**SHOT PS3 Practical 2**

**GCC Inline Assembler - Constraints for PPC use**

The following is taken from: <http://gcc.gnu.org/onlinedocs/gcc/Constraints.html#Constraints>

**6.42.1 Simple Constraints**

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

*whitespace* Whitespace characters are ignored and can be inserted at any position except the first. This enables each alternative for different operands to be visually aligned in the machine description even if they have different number of constraints and modifiers.

`m' A memory operand is allowed, with any kind of address that the machine supports in general. Note that the letter used for the general memory constraint can be re-defined by a back end using the TARGET\_MEM\_CONSTRAINT macro.

`o' A memory operand is allowed, but only if the address is *offsettable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an autoincrement or autodecrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter `o' is valid only when accompanied by both `<' (if the target machine has predecrement addressing) and `>' (if the target machine has preincrement addressing).

`V' A memory operand that is not offsettable. In other words, anything that would fit the `m' constraint but not the `o' constraint.

`<' A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed. In inline asm this constraint is only allowed if the operand is used exactly once in an instruction that can handle the side-effects. Not using an operand with `<' in constraint string in the inline asm pattern at all or using it in multiple instructions isn't valid, because the side-effects wouldn't be performed or would be performed more than once. Furthermore, on some targets the operand with `<' in constraint string must be accompanied by special instruction suffixes like %U0 instruction suffix on PowerPC or %P0 on IA-64.

`>' A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed. In inline asm the same restrictions as for `<' apply.

`r' A register operand is allowed provided that it is in a general register.

`i' An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time or later.

`n' An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use `n' rather than `i'.

`I', `J', `K', ... `P' Other letters in the range `I' through `P' may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, `I' is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

`E' An immediate floating operand (expression code const\_double) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

`F' An immediate floating operand (expression code const\_double or const\_vector) is allowed.

`G', `H' `G' and `H' may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

`s' An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an instruction allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use `s' instead of `i'? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between −128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a `moveq' instruction. We arrange for this to happen by defining the letter `K' to mean “any integer outside the range −128 to 127”, and then specifying `Ks' in the operand constraints.

`g' Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

`X' Any operand whatsoever is allowed.

`0', `1', `2', ... `9' An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This number is allowed to be more than a single digit. If multiple digits are encountered consecutively, they are interpreted as a single decimal integer. There is scant chance for ambiguity, since to-date it has never been desirable that `10' be interpreted as matching either operand 1 *or* operand 0. Should this be desired, one can use multiple alternatives instead.

This is called a *matching constraint* and what it really means is that the assembler has only a single operand that fills two roles which asm distinguishes. For example, an add instruction uses two input operands and an output operand, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

addl #35,r12

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

`p' An operand that is a valid memory address is allowed. This is for “load address” and “push address” instructions.

`p' in the constraint must be accompanied by address\_operand as the predicate in the match\_operand. This predicate interprets the mode specified in the match\_operand as the mode of the memory reference for which the address would be valid.

*other-letters* Other letters can be defined in machine-dependent fashion to stand for particular classes of registers or other arbitrary operand types. `d', `a' and `f' are defined on the 68000/68020 to stand for data, address and floating point registers.

**6.42.3 Constraint Modifier Characters**

Here are constraint modifier characters.

**'='** Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

'+' Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. '=' identifies an output; '+' identifies an operand that is both input and output; all other operands are assumed to be input only.

If you specify `=' or `+' in a constraint, you put it in the first character of the constraint string.

'&' Means (in a particular alternative) 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.

`&' applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires `&' while others do not. See, for example, the `movdf' instruction of the 68000.

An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows GCC to produce better code when only some of the inputs can be affected by the earlyclobber. See, for example, the `mulsi3' instruction of the ARM.

`&' does not obviate the need to write `='.

'%' Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints. GCC can only handle one commutative pair in an asm; if you use more, the compiler may fail. Note that you need not use the modifier if the two alternatives are strictly identical; this would only waste time in the reload pass. The modifier is not operational after register allocation, so the result of define\_peephole2 and define\_splits performed after reload cannot rely on `%' to make the intended instruction match.

'#' Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

'\*' Says that the following character should be ignored when choosing register preferences. `\*' has no effect on the meaning of the constraint as a constraint, and no effect on reloading.

#### 6.42.4 Constraints for Particular Machines

Whenever possible, you should use the general-purpose constraint letters in asm arguments, since they will convey meaning more readily to people reading your code. Failing that, use the constraint letters that usually have very similar meanings across architectures. The most commonly used constraints are 'm' and 'r' (for memory and general-purpose registers respectively; see [Simple Constraints](http://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html#Simple-Constraints)), and `I', usually the letter indicating the most common immediate-constant format.

Each architecture defines additional constraints. These constraints are used by the compiler itself for instruction generation, as well as for asm statements; therefore, some of the constraints are not particularly useful for asm. Here is a summary of some of the machine-dependent constraints available on some particular machines; it includes both constraints that are useful for asm and constraints that aren't. The compiler source file mentioned in the table heading for each architecture is the definitive reference for the meanings of that architecture's constraints.

***PowerPC and IBM RS6000—*config/rs6000/rs6000.h**

b Address base register

d Floating point register (containing 64-bit value)

f Floating point register (containing 32-bit value)

v Altivec vector register

wd VSX vector register to hold vector double data

wf VSX vector register to hold vector float data

ws VSX vector register to hold scalar float data

wa Any VSX register

h `MQ', `CTR', or `LINK' register

q `MQ' register

c `CTR' register

l `LINK' register

lr `LINK' register**\***

x `CR' register (condition register) number 0

y `CR' register (condition register)

cc `CR' register (condition register)**\***

z `XER[CA]' carry bit (part of the XER register)

I Signed 16-bit constant

J Unsigned 16-bit constant shifted left 16 bits (use `L' instead for SImode constants)

K Unsigned 16-bit constant

L Signed 16-bit constant shifted left 16 bits

M Constant larger than 31

N Exact power of 2

O Zero

P Constant whose negation is a signed 16-bit constant

G Floating point constant that can be loaded into a register with one instruction per word

H Integer/Floating point constant that can be loaded into a register using three instructions

m Memory operand. Normally, m does not allow addresses that update the base register. If `<' or `>' constraint is also used, they are allowed and therefore on PowerPC targets in that case it is only safe to use `m<>' in an asm statement if that asm statement accesses the operand exactly once. The asm statement must also use `%U*<opno>*' as a placeholder for the “update” flag in the corresponding load or store instruction.

es A “stable” memory operand; that is, one which does not include any automodification of the base register. This used to be useful when `m' allowed automodification of the base register, but as those are now only allowed when `<' or `>' is used, `es' is basically the same as `m' without `<' and `>'.

Q Memory operand that is an offset from a register (it is usually better to use `m' or `es' in asm statements)

Z Memory operand that is an indexed or indirect from a register (it is usually better to use `m' or `es' in asm statements)

R AIX TOC entry

a Address operand that is an indexed or indirect from a register (`p' is preferable for asm statements)

S Constant suitable as a 64-bit mask operand

T Constant suitable as a 32-bit mask operand

U System V Release 4 small data area reference

t AND masks that can be performed by two rldic{l, r} instructions

W Vector constant that does not require memory

j Vector constant that is all zeros.

**\****these alternatives work in the SCE compiler too.*

***Intel 386—*config/i386/constraints.md (***left here for interest!***)**

R Legacy register—the eight integer registers available on all i386 processors (a, b, c, d, si, di, bp, sp).

q Any register accessible as *r*l. In 32-bit mode, a, b, c, and d; in 64-bit mode, any integer register.

Q Any register accessible as *r*h: a, b, c, and d.

a The a register.

b The b register.

c The c register.

d The d register.

S The si register.

D The di register.

A The a and d registers. This class is used for instructions that return double word results in the ax:dx register pair. Single word values will be allocated either in ax or dx. <example deleted>

f Any 80387 floating-point (stack) register.

t Top of 80387 floating-point stack (%st(0)).

u Second from top of 80387 floating-point stack (%st(1)).

y Any MMX register.

x Any SSE register.

Yz First SSE register (%xmm0).

I Integer constant in the range 0 ... 31, for 32-bit shifts.

J Integer constant in the range 0 ... 63, for 64-bit shifts.

K Signed 8-bit integer constant.

L 0xFF or 0xFFFF, for andsi as a zero-extending move.

M 0, 1, 2, or 3 (shifts for the lea instruction).

N Unsigned 8-bit integer constant (for in and out instructions).

G Standard 80387 floating point constant.

C Standard SSE floating point constant.

e 32-bit signed integer constant, or a symbolic reference known to fit that range (for immediate operands in sign-extending x86-64 instructions).

Z 32-bit unsigned integer constant, or a symbolic reference known to fit that range (for immediate operands in zero-extending x86-64 instructions).