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 위해I선 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 아래처럼 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 싶은 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 C 함수 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 헤더 화일을 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.
