Inline Assembly

kaneton people

January 14, 2012

Outline

Outline

Description

Inline assembly is used to insert assembly code into C source code.

Inline assembly reduces the function-call overhead while reducing the number of little assembly source files.

Moreover, the inline assembly source code can evolve into the C source one, using C variables, C constants etc.

Inline assembly is commonly used to improve performances. Neverthless system programmers appreciate the inline assembly because some processor instructions are only available in assembly.

gcc uses the keyword **asm** to introduce inline assembly.

Outline

Overview

gcc inline assembly uses AT&T assembly syntax which is different from the Intel one.

We will study the major differences between the AT&T syntax and the Intel one.

Source-Destination Ordering

- **Intel**: *opcode dst src*
- **AT&T**: *opcode src dst*

Register Naming

Register names are prefixed by **%** for example *%eax*.

With C input arguments, the registers are prefixed by **%%** like *%%eax*.

Immediate Operand

Immediate operands are prefixed by **$**.

In Intel syntax, hexadecimal numbers are suffixed by **h**. Instead the AT&T syntax uses **0x** as prefix.

Operand Size

Intel syntax uses keywords **byte**, **word** and **dword** to specify the operand size while AT&T syntax uses a suffix to the opcode: **b**, **w**, **l**.

- **Intel**: *mov al, byte ptr foo*
- **AT&T**: *movb foo, %al*

Memory Operands

The Intel syntax uses **[** and **]** to specify the base register while AT&T uses **(** and **)**.

Moreover, the syntaxes for the shifts are also different:

- **Intel**: *[basepointer + indexpointer * indexscale + immed32]*
- **AT&T**: *immed32(basepointer,indexpointer,indexscale)*

You could think of the formula to calculate the address as:

immed32 + basepointer + indexpointer * indexscale

Example

The equivalent C source code:

```
*(p + 1)
```

where **p** is a *char**.

The AT&T assembly code:

```
1(%eax)
```

where **%eax** has the value of **p**.

The Intel assembly code:

```
[eax + 1]
```

Major Differencies

Intel Code	**AT&T Code**
mov eax, 1	movl $1, %eax
mov ebx, 0ffh	movl $0xff, %ebx
int 80h	int $0x80
mov ebx, eax	movl %eax, %ebx
mov eax, [ecx]	movl (%ecx), %eax
mov eax, [ebx + 3]	movl 3(%ebx), %eax
mov eax, [ebx + 20h]	movl 0x20(%ebx), %eax
add eax, [ebx + ecx * 2h]	addl (%ebx, %ecx, 0x2), %eax
lea eax, [ebx + ecx]	leal (%ebx, %ecx), %eax
sub eax, [ebx + ecx * 4h - 20h]	subl -0x20(%ebx, %ecx, 0x4), %eax

Outline

Overview

The format of the gcc inline assembly is the following:

```
asm(‘‘assembly code’’);
```

Example:

```
asm(‘‘movl %eax, %ecx’’);
```

You can also use the keyword **__asm__** in the case of a name conflict with your previous source code.

Registers

You can use multi-lines source code thinking to insert the sequence `\n\t` at the end of each line.

Example:

```
asm(‘‘movl %cr0, %eax\n\t’’
    ‘‘orl %eax, $1\n\t’’
    ‘‘movl %eax, %cr0’’);
```

You can notice in this example that we overwrite the register

* gcc does not know anything about the inline assembly. Overwritting the **eax** register could result with an error because gcc maybe used this register to hold a variable or anything else.

Outline

Overview

In the basic use, we had only instructions. In the extended one we can specify more options like the input C operands, the output C operands and the clobbered registers i.e the modified registers.

The syntax of the inline assembly becomes:

```
asm(assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
   );
```

Rules

Be careful with this syntax:

- the outputs are located before the inputs
- think to use colons to separate sections: inputs, outputs etc.
- do not specify an empty clobbered registers. In fact the last list provided must always contain at least one element

Rules

Be careful with this syntax:

- the outputs are located before the inputs
- think to use colons to separate sections: inputs, outputs etc.
- do not specify an empty clobbered registers. In fact the last list provided must always contain at least one element

Rules

Be careful with this syntax:

- the outputs are located before the inputs
- think to use colons to separate sections: inputs, outputs etc.
- do not specify an empty clobbered registers. In fact the last list provided must always contain at least one element

Example

Let's study some examples:

The following one is incorrect:

```
asm(``...''
    :
    : ``...''
    :
   );
```

Neverthless, this one is correct:

```
asm(``...''
    :
    : ``...''
   );
```

Problem

Come back to our example which is a famous one in the low-level system programming field:

```
asm(‘‘movl %cr0, %eax\n\t’’
    ‘‘orl %eax, $1\n\t’’
    ‘‘movl %eax, %cr0’’);
```

You can notice that the **eax** register is overwritten, so we have to tell gcc to restore the contents of the **eax** register, here come the clobbered register list.

Solution

The clobber list contains the name of the registers modified by the inline assembly source code.

Let's see how to use it to resolve our problem:

```
asm(``movl %cr0, %eax\n\t''
    ``orl %eax, $1\n\t''
    ``movl %eax, %cr0''
    :
    :
    : ``%eax''
   );
```

Example

Now, another example.

We have two C variables **A** and **B** and we want to move the contents of **A** into **B** but using inline assembly for an unknown reason:

```
asm(``movl %1, %%eax\n\t''
    ``movl %%eax, %0''
    : ``=r'' (b)
    : ``r'' (a)
    : ``%eax''
   );
```

Explanations

1. the operands %0-9 specify the input and outputs. Note that the first value is used for the first output. If no output is present, then %0 will specify the first input.
2. the constraint "**r**" is used to precise the nature of the operand. Be careful with it. In this case, it tells gcc to use the operand as a register. The constraint "=" is used for the output operands specifying a write-only operand.
3. the double prefix **%%** is used when using input and/or output operands to avoid conflicts with registers.
4. finally, the clobbered register **eax** is indicated to tell gcc to restore this register's contents.

Now we will look at each field in details.

Note that we do not have to tell gcc the memory was modified because we used the constraints "**r**" so the registers were used.

Explanations

1. the operands %0-9 specify the input and outputs. Note that the first value is used for the first output. If no output is present, then %0 will specify the first input.
2. the constraint "**r**" is used to precise the nature of the operand. Be careful with it. In this case, it tells gcc to use the operand as a register. The constraint "**=**" is used for the output operands specifying a write-only operand.
3. the double prefix **%%** is used when using input and/or output operands to avoid conflicts with registers.
4. finally, the clobbered register **eax** is indicated to tell gcc to restore this register's contents.

Now we will look at each field in details.

Note that we do not have to tell gcc the memory was modified because we used the constraints "**r**" so the registers were used.

Explanations

1. the operands %0-9 specify the input and outputs. Note that the first value is used for the first output. If no output is present, then %0 will specify the first input.
2. the constraint **"r"** is used to precise the nature of the operand. Be careful with it. In this case, it tells gcc to use the operand as a register. The constraint **"="** is used for the output operands specifying a write-only operand.
3. the double prefix **%%** is used when using input and/or output operands to avoid conflicts with registers.
4. finally, the clobbered register **eax** is indicated to tell gcc to restore this register's contents.

Now we will look at each field in details.

Note that we do not have to tell gcc the memory was modified because we used the constraints **"r"** so the registers were used.

Explanations

1. the operands %0-9 specify the input and outputs. Note that the first value is used for the first output. If no output is present, then %0 will specify the first input.
2. the constraint **"r"** is used to precise the nature of the operand. Be careful with it. In this case, it tells gcc to use the operand as a register. The constraint **"="** is used for the output operands specifying a write-only operand.
3. the double prefix **%%** is used when using input and/or output operands to avoid conflicts with registers.
4. finally, the clobbered register **eax** is indicated to tell gcc to restore this register's contents.

Now we will look at each field in details.

Note that we do not have to tell gcc the memory was modified because we used the constraints **"r"** so the registers were used.

Outline

Overview

There are only two rules for the template:

1. finish each line with the sequence `\n\t`
2. use **%0, ...** to use C operands.

Outline

Overview

The syntax used for the operands is:

```
''constraint'' (operand)
```

The constraints are primarily used to decide the addressing mode for operands: register, memory, etc.

The operands are separated by commas inside a list.

Operands are numbered from 0 to n-1.

Output operands must be lvalues but this is obvious. Input operands are not restricted.

gcc will assume that the values in the output operands are dead and can be overwritten.

Read-Write Outputs

gcc inline assembly also allows read-write outputs operands.

```
asm(``orl %0, %0''
    : ``=r'' (x)
    : ``0'' (x)
   );
```

In this example, we specify gcc to use the C variable **x** as the output operand.

For the input operand, we use the constraint **"0"** to tell gcc to use the same register for the input operand as the operand numbered **0** do.

So in this case, the input and output operand will be stored in the same register which is not precised, so gcc will decide.

So, in this example we have a read-write operand **x**.

Specify the Registers

Note that we can also explicitly tell gcc which register to use for storing an operand:

```
asm(``orl %0, %0''
    : ``=c'' (x)
    : ``c'' (x)
   );
```

In this case, we tell gcc to use the **ecx** register.

Notices

You can notice that in the two previous examples, we put nothing into the clobbered list.

In the first example, we told gcc to decide the register to use so gcc knows which register to restore.

In the second one, we specify gcc to use the **ecx** register. Once again, gcc perfectly knows which register to restore.

Outline

Overview

Some instructions clobber the hardware registers.

For this reason, the inline assembly user has to indicate gcc the names of the registers to restore.

Note that gcc already knows the clobbered registers used via the constraints of the input and output operands.

Be careful with instructions which implicitly use some registers.

Example

Let's see an example:

```
asm(``movl %0, %%ebx\n\t''
    ``movl %1, %%ecx\n\t''
    ``call foo''
    :
    : ``g'' (from), ``g'' (to)
    : ``%ebx'', ``%ecx''
   );
```

Condition Code and Memory

The user can also specify two other elements in the clobber list:

1. **"cc"** meaning that the condition code was modified i.e for example the bits of the **eflags** register was modified: *jnz*, *je* etc.
2. **"memory"** when a variable is set as output operand. This is very useful because gcc maybe holds this variable in a register. This constraint tells gcc to update its registers if needed. This constraint is equivalent to set all the registers in the clobber list.

Outline

Overview

First, the keyword **volatile** is used between the **asm** one and the first parenthesis.

You can also use the keyword __**volatile**__ to avoid conflicts.

The keyword **volatile** is useful in one precise case:

When the user wants to tell gcc to avoid optimizations. For example if your assembly code is located inside a loop, gcc will certainly move your assembly code out of the loop. Using volatile avoid this optimization.

Be careful, if your assembly code does not have any side effect, it is better not to put the **volatile** keyword to allow gcc optimizations.

Outline

Overview

You probably noticed within this course that the constraints have got a lot to do with inline assembly.

Indeed, with them it is possible to specify the location of operands: memory, registers etc.

We will now list the different types of operands.

Register Operand Constraint: "r"

When an operand is specified with a register operand constraint, gcc stores the operand in a GPR (General Purpose Register).

The user can also explicitly tell gcc the register to use. For this, he has to use specific constraints listed below:

Constraint	**Register(s)**
r	any General Purpose Register
a	%eax, %ax, %al
b	%ebx, %bx, %bl
c	%ecx, %cx, %cl
d	%edx, %dx, %dl
S	%esi, %si
D	%edi, %di

Memory Operand Constraint: "m"

When the operands are in the memory, any operation performed on them will occur directly in the memory location, as opposed to register constraints.

Register constraints are usually used with instructions which only accept registers as operands. Another case is to speed up the process, for example if a variable is needed for three consecutive instructions.

The memory constraint can be used most effeciently in cases where a C variable needs to be updated inside inline assembly.

For example:

```
asm(‘‘sidt %0\n’’
    : ’’=m’’(idtr)
   );
```

Matching Digit Constraint

As seen before, in some cases the programmer wants to use an operand both in input and in output.

```
asm(‘‘incl %0’’
    : ’’=a’’ (var)
    : ’’0’’ (var)
   );
```

In this example the register **eax** is used as output operand. Then the input operand **var** uses the constraint **“0”** specifying gcc to use the same constraint than the **0th** operand.

After all, this inline assembly source code will use the **eax** register both as input and as output.

Other Constraints

- **“i”**: an immediate integer operand (one with constant value) is allowed.

  this includes symbolic constants whose values will be known only at assembly time.
- **“n”**: an immediate integer operand with a known numeric value is allowed.
- **“g”**: any register, memory or immediate integer operand is allowed, except for registers that are not general purpose registers.

Other Constraints

- **“i”**: an immediate integer operand (one with constant value) is allowed.

  this includes symbolic constants whose values will be known only at assembly time.
- **“n”**: an immediate integer operand with a known numeric value is allowed.
- “g”: any register, memory or immediate integer operand is allowed, except for registers that are not general purpose registers.

Other Constraints

- **“i”**: an immediate integer operand (one with constant value) is allowed.

  this includes symbolic constants whose values will be known only at assembly time.
- **“n”**: an immediate integer operand with a known numeric value is allowed.
- **“g”**: any register, memory or immediate integer operand is allowed, except for registers that are not general purpose registers.

x86 Specific Constraints

- **“q”**: registers a, b, c or d
- “I”: constant in range 0 to 31 (for 32-bit shifts)
- “J”: constant in range 0 to 63 (for 64-bit shifts)
- “K”: 0xff
- “L”: 0xffff
- “M”: 0, 1, 2, 4 (shifts for lea instruction)
- “N”: constant in range 0 to 255 (for out instruction)

x86 Specific Constraints

- **“q”**: registers a, b, c or d
- **“I”**: constant in range 0 to 31 (for 32-bit shifts)
- **“J”**: constant in range 0 to 63 (for 64-bit shifts)
- **“K”**: 0xff
- **“L”**: 0xffff
- **“M”**: 0, 1, 2, 4 (shifts for lea instruction)
- **“N”**: constant in range 0 to 255 (for out instruction)

x86 Specific Constraints

- **“q”**: registers a, b, c or d
- **“I”**: constant in range 0 to 31 (for 32-bit shifts)
- **“J”**: constant in range 0 to 63 (for 64-bit shifts)
- **“K”**: 0xff
- **“L”**: 0xffff
- **“M”**: 0, 1, 2, 4 (shifts for lea instruction)
- **“N”**: constant in range 0 to 255 (for out instruction)

x86 Specific Constraints

- **“q”**: registers a, b, c or d
- **“I”**: constant in range 0 to 31 (for 32-bit shifts)
- **“J”**: constant in range 0 to 63 (for 64-bit shifts)
- **“K”**: 0xff
- **“L”**: 0xffff
- **“M”**: 0, 1, 2, 4 (shifts for lea instruction)
- **“N”**: constant in range 0 to 255 (for out instruction)

x86 Specific Constraints

- **“q”**: registers a, b, c or d
- **“I”**: constant in range 0 to 31 (for 32-bit shifts)
- **“J”**: constant in range 0 to 63 (for 64-bit shifts)
- **“K”**: 0xff
- **“L”**: 0xffff
- **“M”**: 0, 1, 2, 4 (shifts for lea instruction)
- **“N”**: constant in range 0 to 255 (for out instruction)

x86 Specific Constraints

- **“q”**: registers a, b, c or d
- **“I”**: constant in range 0 to 31 (for 32-bit shifts)
- **“J”**: constant in range 0 to 63 (for 64-bit shifts)
- **“K”**: 0xff
- **“L”**: 0xffff
- **“M”**: 0, 1, 2, 4 (shifts for lea instruction)
- **“N”**: constant in range 0 to 255 (for out instruction)

x86 Specific Constraints

- **“q”**: registers a, b, c or d
- **“I”**: constant in range 0 to 31 (for 32-bit shifts)
- **“J”**: constant in range 0 to 63 (for 64-bit shifts)
- **“K”**: 0xff
- **“L”**: 0xffff
- **“M”**: 0, 1, 2, 4 (shifts for lea instruction)
- **“N”**: constant in range 0 to 255 (for out instruction)

Modifiers

There are also constraint modifiers for more precision over the effects of constraints.

The common constraints modifiers are:

- "=": means that this operand is write-only, the previous value is discarded and replaced by output data.
- "&": means that this operand is an earlyclobber operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address. An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written.

Modifiers

There are also constraint modifiers for more precision over the effects of constraints.

The common constraints modifiers are:

- "=": means that this operand is write-only, the previous value is discarded and replaced by output data.
- "&": means that this operand is an earlyclobber operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address. An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written.

Outline


GCC Inline Assembly Howto


IA-32 Intel Architecture
Software Developer's Manual
Volume 2A: Instruction Set Reference, A-M


IA-32 Intel Architecture
Software Developer's Manual
Volume 2B: Instruction Set Reference, N-Z