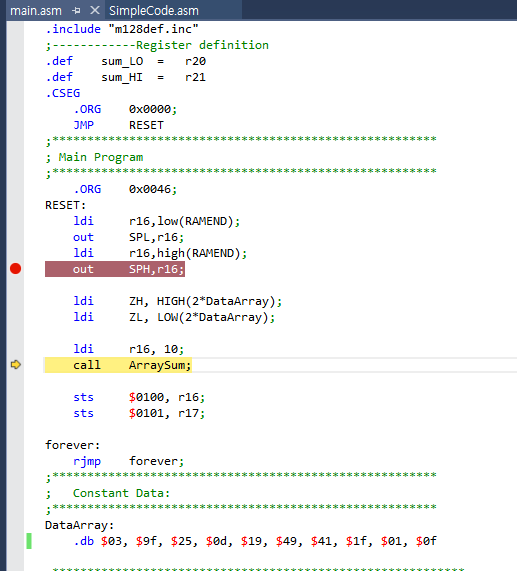
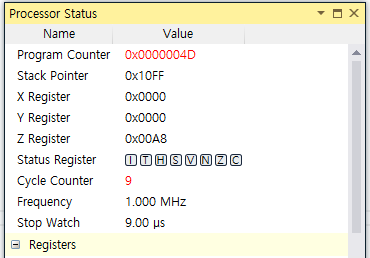
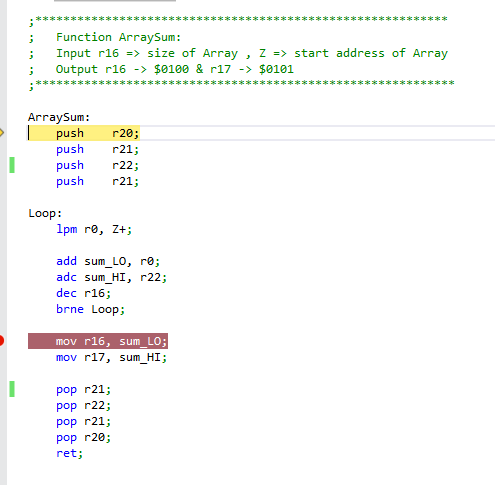
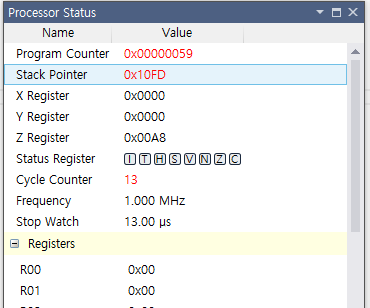
Lab 4-1



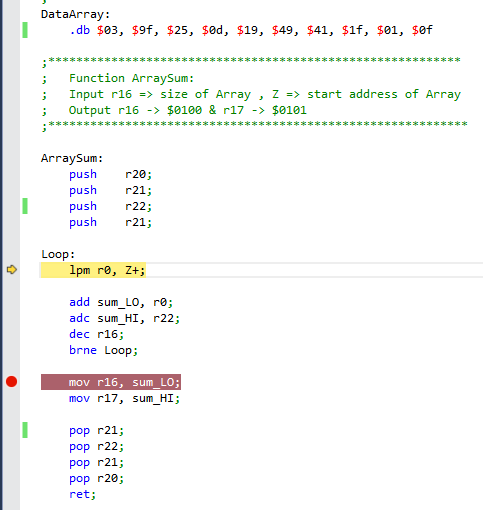


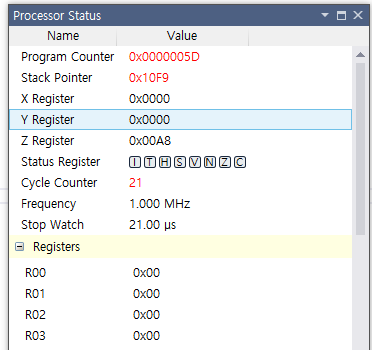
스택포인터, 프로그램메모리의 array를 가리키는 Z레지스터값, array 개수를 나타내는 10개를 r16레지스터에 저장한 상태이다.



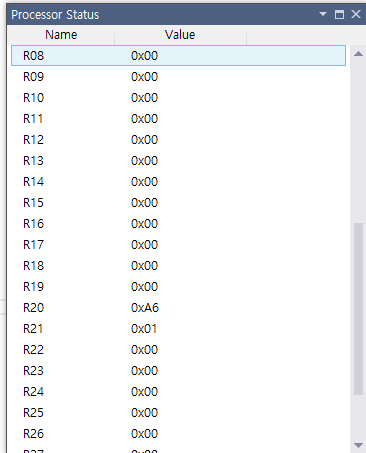
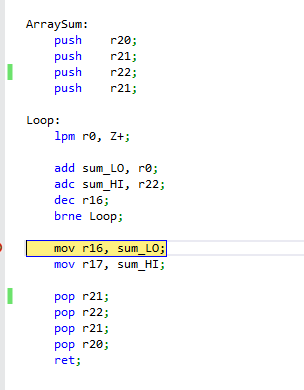


ArraySum 함수를 호출하였다. 따라서 스택포인터 값이 변경되었음을 확인할 수 있다.

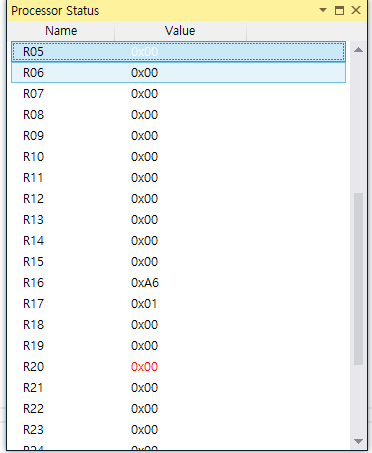
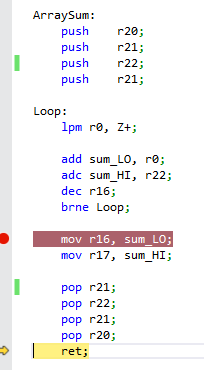




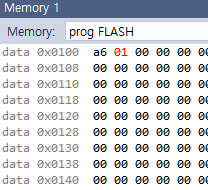
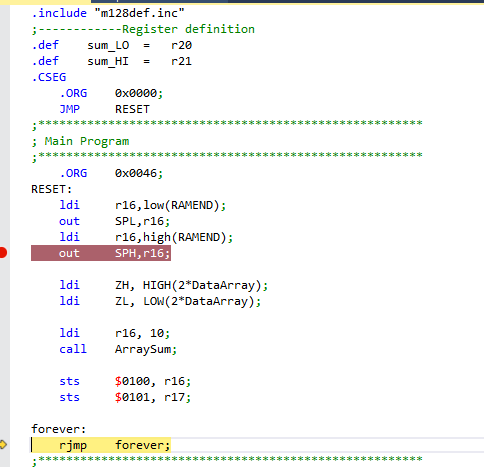
레지스터값들을 stack에 push하였기 때문에 스택포인터가 변했음을 확인할 수 있다.



다음의 루프를 통해 sum\_LO 에 해당하는 r20레지스터와 sum\_HI에 해당하는 r21레지스터의 DataArray값들의 합산된 값들이 저장되었음을 확인할 수 있다.

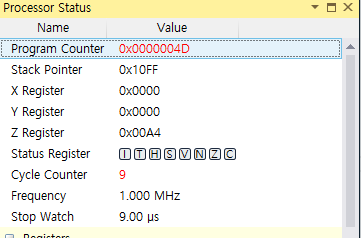
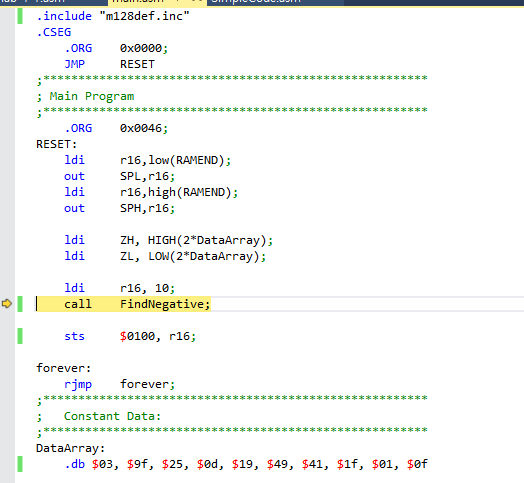


함수호출이 끝나기 전에 output에 해당하는 r16, r17레지스터에 합산한값들을 전달해주고 pop을 통하여 함수호출직전의 레지스터상태로 돌려놓는다.

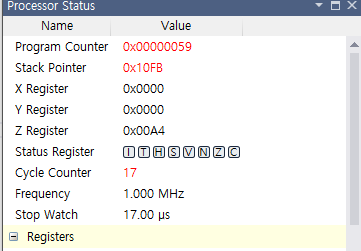
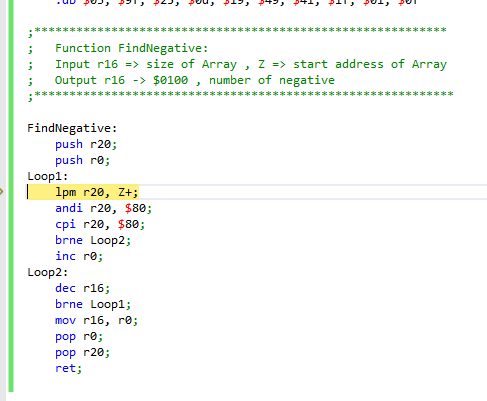


함수호출이 끝나고 다시 메인 프로그램으로 돌아와 sts 명령어를 수행하여 100번지 101번지에 해당하는 값들을 저장한다.

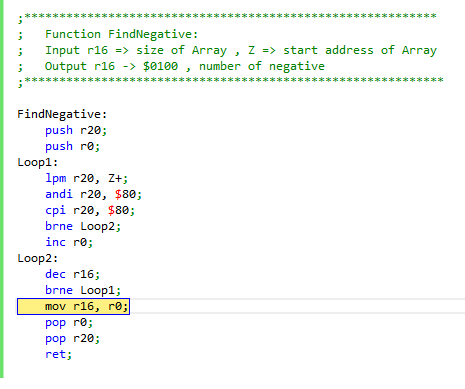
Lab 4-2

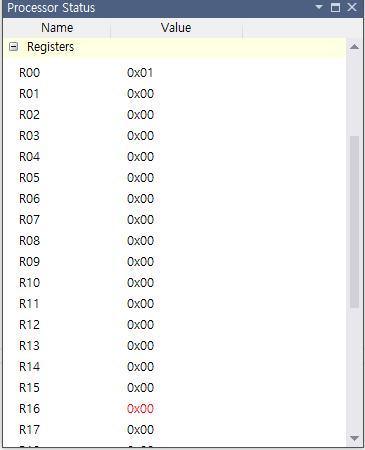


스택포인터, DataArray를 가리키는 Z레지스터, DataArray의 개수를 나타내는 r16레지스터를 설정하였다.



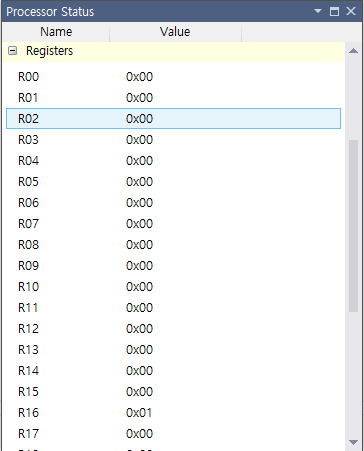
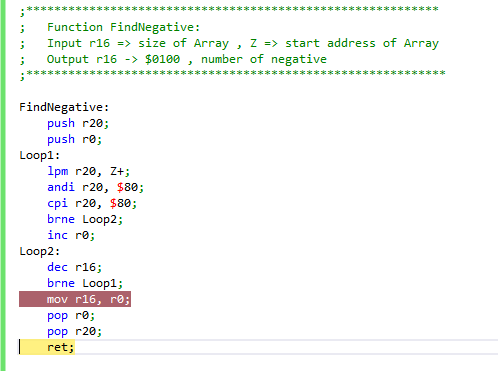
함수를 호출한 뒤 input, output이 아닌 값들을 스택에 push 하였다. 스택포인터가 바뀌었음을 확인할 수 있다.



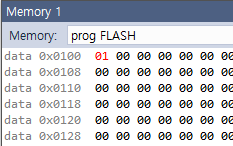
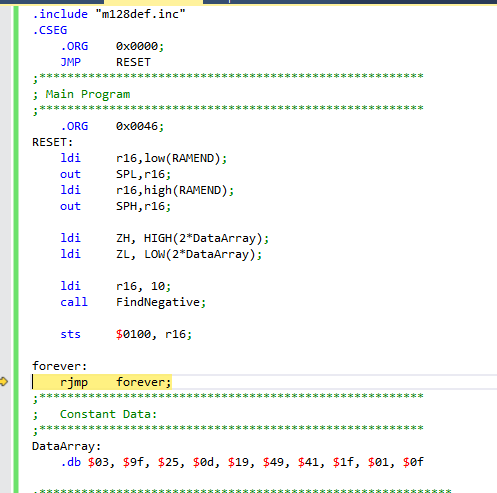


함수 내 루프와 andi, cpi를 통해 DataArray 내의 음수의 개수를 카운트한값을 r0에 저장하였다.

배열개수를 나타내는 r16값은 0이 되었음을 확인할 수 있다.

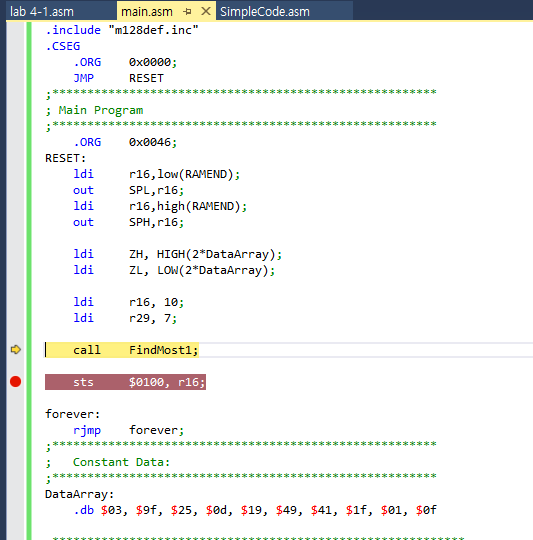


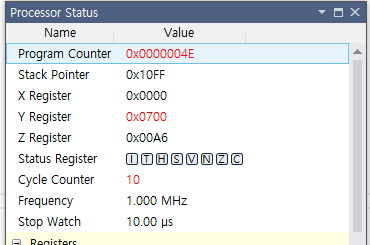
output을 r16을 사용할 것이기 때문에 r0값을 r16에 옮겨주고 pop을 통해 함수호출 이전에 레지스터값들로 복원을 해준다.



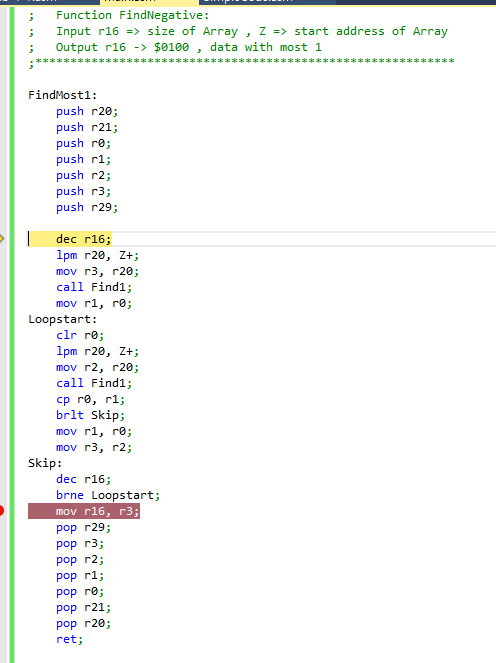
메인 프로그램으로 돌아와 100번지에 음수개수가 저장되었음을 확인할 수 있다.

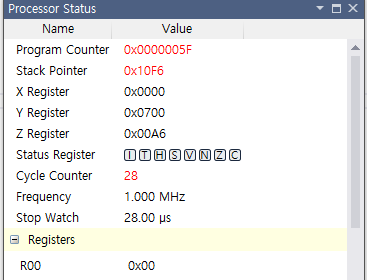
Lab 4-3



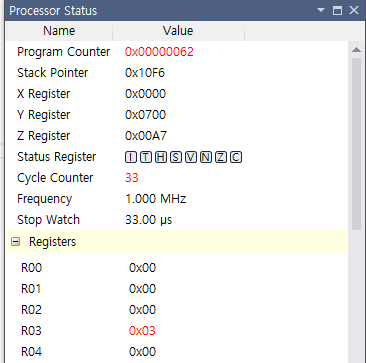
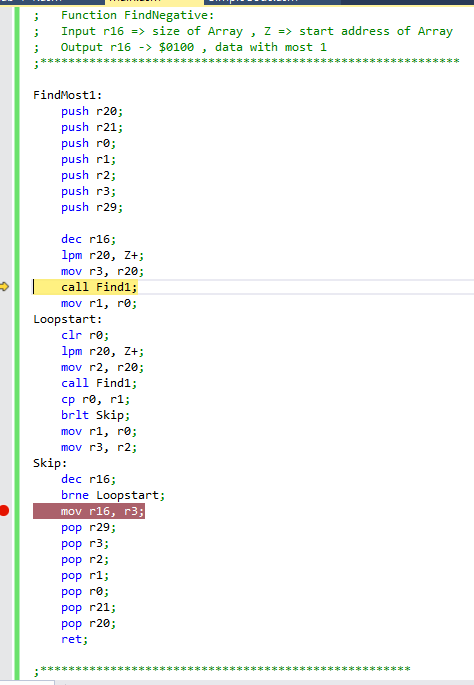


스택포인터, Z레지스터값과 필요한 변수값들을 r16과 r29에 저장하였다. r16은 DataArray개수를 나타내고 r29는 1을 찾을 때 필요한 변수이다.



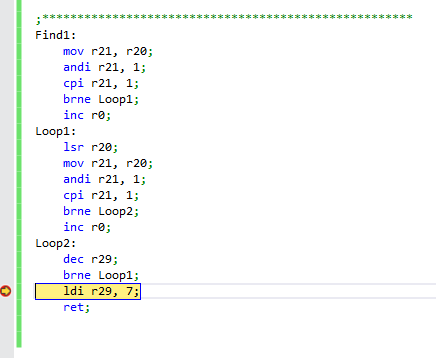


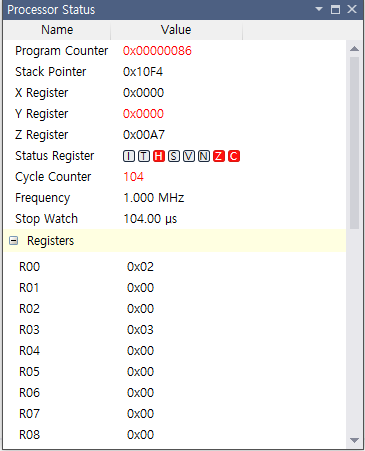
함수의 매개변수가 아닌 값들을 스택에 집어넣어준 모습이다. 바뀐 스택포인터를 통해 알 수 있다.



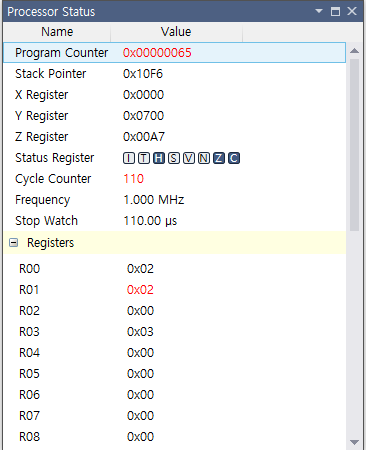
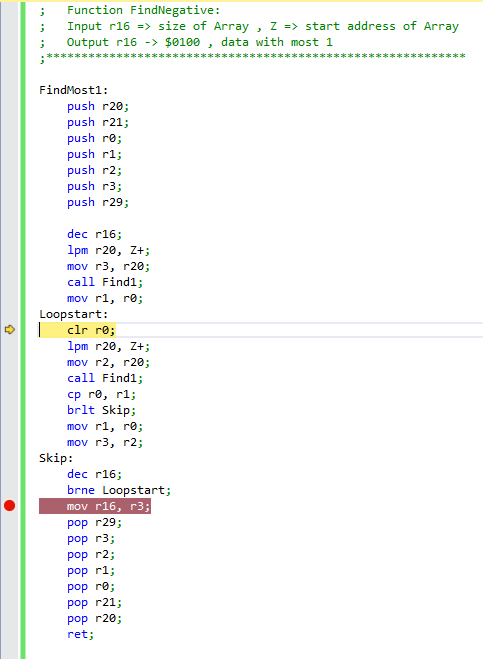
DataArray를 가리키는 Z레지스터를 통해 값을 받아오고 그 값을 r3에 저장하였다.

그 후 Find1 , 1의 개수를 세는 함수를 호출할 것이다.

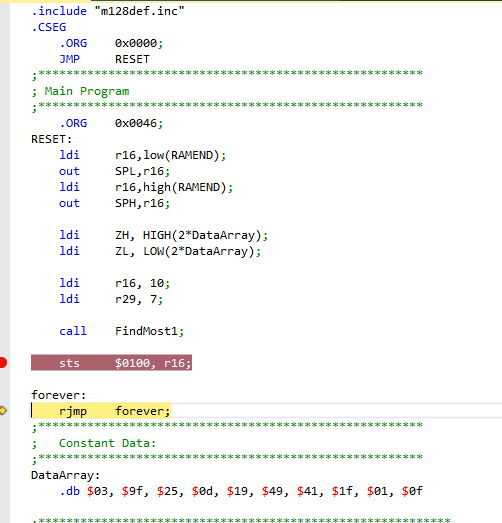


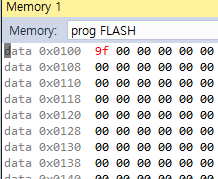


호출된 Find1함수는 andi, cpi, lsr 명령어를 통해 1의 개수를 세고 그 개수를 r0에 저장한다.



1의 개수를 r1에 저장한다 . 이로써 r1에는 이전 DataArray의 1의개수, r3에는 그에 해당하는 값이 저장된다. 다음의 동작을 모든 DataArray 값들에 대해 반복한다. 만약 이전 데이터값의 1의개수보다 이후 데이터값의 1의 개수가 많다면 r1과 r3을 이후 데이터값으로 바꿔준다. 모든 DataArray값들을 비교하고 난 뒤 마지막으로 mov명령어를 통해 r16 output에 r3값을 붙여넣는다.

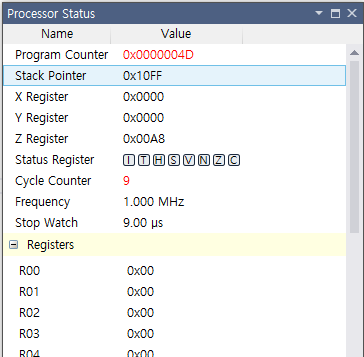
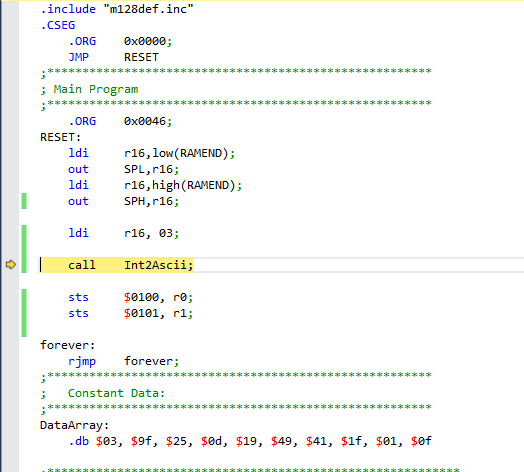




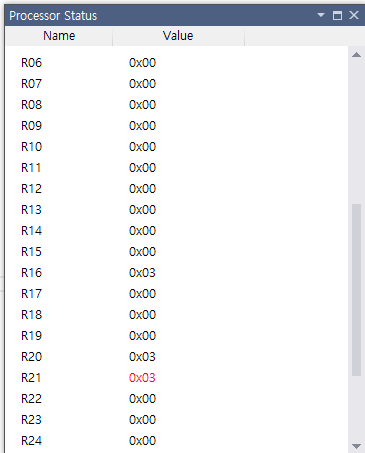
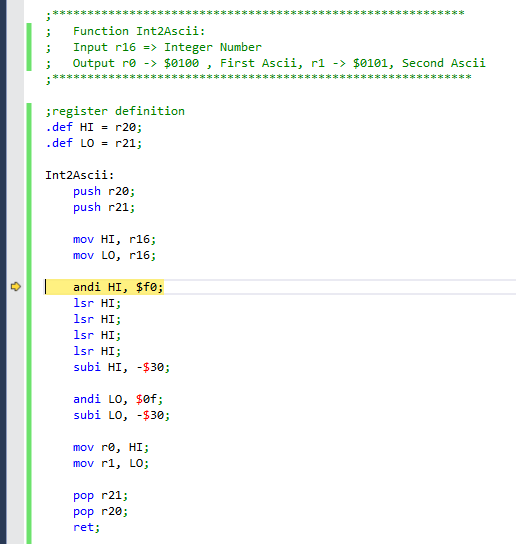
FindMost1함수가 종료되고 난 뒤 메인 프로그램으로 돌아온다.

데이터 100번지에 가장 1이 많은 데이터가 저장됨을 확인할 수 있다.

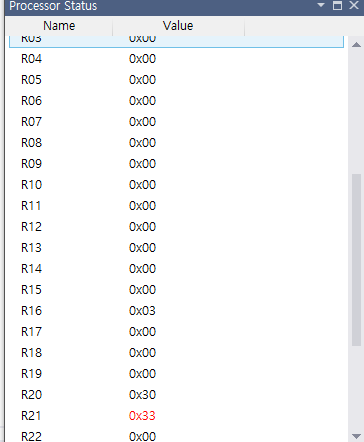
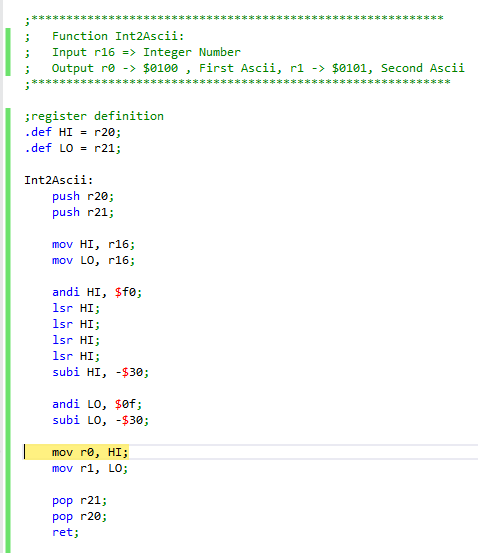
Lab 4-4



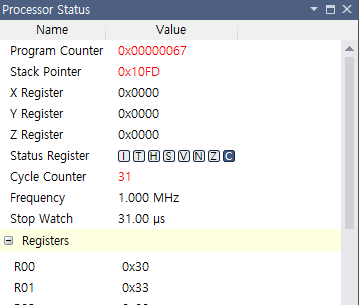
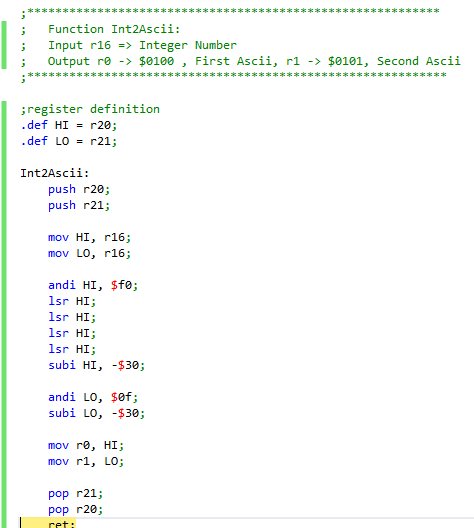
스택포인터와 아스키코드화 할 변수 03을 r16에 저장해주었다.



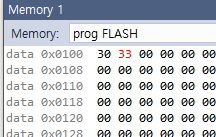
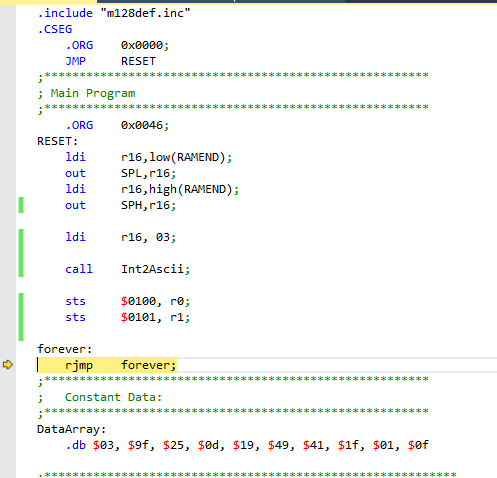
함수를 호출한 뒤 HI, LO 레지스터에 r16레지스터값을 넣어주었다. 또한 매개변수가 아닌 레지스터들은 스택에 저장하였다.



HI 레지스터는 상위비트 and masking 후 4칸 우측으로 쉬프트후 0x30을 더해주었고 LO 레지스터는 하위비트 amd making 후 0x30 을 더해주었다. 0x30을 더해준 이유는 아스키 코드 ‘0’값이 0x30이기 때문이다.

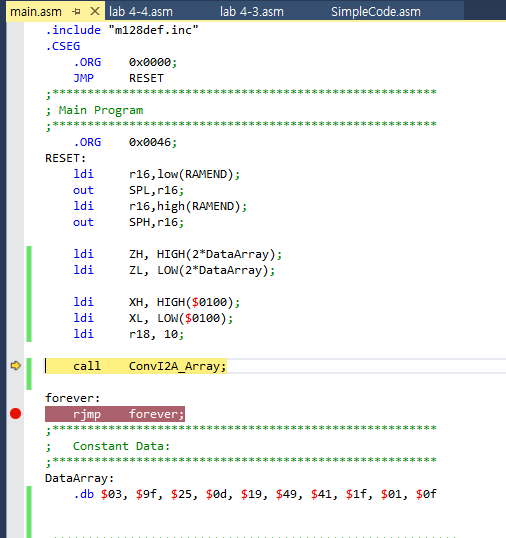


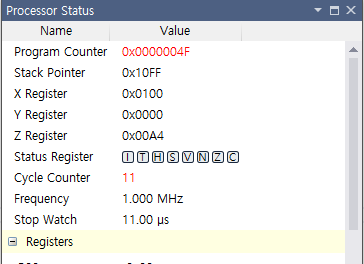
output이 되는 r0, r1에 HI, LO 값을 넣어준다.



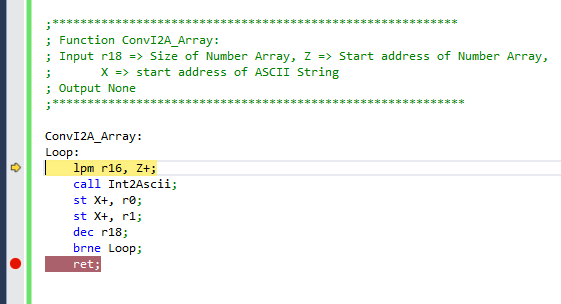
메인 프로그램으로 돌아와 100번지 101번지에 해당하는 값을 저장했음을 확인할 수 있다.

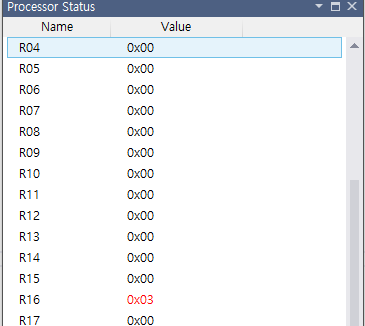
Lab 4-5



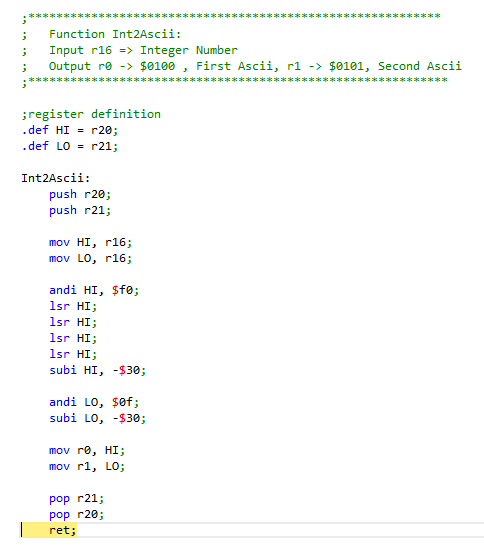


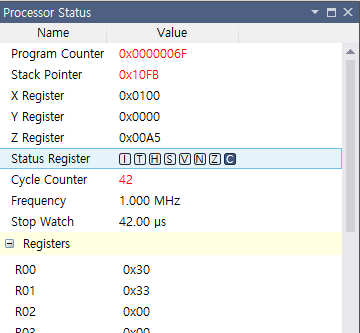
스택포인터와 DataArray값을 불러들일 Z레지스터, datamemory에 저장하기 위한 X레지스터, DataArray 의 개수를 나타내는 r18레지스터를 설정해주었다.



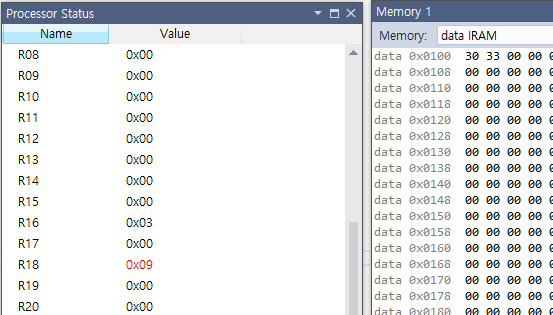
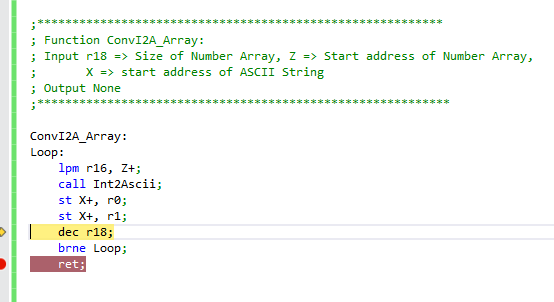


ConvI2A\_Array 함수를 호출한 뒤 r16레지스터에 DataArray값을 넣어주었다. 이 r16레지스터값은 다시 Int2Ascii 함수의 input이 되어 r0, r1이 output으로 나오게 된다.

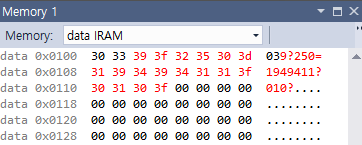
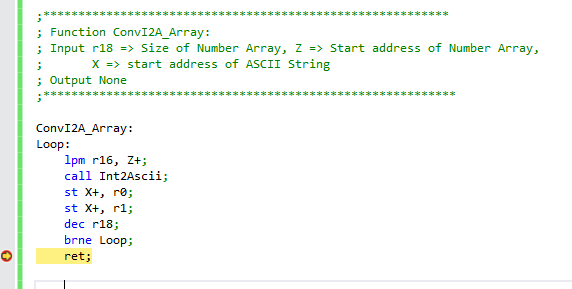




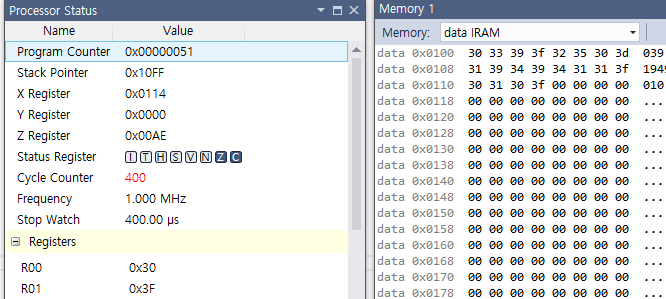
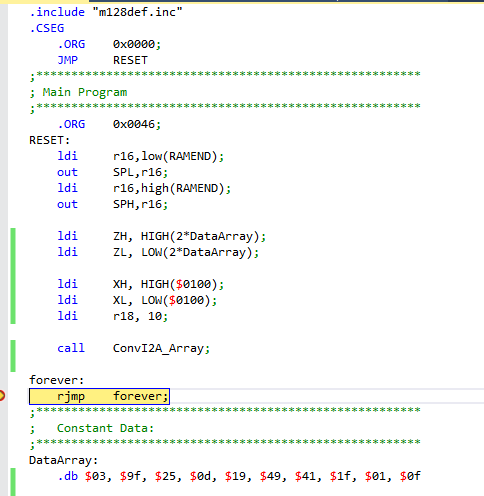
lab 4-4에서와 같이 r16으로 들어온 값은 r0 , r1에 아스키코드값으로 변환되어 반환된다.



ConvI2A\_Array 함수로 돌아와 r0, r1값을 X레지스터가 가리키는 data memory에 집어넣어주고 r18에 있는 개수만큼 동작들을 반복하게 된다.



모든 동작을 마치면 다음과 같이 data memory 공간에 DataArray 값들이 아스키코드화되어 저장되게 되고 함수는 루프를 빠져나간다.



메인 프로그램으로 돌아간 모습으로 데이터가 모두 저장되었음을 확인할 수 있다.

고찰

저번 lab3 과제는 루프를 이용하여 여러가지 동작들을 수행하였다. 이를 위해 코드를 작성하면서 이유모를 답답함을 많이 느꼈었다. 왜냐하면 c언어의 경우 함수를 호출하여 간단하게 반복진행할 수 있을 동작들을 루프를 이용하여 억지로 반복한다는 느낌을 받았었기 때문이다. 특히 요구하는 동작이 조금씩 복잡해질수록 루프가 굉장히 많이 필요했는데 이는 코드의 가독성을 떨어뜨리고 빌드 과정에서 이유모를 오류들이 발생하기 쉽상이었다. 이번 과제를 수행하는데 사용한 함수는 이러한 단점들을 모두 없애준다는 점에서 굉장히 매력적이고 강력한 기술이라고 생각된다. 특히 함수의 가독성이 굉장히 올라가 코딩을 하는사람도 오류가 줄어들고 코드를 읽는 사람도 쉽게 읽고 이해할 수 있다는 점이 좋았다. 다만 함수를 호출할 때 함수의 input , output 이 되는 값들을 잘 기록하여 레지스터가 중간에 남용되지 않도록 해야한다. 또한 함수를 호출하였다면 매개변수가 되는 레지스터를 제외하고는 스택을 이용하여 진행정보가 손실되지 않도록 주의하여야 한다.