# Assignment 4

## 1. Problem & Purpose

i. 10!의 값을 스택과 재귀 함수를 이용하여 구현한다.

ii. 주어진 그림과 같이 r0~r7의 레지스터 간의 값을 바꾸는 코드를 stack 혹은 블록 복사 명령어를 활용하여 작성한다.

iii. r0~r7의 값을 10~17로 주고 doRegister()함수와 doGCD()를 구현하여 만든 값을 메모리에 저장한다.

## 2. Used Instruction

I. 4-1: LDR // LDMFD // STR // STMFD // MOV // MOVHI // CMP // B // BHI // SUBHI // MUL // END

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

ii. LDMFD Rd(!), {operand…} : Rd의 위치에 있는 값을 Full Descending 방식으로 operand 위치에 할당한다. 만약 !가 존재할 경우 Rd의 값을 바뀐 위치로 옮긴다.

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

iv. STMFD Rd(!), {operand…} : operand의 위치에 있는 값을 Full Descending 방식으로 Rd 위치에 stack 방식으로 저장한다. 만약 !가 존재할 경우 Rd의 값을 바뀐 위치로 옮긴다.

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

vi. MOVHI Rd operand1 : CMP로 비교하였을 때 HI일 경우, operand1에 있는 값을 Rd에 저장한다.

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

viii. B operand : operand의 위치로 pc를 이동하여 작업을 수행한다.

ix. BHI operand : CMP로 비교하였을 때 HI일 경우 operand의 위치로 pc를 이동하여 작업

을 수행한다.

x. SUBHI Rd, R0 : CMP로 비교하였을 때 HI일 경우 Rd – R0를 Rd에 저장한다.

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

xii. END : Assembly code가 끝났음을 의미하는 Instruction

II. 4-2 : LDR // LDMFD // STMFD // MOV // END

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

ii. LDMFD Rd(!), {operand…} : Rd의 위치에 있는 값을 Full Descending 방식으로 operand 위치에 할당한다. 만약 !가 존재할 경우 Rd의 값을 바뀐 위치로 옮긴다.

iii. STMFD Rd(!), {operand…} : operand의 위치에 있는 값을 Full Descending 방식으로 Rd 위치에 stack 방식으로 저장한다. 만약 !가 존재할 경우 Rd의 값을 바뀐 위치로 옮긴다.

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

v. END : Assembly code가 끝났음을 의미하는 Instruction

III. 4-3 : LDR // LDMFD // STR // STMFD // MOV // MOVHI // B // BNE // BHI // ADD // END

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

ii. LDMFD Rd(!), {operand…} : Rd의 위치에 있는 값을 Full Descending 방식으로 operand 위치에 할당한다. 만약 !가 존재할 경우 Rd의 값을 바뀐 위치로 옮긴다.

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

iv. STMFD Rd(!), {operand…} : operand의 위치에 있는 값을 Full Descending 방식으로 Rd 위치에 stack 방식으로 저장한다. 만약 !가 존재할 경우 Rd의 값을 바뀐 위치로 옮긴다.

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

vi. MOVEQ Rd operand1 : CMP를 통해 비교한 값이 같을 경우, operand1에 있는 값을 Rd에 저장한다.

vii. B operand : operand의 위치로 pc를 이동하여 작업을 수행한다.

viii. BNE operand : CMP로 비교하였을 때 같지 않을 경우 operand의 위치로 pc를 이동하여 작업을 수행한다.

ix. BHI operand : CMP로 비교하였을 때 HI일 경우 operand의 위치로 pc를 이동하여 작업을 수행한다.

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

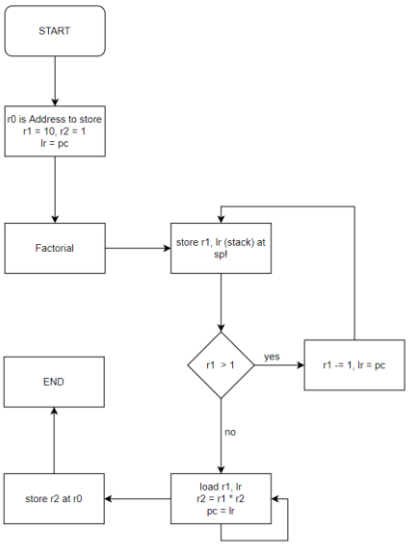
xi. ADD Rd, R0(, R1 ) : Rd에 R0와 R1을 더한 값을 저장한다. R1이 없을 경우 Rd = Rd + R0로 저장한다.

xii. SUBHI Rd, R0(, R1 ) : CMP로 비교하였을 때 HI일 경우, Rd에 R0와 R1을 뺀을 저장한다. R1이 없을 경우 Rd = Rd - R0로 저장한다.

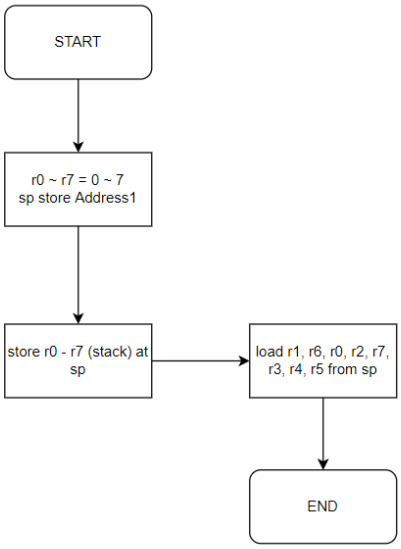
xiii. END : Assembly code가 끝났음을 의미하는 Instruction

## 3. Design(Flow chart)

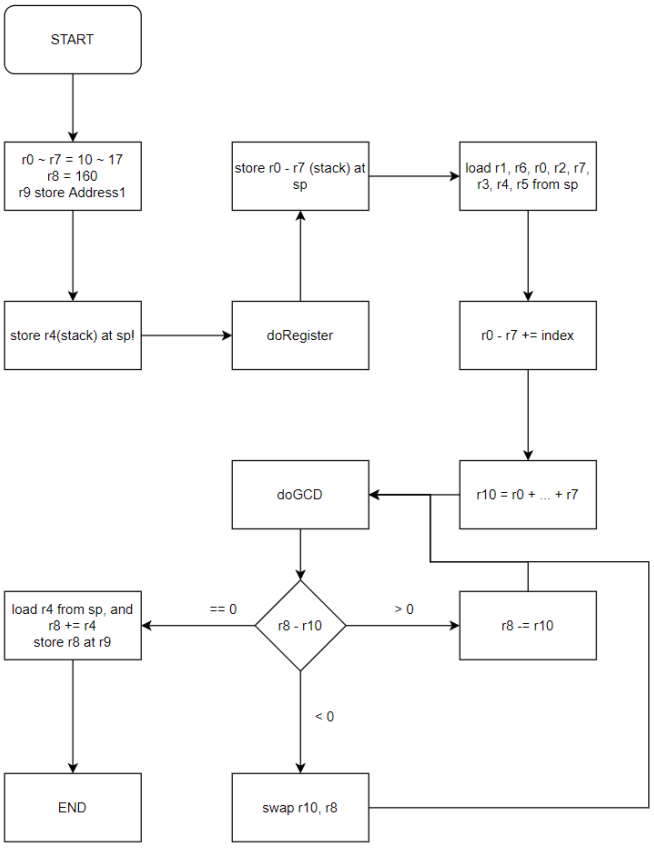
i. 4-1 flow chart



ii. 4-2 flow chart

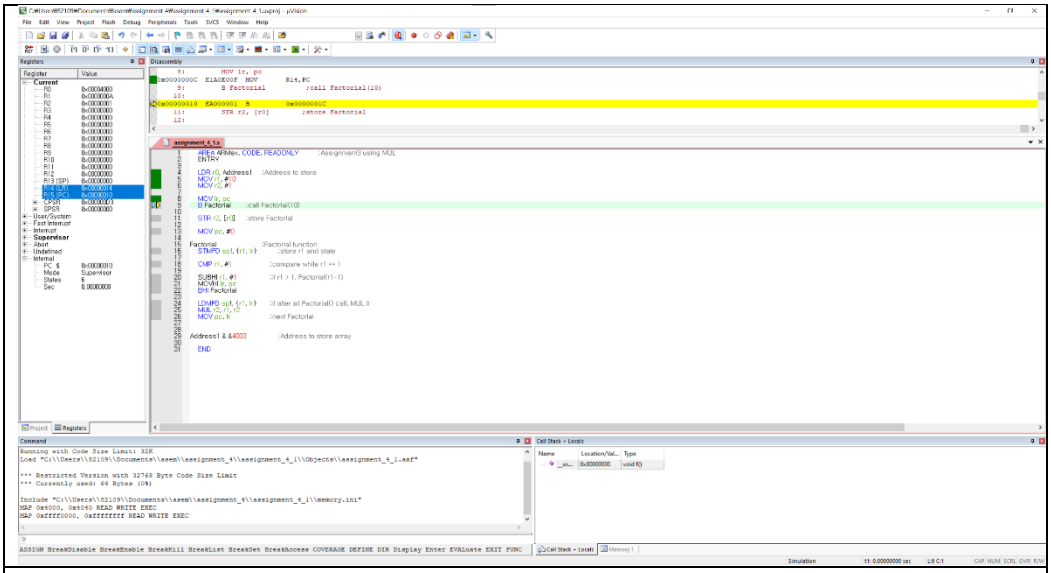



iii. 4-3 flow chart

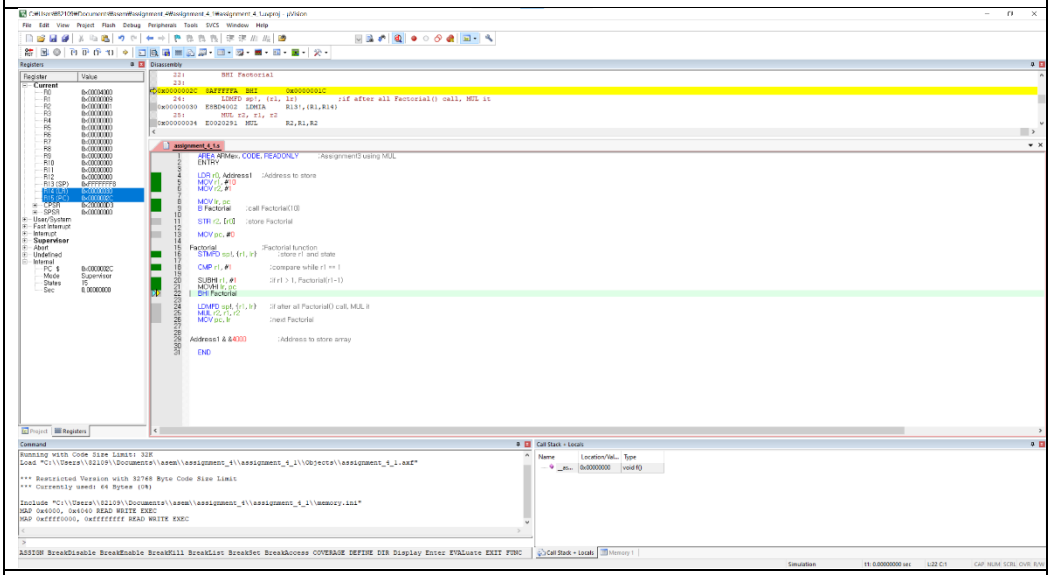


# 4. Conclusion

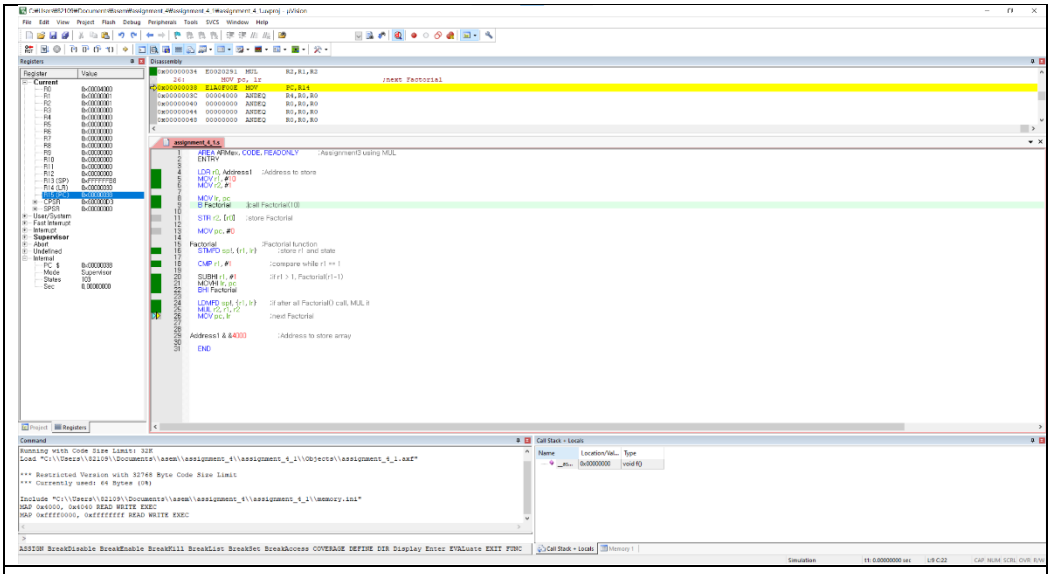
i. 4-1 result



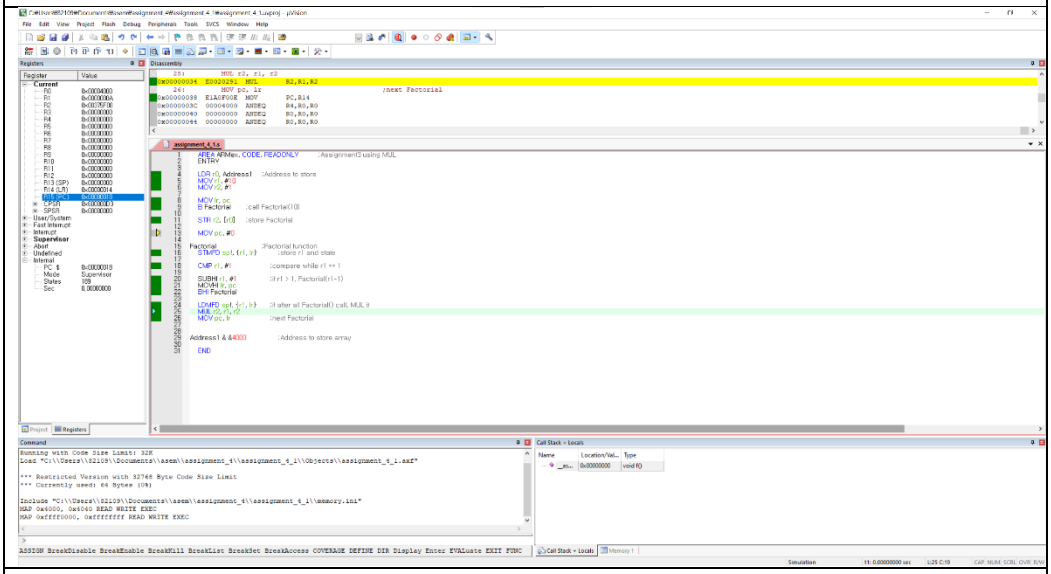
저장할 주소 r0와 10!에서 10이 r1, 그리고 r2에 곱의 결과를 저장할 r2에는 1이 저장된다. 그 후 현재 위치를 lr에 저장하고 Factorial로 Branch를 이동한다.



r1과 lr의 값을 sp에 스택으로 저장한다. 그 후 r1이랑 1을 비교하고 r1이 1보다 크면 r1을 빼고 lr의 현재 상태를 다시 저장하고 Factorial을 다시 수행한다.

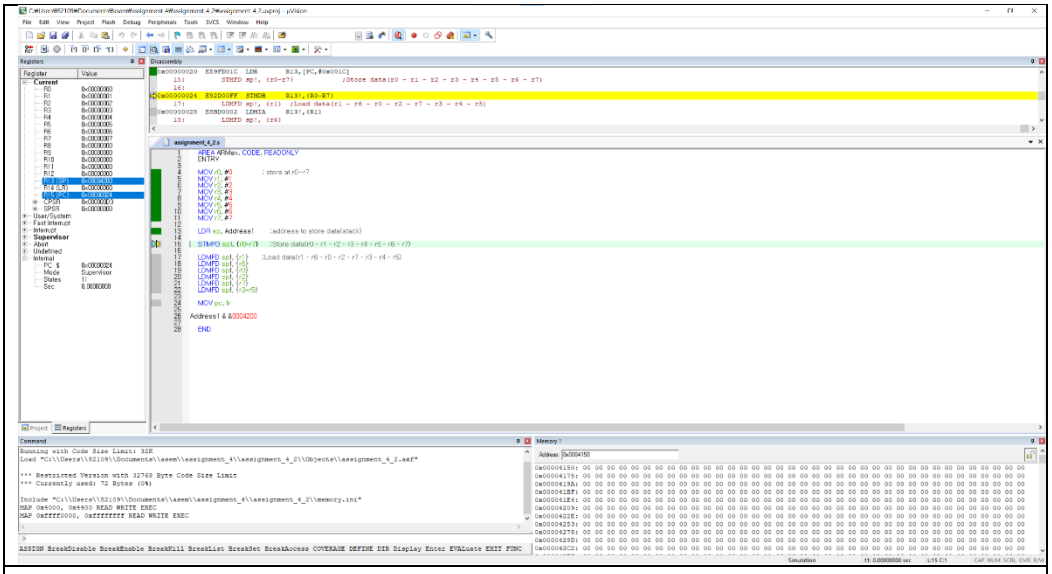


r1이 1인 경우 stack에 저장된 r1과 lr을 꺼내어 r1은 r2와 곱하여 r2에 저장하며 lr로 이동하여 이를 반복한다.

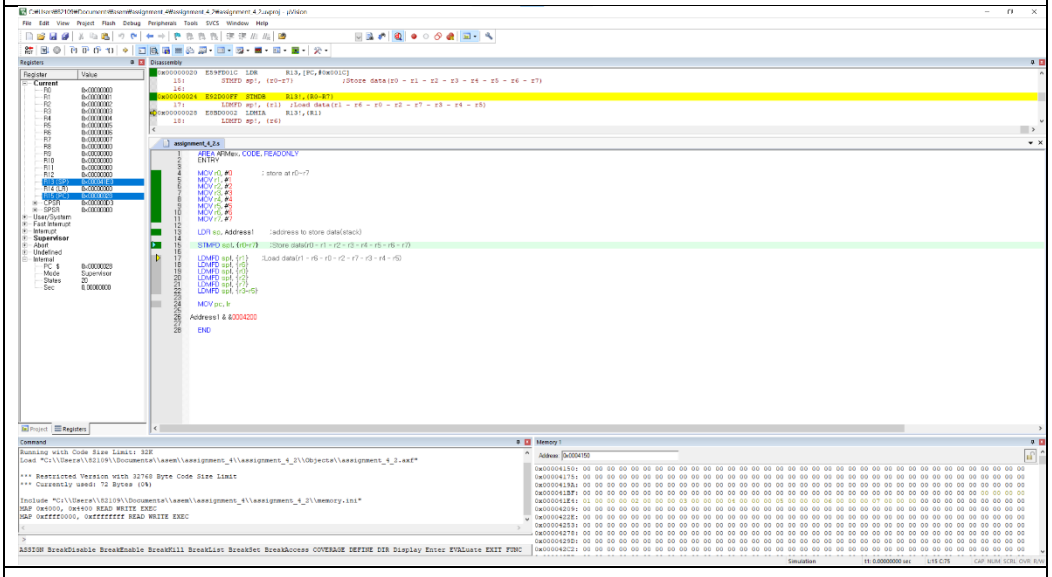


Factorial을 모두 수행하고 완성된 결과 r2를 r0위치에 저장한다.

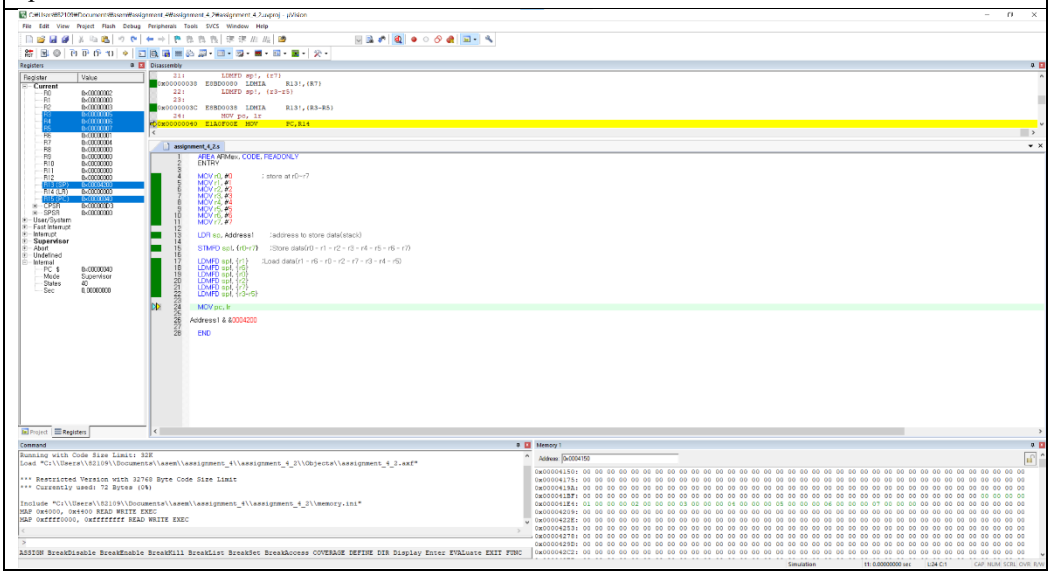
ii. 4-2 result



r0~r7에 0~7의 값을 저장하고, Address1의 위치를 sp에 저장한다.

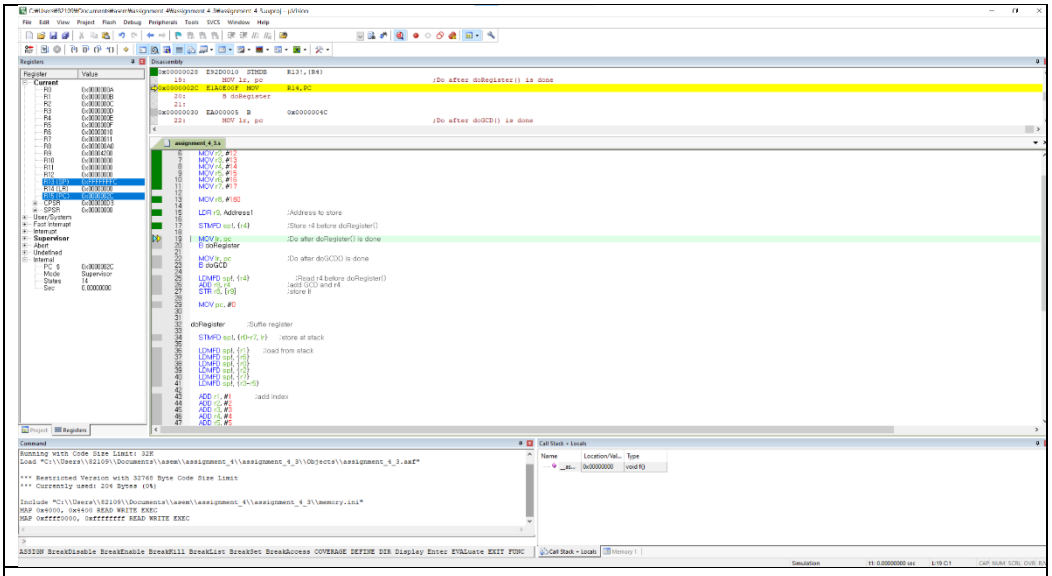


sp에 r0-r7의 값을 stack 형식으로 저장한다.

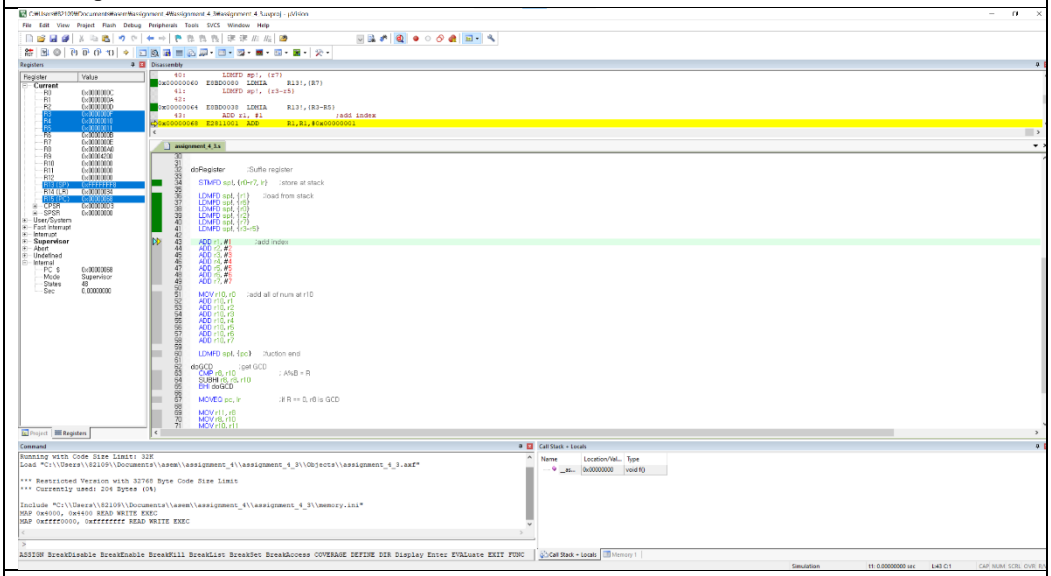


sp에 있던 데이터를 r1 - r6 - r0 - r2 - r7 - r3 - r4 - r5순으로 가져온다.

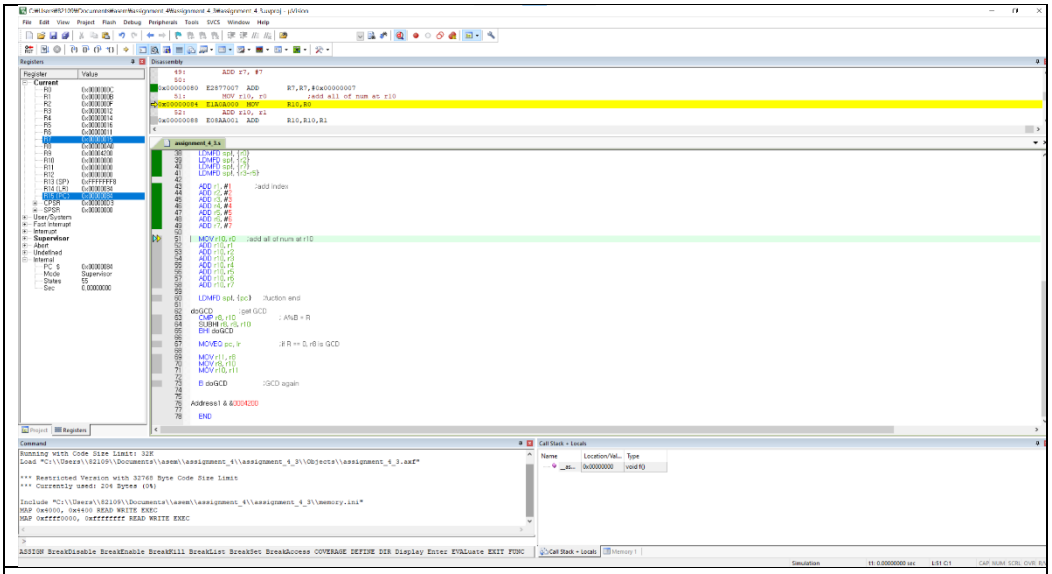
iii. 4-3 result



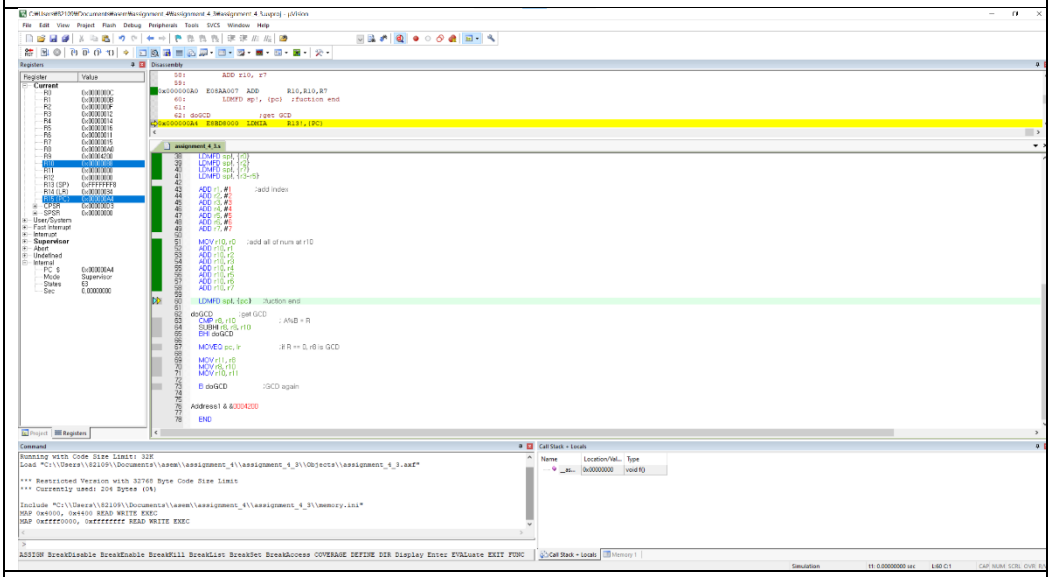
r0-r7에 10-17의 값이, r8에 160, r9에는 저장할 주소가 로드된다. 그 후 r4의 값을 미리 sp에 저장한다.



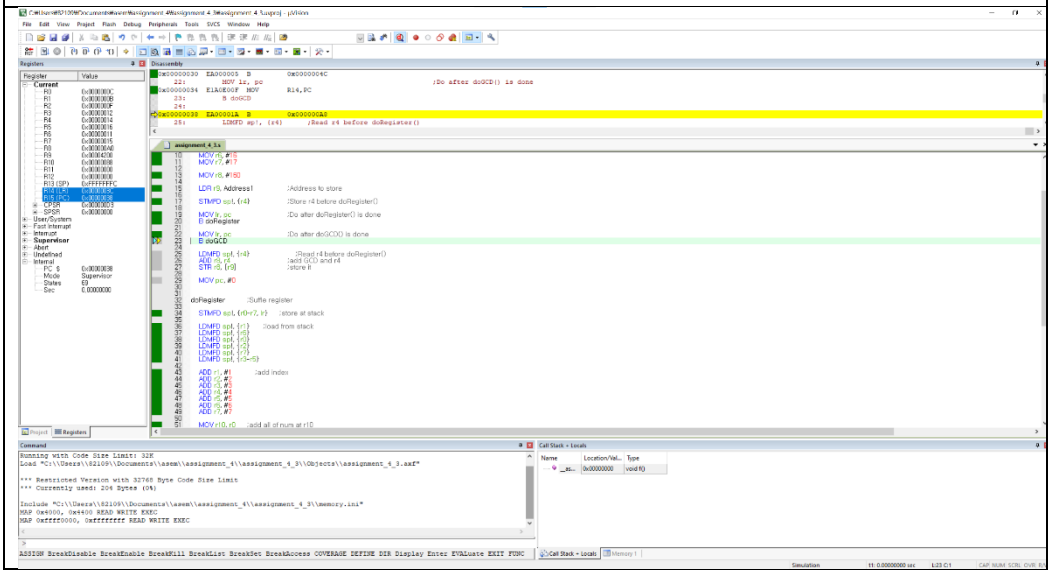
doRegister에서는 2번 문제에서 했던 메커니즘을 우선 그대로 사용된다.



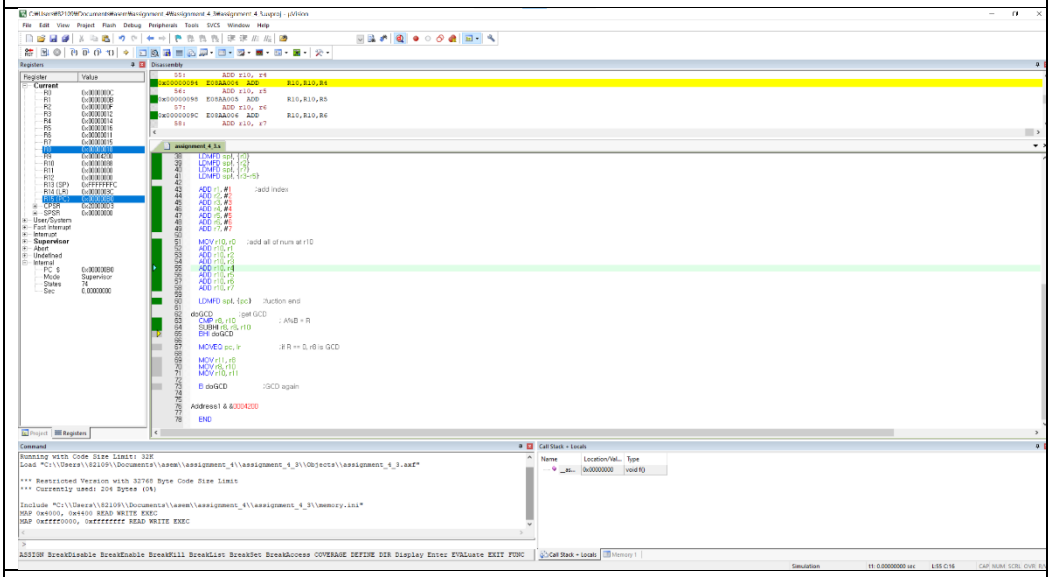
r0~r7에 index를 더한다.



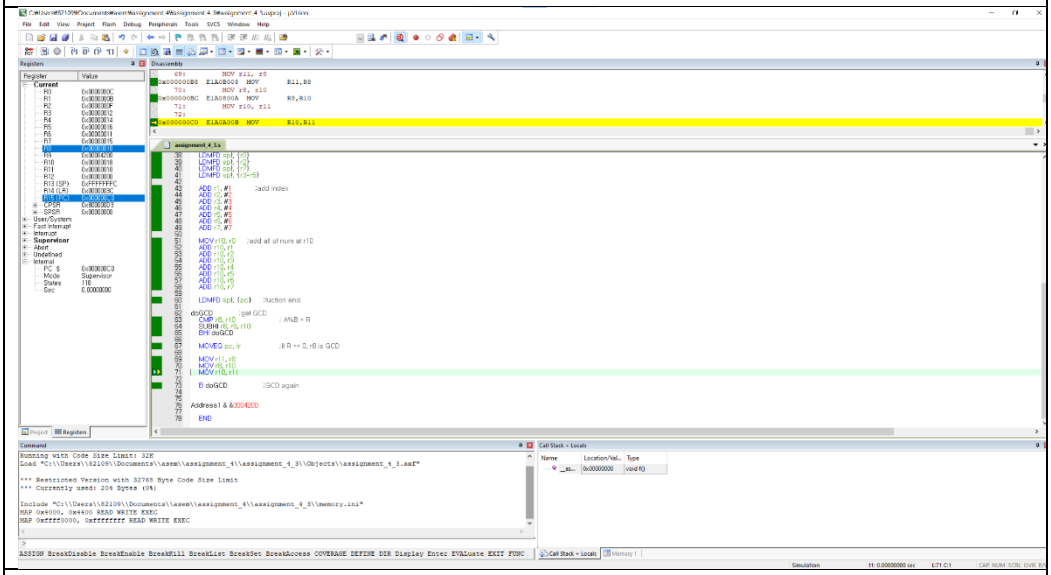
r0~r7의 값을 r10에 더하고 함수를 종료한다.



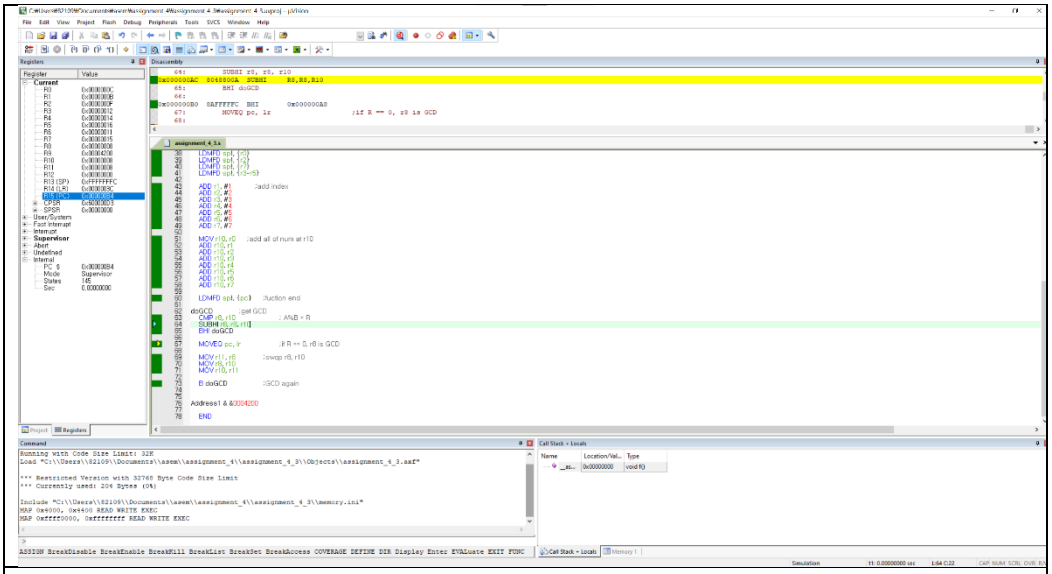
그 후 doGCD를 로드한다.



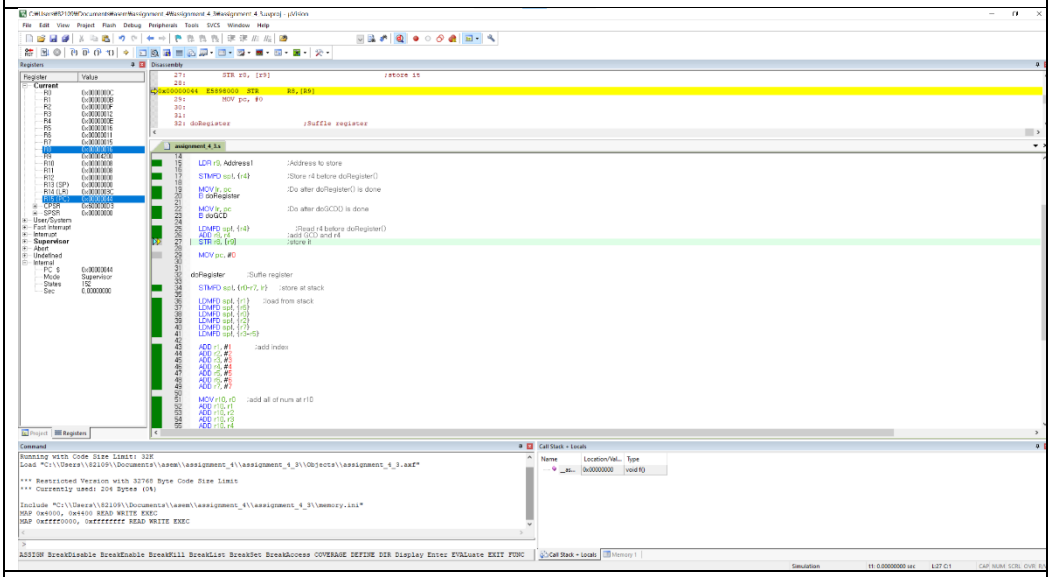
r8과 r10을 비교하여 작거나 같을 때까지 뺀다. 즉 나머지를 구한다.



나머지가 0이 아닐 경우 r8과 r10을 바꾸고 이를 다시 GCD를 수행하도록 한다.



나머지가 0일 경우 함수를 종료한다.



이를 기존의 r4를 불러와 더하여 r9에 저장한다.

## 5. Consideration

- 위의 해당 과제를 수행하면서 처음에는 Branch와 stack을 동시에 활용하는게 쉽지 않았다. 특히 MOV pc, lr을 통해 다시 하던 작업으로 돌아오는 메커니즘을 정확히 이해하지 못했다. 하지만 어셈 수업시간의 ppt를 다시 돌려보며 이해하여 이를 풀 수 있었다. 또한 GCD를 구하는 알고리즘을 LCM을 구하는 알고리즘으로 착각하여 문제를 풀었다. 이를 보고서 작성 과정에서 찾아 다시 고치는데 시간이 소요되었다. 해당 GCD를 구하는 알고리즘은 유클리드의 호제법에 따라 GCD(A, B) = GCD(B, A%B)인 점을 이용하였다.

## 6. Reference

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