Chapter 38: Inline assembly

Section 38.1: gcc Inline assembly in macros

We can put assembly instructions inside a macro and use the macro like you would call a function.

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// some definition and assignment
unsigned char sbox[size][size];
unsigned char sbox[size][size];

///Using
mov(state[0][1], sbox[si][sj]);
```

Using inline assembly instructions embedded in C code can improve the run time of a program. This is very helpful in time critical situations like cryptographic algorithms such as AES. For example, for a simple shift operation that is needed in the AES algorithm, we can substitute a direct `Rotate Right` assembly instruction with C shift operator `>>`.

In an implementation of 'AES256', in 'AddRoundKey()' function we have some statements like this:

```
unsigned int w;          // 32-bit
unsigned char subkey[4]; // 8-bit, 4*8 = 32

subkey[0] = w >> 24;     // hold 8 bit, MSB, leftmost group of 8-bits
subkey[1] = w >> 16;     // hold 8 bit, second group of 8-bit from left
subkey[2] = w >> 8;      // hold 8 bit, second group of 8-bit from right
subkey[3] = w;           // hold 8 bit, LSB, rightmost group of 8-bits

/// subkey <- w
```

They simply assign the bit value of `w` to `subkey` array.

We can change three shift + assign and one assign C expression with only one assembly `Rotate Right` operation.

```
__asm__ ("l.ror  %0,%1,%2" : "=r" (* (unsigned int *) subkey)  : "r" (w), "r" (0x10));
```

The final result is exactly same.

Section 38.2: gcc Basic asm support

Basic assembly support with gcc has the following syntax:

```
asm [ volatile ] ( AssemblerInstructions )
```

where `AssemblerInstructions` is the direct assembly code for the given processor. The volatile keyword is optional and has no effect as gcc does not optimize code within a basic asm statement. `AssemblerInstructions` can contain multiple assembly instructions. A basic asm statement is used if you have an asm routine that must exist outside of a C function. The following example is from the GCC manual:

```
/* Note that this code will not compile with -masm=intel */
#define DebugBreak() asm("int $3")
```

In this example, you could then use `DebugBreak()` in other places in your code and it will execute the assembly instruction `int $3`. Note that even though gcc will not modify any code in a basic asm statement, the optimizer may still move consecutive asm statements around. If you have multiple assembly instructions that must occur in a specific order, include them in one asm statement.

Section 38.3: gcc Extended asm support

Extended asm support in gcc has the following syntax:

```
asm [volatile] ( AssemblerTemplate
                 : OutputOperands
                 [ : InputOperands
                 [ : Clobbers ] ])

 asm [volatile] goto ( AssemblerTemplate
                       :
                       : InputOperands
                       : Clobbers
                       : GotoLabels)
```

where `AssemblerTemplate` is the template for the assembler instruction, `OutputOperands` are any C variables that can be modified by the assembly code, `InputOperands` are any C variables used as input parameters, `Clobbers` are a list or registers that are modified by the assembly code, and `GotoLabels` are any goto statement labels that may be used in the assembly code.

The extended format is used within C functions and is the more typical usage of inline assembly. Below is an example from the Linux kernel for byte swapping 16-bit and 32-bit numbers for an ARM processor:

```
/* From arch/arm/include/asm/swab.h in Linux kernel version 4.6.4 */
#if __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

Each asm section uses the variable `x` as its input and output parameter. The C function then returns the manipulated result.

With the extended asm format, gcc may optimize the assembly instructions in an asm block following the same rules it uses for optimizing C code. If you want your asm section to remain untouched, use the `volatile` keyword for the asm section.