1. 과제 제목

정렬 알고리즘

2. 과제 목표

삽입 정렬, 버블 정렬, 선택 정렬을 구현한다. 16진수로 표현된 정수를 32-bit floating point로 저장하고, 종류별 정렬알고리즘을 사용하여 내림차순으로 정렬하는 동작을 한다. 각 정렬 알고리즘의 동작을 확인하고 성능비교를 진행한다.

3. 각 function 별 알고리즘 (floating 포인트 변환 포함)

(1) 부동소수점 변환

16진수로 나타낸 정수를 부동소수점으로 나타내는 과정은 다음과 같다.

Ex) 0x6b8643ff

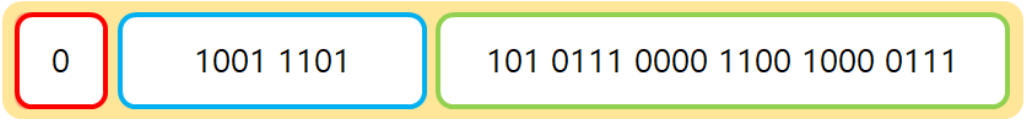
(1) 0x6b8643ff = 0110 1011 1000 0110 0100 0011 1111 1111_2

(2) = 1.10 1011 1000 0110 0100 0011 1111 1111 * 2^30

(3) Sign = MSB

Exponent = 30+127 = 157 = 1001 1101

Mantissa -> 23bit



1) 16진수 정수를 2진수 정수로 나타내기

메모리와 레지스터에 값이 2진수로 저장되기 때문에 이 부분은 생략된다.

2) 2진수 정수를 1.xxxxx와 지수의 곱 형태로 변형하기

32bit 값에서 sign bit인 MSB를 제외하고, 1.xxxx형태로 변형했을 때 가능한 가장 큰 지수의 값은 30이다. exponent bit의 값은 지수에 127을 더한 것과 같으므로, 가장 큰 exponent는 157이다. loop_first1 label의 뒤에서 exponent값을 하나씩 줄이며 가장 처음으로 나오는 1을 찾는다. 이 과정을 통해 찾은 1의 이후 최대 23bit의 값은 mantissa가 된다.

3) 앞에서 구한 sign, exponent, mantissa를 연결하여 부동소수점을 구한다.

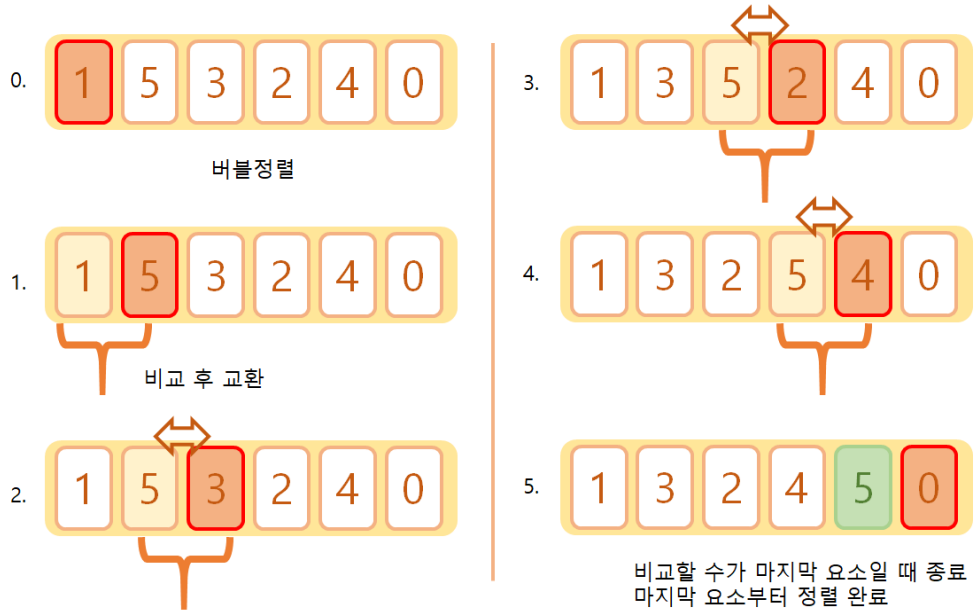
(2) 부동소수점으로 나타난 값의 크기 비교

부동소수점으로 나타낸 수를 비교하는 것은 sign, exponent, mantissa의 순서로 진행된다. 가장 먼저 sign bit을 비교하여 양수와 음수를 비교한다. 두 수의 sign bit이 다른 경우 sign bit가 1인 수가 더 작다. 두 수의 sign bit이 동일한 경우 exponent를 비교한다. 이후의 비교에서 수의 부호에 따라 비교 결과가 바뀐다. 따라서 exponent와 mantissa를 비교할 때는 절댓값을 고려한다. exponent는 1.xxxx에 곱해진 지수에 127을 더한 값과 같으므로, exponent의 값이 큰 수의 절대값이 더 크다. 비교하는 수의 exponent값도 일치할 경우 mantissa를 비교한다. mantissa도 exponent와 동일하게 값이 더 큰 수의 절대값이 더 크다. 이 부분은 모든 정렬 알고리즘에서 공통적으로 이용되었다.

(3) 삽입 정렬

삽입 정렬은 자료의 맨 앞부터 정렬할 자료를 선택하고, 처음부터 선택한 자료 앞까지의 범위에서 해당 자료를 삽입할 위치를 찾아 삽입한다. 이번 프로젝트에서는 오름차순으로 자료를 정렬하므로, 주어진 범위에서 선택한 자료보다 큰 값이 나타난다면 그 앞에 자료를 삽입하고 뒤의 내용을 미루어 정렬한다.

(4) 버블 정렬



버블 정렬은 자료의 맨 앞부터 시작하여 연속하는 자료의 순서를 비교한다. 더 뒤에 위치해야 하는 자료가 뒤에 위치하도록 두 자료의 위치를 교환한다. 이 과정을 자료의 마지막까지 반복한다. 처음부터 끝까지 교환이 종료되면 가장 마지막의 요소가 정렬 완료된다. 이후 다시 자료의 맨 앞부터 순서비교를 반복한다. 자료의 가장 앞에 위치한 정보까지 정렬되면 전체 정렬을 종료한다.

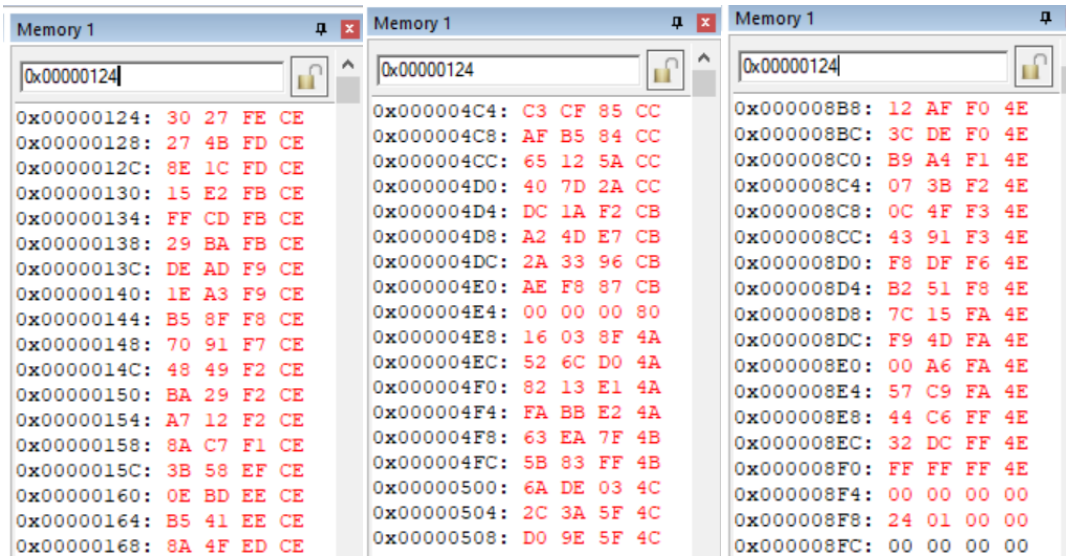
(5) 선택 정렬

선택 정렬은 자료의 맨 앞부터 정렬할 자료를 선택하고, 선택한 자료 이후의 자료와 순서대로 비교하여 가장 작은 수와 자리를 바꾼다. 값을 비교하는 과정은 비교할 수가 0일 때 종료되고, 전체 정렬 과정은 정렬하기 위해 선택한 자료가 마지막 자료일

때 종료된다.

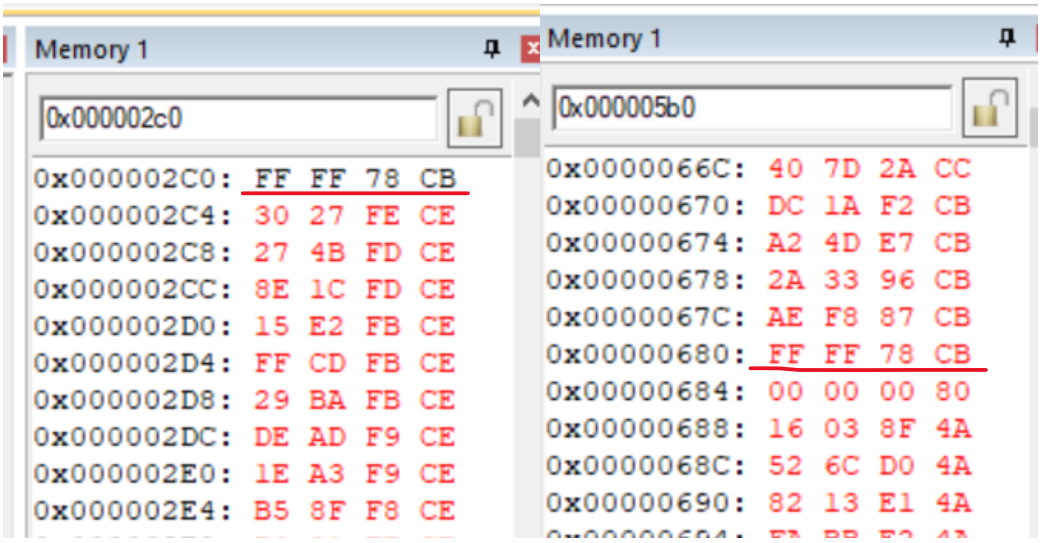
4. 검증 결과

정렬 결과는 다음과 같다. 세 종류의 정렬 결과는 동일하게 나타났다.



주어진 data.txt 파일의 내용을 이용하여 작성한 프로그램을 실행했다. BL 명령어를 이용하여 서브루틴에서 부동소수점 변환과 정렬이 이루어지고, 정렬에 해당하는 명령어를 삽입 정렬, 버블 정렬, 선택 정렬로 바꾸어 세 번 실행하여 결과값을 얻었다. 각각의 서브루틴 실행 후 state는 다음과 같았다.

부동소수점 변환	삽입 정렬	버블 정렬	선택 정렬
27339	1803873	3755418	2982041



제안서에서 검증전략으로 제시한 정렬 후에 하나의 요소를 새로 삽입하여 재정렬 하는 경우는 다음과 같이 설계하였다. 위에서와 동일하게 data.txt의 내용을 한 번 정리한 뒤, (1) 해당 자료의 마지막을 의미하는 0x00000000의 위치에 임의의 값 0xCB78FFFF를 삽입하고, 그 다음에 0x00000000를 삽입하여 자료의 마지막을 표시한다. 이후 정렬을 다시 실행한다. (2) 자료의 가장 처음에 임의의 값인 0xCB78FFFF을 저장한 뒤 재정렬 한다. 그 결과는 다음과 같이 나타났다.

	삽입 정렬		버블 정렬		선택 정렬	
소수점	27339	차이	27339	차이	27339	차이
정렬	1803873	+1776534	3755418	+3728079	2982041	+2954702
(1) 재정렬 (뒤)	4143907	+2340034	7836847	+4081429	5935939	+2953898
(2) 재정렬 (앞)	4134353	+2330480	7854880	+4099462	5861478	+2879437

5. 결과에 대한 고찰

실험 결과 가장 빠른 실행속도를 보인 것은 삽입 정렬이고, 선택 정렬, 버블 정렬 순서로 좋은 성능을 보였다. 특히 버블 정렬은 삽입 정렬의 약 2배에 해당하는 state가 사용되었다. 이 결과는 제안서에서 예상했던 것과 다르다. 버블 정렬은 하나의 요소에 대해 전체 자료를 살펴보고 각각의 비교 결과에 따라 값을 교환하는 계산이 반복적으로 나타나기 때문에 가장 느릴 것이라는 예상은 맞았지만, 정렬 과정에서 변동이 있을 때 모든 값을 한 칸씩 미루는 계산을 해야 하는 삽입 정렬보다 값을 교환하는 선택 정렬이 빠를 것이라는 예상은 틀렸다. 그 이유로는 선택 정렬의 비교횟수가 더 많은 것을 이유로 들 수 있을 것이다. 삽입 정렬의 경우 삽입할 수와 자료의 앞에 위치한 값을 비교하여 중간에 삽입할 자리를 찾으면 비교를 중단하고 값을 삽입한다. 반면 선택 정렬은 값을 비교할 수와 그 뒤에 위치한 모든 값을 끝까지 비교하고 그중 가장 작은 값과 교체한다. 또한 예상에서 근거로 들었던 삽입 정렬의 한 칸씩 미루는 작업은 값을 삽입할 자리부터 기준이 된 자리 까지만 미루기가 실행되기 때문에 필요한 계산량이 상대적으로 적었던 것으로 생각된다.

하나의 수를 추가하여 배열을 재정렬하는 경우에도 삽입 정렬, 선택 정렬, 버블 정렬 순서로 빠른 것으로 나타났다. 이때 삽입 정렬의 state 증가량이 다른 정렬에서보다 크게 나타났다. 선택 정렬의 경우 자료의 앞에서부터 모든 값을 확인하여 순서에 맞는 것을 찾기 때문에 배열의 정렬여부와 관계없이 비슷한 state가 사용된 것을 확인할 수 있다. 실험(1)에서 삽입 정렬의 경우 앞에서부터 요소를 하나씩 선택하여 자료의 가장 앞부터 확

인하는 과정을 새 요소가 추가된 위치인 가장 마지막까지 반복한다. 이때 배열은 이미 정렬되어 있으므로 어떤 요소에 대해 적절한 삽입 위치를 찾는 계산량은 최대가 된다. 버블 정렬의 경우 자료의 가장 앞에서부터 이웃한 자료와 크기를 비교하는 수는 자료의 크기가 하나 증가했기 때문에 약간 늘어났을 것이다. 값을 교체하는 것은 가장 뒤에 추가된 요소가 적절한 자리에 위치하게 될 때까지 앞으로 이동할 때만 발생한다. 가장 앞에 값을 추가한 실험(2)의 경우에도 결과는 유사하게 나타났다.

이번 과제는 세 가지 정렬 알고리즘을 구현하는 과제였다. 정렬한 데이터를 살펴보았을 때 오류는 발견하지 못했지만, 검증 과정에서 몇 가지 아쉬움이 남는다. 자료의 가장 앞에 값을 삽입한 실험에서 삽입한 값의 정렬이 종료되었을 때의 state를 기록하지 못한 점이 아쉽다. 각 알고리즘 별로 해당 값이 정렬된 이후 종료하는 것을 구현하는데 실패하였기 때문이다. 예상되는 문제점이었던 코드 구현의 어려움은 부동소수점 전환과 정렬알고리즘을 작성할 때는 발생하지 않았지만, 각각의 정렬 서브루틴이 실행된 이후에 레지스터가 갖게 되는 정보가 각자 달라 서브루틴 실행 이후 공통적인 조작을 통해 검증하는 것에 어려움을 겪었다. 프로그램에서 어떤 부분을 대체할 수 있는 부분을 작성하는 경우 그 결과값의 형태를 확실하게 정해두고 그것을 준수하는 것이 필요하다는 것을 알 수 있었다.

6. 참고문헌

이형근교수님, 어셈블리프로그램설계및실습 강의자료, 광운대학교 컴퓨터정보공학과, 2020

keil, BX,

https://www.keil.com/support/man/docs/armasm/armasm_dom1361289866466.htm

keil, Condition code suffixes,

https://www.keil.com/support/man/docs/armasm/armasm_dom1361289860997.htm

keil, area

https://www.keil.com/support/man/docs/armasm/armasm_dom1361290002714.htm