IV. Machine Support Library for 8086

NAME

Conventions - the 8086 runtime library

FUNCTION

The first nine functions described in this section are C callable routines that provide access to 8086 I/O and memory management instructions. The remaining functions are called on by code produced by the 8086 C compiler, primarily to perform operations too lengthy to be expanded in-line. These last 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 Intel ASM-86, make the following substitution:

/ becomes ;

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

NAME

cs - jump far or read code segment register

SYNOPSIS

```
BYTES cs(off, para)
    BYTES off, para;
```

FUNCTION

cs is a C callable function to perform a far jump, or read the code segment register. If (off is not equal to -1) a long jump is taken to offset off in paragraph para; if that is a far callable function, it will return properly to the caller of cs, but its first four argument words are generally uninteresting.

If you don´t understand this, don´t try it.

RETURNS

If the jump does not occur, cs returns the contents of the code segment register at the time of the call. Otherwise, cs returns only if the jump target is a far callable function which returns; the value of cs is then whatever that function returns.

EXAMPLE

To jump to location 0xffff0:

```
cs(0, -1);
```

SEE ALSO

ds, es, movs, ss

NAME

ds - load or read data segment register

SYNOPSIS

```
BYTES ds(para)
    BYTES para;
```

FUNCTION

ds is a C callable function to load or read the data segment register. If (para is not equal to -1) it is loaded into the register; this should be done only after considerable forethought.

RETURNS

ds returns the contents of the data segment register at the time of the call.

EXAMPLE

To copy from the code segment:

```
    movs(&buffer, ds(-1), &romtable, cs(-1, 0), sizeof (romtable));
```

SEE ALSO

cs, es, movs, ss

NAME

es - load or read extra segment register

SYNOPSIS

```
BYTES es(para)
    BYTES para;
```

FUNCTION

es is a C callable function to load or read the extra segment register. If (para is not equal to -1) it is loaded into the register; this should not affect conventional C programs.

RETURNS

es returns the contents of the extra segment register at the time of the call.

EXAMPLE

To copy from the code segment:

```
    movs(&buffer, es(-1), &romtable, cs(-1, 0), sizeof (romtable));
```

SEE ALSO

cs, ds, movs, ss

NAME

in - input byte from port

SYNOPSIS

```
COUNT in(port)
    UCOUNT port;
```

FUNCTION

in is a C callable function to input a byte from an arbitrary port, using the instruction "in al,dx" with port in dx.

RETURNS

in returns the byte read as an integer whose more significant byte is zero.

EXAMPLE

```
while (!(in(STATUS) & DONE))
    ;
out(DATA, ch);
```

SEE ALSO

inw, out, outw

NAME

inw - input word from port

SYNOPSIS

```
COUNT inw(port)
    UCOUNT port;
```

FUNCTION

inw is a C callable function to input a word from an arbitrary port, using the instruction "in ax,dx" with port in dx.

RETURNS

inw returns the word read as an integer.

EXAMPLE

```
while (!(inw(STATUS) & DONE))
    ;
outw(DATA, ch);
```

SEE ALSO

in, out, outw

NAME

movs - move memory to memory

SYNOPSIS

```
BYTES movs(di, es, si, ds, cx)
    TEXT *di, *si;
    BYTES es, ds, cx;
```

FUNCTION

movs moves cx bytes, beginning at offset si in paragraph ds, to offset di in paragraph es. It does so by using the string move instructions, for maximum efficiency. No checking is made to see whether this is a good idea.

RETURNS

movs returns (di + cx).

EXAMPLE

To concatenate two arrays into a third:

```
dp = ds(-1);
movs(movs(dest, dp, src1, dp, size1), dp, src2, dp, size2);
```

SEE ALSO

cs, ds, es, ss

NAME

out - output byte to port

SYNOPSIS

```
COUNT out(port, data)
    UCOUNT port, data;
```

FUNCTION

out is a C callable function to output a byte to an arbitrary port, using the instruction "out dx,al" with port in dx and data in ax.

RETURNS

out returns data.

EXAMPLE

```
while (!(in(STATUS) & DONE))
    ;
out(DATA, ch);
```

SEE ALSO

in, inw, outw

NAME

outw - output word to port

SYNOPSIS

```
COUNT outw(port, data)
    UCOUNT port, data;
```

FUNCTION

outw is a C callable function to output a word to an arbitrary port, using the instruction "in ax,dx" with port in dx and data in ax.

RETURNS

outw returns data.

EXAMPLE

```
while (!(inw(STATUS) & DONE))
    ;
outw(DATA, x);
```

SEE ALSO

in, inw, out

NAME

ss - load or read stack segment register

SYNOPSIS

```
BYTES ss(off, para)
    BYTES off, para;
```

FUNCTION

ss is a C callable function to load or read the stack segment register. If (off is not equal to -1) it is loaded into the stack pointer, and para is loaded into the stack segment register; the new stack had better have a suitable new frame pointer on its top, followed by three sacrificial words for the return link and arguments.

If you don′t understand this, don′t try it.

RETURNS

ss returns the contents of the stack segment register after the call.

SEE ALSO

cs, ds, es, movs

NAME

c_cfcc - copy 8087 condition code

SYNOPSIS

```
        call    c_cfcc
```

FUNCTION

c_cfcc is the internal routine called, in the presence of the 8087 coprocessor, to copy the coprocessor condition code to that of the CPU. It moves the more significant byte of the 8087 status word to the CPU flags, leaving all other registers unmodified.

Note that only the C and Z flags are set properly, so relational tests on floating results must be unsigned, not signed.

RETURNS

c_cfcc returns the coprocessor comparison status in the flags, suitable for tests using the C and Z flags. All other registers are preserved.

NAME

c_count - counter for profiler

SYNOPSIS

```
        lea     ax,lbl
        call    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 p2.86. 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 seven bytes beyond a function entry point.

RETURNS

Nothing.

NAME

c_dadd - add double into double

SYNOPSIS

```
    lea     ax,left
    push    ax
    lea     ax,right
    push    ax
    call    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 like 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. ax, cx, and dx 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

```
        lea     ax,left
        push    ax
        lea     ax,right
        push    ax
        call    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 returns with the N, V, and Z flags set analogously to the integer instruction "cmp left,right". ax, cx, and dx are preserved, and the arguments are popped off the stack.

SEE ALSO

c_dsub

BUGS

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

NAME

c_ddiv - divide double into double

SYNOPSIS

```
        lea     ax,left
        push    ax
        lea     ax,right
        push    ax
        call    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 like 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. ax, cx, and dx 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

```
        lea     ax,left
        push    ax
        lea     ax,right
        push    ax
        call    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 like 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. ax, cx, and dx 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_dsub - subtract double from double

SYNOPSIS

```
        lea     ax,left
        push    ax
        lea     ax,right
        push    ax
        call    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 like 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). ax, cx, and dx 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

```
        lea     ax,left
        push    ax
        lea     ax,right
        push    ax
        call    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.

RETURNS

c_dtf returns a float in the location pointed at by left. ax, cx, and dx are preserved, and the arguments are popped off the stack.

SEE ALSO

c_ftd

BUGS

No checks are made for overflow or underflow, which give garbage results.

NAME

c_dtl - convert double to long

SYNOPSIS

```
        lea     ax,right
        push    ax
        call    c_dtl
    /   long now in dx/ax
```

FUNCTION

c_dtl is the internal routine called in the absence of floating hardware to convert a double at right into a long integer in dx/ax. 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 in dx/ax which is the low-order 32 bits of the integer representation of the double pointed at by the argument, truncated toward zero. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_ltd, c_unpk

NAME

c_fac - the floating accumulators

SYNOPSIS

```
        lea     ax,c_fac
        push    ax
        lea     ax,c_fac[8]
        push    ax
        call    c_dadd
```

FUNCTION

c_fac is a data area large enough to hold two contiguous doubles; it is used in lieu of fr0 and fr7 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_fret - return from a far C function

SYNOPSIS

```
jmp c_fret
```

FUNCTION

c_fret restores the stack frame and registers in effect on a C call and returns, via an intersegment return, 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 bp is used to locate the stored di, si, bx, and bp 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_fret 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_fret.

EXAMPLE

The C function:

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

    return (i);
    }
```

can be written:

```
idiot:
    call    c_sav
    mov     ax,si
    jmp     c_fret
```

SEE ALSO

c_ret, c_rets, c_sav, c_savs

NAME

c_ftd - convert float to double

SYNOPSIS

```
        lea     ax,left
        push    ax
        lea     ax,right
        push    ax
        call    c_ftd
```

FUNCTION

c_ftd is the internal routine called in the absence of floating hardware to convert the float at right into a double at left. It does so without destroying any volatile registers, so the call can be used much like an ordinary machine instruction.

RETURNS

c_ftd replaces the operand at left with the double representation of the float integer value at right. ax, cx, and dx are preserved, and the arguments are popped off the stack.

SEE ALSO

c_dtf

NAME

c_iax - jump indirect on ax

SYNOPSIS

```
        call    c_iax
```

FUNCTION

c_iax performs an indirect intrasegment jump on the contents of ax. It is used to call a function specified by a pointer expression.

RETURNS

c_iax does not return to its caller, but it typically is used to jump to a function that does return to the caller.

NAME

c_lcmp - compare two signed longs

SYNOPSIS

```
        push    right[2]
        push    right
    /   left operand in dx/ax
        call    c_lcmp
```

FUNCTION

c_lcmp is the internal routine called to compare the signed long integer in dx/ax to the signed long right.

RETURNS

c_lcmp returns with the N, V, and Z flags set analogously to the integer instruction "cmp left,right". ax, cx, and dx are preserved, and the argument is popped off the stack.

NAME

c_ldiv - divide signed long by long

SYNOPSIS

```
        push    right[2]
        push    right
    /   left operand in dx/ax
        call    c_ldiv
```

FUNCTION

c_ldiv divides the signed long in dx/ax by the long right to obtain the signed long quotient. The sign of a non-zero result is negative only if the signs of left and right differ. No check is made for division by zero, which currently gives a quotient of -1 or +1.

RETURNS

The value returned is the signed long quotient in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_lmod, c_lmul, c_uldiv, c_ulmod

NAME

c_llsh - shift long left by count

SYNOPSIS

```
        push    right
    /   left operand in dx/ax
        call    c_llsh
```

FUNCTION

c_llsh shifts the long in dx/ax left by the count right.

RETURNS

The value returned is (left << right) in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_lrsh, c_ulrsh

NAME

c_lmod - remainder signed long by long

SYNOPSIS

```
        push    right[2]
        push    right
    /   left operand in dx/ax
        call    c_lmod
```

FUNCTION

c_lmod divides the signed long in dx/ax by the long right to obtain the signed long remainder. The sign of a non-zero result is negative only if the signs of left and right differ. No check is made for division by zero, which currently gives a remainder equal to the left operand.

RETURNS

The value returned is the signed long remainder in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_ldiv, c_lmul, c_uldiv, c_ulmod

NAME

c_lmul - multiply long by long

SYNOPSIS

```
        push    right[2]
        push    right
    /   left operand in dx/ax
        call    c_lmul
```

FUNCTION

c_lmul multiplies the long in dx/ax by the long right to obtain the long product. The sign of a non-zero result is negative only if the signs of left and right 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 (left * right) in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_ldiv, c_lmod, c_uldiv, c_ulmod

NAME

c_lrsh - shift signed long right by count

SYNOPSIS

```
        push    right
    /   left operand in dx/ax
        call    c_lrsh
```

FUNCTION

c_lrsh shifts the signed long in dx/ax right by the count right.

RETURNS

The value returned is (left >> right) in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_llsh, c_ulrsh

NAME

c_ltd - convert signed long to double

SYNOPSIS

```
        lea     cx,left
        push    cx
    /   right operand in dx/ax
        call    c_ltd
```

FUNCTION

c_ltd is the internal routine called in the absence of floating hardware to convert the signed long in dx/ax into a double at left. It does so by extending the signed 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 like an ordinary machine instruction.

RETURNS

c_ltd replaces the operand at left with the double representation of the signed long integer in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_dtl, c_repk, c_ultd

NAME

c_repk - repack a double number

SYNOPSIS

```
        push    char
        lea     ax,frac
        push    ax
        call    c_repk
        pop     cx
        pop     cx
```

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 1023 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 53 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

Values of char very large in magnitude 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 bp is used to locate the stored di, si, bx, and bp 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:
    call    c_sav
    mov     ax,si
    jmp     c_ret
```

SEE ALSO

c_rets, c_sav, c_savs

NAME

c_rets - return from a C function, restoring no registers

SYNOPSIS

```
    jmp c_rets
```

FUNCTION

c_rets restores the stack frame 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 bp is used to locate the stored bp 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_rets restores bp and leaves unchanged all others, so as not to disturb a returned value. The condition code is undefined on return from c_rets.

EXAMPLE

The C function:

```
    COUNT idiot()
        {
        COUNT i;

        return (i);
        }
```

can be written:

```
    idiot:
        call    c_sav
        push    cx
        mov     ax,[bp][-8]
        jmp     c_rets
```

SEE ALSO

c_ret, c_sav, c_savs

NAME

c_sav - save register on entering a C function

SYNOPSIS

```
        call    c_sav
```

FUNCTION

c_sav sets up a new stack frame and stacks bx, si, and di. It is designed to be called on entry to a C function, at which time:

```
[sp][2]  holds the first argument
[sp][0]  holds the return link
```

On return fom c_sav bp holds sp+8 and:

```
[bp][4]  holds first argument
[bp][2]  holds return link
[bp][0]  holds old bp
[bp][-2] holds old bx
[bp][-4] holds old si
[bp][-6] holds old di
```

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at [bp][-7] on down.

RETURNS

c_sav alters sp and bp to make the new stack frame. ax, cx, and dx are not necessarily preserved.

EXAMPLE

The C function:

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

        return (i);
        }
```

can be written:

```
    idiot:
        call    c_sav
        mov     ax,si
        jmp     c_ret
```

SEE ALSO

c_ret, c_rets, c_savs

NAME

c_savs - save register and check for stack overflow on

SYNOPSIS

```
        mov     ax,-nautos
        call    c_savs
```

FUNCTION

c_savs sets up a new stack frame, stacks bx, si, and di, and ensures that ax+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:

```
[sp][2]  holds the first argument
[sp][0]  holds the return link
```

On return from c_savs bp holds sp+8 and:

```
[bp][4]  holds first argument
[bp][2]  holds return link
[bp][0]  holds old bp
[bp][-2] holds old bx
[bp][-4] holds old si
[bp][-6] holds old di
```

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at [bp][-7] on down.

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 bp to make the new stack frame. ax, cx, and dx are not necessarily preserved.

EXAMPLE

The C function:

```
    COUNT idiot()
        {
        COUNT i;

        return (i);
        }
```

can be written:

```
idiot:
    mov     ax,-24
    call    c_savs
    push    cx
    mov     ax,[bp][-8]
    jmp     c_ret
```

SEE ALSO

c_ret, c_rets, c_sav

NAME

c_switch - perform C switch statement

SYNOPSIS

```
        mov     ax,val
        lea     dx,swtab
        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.

Note that the switch table is assumed to be in the code segment.

RETURNS

c_switch exits to the appropriate case or default; it never returns.

NAME

c_uldiv - divide unsigned long by long

SYNOPSIS

```
        push    right[2]
        push    right
    /   left operand in dx/ax
        call    c_uldiv
```

FUNCTION

c_uldiv divides the unsigned long in dx/ax by the long right to obtain the unsigned long quotient. No check is made for division by zero, which currently gives a quotient of -1.

RETURNS

The value returned is the unsigned long quotient in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_ldiv, c_lmod, c_lmul, c_ulmod

NAME

c_ulmod - remainder unsigned long by long

SYNOPSIS

```
        push    right[2]
        push    right
    /   left operand in dx/ax
        call    c_ulmod
```

FUNCTION

c_ulmod divides the unsigned long in dx/ax by the long right to obtain the unsigned long remainder. No check is made for division by zero, which currently gives a remainder equal to the left operand.

RETURNS

The value returned is the unsigned long remainder in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_ldiv, c_lmod, c_lmul, c_uldiv

NAME

c_ulrsh - shift unsigned long right by count

SYNOPSIS

```
        push    right
    /   left operand in dx/ax
        call    c_ulrsh
```

FUNCTION

c_ulrsh shifts the unsigned long in dx/ax right by the count right.

RETURNS

The value returned is (left >> right) in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_llsh, c_lrsh

NAME

c_ultd - convert unsigned long to double

SYNOPSIS

```
        lea     cx,left
        push    cx
    /   right operand in dx/ax
        call    c_ultd
```

FUNCTION

c_ultd is the internal routine called in the absence of floating hardware to convert the unsigned long in dx/ax 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 like an ordinary machine instruction.

RETURNS

c_ultd replaces the operand at left with the double representation of the unsigned long integer in dx/ax. cx is preserved, and the argument is popped off the stack.

SEE ALSO

c_dtl, c_ltd, c_repk

NAME

c_unpk - unpack a double number

SYNOPSIS

```
        lea     ax,double
        push    ax
        lea     ax,frac
        push    ax
        call    c_unpk
        pop     cx
        pop     cx
```

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 1023 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 ax as the value of the function.

SEE ALSO

c_repk