Assembler Instructions with C Expression Operands

In an assembler instruction using ‘asm’, you can specify the operands of the instruction using C expressions. This means you need not guess which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881’s ‘fsinx’ instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here ‘angle’ is the C expression for the input operand while ‘result’ is that of the output operand. Each has ‘"f"’ as its operand constraint, saying that a floating point register is required. The ‘=’ in ‘=f’ indicates that the operand is an output; all output operands’ constraints must use ‘=’. The constraints use the same language used in the machine description (*note Constraints::.).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand and another separates the last output operand from the first input, if any. Commas separate the operands within each group. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means or even whether it is valid assembler input. The extended ‘asm’ feature is most often used for machine instructions the compiler itself does not know exist. If the output expression cannot be directly addressed (for example, it is a bit field), your constraint must allow a register. In that case, GNU CC will use the register as the output of the ‘asm’, and then store that register into the output.

The ordinary output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm supports input-output or read-write operands. Use the constraint character ‘+’ to indicate such an operand and list it with the output operands.

When the constraints for the read-write operand (or the operand in which only some of the bits are to be changed) allows a register, you may, as an alternative, logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) ‘combine’ instruction with ‘bar’ as its read-only source operand and ‘foo’ as its read-write destination:

```
     asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint ‘"0"’ for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that ‘foo’ is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work reliably:

```
     asm ("combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of ‘foo’ in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to ‘foo’’s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can’t tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the VAX:

```
     asm volatile ("movc3 %0,%1,%2"
                   : /* no outputs */
                   : "g" (from), "g" (to), "g" (count)
                   : "r0", "r1", "r2", "r3", "r4", "r5");
```

It is an error for a clobber description to overlap an input or output operand (for example, an operand describing a register class with one member, mentioned in the clobber list). Most notably, it is invalid to describe that an input operand is modified, but unused as output. It has to be specified as an input and output operand anyway. Note that if there are only unused output operands, you will then also need to specify ‘volatile’ for the ‘asm’ construct, as described below.

If you refer to a particular hardware register from the assembler code, you will probably have to list the register after the third colon to tell the compiler the register’s value is modified. In some assemblers, the register names begin with ‘%’; to produce one ‘%’ in the assembler code, you must write ‘%%’ in the input.

If your assembler instruction can alter the condition code register, add ‘cc’ to the list of clobbered registers. GNU CC on some machines represents the condition codes as a specific hardware register; ‘cc’ serves to name this register. On other machines, the condition code is handled differently, and specifying ‘cc’ has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add ‘memory’ to the list of clobbered registers. This will cause GNU CC to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single ‘asm’ template, separated either with newlines (written as ‘\n’) or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and most Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands’ addresses, so you can

read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes the subroutine ‘_foo’ accepts arguments in registers 9 and 10:

```
     asm ("movl %0,r9;movl %1,r10;call _foo"
          : /* no outputs */
          : "g" (from), "g" (to)
          : "r9", "r10");
```

Unless an output operand has the ‘&’ constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use ‘&’ for each output operand that may not overlap an input. *Note Modifiers::.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the ‘asm’ construct, as follows:

```
     asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
          : "g" (result)
          : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one ‘asm’ to another are not supported. The compiler’s optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these ‘asm’ instructions is to encapsulate them in macros that look like functions. For example,

```
     #define sin(x)       \
     ({ double __value, __arg = (x);   \
        asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
        __value; })
```

Here the variable ‘__arg’ is used to make sure that the instruction operates on a proper ‘double’ value, and to accept only those arguments ‘x’ which can convert automatically to a ‘double’.

Another way to make sure the instruction operates on the correct data type is to use a cast in the ‘asm’. This is different from using a variable ‘__arg’ in that it converts more different types. For example, if the desired type were ‘int’, casting the argument to ‘int’ would accept a pointer with no complaint, while assigning the argument to an ‘int’ variable named ‘__arg’ would warn about using a pointer unless the caller explicitly casts it.

If an ‘asm’ has output operands, GNU CC assumes for optimization purposes the instruction has no side effects except to change the output operands. This does not mean instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren’t used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an ‘asm’ instruction from being deleted, moved significantly, or combined, by writing the keyword ‘volatile’ after the

‘asm’. For example:

```
#define get_and_set_priority(new)  \
({ int __old; \
   asm volatile ("get_and_set_priority %0, %1": "=g" (__old) : "g" (new)); \
   __old; })
```

If you write an ‘asm’ instruction with no outputs, GNU CC will know the instruction has side-effects and will not delete the instruction or move it outside of loops. If the side-effects of your instruction are not purely external, but will affect variables in your program in ways other than reading the inputs and clobbering the specified registers or memory, you should write the ‘volatile’ keyword to prevent future versions of GNU CC from moving the instruction around within a core region.

An ‘asm’ instruction without any operands or clobbers (and "old style" ‘asm’) will not be deleted or moved significantly, regardless, unless it is unreachable, the same wasy as if you had written a ‘volatile’ keyword.

Note that even a volatile ‘asm’ instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can’t expect a sequence of volatile ‘asm’ instructions to remain perfectly consecutive. If you want consecutive output, use a single ‘asm’.

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn’t arise for ordinary "test" and "compare" instructions because they don’t have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write ‘__asm__’ instead of ‘asm’. *Note Alternate Keywords::.

i386 floating point asm operands

There are several rules on the usage of stack-like regs in asm_operands insns. These rules apply only to the operands that are stack-like regs:

1. Given a set of input regs that die in an asm_operands, it is necessary to know which are implicitly popped by the asm, and which must be explicitly popped by gcc.

   An input reg that is implicitly popped by the asm must be explicitly clobbered, unless it is constrained to match an output operand.

2. For any input reg that is implicitly popped by an asm, it is necessary to know how to adjust the stack to compensate for the pop. If any non-popped input is closer to the top of the reg-stack than the implicitly popped reg, it would not be possible to know what the stack looked like -- it’s not clear how the rest of the stack "slides up".

   All implicitly popped input regs must be closer to the top of the reg-stack than any input that is not implicitly popped.

It is possible that if an input dies in an insn, reload might use the input reg for an output reload. Consider this example:

```
asm ("foo" : "=t" (a) : "f" (b));
```

This asm says that input B is not popped by the asm, and that the asm pushes a result onto the reg-stack, ie, the stack is one deeper after the asm than it was before. But, it is possible that reload will think that it can use the same reg for both the input and the output, if input B dies in this insn.

If any input operand uses the ‘f’ constraint, all output reg constraints must use the ‘&’ earlyclobber.

The asm above would be written as

```
asm ("foo" : "=&t" (a) : "f" (b));
```

3. Some operands need to be in particular places on the stack. All output operands fall in this category -- there is no other way to know which regs the outputs appear in unless the user indicates this in the constraints.

   Output operands must specifically indicate which reg an output appears in after an asm. ‘=f’ is not allowed: the operand constraints must select a class with a single reg.

4. Output operands may not be "inserted" between existing stack regs. Since no 387 opcode uses a read/write operand, all output operands are dead before the asm_operands, and are pushed by the asm_operands. It makes no sense to push anywhere but the top of the reg-stack.

   Output operands must start at the top of the reg-stack: output operands may not "skip" a reg.

5. Some asm statements may need extra stack space for internal calculations. This can be guaranteed by clobbering stack registers unrelated to the inputs and outputs.

Here are a couple of reasonable asms to want to write. This asm takes one input, which is internally popped, and produces two outputs.

```
asm ("fsincos" : "=t" (cos), "=u" (sin) : "0" (inp));
```

This asm takes two inputs, which are popped by the ‘fyl2xp1’ opcode, and replaces them with one output. The user must code the ‘st(1)’ clobber for reg-stack.c to know that ‘fyl2xp1’ pops both inputs.

```
asm ("fyl2xp1" : "=t" (result) : "0" (x), "u" (y) : "st(1)");
```