어셈블리 프로그래밍 설계 및 실습  
Term project 결과 보고서

제출일자: 2018년 11월 27일 (화)

학 과: 컴퓨터정보공학부

담당교수: 이형근 교수님

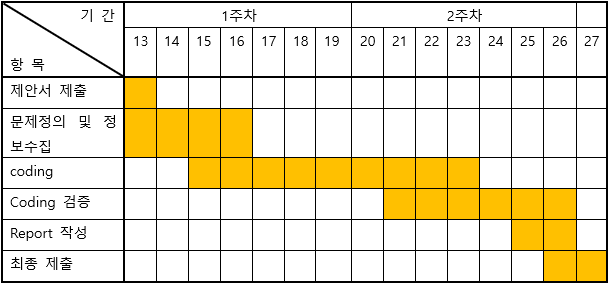
실습분반: 화 6,7 , 수 5

학 번: 2017202087

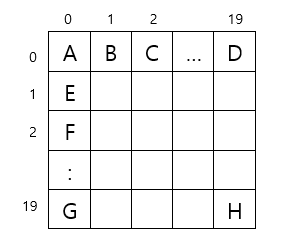
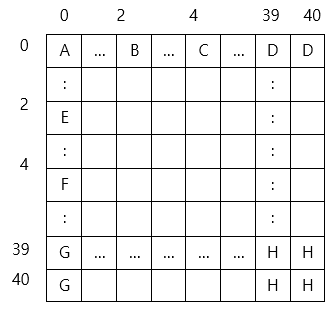
성 명: 홍 세 정

1. Introduction
   1. 제목 및 목적

임의의 부동소수점 데이터로 이루어진 N x N(2 <= N <= 20) 정방행렬에 대한 bilinear interpolation을 수행하여 행과 열을 각 2배, 4배, 8배로 확장한 정방행렬을 계산한다. Bilinear interpolation(이중선형 보간)을 이용하여 행렬을 확장할 수 있고, Linear interpolation의 수식을 알고 이해하며 가운데 값을 추정할 수 있다. 또한 성능이 좋은 코드를 구현할 수 있다.

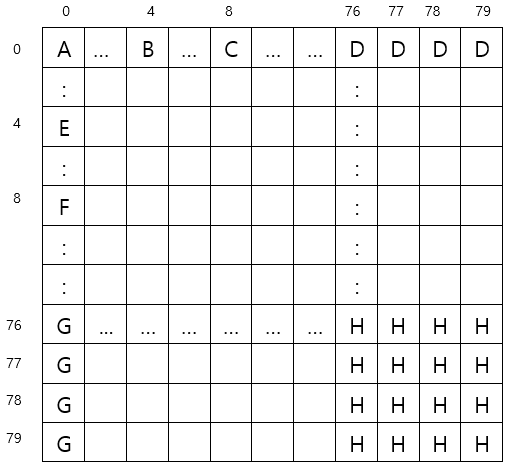
* 1.  일정

1. Project specification
   1. Bilinear interpolation conditions



<20x20행렬>

<20x20행렬의 2배>



<20x20행렬의 4배>

2배로 확장했을 경우 들어온 input 값 사이의 1개의 수가 더 들어가게 되어 위의 그림과 같이 나타낼 수 있다. 두 개의 input에서는 사이의 수는 bilinear interpolation을 하여 알맞은 값을 구할 수 있다. 그렇게 해서 만들어진 행렬은 행과 열의 수가 N-1인 행렬을 만들 수 있다. Interpolation이 불가능한 외각에 추가된 값들은 현재 위치와 가장 가까운 값으로 복사하는 padding을 수행할 수 있다.

4배 확장하는 경우는 2배 확장한 행렬을 2배 확장한 행렬과 같다. 즉, 2배 확장하는 연산을 두 번 반복한다.

8배 하는 경우도 마찬가지로, 4배 확장한 행렬을 2배 확장한 행렬과 같다. 즉, 2배 확장하는 연산을 세 번 반복하는 것과 똑같다.

행렬( NxN )을 두 배로 확장하기 위해서 가로열부터 먼저 두배로 확장한 후( 2NxN ), 확장한 가로열을 세로행으로 두배 확장한다( 2Nx2N ).

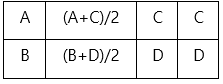
구현한 코드에서는 세로행을 두 번 입력 받아서 두배로 확장하게 되는데, 가로열을 먼저 확장하고 세로행을 확장하게 되면 행과 열이 바뀐 형태인 전치행렬이 나오게 되기 때문에 결과값을 저장할 때 한번 더 바꿔주어 저장해야 된다. 따라서 세로행을 먼저 입력 받게 되면 2배하였을 때 올바른 행렬의 값이 나올 수 있다.

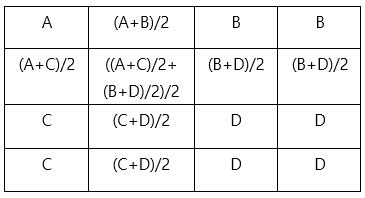
예를 들어 다음과 같은 2x2행렬이 있다고 하면

A B C D 와 같은 순서대로 저장된다.

A와 C의 데이터를 읽어 사이에 들어갈 알맞은 값(r7)을 구하고

A, r7, C 순서대로 값을 저장한다. C의 자릿수가 N이라면 padding을 수행하여 외각에 수를 채울 수 있다.

위의 행렬 세로행을 2배해주면 다음과 같은 4x2행렬을 만들 수 있다. A, (A+C)/2, C, C, B, (B+D)/2, D, D 와 같은 순서로 저장된다.



위의 4x2행렬의 세로행을 2배 해주면 4x4행렬을 만들 수 있다.

A, A’B, B, B, A’C, A’D, B’D, B’D, C, C’D, D, D, C, C’D, D, D의 순서로 저장할 수 있다.

다음과 같은 방법으로 구현하면 입력데이터들의 중점만 구하게 되면 2배하는 행렬을 구할 수 있다. 따라서 floating point multiplication을 따로 구현하지 않아도 x1/2하는 연산만으로도 bilinear interpolation을 할 수 있다. Floating point에서의 exponent의 성질을 이용해서 exponent – 1을 해주면 x 1/2과 같은 역할을 하게 된다. 구현한 코드에서 cal이라는 함수에서 두 개의 입력을 받아 중점을 구할 수 있다.

* 1. Matrix\_data

Matrix\_data에서는 DCD 명령어를 이용하여 행렬의 크기(NxN), interpolation scale(k), NxN개의 데이터가 순서대로 저장되어 있다. 임의의 데이터는 -2000 ~ 2000 사이의 부동소수점 값을 가진다.

* 1. Result\_data

NxN행렬 Matrix\_data를 k(1~3)배 한 결과 값을 0x60000000번지에 순서대로 저장된다. 데이터를 저장할 때에는 1 word 단위로 저장한다.

* 1. SAVE\_data

SAVE\_data & 0x90000 : padding할 때 +1 / N + 2N = 3N (2배를 했을 때 최종 값)

SAVE\_data2 & 0x90008 : low함수를 다 읽었을 때 +1 / N (2배를 했을 때 최종 값)

SAVE\_data3 & 0x90010 : column함수를 다 읽었을 때 +1 / 2N(2배를 했을 때 최종 값)

SAVE\_data4 & 0x90018 : 2배를 완료했을 때 +1 / k (최종 값)

SAVE\_data5 & 0x90020 : column함수를 읽을 때마다 +N / N x N (2배를 했을 때 최종 값)

SAVE\_data는 상황에 따라서 필요한 data를 각 주소에서 읽어오고 저장할 수 있다.

* 1. TEST\_Data

TEST\_data1 & 0x30000 : Matrix\_data 세로를 2배 한 행렬. (2N x N)

TEST\_data2 & 0x40000 : TEST\_Data1 세로를 2배 한 행렬. (2N x 2N)

N x N을 2배한 행렬

TEST\_data3 & 0x50000 : TEST\_Data2 세로를 2배 한 행렬. (4N x 2N)

TEST\_data4 & 0x60000 : TEST\_Data3 세로를 2배 한 행렬. (4N x 4N)

N x N을 4배한 행렬

TEST\_data5 & 0x70000 : TEST\_Data4 세로를 2배 한 행렬. (8N x 4N)

TEST\_data6 & 0x80000 : TEST\_Data5 세로를 2배 한 행렬. (8N x 8N)

N x N을 8배한 행렬

K가 1일 때는 2배한 행렬(TEST\_data2)을, k가 2일 때는 4배한 행렬(TEST\_data4)을, k가 3일 때는 8배한 행렬(TEST\_data6)을 Result\_data에 저장한다.

* 1. Register

r12 : 데이터 값을 읽어오는 register.

r13 : 연산이 완료된 데이터를 저장하는 register.

r10 : N (고정으로 사용되며 변하지 않는다.)의 값을 가지는 register.

r11 : k (고정으로 사용되며 변하지 않는다.)의 값을 가지는 register.

r14 : count하는 register. padding할 때 0으로 초기화 해준다.

r15 : pc register(N x N의 행렬이 k배가 다 끝나면 0으로 초기화).

r1 : 연산해야 할 첫번째 data의 값.

r2 : 연산해야 할 두번째 data의 값.

r7 : 두개의 값(r1, r2)의 중점.

나머지의 register값들은 중점을 구하는 연산을 할 때나 다른 값들을 더해줄 때 유동성 있게 쓰인다.

* 1. 사용된 함수의 역할

Low1, low2, low3 : 데이터를 읽어오는 r12를 지정하는 함수.

low\_load : 행렬의 가로를 2배로 증가시켜주는 함수. 전의 행렬의 세로행을 읽어 2배로 확장해 준다. (2N x N)

column1, column2, column3, : 데이터를 읽어오는 r12를 지정하는 함수.

column\_load : 행렬의 세로를 2배로 증가시켜주는 함수. 전의 행렬의 세로행을 읽어 2배로 확장해 준다. (2N x 2N)

floating\_save : 연산이 완료된 data를 r1, r7, r2 차례대로 r13에 저장해주는 함수.

padding : 1부터 N개 사이의 연산이 끝나면 Interpolation이 불가능한 외각에 추가된 값을 N으로 복사하는 함수.

Twice : k가 1일 때 저장하는 r13 지정하는 함수 (2배).

Quadruple : k가 2일 때 저장하는 r13 지정하는 함수 (4배).

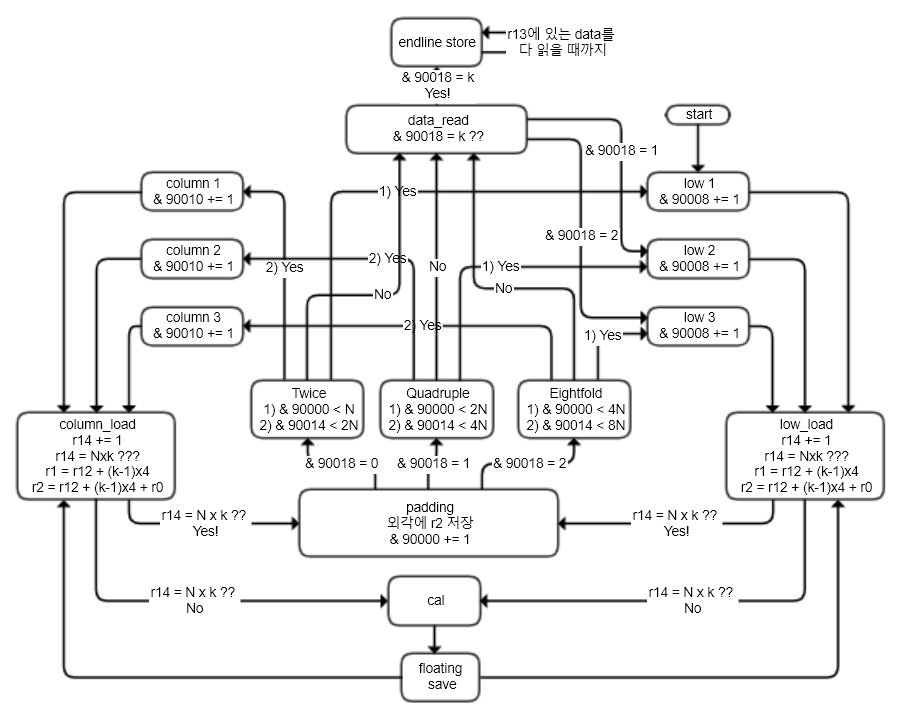
Eightfold : k가 3일 때 저장하는 r13 지정하는 함수 (8배).

data\_read : SAVE\_data, SAVE\_data2, SAVE\_data3을 모두 0으로 초기화 k가 SAVE\_data4와 같으면 endline.

store : k배한 행렬을 Result\_data에 저장하는 함수.

endline : 읽어올 주소, 저장할 주소을 지정.

cal : 2개의 floating\_point input 사이에 중점을 구해주는 함수.

1.  Algorithm
2. Performance & Result
   1. Test1

Input\_data\_binary

DCD 3 = N

DCD 1 = k

DCD 2\_01000010110111000000000000000000 = 110

DCD 2\_01000011011110000000000000000000 = 248

DCD 2\_11000001100000000000000000000000 = -16

DCD 2\_11000010101010000000000000000000 = -84

DCD 2\_11000001010110000000000000000000 = -13.5

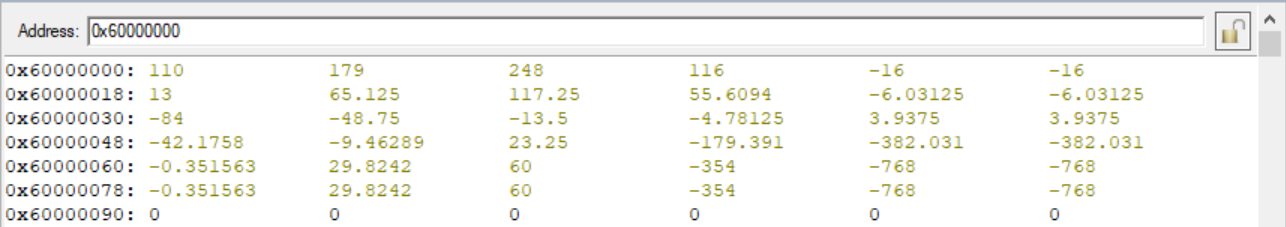
DCD 2\_01000000011111000000000000000000 = 3.9375

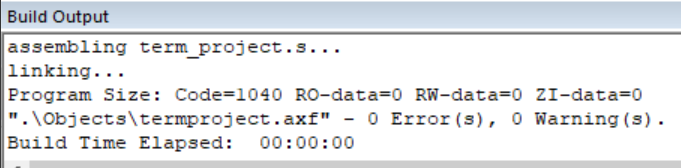
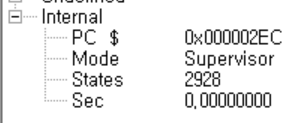
DCD 2\_10111110101101000000000000000000 = -0.351563

DCD 2\_01000010011100000000000000000000 = 60

DCD 2\_11000100010000000000000000000000 = -768

3x3행렬이 2배가 되면 6x6행렬이 완성되어 다음과 같은 행렬이 60000000번지에 저장된다.

interpolated\_output\_data\_decimal

performance

* 1. Test2

Input\_data\_binary

DCD 3 = N

DCD 2 = k

DCD 2\_01000010011010000000000000000000 = 58

DCD 2\_10111111111101000000000000000000 = -1.90625

DCD 2\_11000000110000000000000000000000 = -6

DCD 2\_11000010111000000000000000000000 = -112

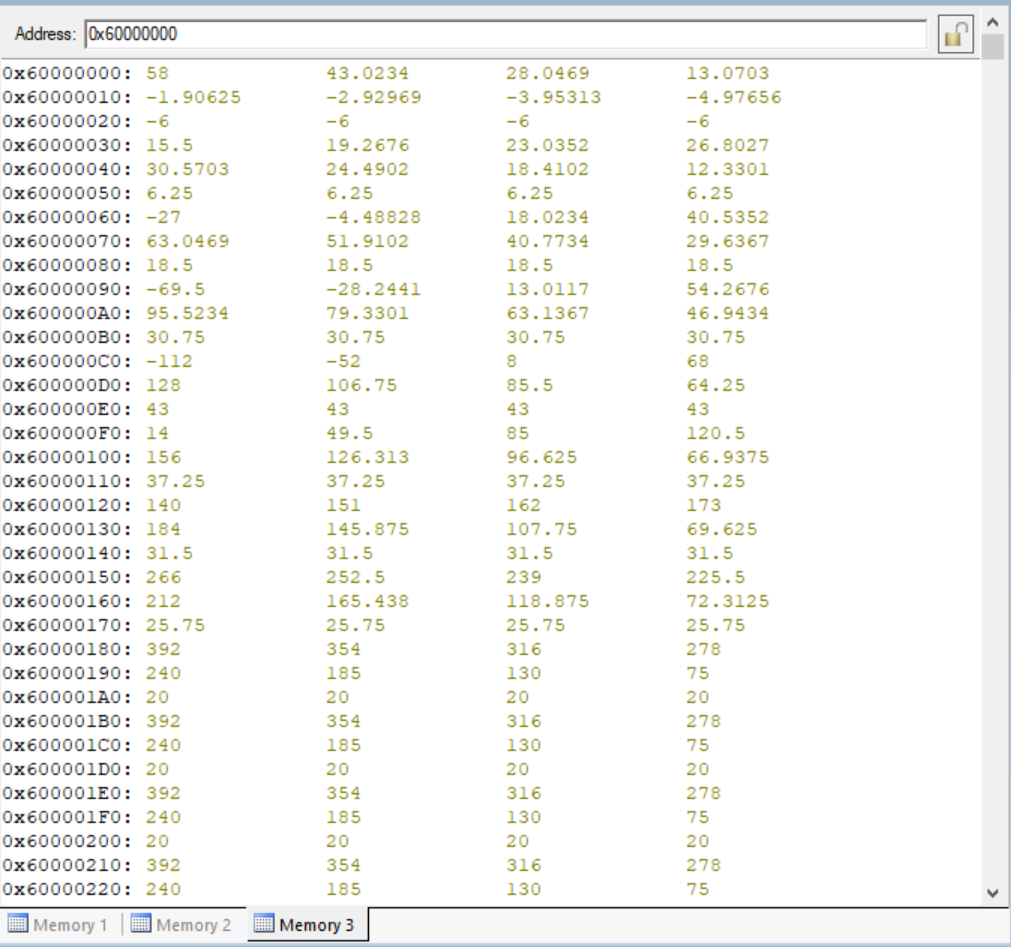
DCD 2\_01000011000000000000000000000000 = 128

DCD 2\_01000010001011000000000000000000 = 43

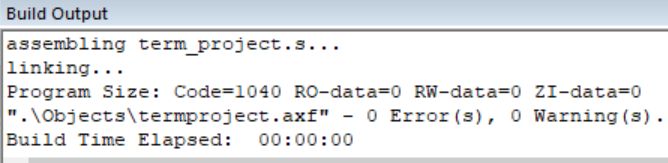
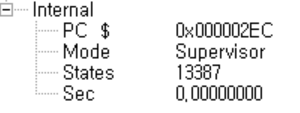
DCD 2\_01000011110001000000000000000000 = 392

DCD 2\_01000011011100000000000000000000 = 240

DCD 2\_01000001101000000000000000000000 = 20

interpolated\_output\_data\_decimal

performance



* 1. Test3

Input\_data\_binary

DCD 2 = N

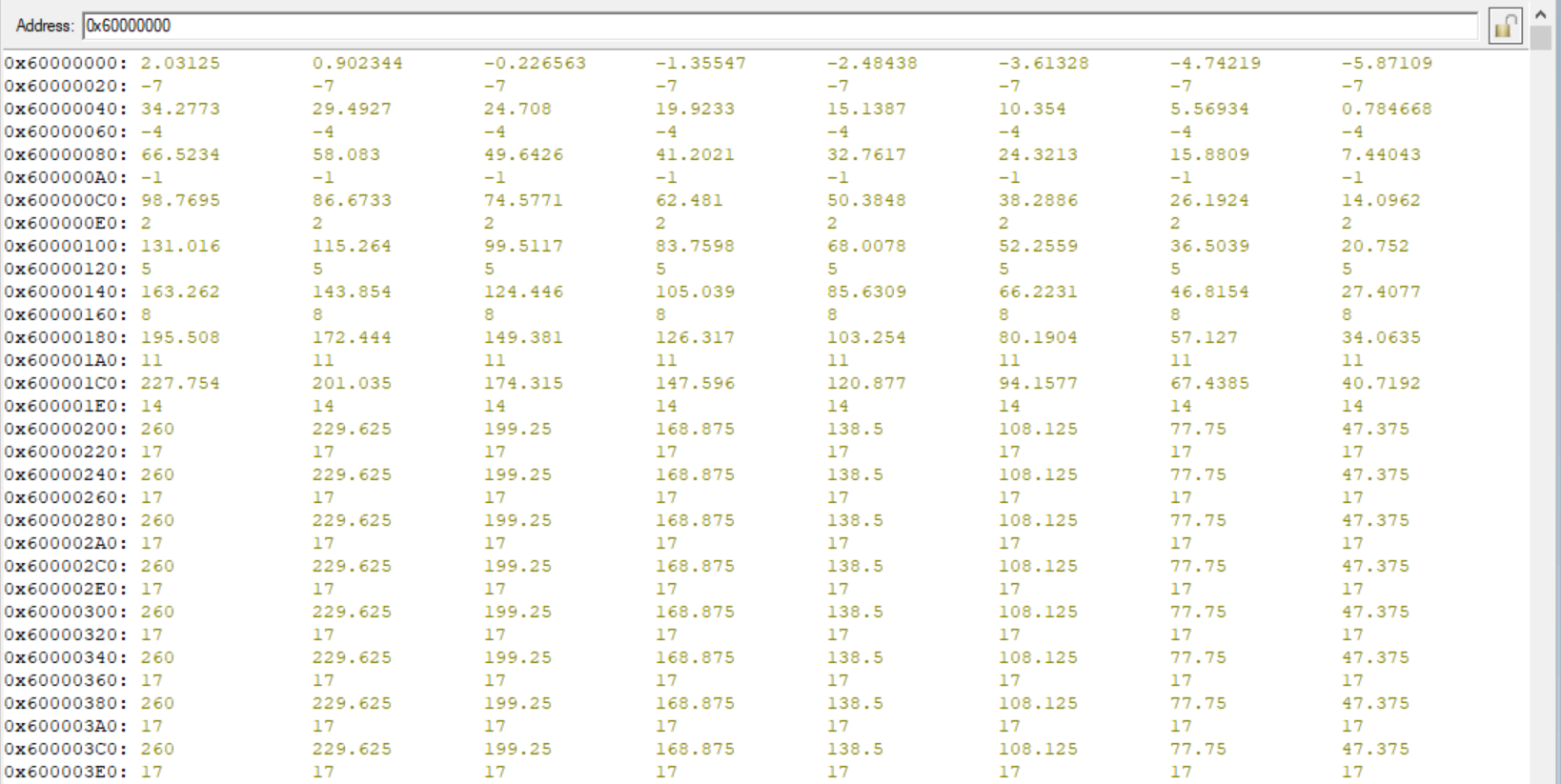
DCD 3 = k

DCD 2\_01000000000000100000000000000000 = 2.03125

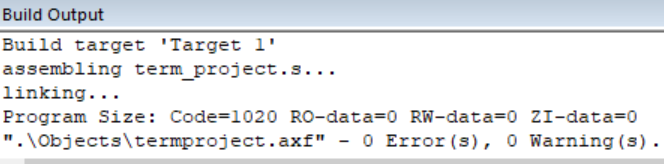
DCD 2\_11000000111000000000000000000000 = -7

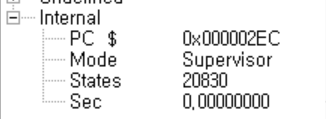
DCD 2\_01000011100000100000000000000000 = 260

DCD 2\_01000001100010000000000000000000 = 17

interpolated\_output\_data\_decimal

performance





1. Conclusion
   1. 고찰

처음 과제를 이해하고 생각하는데 만 하루가 걸렸다. 행렬의 2배 4배 8배 확장할 때 2^k개로 만 확장하는 것을 보고 2배로 확장하는 행렬을 2배는 1번, 4배는 2번, 8배는 3번하는 방식으로 구현하면 되겠구나 생각할 수 있었다. 문제는 2배를 어떻게 확장시키는 함수를 구현할 수 있는가였다. 처음에는 행렬에서 bilinear interpolation하는 4개의 input을 받아와서 9개의 output을 만들고 차례대로 만들려고 생각했었는데, 4개를 입력 받는것 뿐만 아니라, 9개의 숫자를 저장하고 다시 불러오는데 문제가 생겼다. 그래서 reset하여 새롭게 시작하였다. 다음 방법은 가로 먼저 2배로 확장하고 세로를 확장하는 방법으로 구현할 수 있었다. 다음과 같이 구현하다 보면 2배를 한 행렬이 행과 열이 바뀐 형태인 전치행렬과 같이 나타난다. 이 행렬을 다시 행과 열을 바꾸어 60000000번지에 저장해주면 되지만, 생각했을 때 보기 불편할 것 같아 가로에서 세로의 data를 받아오는 것이 아니라, 세로를 먼저 두배 해주고 그 행렬의 행을 읽어 행렬을 두배 할 수 있었다.

SAVE\_data data를 저장하고 읽어올 수 있다. 따라서 SAVE\_data 90000번지에 있는 값을 레지스터처럼 사용이 가능하다고 할 수 있다. 어셈블리어에서는 register의 개수가 한정되어 있기 때문에 SAVE\_data처럼 읽고 저장하여 register의 개수가 한정적인 부분을 보완할 수 있다.

* 1. 결론

State를 줄이는 방법

성능의 기준은 code size와 state2를 곱한 값이다. State가 제곱이기 때문에 state가 줄어들수록 성능이 좋아진다고 볼 수 있다. 일단 생각하는 대로 코드를 구현하여 알맞은 결과값을 출력해내면 그 때 코드를 분석하여 같은 연산을 하는 부분은 함수로 만들어 주어 code size와 state를 줄일 수 있다. 구현한 코드에서는 4배, 8배를 구현하는 부분이 따로 없고 2배를 구현하는 함수만 있다. 따라서 2배한 code에서는 padding하여 N-1과 N이 같은 값이 나오게 된다. 이 두 수의 bilinear interpolation은 따로 계산할 필요없이 N-1과 N이 같은 값이므로 중점은 r7=r1=r2이라는 것을 알 수 있다. 그래서 cal연산을하지 않고 바로 값을 출력, 저장할 수 있다.