IV. Machine Support Library for PDP-11

NAME

Conventions - the PDP-11 runtime library

FUNCTION

The modules in this section are called upon by code produced by the PDP-11 C compiler, primarily to perform operations too lengthy to be expanded in-line. Functions are described in terms of their assembler interface, since they operate outside the conventional C calling protocol. The notation used is that of the Idris assembler. To read code correctly for DEC MACRO-11, make the following substitutions:

~ becomes	$
$	#
*	@
/	;

Thus for example, c~count becomes c$count under DEC operating systems.

Unless explicitly stated otherwise, every function does obey the normal C calling convention that the condition code and registers r0 and r1 are not preserved across a call.

NAME

c~count - counter for profiler

SYNOPSIS

```
mov $lbl,r0
jsr pc,c~count
```

FUNCTION

c~count is the function called on entry to each function when the compiler is run with the flag "-p" given to p2.11. lbl is the address of a pointer variable, initially NULL, that is used by c~count to aid in its bookkeeping.

On entry to c~count, the return link is always presumed to be 010 bytes beyond a function entry point.

RETURNS

Nothing.

NAME

c~dadd - add double into double

SYNOPSIS

```
mov $right,-(sp)
mov $left,-(sp)
jsr pc,c~dadd
```

FUNCTION

c~dadd is the internal routine called in the absence of floating hardware to add the double at right into the double at left. It does so without destroying any volatile registers, so the call can be used much as an ordinary machine instruction.

If right is zero, left is unchanged (x+0); if left is zero, right is copied into it (0+x). Otherwise the number with the smaller characteristic is shifted right until it aligns with the other and the addition is performed algebraically. The answer is rounded.

RETURNS

c~dadd replaces its left operand with the closest internal representation to the rounded sum of its operands. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~ddiv, c~dmul, c~dsub, c~repk, c~unpk

BUGS

It doesn't check for characteristics differing by huge amounts, to save shifting. If the right operand is zero, an unnormalized left operand is left unchanged.

NAME

c~dcmp - compare two doubles

SYNOPSIS

```
mov $right,-(sp)
mov $left,-(sp)
jsr pc,c~dcmp
```

FUNCTION

c~dcmp is the internal routine called in the absence of floating hardware to compare the double at left with the double at right. The comparison involves no floating arithmetic and so is comparatively fast. -0 compares equal with +0.

RETURNS

c~dcmp computes the value +1 in r0 if (left > right), 0 if (left == right), or -1 if (left < right); the condition code is set to reflect this result. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~dsub

BUGS

Unnormalized zeros must have at least the leading seven fraction bits zero.

NAME

c~ddiv - divide double into double

SYNOPSIS

```
mov $right,-(sp)
mov $left,-(sp)
jsr pc,c~ddiv
```

FUNCTION

c~ddiv is the internal routine called in the absence of floating hardware to divide the double at right into the double at left. It does so without destroying any volatile registers, so the call can be used much as an ordinary machine instruction.

If right is zero, left is set to the largest representable floating number, appropriately signed (x/0); if left is zero, it is unchanged (0/x). Otherwise the right fraction is divided into the left and the right exponent is subtracted from that of the left. The sign of a non-zero result is negative if the left and right signs differ, else it is positive. The result is rounded.

RETURNS

c~ddiv replaces its left operand with the closest internal representation to the rounded quotient (left/right), or a huge number if right is zero. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~dadd, c~dmul, c~dsub, c~repk, c~unpk

BUGS

If the left operand is an unnormalized zero, it is left unchanged.

NAME

c~dmul - multiply double into double

SYNOPSIS

```
mov $right,-(sp)
mov $left,-(sp)
jsr pc,c~dmul
```

FUNCTION

c~dmul is the internal routine called in the absence of floating hardware to multiply the double at right into the double at left. It does so without destroying any volatile registers, so the call can be used much as an ordinary machine instruction.

If either right or left is zero, the result is zero (0*x, x*0). Otherwise the right fraction is multiplied into the left and the right exponent is added to that of the left. The sign of a non-zero result is negative if the left and right signs differ, else it is positive. The result is rounded.

RETURNS

c~dmul replaces its left operand with the closest internal representation to the rounded product of its operands. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~dadd, c~ddiv, c~dsub, c~repk, c~unpk

BUGS

If the left operand is an unnormalized zero, it is left unchanged.

NAME

c~dneg - negate double

SYNOPSIS

```
mov $left,-(sp)
jsr pc,c~dneg
```

FUNCTION

c~dneg is the internal routine called in the absence of floating hardware to negate the double at left. It does so without destroying any volatile registers, so the call can be used much as an ordinary machine instruction.

If the number is a normalized zero, it is left alone; otherwise the sign bit is toggled.

RETURNS

c~dneg replaces its left operand with its negative. r0 and r1 are preserved, and the argument is popped off the stack.

SEE ALSO

c~dadd, c~ddiv, c~dsub, c~repk, c~unpk

NAME

c~done - double literal one

SYNOPSIS

```
mov $c~done,-(sp)
mov $c~fac,-(sp)
jsr pc,c~dadd
```

FUNCTION

c~done is a double constant 1.0, which is occasionally used by the compiler in expressions like ++d, or d =+ 1. It resides in the same file as c~fac.

BUGS

It should not be packaged with c~fac, since it may want to be steered into ROM.

NAME

c~dsub - subtract double from double

SYNOPSIS

```
mov $right,-(sp)
mov $left,-(sp)
jsr pc,c~dsub
```

FUNCTION

c~dsub is the internal routine called in the absence of floating hardware to subtract the double at right from the double at left. It does so without destroying any volatile registers, so the call can be used much as an ordinary machine instruction.

c~dsub copies its right operand, negates the copy, and calls c~dadd.

RETURNS

c~dsub replaces its left operand with the closest internal representation to the rounded difference (left - right). r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~dadd, c~dcmp, c~ddiv, c~dmul, c~repk, c~unpk

BUGS

-0 - 0 and -0 - -0 return -0.

NAME

c~dtf - convert double to float

SYNOPSIS

```
mov $right,-(sp)
mov $left,-(sp)
jsr pc,c~dtf
```

FUNCTION

c~dtf is the internal routine called in the absence of floating hardware to convert the double at right into a float at left. It does so by rounding the fraction up if the first discarded bit is a one, adjusting the characteristic as necessary.

RETURNS

c~dtf returns a float in the location pointed at by float. r0 and r1 are preserved, and the arguments are popped off the stack.

NAME

c~dtl - convert double to long

SYNOPSIS

```
mov $right,-(sp)
jsr pc,c~dtl
```

FUNCTION

c~dtl is the internal routine called in the absence of floating hardware to convert a double at right into a long integer on the stack. It does so by calling c~unpk, to separate the fraction from the characteristic, then shifting the fraction until the binary point is at a known fixed place. The long integer immediately to the left of the binary point is delivered, with the same sign as the original double. Truncation occurs toward zero.

RETURNS

c~dtl returns a long on the stack which is the low-order 32 bits of the integer representation of the double pointed at by the argument, truncated toward zero. r0 and r1 are preserved, and the argument is popped off the stack.

SEE ALSO

c~ltd

NAME

c~exch - exchange pointers on stack

SYNOPSIS

```
jsr pc,c~exch
```

FUNCTION

c~exch is the internal routine called whenever two pointers on top of stack are in reverse order.

RETURNS

c~exch returns with the top two words of the stack interchanged. r0 and r1 are preserved, but the condition codes are unpredictable.

NAME

c~fac - the floating accumulators

SYNOPSIS

```
mov $c~fac,-(sp)
mov $c~fac+10,-(sp)
jsr pc,c~dadd
```

FUNCTION

c~fac is a data area large enough to hold two contiguous doubles; it is used in lieu of fr0 and fr1 in the absence of floating point hardware. All C functions that use floating point arithmetic assume both accumulators are volatile on entry; a double (or float widened to double) result is returned in c~fac.

If by any chance interrupt routines make use of float or double data types, then the interrupt entry/exit code must save these quasi registers.

NAME

c~ilsh - signed left shift

SYNOPSIS

```
mov v,-(sp)
mov c,-(sp)
jsr pc,c~ilsh
```

FUNCTION

c~ilsh is the internal routine called in the absence of EIS hardware to shift the integer v right by the count c.

RETURNS

The value returned is the shifted result v << c on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

BUGS

Negative counts are taken as huge positive.

NAME

c~irsh - signed right shift

SYNOPSIS

```
mov v,-(sp)
mov c,-(sp)
jsr pc,c~irsh
```

FUNCTION

c~irsh is the internal routine called in the absence of EIS hardware to shift the integer v right by the count c with sign extension.

RETURNS

The value returned is the shifted result v >> c on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

BUGS

Negative counts are taken as huge positive.

NAME

c~ldiv - divide long by long

SYNOPSIS

```
mov n+2,-(sp)
mov n,-(sp)
mov d+2,-(sp)
mov d,-(sp)
jsr pc,c~ldiv
```

FUNCTION

c~ldiv divides the long n by the long d to obtain the long quotient. The sign of a non-zero result is negative only if the signs of n and d differ. No check is made for division by zero, which currently gives a quotient of -1 or +1.

RETURNS

The value returned is the long quotient of n / d on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~lmod, c~lmul

NAME

c~lils - long left shift

SYNOPSIS

```
mov v+2,-(sp)
mov v,-(sp)
mov c,-(sp)
jsr pc,c~lils
```

FUNCTION

c~lils shifts the long v left by the count c.

RETURNS

The value returned is the shifted result v << c on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

BUGS

Negative counts are taken as huge positive.

NAME

c~lirs - signed long right shift

SYNOPSIS

```
mov v+2,-(sp)
mov v,-(sp)
mov c,-(sp)
jsr pc,c~lirs
```

FUNCTION

c~lirs shifts the long v right by the count c with sign extension.

RETURNS

The value returned is the shifted result v >> c on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

BUGS

Negative counts are taken as huge positive.

NAME

c~lmod - divide long by long and return remainder

SYNOPSIS

```
mov n+2,-(sp)
mov n,-(sp)
mov d+2,-(sp)
mov d,-(sp)
jsr pc,c~lmod
```

FUNCTION

c~lmod divides the long n by the long d to obtain the long remainder. The sign of a non-zero result is negative only if the sign of n is negative. No check is made for division by zero, which currently gives a remainder equal to the value of n.

RETURNS

The value returned is the long remainder of n / d on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~ldiv, c~lmul

NAME

c~lmul - multiply long by long

SYNOPSIS

```
mov l+2,-(sp)
mov l,-(sp)
mov r+2,-(sp)
mov r,-(sp)
jsr pc,c~lmul
```

FUNCTION

c~lmul multiplies the long l by the long r to obtain the long product. The sign of a non-zero result is negative only if the signs of l and r differ. No check is made for overflow, which currently gives the low order 32 bits of the correct product.

RETURNS

The value returned is the long product l * r on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~ldiv, c~lmod

NAME

c~lret - return from long runtime function

SYNOPSIS

```
jmp c~lret
```

FUNCTION

c~lret is the code sequence used to return from several of the long runtime functions. It assumes that the frame pointer r5 has been setup as follows:

```
14(r5)  long left operand
10(r5)  long right operand
6(r5)   return link
4(r5)   old r0
2(r5)   old r1
0(r5)   old r5
-2(r5)  old r4
-4(r5)  old r3
-6(r5)  old r2
```

It further assumes that the left operand has been overwritten with the result of the function, which is to be left on the stack.

RETURNS

c~lret returns with the old registers restored and just the result left on the stack. The condition codes are undefined.

NAME

c~ltd - convert long to double

SYNOPSIS

```
mov r+2,-(sp)
mov r,-(sp)
mov $left,-(sp)
jsr pc,c~ltd
```

FUNCTION

c~ltd is the internal routine called in the absence of floating hardware to convert the long r into a double at left. It does so by extending the long to an unpacked double fraction, then calling c~repk with a suitable characteristic. It does so without destroying any volatile registers, so the call can be used much as an ordinary machine instruction.

RETURNS

c~ltd replaces the operand at left with the double representation of the long integer value passed as an argument. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~dtl, c~ultd

NAME

c~lxor - exclusive or long by long

SYNOPSIS

```
mov l+2,-(sp)
mov l,-(sp)
mov r+2,-(sp)
mov r,-(sp)
jsr pc,c~lxor
```

FUNCTION

c~lxor is the internal routine called in the absence of EIS hardware to exclusive or the long l by the long r. Each bit of the result is set only if the corresponding bits in l and r differ.

RETURNS

The value returned is the long exclusive or l * r on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

NAME

c~repk - repack a double number

SYNOPSIS

```
mov char,-(sp)
mov $frac,-(sp)
jsr pc,c~repk
cmp (sp)+,(sp)+
```

FUNCTION

c~repk is the internal routine called by various floating runtime routines to pack a signed fraction at frac and binary characteristic char into a standard form double representation. The fraction occupies five two-byte integers, starting at frac, and may contain any value; there is an assumed binary point immediately to the right of the most significant byte. The characteristic is 0200 plus the power of two by which the fraction must be multiplied to give the proper value.

If the fraction is zero, the resulting double is all zeros. Otherwise the fraction is forced positive and shifted left or right as needed until the most significant byte is exactly equal to 1, with the characteristic being incremented or decremented as appropriate. The fraction is then rounded to 56 binary places. If the resultant characteristic can be properly represented in a double, it is put in place and the sign is set to match the original fraction sign. If the characteristic is zero or negative, the double is all zeros. Otherwise the characteristic is too large, so the double is set to the largest representable number, and is given the sign of the original fraction.

RETURNS

c~repk replaces the four most significant two-byte integers of the fraction with the double representation. The value of the function is VOID, i.e., garbage.

SEE ALSO

c~unpk

BUGS

Really large magnitude values of char might overflow during normalization and give the wrong approximation to an out of range double value.

NAME

c~ret - return from a C function

SYNOPSIS

```
jmp c~ret
```

FUNCTION

c~ret restores the stack frame and registers in effect on a C call and returns to the routine that called the C function. It is assumed that the new frame was set up by a call to c~sav. The stack frame pointer r5 is used to locate the stored r2, r3, r4, and r5 and to roll back the stack, so sp need not be in a known state (i.e., junk may be left on the stack).

RETURNS

c~ret restores all non-volatile registers and leaves unchanged all others, so as not to disturb a returned value. The condition code is undefined on return from c~ret.

EXAMPLE

The C function:

```
COUNT idiot()
    {
    FAST COUNT i;

    return (i);
    }
```

can be written:

```
idiot:
    jsr r5,c~sav
    mov r4,r0       /return value
    jmp c~ret
```

SEE ALSO

c~rets, c~sav

NAME

c~rets - return from a C function

SYNOPSIS

```
jmp c~rets
```

FUNCTION

c~rets restores the stack frame and registers in effect on a C call and returns to the routine that called the C function. It is assumed that the new frame was set up by a call to c~sav. The stack frame pointer r5 is used to locate the stored r2, r3, r4, and r5 and to roll back the stack, so sp need not be in a known state (i.e., junk may be left on the stack).

RETURNS

Unlike c~ret, c~rets does not restore r2, 23 and r4. The condition code is undefined on return from c~rets.

EXAMPLE

The C function:

```
COUNT idiot()
    {
    COUNT i;

    return (i);
    }
```

can be written:

```
idiot:
    jsr r5,c~sav
    tst -(sp)          / space for i
    mov -10(r5),r0       / return value
    jmp c~rets
```

SEE ALSO

c~ret, c~sav

NAME

c~sav - save register on entering a C function

SYNOPSIS

```
jsr r5,c~sav
```

FUNCTION

c~sav sets up a new stack frame and stacks r4, r3, and r2. It is designed to be called on entry to a C function, at which time:

```
2(sp)  holds the first argument
0(sp)  holds the return link
```

On return fom c~sav, r5 holds sp+012 and:

```
4(r5)  holds first argument
2(r5)  holds return link
0(r5)  holds old r5
-2(r5) holds old r4
-4(r5) holds old r3
-6(r5) holds old r2
```

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at -7(r5) on down. Note that an extra word is pushed on the stack by c~sav, so that C code can use (sp) as a scratch cell; if auto storage is allocated the cell implicitly percolates to (sp) [think about it].

RETURNS

c~sav alters sp and r5 to make the new stack frame. r0 and r1 are not necessarily preserved.

EXAMPLE

The C function:

```
COUNT idiot()
    {
    COUNT i;

    return (i);
    }
```

can be written:

```
idiot:
    jsr r5,c~sav
    tst -(sp)           / space for i
    mov -10(r5),r0      / return value
    jmp c~ret
```

SEE ALSO

c~ret

NAME

c~savs - save register on entering a C function

SYNOPSIS

```
mov $-nautos,r0
jsr r5,c~savs
```

FUNCTION

c~savs sets up a new stack frame, stacks r4, r3, and r2, and ensures that r0+sp is not lower than _stop, to check for stack overflow. It is designed to be called on entry to a C function, at which time:

2(sp) holds the first argument
0(sp) holds the return link

On return from c~savs r5 holds sp+012 and:

4(r5) holds first argument
2(r5) holds return link
0(r5) holds old r5
-2(r5) holds old r4
-4(r5) holds old r3
-6(r5) holds old r2

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at -7(r5) on down. Note that an extra word is pushed on the stack by c~savs, so that C code can use (sp) as a scratch cell; if auto storage is allocated the cell implicitly percolates to (sp) [think about it].

If stack overflow would occur, the _memerr condition is raised. This will never happen if _stop is set to zero.

RETURNS

c~sav alters sp and r5 to make the new stack frame. r0 and r1 are not necessarily preserved.

EXAMPLE

The C function:

```
COUNT idiot()
    {
    COUNT i;

    return (i);
    }
```

can be written:

```
idiot:
        mov $-24,r0
        jsr r5,c~sav
        tst -(sp)               / space for i
        mov -10(r5),r0          / return value
        jmp c~ret
```

SEE ALSO

c~ret

NAME

c~switch - perform C switch statement

SYNOPSIS

```
mov val,r0
mov $swtab,r1
jmp c~switch
```

FUNCTION

c~switch is the code that branches to the appropriate case in a switch statement. It compares val against each entry in swtab until it finds an entry with a matching case value or until it encounters a default entry. swtab entries consist of zero or more (lbl, value) pairs, where lbl is the (nonzero) address to jump to and value is the case value that must match val.

A default entry is signalled by the pair (0, deflbl), where deflbl is the address to jump to if none of the case values match. The compiler always provides a default entry, which points to the statement following the switch if there is no explicit default statement within the switch.

RETURNS

c~switch exits to the appropriate case or default; it never returns.

NAME

c~uldiv - unsigned long divide

SYNOPSIS

```
mov n+2,-(sp)
mov n,-(sp)
mov d+2,-(sp)
mov d,-(sp)
jsr pc,c~uldiv
```

FUNCTION

c~uldiv divides the unsigned long n by the unsigned long d, producing an unsigned long quotient.

No check is made for zero divisor, which currently gives a quotient of -1.

RETURNS

The value returned is the long quotient n / d on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

NAME

c~ulirs - unsigned long right shift

SYNOPSIS

```
mov v+2,-(sp)
mov v,-(sp)
mov c,-(sp)
jsr pc,c~ulirs
```

FUNCTION

c~ulirs shifts the long v right by the count c with no sign extension.

RETURNS

The value returned is the shifted result v >> c on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

BUGS

Negative counts are taken as huge positive.

NAME

c~ulmod - unsigned long remainder

SYNOPSIS

```
mov n+2,-(sp)
mov n,-(sp)
mov d+2,-(sp)
mov d,-(sp)
jsr pc,c~ulmod
```

FUNCTION

c~ulmod divides the unsigned long n by the unsigned long d, producing an unsigned long remainder.

No check is made for zero divisor, which currently gives a remainder of n.

RETURNS

The value returned is the long remainder n / d on the stack. r0 and r1 are preserved, and the arguments are popped off the stack.

NAME

c~ultd - convert unsigned long to double

SYNOPSIS

```
mov r+2,-(sp)
mov r,-(sp)
mov $left,-(sp)
jsr pc,c~ultd
```

FUNCTION

c~ultd is the internal routine called in the absence of floating hardware to convert the unsigned long r into a double at left. It does so by extending the unsigned long to an unpacked double fraction, then calling c~repk with a suitable characteristic. It does so without destroying any volatile registers, so the call can be used much as an ordinary machine instruction.

RETURNS

c~ultd replaces the operand at left with the double representation of the unsigned long integer value passed as an argument. r0 and r1 are preserved, and the arguments are popped off the stack.

SEE ALSO

c~dtl, c~ltd

NAME

c~unpk - unpack a double number

SYNOPSIS

```
mov $double,-(sp)
mov $frac,-(sp)
jsr pc,c~unpk
cmp (sp)+,(sp)+
```

FUNCTION

c~unpk is the internal routine called by various floating runtime routines to unpack a double at double into a signed fraction at frac and a characteristic. The fraction consists of five two-byte integers at frac; the binary point is immediately to the right of the most significant byte. If the double at double is not zero, c~unpk guarantees that the most significant fraction byte is exactly equal to 1, and the least significant fraction integer (the guard word) is zero.

The characteristic returned is 0200 plus the power of two by which the fraction must be multiplied to give the proper value; it will be zero for any flavor of zero at double (i.e., having a characteristic of zero, irrespective of other bits).

RETURNS

c~unpk writes the signed fraction as five two-byte integers starting at frac and returns the characteristic in r0 as the value of the function.

SEE ALSO

c~repk