1. 배경지식

(1) 배열과 문자열

배열은 같은 크기를 갖는 여러 개의 값들을 연속되는 주소에 저장할 때 사용하는 자료구조이다. 어셈블리에서 배열을 생성하면 지정된 값을 배열에 저장한 뒤, 배열에 첫 번째로 저장된 요소의 주소 값을 반환해준다. 이 주소 값에 배열에 저장한 각 요소의 크기와, 찾으려는 값이 저장된 순서(n)를 곱하여 더하면 배열의 n번째 요소에 접근할 수 있다. 배열에 저장한 값이 문자라면 그 배열은 문자열이 된다. 문자열의 경우 마지막에 0을 저장하여 문자열의 끝을 표시한다는 차이점이 있다. 문자열도 다른 배열과 동일하게 주소 값을 이용하여 특정 문자에 접근할 수 있다.

(2) Branch와 반복문

branch문은 프로그램의 흐름을 제어하는 역할을 한다. 위에서 아래의 순서대로 코드를 실행하다가 branch문이 실행되면 branch문에서 지정한 label의 위치로 이동하여 그 이후의 코드를 실행시킨다. 이러한 동작은 C언어의 goto문과 일치한다. 프로그램의 모든 문장을 실행하지 않는다는 점에서는 조건문과 유사하지만, branch문은 기본적으로 조건을 검사하지 않고 label로 이동한다는 차이점이 있다. branch문에 조건flag를 붙여서 BEQ와 같이 사용하는 것도 가능하다. branch문을 이용해서 반복문을 구현할 수 있다.

(3) Strcmp

strcmp()함수는 인자로 두 개의 문자열의 주소 값을 전달받아, 두 문자열의 내용을 비교하는 동작을 수행하는 함수이다. 문자열 끝의 0을 만날 때까지 동작하므로 인자로 전달되는 문자열의 마지막에는 0이 저장되어 있어야 정상적으로 동작한다. 인자로 전달된 두 개의 문자열 중 첫 번째 문자열에서 두 번째 문자열을 뺄셈하는 것과 비슷하게 값을 반환한다. 두 문자열이 일치하는 경우 0을 반환하고, 첫 번째 문자열이 더 작은 경우 0보다 작은 값을, 더 큰 경우에는 0보다 큰 값을 반환한다. 이번 과제에서는 문자열의 내용이 같은 경우 10을, 다른 경우 11을 저장하는 동작을 수행하기로 한다.

(4) LSL

LSL은 bit stream을 왼쪽으로 이동시키는 연산이다. 2진수 표현에서 한 자리 왼쪽에 위치하는 수는 원래 수의 2배이므로, 수에 2를 곱한 것과 유사하게 동작한다. 가장 오른쪽에는 0이 새로 생성되고, 가장 왼쪽에서는 지정된 비트 수를 넘어선 비트가 잘리게 된다. Bit stream을 오른쪽으로 이동시키는 연산에는 LSR과 ASR이 있다. 이 경우 수에 2를 나눈 것과 유사하게 동작한다. 가장 오른 쪽에 있는 비트는 삭제되고, 가장 왼쪽에 새로 생성되는 비트는 두 명령어에서의 동작이 다르다. LSR은 LSL과 같이 가장 왼쪽의 비트를 0으로 채우고, ASR은 MSB와 같은 비트로 생성한다. ASR에서 사용하는 방식은 연산 후에도 수의 부호를 변화시키지 않는다. [i]

(5) Unrolling

Unrolling은 반복문에서 반복되는 실행문의 일정량을 직접 풀어쓰고, 반복문의 반복 횟수를 줄이는 것이다. 하나의 명령을 10번 반복하는 반복문이 있다면, 다섯 개의 동일한 명령을 두 번 반복하는 것으로 바꾸는 것이다. 반복문에서 조건을 비교하는 횟수가 줄어들어서 속도가 빨라질 수 있지만, 코드의 길이가 길어진다.

2. 코드내용

(1) Problem 1 – strcmp.s

r0과 r1에 각각 문자열이 저장된 주소 값을 저장하고, r3에는 결과를 저장할 메모리 주소를 저장한다. Loop라벨 이후에서 두 개의 문자열에서 각각 한 비트를 읽어 r4와 r5에 저장하고, 비교한다. NE라벨을 이용하여 두 값이 다른 경우 Loop를 빠져나가 r6의 값에 0x0B을 저장한다. 두 값이 같은 경우 B Loop를 이용하여 다시 Loop라벨 이후로 돌아간다. 각 자리 수에 대해 비교하는 과정을 반복한 후, 문자열의 마지막인 0까지 일치한다면 r6에 0x0A를 저장한다. 이후 r6에 저장된 값을 r3을 이용해 4000번지에 저장한다.

(2) Problem 2 – sort.s

R0에 문제에서 제시된 배열을 저장하고, r1에는 결과를 저장할 메모리 주소를 저장한다. R2에는 Loop를 반복할 횟수를 저장한다. Loop라벨 이후에서 r2에 저장된 반복 회수를 하나 빼고, 그 값을 r1에 저장된 주소 값에 더하여 메모리의 뒤에서부터 숫자를 하나씩 저장한다. 반복회수가 0이 되면 Loop를 벗어나고, 아닌 경우 위의 과정을 반복한다. 이 프로그램에서 r2의 값은 Loop를 반복하는 횟수이며 동시에 수가 메모리에 저장되는 순서를 의미한다.

(3) Problem 3

- Oddsum1.s

R0에 1을 저장한 뒤, LSL연산을 이용해 11을 만든다. 1을 왼쪽으로 한 비트 이동해서 2를 만들고, 2를 왼쪽으로 두 비트 이동해서 8을 만든 뒤, 1, 2, 8을 모두 더해 11을 구할 수 있다. Loop를 반복할 횟수 10을 r4에 저장하고, 결과를 저장할 메모리의 주소 값을 r5에 저장한다. 홀수들의 합을 구할 r6에 11을 저장하고, Loop 라벨 이후에서 더하기 계산을 한다. 더하기 계산은 새로 더해질 수를 구하는 계산 (16번째 줄)과, 그 수를 총합에 더하는 계산(17번째 줄)로 두 번 이루어진다. 이후 반복횟수를 확인하여 Loop를 반복하거나 메모리에 결과를 1word 크기로 저장한다.

- Oddsum2.s

R0에 1을 저장한 뒤, LSL연산을 이용해 10을 만든다. 2와 8을 만드는 과정은 oddsum1.s에서 이용한 방법과 같다. 2와 8을 더해 10을 만들고, r3에는 결과를 저장할 주소 값을 저장한다. R1에 n

값인 10을 저장하고, n+10을 계산하여 r2에 저장한다. MUL 연산을 이용하여 r1과 r2의 값을 곱하여 n(n+10)을 계산한 뒤, 결과를 1halfword 크기로 4000번지에 저장한다.

- Oddsum3.s

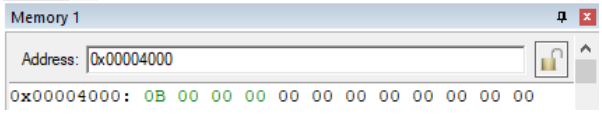
더해질 수 11, 총합 0, 반복횟수 2를 각각 r0, r1, r2에 저장하고, 결과를 저장할 메모리 주소값을 r3에 저장한다. Loop라벨 뒤에서 r1에 r0 값을 저장하고, r0에 2를 더한다. 이것을 4번 더 반복하여 총 5회 실행한 후, 남은 반복횟수를 확인한다. 반복횟수가 남았다면 다시 한 번 Loop라벨 이후 내용을 실행하고, 아닌 경우 총합을 1byte 크기로 4000번지에 저장한다.

3. 결과

(1) Problem1

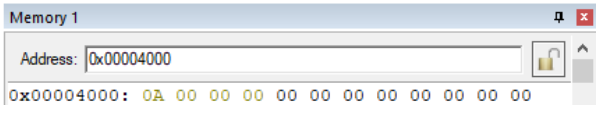
일치하치 않는 경우

```
25  TEMPADDR1 & &00004000
26  string00 DCB "12345",0
27  string01 DCB "12345678",0
```



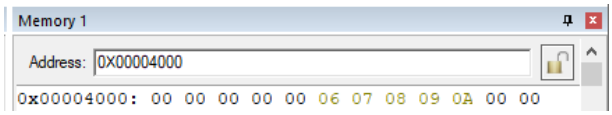
일치하는 경우

```
25  TEMPADDR1 & &00004000
26  string00 DCB "12345",0
27  string01 DCB "12345",0
```

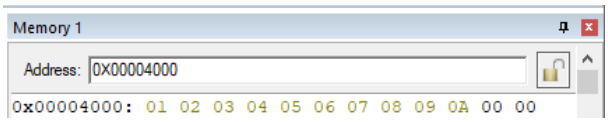


(2) Problem2

저장과정

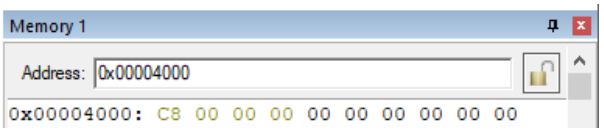


저장완료

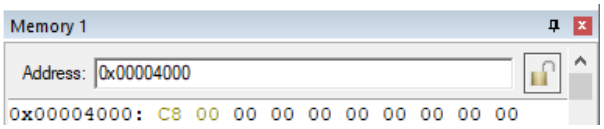


(3) Problem3

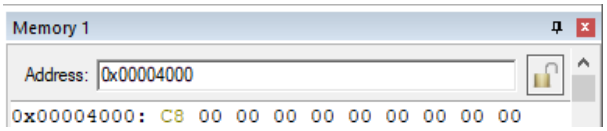
1. oddsum1.s



2. oddsum2.s



3. oddsum3.s



4. problem3 각 방법의 성능 비교

problem3을 해결하는 세 가지 방법 중 n(n+10)을 이용한 방법이 가장 빠르게 실행되고, Loop만을 이용한 방법이 가장 느리게 실행될 것이라 생각한다. N(n+10)의 속도가 가장 빠를 것이라고 생각한 이유는 실행되는 명령어의 수가 가장 적기 때문이다. 반면 Loop만을 이용한 방법이 Unrolling을 이용한 방법보다 느릴 것이라고 생각한 이유는 Loop에 의해서 계산되는 ADD의 수는 두 경우가 같지만, Loop에서 반복 조건을 확인하는 횟수가 Loop만을 이용한 경우에는 10번이지만, Unrolling을 사용하면 2 번으로 적어져서 연산량이 줄기 때문이다.

5. Branch와 conditional execution의 차이점과 성능차이

Branch와 conditional execution 모두 실행문을 전부 실행하지 않고 뛰어넘을 수 있다는 공통점이있다. 그러나 Branch는 프로그램 코드의 특정 부분으로 이동하여 이미 실행된 명령을 다시 수행할 수 있으며, Branch명령어에 조건flag를 붙이지 않았다면 조건을 고려하지 않고 해당 라벨 이후로 이동한다. 반대로 conditional execution는 조건을 확인하고 조건에 해당하지 않는 경우에만 연산을 뛰어넘는다.

두 경우를 비교했을 때 Branch를 이용한 방법은 조건을 비교하는 과정이 줄어들고, conditional execution을 이용한 방법은 특정 라벨을 찾아가는 과정을 줄일 수 있다. 이를 통해 Branch를 이용하는 방법이 대부분의 경우 더 빠를 것이라고 추측할 수 있다. 특정 라벨을 찾는 과정의 횟수는 각 실행문에서 조건을 비교하는 횟수보다 훨씬 적기 때문이다. 만약 라벨을 찾는 과정의 횟수가 조건을 비교하는 횟수보다 매우 많은 경우가 있다면 반대로 conditional execution의 성능이 더 좋을 수도 있을 것이라고 생각한다.

6. 고찰

(1) Problem1

두 문자열의 내용이 다를 때 0x0B가, 같을 때 0x0A가 4000번지에 정상적으로 저장되었다. 문자열의 내용이 다를 때뿐만 아니라 문자열의 길이가 다른 경우에도 정상적으로 비교되는 것을 확인할 수 있다. 배열에 대해 이해하고 branch를 이용한 반복문을 이용하는 것이 필요했다.

(2) Problem2

배열에 저장된 순서대로 각 요소가 메모리의 끝부분부터 저장되는 것을 확인할 수 있었다. 모든 요소가 저장된 후에 메모리를 확인해보면, 4000번지부터 1,2,3,4로 이어지는 오름차순으로 적절히 저장된 것이 보인다. 배열에 저장된 순서대로 메모리의 뒤쪽에서부터 값을 저장했지만, 반대로 메모리에 저장할 순서대로 배열의 마지막부터 요소에 접근하는 방법을 사용할 수도 있을 것 같다.

(3) Problem3

세 가지 경우 모두에서 11부터 29까지 홀수의 합인 200(C8)이 저장된 것을 확인할 수 있었다. LSL 연산을 통해 11을 만드는 과정이 특이했다. 또한 Unrolling이라는 것을 처음 알게 되었는데

반복되는 코드의 양이 늘어나지만 그것을 통해 속도에서 이득을 볼 수 있다는 특징이 인상깊었다. 그러나 프로그래머가 코드를 이해할 수 있는 정도나, 메모리에 저장하게 되는 코드 자체의 길이도 신경을 써서 프로그램을 작성하는 것이 좋을 것이라고 생각했다.

세 가지 문제에서 공통적으로 배열을 사용하였는데, 레지스터에 배열의 주소 값을 저장할 때 '=' 기호를 누락시켜서 메모리에 접근할 수 있는 권한이 없다는 메시지가 나타났었다. '='기호를 쓰지 않으면 레지스터에 다른 값이 저장되는 것 같지만 자세한 내용을 찾지 못해서 아쉬움이 남았다.

---

[i] LSR, https://www.keil.com/support/man/docs/armasm/armasm_dom1361289876525.htm

ASR, https://www.keil.com/support/man/docs/armasm/armasm_dom1361289863407.htm