Freescale Semiconductor

Extended ASM - Assembler Instructions with C Expression Operands

Code Warrior

By: Abigail Inzunza / Jorge Gonzalez

freescale semiconductor

About this document

It describes how to work on a project in which we have to combine some assembly instructions with our C code.

The C Expression Operands needed for the use of assembly instructions on some basic cases will be briefly explained here.

Single assembly instruction

In CW, if we want to add an assembly instruction to the code in our file.c it is necessary to write it like this:

asm(“Inst Operand1, Operand2”);

Example

```
asm(“mov r0, #0x0A”);
```

In this case, the instruction copies the value of the operand2 (0x0A), to the operand1 (register r0).

Multiple assembly instruction

You can also write multiple assembly instructions in a sinlge asm statement:

asm(
“Inst Operand1, Operand2\n\t”
“Inst Operand3, Operand4\n\t”
“Inst Operand5, Operand6”
);

Example

```
asm(
"mov r2, #0x00\n\t"
"mov r3, #0x01\n\t"
"mov r4, #0x02"
);
```

With this group of instructions we are copying the values 0x00, 0x01, 0x02 to the registers r2, r3 and r4, respectively.

Creating assembly functions

It is important to know that every assembly function that we want to use on a CW Project must be described on a **.s** file. On this file we have to define the name of our function and then start describing each instruction.

Also, it is important to make the name of the function global, so that it can be used from a different file.

Example

We create a file called add.s in which we are going to describe the function _add using assembly code. Our file will look like the following:

```
.global _add

_add:

add r0, r0, r1
add r2, r2, #0x01
bx lr
```

The last instruction, 'BX' (Branch and Exchange), generates a branch to the return address from the subroutine and the instruction set specified by 'LR' (Link Register).

Moreover, it is necessary to define this function as extern on the file that we want to call it. So, if we want to use our function _add on our main.c we have to add the following line of code:

```
extern void _add();
```

Calling assembly function using assembly code

In order to call an assembly function using assembly code you have to use the instruction 'BL' to generate a Branch with Link that will call the subroutine that follows. Before the name of the function you have to add an underscore ('_') just like this:

asm("bl _fn");

Example

```
asm("bl _add");
```

This line of code will call our assembly function _add.

Calling assembly function using C code

On the other hand, if we want to call an assembly function using C code we just need to write on a line the name of the function and after it '()'.

_fn();

Example

```
_add();
```

This is how we would call our function, _add, using C code.

Using registers for storing the values of variables

The combination of C and assembly code also let us store the value of a variable in an specific register that we want to use, defining our variables like this:

register VarType VarName asm ("r**x**");

Note: **'x'** is the number of the desired register.

Example

```
register int ra asm ("r4");
register int rb asm ("r5");
```

With these lines we are creating a pseudonym for the registers r4 and r5, so if the value of the variables ra or rb are modified we will see the change on these registers.

Assembly language and C operands

If we need to use assembly instructions involving C operands, there is a specific format that we need to follow in order to write an inline assembler statement:

asm(code **:** output operand list **:** input operand list **:** clobber list);

On the code section, after writing the instruction it is necessary to make a reference to the operands that we would use. We refer to the C operands asigning numbers to it, the first operand must be numbered 0 and continue in increasing order (adding a % before every number).

We can also assign labels to our operands instead of using the reference number, this labels would be written as %[label].

Operand constraints

Constraints are used to indicate the operand types accepted by the assembly instructions.

Commonly used constraints:

"r"	General register (r0 - r15).
"m"	Memory operand.
"i"	Immediate integer.
"g"	Register, memory or immediate integer.

The constraints may include a modifier which indicates the access nature of the operand. The usual modifiers are:

=	Write only operand.
+	Read/Write operand.
&	Output only register.

Examples

```
asm ("mov %[test], %1" : [test] "=r" (test) : "g" (rb));
```

On this instruction we use a label for the variable 'test', our output operand, and we numbered the input operand 'rb', with its corresponding %1. On the other hand, we don't have any element on the clobber list.

```
asm ("add %0, %1, %2"  : "=r"(test) : "r"(test), "r"(0x01));
```

In this case, we have an output operand and two more input operands, so we use %0, %1, %2, respectively, for them.

Clobbered registers

Clobber list includes registers that even if not part of the input/output operands, might be modified by the assembly instructions.

Example

```
asm ("mov r4, %1\n\t"
     "mov %0, r4\n\t"
     :"=r"(a)          // output
     :"r"(b)           // input
     :"r4"             // clobbered register
     );
```

In this case the register r4 is not listed as output or input, but it is still used within the assembly code.

For more information about inline assembly and extended ASM please visit gcc.gnu.org