Assignment2

1. Problem & Purpose

i. 두 문자열이 같을 경우에는 0x0A를, 그렇지 않을 경우에는 0x0B를 4000번지에 저장하는 Strcmp 함수 작성하기

ii. 배열을 Array[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}을 역순으로 4000번지부터 저장하시오

iii. 11+13+…+27+29를 계산하여 결과값을 4000번지에 저장하고 각 구현방법의 성능을 비교해보자(Loop을 이용한 방법[11을 shift 연산으로 구현하고 숫자는 #1만 사용], 일반화된 식 n(n+10) 이용, Unrolling 이용)

iv. Branch와 conditional execution의 차이점과 성능 차이

2. Used Instruction

I. 2-1: LDR // LDRB // STR // CMP // MOV // MOVNE // B // BEQ // DCB // DCD // END

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

ii. LDRB Rd, operand1 : operand1의 메모리 위치의 값을 byte 크기만큼 Rd에 저장한다.

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

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

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

vi. MOVNE Rd operand1 : Z가 0인 경우(CMP로 비교한 두 값이 다를 경우) operand1에 있는 값을 Rd에 저장한다.

vii. B Label1 : Label1의 위치로 이동한다.

viii. BEQ Label1 : Z가 1인 경우(CMP로 비교한 두 값이 같을 경우) Label1의 위치로 이동한다.

ix. Value1 DCB expr{expr} : expr의 정보를 1byte 단위로 저장하고 Value1을 통해 해당 위치를 읽어올 수 있다.

x. Value1 DCD expr{expr} : expr의 정보를 4byte 단위로 저장하고 Value1을 통해 해당 위치를 읽어올 수 있다.

xi. END : Assembly code가 끝났음을 의미하는 Instruction

II. 2-2 : LDR // STR // MOV // CMP // B // BEQ // DCD // END

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

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

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

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

v. B Label1 : Label1의 위치로 이동한다.

vi. BEQ Label1 : Z가 1인 경우(CMP로 비교한 두 값이 같을 경우) Label1의 위치로 이동한다.

vii. Value1 DCD expr{expr} : expr의 정보를 4byte 단위로 저장하고 Value1을 통해 해당 위치를 읽어올 수 있다.

viii. END : Assembly code가 끝났음을 의미하는 Instruction

III. 2-3 : LDR // STR // ADD // ADDNE // MUL // LSL // MOV // CMP // BNE // BEQ // DCD // END

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

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

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

iv. ADDNE Rd, R0(, R1) : Z가 0인 경우(CMP로 비교한 두 값이 다를 경우) Rd에 R0와 R1을 더한 값을 저장한다. R1이 없을 경우 Rd = Rd + R0로 저장한다.

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

vi. Rd, LSL, operand1 : Rd를 operand1만큼 왼쪽으로 비트 쉬프트한 값을 사용한다.

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

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

ix. BNE Label1 : Z가 0인 경우(CMP로 비교한 두 값이 다를 경우) Label1의 위치로 이동한다.

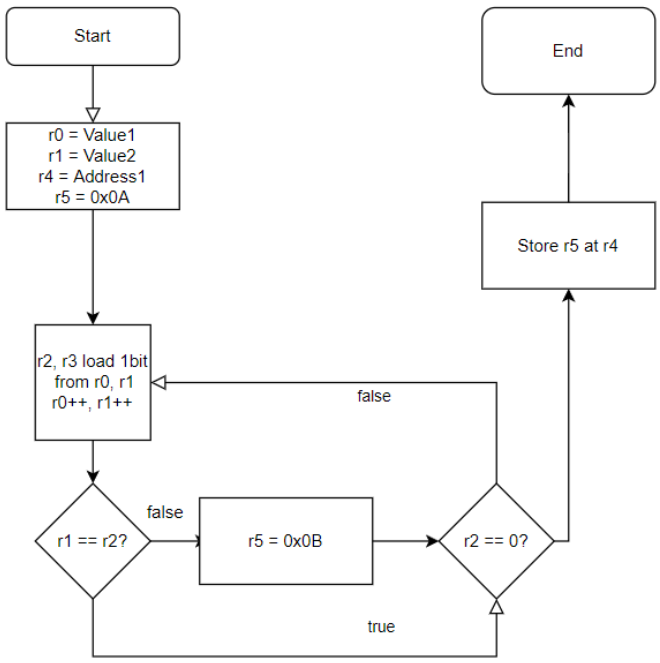
x. BEQ Label1 : Z가 1인 경우(CMP로 비교한 두 값이 같을 경우) Label1의 위치로 이동한다.

xi. Value1 DCD expr{expr} : expr의 정보를 4byte 단위로 저장하고 Value1을 통해 해당 위치를 읽어올 수 있다.

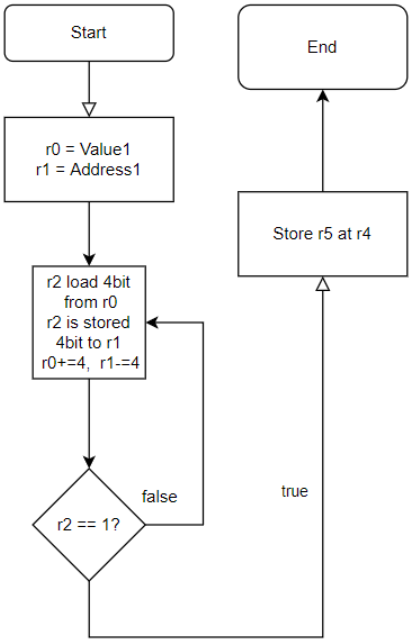
xii. END : Assembly code가 끝났음을 의미하는 Instruction

3. Design(Flow chart)

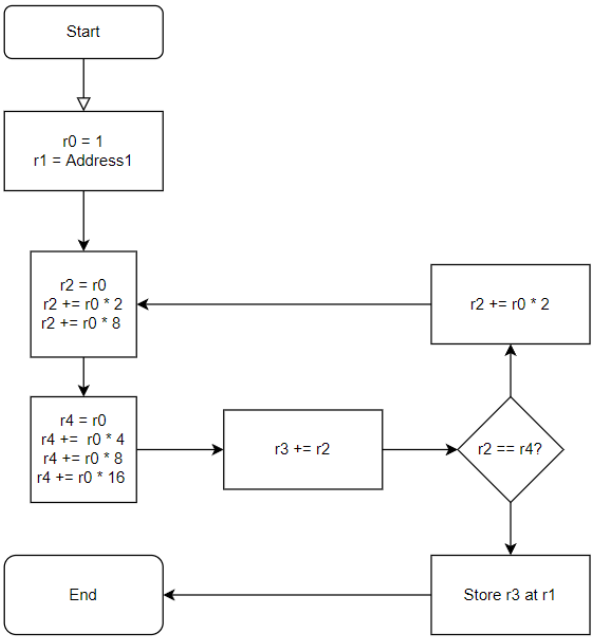
i. 2-1 flow chart



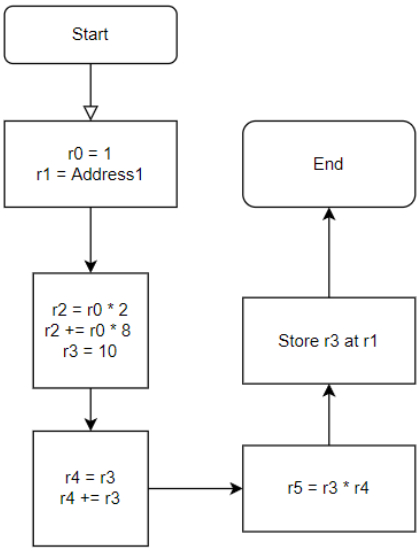
ii. 2-2 flow chart



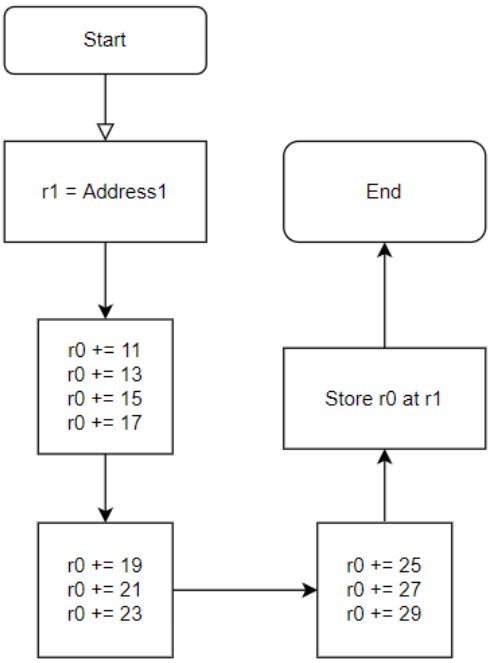
iii. 2-3-1 flow chart



iv. 2-3-2 flow chart

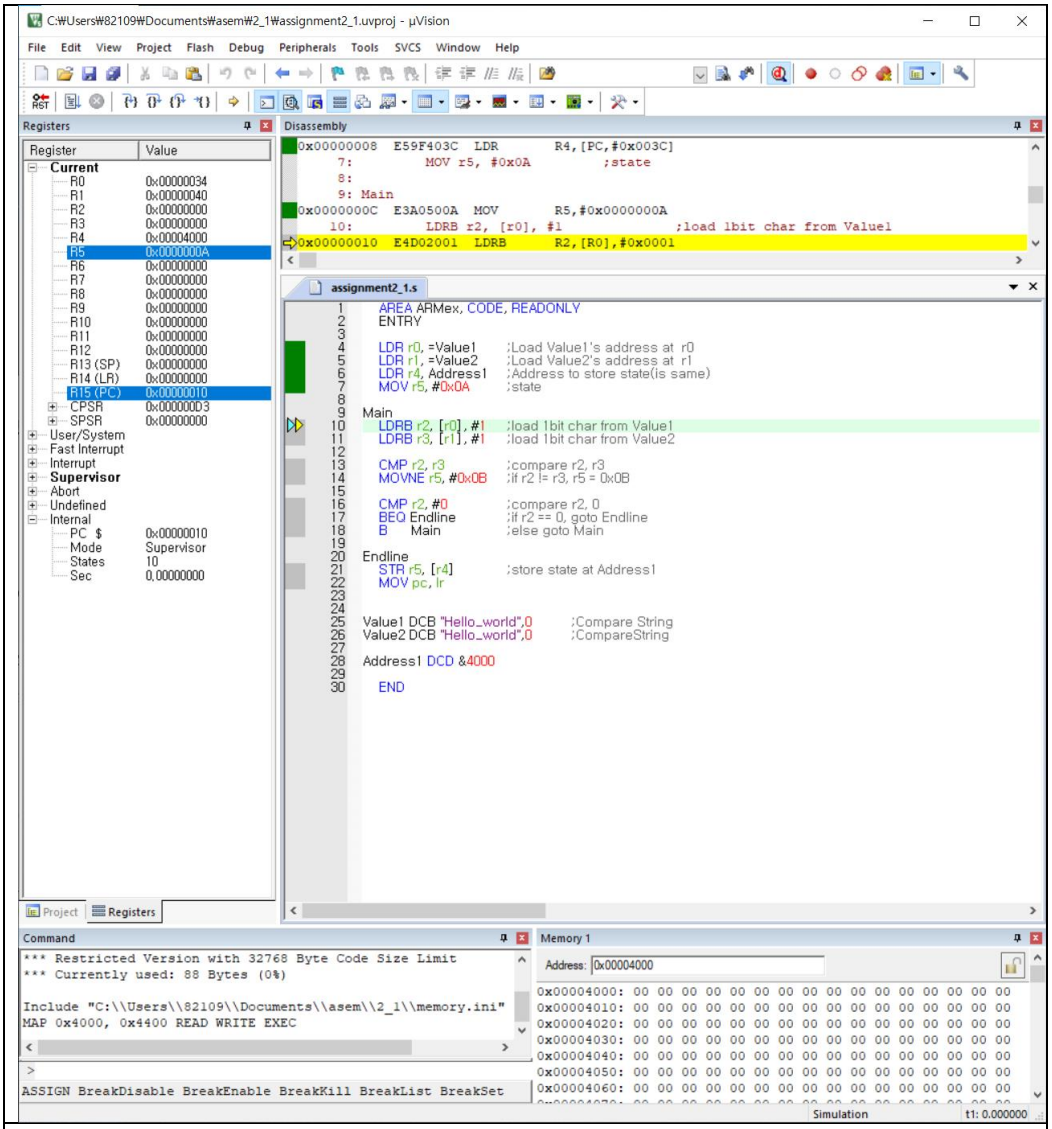


v. 2-3-3 flow chart

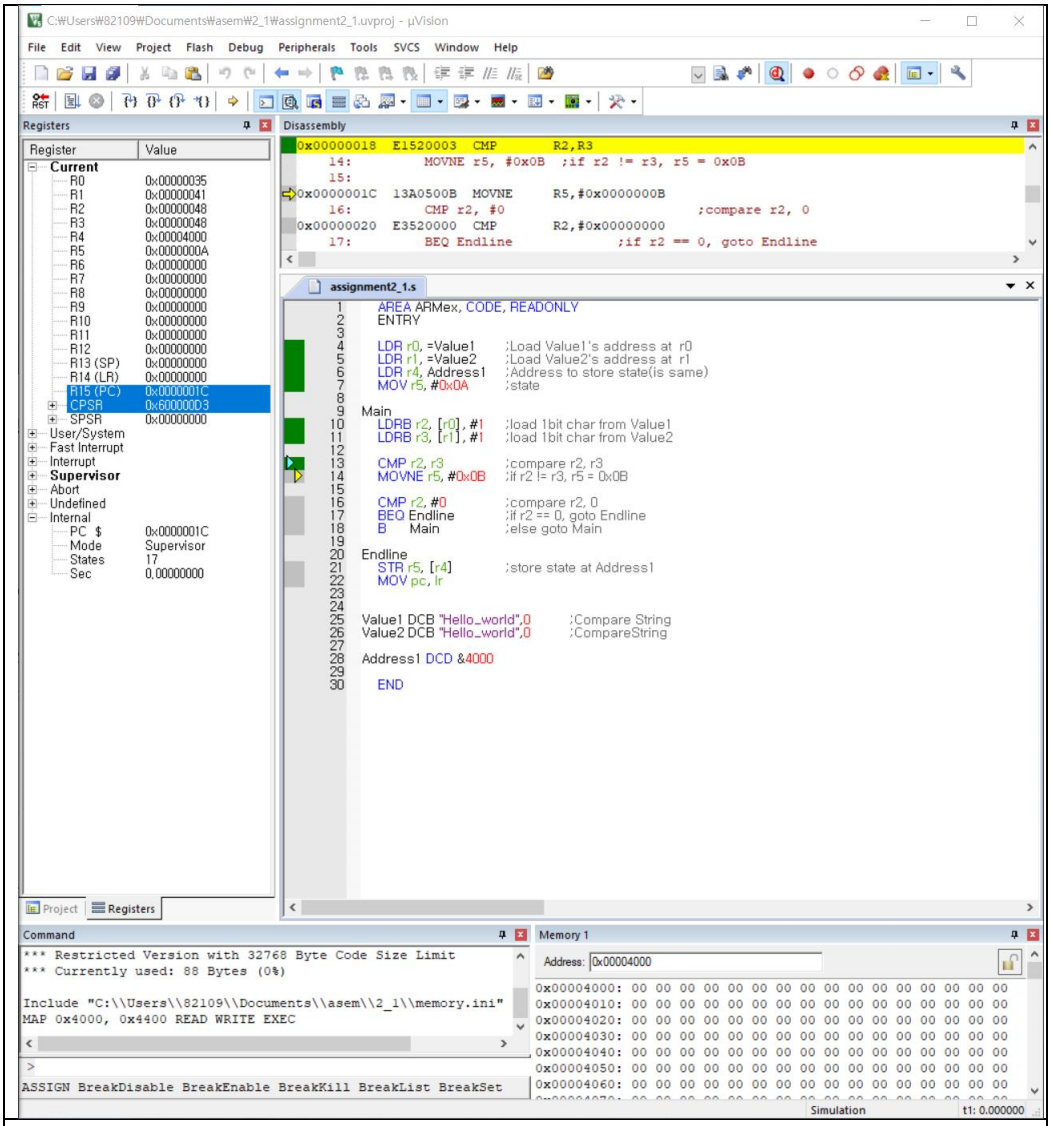


4. Conclusion

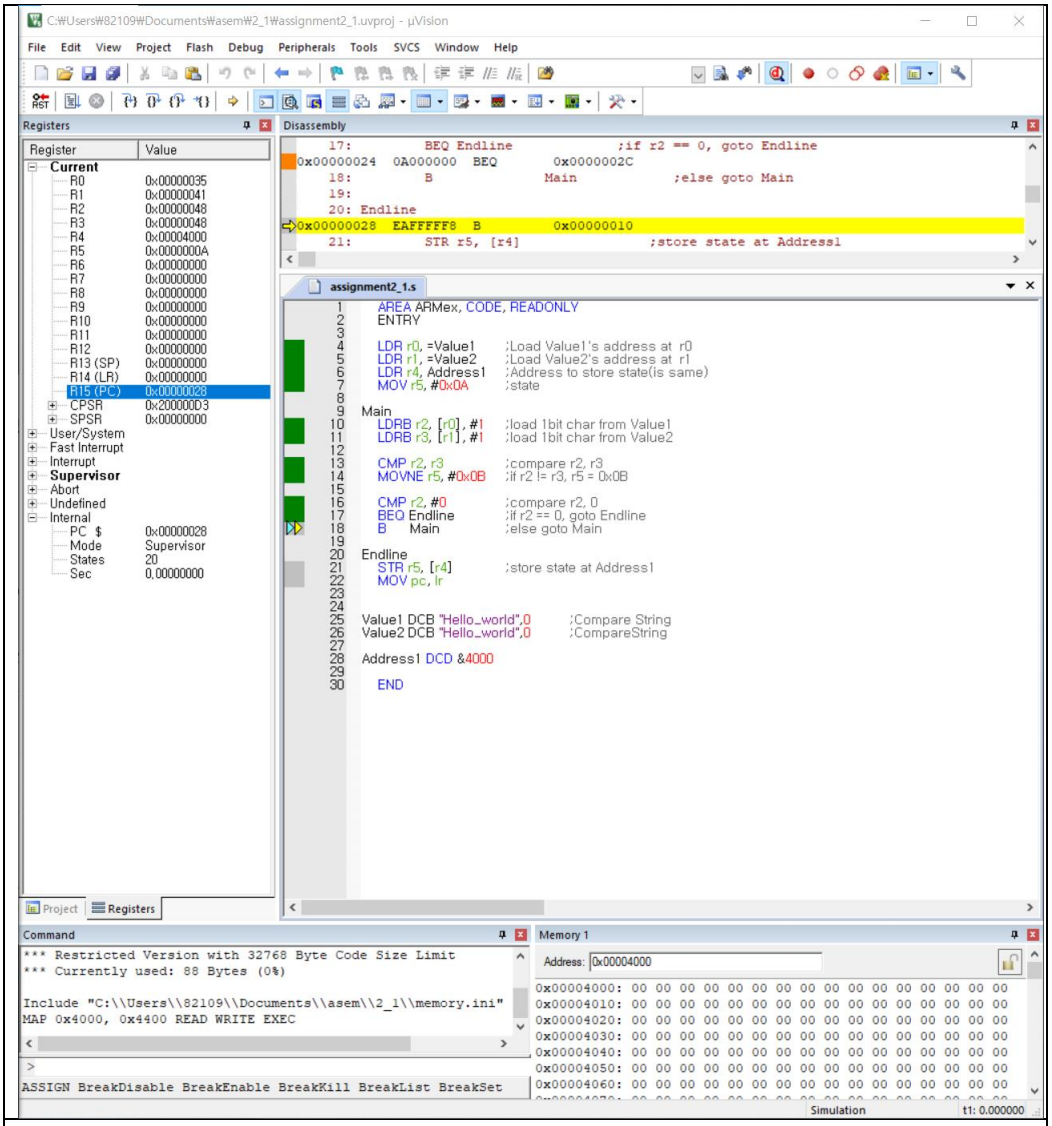
i. 2-1 result



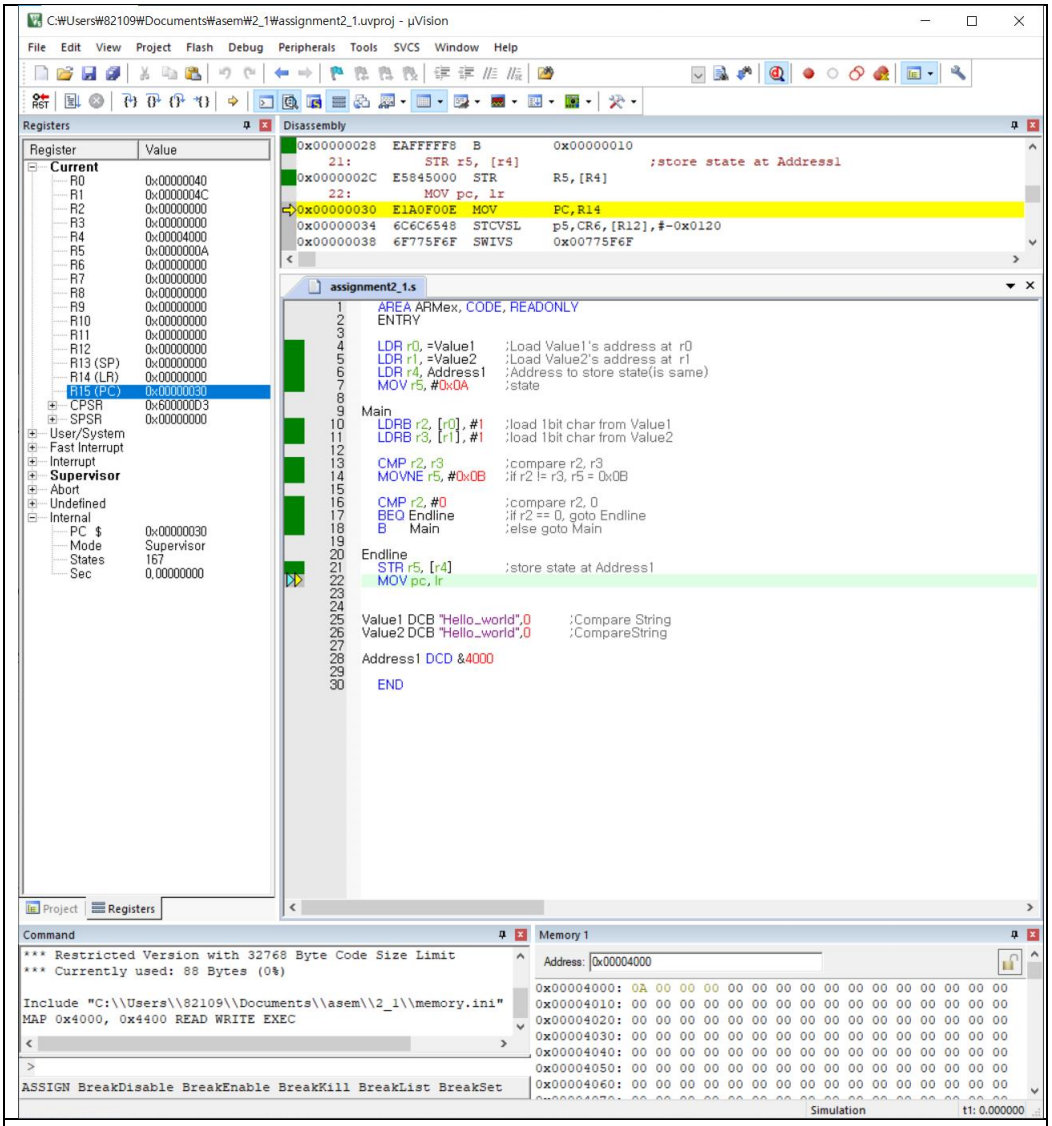
r0, r1에 Value1과 Value2가 저장된 위치가 들어갔고, r4에는 저장할 위치, r5에는 현재 상태인 0x0A가 정상적으로 들어갔다.



r2, r3는 각각 r0, r1에서 1바이트를 읽어오고 이 주소를 1 늘린다. 그 후 데이터를 비교하여 현재 cpsr를 업데이트 한다.

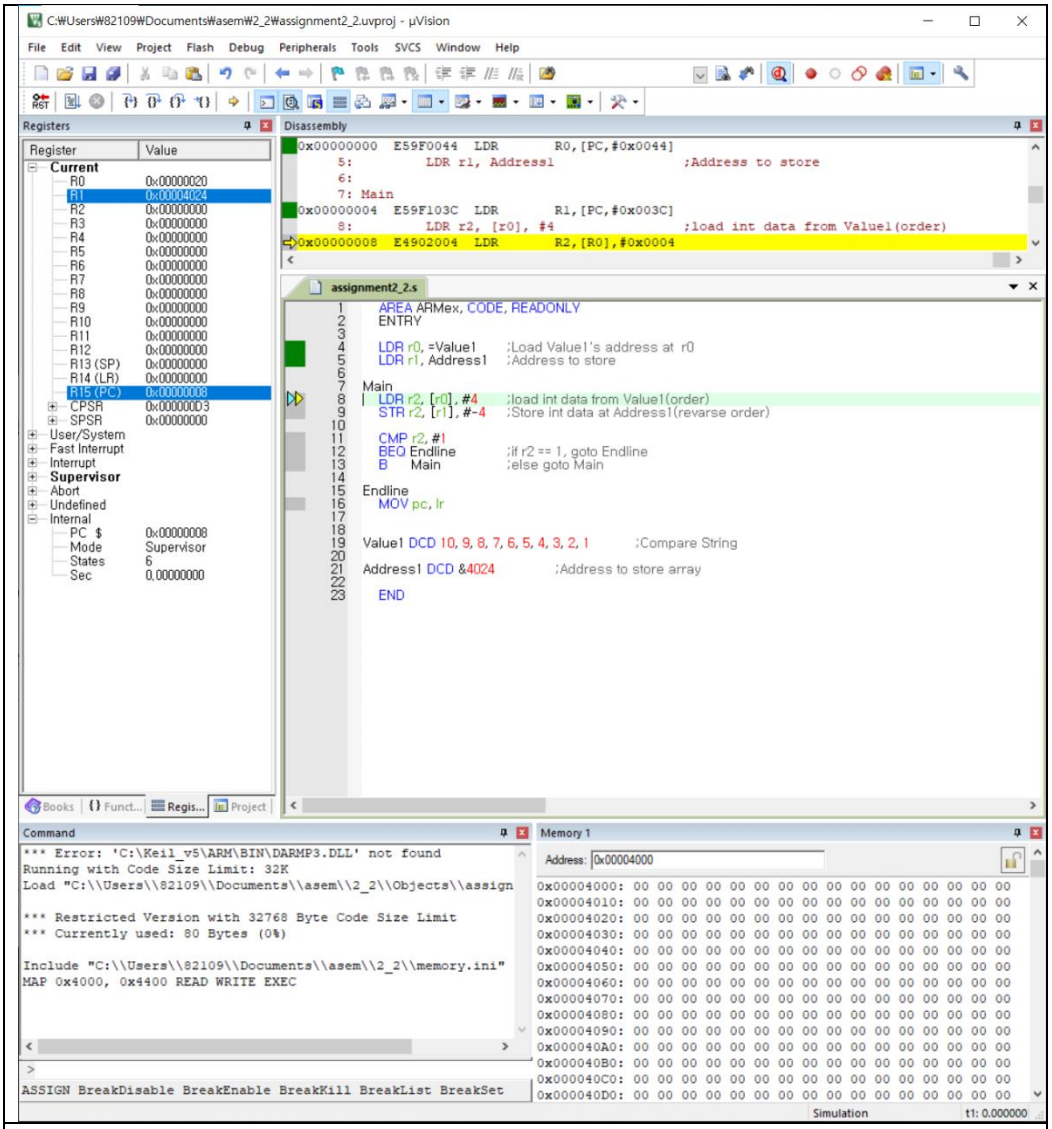


만약 r2 != r3일 경우 현재 r5의 값을 0x0B로 변경한다. 그 후 r2가 0일 경우 Endline으로 이동하고 아니면 Main으로 이동한다.

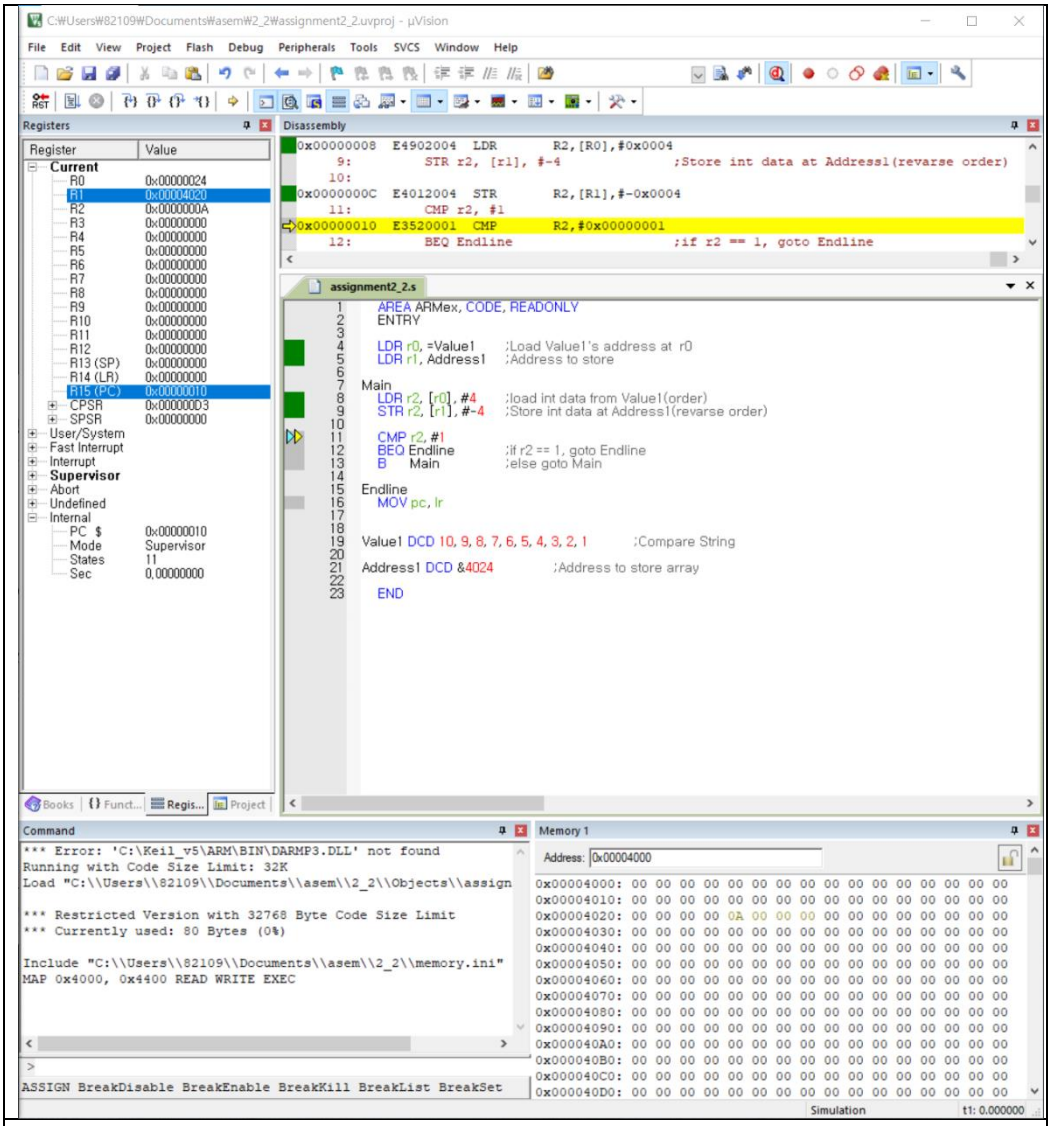


현재 상태인 r5를 r4의 위치(4000번지)에 저장한다.

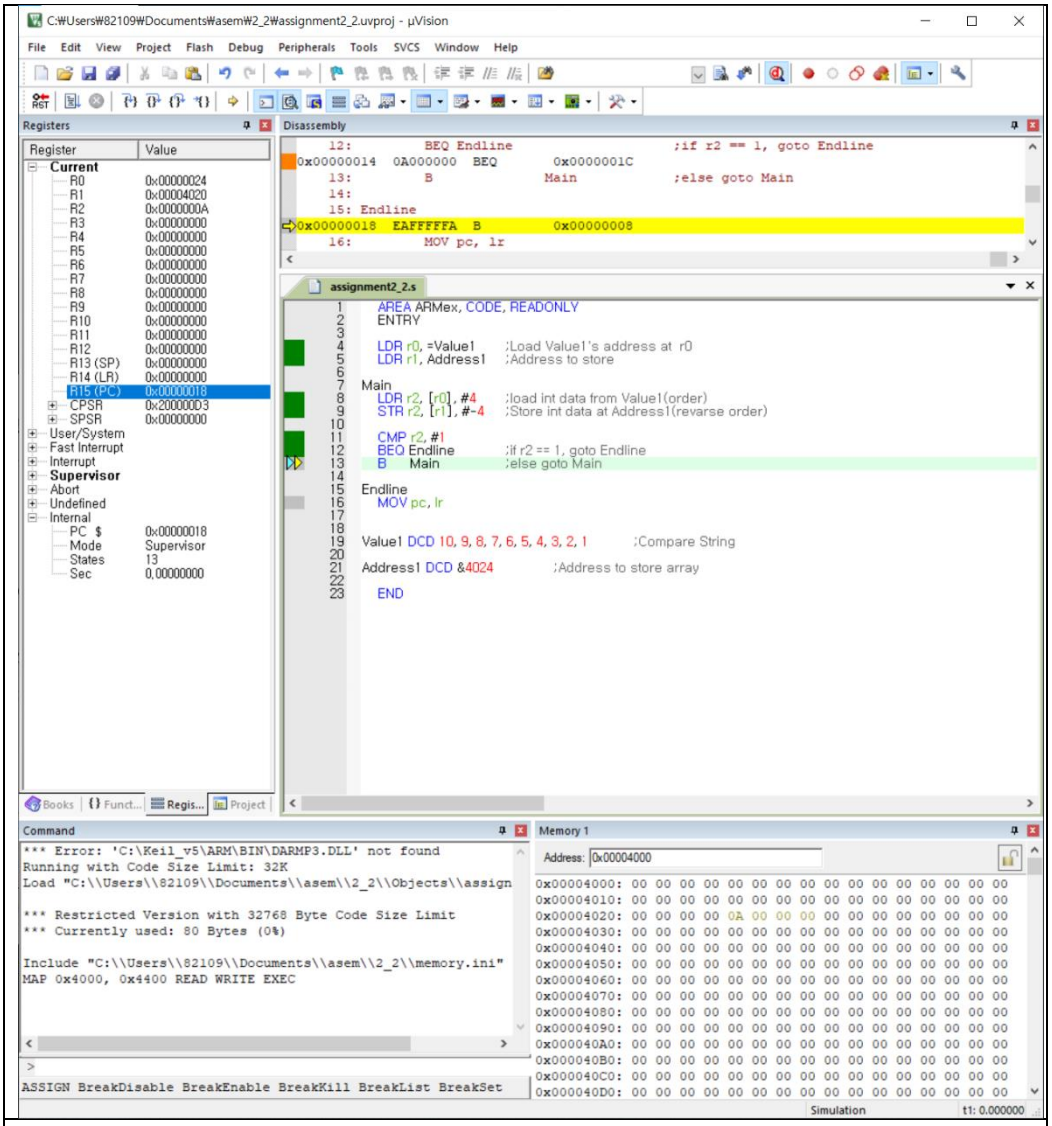
ii. 2-2 result



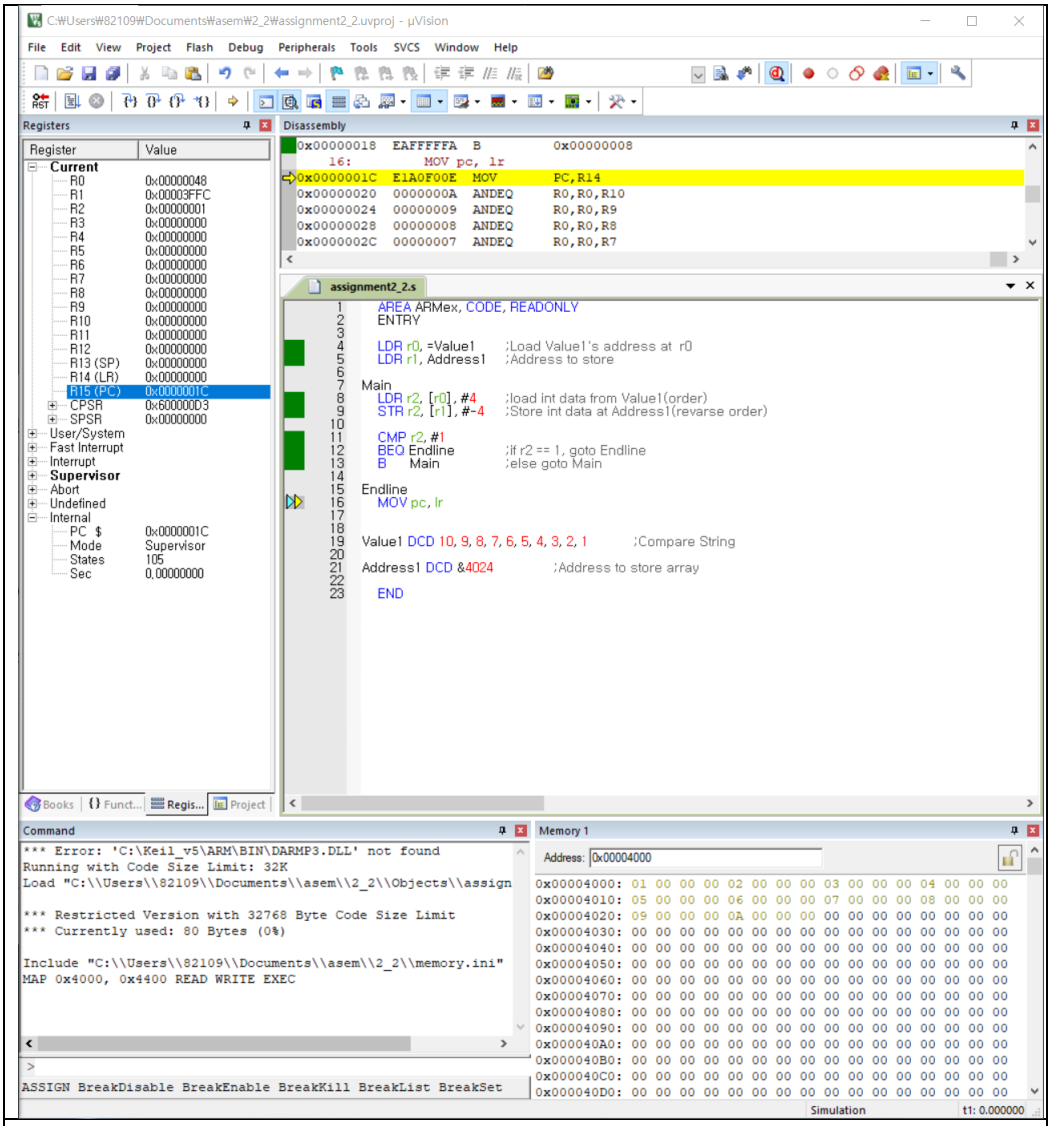
r0에 배열이 들어갔고, r1에는 4024번지의 주소가 들어갔다.



r0에 있던 데이터를 r2를 거쳐 r1으로 이동시킨다. 이때 r0의 주소는 4byte 늘리고, r1의 주소는 4byte 줄인다.

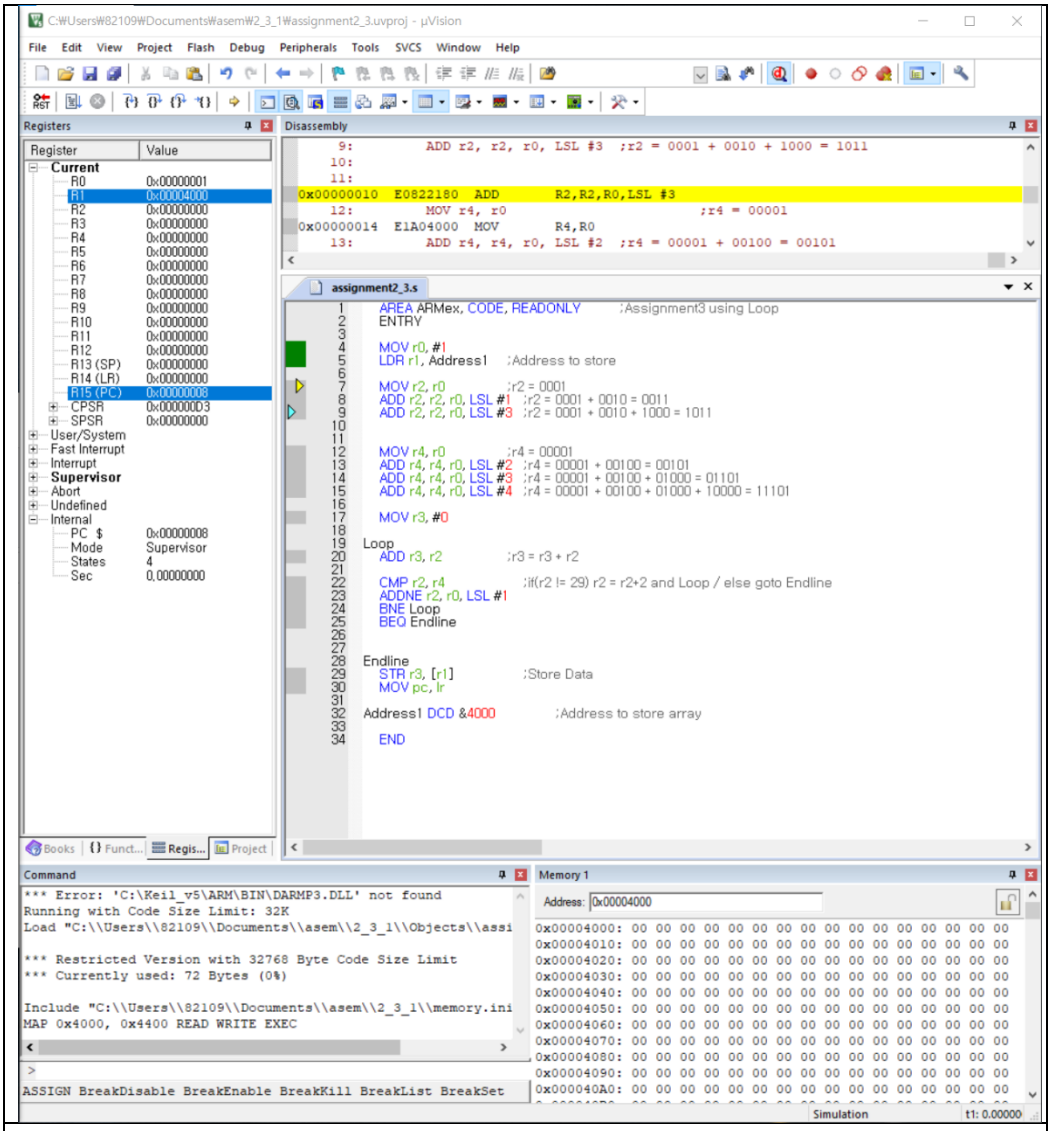


만약 현재 저장한 데이터 r2가 1이 맞으면 Endline으로 이동하고, 아닐 경우 Main으로 이동한다.

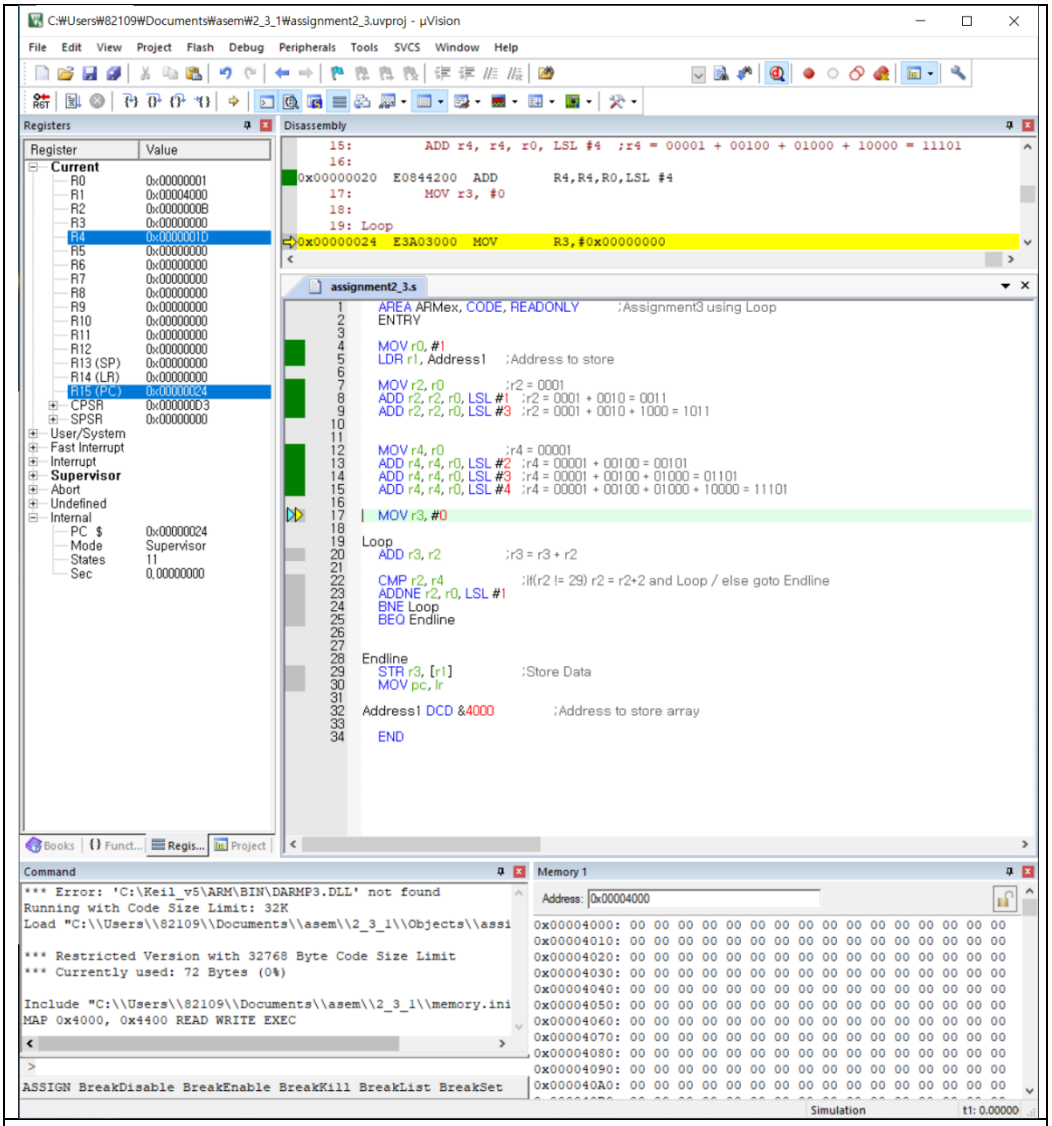


사진에서 볼 수 있듯이 4000번지부터 데이터가 정렬되어서 저장되어있다.

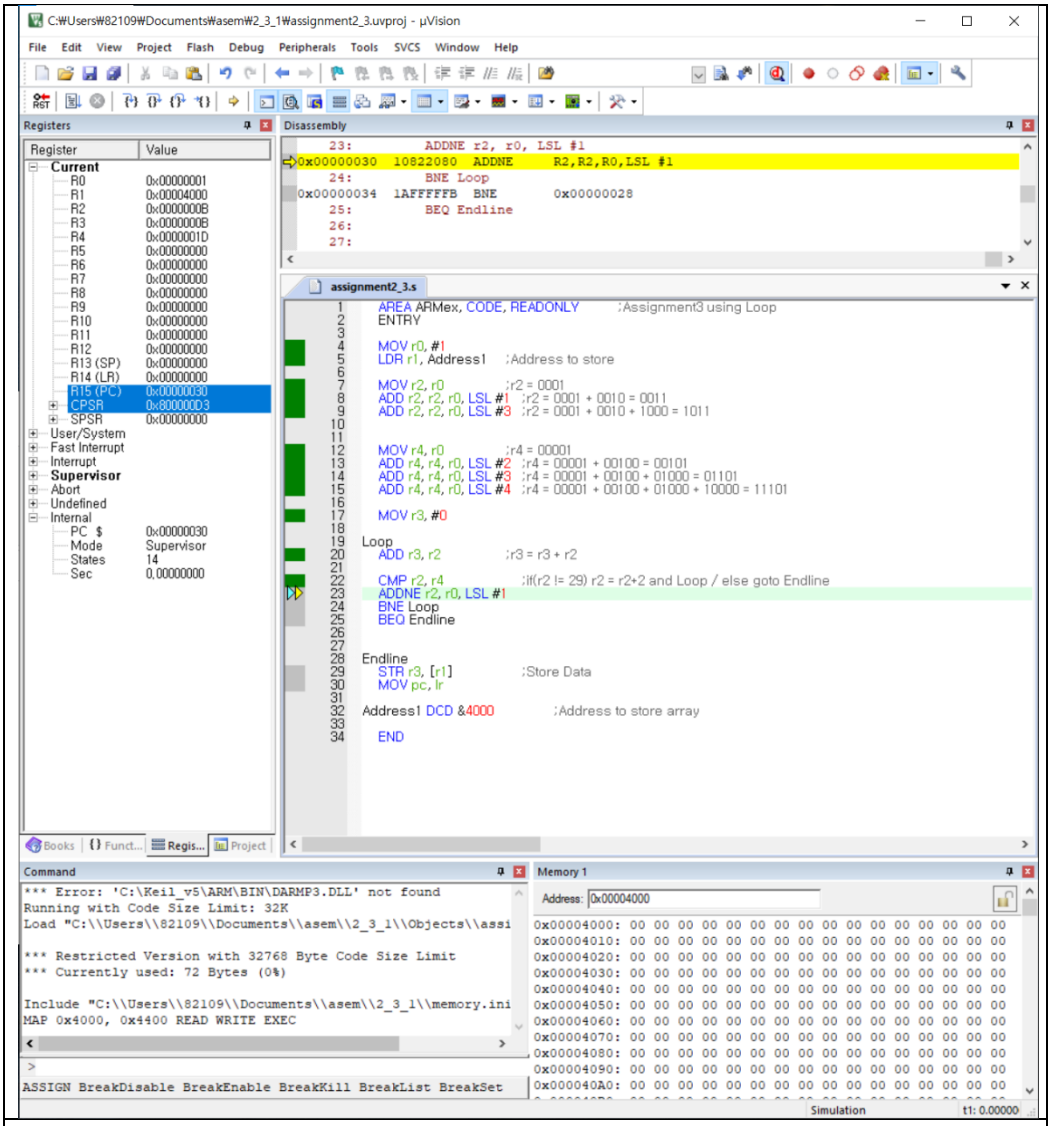
iii. 2-3-1 result



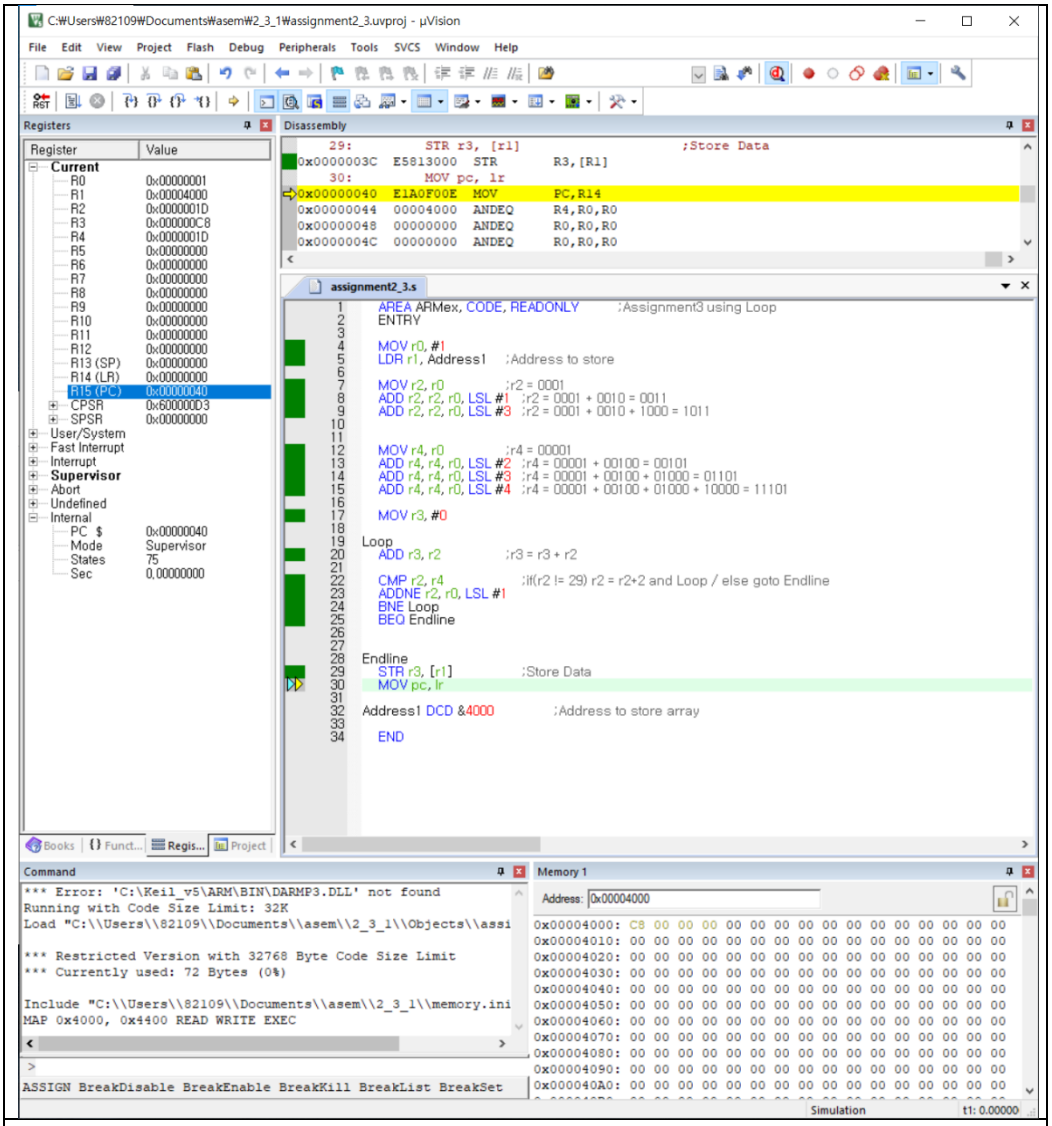
r0에는 비트 쉬프트에 사용될 1이, r1에는 저장할 위치 4000번지가 들어갔다.



r2, r4는 비트 쉬프트를 통해 덧셈을 하여 11과 29를 만든다.

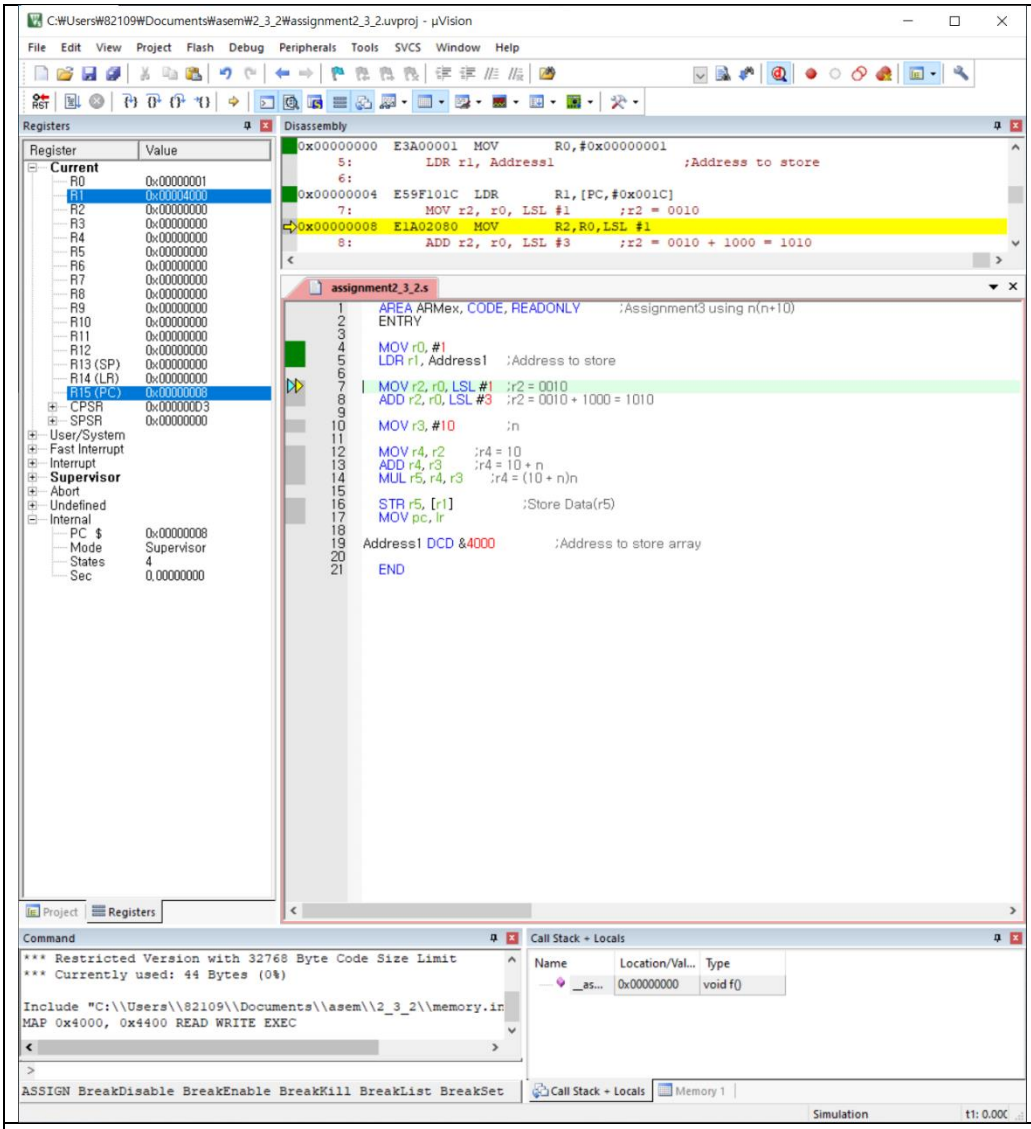


r3에 r2를 저장하고 CMP를 통해 r2와 r4(29)를 비교한다. 29가 아닐 경우에는 r2에 2를 더하고 Loop를 돌고, 같을 경우 EndLine으로 이동한다.

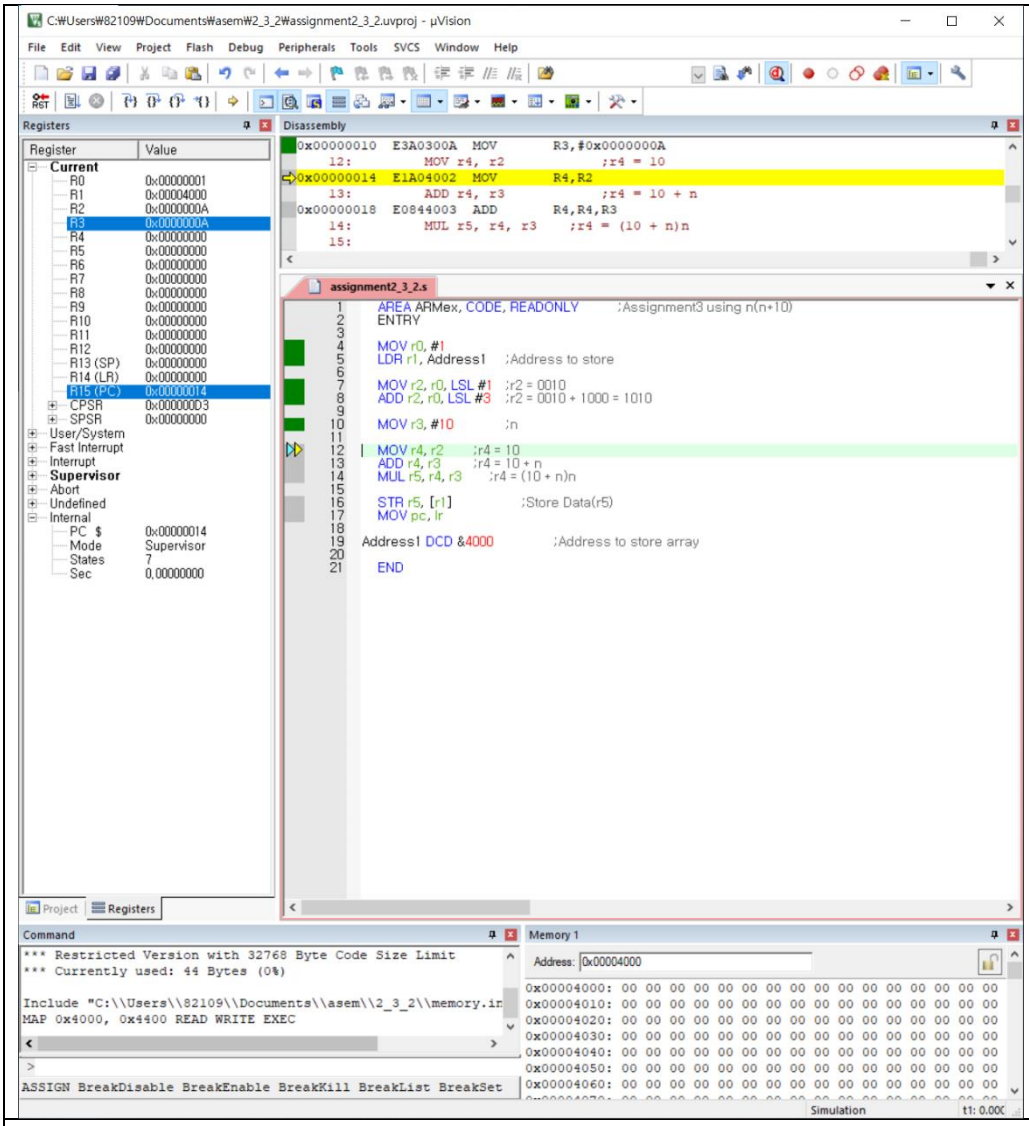


마지막에 현재 r3을 4000번지에 저장한다.

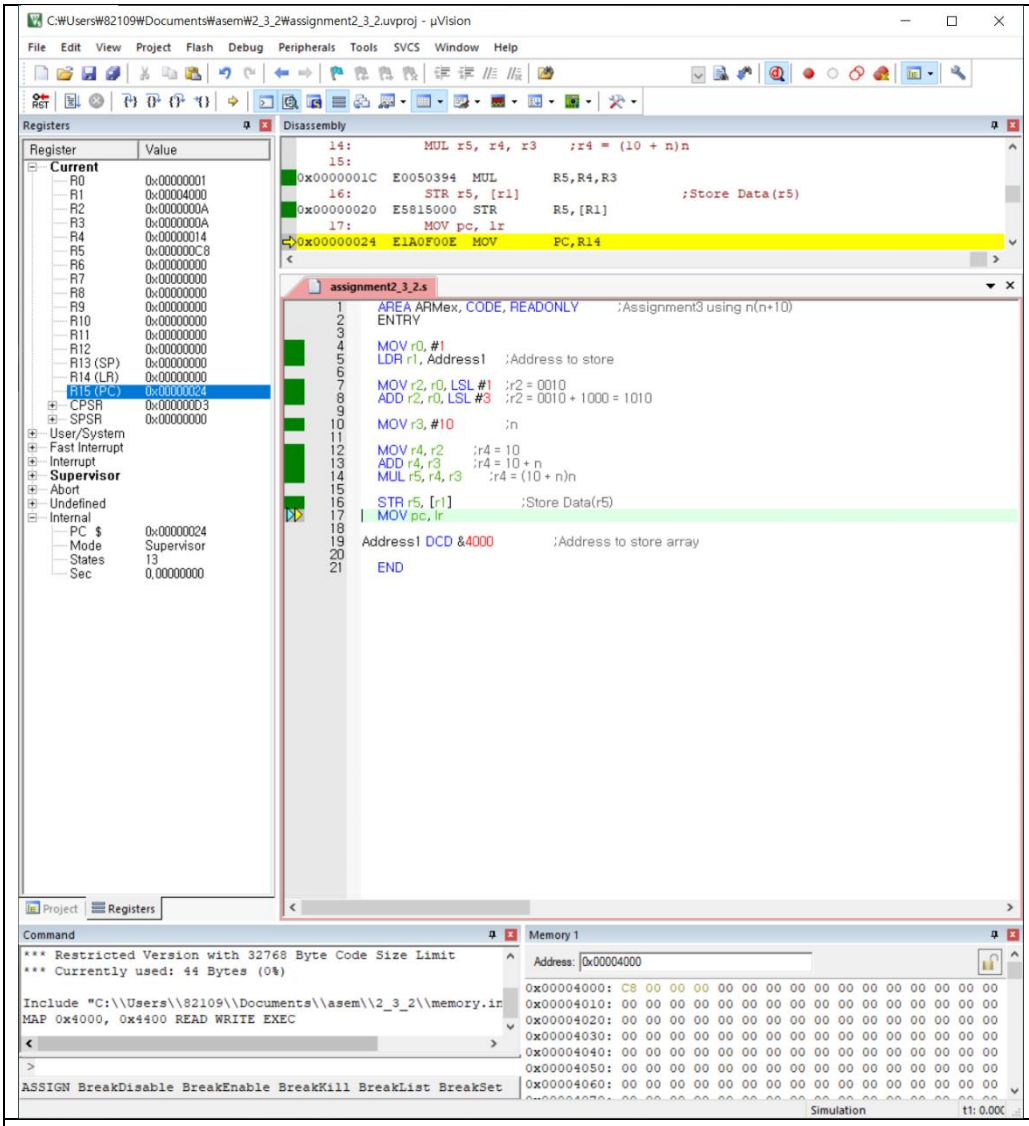
iv. 2-3-2 result



r0에는 비트 쉬프트에 사용될 1이, r1에는 저장할 위치 4000번지가 들어갔다.

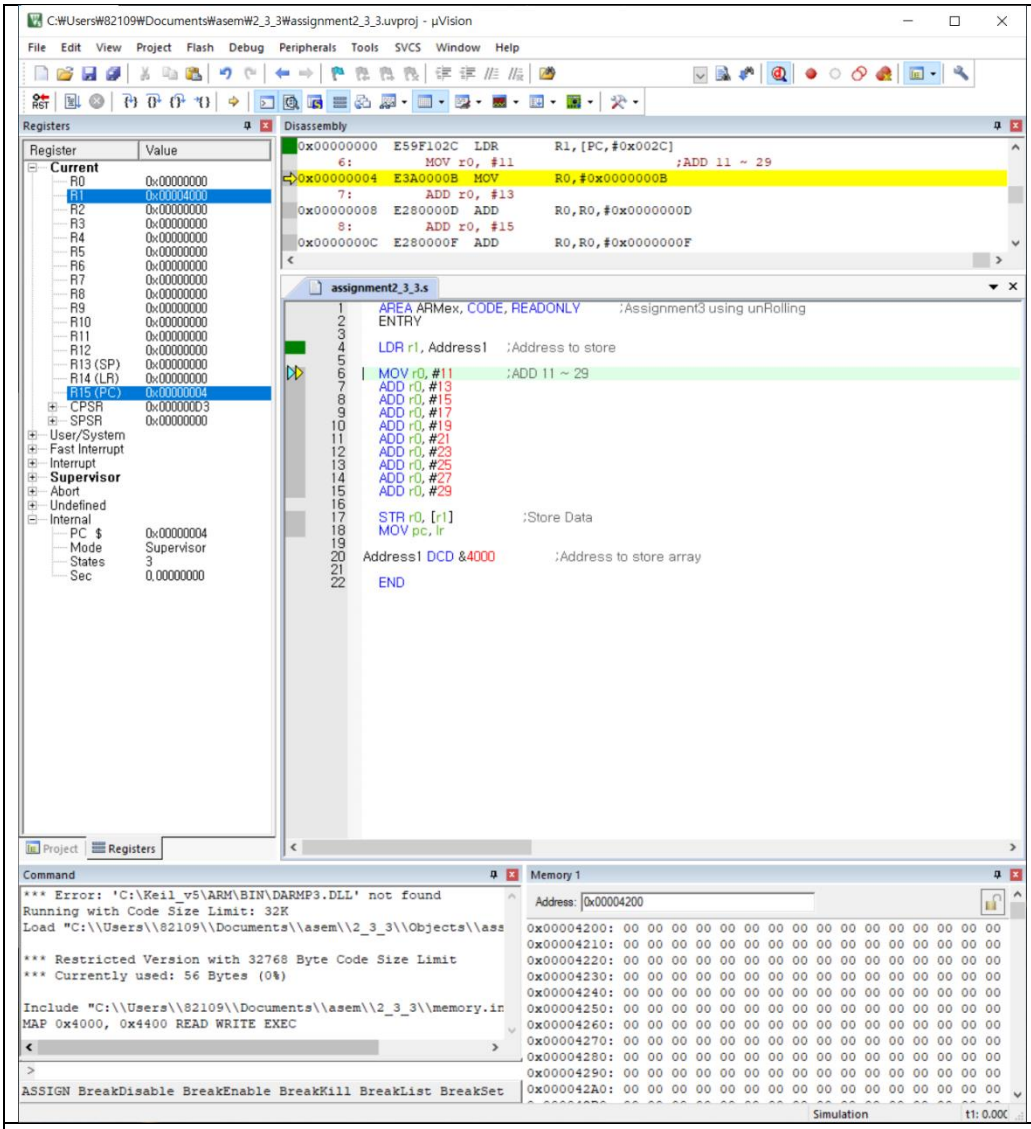


r2는 비트 쉬프트 연산을 통해 10이 저장되었고, r3에는 n값인 10이 저장되었다.

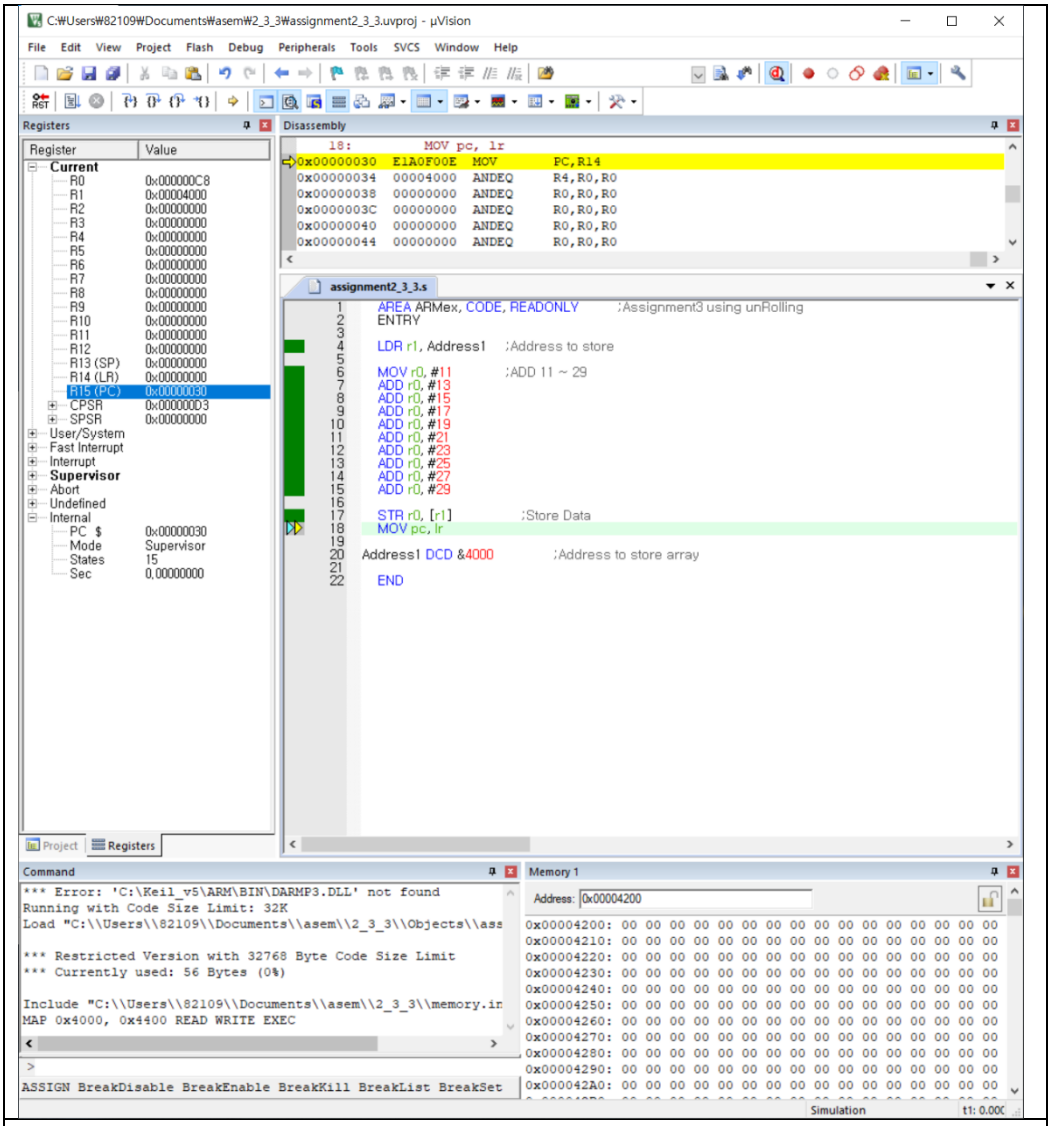


r4에는 n + 10의 값, 즉 20이 저장되고, r5에는 n(n + 10)의 곱연산의 값 200이 저장된다. 그리고 이 데이터를 4000번지에 저장한다.

v. 2-3-3 result



r1에는 저장할 위치 4000번지가 들어갔다.



r0에 11부터 29의 합을 저장하였고, 이 값을 r1에 저장하였다.

vi. 2-3 compare

- 위의 예제들을 비교하였을 때 Loop > unrolling > 일반식 순으로 state를 사용하였다. 이를 통해 우리는 프로그램을 설계할 때 일반적인 식으로 나타낼 수 있을 경우 일반적인 식으로 나타내고, 아니면 풀어서 작성하는 것이 유리하다는 것을 알 수 있다.

vii. Branch와 Conditional execution

- Branch의 경우 Pipline에서 기존의 데이터를 버려야 하기 때문에 2번의 연산 손실이 발생한다. Branch를 포함해 3번의 연산 손실이 발생하기 때문에 Conditional execution을 3번 이하로 사용할 수 있을 경우 Conditional execution을 사용하고 그

외의 경우에는 Branch를 사용하는 것이 효율적이다.ㄴ

5. Reference

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