**임베디드 실습 어셈블리 프로그래밍**

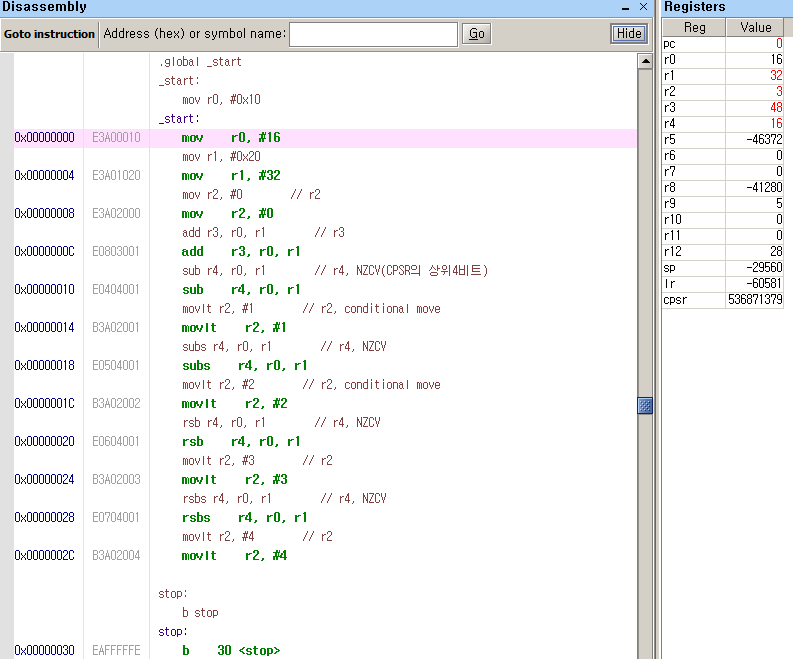
**2015253039 권진우**

4/21

**1**. (모니터 프로그램 사용하여 ARM 명령어 동작 확인)

(1) 압축파일로 주어진 4개의 프로그램(proj1.s – proj4.s)을 사용하여 소스코드에서 지시한 레지스터의 결과 등을 적고, 필요하면 간단한 설명을 적으시오.

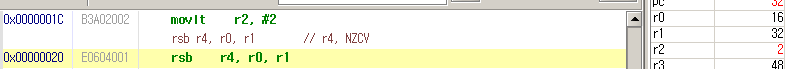
**proj1.s**



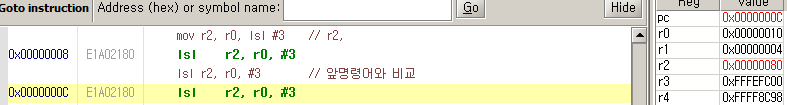
**-** 위는 proj1 프로그램을 한번 실행하고 난 결과이다.

- **r0, r1은 각각 mov로 16, 32라는 값이 들어갔고 r3는 r0+r1으로 48이라는 값이 들어갔다. 하지만 여기서 상태레지스터는 업데이트되지 않았다. 따라서 0x00000014번지 명령어의 movlt의 명령어는 조건부 실행 명령어지만 조건이 충족되지 않기에 수행되지 않고 0x00000018의 subs의 경우 'S'접미사가 붙기에 상태레지스터도 업데이트 된다. r0-r1의 결과는 음수가 되므로 'N'bit가 1이 되므로 lt조건이 충족되어 movlt r2, #2 가 수행되게 된다. 이후 rsb는 Reverse Sub로 r1-r0를 수행하게 되며 rsb의 경우도 'S'접미사가 없으므로 상태레지스터가 업데이트 되지 않고 이전의 값이 그대로 있으므로 movlt가 수행되어 r2에는 3이 들어간다.. 마지막 rsbs의 연산에 의해 상태 레지스터가 변경되지만 lt조건이 만족되지 않으므로 최종 r2레지스터의 값은 3이 되는 것을 확인 할 수 있다.**

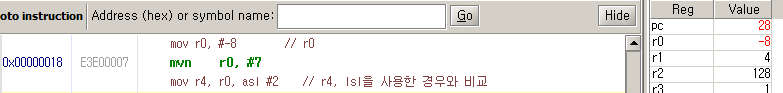
\* 아래는 중간과정 r2에 2가 들어가는 것 확인



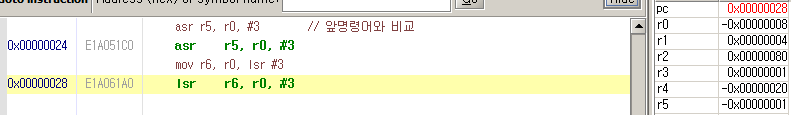
**proj2.s**



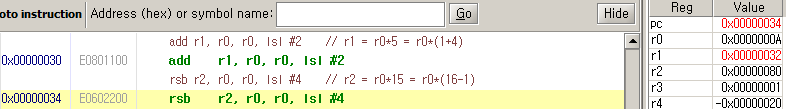
**- mov r2, r0, lsl #3과 lsl r2, r0, #3은 같은 수행을 하는 명령어이다. Logical Shift Left이므로 r0의 값을 좌 3 쉬프트 한 값을 r2로 넣는다. 0x00000010 -> 0x00000080이 된다.**



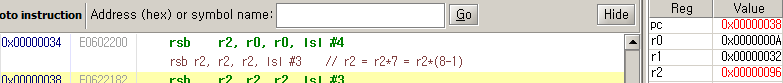
**- mov r0, #-8의 경우 immediate값을 넣을 수 없기에 mvn연산을 사용한다. move not 연산으로 써 #7에 not연산을 수행한 값을 r0에 넣게 된다. ~(00000111) = 11111000 -> Signed 표현 -> -{(00000111)+1} = -8이므로 Signed로 출력하게 되면 -8의 값이 나오게 된다.**



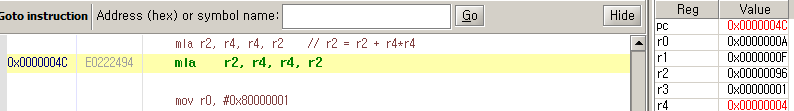
**- asl은 Arithmetic Shift Right이며 산술 우측 쉬프트이다. 따라서 부호 비트는 유지되며 우측쉬프트가 된다. -0x00000008이었으므로 산술 우측 쉬프트 3번을 하면 부호비트는 유지되고 8->4->2->1순으로 -0x00000001이 된다.**



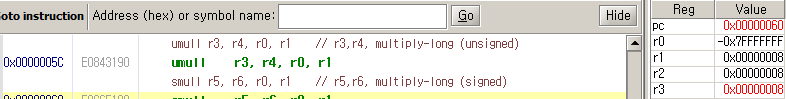
**- 다음은 add r1, r0, r0, lsl #2 명령어가 수행된 직후이다. 다음은 r0 \* 5와 같은 연산을 수행하게 되는데 r0 를 lsl #2이므로 r0값의 4배 값과 r0 값을 더하는 것이므로 r1=r0+(r0\*4)가 된다.**



**- rsb는 Reverse Sub이므로 r0를 16배 한 것에서 r0를 빼는 연산이므로 r2=(r0\*16)-r0 <-- r2=r0\*15와 같다.**

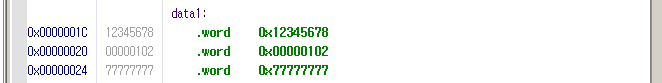


**- mla는 Multiplier Accumulate이므로 r2 = r4\*r4+r2를 수행한다.**

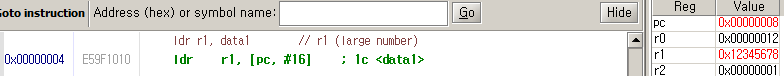


**- umull은 Unsigned 64bit를 저장하는 곱셈 연산으로써 r4에는 상위32bit r3에는 하위 32bit가 저장된다. -0x7FFFFFFF은 0x80000001과 같기에 r3의 값은 1\*8해서 8이 나오게 되는 것이다.**

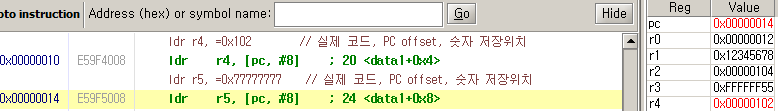
**proj3**



**- 위 소스는 .word를 이용하여 data1에 대한 값을 정의하여 놓은 것이다.**

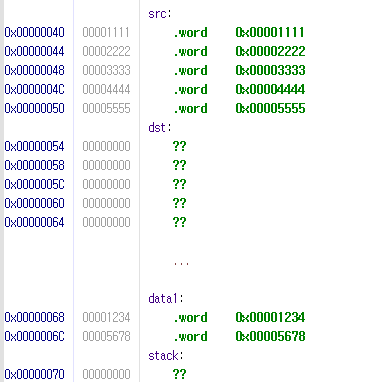


**- ldr은 LOAD Register 명령으로 [pc, #16] 주소의 값을 레지스터 r1으로 가져오는 명령어이다. [pc, #16]은 현재 수행중인 pc값에서 +16을 한 Memory 주소의 값을 r1에 넣게 된다. 위에서 주의할 점은 Program Counter값은 파이프라이닝으로 인하여 실제 pc값은 다음다음 명령어(Inst Fetch)의 주소를 갖기 때문에 현재 실행하는 명령어 +8(명령어 1개 : 4Byte)값을 가지게 된다. 따라서 0x00000004에서 +8 +16을 한 값을 메모리 주소로 가리키게 된다. 16진수로 표현하면 0x0000001C이므로 0x12345678 를 r1에 넣게된다.**

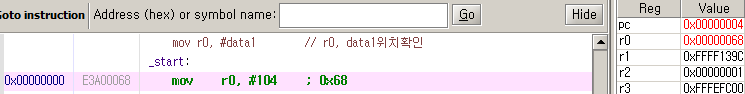


**- 0x102는 immediate로 쓸 수 없는 수이므로 0x00000020 메모리 주소에 접근하여 값을 가져온다. 0x00000010 + 8 + 8 = 0x00000020 이다.**

**proj4**



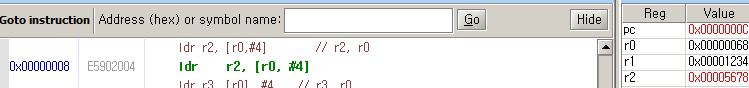
**- 다음과 같이 src, dst, data1, stack이 정의 되어있다.**



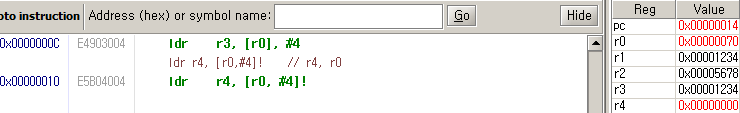
**- #data1은 data1값이 들어있는 주소를 가져오는 것을 의미한다. data1이 저장 되어있는 시작주소는 0x00000068이다. 따라서 10진수로 mov r0, #104가 수행된다.**



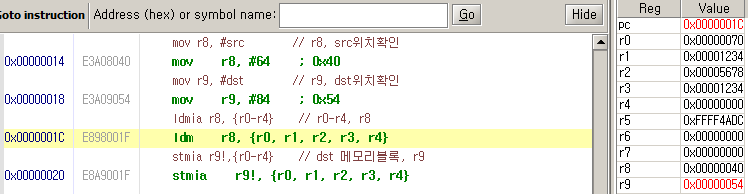
**- ldr r1, [r0] 이므로 r0값이 가리키는 주소인 0x00000068의 값을 가져온다. 따라서 0x00001234가 r1에 실린다.**



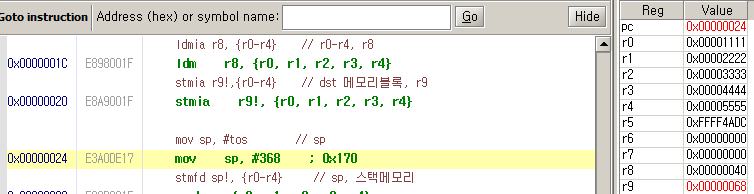
**- [r0, #4]는 0x0000006C이다.**



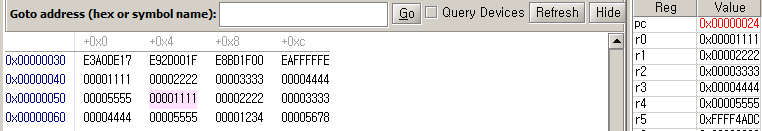
**- ldr r3, [r0], #4는 r0가 가리키는 주소 값(0x00001234)을 r3에 넣은 후 r0값에 +4를 한다. 따라서 r0는 0x0000006C가 된다. 이후 ldr r4, [r0, #4]!를 수행하였으므로 r0에 +4를 한 주소의 값을 r4에 넣고 r0값을 +4 값으로 업데이트 한다. 0x00000070에는 '0'이 들어있었으므로 r4에는 0값이 들어간다.**



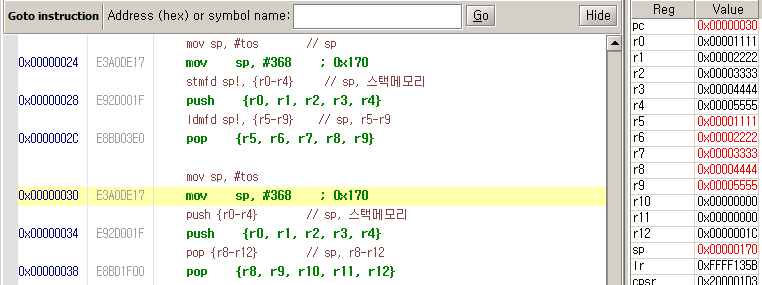
**- mov를 통해 r8, r9에 src와 dst가 될 주소를 저장한다.**



**-ldm과 stm을 통해서 메모리의 값을 연속적으로 불러오고 r9이 가리키는 주소부터 연속적으로 저장을 하였다.**

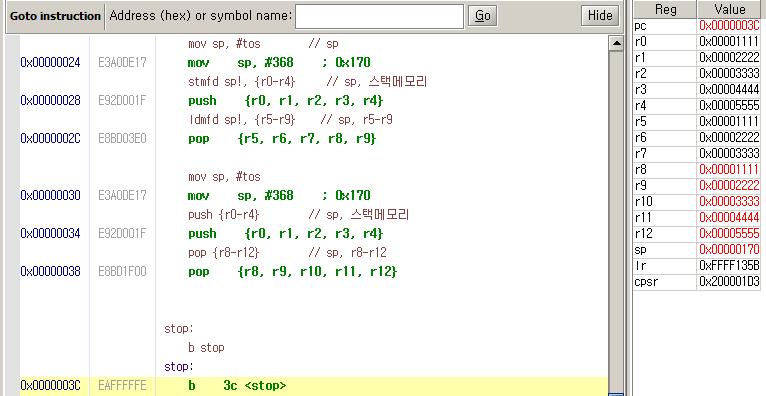


**- 위 메모리 주소 0x00000054부터 연속적으로 r0~r4까지의 값이 저장되었다.**



**- sp 레지스터를 통해 스택메모리에 push(STORE), pop(LOAD)를 수행한다.**

**- stmfd는 Full Stack Descending으로써 sp가 가리키는 주소가 채워져 있으며 Push시 메모리의 주소가 감소하는 방향으로 스택에 채워진다. 따라서 push시 pre-decrease 되어야 한다.**

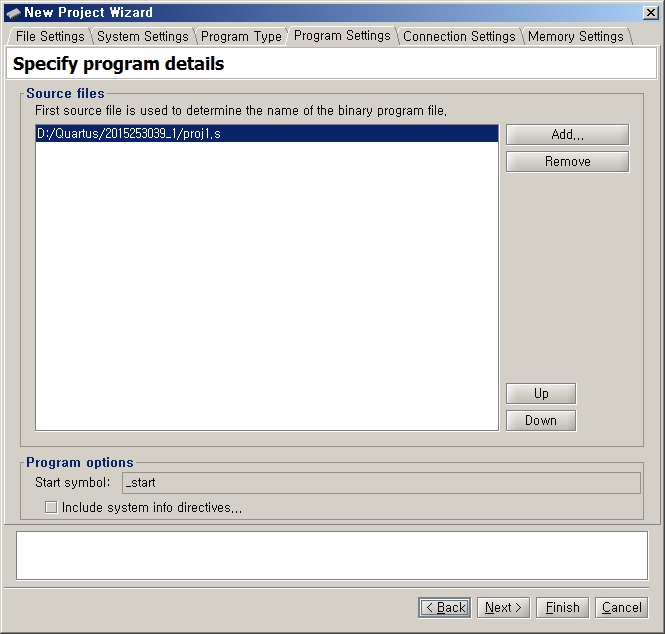
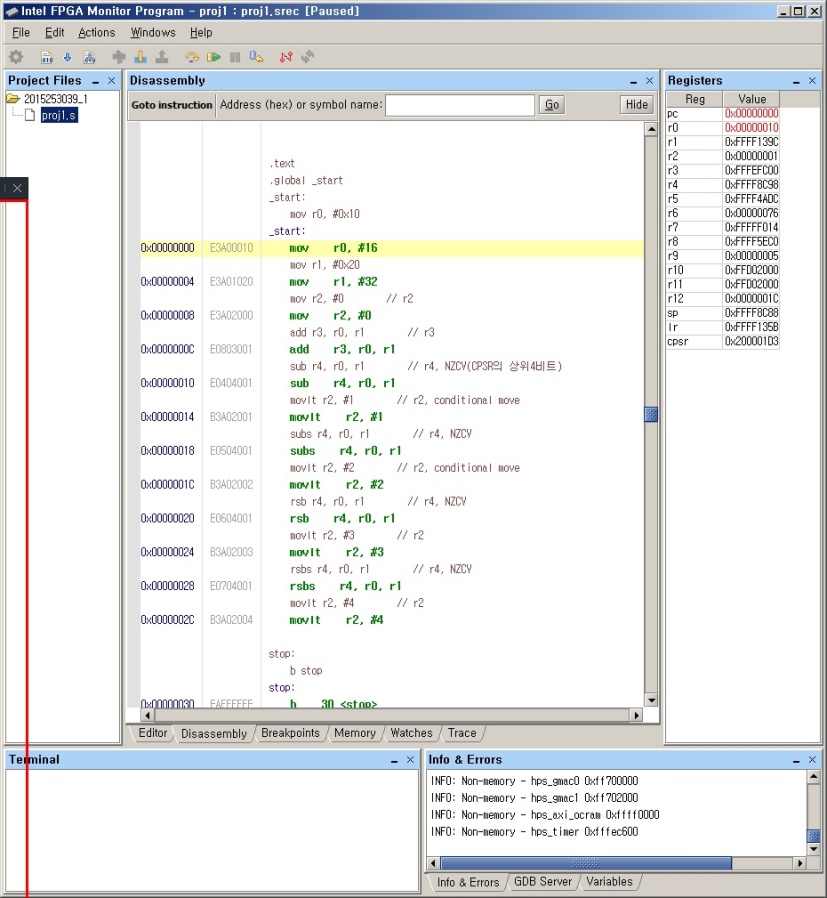
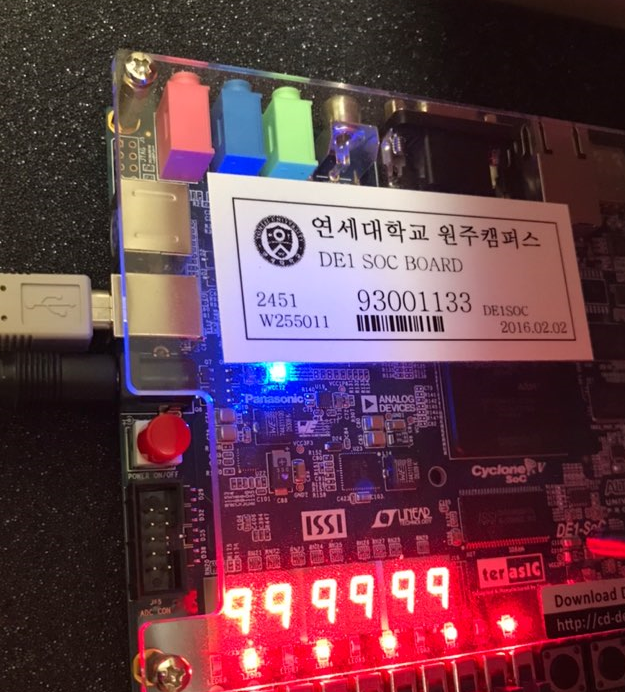


**- proj4의 최종 레지스터 결과 값이다.**

소스코드 분석 및 리뷰 끝

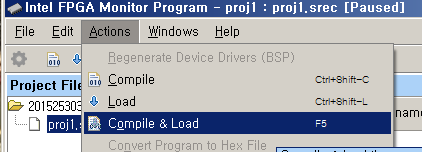
다음 장에 계속

**(2) Intel FPGA Monitor Program을 실행하고 다음과 같이 project lab1을 생성**

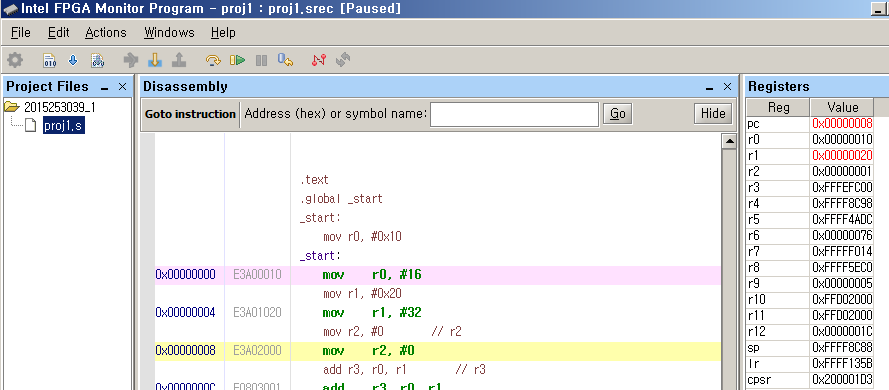


**Host PC와 ARM 기반 DE1 보드를 연결한 후 Monitor 프로그램을 통해 프로젝트를 구성 후 프로그램을 복사하여 컴파일 & 로드를 하여 프로그램 동작을 확인하였다.**

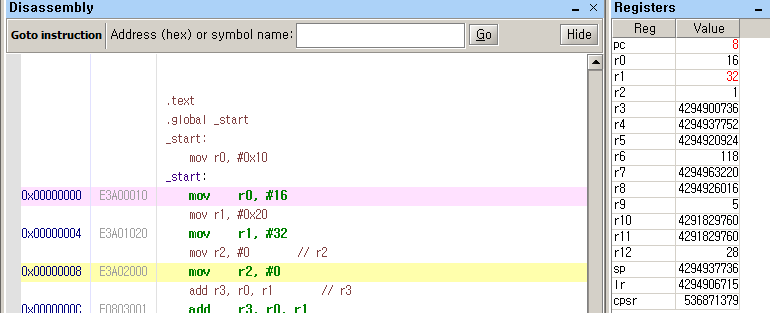
**(3) 프로그램 컴파일 및 적재: Actions > Compile & Load 선택 (단축키 F5)**

<- 단축키 F5로도 가능하다.

**(4) Disassembly 창에서 single step 실행을 통한 동작 확인**

<- Single Step

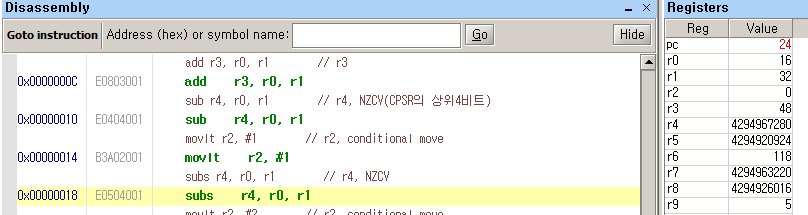
- Single Step 수행 과 Register의 Decimal 표기 변경



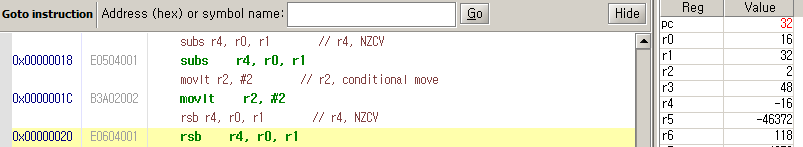
- Decimal 표기로 변경하여 보았고 현재 pc값은 0x00000008이다. mov r0, #16을 통해 r0 레지스터에 16이라는 값이 들어갔고 mov r1, #32를 통해 r1 레지스터로 32라는 값이 들어갔다.



- 현재 pc는 16(10진수)이며 16진수로는 0x00000010이다. 따라서 add r3, r0, r1이 진행된 상태이다. Register의 값을 확인해보면 r0 : 16, r1 : 32의 값이 들어있었으며 r3 레지스터가 dst이므로 48이라는 값이 들어갔다.

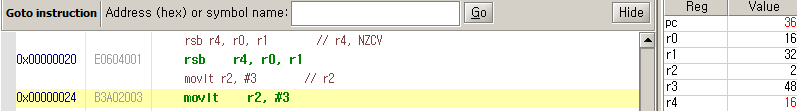


- 현재 movlt 명령어까지 수행된 상태이며 movlt의 경우 less than 이므로 a<b일 때 조건이 참이되어서 조건부 mov 실행이 일어나게 된다. 하지만 앞 전에 상태레지스터를 업데이트하는 "S" 접미사가 없었기 때문에 lt조건이 거짓이 되어 mov가 수행되지 않는다. 우측 r2를 보면 값이 0이다.



- subs 로 상태레지스터를 갱신하여 주었으며 r4에는 r0 - r1 = -16의 값이 들어가게 된다.

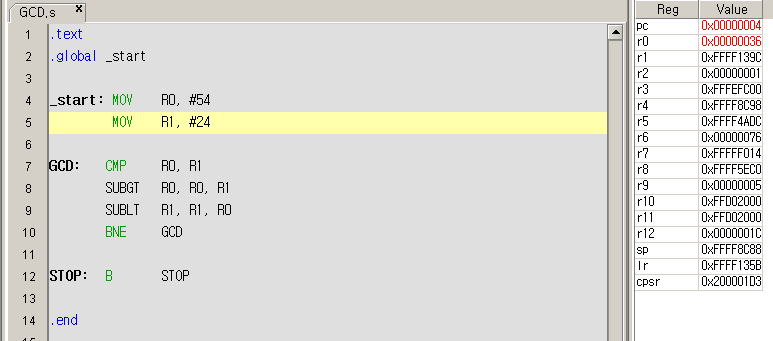
상태 레지스터가 subs명령어를 통해 갱신되었으므로 r0-r1=-16, 'N' bit가 1이 되므로 lt조건이 참이 된다. 따라서 mov가 수행되며 r2에는 '2'의 값이 들어간 것을 확인 할 수 있다.



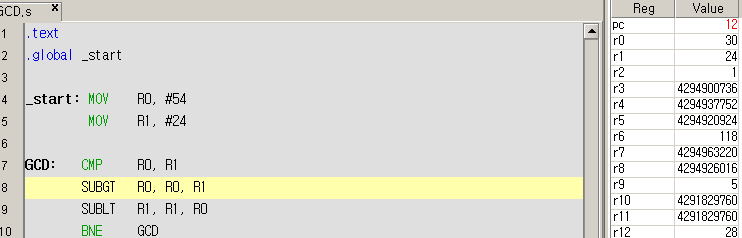
- rsb는 Reverse Sub이므로 r1에서 r0를 빼게 된다.

**2**. 강의자료(e5)의 Exercise 1 어셈블리 프로그램의 동작을 설명하고 다음과 같이 실행하시오.

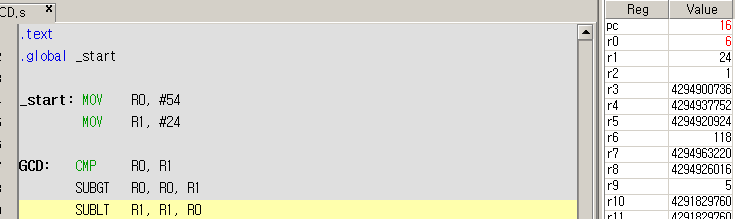
(1) Disassembly 창에서 single step 실행을 통한 동작 확인



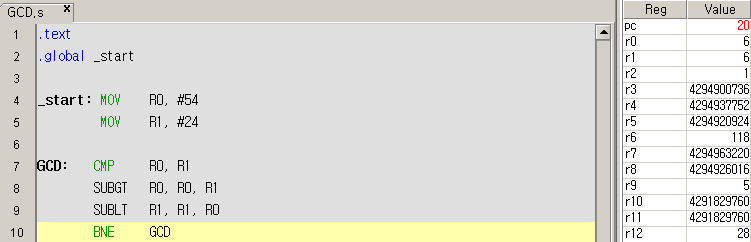
**- Single Step으로 동작 확인**



**- Decimal로 바꾸고 r0, r1 값 분석하기**

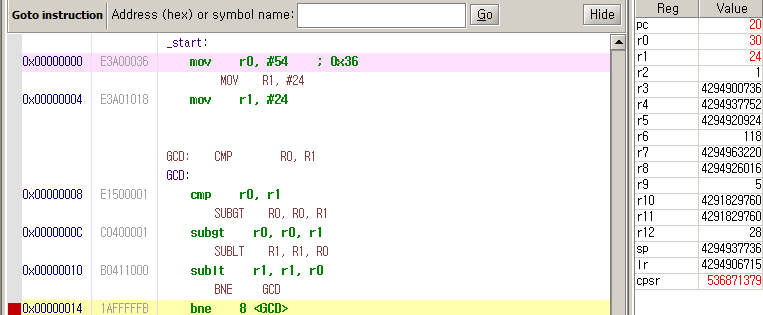


**- 위와 같이 r0 : 30, r1 : 24 였으므로 CMP는 "S" 접미사 없이도 상태 레지스터가 갱신된다. 따라서 30 - 24는 'N' bit가 0이므로 GT조건을 만족하여 R0 - R1을 수행하게 된다. 30-24를 수행한 값을 다시 R0으로 넣기 때문에 위와 같은 결과가 나타난다.**

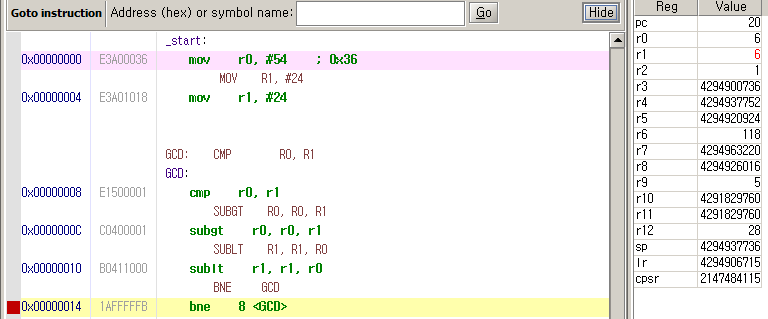


**- 최종적으로 r0, r1이 6이 되는 값에서 Equal이 되기 때문에 GCD 루프를 벗어나서 종료되게 된다. 최대공약수는 '6'이 된다.**

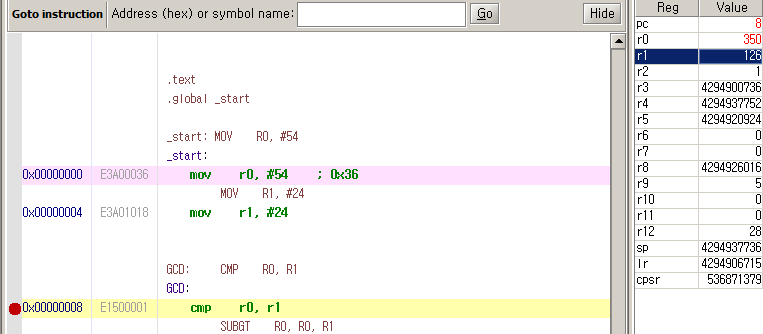
(2) Disassembly 창에서 breakpoint 설정 후 계속 실행



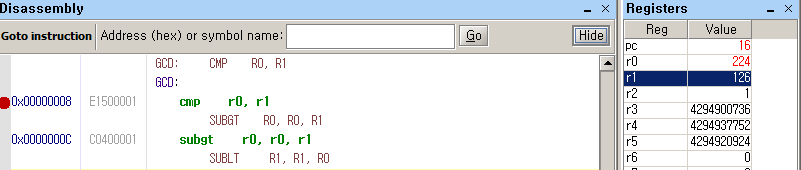
**- Break Point 설정**



**- r0, r1 값이 같아지며 최대 공약수는 6이다.**

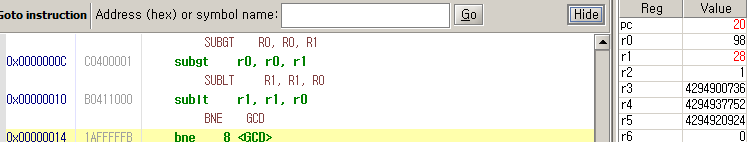


**- r0, r1 값을 350, 126으로 변경**

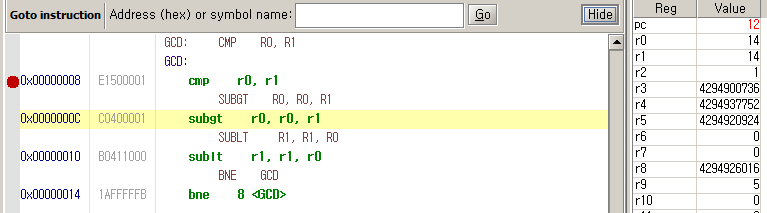


- subgt가 수행된 후의 r0, r1의 값은 350-126이 되어 r0는 224가 되었고 r1>r0인 경우이며 상태레지스터의 값은 아래 캡쳐와 같이 **최상위부터 4비트의 값이 2이므로 0010이다. NZCV순이므로 빌림수로 인하여 C만 1이 되었고 350-126=224이므로 N bit는 0, Z bit는 0, V bit도 0이다.**





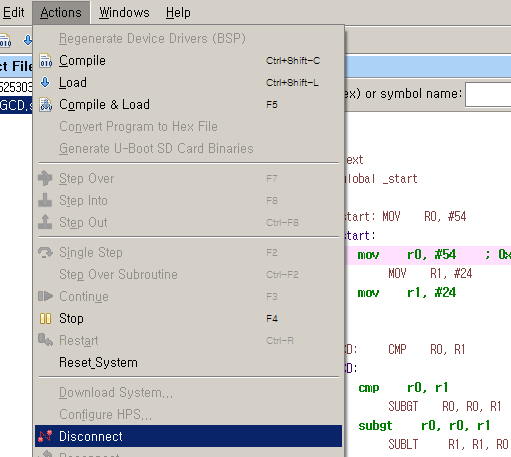
**r0<r1으로 sublt가 수행 된 후이며 r1에서 r0를 빼서 r1에 다시 넣었기 때문에 126-98 = 28이 들어가게 된다. 8 -> 1000 이므로 Negative bit가 '1'이 되었다.**



**- 다음과 같이 r0 = r1이 된 상황이며 아래 그림과 같이 Zero bit와 Carry bit가 '1'이 된다.**

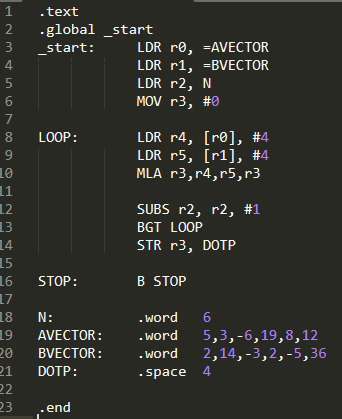
 **<- 6 -> 0110이므로 NZCV 순으로 Zero, Carry flag 가 1이 된다.**

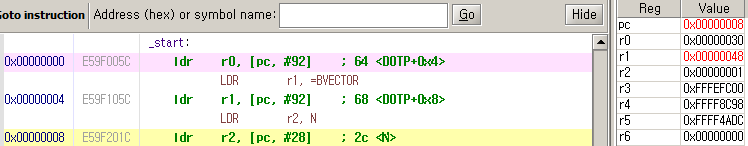
 <- **최종 결과 최대 공약수는 14이다.**



- **마지막으로 Disconnect하여 DE1보드와의 연결을 종료**하였습니다.

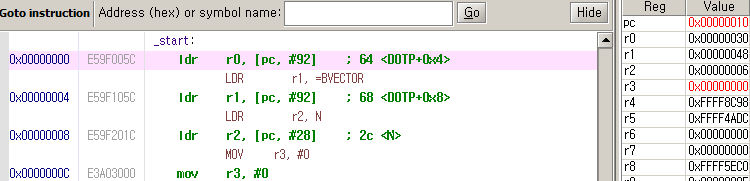
**3**. (메모리 사용 프로그램 실행하기) 강의자료(e5b)의 Exercise 2 (Dot Product 계산) 프로그램의 동작을 설명하고, 실행시켜서 동작을 확인하시오.

- 프로그램을 작성하였고 Vector 곱 연산 수행이다.

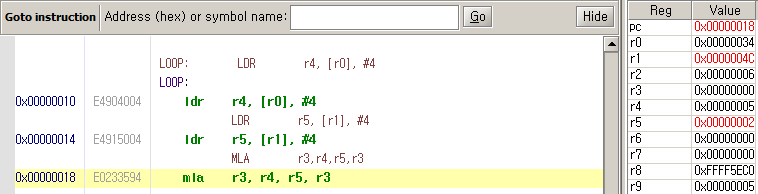




**- 먼저 AVECTOR와 BVECTOR의 시작주소를 ldr을 사용하여 r0과 r1에 넣는다. 주소는 pc값과 offset address를 사용하여 찾아진다. 예를 들어, N의 경우 0x0000002C의 주소에 위치하므로 0x00000008 + 8 + 28 = 44(10진수) ->2C(16진수)가 된다.**



**- r3 값에 0을 넣어줌으로써 초기화까지 완료**된다.

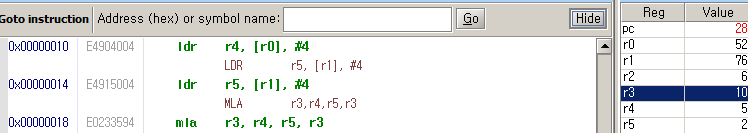


**- ldr연산을 통해 r4, r5에 r0, r1이 가리키는 주소의 값 즉, AVECTOR의 첫 원소, BVECTOR의 첫 원소를 가져오고 다음 루프에서 다음 Vector 값을 접근하기 위해 r0와 r1에 +4를 하는 ldr r4, [r0], #4연산을 한다.**

**- 이후 mla 연산을 통해 r3 = r4 \* r5 + r3를 수행하면서 아래 수식의 연산을 수행하게 된다.**

**r3는 0으로 초기화 되었으므로 각 Vector끼리의 곱을 저장하고 누적시키면서 최종 결과를 출력하게 된다.**

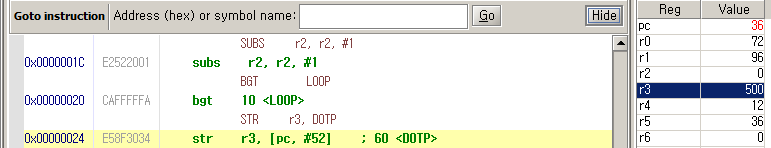




**- 첫 mla 연산이 수행 된 후 r4 \* r5 + r3 = 5 \* 2 + 0 = 10이 된 것을 확인할 수 있다.**



**r2 - r2를 함으로써 카운터 값을 -1수행하며 상태레지스터를 업데이트 시킨다. branch gt를 통해서 r2값이 0보다 크면 LOOP를 반복하는 동작을 수행한다. 즉, Counter(r2)는 VECTOR의 개수를 저장하고 있으며 LOOP마다 -1씩 수행함으로써 마지막 원소까지 진행 후 LOOP를 멈추게 한다.**



- **다음과 같이 r2가 0이 되었을 때(Zero flag = '1') branch gt조건이 불만족 되므로 루프를 탈출하여 str 명령어를 수행한다. str은 값을 저장하는 역할을 한다.**

**또한 r3에 저장된 최종 결과 값은 500이다.**



**- DOTP에 할당한 공간의 시작주소는 0x00000060이므로 해당 메모리주소로 가서 확인해 볼 수 있다.**

<- 메모리

**- 다음과 같이 0x000001F4 = 500 을 확인 할 수 있다.**

감사합니다.

2015253039 권진우