어셈블리프로그램설계및실습 보고서

Term Project :

Convolution with Floating Point Number

학 과: 컴퓨터공학과

담당교수: 이형근 교수님

실습분반: 목요일 6, 7

학 번: 2019202053

성 명: 신윤석

제 출 일: 2022.10.07(금)

목차

1. Introdution

본 프로젝트는 2022학년도 2학기 어셈블리프로그램설계및실습 과목의 term project로, 이산 합성곱(discrete convolution)을 계산하는 프로그램을 ARM Assembly로 작성하는 프로젝트이다. 합성곱을 위한 data와 filter는 부동 소수점 숫자(Floating Point Number)로 주어지며, filter와 data의 합성곱을 취해 특정 조건을 만족하는 계산결과와 그 index를 메모리에 저장하는 프로그램을 작성한다. 이때, 주어지는 data의 개수는 480개, 범위는 -4 ~ 8사이의 실수이며, filter는 6개의 실수 [1.95, 1.72, -0.431, -1.278, -0.8022, -0.2115]이다. 메모리에 합성곱 결과를 저장하는 조건은 합성곱의 결과가 -4.7이하인 data와 그 index를 메모리 0xF0000000번지부터 저장을 하면 된다.

2. Background

A. Convolution

Convolution, 합성곱은 두 함수 간의 연산이다. 함수 f, g가 적분이 가능할 때, 합성곱은 다음과 같이 정의된다:

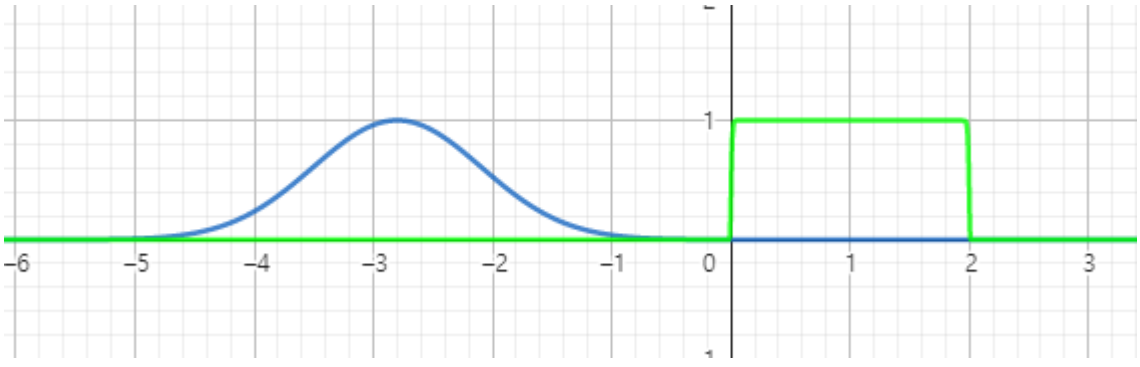
$$(f * g)(t) = f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

이때, 합성곱 연산은 교환법칙이 성립한다는 특징이 있다. $z = t - \tau$로 놓으면 다음이 성립한다:

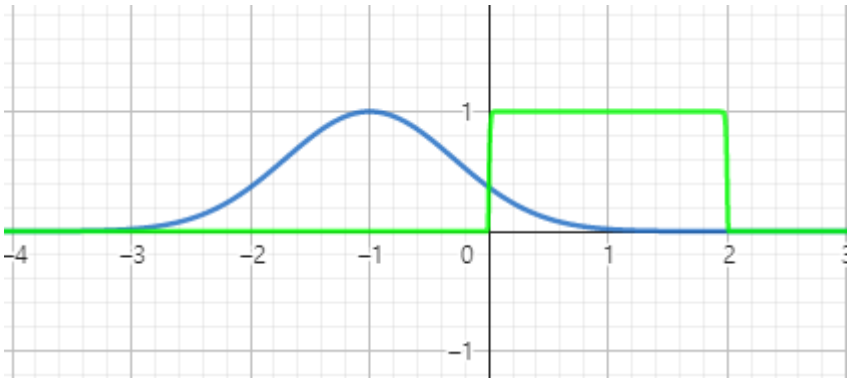
$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{\infty}^{-\infty} f(t-z)g(z)(-dz) = \int_{-\infty}^{\infty} g(z)f(t-z)dz = (g * f)(t)$$

이 성질은 이후 이번 프로젝트를 진행하는데 큰 도움이 된다. 원칙적으로는 data를 뒤집어서 index에 따라 이동시켜야 하지만 convolution연산은 교환법칙이 성립하기 때문에 filter를 뒤집어서 index에 맞춰 이동시키면서 data와 곱셈을 할 수 있기 때문이다.

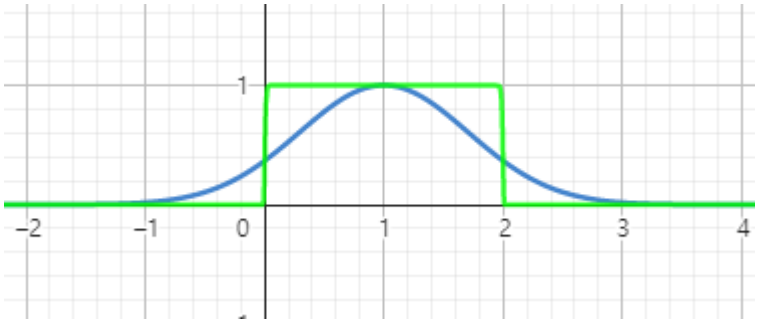
이러한 합성곱을 시각화 해보자.



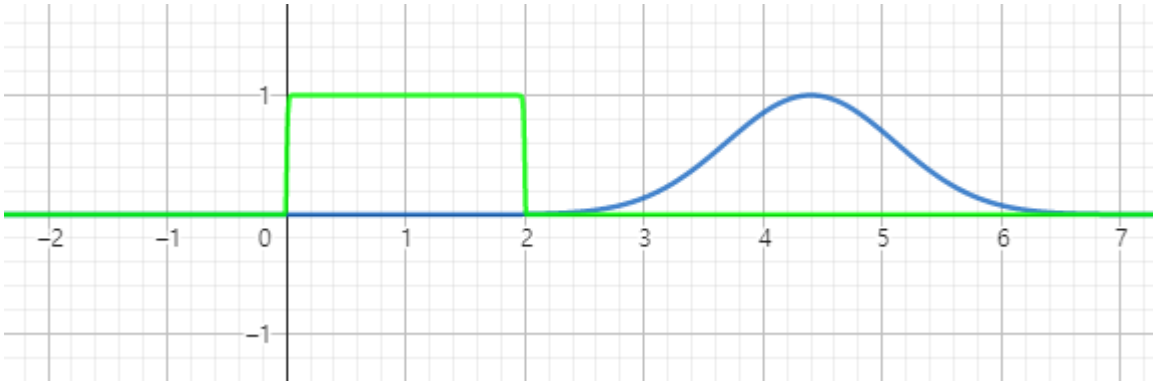
파란색 함수가 f(a-t)이고 초록색 함수가 g(t)일 때, 두 함수의 곱은 어느곳에서나 0이다. 0을 실수 전체에서 적분을 하더라도 결과값은 0이므로 위 상태에서의 두 함수의 합성곱은 0이게 된다.



파란색 함수가 x > 1인 부분에서 0이라 한다면 두 함수를 곱한 결과는 0 < x <1인 범위를 제외하고는 다 0이게 된다. 따라서, 합성 곱을 구하기 위해서는 0 < x < 1인 범위에서만 f(a-t)와 g(t)를 곱한 부분만 적분을 해 주면 된다.



위 두 사진도 마찬가지로 두 함수가 0이 아닌 부분에 대해서만 함수에 곱을 적분하면 합성곱을 얻을 수 있다.



마지막으로 필터가 이동하여 다시 data부분에 0만 남은 경우, 두 함수의 곱은 실수 전체에서 0이

기 때문에 이를 적분한 합성곱 결과도 합성곱을 구하지 않고도 0이라는 사실을 알 수 있다. 결론적으로 합성곱의 정의에서는 실수 전체의 곱을 유도하지만, data가 0이 아닌 구간의 범위가 유계라면 혹은 filter가 0이 아닌 구간의 범위가 유계라면, data와 filter 둘 다 0이 아닌 지점에 대해서만 합성곱이 0이 아니라는 사실을 알 수 있다.

지금까지는 연속함수에 대한 합성곱을 다뤘다. 이산수학에서의 합성곱은 다음과 같다:

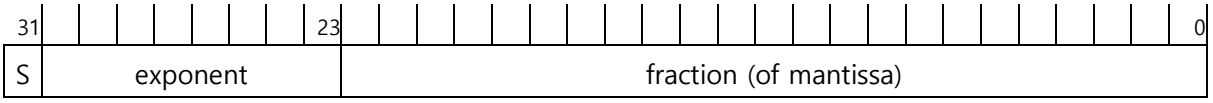
$$y[n] = \sum_{k=0}^{M} h[k] \cdot x[n-k]$$

위 식에서 M은 filter의 개수를 의미한다. filter의 개수가 6개인 경우, k가 0부터 시작하므로 M은 5이다. 특히, 프로그램을 구현하는데 있어 원칙적으로는 data를 순서를 거꾸로 해서 filter와의 합성곱을 구해야 하는데, 교환법칙이 성립함에 따라 방대한 양의 data를 거꾸로 할 필요 없이 filter의 순서를 거꾸로 해서 정방향의 data와의 내적을 통해 구할 수 있다.

B. IEEE 754 Floating Point

컴퓨터는 기본적으로 0과 1만을 표현할 수 있고 해당 표현법으로 2진수를 구현하여 컴퓨터를 사용하게 된다. 0과 자연수는 당연히 2진수로 표현할 수 있다. 음수의 경우는 2의 보수를 적용하여 음수 표현이 가능하다. 실수의 경우는 조금 복잡한데, 일반적으로는 부동(浮動) 소수점 표기 방식을 이용한다. 소수점이 물 위에 둥둥 떠서 미끄러지듯이 자유롭게 움직일 수 있는 표기 방식이다. 1보다 작은 수, 소수를 표현하는 방식은 과학적 표기법에서 시작한다. 과학적 표기법은 매우 큰 수나 매우 작은 수를 표기하는데 유리하다. 예를 들어 지구와 태양까지의 거리는 약 150,000,000km이다. 이렇게 쓰면 일, 십, 백, 천, 만... 세어가면서 읽는데 오래 걸리고 다른 수와 계산하는데도 시간이 조금 걸린다. 과학적 표기법으로 표기를 하면 1.5×10^8km로 쓸 수 있다. 얼핏 보면 계산이 조금 더 불편해 보일 수 있지만 곱 연산을 매우 효율적으로 할 수 있다. "태양에서 지구만큼 떨어진 거리에서 전자를 하나 태양 방향으로 떨어뜨렸을 때 태양 표면에서의 전자의 속력"을 계산한다고 생각해 보자. 그럼 역학적 에너지 보존 법칙에 따라 $\frac{1}{2}mv^2 = mgh$로부터 v를 구할 수 있다. 이때, 필연적으로 매우 작은 전자의 질량과 매우 큰 태양의 거리를 곱해야 하는 경우가 생긴다. 부동 소수점 숫자를 이용하면 매우 쉽게 연산이 가능하다. 그리고 컴퓨터에서도 소수 표현이 가능하게 해준다.

이번 프로젝트에서는 부동 소수점을 표시하기 위한 방식으로 IEEE 754 Floating Point 규격을 사용한다. 해당 규격에서 필요한 정보는 세 파트로 나눠진다. 부호, 지수, 가수(mantissa)이다. IEEE 754규격은 $(-1)^s 2^{e-k} \times mantissa$형태에서 부호인 s와 지수인 e와 가수인 mantissa를 한대 묶어서 표현을 한다. 32bit인 경우엔 k=127, s는 1bit, e는 8bit, mantissa는 나머지 23bit를 할당하여 만든다. mantissa는 2진수에서 정수 부분은 무조건 1이므로 정수부분 1을 생략하고 소수점 부분 23bit만을 적는다. 정리하자면 다음 그림과 같다:

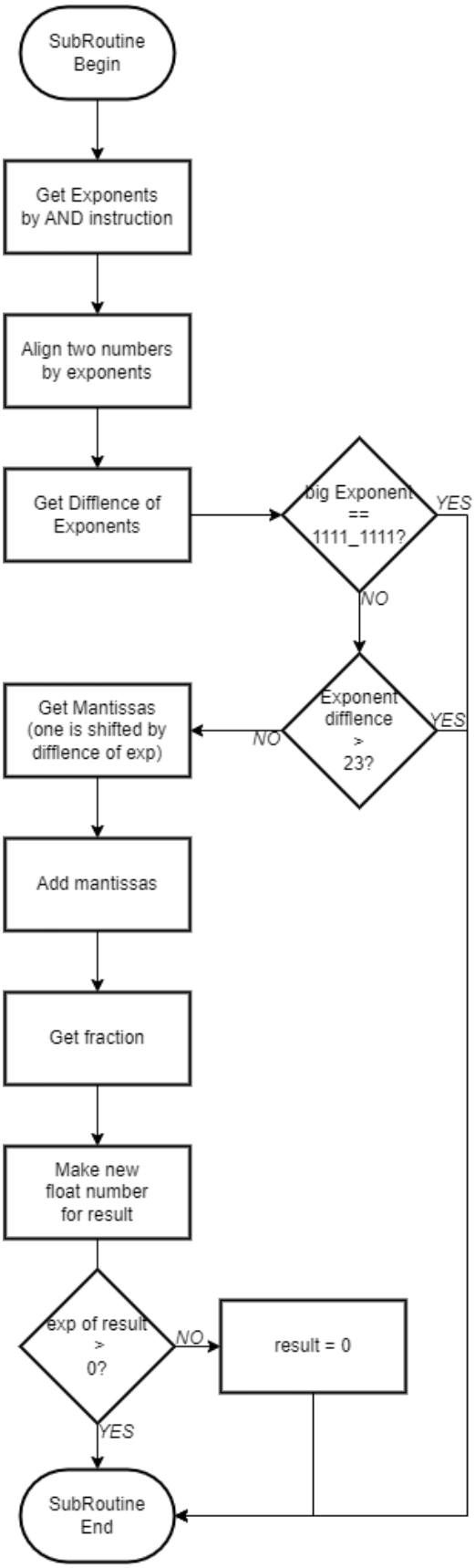


그런데, 이런 표기법을 하는데 있어서 근사치를 나타내야 하는 경우도 있다. 대표적으로는 소수가 1/2의 거듭제곱의 합으로 표현할 수 없는 경우이다. 0.1이 그런 경우이다. 이런 경우는 필연적으로 반올림이 일어나게 된다. 반올림도 4가지 방법이 있다. 우선 반올림하려는 수의 LSB를 L이라 하자. 그리고 그 수보다 더 작은 수를 G, 그 이후의 수 중에 1이 하나라도 있다면 1이 되는 수를 S라고 하자. 그렇다면 반올림을 다음과 같이 할 수 있다:

1. 가장 가까운 숫자로 반올림하는 방법 : +(L&G)|(G&S)를 한다.
2. 0의 방향으로 반올림하는 방법 : LSB이하의 숫자를 무시한다. (+0을 한다)
3. 양의 방향으로 올림 하는 방법 : + !sign & (G|S)를 한다.
4. 음의 방향으로 내림 하는 방법 : + sign & (G|S)를 한다.

이 외의 부동 소수점 숫자의 덧셈과 곱셈은 3. algorithm파트에서 다루도록 하겠다.

3. Algorithm



A. Float Addition

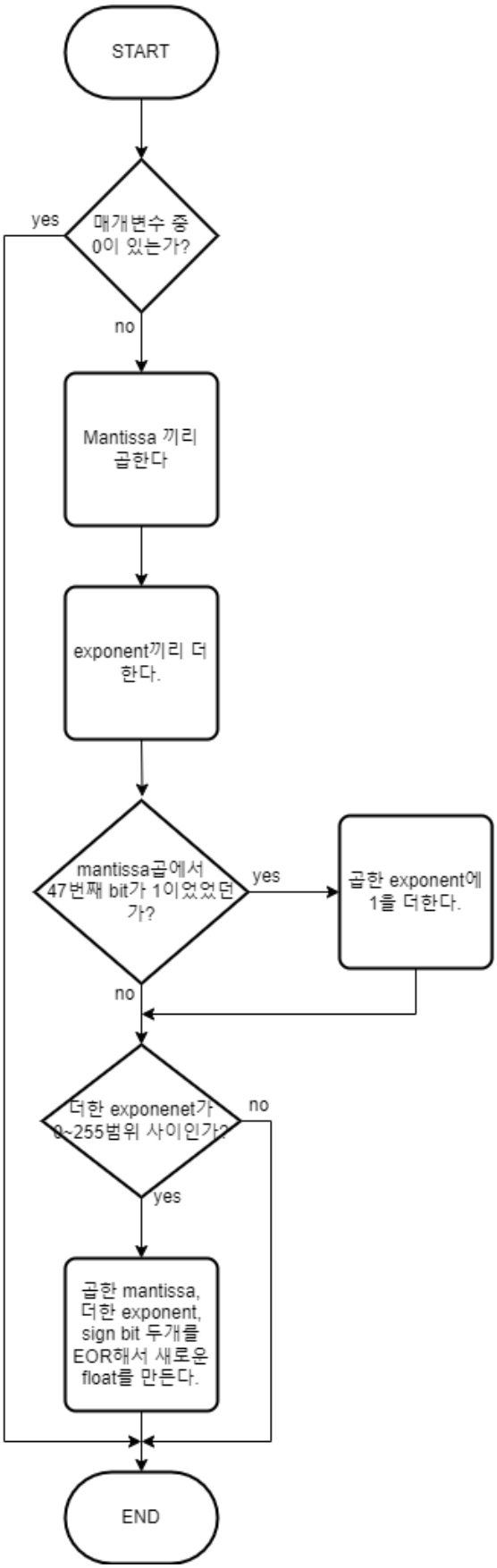
float덧셈의 대략적인 진행 순서는 왼쪽 순서도와 같다. 대략적인 계산 순서가 exponent에 맞춰 mantissa를 shift하고 그 이후에 mantissa를 계산을 하고, 계산 결과에 맞춰 exponent를 계산하고 마지막으로 sign bit도 계산하여 계산을 마무리하는 순서로 이루어져 있다.

Argument로 R0와 R1에 들어온 숫자를 우선 Exponent가 더 큰 수를 R0에 놔줘서 이후 계산을 간단하게 하는 것이 포인트이다. 또한 R0가 연산 결과를 반환하는 Register이므로, 무한대의 숫자가 들어온다면 아무런 계산없이 바로 반환을 하여 효율성을 도모할 수 있다.

Add mantissa과정에서는 sign bit에 따라 덧셈과 뺄셈이 결정된다. 특히 뺄셈의 경우 R0가 음수였냐, R1이 음수였냐에 따라 SUB명령어에 들어가야 하는 operand 순서가 달라지고, 뺄셈에 의한 flag를 통해 이 뺄셈 결과가 음수인지 양수인지 파악하기 위해 또 flag setting을 바꿔야 한다. 오른쪽 표와 같은 구문으로 해당 고민을 해결할 수 있었다. 우선 CMP명령어로 숫자 하나의 부호를 판별한 뒤 SUBSNE명령어를 호출하기 전에 숫자 1을 더해준다. 이렇게 하는 이유는 그 다음에 오는 명령어가 SUBSEQ인데 EQ의 실행 조건이 Zero flag set이다. 만약 숫자가 같다면 뺄셈을 했을 때, Zero flag set되기 때문에 1을 더한 것이다. 그리고 R2와 R3의 값은 mantissa를 LSL실행한 상태이기 때문에 7:0 bit는 0임이 보장된 상태이다 따라서 +1,-1연산은 결과에 영향을 끼치지 않는다.

```
CMP R4, #0x80000000
ADD R2, R2, #1
SUBSNE R2, R2, R3
SUB R2, R2, #1
SUBSEQ R2, R3, R2
```

B. Float Multiplication



곱셈을 처리하는 과정은 왼쪽 순서도와 같다. 대략적인 순서를 설명하자면, mantissa끼리의 곱을 먼저 구하고, exponent합을 구하고 마지막에 sign bit를 구한다. 이때, exponent에 의한 예외처리를 mantissa끼리의 곱을 하고 처리를 하게 되는데, 이유가 두가지가 있다. 첫번째 이유는 Register를 R0 ~ R5까지를 가용 Register를 두고 계산하도록 설계를 하였는데 Mantissa를 곱하는 과정에서 UMULL 명령어를 위한 Register가 필요한데, 아무런 작업을 안 한 상태에서만 R5 register가 비기 때문에 우선적으로 mantissa값을 계산하고 이후에 exponent를 계산하였다. 또한 mantissa끼리의 곱을 진행한 결과 중에서 mantissa곱의 최소값이 0x4000 0000 0000(=0x80 0000 * 0x80 0000)이고 mantissa곱의 최대값이 0xFFFF FE00 0001(=0xFF FFFF * 0xFF FFFF)로 bit가 하나가 차이가 난다. 여러가지 연산을 해 본 결과 계산결과가 0x8000 0000 0000 보다 같거나 크면 exponent를 1올려줘야 알맞은 계산 결과가 나왔다. 결론적으로 mantissa 곱셈 또한 exponent를 수정하기 때문에 exponent 범위에 의한 예외처리는 mantissa의 곱이 끝난 이후에 해야 한다.

또한 이번 프로젝트에서 입력으로 들어오는 값의 범위는 -4~8사이의 실수이고, 주어진 필터 또한 -2~2범위 사이로 두 곱의 최댓값은 16, 최솟값은 -16으로 결과 값이 발산을 하지 않는다. 따라서 곱셈 결과가 exponent가 255가 되는 무한이 되는 경우는 예외처리를 따로 하지 않았다.

또한 Mantissa의 곱셈을 하면서 하위 bit에 계속해서 이진수가 나오지만 해당 이진수는 무시했다. 0의 방향으로 반올림을 한 셈이다.

C. Convolution & Overall Program Flow

이번 프로젝트에서 프로그램의 전체적인 흐름은 두 단계로 나눌 수 있다.

1. Convolution구하기
2. 결과가 -4.7이하인 Convolution값과 해당 index를 Memory에 저장하기

이때, 2번 과정을 위해 1번 과정 중간중간 Convolution 값이 나오면 그 값이 -4.7보다 작은지 계산을 하고 그 값이 -4.7보다 작다면 stack에 저장을 한다. stack은 선입선출 구조임으로 stack에서 data를 오름차순으로 출력하고 싶으면 stack에는 data를 내림차순으로 넣어야 한다. 그렇기 때문에 convolution은 index 479부터 5까지 역순으로 계산을 진행한다. 또한 index의 기준은 다음 식의 n과 같다:

$$y[n] = \sum_{k=0}^{5} h[k] \cdot x[n-k]$$

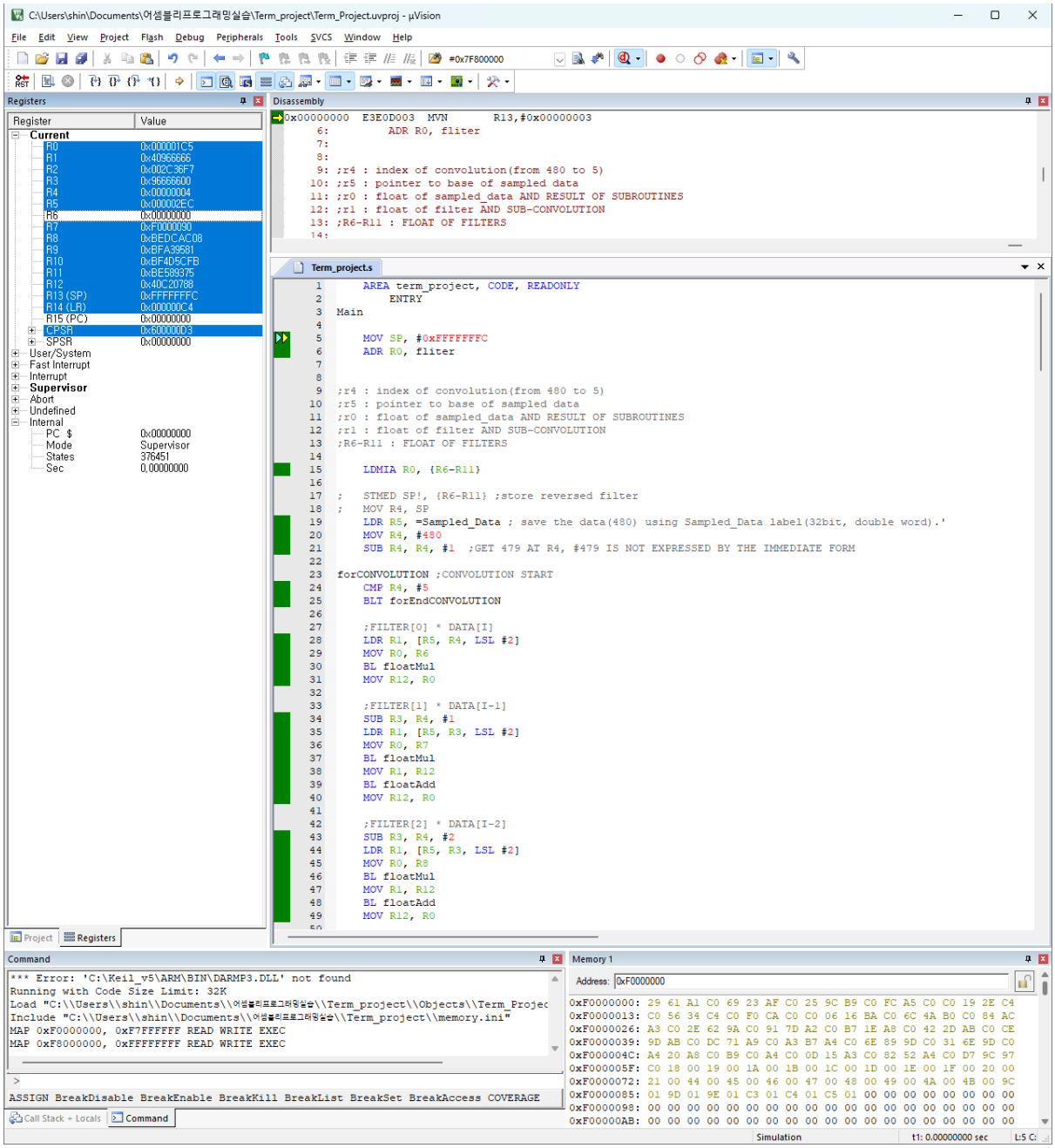
해당 식에서 x는 data, h는 filter를 의미하고 n이 index이다. 또한 data의 확장은 고려하지 않기 때문에 n-k가 0보다 작거나 479보다 큰 경우가 오는 index는 계산하지 않는다. 그렇기 때문에 convolution을 계산하는 index의 범위는 5~479인 것이다.

convolution계산이 끝나면 결과가 -4.7이하인 convolution 값과 해당 index는 교차로 stack안에 들어있을 것이다. stack pointer의 주소를 받아와서 주소를 8씩 올려가며 convolution 값만 우선 저장을 한 뒤에 다시 stack pointer의 주소를 받아와서 이번엔 첫 index 값으로 이동해서 index 값 만을 STRH명령어를 이용해 16bit단위로 저장한다.

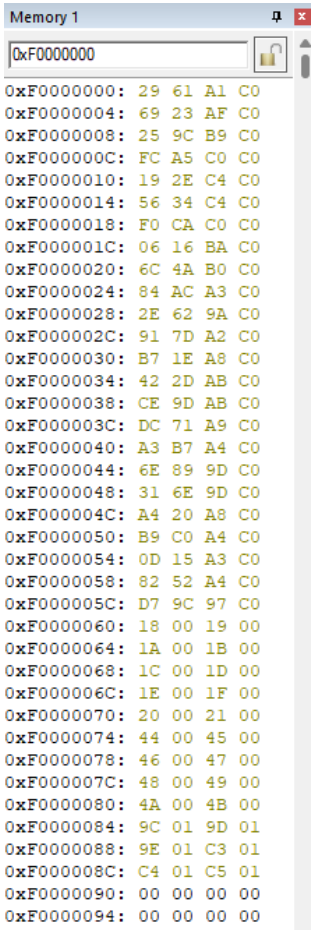
4. Performance & Result

```
Build Output
assembling Term_project.s...
Term_project.s(164): warning: A1608W: MOV pc,<rn> instruction used, but BX <rn> is preferred
Term_project.s(192): warning: A1608W: MOV pc,<rn> instruction used, but BX <rn> is preferred
Term_project.s(245): warning: A1608W: MOV pc,<rn> instruction used, but BX <rn> is preferred
Term_project.s(257): warning: A1608W: MOV pc,<rn> instruction used, but BX <rn> is preferred
linking...
Program Size: Code=2672 RO-data=0 RW-data=0 ZI-data=0
".\Objects\Term_Project.axf" - 0 Error(s), 4 Warning(s).
Build Time Elapsed:  00:00:00
```

code size는 2672가 나왔다. 또한 프로그램 전체 실행 결과는 다음 사진과 같다.



작동 완료시의 state가 376451이 나왔다.



또한 -4.7보다 작은 합성곱 결과는 왼쪽 사진과 같이 메모리 0xF000 0000 부터 저장이 잘 되어 있다. DATA가 0xF000 0000 ~ 0xF000 005C 까지 저장이 되어있는데, 5C를 4로 나누면 10진수로 23이다. 이는 합성곱 결과가 -4.7보다 낮은 값을 가진 data를 24개 찾았다는 의미이다.

5. Consideration

이번 프로젝트의 핵심은 sub-routine 만들기라고 생각한다. float 덧셈과 곱셈을 해주는 sub-routine을 만드는데 가장 긴 시간이 걸렸고 공을 들였기 때문이다. 특히 float 덧셈을 하는 sub-routine은 이전에 과제로도 나왔으므로 범용성에 초점을 두고 exception handling을 처리했다. 그렇기에 이번 프로젝트의 제약사항 이외의 예외사항도 처리하기 위해 state를 더 사용했다는 특징이 있다. 하지만 float 곱셈을 위한 sub-routine은 이번 프로젝트 제약사항에 맞춰 작성되었기 때문에 곱셈 결과가 float의 표현범위보다 절댓값이 큰 수 (exponent가 254를 넘어가는 경우)가 되는 경우를 상정하지 않았고 exception handling을 하지 않는다.

또한 곱셈 sub-routine에서는 반올림을 가장 가까운 값으로 하지 않고 0의 방향으로 하게 되는데 사실 이 부분은 설계를 마친 이후에 해당 내용을 확인했고 수정할까 고민을 했지만 Register를 더 사용해야 하기 때문에 반올림은 0의 방향으로 하게끔 하였다. 사실 더욱 정확한 값을 얻고 싶다면 32bit float number보다 64bit float number를 쓰는 편이 더 나았을 것이라고 생각한다. 비록 정확도는 조금 떨어지지만 32bit float도 10진수 기준 숫자 6개까지는 유효 숫자이기 때문에 어느 정도 실용적인 범위의 정확성은 확보가 될 것이다.

어셈블리 프로그램을 설계하면서 sub-routine에서 신경 쓰는 것이 있는데 바로 preserved register, non-preserved register이다. 이 내용은 참고서적인 Sarah L. Harris & David Money Harris 의 *Digital Design and Computer Architecture ARM Edition*에 나와있는 내용이다. ARM register는 register별로 쓰임이 있다.

register	쓰임
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

특히 이 register구분은 sub-routine을 호출할 때 중요해지는데 preserved register는 callee가 save 해야하고 non-preserved register는 caller가 save해야하는 특징이 있다.

Preserved	Non-preserved
Saved registers: R4-R11	Temporary register: R12
Stack pointer: SP (R13)	Argument register: R0-R3
Return address: LR (R14)	Current Program Status Register (CPSR)
Stack above the stack pointer	Stack below the stack pointer

즉, sub-routine내부에서 따로 stack에 저장하지 않고 쓸 수 있는 register는 Non-preserved register인 R0-R3인 것이다. 또한 R12도 원래는 sub-routine에서 쓰이는 register이지만, 이번 프로젝트에서 filter를 위한 register를 6개나 써서 그냥 R12도 R4-R11과 같은 Saved register로 보고 사용하였다. convolution의 합산 값을 저장하는 용도로 R12가 사용되었다. R12의 원 용도는 Thumb instruction을 위해 쓰이거나 branch시 32bit로는 표현할 수 없는 먼 곳으로 갈 때 사용된다고 한다. 하지만 이번 프로젝트에서는 Thumb instruction을 사용하지 않고, 먼 곳으로 branch하지도 않으므로 그냥 Saved register로 보고 사용하였다.

프로젝트를 진행하면서 버그가 두개 있었는데 처음은 memory.ini가 규정하는 부분에 대한 읽기 쓰기 권한이 truncated 되어서 0xF0000000~0xF7FFFFFF 까지만 되는 오류가 있었다. 구글에 검색해 보니까 이 오류에 관한 내용이 많이 없었다. 그러나 누군가가 이번 상황이랑 똑같은 오류에 대해 물었고 what should I do?라고 물어봤는데 답변이 "WHAT SHOULD I DO???"하면서 정색하는 답변이었던 것이 기억이 남는다. 그 답변자는 이런거 물어볼 시간에 사용자 매뉴얼이나 더 읽어보라며 핀잔을 주며 정작 해결방안은 안 알려주었다. 그래서 사용자 매뉴얼을 봤는데 뭔진 모르겠지만 뭔가 memory 읽기/쓰기 권한을 부분을 나눠서 여러 번 요청하는 부분을 보았다. 그리고 debugger 내부에서도 memory 읽기/쓰기 권한을 제어할 수 있는 창이 있어서 해당 창으로 이것

저것 해 봤는데 얻은 결론은 0xF0000000~0xF7FFFFFF부분에 대해 읽기/쓰기 권한을 요청하고 0xF8000000~0xFFFFFFFF 부분에 대해 읽기/쓰기 권한을 따로따로 요청하면 되는 것이었다.

또 다른 버그는 분명 convolution을 했는데 -4.7보다 작은 값이 안 나오는 버그였다. 새로 만든 곱셈 sub-routine이 이상해서 그런 건지 시간을 들여서 검사 해 보았음에도 이상이 없었었기에 다시 convolution부분을 보았다. R5에 Sampled_Data주소를 저장해서 Sampled Data를 불러올 때 LDR 명령어를 immediate shifted register offset mode로 가져오기 위해 R3에서 R4에 저장된 index를 기준으로 immediate로 빼서 저장한 다음에 가져오는 것이었는데 굳이 R3에 계산 다 해놓고 R4를 기준으로 가져오니 가져오는 data가 같아져서 convolution이 제대로 이루어지지 않았던 것이다. 좀 더 간단히 설명하자면 다음 표와 같은 오류가 있었고 바로잡았다.

버그상황	해결
;FILTER[1] * DATA[I-1]	;FILTER[1] * DATA[I-1]
SUB R3, R4, #1	SUB R3, R4, #1
LDR R1, [R5, R4, LSL #2]	LDR R1, [R5, R3, LSL #2]

이번 프로젝트는 어셈블리로 프로그래밍 해본 것 중에 가장 긴 프로그램이었던 것 같다. 어셈블리는 군대에서 심심해서 처음에 Sarah L. Harris & David Money Harris/*Digital Design and Computer Architecture ARM Edition*/Elsevier Korea L.L.C/2016 책으로 공부를 했었다. 어셈블리 수업시간에 들은 내용은 사실 이미 대략적으로 알고 있는 내용이었지만 중간중간 디테일한 부분을 한국어로 설명을 들을 수 있어서 좋았다. 예를 들어 뺄셈을 하면서 왜 큰 수에서 작은 수를 빼면 carry가 set되는지 예시를 들면서 칠판에 적으면서 교수님께서 설명해 주신 부분이 기억에 남는다. 또한 책에서는 언급이 없었지만 실제로 게임 같은 것을 코딩하면서 자주 쓰는 부동 소수점에 대한 내용도 들을 수 있었던 점이 좋았다. 마지막으로 한 학기동안 강의해 주신 교수님, 과제를 내고 질문을 받고 채점을 해 주신 조교님께 감사함을 전합니다!

6. Reference

이형근/수업 및 강의자료, 어셈블리프로그램설계및실습/광운대학교/2022

엄태현, 이영복 /강의자료, 어셈블리프로그램설계및실습/광운대학교/2022

Sarah L. Harris & David Money Harris/*Digital Design and Computer Architecture ARM Edition*/Elsevier Korea L.L.C/2016

김상목 외 10인 / *공학수학* / 경문사 / 2017

ARM Cortex-A Series Programmer's Guide for ARMv7-A / https://developer.arm.com/documentation/den0013/d/Application-Binary-Interfaces/Procedure-Call-Standard

IEEE-754 Floating Point Converter / https://www.h-schmidt.net/FloatConverter/IEEE754.html

flow chart / Diagram Software and Flowchart Maker / https://www.diagrams.net/

graph자료 / GeoGebra / https://www.geogebra.org/?lang=ko