2.1. 둘러보기

inline assembly에서 정해주어야 하는 것들은 다음과 같습니다.

- assembly 코드
- output 변수들
- input 변수들
- output외에 값이 바뀌는 레지스터들

```
__asm__ __volatile__ (asms : output : input : clobber);
```

asms

> 쌍따옴표로 둘러싸인 assembly 문자열. 문자열안에서 %n 형태로 input, output 인자들을 사용할 수 있으며 인자들이 치환된 후 그대로 컴파일 된 assembly에 나타납니다.

output

> 쉼표로 구분된 "constraint" (variable)들의 리스트이며 각각이 이 inline assembly에서 쓰이는 output 인자를 나타냅니다.

input

> output과 같은 형태이며 input 인자들을 나타냅니다.

clobber

> 쉼표로 구분되는 쌍따옴표로 둘러싸인 레지스터 이름들의 리스트이며 input, output에 나오진 않았지만 해당 assembly를 수행한 결과로 값이 바뀌는 레지스터들을 나타냅니다.

ouput, input, clobber는 비어있다면 뒤에서 부터 생략될 수 있습니다. 하지만 앞에 오는 파라미터가 비어있을 때는 :로 표시를 해주어야 합니다. 즉,

```
__asm__ __volatile__(asms : output : input);  /* clobber 없을 때 */
__asm__ __volatile__(asms : : input);         /* output, clobber 없을 때 */
```

의 형태로 생략 가능합니다.

__asm__ 키워드는 asm으로도 쓸 수 있지만 ansi옵션으로 컴파일하게 되면 asm은 정의되어있지 않기 때문에 __asm__으로 쓰는 것이 좋습니다.

__volatile__은 해당하는 inline assembly를 optimization으로 없애거나 위치를 바꾸지말라는 뜻입니다. GCC manual에 따르면 side effect가 없다고 여겨지는 경우 assembly를 없애거나 loop의 밖으로 빼는 optimization을 할 수 있다고 합니다. 예를 들어 output이 있지만 실제로 output으로 쓰인 변수가 그 이후로 한 번도 쓰이지 않았다면 그 inline assembly는 프로그램의 수행에 아무런 영

향을 끼치지 않는다고 생각하고 없애버리는 것입니다. 물론 조건을 정확하게 정해주면 굳이 __volaitile__을 붙이지 않더라도 제대로 작동하겠지만 가끔씩 엉뚱하게 되버리는 경우도 있기때문에 잘 생각해서 inline assembly를 쓰고 __volatile__을 붙여주는 것이 좋습니다.

실제로 input, output이 쓰인 예를 보겠습니다.

```
int test_and_set_bit(int nr, volatile unsigned * addr)
{
        int oldbit;

        __asm__ __volatile__(
                "lock; btsl %2,%1₩n₩tsbbl %0,%0"
                :"=r" (oldbit),"=m" (*addr)
                :"r" (nr));
                return oldbit;
}
```

asms

```
lock; btsl %2, %1
sbbl %0, %0
```

output

```
"=r" (oldbit), "=m" (*addr)
```

input

```
"r" (nr)
```

asms에서 ₩n₩t 대신 ;을 적어도 되지만 gcc에 -S옵션을 주어 assembly output을 볼 때에 나머지 부분과 줄을 맞추려면 각 인스트력션 사이를 ₩n₩t로 구분해주는 것이 좋습니다. %0 %1 %2 각각은 인자들을 나타내는데 output, input에 있는 순서대로 번호가 주어집니다. 즉, %0은 oldbit, %1은 *addr, %2는 r이 됩니다. 위의 assembly는

```
lock; btsl nr, *addr
sbbl oldbit
```

와 같은 의미입니다. 하지만 instruction에 따라서 인자로 무엇을 쓸 수 있는 지 제약이 있습니다. 예를 들어 btsl의 경우에는 첫번째 인자는 범용 레지스터만을, 두번째 인자로는 범용 레지스터나 memory상의 변수가 될 수 있습니다. 따라서 gcc에게 어떤 인자들을 inline assembly에서 쓰겠다는 것 뿐만아니라 그 인자들이 어디에 있어야 하는지도 정해주어야 합니다. 이것을 constraint에서 정해줍니다.

output을 보면 oldbit, *addr로 인자를 정해주었고 oldbit에 대해서는 "=r", *addr에 대해서는 "=m"을 constraint로 주었습니다. r은 범용 레지스터를 뜻하고 m은 memory operand를 뜻합니다. 즉, oldbit과 *addr은 output으로 쓰이며 oldbit은 범용 레지스터여야하고 *addr은 memory operand이어야 한다는 뜻입니다. Output 인자의 경우엔 항상 =를 constraint에 포함시켜야하는데 이것은 이 인자의 값은 inline assembly의 결과로 바뀔 수 있다라는 것을 뜻합니다.

Input도 같습니다. nr은 input으로 쓰이며 범용 레지스터이어야 한다는 constraint를 가지고 있습니다. 하지만 input의 경우에는 =이 없습니다.

여러개의 constraint를 같이 쓸 수도 있습니다. 즉, "ir" (nr) 처럼 쓸 수 있습니다. 이렇게 쓰면 주어진 여러개의 constraint중 하나를 만족하면 된다는 뜻으로 "ir"은 immediate operand나 범용 레지스터중 하나면 된다는 뜻입니다.

위의 함수는 이름처럼 addr로 주어진 word의 nr번째 bit을 atomic test and set합니다. 보통 spin lock이나 semaphore등의 synchornization construct들을 만들 때 쓰입니다. 기능을 생각해보면 addr의 constraint가 왜 범용 레지스터거나 memory operand가 아니라 memory operand로만 고정이되어 있는 지 알 수 있습니다. 만약 범용 레지스터로 할당되어 버리면 *addr에 있는 값을 할당된 레지스터로 load한 후에 btsl과 sbbl이 수행되고 그 결과값이 다시 *addr로 store되므로 atomic하지 않게 되버립니다.

Gcc는 constraint에 따라서 각각의 인자들을 할당한 후에 필요하면 input 변수들을 할당된 곳에 load하는 code를 생성하고 inline assembly에 %n 형태의 변수들을 할당된 실제 변수로 치환해서 code를 내어놓습니다. 컴파일러는 inline assembly가 실행된 후에 그 결과값들이 어디에 있는 지 알고 있으므로 그 이후의 컴파일을 계속 진행할 수 있습니다.

그럼 위의 함수가 실제로 컴파일 되었을 때 어떤 결과가 나오는 지 보겠습니다.

```
.globl test_and_set_bit
        .type    test_and_set_bit,@function
test_and_set_bit:
        movl 4(%esp),%eax
        movl 8(%esp),%edx
#APP
        lock; btsl %eax,(%edx)
        sbbl %eax,%eax
#NO_APP
        ret
```

호출하는 부분에서 stack에 addr, nr, return address를 push하고 test_and_set_bit으로 control이 넘어오면, nr을 eax에 addr을 edx에 load한 후 #APP, #NO_APP사이의 inline asembly가 실행됩니다.

btsl의 첫번째 인자 %2는 nr이 load된 %eax로, 두번째 인자 %1은 addr이 load된 %edx의 indirect addressing인 (%edx)로, sbbl의 인자인 %0는 %eax로 치환된 것을 알 수 있습니다. Return 값은 크기가 맞는 경우 %eax를 통해서 가고 inline assembly 실행 후 return할 값이 이미 %eax에 있으므로 그냥 ret를 실행합니다.

%0와 %2가 같은 레지스터로 할당되었는데, gcc는 기본적으로 모든 input변수들은 output 변수가 사용되기 전에 모두 사용된다고 생각해서 겹치게 할당할 수도 있습니다. 위의 경우에선 %2가 %0가 사용되기 전에 사용되었으므로 문제가 없지만 그렇지 않은 경우엔 output 변수의 constraint에 '&'를 더해 early clobber를 정해주어야 합니다. Early clobber에 대해선 output 변수 항목에서 설명하겠습니다.

이제 전체를 한 번 보았습니다. 그다지 복잡하지 않지요? 이제 각 부분에대해 자세히 알아보도록 하겠습니다.

이전
Inline assembly basics

처음으로
위로

다음
Assembly