

## 9. 부록

### 9-1. Function #1

#### ① 기본 코드

```

AREA    RESET, CODE, READONLY
ENTRY

main    PROC
; 초기화
    LDR    R0, =0x40000000    ; 소스 데이터 시작 주소 (ROM2)
    LDR    R9, =0x4003FFFF    ; ROM2 끝 주소 (256KB)
    LDR    R8, =0x20000000    ; RGBA 저장 시작 주소 (RAM, 4 바이트 정렬)
    LDR    R12, =0x20009600    ; 결과 카운트 저장 주소 (9600*4 뒤,
0x20000000 + 9600*4)
    MOV    R1, #0            ; 픽셀 카운터
    LDR    R6, =9600          ; 최대 픽셀 수

search_next_idat
    CMP    R0, R9
    BGT    start_counting
    CMP    R1, R6
    BGE    start_counting

    LDRB   R2, [R0], #1
    CMP    R2, #'I'
    BNE    search_next_idat
    LDRB   R2, [R0], #1
    CMP    R2, #'D'
    BNE    search_next_idat
    LDRB   R2, [R0], #1
    CMP    R2, #'A'
    BNE    search_next_idat
    LDRB   R2, [R0], #1
    CMP    R2, #'T'
    BNE    search_next_idat

; 'IDAT' 시그니처 발견!
    SUB    R10, R0, #8
    LDRB   R2, [R10], #1
    LDRB   R3, [R10], #1
    LDRB   R4, [R10], #1
    LDRB   R5, [R10], #1
    MOV    R11, R5, LSL #24
    ORR    R11, R11, R4, LSL #16
    ORR    R11, R11, R3, LSL #8
    ORR    R11, R11, R2
    MOV    R10, R0
    ADD    R11, R10, R11    ; 데이터 끝 주소

```



```
store_rgba
    CMP    R10, R11
    BGE    next_idat
    CMP    R1, R6
    BGE    start_counting

    LDRB    R2, [R10], #1        ; R
    LDRB    R3, [R10], #1        ; G
    LDRB    R4, [R10], #1        ; B
    LDRB    R5, [R10], #1        ; A
    ORR     R2, R2, R3, LSL #8
    ORR     R2, R2, R4, LSL #16
    ORR     R2, R2, R5, LSL #24
    STR     R2, [R8], #4
    ADD     R1, R1, #1
    B       store_rgba

next_idat
    ADD     R0, R11, #4
    B       search_next_idat

start_counting
    LDR     R8, =0x20000000
    MOV     R1, #0
    MOV     R2, #0

count_loop
    CMP     R1, R6
    BGE     final_store
    LDRB    R3, [R8], #4
    CMP     R3, #128
    ADDGE   R2, R2, #1
    ADD     R1, R1, #1
    B       count_loop

final_store
    STR     R2, [R12]            ; 최종 결과 저장
    BX     LR                    ; 종료

    LTORG
    ENDP
    END
```

## ② Relocation 1 – Alpha 제외하고 저장

```

AREA      RESET, CODE, READONLY
ENTRY

main      PROC
; 초기화
LDR       R0, =0x40000000
LDR       R9, =0x4003FFFF
LDR       R8, =0x20000000
LDR       R12, =0x20007080
MOV       R1, #0
LDR       R6, =9600

search_next_idat
    CMP     R0, R9                ; 메모리 범위 검사
    BGT     start_counting
    CMP     R1, R6                ; 최대 픽셀 도달
    BGE     start_counting

; IDAT 시그니처 검색
LDRB      R2, [R0], #1
CMP       R2, #'I'
BNE       search_next_idat
LDRB      R2, [R0], #1
CMP       R2, #'D'
BNE       search_next_idat
LDRB      R2, [R0], #1
CMP       R2, #'A'
BNE       search_next_idat
LDRB      R2, [R0], #1
CMP       R2, #'T'
BNE       search_next_idat

; 'IDAT' 시그니처 발견, 청크 길이(4 바이트) 읽기 (시그니처 4 바이트 앞)
SUB       R10, R0, #8            ; R0는 'T' 뒤 1 바이트, IDAT 앞 4 바이트는 길이
LDRB      R2, [R10], #1          ; byte 0 (LSB)
LDRB      R3, [R10], #1          ; byte 1
LDRB      R4, [R10], #1          ; byte 2
LDRB      R5, [R10], #1          ; byte 3 (MSB)
; Little-Endian 변환
MOV       R11, R5, LSL #24
ORR       R11, R11, R4, LSL #16
ORR       R11, R11, R3, LSL #8
ORR       R11, R11, R2

; 데이터 시작 주소는 IDAT 시그니처 바로 뒤 (R0)
MOV       R10, R0
ADD       R11, R10, R11          ; 데이터 끝 주소

store_rgb
    CMP     R10, R11             ; 청크 데이터 끝 확인

```

```

BGE    next_idat
CMP    R1, R6                ; 최대 픽셀 확인
BGE    start_counting

; RGBA 4 바이트 처리 (비정렬 대응)
LDRB   R2, [R10], #1        ; R
LDRB   R3, [R10], #1        ; G
LDRB   R4, [R10], #1        ; B
LDRB   R5, [R10], #1        ; A (저장 안 함)

; RGB 만 저장 (RGBRGB... 인터리브)
STRB   R2, [R8], #1         ; R 저장
STRB   R3, [R8], #1         ; G 저장
STRB   R4, [R8], #1         ; B 저장

ADD    R1, R1, #1
B      store_rgb

next_idat
; 다음 IDAT 검색 (IDAT 데이터 끝 + CRC 4 바이트)
ADD    R0, R11, #4
B      search_next_idat

start_counting
; 저장된 데이터에서 Red 값 카운팅
LDR    R8, =0x20000000      ; RGB 저장 시작 주소
MOV    R1, #0                ; 루프 카운터
MOV    R2, #0                ; Red 카운터

count_loop
CMP    R1, R6                ; 9600 픽셀 확인
BGE    final_store
; 3 바이트 단위로 R 값만 읽기
LDRB   R3, [R8], #3          ; R 값만 읽음 (3 바이트 단위, 정렬 보장)
CMP    R3, #128
ADDGE  R2, R2, #1            ; 카운트 증가
ADD    R1, R1, #1
B      count_loop

final_store
STR    R2, [R12]             ; 최종 결과 저장
BX     LR                    ; 종료

LTORG
ENDP
END

```

## ③ Relocation 2 – Alpha 제외, R, G, B 각각 따로 저장

```

AREA RESET, CODE, READONLY
    ENTRY

main PROC
    ; 초기화 - 레지스터 사용 최적화
    LDR    R0, =0x40000000    ; 소스 데이터 시작 주소 (ROM2)
    LDR    R8, =0x4003FFFF    ; ROM2 끝 주소 (256KB)
    LDR    R1, =0x20000000    ; R 배열 시작 주소 (9600 바이트)
    LDR    R2, =0x20002580    ; G 배열 시작 주소 (R 배열 + 9600)
    LDR    R3, =0x20004B00    ; B 배열 시작 주소 (G 배열 + 9600)
    LDR    R12, =0x20007080    ; 결과 저장 주소 (B 배열 + 9600)
    MOV    R4, #9600          ; 최대 픽셀 수
    MOV    R9, #0             ; Red 카운터 (>= 128)
    MOV    R13, #0            ; 픽셀 카운터

search_next_idat
    CMP    R0, R8              ; 메모리 범위 검사
    BGT    store_result
    CMP    R13, R4              ; 최대 픽셀 도달
    BGE    store_result

    ; IDAT 시그니처 검색
    LDRB   R5, [R0]
    CMP    R5, #'I'
    BNE    skip_byte
    LDRB   R6, [R0, #1]
    CMP    R6, #'D'
    BNE    skip_byte
    LDRB   R7, [R0, #2]
    CMP    R7, #'A'
    BNE    skip_byte
    LDRB   R14, [R0, #3]
    CMP    R14, #'T'
    BNE    skip_byte

    ; 청크 길이 계산 (IDAT 시그니처 4 바이트 앞)
    SUB    R10, R0, #4          ; 청크 길이 주소
    LDRB   R5, [R10]            ; byte0
    LDRB   R6, [R10, #1]        ; byte1
    LDRB   R7, [R10, #2]        ; byte2
    LDRB   R14, [R10, #3]       ; byte3
    MOV    R11, R14, LSL #24
    ORR    R11, R11, R7, LSL #16
    ORR    R11, R11, R6, LSL #8
    ORR    R11, R11, R5

    ADD    R10, R0, #4          ; 데이터 시작 주소 = IDAT 시그니처 다음
    ADD    R11, R10, R11        ; 데이터 끝 주소 = 시작 + 청크 길이

process_idat_pixels_loop

```

```

CMP      R10, R11          ; 체크 데이터 끝 확인
BGE      next_idat_search
CMP      R13, R4           ; 최대 픽셀 확인
BGE      store_result

; RGBA 4 바이트 읽기
LDRB     R5, [R10], #1     ; R
LDRB     R6, [R10], #1     ; G
LDRB     R7, [R10], #1     ; B
ADD      R10, R10, #1      ; A 스킵

; R, G, B 저장
STRB     R5, [R1], #1      ; R 배열
STRB     R6, [R2], #1      ; G 배열
STRB     R7, [R3], #1      ; B 배열

; Red 값 검증
CMP      R5, #128
ADDGE    R9, R9, #1        ; 카운트 증가

ADD      R13, R13, #1      ; 픽셀 카운터 증가
B        process_idat_pixels_loop

next_idat_search
ADD      R0, R11, #4       ; 다음 IDAT 검색 (CRC 4 바이트 스킵)
B        search_next_idat

skip_byte
ADD      R0, R0, #1        ; 1 바이트 이동
B        search_next_idat

store_result
STR      R9, [R12]         ; 결과 저장
BX       LR

LTORG
ENDP

AREA DATA, DATA, READWRITE

R_array  SPACE  9600       ; 0x20000000 ~ 0x2000257F
G_array  SPACE  9600       ; 0x20002580 ~ 0x20004AFF
B_array  SPACE  9600       ; 0x20004B00 ~ 0x2000707F
result   SPACE  4          ; 0x20007080

END

```

## ④ 명령어 변경 기반 ARM code 최적화

```

AREA RESET, CODE, READONLY
    ENTRY

main PROC
    LDR    R0, =0x40000000    ; 소스 데이터 시작 주소 (ROM2)
    LDR    R8, =0x4003FFFF    ; ROM2 끝 주소 (256KB)
    LDR    R1, =0x20000000    ; R 배열 시작 주소
    LDR    R2, =0x20002580    ; G 배열 시작 주소
    LDR    R3, =0x20004B00    ; B 배열 시작 주소
    LDR    R12, =0x20007080   ; 결과 저장 주소
    MOV    R4, #9600          ; 최대 픽셀 수
    MOV    R9, #0             ; Red 카운터
    MOV    R13, #0            ; 픽셀 카운터

search_next_idat
    CMP    R0, R8
    BGT    store_result
    CMP    R13, R4
    BGE    store_result

    ; IDAT 시그니처 검색
    LDRB   R5, [R0]
    CMP    R5, #'I'
    BNE    skip_byte
    LDRB   R6, [R0, #1]
    CMP    R6, #'D'
    BNE    skip_byte
    LDRB   R7, [R0, #2]
    CMP    R7, #'A'
    BNE    skip_byte
    LDRB   R14, [R0, #3]
    CMP    R14, #'T'
    BNE    skip_byte

    ; 청크 길이 계산
    SUB    R10, R0, #4
    LDRB   R5, [R10]
    LDRB   R6, [R10, #1]
    LDRB   R7, [R10, #2]
    LDRB   R14, [R10, #3]
    MOV    R11, R14, LSL #24
    ORR    R11, R11, R7, LSL #16
    ORR    R11, R11, R6, LSL #8
    ORR    R11, R11, R5

    ADD    R10, R0, #4        ; 데이터 시작 주소
    ADD    R11, R10, R11      ; 데이터 끝 주소

process_idat_pixels_loop
    CMP    R10, R11

```

```
BGE    next_idat_search
CMP    R13, R4
BGE    store_result

; RGBA 4 바이트 한 번에 읽기
LDR    R5, [R10], #4      ; RGBA 4 바이트 읽기

; R, G, B 추출 및 저장
AND    R6, R5, #0xFF      ; R
AND    R7, R5, #0xFF00    ; G
MOV    R7, R7, LSR #8
AND    R14, R5, #0xFF0000 ; B
MOV    R14, R14, LSR #16

STRB   R6, [R1], #1
STRB   R7, [R2], #1
STRB   R14, [R3], #1

; Red 값 카운트
CMP    R6, #128
ADDGE  R9, R9, #1

ADD    R13, R13, #1
B      process_idat_pixels_loop

next_idat_search
ADD    R0, R11, #4
B      search_next_idat

skip_byte
ADD    R0, R0, #1
B      search_next_idat

store_result
STR    R9, [R12]
BX     LR

LTORG
ENDP

AREA DATA, DATA, READWRITE

R_array    SPACE    9600
G_array    SPACE    9600
B_array    SPACE    9600
result     SPACE    4

END
```

## 9-2. Function #2

## ① 기본 코드

```

AREA    RESET, CODE, READONLY
ENTRY
EXPORT  SeparateRGBA_Pack

SeparateRGBA_Pack
    LDR    R0, =0x40000000    ; PNG 이미지 시작 주소
    LDR    R1, =0x4003FFFF    ; PNG 이미지 끝 주소
    LDR    R2, =9600          ; 처리할 픽셀 수
    MOV    R3, #0             ; 픽셀 카운터

    LDR    R4, =0x20002000    ; RGBA 연속 저장 주소

FindIDAT
    CMP    R0, R1
    BGT    EndProgram

    LDRB   R5, [R0]
    CMP    R5, #'I'
    BNE    Skip
    LDRB   R5, [R0, #1]
    CMP    R5, #'D'
    BNE    Skip
    LDRB   R5, [R0, #2]
    CMP    R5, #'A'
    BNE    Skip
    LDRB   R5, [R0, #3]
    CMP    R5, #'T'
    BNE    Skip

    ADD    R0, R0, #8          ; IDAT + CRC 4 바이트 건너뛰기

ProcessLoop
    CMP    R3, R2
    BGE    EndProgram

    LDRB   R5, [R0]            ; R
    LDRB   R6, [R0, #1]       ; G
    LDRB   R7, [R0, #2]       ; B
    LDRB   R8, [R0, #3]       ; A

    RSB    R5, R5, #255        ; R 반전
    RSB    R6, R6, #255        ; G 반전
    RSB    R7, R7, #255        ; B 반전

    STRB   R5, [R4], #1        ; R 저장
    STRB   R6, [R4], #1        ; G 저장
    STRB   R7, [R4], #1        ; B 저장
    STRB   R8, [R4], #1        ; A 저장

```



```
    ADD    R0, R0, #4
    ADD    R3, R3, #1
    B      ProcessLoop

Skip
    ADD    R0, R0, #1
    B      FindIDAT

EndProgram
    B      EndProgram

    END
```

## ② Relocation 1 – Alpha 제외하고 저장

```

AREA    RESET, CODE, READONLY
ENTRY
EXPORT  InterleavedRGB

InterleavedRGB
    LDR    R0, =0x40000000      ; 입력 이미지 시작 주소
    LDR    R1, =0x4003FFFF      ; 이미지 끝 주소
    LDR    R2, =9600            ; 총 픽셀 수
    MOV    R3, #0              ; 픽셀 카운터

    LDR    R10, =0x20001800     ; RGB 인터리브 저장 시작 주소

FindIDAT2
    CMP    R0, R1
    BGT    EndProgram

    LDRB   R7, [R0]
    CMP    R7, #'I'
    BNE    Skip2
    LDRB   R7, [R0, #1]
    CMP    R7, #'D'
    BNE    Skip2
    LDRB   R7, [R0, #2]
    CMP    R7, #'A'
    BNE    Skip2
    LDRB   R7, [R0, #3]
    CMP    R7, #'T'
    BNE    Skip2

    ADD    R0, R0, #8          ; 'IDAT' + CRC 건너뛰기

ProcessLoop2
    CMP    R3, R2
    BGE    EndProgram

    LDRB   R7, [R0]            ; R
    LDRB   R8, [R0, #1]        ; G
    LDRB   R9, [R0, #2]        ; B

    RSB    R7, R7, #255        ; 반전: 255 - R
    RSB    R8, R8, #255        ; 반전: 255 - G
    RSB    R9, R9, #255        ; 반전: 255 - B

    STRB   R7, [R10], #1       ; R 저장
    STRB   R8, [R10], #1       ; G 저장
    STRB   R9, [R10], #1       ; B 저장

    ADD    R0, R0, #4          ; 다음 픽셀 (4 바이트)
    ADD    R3, R3, #1          ; 픽셀 카운터 증가
    B      ProcessLoop2

```

```
Skip2      ADD    R0, R0, #1
           B      FindIDAT2

EndProgram
           B      EndProgram      ; 무한 루프

           END
```

## ③ Relocation 2 – Alpha 제외, R, G, B 각각 따로 저장

```

AREA    RESET, CODE, READONLY
        ENTRY
        EXPORT SeparateRGB

SeparateRGB
    LDR    R0, =0x40000000    ; 입력 이미지 시작 주소
    LDR    R1, =0x4003FFFF    ; 끝 주소
    LDR    R2, =9600          ; 총 픽셀 수
    MOV    R3, #0             ; 카운터

    LDR    R4, =0x20000000    ; R 저장 주소
    LDR    R5, =0x20000800    ; G 저장 주소
    LDR    R6, =0x20001000    ; B 저장 주소

FindIDAT
    CMP    R0, R1
    BGT    EndProgram

    LDRB   R7, [R0]
    CMP    R7, #'I'
    BNE    Skip
    LDRB   R7, [R0, #1]
    CMP    R7, #'D'
    BNE    Skip
    LDRB   R7, [R0, #2]
    CMP    R7, #'A'
    BNE    Skip
    LDRB   R7, [R0, #3]
    CMP    R7, #'T'
    BNE    Skip

    ADD    R0, R0, #8          ; IDAT + CRC 건너뛰기

ProcessLoop
    CMP    R3, R2
    BGE    EndProgram

    LDRB   R7, [R0]            ; R
    LDRB   R8, [R0, #1]        ; G
    LDRB   R9, [R0, #2]        ; B

    RSB    R7, R7, #255
    RSB    R8, R8, #255
    RSB    R9, R9, #255

    STRB   R7, [R4], #1
    STRB   R8, [R5], #1
    STRB   R9, [R6], #1

    ADD    R0, R0, #4
    ADD    R3, R3, #1

```

```
    B      ProcessLoop
Skip
    ADD    R0, R0, #1
    B      FindIDAT
EndProgram
    B      EndProgram
    END
```

## ④ 명령어 변경 기반 ARM code 최적화

```

AREA    RESET, CODE, READONLY
ENTRY
EXPORT  OptimizedSeparateRGB

OptimizedSeparateRGB
    LDR    R0, =0x40000000      ; 입력 이미지 시작 주소
    LDR    R1, =0x4003FFFF      ; 이미지 끝 주소
    LDR    R2, =9600            ; 총 픽셀 수
    MOV    R3, #0              ; 픽셀 카운터

    LDR    R4, =0x20000000      ; R 저장 주소
    LDR    R5, =0x20000800      ; G 저장 주소
    LDR    R6, =0x20001000      ; B 저장 주소

FindIDAT
    CMP    R0, R1
    BGT    EndProgram

    LDRB    R7, [R0]
    CMP    R7, #'I'
    BNE    SkipByte
    LDRB    R7, [R0, #1]
    CMP    R7, #'D'
    BNE    SkipByte
    LDRB    R7, [R0, #2]
    CMP    R7, #'A'
    BNE    SkipByte
    LDRB    R7, [R0, #3]
    CMP    R7, #'T'
    BNE    SkipByte

    ADD    R0, R0, #8          ; 'IDAT' + CRC 4 바이트 건너뛰기

ProcessLoop
    CMP    R3, R2
    BGE    EndProgram

    ; R 채널
    LDRB    R7, [R0]
    RSB    R8, R7, #255
    STRB    R8, [R4], #1

    ; G 채널
    LDRB    R7, [R0, #1]
    RSB    R8, R7, #255
    STRB    R8, [R5], #1

    ; B 채널
    LDRB    R7, [R0, #2]
    RSB    R8, R7, #255

```

```
STRB    R8, [R6], #1

ADD     R0, R0, #4           ; 다음 픽셀 (4 바이트)
ADD     R3, R3, #1           ; 픽셀 카운터 증가
B       ProcessLoop

SkipByte
ADD     R0, R0, #1
B       FindIDAT

EndProgram
B       EndProgram

END
```

### 9-3. Function #3

#### ① 기본 코드

```

AREA    RESET, CODE, READONLY
ENTRY

main    PROC
    BL    extract_rgba_data      ; 1 단계: RGBA 전체 저장
    BL    calculate_grayscale_rgba ; 2 단계: 복사본에서 계산
    BX    LR

; 1 단계: RGBA 4 바이트씩 0x40002000 에 연속 저장 (9600 픽셀 → 38400 바이트)
extract_rgba_data PROC
    LDR    R0, =0x40000000      ; 이미지 시작 주소
    LDR    R8, =0x40002000      ; RGBA 저장 주소
    LDR    R9, =0x4000FFFF      ; 이미지 끝 주소 한계
    MOV    R1, #0               ; 픽셀 카운터
    LDR    R6, =9600            ; 최대 9,600 픽셀

search_next_idat_rgba
    CMP    R0, R9
    BGT    extract_rgba_done

    ; IDAT 청크 찾기
    LDRB   R3, [R0]
    CMP    R3, #'I'
    BNE    skip_byte_rgba
    LDRB   R3, [R0, #1]
    CMP    R3, #'D'
    BNE    skip_byte_rgba
    LDRB   R3, [R0, #2]
    CMP    R3, #'A'
    BNE    skip_byte_rgba
    LDRB   R3, [R0, #3]
    CMP    R3, #'T'
    BNE    skip_byte_rgba

    ; IDAT 청크 크기 읽기 (Little-endian)
    SUB    R7, R0, #4
    LDRB   R3, [R7]
    LDRB   R4, [R7, #1]
    LDRB   R5, [R7, #2]
    LDRB   R12, [R7, #3]
    MOV    R11, R12, LSL #24
    ORR    R11, R11, R5, LSL #16
    ORR    R11, R11, R4, LSL #8
    ORR    R11, R11, R3

    ADD    R10, R0, #4          ; IDAT 데이터 시작
    ADD    R11, R10, R11        ; IDAT 데이터 끝

```



```

extract_rgba_pixels
    CMP     R1, R6                ; 9600 개 초과 여부
    BGE     extract_rgba_done

    CMP     R10, R11              ; IDAT 청크 끝
    BGE     next_idat_search_rgba

    ; RGBA 4 바이트 연속 저장
    LDRB    R3, [R10, #0]
    STRB    R3, [R8], #1
    LDRB    R4, [R10, #1]
    STRB    R4, [R8], #1
    LDRB    R5, [R10, #2]
    STRB    R5, [R8], #1
    LDRB    R12, [R10, #3]
    STRB    R12, [R8], #1

    ADD     R10, R10, #4
    ADD     R1, R1, #1
    B       extract_rgba_pixels

next_idat_search_rgba
    ADD     R0, R11, #4
    B       search_next_idat_rgba

skip_byte_rgba
    ADD     R0, R0, #1
    B       search_next_idat_rgba

extract_rgba_done
    BX      LR
    ENDP

; 2 단계: RGBA 데이터(0x40002000)에서 RGB 만 꺼내서 Grayscale 계산
calculate_grayscale_rgba PROC
    LDR     R0, =0x40002000        ; RGBA 데이터 시작 주소
    LDR     R8, =0x40004000        ; Grayscale 결과 저장
    LDR     R6, =9600              ; 최대 9,600 픽셀
    MOV     R1, #0

process_rgba_pixels
    CMP     R1, R6
    BGE     calculate_done_rgba

    LDRB    R3, [R0], #1           ; R
    LDRB    R4, [R0], #1           ; G
    LDRB    R5, [R0], #1           ; B
    ADD     R0, R0, #1             ; A 는 그냥 스킵

    ; Grayscale 공식: 3*R + 6*G + B
    MOV     R7, R3
    ADD     R7, R7, R7, LSL #1      ; 3*R

```

```
MOV    R12, R4
ADD    R12, R12, R12, LSL #1 ; 3*G
ADD    R12, R12, R12      ; 6*G
ADD    R7, R7, R12
ADD    R7, R7, R5

STRH   R7, [R8], #2      ; 16 비트로 저장

ADD    R1, R1, #1
B      process_rgba_pixels

calculate_done_rgba
BX     LR
ENDP

LTORG
END
```

## ② Relocation 1 – Alpha 제외하고 저장

```

AREA    RESET, CODE, READONLY
ENTRY

main    PROC
    BL    extract_rgb_data      ; 1 단계: RGBA 에서 RGB 만 추출
    BL    calculate_grayscale   ; 2 단계: RGB 로 Grayscale 계산
    BX    LR

; 1 단계: RGBA 에서 RGB 만 추출하여 0x30000000 에 저장
extract_rgb_data PROC
    LDR    R0, =0x40000000      ; 이미지 시작 주소
    LDR    R8, =0x40002000      ; RGB 저장 주소
    LDR    R9, =0x4000FFFF      ; 이미지 끝 주소 한계
    MOV    R1, #0               ; 픽셀 카운터
    LDR    R6, =9600            ; 최대 9,600 픽셀

search_next_idat_step1
    CMP    R0, R9
    BGT    extract_done

    ; IDAT 청크 찾기
    LDRB   R3, [R0]
    CMP    R3, #'I'
    BNE    skip_byte_step1
    LDRB   R3, [R0, #1]
    CMP    R3, #'D'
    BNE    skip_byte_step1
    LDRB   R3, [R0, #2]
    CMP    R3, #'A'
    BNE    skip_byte_step1
    LDRB   R3, [R0, #3]
    CMP    R3, #'T'
    BNE    skip_byte_step1

    ; IDAT 청크 크기 읽기 (Little-endian)
    SUB    R7, R0, #4
    LDRB   R3, [R7]
    LDRB   R4, [R7, #1]
    LDRB   R5, [R7, #2]
    LDRB   R12, [R7, #3]
    MOV    R11, R12, LSL #24
    ORR    R11, R11, R5, LSL #16
    ORR    R11, R11, R4, LSL #8
    ORR    R11, R11, R3

    ADD    R10, R0, #4          ; IDAT 데이터 시작 (타입 바로 뒤)
    ADD    R11, R10, R11        ; IDAT 데이터 끝

extract_rgb_pixels

```

```

    CMP    R1, R6                ; 9600 개 초과 여부
    BGE    extract_done

    CMP    R10, R11              ; IDAT 청크 끝
    BGE    next_idat_search_step1

    ; RGBA 에서 RGB 만 추출하여 연속 저장
    LDRB   R3, [R10, #0]         ; Red
    LDRB   R4, [R10, #1]         ; Green
    LDRB   R5, [R10, #2]         ; Blue
    ; Alpha [R10, #3]는 무시

    STRB   R3, [R8], #1          ; R 저장
    STRB   R4, [R8], #1          ; G 저장
    STRB   R5, [R8], #1          ; B 저장

    ADD    R10, R10, #4          ; 다음 RGBA 픽셀
    ADD    R1, R1, #1            ; 픽셀 카운터 증가
    B      extract_rgb_pixels

next_idat_search_step1
    ADD    R0, R11, #4           ; CRC 4byte 건너뛰고 다음 청크로
    B      search_next_idat_step1

skip_byte_step1
    ADD    R0, R0, #1
    B      search_next_idat_step1

extract_done
    BX     LR
    ENDP

; 2 단계: RGB 데이터로 Grayscale 계산 (0x30000000 → 0x20000000)
calculate_grayscale PROC
    LDR    R0, =0x40002000        ; RGB 데이터 시작 주소
    LDR    R8, =0x40004000        ; Grayscale 결과 저장 주소
    LDR    R6, =9600              ; 최대 9,600 픽셀
    MOV    R1, #0                 ; 픽셀 카운터

process_rgb_pixels
    CMP    R1, R6                ; 9600 개 처리했는지 확인
    BGE    calculate_done

    LDRB   R3, [R0], #1          ; Red
    LDRB   R4, [R0], #1          ; Green
    LDRB   R5, [R0], #1          ; Blue

    ; Grayscale 공식: 3*R + 6*G + B
    MOV    R7, R3
    ADD    R7, R7, R7, LSL #1     ; R7 = R + 2*R = 3*R
    MOV    R12, R4
    ADD    R12, R12, R12, LSL #1 ; R12 = G + 2*G = 3*G

```

```
    ADD    R12, R12, R12      ; 3*G + 3*G = 6*G
    ADD    R7, R7, R12
    ADD    R7, R7, R5

    STRH    R7, [R8], #2      ; 16 비트로 저장하고 주소 증가

    ADD    R1, R1, #1
    B       process_rgb_pixels

calculate_done
    BX      LR
    ENDP

    LTORG
    END
```

## ③ Relocation 2 – Alpha 제외, R, G, B 각각 따로 저장

```

AREA    RESET, CODE, READONLY
ENTRY

main    PROC
    BL    extract_separate_rgb    ; 1 단계: RGB 각각 분리 저장
    BL    calculate_from_separate ; 2 단계: 분리된 RGB 로 계산
    BX    LR

; 1 단계: RGBA 에서 R, G, B 를 각각 분리해서 저장
extract_separate_rgb PROC
    LDR    R0, =0x40000000    ; 이미지 시작 주소
    LDR    R2, =0x40001000    ; R 채널 저장 시작
    LDR    R3, =0x40004000    ; G 채널 저장 시작
    LDR    R4, =0x40007000    ; B 채널 저장 시작
    LDR    R9, =0x4000FFFF    ; 이미지 끝 주소 한계
    MOV    R1, #0              ; 픽셀 카운터
    LDR    R6, =9600           ; 최대 9,600 픽셀

search_next_idat_step1
    CMP    R0, R9
    BGT    extract_separate_done

; IDAT 청크 찾기
    LDRB    R5, [R0]
    CMP    R5, #'I'
    BNE    skip_byte_step1
    LDRB    R5, [R0, #1]
    CMP    R5, #'D'
    BNE    skip_byte_step1
    LDRB    R5, [R0, #2]
    CMP    R5, #'A'
    BNE    skip_byte_step1
    LDRB    R5, [R0, #3]
    CMP    R5, #'T'
    BNE    skip_byte_step1

; IDAT 청크 크기 읽기 (Little-endian)
    SUB    R7, R0, #4
    LDRB    R5, [R7]
    LDRB    R10, [R7, #1]
    LDRB    R11, [R7, #2]
    LDRB    R12, [R7, #3]
    MOV    R12, R12, LSL #24
    ORR    R12, R12, R11, LSL #16
    ORR    R12, R12, R10, LSL #8
    ORR    R12, R12, R5

    ADD    R10, R0, #4    ; IDAT 데이터 시작
    ADD    R11, R10, R12  ; IDAT 데이터 끝

```

```

extract_separate_pixels
    CMP    R1, R6
    BGE    extract_separate_done
    CMP    R10, R11
    BGE    next_idat_search_step1

    ; RGB 각각 분리해서 저장 (R12 는 B 에만 사용)
    LDRB    R5, [R10, #0]        ; Red
    STRB    R5, [R2, R1]        ; R 저장

    LDRB    R7, [R10, #1]        ; Green
    STRB    R7, [R3, R1]        ; G 저장

    LDRB    R12, [R10, #2]       ; Blue
    STRB    R12, [R4, R1]       ; B 저장

    ADD    R10, R10, #4          ; 다음 RGBA 픽셀
    ADD    R1, R1, #1
    B      extract_separate_pixels

next_idat_search_step1
    ADD    R0, R11, #4           ; CRC 4byte 건너뛰고 다음 청크로
    B      search_next_idat_step1

skip_byte_step1
    ADD    R0, R0, #1
    B      search_next_idat_step1

extract_separate_done
    BX     LR
    ENDP

; 2 단계: 분리된 R, G, B 배열로부터 Grayscale 계산
calculate_from_separate PROC
    LDR    R2, =0x40001000        ; R 채널 배열 시작
    LDR    R3, =0x40004000        ; G 채널 배열 시작
    LDR    R4, =0x40007000        ; B 채널 배열 시작
    LDR    R8, =0x4000A000        ; Grayscale 결과 저장
    LDR    R6, =9600              ; 최대 9,600 픽셀
    MOV    R1, #0                 ; 픽셀 카운터

process_separate_pixels
    CMP    R1, R6
    BGE    calculate_separate_done

    ; 각 채널 배열에서 값 읽기
    LDRB    R5, [R2, R1]          ; R 값
    LDRB    R7, [R3, R1]          ; G 값
    LDRB    R12, [R4, R1]         ; B 값

    ; Grayscale 공식: 3*R + 6*G + B

```

```
ADD    R5, R5, R5, LSL #1    ; 3*R
ADD    R7, R7, R7, LSL #1    ; 3*G
ADD    R7, R7, R7            ; 6*G
ADD    R5, R5, R7            ; 3*R + 6*G
ADD    R5, R5, R12          ; 3*R + 6*G + B

MOV     R10, R1, LSL #1      ; 오프셋 계산 (픽셀 인덱스 × 2)
STRH    R5, [R8, R10]        ; Grayscale 저장

ADD     R1, R1, #1
B       process_separate_pixels

calculate_separate_done
BX      LR
ENDP

LTORG
END
```



## ④ 명령어 변경 기반 ARM code 최적화

```

AREA    RESET, CODE, READONLY
ENTRY

main    PROC
    BL    extract_separate_rgb    ; 1 단계: RGB 각각 분리 저장
    BL    calculate_from_separate ; 2 단계: 분리된 RGB 로 Grayscale 계산
    BX    LR
    ENDP

extract_separate_rgb PROC
    LDR    R0, =0x40000000    ; 이미지 시작 주소
    LDR    R2, =0x40001000    ; R 채널 저장 시작
    LDR    R3, =0x40004000    ; G 채널 저장 시작
    LDR    R4, =0x40007000    ; B 채널 저장 시작
    LDR    R9, =0x4000FFFF    ; 이미지 끝 주소 한계
    MOV    R1, #0             ; 픽셀 카운터
    LDR    R6, =9600           ; 최대 9,600 픽셀

search_next_idat_step1
    CMP    R0, R9
    BGT    extract_separate_done

    ; IDAT 청크 찾기
    LDRB   R5, [R0]
    CMP    R5, #'I'
    BNE    skip_byte_step1
    LDRB   R5, [R0, #1]
    CMP    R5, #'D'
    BNE    skip_byte_step1
    LDRB   R5, [R0, #2]
    CMP    R5, #'A'
    BNE    skip_byte_step1
    LDRB   R5, [R0, #3]
    CMP    R5, #'T'
    BNE    skip_byte_step1

    ; IDAT 청크 길이 읽기
    SUB    R7, R0, #4
    LDRB   R5, [R7]           ; lowest byte
    LDRB   R10, [R7, #1]
    LDRB   R11, [R7, #2]
    LDRB   R12, [R7, #3]     ; highest byte
    MOV    R12, R12, LSL #24
    ORR    R12, R12, R11, LSL #16
    ORR    R12, R12, R10, LSL #8
    ORR    R12, R12, R5      ; R12 = IDAT 데이터 길이

    ADD    R10, R0, #4        ; IDAT 데이터 시작 주소
    ADD    R11, R10, R12      ; IDAT 데이터 끝 주소

```

```

extract_separate_pixels
    CMP     R1, R6
    BGE     extract_separate_done
    CMP     R10, R11
    BGE     next_idat_search_step1

    ; R, G, B 각각 분리 → 저장
    LDRB    R5, [R10, #0]      ; Red
    STRB    R5, [R2, R1]      ; R 채널 저장

    LDRB    R7, [R10, #1]      ; Green
    STRB    R7, [R3, R1]      ; G 채널 저장

    LDRB    R12, [R10, #2]     ; Blue
    STRB    R12, [R4, R1]     ; B 채널 저장

    ADD     R10, R10, #4       ; 다음 RGBA 픽셀 위치
    ADD     R1, R1, #1
    B       extract_separate_pixels

next_idat_search_step1
    ADD     R0, R11, #4        ; CRC(4byte) 건너뛰고 다음 청크로
    B       search_next_idat_step1

skip_byte_step1
    ADD     R0, R0, #1
    B       search_next_idat_step1

extract_separate_done
    BX      LR
    ENDP

calculate_from_separate PROC
    ; 채널별 버퍼 시작 주소 로드
    LDR     R2, =0x40001000    ; R 채널 배열 시작
    LDR     R3, =0x40004000    ; G 채널 배열 시작
    LDR     R4, =0x40007000    ; B 채널 배열 시작
    LDR     R8, =0x4000A000    ; Grayscale 결과 저장 버퍼 시작

    ; 상수 3, 6 을 미리 로드 (루프 내에서는 매번 로드하지 않도록)
    MOV     R9, #3             ; 3×R
    MOV     R10, #6            ; 6×G

    LDR     R6, =9600          ; 최대 9,600 픽셀
    MOV     R1, #0             ; 픽셀 인덱스 (0 부터 시작)

process_separate_pixels
    CMP     R1, R6
    BGE     calculate_separate_done

    ; R/G/B 채널 버퍼에서 1 바이트씩 읽기
    LDRB    R5, [R2, R1]      ; R 값

```

```
LDRB    R7, [R3, R1]      ; G 값
LDRB    R12, [R4, R1]     ; B 값

MLA      R11, R7, R10, R12  ; R11 = (R7 × R10) + R12 → 6*G + B

MLA      R12, R5, R9, R11   ; R12 = (R5 × R9) + R11 → 3*R + (6*G + B)

MOV      R11, R1, LSL #1    ; offset = R1 × 2

; 16-bit Grayscale 저장
STRH     R12, [R8, R11]

ADD      R1, R1, #1
B        process_separate_pixels

calculate_separate_done
BX       LR
ENDP

LTORG
END
```