GCC inline assembly guide

허태준
아라기술 (http://www.aratech.co.kr)

GCC inline assembly guide

지은이 허태준

펴냄 2001년 10월 4일

이 문서는 gcc에서 inline assembly를 설명합니다. 설명이나 예제는 모두 ix86을 기준으로 하며 쓰이는 언어는 c입니다. GCC manual과 linux/freeBSD source 및 자신의 경험을 바탕으로 썼습니다. 사용방법정도의 설명보다는 사용예와 응용들을 보여주고 설명하는 것을 목표로합니다.

이 문서를 읽기위해서는 C를 능숙하게 다룰 수 있으며 processor(ix86)와 assembly에 대해 어느정도는 알고 있어야 합니다. (todo: 레퍼런스)

이 문서의 최신 버전은 이곳 (http://home-tj.dyndns.org/programming/) 에서 구할 수 있으며 틀린 점 지적, 제안, 질문등은 여기 (http://home-tj.dyndns.org/ez2000/ezboard.cgi?db=forum_misc) 에 하시기 바랍니다.

고친 과정

고침 0.12001-10-04고친이 tj

최초작성

차례

1. 서문 4
1.1. 저작권 정보 4
1.2. 책임의 한계 4
1.3. Inline assembly? 4
2. Inline assembly basics 6
2.1. 둘러보기 6
2.2. Assembly 9
2.3. Output/Input List 13
2.4. Clobber list 19
3. Applications 24
3.1. Atomic bit operations & spin lock 24
3.2. Function call with stack switch 27

1장. 서문

1.1. 저작권 정보

1.2. 책임의 한계

본 저자는 문서의 내용이 야기할 수 있는 어떠한 결과에 대해서도 책임을 지지 않습니다. 본 문서에서 내포하고 있는 정보들 및 예제들은 여러분이 알아서 활용하십시오. 비록 최선을 다 했으나 이 문서는 틀린 점이나 오류가 있을 수도 있습니다. 만약 여러분이 틀린 점을 발견했다면 꼭 저에게 알려 주시기 바랍니다.

1.3. Inline assembly?

System programming을 하거나 성능이 중요한 프로그램의 tight loop을 짤 때 assembly를 써야 하는 경우가 생깁니다. 즉, 프로그램의 대부분을 C나 C++등으로 만들고 고급언어로는 할 수 없거나, 그 부분을 assembly로 써서 속도 향상이 가능할 때 그 부분만을 assembly로 만들어서 C로 (C/C++ 모두 적용되지만 앞으로는 그냥 C라고만 하겠습니다) 만든 나머지 부분과 같이 쓰게 됩니다. C로 된 부분과 assembly로 된 부분을 같이 동작시키는 데는 크게 두 가지 방법이 있는데 하나는 assmbly로 쓴 부분을 독립된 함수로 만들어 따로 어셈블 한 후에 오브젝트 화일을 링크시키는 방법이 있고, 나머지 하나는 inline assembly를 쓰는 방법이 있습니다.

따로 assembly화일을 만들면 C의 함수호출방식에 맞추어 함수의 entry와 exit에서 인자들을 받고 return값을 돌려주는 부분만 신경쓰면 됩니다. gcc(gasm)외의 nasm 같은 다른 assembler를 쓸 수도 있으므로 어느정도 크기가 되는 부분을 assembly로 작성할 때는 이 방법을 쓰는 것이 좋습니다.

하지만 많은 경우에 전체 로직의 극히 일부분에서만 assembly가 필요하고 특히 compiler가 사용하지 못하는 processor의 특정한 기능을 쓰기위해 assembly를 쓸 때는 적게는 몇개, 많아도 이삼십개 정도의 instruction만을 assembly로 만들면 되는 경우가 대부분이고 이를 위해서 따로 함수를 만들어 링크하는 것은 번거로운데다가 자주 호출되는 경우라면 부가적인 function entry/exit 때문에 성능에도 좋지않습니다. 이런 경우에 inline assembly를 쓰게됩니다. 우선 어떤 것인지 감을 잡기 위해 예제를 보겠습니다.

```
void die(void)
{
```

```
dump_processor();
printk("halting the machine\n");
__asm__ __volatile__("cli; hlt;");
}
```

dump_processor, printk의 호출까지는 일반적인 C함수의 모습을 가지고 있습니다. __asm__으로 시작하는 부분이 inline assembly인데 괄호안의 문자열 "cli; hlt;"가 compile을 된 assembly 코드의 해당하는 자리에 그대로 출력이 되어 같이 어셈블됩니다. cli는 clear interrupt이고 hlt 는 halt입니다. 즉 cli; hlt;의 두 인스트럭션을 수행하게 되면 프로세서는 귀를 막고 잠들게 됩니다. 즉 위의 함수는 프로세서의 상태를 dump하고 멈출 것이라는 것을 출력한 다음에 기계를 멈춥니다.

위의 경우처럼 몇개의 특수한 instruction을 써야하는 경우에 매우 간편하게 쓸 수 있습니다. 이외에도 inline assembly를 시용하면 C의 변수를 assembly에서 쓸 수 있고 레지스터들을 어떻게 다룰 지를 지정할 수도 있습니다. 차차 살펴보도록 하겠습니다.

2장. Inline assembly basics

우선 둘러보기에서 예제를 통해 어떻게 생겼고 각각의 부분의 의미는 무엇이며 gcc가 컴파일했을 때 어떤 결과를 내어놓는 지를 간단히 살펴본 후 각각의 부분에 대해 자세히 설명하겠습니다.

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 정의되어 있지 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 영향을 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 인스트럭션 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.

2.2. Assembly

이 섹션에서는 gcc inline assembly에서 실제 assembly를 적는 부분 (앞으로는 이 부분을 asms라고 부르겠습니다)에 대해 설명합니다.

asms의 내용은 그대로 컴파일 된 assembly와 함께 gasm으로 넘어가므로 gasm의 문법을 따라야 합니다. Gasm은 target 인자가 뒤에오는 AT&T 문법을 따르며 instruction 사이의 구분은 세미콜론이나 개행문자로 하고 레지스터들은 %register의 형태로 표현합니다. ix86 계열의 대부분의 어셈블러와 intel manual은 target 인자가 앞에오는 intel 문법을 따르고 있으므로 manual을 보거나 다른 어셈블러의 코드를 볼 때 주의하시기 바랍니다. 더 자세한 내용은 gasm 메뉴얼과 intel processor manual을 참조하세요.

2.2.1. 들여쓰기 & 커멘트 달기

Gcc가 생성하는 assembly 코드는 심볼 정의등을 제외하고는 모두 탭 하나만큼 들여쓰기가 되어있습니다. Inline assembly는 #APP와 #NO APP사이에 들어가는데 탭 하나만큼 들여쓰기가 된 상태에서 문자열이 그대로 들어갑니다. 따라서 Assembly output을 읽기 쉽게 하기 위해1선 각 instruction들 사이를 \n\t로 구분해주면 됩니다.

```
__asm__ __volatile__("line1\n\tline2\line3");
```

을 컴파일하면

```
#APP
line1
line2
line3
#NO_APP
```

가 됩니다. 그런데 내용이 많아지면 위처럼 한 줄로 적기가 힘들어집니다. 그런 경우엔 아래 처럼 하면 됩니다.

```
__asm__ __volatile__(
"pushl  %%ebp # comment \n\t"
"movl   %%esp, %%eax \n\t"
"cli" \n\t"
"incl   %0 \n\t"
"movl   %8, %%esp \n\t"
"sti" \n\t"
"pushl  %%eax \n\t"
"call   *%7 \n\t"
"cli \n\t"
"decl   %0 \n\t"
"popl   %%esp # comment 2 \n\t"
"sti \n\t"
"popl   %%ebp "
: "=m" (processor0->intr_lv),
  "=&a" (a), "=b" (b), "=c" (c), "=d" (d), "=D" (D), "=S" (S)
: "m" (jmpaddr), "g" (processor0->bh_stack_top));
```

컴파일 하면,

```
#APP
pushl   %ebp # comment
movl    %esp, %eax
cli
incl    processor0+20
movl    processor0+76, %esp
sti
pushl   %eax
call    *-40(%ebp)
cli
decl    processor0+20
popl    %esp # comment 2
sti
popl    %ebp
#NO_APP
```

Gasm에서 커멘트는 #부터 그 줄의 끝까지이며 inline assembly에서도 같은 방법으로 쓸 수 있습니다.

2.2.2. Register 직접 지정하기

위의 예에서 %%eax, %%esp같은 것들을 볼 수 있는데 %%는 실제 output에서 % 하나를 출력합니다. 특정 레지스터를 써야할 때는 %n으로 정해줘도 되지만 input, output 지정에서 쓰고 싶

은 레지스터를 지정하고 어차피 그 레지스터가 할당될 것을 알고 있으므로 %%register를 쓰는 것이 더 읽고 쓰기 편합니다.

한 가지 주의할 점은 만약 input, output이 하나도 없는 경우에는 asms에 대한 인자치환이 전혀 일어나지 않고 %%도 %로 바뀌지 않습니다. 즉, input, output이 모두 없을 때는 %%register대신 %register라고 해야합니다.

2.2.3. Inline assembly안에서 함수 정의하기

함수를 assembly로 정의해야하는 경우는 주로 함수의 entry나 exit이 C의 convention과 달라서 C함수 안에서 inline assembly로 만들 수 없는 경우입니다. 물론 따로 어셈블리 화일을 만들어도 되지만 이런 함수가 몇 개 되지 않을 때는 번거롭기 때문에 inline assembly로 만드는 것이 더 편합니다.

또, assembly로 만든 함수에서 C 프로그램에서 쓰던 전역변수, 매크로등을 쓰게되는 경우가 있는데, 일반적인 경우 assembly 소스를 C preprocessor로 처리하고 링크하면 되지만 전역으로 선언된 structure를 사용하려면 문제가 됩니다. 스크립드등을 사용해서 member들의 offset을 포함한 헤더화일을 만든 후 C preprocessor를 사용하면 가능하긴 하지만 (실제로 freebsd 커널에선 이 방법을 사용합니다) 상당히 귀찮은 일이 되버립니다. 이런 경우에도 inline assembly로 함수를 정의해 주는 것이 훨씬 간편합니다.

Asms의 내용이 output에 그대로 출력되기 때문에 심볼을 정의하는 directive도 사용할 수 있습니다. 우선 예를 보겠습니다.

```
struct {
int a;
int b;
} mine = { 0, 0 };

int iasm_test_func(int arg);
static int iasm_test_func2(void);

static void
__iasm_function_dummy(void)
{
__asm__ __volatile__(
".globl iasm_test_func \n\t"
"iasm_test_func: \n\t"
"pushl 4(%%esp) \n\t"
"pushl %2 \n\t"
"pushl %1 \n\t"
"pushl %0 \n\t"
"call printf \n\t"
"addl $16, %%esp \n\t"
"ret "
:
: "i" ("hello world.. mine.a=%d mine.b=%d arg=%d\n"),
  "g" (mine.a), "g" (mine.b));
```

```
__asm__ __volatile__(
"iasm_test_func2: \n\t"
"pushl $32 \n\t"
"call iasm_test_func \n\t"
"addl $4, %esp ");
}

int
main(void)
{
mine.a = 123;
mine.b = 321;
iasm_test_func(16);
return iasm_test_func2();
}
```

우선 정의할 두 함수의 prototype을 볼 수 있습니다. iasm_test_func2는 static인 걸 주의해서 보시기 바랍니다.

Inline assembly는 함수밖에서는 쓰일 수 없으므로 __iasm_function_dummy라는 함수를 만들고 그 안의 두 inline assembly에서 각각 함수를 정의했습니다. 첫번째 함수는 iasm_test_func로 심볼 정의 바로위의 .globl directive로 링크시 외부에 보이는 심볼임을 알려주었고 두번째의 iasm_test_func2함수는 .globl이 없으므로 링크시 외부에서 보이지 않는 static 함수가 됩니다.

Gcc는 컴파일 할 때 iasm_test_func2가 inline assembly안에 정의되어 있는 걸 알지 못하므로 static함수가 정의되지 않았다고 경고를 하지만 무시하면 됩니다.

iasm_test_func는 정수 인자를 하나 받고, 전역 구조체인 mine의 내용과 받은 인자의 값을 printf를 사용해 출력하는 함수 입니다. Inline assembly를 보면 input으로 format 문자열, mine.a, mine.b가 사용되는데 "i"는 immediate integer operand로 format 문자열의 시작 주소가 그대로 operand가 됩니다. "g"는 immediate, 범용 레지스터 또는 memory operand를 뜻하는 것으로 compiler가 적절히 선택합니다.

iasm_test_func2는 iasm_test_func(32)를 호출하는 함수입니다.

위의 프로그램을 컴파일하면 아래와 같은 assembly가 됩니다. (gcc -fomit-frame-pointer -mpreferred-stack-boundary=2 -O2 -S iasm_function.c)

```
.file "iasm_function.c"
.version "01.01"
gcc2_compiled.:
.globl mine
.data
.align 4
.type  mine,@object
.size  mine,8
mine:
.long 0
.long 0
```

```
.section .rodata
.align 32
.LC0:
.string "hello world.. mine.a=%d mine.b=%d arg=%d\n"
.text
.align 4
.type  __iasm_function_dummy,@function
__iasm_function_dummy:
#APP
.globl iasm_test_func
iasm_test_func:
pushl 4(%esp)
pushl mine+4
pushl mine
pushl $.LC0
call printf
addl $16, %esp
ret
iasm_test_func2:
pushl $32
call iasm_test_func
addl $4, %esp
#NO_APP
ret
.Lfe1:
.size  __iasm_function_dummy,.Lfe1-__iasm_function_dummy
.align 4
.globl main
.type  main,@function
main:
movl $123,mine
movl $321,mine+4
pushl $16
call iasm_test_func
call iasm_test_func2
addl $4,%esp
ret
.Lfe2:
.size  main,.Lfe2-main
.ident "GCC: (GNU) 2.95.4 20010902 (Debian prerelease)"
```

__iasm_function_dummy안에 두 개의 inline assembly가 #APP, #NOAPP 안에서 iasm_test_func, iasm_test_func2를 정의하고 있고, iasm_test_func의 %1, %2는 direct addressing으로 처리된 것을 볼 수 있습니다.

2.3. Output/Input List

Output과 input은 "constraint" (variable)들의 쉼표로 구분된 리스트로 구성됩니다.

Constraint는 아래의 문자들과 몇가지 modifier들의 조합으로 허용되는 operand의 종류와 그 operand가 inline assembly에서 어떻게 사용되는지를 나타냅니다. 아래의 리스트는 완전하지 않습니다. Gcc 메뉴얼을 참조하세요.

2.3.1. Constraints

2.3.1.1. Basic constraints

’m’

Architecture가 일반적으로 지원하는 모든 addressing mode중 하나를 사용하는 memory operand.

’r’

범용 레지스터 operand.

’d’, ’a’, ’f’, ...

레지스터를 적접 지정하는 constraint. Architecture마다 다른 문자들을 정의합니다. ’d’, ’a’, ’f’는 68000/68020에서 쓰는 문자들입니다.

’i’

정수 immediate operand. Assemble할 때가 되어야 알 수 있는 symbol값(주소)들도 해당됩니다.

’n’

값을 정확히 알고 있는 정수. Symbol값들은 해당되지 않습니다.

’I’, ’J’, ’K’, ... ’P’

미리 정해진 범위안의 정수 immediate operand. 예를 들어 68000에서 ’I’는 1에서 8사이의 immediate operand를 뜻하며 shift operation의 shift cound로 쓰입니다.

’E’

Compling machine과 target machine의 floating 표현 방식이 같을 때 immediate floating operand를 허용합니다.

’F’

Immediate floating operand.

'G', 'H'

Machine에 따라 정의되는 특정한 범위내의 immediate floating point operand.

'g'

범용 레지스터, memory, immediate operand 중 무엇이라도 됩니다.

'X'

어떠한 operand라도 허용합니다.

'p'

유효한 주소가 허용됩니다. Load address, push address등의 instruction에 쓰입니다.

'Q', 'R', 'S', ... 'U'

Machine-dependent.

2.3.1.2. i386 specific

'q'

a, b, c, or d register

'A'

a, or d register (for 64-bit ints)

'f'

Floating point register

't'

First (top of stack) floating point register

'u'

Second floating point register

'a'

a register

'b'

b register

'c'

c register

'd'

d register

'D'

di register

'S'

si register

'I'

Constant in range 0 to 31 (for 32bit shifts)

'J'

Constant in range 0 to 63 (for 64bit shifts)

'K'

0xff

'L'

0xffff

'M'

0, 1, 2, or 3 (shifts for lea instruction)

'N'

Constant in range 0 to 255 (for out instruction)

'G'

Standard 80387 floating point constant

2.3.1.3. Modifiers

Constraint modifier들은 그 변수가 어떻게 사용되는 지를 compiler에게 알려줍니다. 아래의 리스트는 완전하지 않습니다. Gcc 메뉴얼을 참조하세요.

'='

변수의 값이 바뀜을 나타냅니다. Output들에 대해서는 이 modifier가 반드시 지정되어 있어야 합니다.

'&'

Early clobber. 다음 절에서 자세히 설명하겠습니다.

2.3.2. Early clobber

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
int a, sum;

a = atoi(argv[1]);

__asm__ __volatile__(
"movl %1, %0 \n\t"
"addl %2, %0 \n\t"
"addl %3, %0 \n\t"
"addl %4, %0 \n\t"
"addl %5, %0 "
: "=g" (sum)
: "g" (a), "g" (a+1), "g" (a+2), "g" (a+3), "g" (a+4));

printf("a=%d, sum=%d\n", a, sum);
return 0;
}
```

위의 프로그램을 컴파일하면

```
.file "early_clobber.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "a=%d, sum=%d\n"
.text
.align 4
.globl main
.type  main,@function
main:
pushl %esi
pushl %ebx
movl 16(%esp),%eax
pushl $0
pushl $10
pushl $0
pushl 4(%eax)
call __strtol_internal
movl %eax,%esi
addl $16,%esp
leal 1(%esi),%edx
```

```
leal 2(%esi),%ebx
leal 3(%esi),%ecx
leal 4(%esi),%eax
#APP
movl %esi, %edx
addl %edx, %edx
addl %ebx, %edx
addl %ecx, %edx
addl %eax, %edx
#NO_APP
pushl %edx
pushl %esi
pushl $.LC0
call printf
xorl %eax,%eax
addl $12,%esp
popl %ebx
popl %esi
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.4 20010902 (Debian prerelease)"
```

#APP와 #NOAPP 사이의 inline assembly에서 %0 (sum)에 %edx가 할당되었음을 알 수 있습니다. 그런데 #APP아래 두번째줄이 addl %edx, %edx로 %2 (a+1)도 %edx로 할당되었습니다. Gcc는 항상 모든 input 변수들이 다 사용된 후에 output 변수들이 쓰인다고 가정해서 input 변수와 output 변수를 같은 operand에 할당하기도 합니다. Input, output이 하나의 operand에 할당되고 위의 예처럼 input보다 output으로 먼저쓰이게 되면 틀린 결과가 나옵니다.

이런 경우에는 gcc에게 그 output 변수는 input의 값들이 모두 사용되기 전에 값이 바뀔 수 있다는 것을 알려주어야 합니다. 이것을 알려주기 위한 modifier가 early clobber modifier '&' 입니다. 위의 프로그램에서 output constraint "=g" (sum)을 "=&g" (sum)으로 바꾸고 다시 컴파일하면 다음과 같은 결과가 나옵니다.

```
.file "early_clobber.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "a=%d, sum=%d\n"
.text
.align 4
.globl main
.type  main,@function
main:
pushl %edi
pushl %esi
pushl %ebx
movl 20(%esp),%eax
```

```
pushl $0
pushl $10
pushl $0
pushl 4(%eax)
call __strtol_internal
movl %eax,%esi
addl $16,%esp
leal 1(%esi),%ebx
leal 2(%esi),%ecx
leal 3(%esi),%edx
leal 4(%esi),%eax
#APP
movl %esi, %edi
addl %ebx, %edi
addl %ecx, %edi
addl %edx, %edi
addl %eax, %edi
#NO_APP
movl %edi,%eax
pushl %eax
pushl %esi
pushl $.LC0
call printf
xorl %eax,%eax
addl $12,%esp
popl %ebx
popl %esi
popl %edi
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.4 20010902 (Debian prerelease)"
```

Output 변수는 %edi로 할당되었고 어떤 input도 겹치게 할당되지 않았음을 알 수 있습니다.

2.4. Clobber list

Input이나 output으로 사용되지 않지만 어떤 레지스터를 inline assembly에서 임시로 사용할 때 clobber list에 그 레지스터를 적습니다. Clobber list에 있는 레지스터는 input, output에 있는 레지스터와 겹칠 수 없습니다. 즉, clobber list에 지정된 레지스터는 input, output을 위한 레지스터 할당에서 빠지게 됩니다.

만약 어떤 레지스터가 input으로 쓰이고 그 값이 바뀌지만 output으로 쓰이진 않는다면 clobber list에 그 레지스터를 정해줄 수 없습니다. 이런 경우엔 dummy 변수를 하나 선언한 후 output 인자로도 지정해주어야 합니다.

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
int a, b, r;

a = atoi(argv[1]);
b = atoi(argv[2]);

__asm__ __volatile__(
"movl %1, %%eax \n\t"
"addl %2, %%eax \n\t"
"mull %%eax "
: "=&a" (r)
: "g" (a), "g" (b));

printf("a=%d, b=%d, r=%d\n", a, b, r);
return 0;
}
```

두 정수를 입력받아 합의 제곱을 구하는 프로그램입니다. 위의 프로그램을 컴파일하면 다음과 같은 결과가 나옵니다.

```
.file "clobber_list.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "a=%d, b=%d, r=%d\n"
.text
.align 4
.globl main
.type  main,@function
main:
pushl %esi
pushl %ebx
movl 16(%esp),%ebx
pushl $0
pushl $10
pushl $0
pushl 4(%ebx)
call __strtol_internal
movl %eax,%esi
addl $16,%esp
pushl $0
pushl $10
```

```
pushl $0
pushl 8(%ebx)
call __strtol_internal
movl %eax,%edx
addl $16,%esp
#APP
movl %esi, %eax
addl %edx, %eax
mull %eax
#NO_APP
pushl %eax
pushl %edx
pushl %esi
pushl $.LC0
call printf
xorl %eax,%eax
addl $16,%esp
popl %ebx
popl %esi
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.4 20010902 (Debian prerelease)"
```

b가 %edx로 할당되었음을 알 수 있습니다. b는 input이므로 값이 변하지 않은 것으로 생각해 inline assembly후에 printf를 부를 때도 %edx의 값을 그대로 사용하는 것을 볼 수 있습니다. 위의 프로그램을 실행해보겠습니다.

```
$ ./a.out 4 6
a=4, b=0, r=100
```

결과값은 맞지만 b의 값이 0으로 출력됩니다. 이는 mull instruction이 결과값을 %edx와 %eax에 걸쳐 저장하기 때문입니다. 위쪽 결과값인 %edx가 0이 되지만 컴파일러는 그 값이 변하지 않았다고 생각하기 때문에 b의 값이 엉뚱하게 된 것 입니다. 이처럼 input, output 어느 것으로도 쓰이지 않지만 그 값이 변하는 경우에는 clobber list에 그 레지스터의 이름을 적어주면 됩니다.

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
int a, b, r;

a = atoi(argv[1]);
b = atoi(argv[2]);
```

```
__asm__ __volatile__(
"movl %1, %%eax \n\t"
"addl %2, %%eax \n\t"
"mull %%eax "
: "=&a" (r)
: "g" (a), "g" (b)
: "edx" );

printf("a=%d, b=%d, r=%d\n", a, b, r);
return 0;
}
```

컴파일된 결과는 다음과 같습니다.

```
.file "clobber_list.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "a=%d, b=%d, r=%d\n"
.text
.align 4
.globl main
.type  main,@function
main:
pushl %esi
pushl %ebx
movl 16(%esp),%ebx
pushl $0
pushl $10
pushl $0
pushl 4(%ebx)
call __strtol_internal
movl %eax,%esi
addl $16,%esp
pushl $0
pushl $10
pushl $0
pushl 8(%ebx)
call __strtol_internal
movl %eax,%ecx
addl $16,%esp
#APP
movl %esi, %eax
addl %ecx, %eax
mull %eax
#NO_APP
```

```
pushl %eax
pushl %ecx
pushl %esi
pushl $.LC0
call printf
xorl %eax,%eax
addl $16,%esp
popl %ebx
popl %esi
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.4 20010902 (Debian prerelease)"
```

b의 값을 ecx로 할당한 것을 볼 수 있습니다.

물론 결과값도 제대로 나옵니다.

```
$ ./a.out 6 4
a=6, b=4, r=100
```

위의 예에서 볼 수 있듯이 clobber list에는 레지스터의 이름을 직접 적습니다. 쓰이는 이름들은 다음과 같습니다.

```
i386 specific

 ah, al, ax, eax
 bh, bl, bx, ebx
 ch, cl, cx, ecx
 dh, dl, dx, edx
 si, esi
 di, edi
```

(floating 레지스터들은 어떻게 지정하는 지 모르겠습니다. 아시는 분?)

Condition code의 값이 바뀜을 나타내는 cc가 있지만, ix86에선 필요하지 않습니다. 또, stack, frame pointer인 esp/sp, ebp/bp도 지정은 할 수 있지만 아무런 효력도 없습니다. esp, ebp의 값을 변경하는 경우엔 원래의 값으로 복원해야합니다.

3장. Applications

이 챕터에선 inline assembly를 사용하는 몇가지 예들을 보이겠습니다.

3.1. Atomic bit operations & spin lock

Multithread 프로그램에서 가장 큰 문제가 동시에 실행하는 여러개의 thread간의 동기화입니다. UP의 경우 기본적으로 어느 시점에 control을 다른 thread로 넘길 지를 알 수 있으므로 프로세서의 별다른 지원없이도 동기화가 가능하지만 MP에선 동시에 작동하는 여러개의 프로세서들을 동기화시키려면 특수한 기능을 제공해야합니다. 기본적으론 메모리의 어떤 값을 읽어서 조건을 확인하고 그 값에 어떤 변화를 주는 작업을 atomic하게 수행할 수 있어야합니다. 보통 test and set bit이나 exchange등을 atomic하게 수행하는 기능을 제공하게 되는데 ix86에선 두가지 모두 지원됩니다. (ix86에선 instruction에 LOCK prefix를 붙이면 대부분의 instruction들을 atomic하게 수행할 수 있습니다.)

이 절에선 atomic test and set bit을 이용해 동기화의 기본적인 구성요소인 spin lock을 만들어 보겠습니다. Pthread를 사용해 몇개의 thread들 사이에서 spin을 이용한 동기화를 해볼텐데, 각각의 thread들이 독립된 processor에서 작동하는 것이 아니라 time slice되어 작동하기 때문에 spin을 사용하는 것은 좋은 생각은 아닙니다. 예를 들기위한 것이라고 생각하시기 바랍니다.

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>

static __inline__ int
test_and_set_bit(int nr, volatile void *addr)
{
int oldbit;

__asm__ __volatile__(
"lock \n\t"
"btsl %2, %1 \n\t"
"sbbl %0, %0 "
: "=r" (oldbit), "=m" (*(unsigned *)addr)
: "Ir" (nr));
return oldbit;
}

typedef unsigned spin_t;

static __inline__ void
spin_lock(spin_t *spin)
{
if (test_and_set_bit(0, spin))
while (*(__volatile__ spin_t *)spin) ;
}
```

```
static __inline__ void
spin_unlock(spin_t *spin)
{
*(__volatile__ spin_t *)spin = 0;
}

spin_t spin = 0;

static __inline__ void
waste_time(void) {
time_t ltime;

ltime = time(NULL);
while (time(NULL) == ltime) ;
}

#define pp(fmt, arg...) printf("thread %d : "fmt, thread_no , ##arg)

static void *
thread_func(void *_arg)
{
int thread_no = (int)_arg;

while (1) {
pp("wasting time\n");
waste_time();
pp("entering critical section\n");
pp("now in critical section, wasting time\n");
waste_time();
pp("leaving critical section\n");
}

return NULL;
}

static void *
thread_sfunc(void *_arg)
{
int thread_no = (int)_arg;
time_t ltime;

while (1) {
pp("wasting time\n");
waste_time();
pp("entering critical section\n");
spin_lock(&spin);
pp("now in critical section, wasting time\n");
waste_time();
```

```
pp("leaving critical section\n");
spin_unlock(&spin);
}

return NULL;
}

int
main(void)
{
pthread_t thread0, thread1, thread2, thread3;

pthread_create(&thread0, NULL, thread_func, (void *)0);
pthread_create(&thread1, NULL, thread_func, (void *)1);
pthread_create(&thread2, NULL, thread_sfunc, (void *)2);
pthread_create(&thread3, NULL, thread_sfunc, (void *)3);

pthread_join(thread0, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);

return 0;
}
```

4개의 thread가 시간보내기, critical section들어가기, 시간보내기, critical section나오기의 과정을 반복합니다. 0번과 1번 thread는 아무런 동기화도 하지 않고 2, 3은 spin을 이용해 critical section을 보호합니다.

우선 test_and_set_bit 함수를 살펴보도록 하겠습니다.

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

Atomic하게 *addr의 nr번째 bit을 test and set합니다. Test의 결과는 carry flag에 기록이됩니다.

```
sbbl %0, %0
```

위의 btsl에서 CF기록된 test결과를 %0에 저장합니다. sbbl은 subtraction with carry로 위처럼 같은 두 수로 실행하면 carry flag과 같은 논리값이 operand에 저장됩니다.

Output, input 지정에선 nr의 constrait가 "Ir"인 점을 제외하면 별다른 점은 없습니다. nr이 한 word안에서의 bit offset이므로 immediate일 때 0~31사이로 제한을 둔 것이고, 레지스터 operand일 때는 컴파일타임에 확인할 수 없기 때문에 그냥 'r'을 constraint로 준 것입니다.

test_and_set_bit이 있으면 spin의 구현은 간단한데요. test_and_set_bit의 결과값이 0일 때까지 반복적으로 수행하면 됩니다. 위의 프로그램에서 기다리는 부분을 while로 처리한 것은 lock; btsl보다 보통의 memory read가 bus에 부담을 덜 주기 때문입니다. while의 조건에서

__volatile__로 캐스팅후 사용하는 건 gcc가 레지스터에 spin의 값을 읽은 후 그 값을 계속 test하는 것을 막기 위해서 입니다. Gcc에게 이 값은 현재 프로그램의 진행과 관계없이 바뀔 수 있다는 것을 알려주는 것입니다.

Unlock은 그냥 spin의 값을 0으로 하면 됩니다. 역시 memory에 직접쓰도록 하기위해 __volatile__로 캐스팅후 사용하는 것을 볼 수 있습니다. 그냥 spin을 __volatile__로 지정해주어도 됩니다.

Inline assembly는 단순하므로 컴파일된 assembly는 생략하고 실행 결과를 보겠습니다.

```
thread 0 : wasting time
thread 1 : wasting time
thread 2 : wasting time
thread 3 : wasting time
thread 0 : entering critical section
thread 0 : now in critical section, wasting time
thread 3 : entering critical section
thread 3 : now in critical section, wasting time
thread 2 : entering critical section
thread 1 : entering critical section
thread 1 : now in critical section, wasting time
thread 1 : leaving critical section
thread 1 : wasting time
thread 0 : leaving critical section
thread 0 : wasting time
thread 3 : leaving critical section
thread 3 : wasting time
thread 2 : now in critical section, wasting time
thread 2 : leaving critical section
thread 2 : wasting time
thread 1 : entering critical section
thread 1 : now in critical section, wasting time
thread 0 : entering critical section
thread 0 : now in critical section, wasting time
thread 3 : entering critical section
thread 3 : now in critical section, wasting time
thread 2 : entering critical section
thread 1 : leaving critical section
thread 1 : wasting time
thread 0 : leaving critical section
thread 0 : wasting time
thread 3 : leaving critical section
thread 3 : wasting time
thread 2 : now in critical section, wasting time
```

0과 1의 critical section은 겹치지만 2와 3은 겹치지 않음을 알 수 있습니다.

3.2. Function call with stack switch

Stack 사용량을 어느 정도 이하로 보장하기 위해서 어떤 기능들은 stack을 바꾸어 가면 실행하는 경우가 있습니다. 예를 들어, OS에서 interrupt handler와 그에 따라 실행되는 bottom half handler들의 경우 interrupt 발생시의 kernel stack에서 실행된다면 interrupt nesting 등을 생각할 때 모든 kernel thread의 stack 크기가 꽤 커져야 하는데다가 필요한 크기를 정확히 알기도 힘듭니다.

아래의 프로그램에서 call_with_stack_switch는 funcaddr로 주어진 함수를 arg를 인자로 해서 altstack에서 수행하고 그 결과값을 돌려줍니다.

```
#include <stdio.h>

#define fetch_esp(_esp) \
__asm__ __volatile__("movl %%esp, %0" : "=g" (*_esp))

static __inline__ int
call_with_stack_switch(void *funcaddr, unsigned int arg, void *altstack)
{
int a, b, c, d, D, S;

__asm__ __volatile__(
"pushl %%ebp \n\t"
"movl %%esp, %%eax \n\t"
"movl %8, %%esp \n\t"
"pushl %%eax \n\t"
"pushl %7 \n\t"
"call *%6 \n\t"
"addl $4, %%esp \n\t"
"popl %%esp \n\t"
"popl %%ebp "
: "=&a" (a), "=b" (b), "=c" (c), "=d" (d), "=D" (D), "=S" (S)
: "r" (funcaddr), "ri" (arg), "ri" (altstack));

return a;
}

static int
say_hello(unsigned int arg)
{
unsigned esp;

fetch_esp(&esp);

printf("say_hello : hello world... esp=%08x, arg=%d\n", esp, arg);
arg *= arg;
printf("say_hello : returning %d\n", arg);
return arg;
}
```

```
static char _altstack[8192];
static void *altstack = _altstack + sizeof(_altstack);

int
main(void)
{
unsigned esp;
int rv, arg = 1096;

fetch_esp(&esp);
printf("main      : current esp=%08x, altstack=%08p-%08p\n",
       esp, _altstack, altstack);
printf("main      : calling say_hello w/ stack switch (arg=%d)\n",
       arg);

rv = call_with_stack_switch(say_hello, arg, altstack);

fetch_esp(&esp);

printf("main      : esp=%08x, arg=%d, rv=%d\n", esp, arg, rv);
return 0;
}
```

call_with_stack_switch에서 6개의 변수가 선언되어 있는데 이 변수들은 모두 레지스터들의 output으로 쓰입니다. a:eax, b:ebx, c:ecx, d:edx, D:edi, S:edi로 대응이 됩니다. a외에는 output이라고 정의된 후 쓰이지 않는데, 단지 그 레지스터들의 값이 바뀐다는 것을 컴파일러에게 알려주는 역활만을 하게됩니다. Clobber list와 거의 같은 기능이라고 할 수 있지만 clobber로 지정된 레지스터는 input, output 어느것으로도 쓰일 수 없고 위의 inline assembly에 있는 세개의 input 변수들이 그 레지스터로 할당될 수 없게됩니다. 즉, dummy 변수를 써서 output으로 정해주게 되면 'input으로 할당될 수 있지만 결과적으로 값은 변한다'라는 뜻입니다.

각 라인을 살펴보도록 하겠습니다.

```
1: pushl %%ebp
```

inline assembly의 앞과 끝에서 ebp를 저장하고 복구하는데 ebp는 ix86에서 frame pointer로 쓰입니다. 만약 -fomit-frame-pointer 옵션을 주지않고 컴파일하면 frame pointer의 관리가 함수 entry/exit에서 되어 신경 쓸 필요가 없지만 frame pointer를 생략하게 되면 컴파일러가 ebp를 다른 용도로 쓰게됩니다. 하지만 gcc에게 ebp가 변함을 알려줄 방법이 없기때문에 컴파일러가 모르는 체로 ebp의 값이 바뀌어 버릴 수가 있습니다. 따라서 다른 레지스터들과 달리 따로 저장/복구 해 줄 필요가 있습니다.

ebp와 gcc에 대해 조금 더 설명하겠습니다. ebp는 완전한 범용 레지스터는 아니지만 대부분의 주소계산에 사용될 수 있고 값들을 잠시 저장하는 장소로도 사용될 수 있습니다. gcc는 frame pointer로 쓰지 않을 때 ebp를 이런 용도로 사용하지만 input/output에서 직접 ebp를 지정해줄 수 있는 방법이 없고, clobber list에서 지정을 할 수 있지만 무시되기때문에 inline assembly에서 ebp의 값을 변화시킬 때는 반드시 저장/복구 해주어야 합니다. Gcc의 버그라고도 할 수 있습니다.

```
2: movl %%esp, %%eax
```

현재 esp값을 eax에 저장합니다. altstack으로 바꾸어 함수를 실행하고 원래의 stack으로 돌아와야하기 때문에 원래 stack pointer를 기억하고 있어야 합니다. 이를 altstack으로 바꾸고 제일 처음에 push하기 위해 eax에 저장해 두는 것입니다.

```
3: movl %8, %%esp
```

%8은 altstack입니다. altstack으로 stack을 바꿉니다.

```
4: pushl %%eax
```

원래의 stack pointer를 stack에 저장합니다.

```
5: pushl %7
```

%7은 arg입니다. 함수 호출을 위해 arg를 새로 바뀐 stack에 push합니다.

```
6: call *%6
```

funcaddr을 호출합니다. *는 indirect call임을 나타냅니다. Input에서 더 자세히 설명하겠습니다.

```
7: addl $4, %%esp
```

funcaddr의 실행이 끝났으므로 arg를 없앱니다.

```
8: popl %%esp
```

원래의 stack으로 돌아옵니다.

```
9: popl %%ebp
```

1에서 저장해둔 ebp를 복구합니다.

Output에서 a가 early clobber로 지정된 것 이외에는 특별한 점이 없습니다. eax를 제외한 레지스터들은 funcaddr의 함수가 실행하면서 즉, 모든 input이 다 쓰인 후에 바뀔 수 있기 때문에 early clobber로 지정할 필요가 없고 따라서 input에 할당될 수 있습니다.

Input에서 funcaddr은 범용 레지스터, arg와 altstack은 범용 레지스터 또는 immediate constraint를 가지고 있습니다. Memory operand는 esp에 대한 offset addressing으로 표현될 수 있고, esp를 바꾼 후에 input들을 사용하기 때문에 memory operand는 혀용할 수 없으므로 레지스터나 immediate을 사용해야 하는데 ix86의 call instruction은 immediate operand로는 relative call밖에 지원하지 않기 때문에 indirect call을 해야하고 따라서 'r' constraint를 써야합니다. 나머지 둘은 immediate이어도 관계가 없기 때문에 'ri' constraint를 가지고 있습니다.

arg와 altstack이 call_with_stack_switch의 인자이기 때문에 immediate이 의미없다고 생각할 수도 있지만, __inline__으로 선언되어 있기 때문에 인자가 compile time에 결정될 수 있으면 immediate이 됩니다. 아래의 컴파일한 assembly를 보면 알 수 있습니다.

```
.file "call_with_stack_switch.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.align 32
.LC0:
.string "say_hello : hello world... esp=%08x, arg=%d\n"
.LC1:
.string "say_hello : returning %d\n"
.text
.align 4
.type  say_hello,@function
say_hello:
subl $4,%esp
pushl %ebx
movl 12(%esp),%ebx
#APP
movl %esp, 4(%esp)
#NO_APP
pushl %ebx
pushl 8(%esp)
pushl $.LC0
call printf
imull %ebx,%ebx
pushl %ebx
pushl $.LC1
call printf
movl %ebx,%eax
addl $20,%esp
popl %ebx
popl %ecx
ret
.Lfe1:
.size  say_hello,.Lfe1-say_hello
.data
.align 4
.type  altstack,@object
.size  altstack,4
```

```
altstack:
.long _altstack+8192
.section .rodata
.align 32
.LC2:
.string "main      : current esp=%08x, altstack=%08p-%08p\n"
.align 32
.LC3:
.string "main      : calling say_hello w/ stack switch (arg=%d)\n"
.align 32
.LC4:
.string "main      : esp=%08x, arg=%d, rv=%d\n"
.text
.align 4
.globl main
.type  main,@function
main:
subl $4,%esp
pushl %ebp
pushl %edi
pushl %esi
pushl %ebx
#APP
movl %esp, 16(%esp)
#NO_APP
pushl altstack
pushl $_altstack
pushl 24(%esp)
pushl $.LC2
call printf
pushl $1096
pushl $.LC3
call printf
movl $say_hello,%edx
movl altstack,%eax
addl $24,%esp
movl %eax,%ebp
#APP
pushl %ebp
movl %esp, %eax
movl %ebp, %esp
pushl %eax
pushl $1096
call *%edx
addl $4, %esp
popl %esp
popl %ebp
movl %esp, 16(%esp)
#NO_APP
```

```
pushl %eax
pushl $1096
pushl 24(%esp)
pushl $.LC4
call printf
xorl %eax,%eax
addl $16,%esp
popl %ebx
popl %esi
popl %edi
popl %ebp
popl %ecx
ret
.Lfe2:
.size  main,.Lfe2-main
.local _altstack
.comm _altstack,8192,32
.ident "GCC: (GNU) 2.95.4 20010902 (Debian prerelease)"
```

call_with_stack_switch가 main안에 inlining 되었고, altstack이 %ebp로, arg는 immediate operand로, funcaddr이 %edx로 할당된 것을 볼 수 있습니다. 또, Dummy 변수들은 모두 사라졌고, return 값인 a도 %eax에 있는 그대로 사용되고 있습니다.

위의 프로그램을 실행하면 다음과 같은 결과가 나옵니다.

```
% ./call_with_stack_switch
main      : current esp=bffffc3c, altstack=0x80497c0-0x804b7c0
main      : calling say_hello w/ stack switch (arg=1096)
say_hello : hello world... esp=0804b7ac, arg=1096
say_hello : returning 1201216
main      : esp=bffffc3c, arg=1096, rv=1201216
```

Inline assembly를 사용할 때는 레지스터 할당이 정확히 어떻게 되는지 프로그램을 쓰면서는 알 수 없고, 특히 early clobber 옵션은 잊기가 쉽고 잘못되었을 때 찾기가 상당히 힘들기 때문에 제대로 작동하는 것 같더라도 -S 옵션을 주어 원하는 코드가 생성되었는지를 확인해보는 것이 좋습니다.