어셈블리 프로그래밍 설계 및 실습
프로젝트 보고서

실험제목: Bilinear Interpolation

제출일자: 2018년 11월 27일 (화)

학 과: 컴퓨터공학과

담당교수: 이형근 교수님

실습 분반: 화 6,7 , 수 5

학 번: 2017202010

성 명: 박유림

1. Introduction

A. 프로젝트 설명

이번 프로젝트에서는 floating-point로 이루어진 N-by-N 정방행렬에 대한 Bilinear Interpolation을 수행하는 것이다. N의 범위는 2<=N<=20이며 행과 열 각 2^k (이때, k = 1, 2, 3)배로 extend할 수 있어야 한다.

B. 프로젝트 목적

실수를 floating point로 나타내는 방법에 대해서 공부하고, 적용할 수 있다. 또한 floating point끼리의 덧셈연산을 진행할 수 다.

위의 addition와 shifter를 사용해서 multiplication을 진행한다.

보간법에 대해서 알아보고 주어진 보간법을 이용하여 N-by-N의 정방행렬에 대한 Bilinear Interpolation을 수행할 수 있다.

C. 일정, 계획 및 실천 여부

일정과 계획을 표로 나타내면 아래와 같다.

내용	목표 일정
제안서	11/13
알고리즘 정리	11/10 ~ 11/13
코드 작성	11/14~11/20
코드 검증	11/20 ~ 11/27
결과 보고서	11/20 ~ 11/27

실천된 상황을 표로 나타내면 다음과 같다.

내용	목표 일정
제안서	11/13
알고리즘 정리	11/10 ~ 11/20
코드 작성	11/20~11/25
코드 검증	11/25 ~ 11/26
결과 보고서	11/26 ~ 11/27

2. Project Specification

A. Biliear interpolation conditions

행과 열 각각 2^k(이때, k = 1, 2, 3)배를 진행한다. K는 Matix_data에서 주어진 숫자로 결정이 된다.

마지막 행, 마지막 열의 데이터는 padding을 적용하여 interpolation을 수행한다. 이때, padding은 생성될 데이터와 위치가 가장 가까운 데이터의 값을 pick한다.

예를 들면 아래와 같다.

-input data

(1, 5, 9, 13)

1	5
9	13

아래는 지정된 interpolation이 진행된 뒤의 result_data값의 행렬이다.

-k=1일 때, Bilinear interpolation된 result_data

1	3	5	5
5	7	9	9
9	11	13	13
9	11	13	13

-k=2일 때, Bilinear interpolation된 result_data

1	2	3	4	5	5	5	5
3	4	5	6	7	7	7	7
5	6	7	8	9	9	9	9
7	8	9	10	11	11	11	11
9	10	11	12	13	13	13	13
9	10	11	12	13	13	13	13
9	10	11	12	13	13	13	13
9	10	11	12	13	13	13	13

-k=2일 때, Bilinear interpolation된 result_data

1	1.5	2	2.5	3	3.5	4	4.5	5	5	5	5	5	5	5	5
2	2.5	3	3.5	4	4.5	5	5.5	6	6	6	6	6	6	6	6
3	3.5	4	4.5	5	5.5	6	6.5	7	7	7	7	7	7	7	7
4	4.5	5	5.5	6	6.5	7	7.5	8	8	8	8	8	8	8	8
5	5.5	6	6.5	7	7.5	8	8.5	9	9	9	9	9	9	9	9
6	6.5	7	7.5	8	8.5	9	9.5	10	10	10	10	10	10	10	10
7	7.5	8	8.5	9	9.5	10	10.5	11	11	11	11	11	11	11	11
8	8.5	9	9.5	10	10.5	11	11.5	12	12	12	12	12	12	12	12
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13
9	9.5	10	10.5	11	11.5	12	12.5	13	13	13	13	13	13	13	13

B. Matrix_data

Input data가 들어있는 배열이다.

- Input data의 N(가로, 세로 길이), k(interpolation scale), 데이터1, 데이터2, ... 가 순서대로 DCD명령어를 통해 선언되어 있다.
- “DataArea”라는 이름의 DATA AREA에 저장이 되어있다.
- N, k의 값과 임의의 데이터 N^2개가 들어있는 메모리의 첫 주소값을 나타낸다. (순서는 위에서 말한것과 같다)
- 데이터들은 -2000 ~ 2000 사이의 floating-point로 이루어져 있다.

```
445     AREA DataArea, DATA
446 Matrix_data
447     ALIGN
448     DCD 10
449     DCD 1
450     DCD 2_00111111100010000000000000000000
451     DCD 2_11000010100001000000000000000000
452     DCD 2_01000010101001000000000000000000
453     DCD 2_11000011001110000000000000000000
454     DCD 2_00111111100011000000000000000000
455     DCD 2_11000010000010000000000000000000
456     DCD 2_01000100000010000000000000000000
457     DCD 2_11000010000000100000000000000000
458     DCD 2_11000001101010000000000000000000
459     DCD 2_11000010111111000000000000000000
460     DCD 2_11000010100000000000000000000000
```

445 : DataArea라는 이름을 가졌으며, DATA AREA를 의미하고 있다.

446 : DCD 명령어를 이용하여 Matirx_data가 선언되어 있다.

448 : 입력 데이터의 가로(=세로) 길이(N)이 첫번째 요소로 저장되어 있다.

450 : interpolation scale(k)가 두번째 요소로 저장이 되어 있는 것을 확인할 수 있다.

451~ : -2000 ~ 2000 사이의 부동소수점의 데이터들이 왼쪽에서 오른쪽, 위에서 아래 순서로 선언되어 있다.

C. Result_data

Matrix_data의 원소들을 이용하여 Bilinear Interpolation condition에 맞게 Interpolation을 수행한 결과가 저장되는 데이터 공간의 시작 주소 값을 나타낸다.

- 1 word단위로 저장한다.
- 0x60000000 번지로 지정한다.

```
442     AREA AdrArea, DATA
443 Result_data &   &60000000   ;address of result
```

442 : AdrArea라는 이름을 가졌으며, DATA AREA를 의미하고 있다.

443 : Result_data가 저장된 공간의 시작 주소 값인 0x60000000을 저장하고 있다.

D. Try

Result_data, odd_Result_data, odd 연산 과정에서 Matrix_Data의 위치, even 연산 과정에서 Matrix_Data의 위치를 저장하고 있는 공간의 시작 주소 값을 나타낸다.

try

[0x40000000] Result_data의 현 위치	[0x40000004] Odd Result data의 현 위치	[0x40000008] Odd 연산 과정中 Matrix_Data의 위치	[0x4000000C] even 연산 과정中 Matrix_Data의 위치

```
438  try &    &40000000
```

438 : try는 위의 주소값들을 저장하고 있는 공간의 시작 주소인 0x40000000을 담고 있다.

E. Load_data

unrolling을 통해 2배하는 과정을 두 번 거쳐 4배, 세번 거쳐 8배하는 과정에서 Result_data의 값들이 바뀌지만 이들의 값들을 저장하고 있어야 한다. 따라서 이 값들을 읽고 저장하는 공간이다.

```
440  load_data    &    &30000000
```

440 : load_data는 Result_data의 값들을 load해와 저장하고 있는 공간의 시작 주소인 0x30000000을 담고 있다.

F. Odd_Result_data

Interpolation을 할 때, 가장 윗 줄을 0번째 줄로 둔다. 이때, 홀수번째 줄의 0, 2, .. 번째 칸은 윗칸과 아랫칸의 보간을 통해 이루어져야 한다. 이 보간의 결과값을 저장하는 공간의 시작 주소 값을 나타낸다.

```
436  odd_Result_data &    &50000000
```

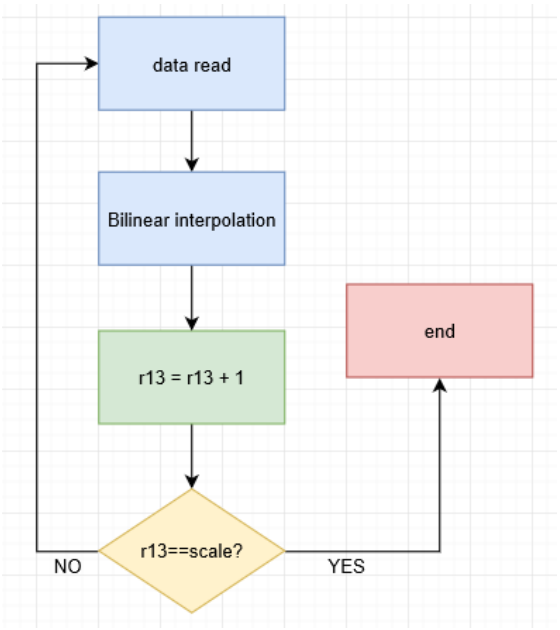
436 : odd_Result_data는 홀수번째 줄의 짝수번째 값들을 저장하고 있는 공간의 시작 주소인 0x50000000을 담고 있다.

3. Algorithm

A. Entire

이번 프로젝트에서 주어진 input 행렬의 2배 보간을 k번 진행하면 2*k배 보간이 된다는 점을 이용하였다. 따라서 r13이라는 2배 보간을 진행한 횟수를 저장하는 register를 이용하여, 진행을 하였다.

일단 보간을 진행하는데 필요한 알맞은 데이터를 읽어온다. 이때, 필요한 데이터를 읽어오는 주소값이 다를 수 있음에 주의한다. 읽어온 데이터를 중심으로 2배 보간을 진행한다. r13의 값을 하나 올려준 후 r13과 읽어온 scale값과 비교한다. 만약 같지 않으면 위의 방법을 다시 진행한다(rolling). 같다면 원하는 행렬이 보간을 완료했으므로 끝내준다.



B. Even row interpolation

행렬의 2배 확장을 진행하는데 있어 조건을 따르면 아래와 같다. 가장 윗 줄을 0번째 줄이라 명할 때, 짝수번째 줄에는 확장 전의 데이터들이 한 칸 씩 떨어져 분포되어 있게 된다. 이 데이터를 이용하여 보간을 진행한다. 보간을 해야 하는 data의 위치 왼쪽에 있는 data를 data1, 오른쪽에 위치해 있는 data를 data2라 이름 붙인다. 보간을 해야 하는 위치는 이 두 데이터의 가운데이고, 보간법에 따르면 그 값은 $\frac{\text{data1} + \text{data2}}{2}$가 된다.

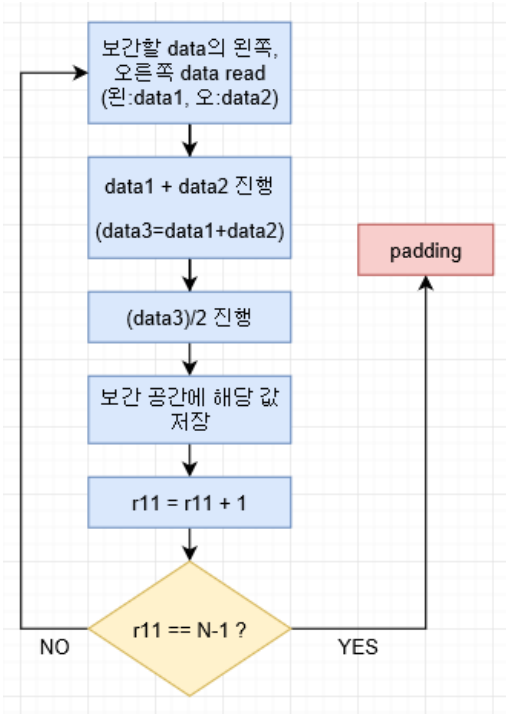
따라서 data1 + data2의 연산을 진행한다. 덧셈의 방법은 C에서 설명하도록 한다. 이렇게 나온 data1 + data2의 값을 data3이라 칭한다.

다음 할 일은 data3에 0.5를 곱해주는 일이다. 0.5를 floating point로 나타내면 다음과 같다.

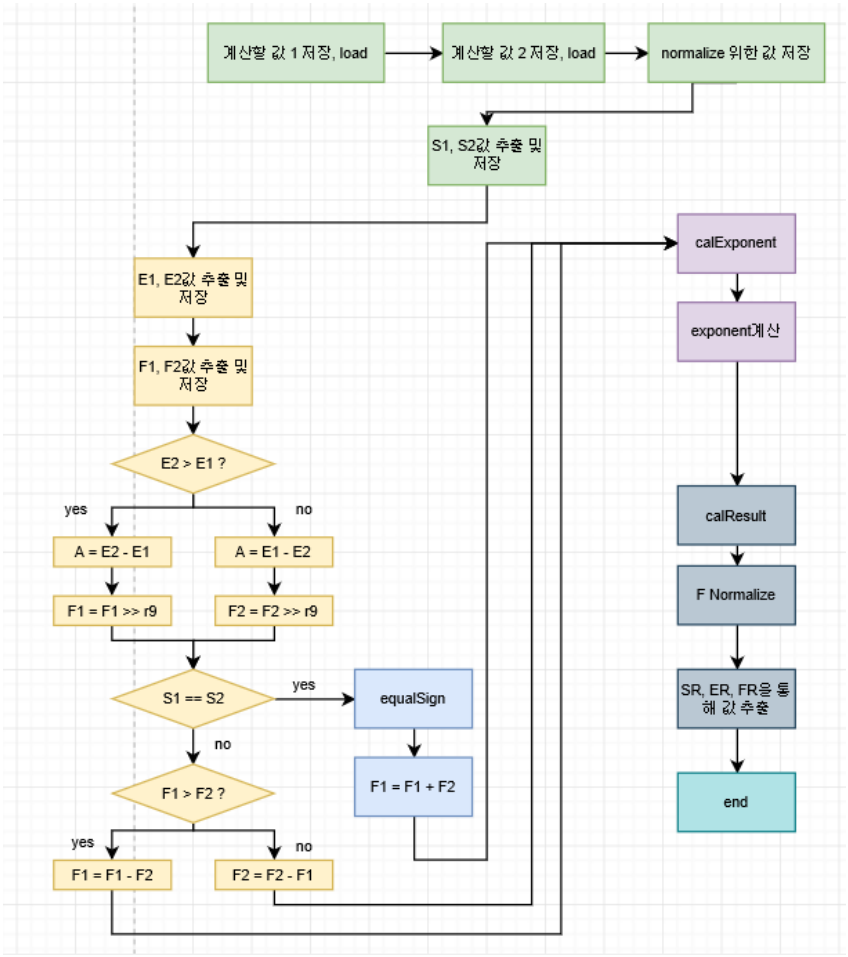
0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

즉, 0.1이 된다. 이 수를 multiplier로 가지고 binary multiplication을 통하면, multiplicand의 mantissa와 sign값은 같고 binary multiplication중 shift로 인해 exponent만 1이 주는 것을 알 수 있다. 따라서 data3의 exponent에서 1을 빼주면, $\frac{data3}{2}$가 완료된 것이다. 이제 보간할 데이터의 연산이 끝났다.

마지막으로 보간 공간에 해당 값을 저장시킨다. 보간이 하나 완료되었다는 신호를 주는 11번째 register값을 하나 올려주고 r11과 N-1을 비교한다. 같지 않으면, 왼쪽, 오른쪽 데이터가 모두 존재한다는 의미로 보간을 위한 연산을 진행하고 그렇지 않으면, 오른쪽 데이터가 없다는 뜻으로 왼쪽 데이터를 복사해 보간할 공간에 넣어주는 padding을 진행한다.



C. Floating-point addition



- Sign bit, Exponent bits, Mantissa(Fraction) bits를 각 숫자에 대해서 추출한다.
- Matissa 형태 앞에 1을 붙인다
- Exponent를 비교하고 두 floating point의 exponent의 차이 값을 구하고 그 값을 shift num이라 한다.(|Exponent1 – Exponent2| = shift num)
- Shift num이 0이 아닌 경우, Exponent 값이 작은 값의 mantissa값을 shift num만큼 오른쪽으로 shift 해준다.
- Sign bit를 비교하여 다르면 큰 Mantissa값에서 작은 Mantissa값(|Mantissa1 – Matissa2|)을, 같으면 두 값을 더해준다.
- 연산에서 나온 Mantissa값을 Normalize해준다.
- 수정이 완료된 Mantissa 값과 exponent 값을 사용하여 결과를 도출한다.

D. Odd row interpolation

주어진 보간법에 따라서 홀수 번째 줄에는 확장되기 전 DATA가 전해지지 않는다. 이에 짝수 번째 줄에 있는 확장 전 data가 위아래로 존재하는 보간 부분을 먼저 보간해주어야 한다. 예를 들면 아래 그림에서 초록색 부분을 우선적으로 보간해주어야 한다.

A	$\frac{(A+B)}{2}$	B	B
C		D	

보간할 부분에서 위의 data를 data1, 아래 데이터를 data2라고 했을 때, 보간할 부분의 data는 $\frac{data1 + data2}{2}$가 된다. Even row interpolation에서 했던 연산과 같게 진행을 해주면 값이 나오고, 이 값을 odd_Result_data에 저장을 한다.

Odd_Result_data의 값들이 위의 그림에서 초록색 부분에 들어갈 floating point들이고 이에 초록색 부분이 채워져 있다고 가정할 수 있다. 그렇다면, 다음 연산부터는 even row interpolation과 다를 바 없다.

A	$\frac{(A+B)}{2}$	B	B
$\frac{(A+C)}{2}$	$\frac{(A+B+C+D)}{4}$	$\frac{(B+D)}{2}$	$\frac{(B+D)}{2}$
C	$\frac{(A+B)}{2}$	D	D

만약 현재 보간할 행이 마지막 행일 경우에는, 바로 위 행을 padding하는 방식으로 진행한다.

E. Padding

간단히, 보간할 자리의 전 자자리의 data를 읽어서 보간할 자리에 써준다.

4. Performance & Result

Performance는 state^2 x code size로 진행이 된다. Performance를 줄이기 위해서 명령어가 몇 개 없는 경우에는 branch보다 condition을 통해서 조건에 따른 명령어를 실행했다.

주어진 test를 실행한 결과 화면은 다음과 같다.

<test1 과정, 및 결과>

-빌드 결과

Build Output

```
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Target 1'
assembling project.s...
project.s(434): warning: A1608W: MOV pc,<rn> instruction used, but BX <rn> is preferred
linking...
Program Size: Code=1256 RO-data=0 RW-data=48 ZI-data=0
".\Objects\project.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed:  00:00:02
```

Code의 size가 1256인 것을 확인할 수 있다.

-Matrix_data

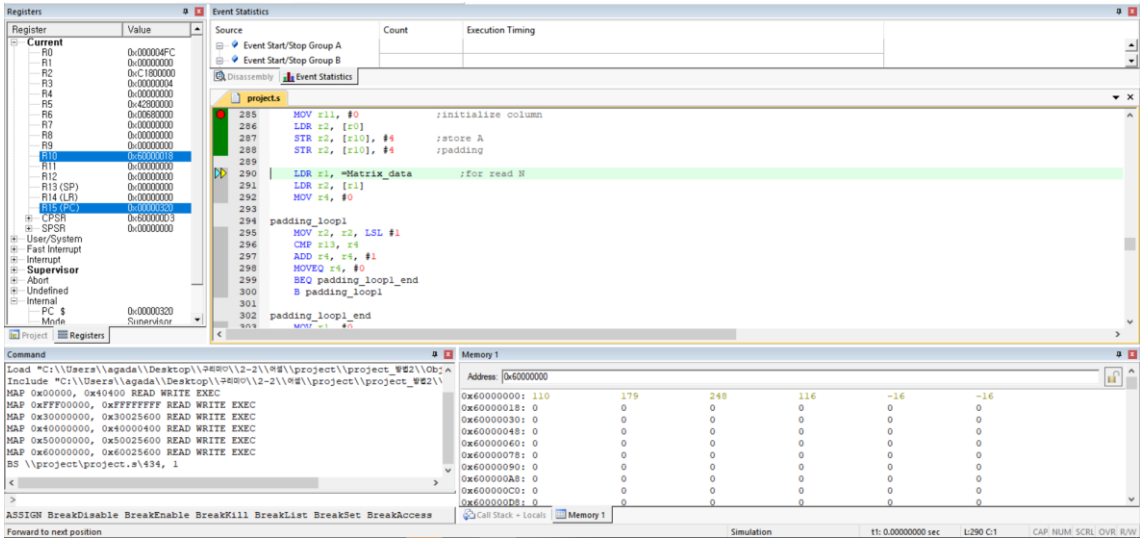
Memory 1

Address: 0x4ec

```
0x000004EC: 00000003 00000001 42DC0000 43780000 C1800000 C2A80000 C1580000 407C0000
0x0000050C: BEB40000 42700000 C4400000 00000000 00000000 00000000 00000000 00000000
0x0000052C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x0000054C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x0000056C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x0000058C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x000005AC: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x000005CC: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x000005EC: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x0000060C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

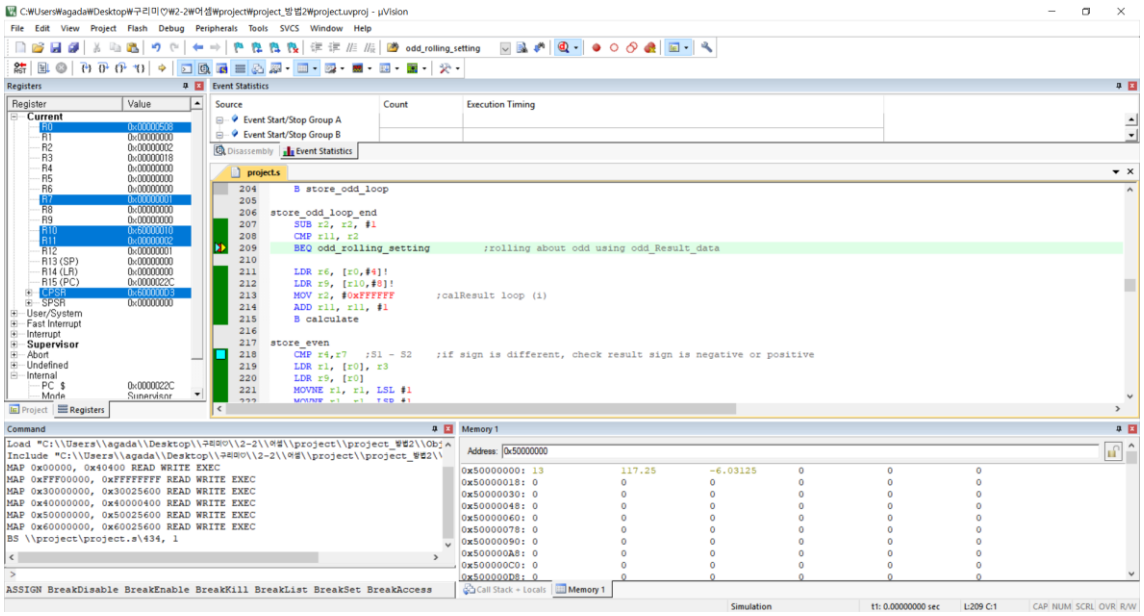
-한 줄 완료

계산을 두 번 진행하고, padding을 한번 진행하면 한 줄의 연산이 완료가 된다. 완료된 결과 result_data에 해당하는 메모리에 알맞은 값이 들어간 것을 확인할 수 있다.



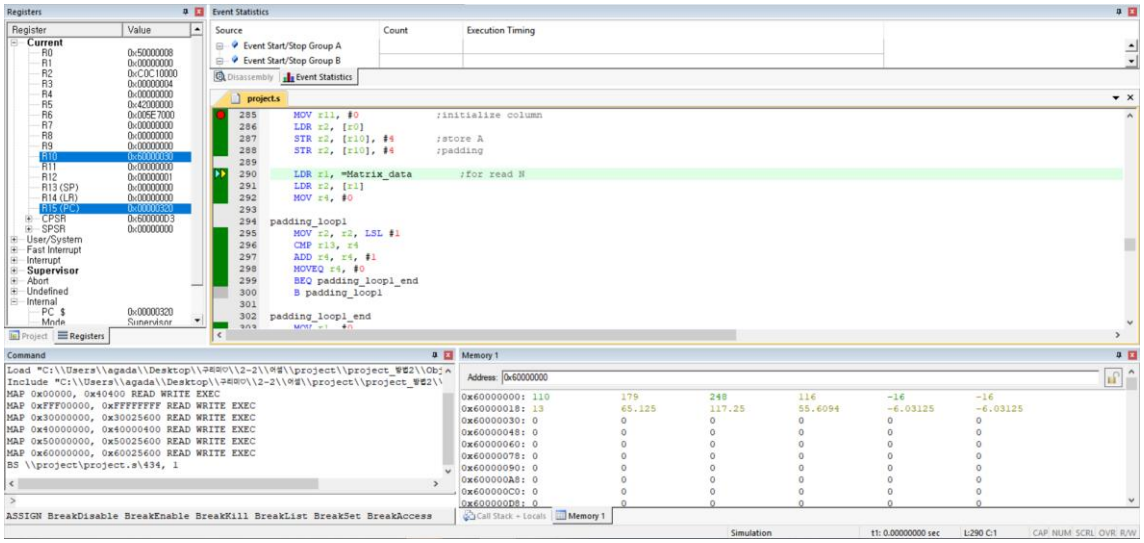
-홀수 번째 줄 보간1 완료

0번째 줄이 끝난 후, 1번째 줄의 연산을 시작한다. 위의 알고리즘에서 설명했듯이 홀수 줄의 연산은 초록부분을 먼저 odd_result_data에 저장해야 한다. 이를 진행한 결과화면이다. 알맞은 값이 들어가 있는 것을 확인할 수 있다.



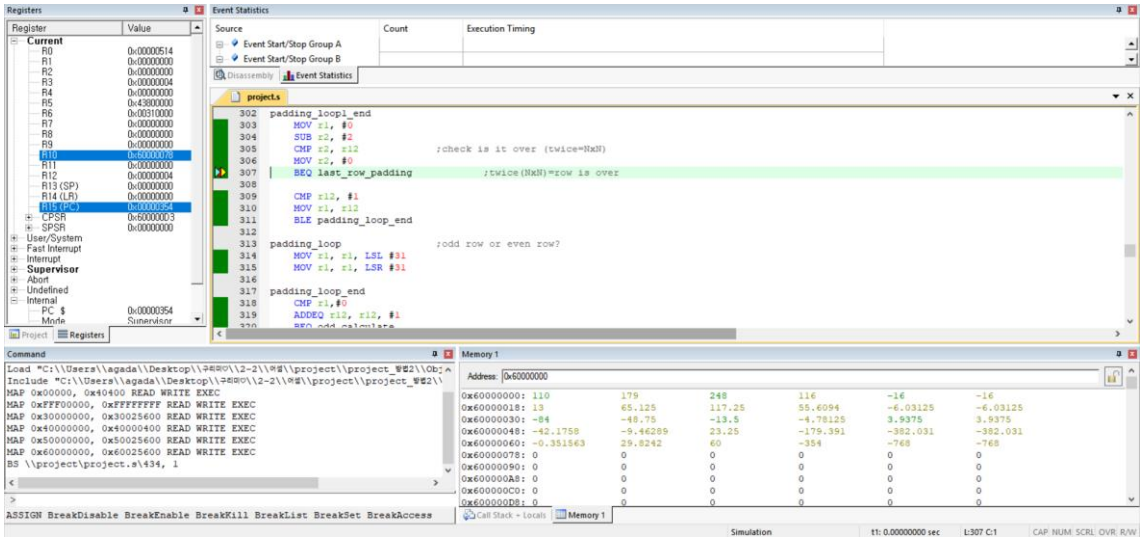
-홀수 번째 줄 보간2 완료

홀수 번째 줄의 보간2를 진행하기 위해서 보간1에서 odd_result_data에 알맞은 초록부분 값을 넣어주었다. 보간2에서는 odd_result_data를 가지고 보간을 완료한다. 그 결과 값은 아래와 같다.



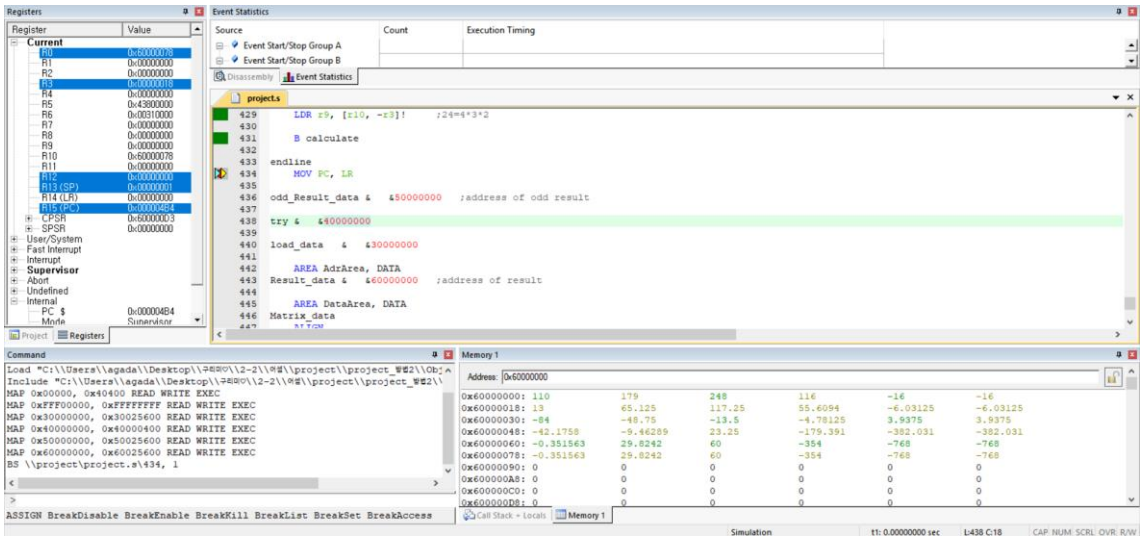
-마지막 줄 padding 전

N*2-1번 odd or even row 연산을 끝내게 되면, 마지막 줄이 나온다. 이때 마지막 줄은 그 바로 윗줄을 padding한 결과로 이루어지게 된다.



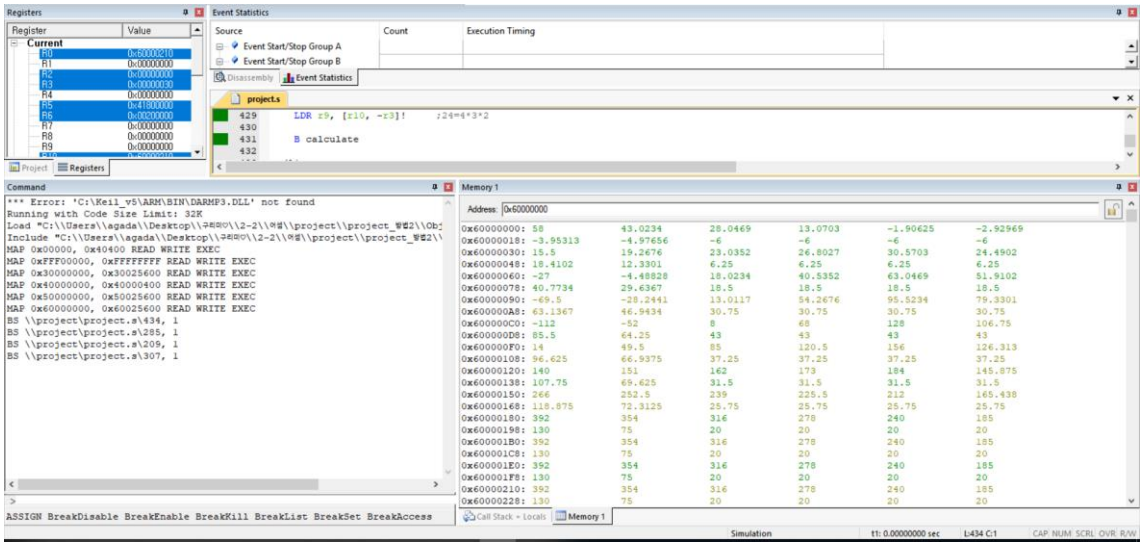
-마지막 줄 padding 후

마지막 줄을 padding한 결과 마지막 줄과 마지막의 바로 전 줄의 결과 값이 같게 나오는 것을 확인할 수 있다.



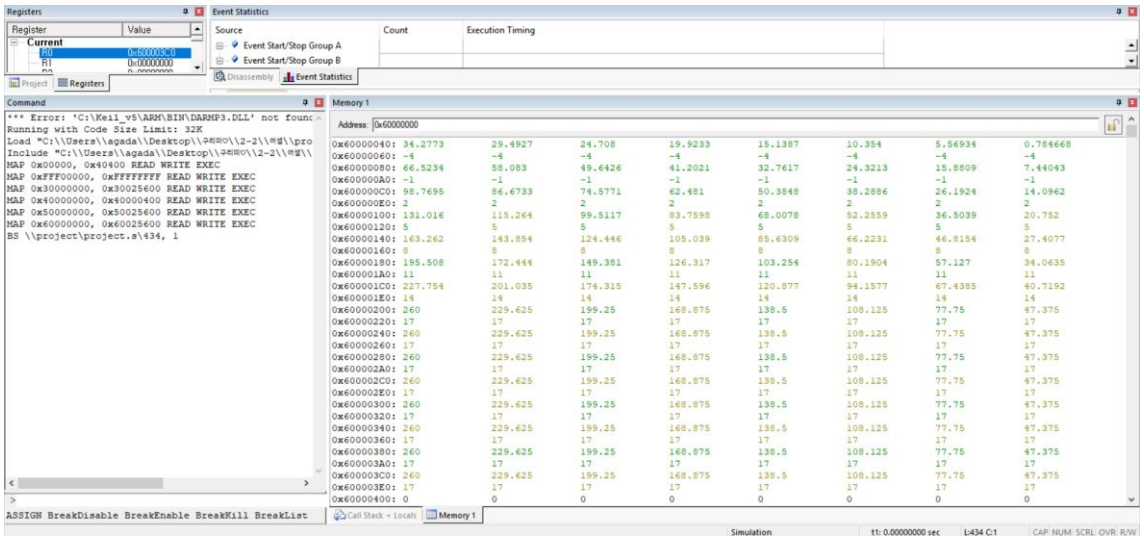
<test2 결과화면>

주어진 결과 text file과 비교하여 봤을 때, 보간이 적절히 잘 된 것을 확인하였다.



<test3 결과화면>

주어진 결과 text file과 비교하여 봤을 때, 보간이 적절히 잘 된 것을 확인하였다.



<test4부터는 생략>

5. Conclusion

이번 프로젝트는 N-by-N의 정방행렬을 Bilinear interpolation하는 것이었다. 특이한 점은 multiplication을 진행할 때, multiplication 명령어인 MUL의 사용이 금지된 것이었다.

프로젝트를 진행하면서 가장 힘들었던 점은 어떠한 값을 저장할 수 있는 register가 한정이 되어있다는 것이었다. Input data를 읽어올 위치, 중간 data를 읽고 쓸 위치, output data를 쓸 위치 등 memory에 access하기 위해서 필요한 주소값들부터 계산과정에서 계산중인 결과, 어떠한 loop를 돌 때 count 등등 저장해야 할 요소들이 너무나 많지만 사용할 수 있는 register의 개수는 14개로 지정이 되어있었다.

무작정 register를 이용해서 코드를 짜다가 register의 개수가 모자라서 더 이상 나아가지 못하는 문제점을 겪게 되어있다. 이에 stack에 대해서 배울때 memory와 register간의 교류를 활용해서 프로그래밍을 진행해야 한다던 교수님의 말씀이 떠올랐다. 이에 주소값들을 저장하던 register를 모두 없애버리고 이 모든 주소값들을 저장하고 있는 memory의 시작 주소하나만을 register로 저장했다. 또한 계산이 끝난 register에 대해서는 초기화 후 재사용했다. 이렇게 하니 초기화 과정이나 값들을 memory에서 register로 불러오거나 register값을 memory에 저장하는 과정에서 더 복잡해지긴 했지만 register가 한정되어 있다는 한계를 극복할 수 있었다.

다음으로는 개념에 대한 이해가 부족한 채로 프로그래밍을 진행해서 고치는 과정에서 시간을 많이 소비했다. 처음에는 홀수 번째 행일 때, 위 행과 아래 행의 data들을 가지고 보간1을 해야한다는 미처 생각하지 못하고 코드를 구현했더니 홀수 행에 나와야 하는 값들이 전부 나오지 않는 것을 확인할 수 있었다.

이에 한줄씩 디버깅을 해가면서 어디가 문제인지 찾아보았고 결국, 개념에 대한 이해가 부족했고 이 때문에 오류가 생긴 것을 깨달았다. 개념을 애초에 알고 알고리즘을 짰더라면 덜 어려웠을 것들이 반쪽자리 알고리즘에 개념을 통해 완벽한 알고리즘으로 바꾸려고 하니 더욱 어렵게 느껴졌다. 하지만 반복적인 디버깅과 코드 수정을 통해서 이를 고쳤다. 개념에 대해서 확실히 알고 프로그래밍을 하지 않으면 안되겠다는 생각을 다시 한번 가지게 되었다.

과제 완료 후 우선 floating point에 대한 이해가 확실해졌다. 또한 register의 개수가 한정되어 있을 때 memory와 register사이에서 읽고 쓰는 과정을 통해서 이를 극복해나갈 수 있다는 것을 알게 되었다. 이번 프로젝트를 통해서 assembly처럼 register의 개수가 한정되어있는 언어를 사용할 때, memory와 register 사이에 읽고 쓰는 과정을 통해서 프로그래밍을 잘 진행할 수 있을 것이라 생각된다.