Assignment3

1. Problem & Purpose

i. 1에서 10까지의 Factorial값을 Second operhand로 구현하고, 이를 0x40000번지에 순서대로 저장한다.

ii. Problem 1의 내용을 Multiplication operation을 사용하여 구현하고, 1과의 성능 차이를 비교한다.

iii. Branch와 conditional execution의 차이점과 성능 차이

2. Used Instruction

I. 3-1: LDR // STR // MOV // ADD // END

i. LDR Rd, operand1 : operand1의 메모리 위치의 값을 word 크기만큼 Rd에 불러온다.

ii. STR Rd, [R0, offset] : R0으로부터 offset만큼 이동한 위치에 R0의 값을 word 크기만큼 저장한다.

iii. MOV Rd operand1 : operand1에 있는 값을 Rd에 저장한다.

iv. ADD Rd, R0(, R1) : Rd에 R0와 R1을 더한 값을 저장한다. R1이 없을 경우 Rd = Rd + R0로 저장한다.

v. END : Assembly code가 끝났음을 의미하는 Instruction

II. 2-2 : LDR // STR // MOV // MUL // END

i. LDR Rd, operand1 : operand1의 메모리 위치의 값을 word 크기만큼 Rd에 저장한다.

ii. MOV Rd operand1 : operand1에 있는 값을 Rd에 저장한다.

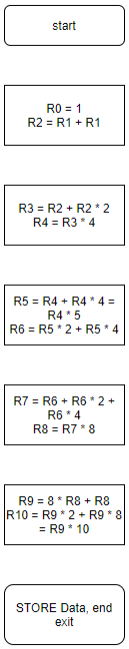
iii. CMP Rd, operand1 : Rd – operand1을 한 state를 cpsr에 업데이트한다.

iv. MUL Rd, R0, R1 : Rd에 R0와 R1의 곱셈 값을 저장한다.

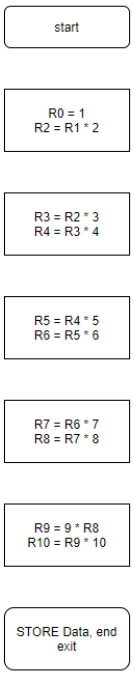
v. END : Assembly code가 끝났음을 의미하는 Instruction

3. Design(Flow chart)

i. 3-1 flow chart

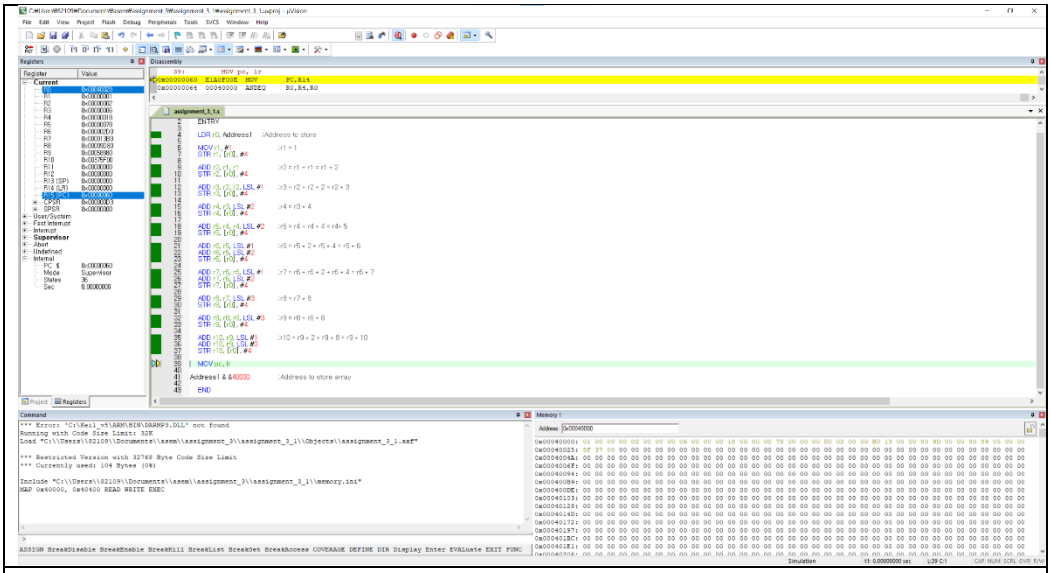


ii. 3-2 flow chart



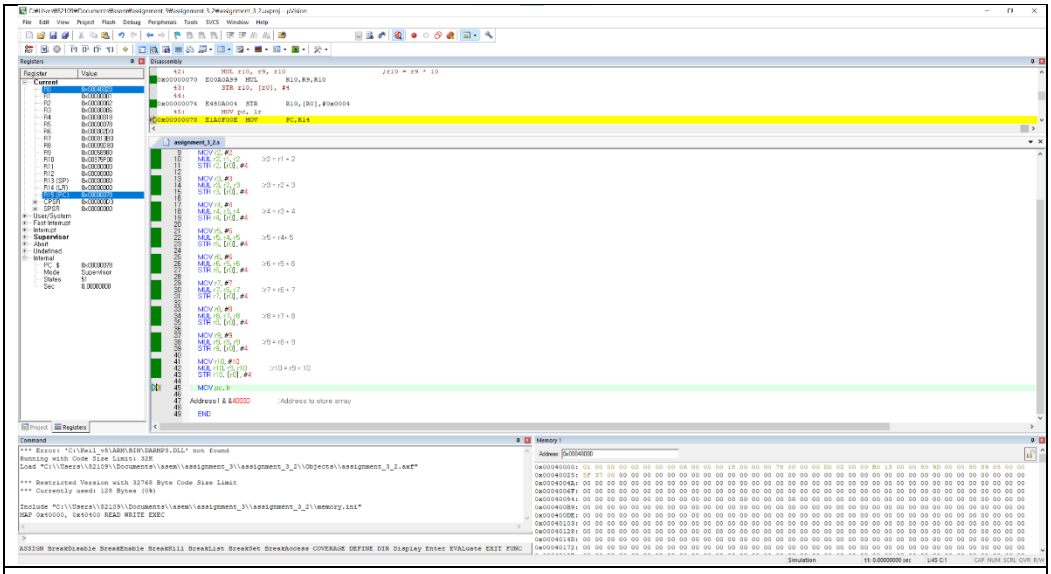
4. Conclusion

i. 2-1 result



해당 결과를 살펴보면 0x40000번지부터 데이터가 1!~10!까지 저장된 모습을 확인할 수 있다. 이 때 소요된 state는 36개다.

ii. 2-2 result



해당 결과 또한 0x40000번지부터 데이터가 위의 문제와 같이 1!~10!의 값이 정상적으로 저장된 것을 확인할 수 있다. 이 때 소요된 state는 51개다.

iii. compare

- 위의 예제들을 비교하였을 때 우리는 Second operand를 통해 값을 구할 때가 MUL을 사용할 때보다 더 적은 state를 가진 것을 확인할 수 있다. 이는 MUL을 사용할 때 더욱 복잡한 연산 과정을 요구한다는 것이다.

5. Consideration

- 위의 예제들을 비교하였을 때 우리는 Second operand를 통해 값을 구할 때가 MUL을 사용할 때보다 더 적은 state를 가진 것을 확인할 수 있다. 이는 MUL을 사용할 때 더욱 복잡한 연산 과정을 요구한다는 것이다. 다만 곱하는 수가 커질수록 Second operand는 더욱 복잡한 연산을 요구해갔다. 이를 통해 알 수 있는 것은 간단한 곱셈을 진행하는 경우 Second operand를 이용하면 더욱 빠르게 동작하도록 만들 수 있다는 것과, 특정 복잡한 숫자의 경우 Multiplication operation을 통해 연산하는 것이 빠를 수도 있다는 것이다.

6. Reference

i. 이준환 교수님/어셈블리프로그램설계및실습/광운대학교(컴퓨터정보공학부)/2021