1. 배경지식

(1) Second operand [i]

어셈블리에서는 LSL, LSR등의 비트 시프트 연산자를 다른 연산자와 함께 사용할 수 있다. 명령어의 뒷부분에 위치하는 비트 시프트 연산자를 먼저 계산하고, 앞부분의 명령어를 수행한다. 예를 들면 다음과 같다.

```
4  start
5      MOV r0, #5 ;0101
6      ADD r1, r0, r0, LSR #1  ;r1 = 0101 + 0010 = 0111
7
8      MOV r2, #29 ; 1_1101
9      SUB r3, r2, r1, LSL #2  ;r2 = 1_1101 - 1_1100 = 0001
10
11 END
```

Register	Value
Current	
R0	0x00000005
R1	0x00000007
R2	0x0000001D
R3	0x00000001

6번째 줄의 ADD r1, r0, r0, LSR #1은 r0에 저장된 0101이 오른쪽으로 1비트 이동하여 0010이 되고, 이후에 이동되지 않은 r0의 값 0101과 더해져 0111인 7이 r1에 저장되었다.

9번째 줄의 SUB r3, r2, r1, LSL #2는 r1에 저장된 0111이 왼쪽으로 2비트 이동하여 011100이 되고, 이후 r2에 저장된 11101과 뺄셈 연산을 해서 r3에 1이 저장된다.

(2) Factorial

팩토리얼은 1부터 n까지의 모든 자연수를 곱한 값이다. 팩토리얼 계산을 할 때는 반복문이나 재귀함수를 이용할 수 있다. 반복문을 이용해 C언어로 문제를 풀면 다음과 같이 나타난다.

```
#include <stdio.h>
int main(void) {
    int res, i,j;

    for (i = 1; i < 11; i++) {
        res = 1;
        for (j = i; j > 1; j--) res *= j;
        printf("%d! = %x₩n", i,res);
    }

    return 0;
}
```

```
Mic...
1! = 1
2! = 2
3! = 6
4! = 18
5! = 78
6! = 2d0
7! = 13b0
8! = 9d80
9! = 58980
10! = 375f00
```

이 결과를 이용하여 실습의 결과가 정상적으로 출력되었는지 확인하였다.

2. 코드내용

(1) Problem 1

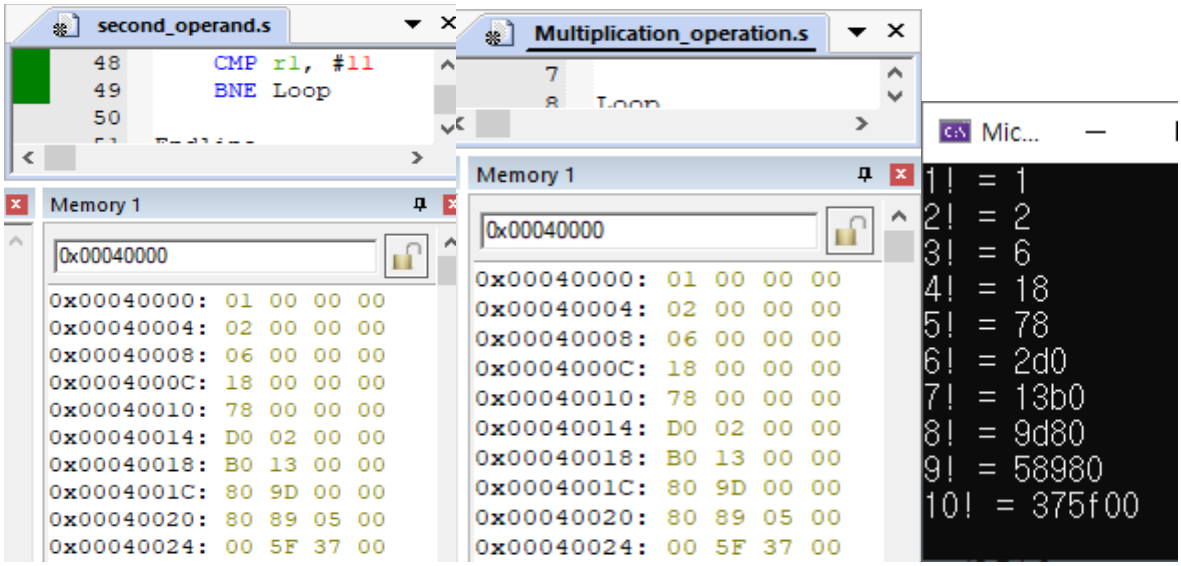
r0에 팩토리얼 계산 결과를 저장할 주소 값을 저장하고, r1에 반복문의 반복 횟수를 저장하는 값을 저장한다. r1은 팩토리얼 값을 계산하는 것에 사용되기도 한다. Loop 라벨 이후에서 연산 결과가 저장될 r2에 r1의 값을 저장한다. 이것은 n(n-1)(n-2)..의 가장 처음 n을 저장한 것이다. r1의 값을 2와 비교하여 r1이 2보다 작거나 같은 경우, 즉 r1이 1이거나 2인 경우 팩토리얼의 결과가 이미 r2에 저장되었으므로 break라벨 이후로 이동한다. r1의 값이 3이상인 경우 아래에서 2를 곱하고, 다시 크기를 비교해 break라벨 이후로 이동하거나 다음 수를 곱하는 다음 명령어로 이동한다. break문 이후에서는 팩토리얼의 결과를 메모리에 저장하고, r1에 1을 더하여 다음 팩토리얼을 구할 수를 만든다. r1이 11인 경우 문제에서 제시된 모든 수에 대한 팩토리얼 계산을 종료했으므로 프로그램을 종료한다.

(2) Problem 2

r0, r1, r2에 값을 저장하는 과정은 Problem1에서와 동일하다. 이후 r2에 곱하게 될 수인 r3에 1을 저장한 뒤 Loop 내부의 Loop2 라벨로 넘어간다. Loop2에서는 1부터 n-1까지의 자연수를 하나씩 곱하는 과정을 반복하게 된다. r2와 r3을 곱한 값을 r4에 저장한다. 이후 r3에 1을 더하여 다음에 곱하게 될 수를 계산한다. CMP 명령어를 이용하여 r1과 r3을 비교한다. 두 값이 같은 경우 직전에 곱한 값은 n-1값이었으며, 맨 처음에 n을 곱하고 시작하였으므로 팩토리얼 연산이 종료된 상태이다. 이 경우 Loop2를 탈출하여 break라벨의 아래로 이동한다. 1의 경우 예외로 CMP결과가 음수가 나온다. 이 경우에도 마찬가지로 break라벨 아래로 이동한다. 두 값이 같지 않은 경우 r4에 저장된 곱셈의 결과를 r2에 저장하고 Loop2라벨 아래로 이동하여 다시 곱셈연산을 실행한다. break라벨 아래에서 연산 결과를 저장하고, 10!까지의 계산이 종료되었는지 여부를 확인하여 프로그램을 종료하거나, 다음 팩토리얼 연산을 위해 Loop라벨 아래로 이동한다.

3. 결과

problem1 problem2 c언어로 확인한 결과



4. 고찰 - 성능 비교

problem1에서 작성한 코드를 problem2에서 MUL명령어를 이용하여 다시 작성한 것이기 때문에, 두 프로그램의 차이점은 곱셈을 연산하는 명령어이다. problem2에서 Loop2를 이용하는 것으로 코드를 더 짧게 수정하였지만, 수를 한 번 곱할 때마다 비교하게 되는 것은 problem1에서와 같으므로 큰 차이는 나지 않을 것이라 생각한다. 따라서 두 방법에 성능차이가 있다면 곱하기 연산을 수행하는 방법일 것이다.

1부터 10까지의 팩토리얼 값을 계산하는 두 가지 방법 중 Second_operand를 이용한 probelm1의 방식이 더 빠를 것이라고 생각한다. 그 이유는 곱셈연산을 수행하는 것 보다 LSL명령어를 통해 비트를 이동시키는 연산이 더 단순하기 때문이다. LSL명령어를 사용해 모든 비트를 일괄적으로 처리할 수 있어 계산이 더 빠를 것 같다.

MUL 명령어를 이용할 때 MUL r2,r2,r3과 같이 곱셈연산의 결과를 저장할 레지스터와 곱셈에 사용하는 레지스터가 같은 경우 컴파일 과정에서 오류가 발생하였다. 이 경우 r4를 이용하여 연산결과를 다른 레지스터에 저장하는 것으로 수정하여 문제를 해결할 수 있었다. 이렇듯 MUL을 이용한 경우 사용하게 되는 레지스터의 수가 많아지는 특징이 있기도 했다.

---

[i] ARM Developer, flexible-second-operand,

https://developer.arm.com/documentation/dui0068/b/arm-instruction-reference/arm-general-data-processing-instructions/flexible-second-operand