APPENDIX C

Z80/HD64180 MACHINE LIBRARY

This appendix describes each of the functions in the Machine Library **(libm).** These functions provide the interface between the Z80/HD64180 microcontroller hardware and the functions in the C Library **(liba).** They are described in reference form, and listed alphabetically.

NAME

Conventions - using the Z80/HD64180 Machine Support Library

FUNCTION

The Z80/HD64180 Machine Support Library is a collection of those routines needed by the C compiler to augment the code it produces. It also turns out to be pretty useful to anyone who must write machine-level code for the Z80/HD64180. To use it, however, requires at least a basic knowledge of how C does business.

Unless explicitly stated, every function does obey the normal C calling convention that registers af, bc, and hl are not preserved across a call.

The data types of C are:

char - or one byte integer.

short - or two-byte integer,also know simply as int or integer. Stored less significant byte first, as the Z80/HD64180 prefers.

unsigned - is the same as int, except that the sign bit is just another magnitude bit. All memory addresses are treated as unsigned, to byte level.

long - or long integer, is a four-byte integer. Stored as two integers, more significant integer first. Note that this means the order of bytes in memory is (2, 3, 0, 1), where 0 is the least significant byte. This particular representation is more useful than may at first be apparent. A long may also be unsigned.

float - is a four-byte floating point number. Representation is the same as double, with the last four bytes discarded, i.e., the four least significant fraction bytes.

double - is an eight-byte floating point number. It is stored as four integers, most significant integer first, i.e., in the order (6, 7, 4, 5, 2, 3, 0, 1). Representation is the same as for PDP-11 computers: most significant bit is one for negative numbers, else zero; next eight bits are the characteristic, biased such that the binary expo- nent of the number is the characteristic minus 0200; remaining bits are the fraction, starting with the 1/4 weighted bit. If the characteristic is zero, the entire number is taken as zero and should be all zeros to avoid confusing some routines that take shortcuts. Otherwise, there is an assumed 1/2 added to all fractions to put them in the interval [0.5, 1.0). The value of the number is the fraction, times -1 if the sign bit is set, times two raised to the exponent.

Names in C may contain letters, digits, and underscores '_'. To avoid collisions with predefined identifiers, the compiler prepends an underscore '_' to each symbol. Thus, the function name "func" becomes "_func".

NAME

c.bbtou - unpack bits (in byte bitfield) to unsigned

SYNOPSIS

```
/ pointer to bits on stack
/ offset/size on stack
  call c.bbtou
/ unsigned on stack
```

FUNCTION

c.bbtou is the internal routine called by C to unpack the bitfield at bits into an unsigned on the stack. The field is specified by the two bytes offset/size, where the less significant byte offset is the number of places the bitfield must be shifted right to align it as an integer, and the more significant byte size is the number of bits in the field. offset is assumed to be in the range [0, 8), while size is in the range (0,8].

RETURNS

c.bbtou returns the bitfield unpacked into an unsigned integer, left on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.utob

NAME

c.btou - unpack bits to unsigned

SYNOPSIS

```
/ pointer to bits on stack
/ offset/size on stack
  call c.btou
/ unsigned on stack
```

FUNCTION

c.btou is the internal routine called by C to unpack the bitfield at bits into an unsigned on the stack. The field is specified by the two bytes offset/size, where the less significant byte offset is the number of places the bitfield must be shifted right to align it as an integer, and the more significant byte size is the number of bits in the field. offset is assumed to be in the range [0, 16), while size is in the range (0,16].

RETURNS

c.btou returns the bitfield unpacked into an unsigned integer, left on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.utob

NAME

c.butob - pack unsigned into (byte bitfield) bits

SYNOPSIS

```
/ pointer to bits on stack
/ unsigned on stack
/ offset/size on stack
  call c.butob
/ pointer to bits still on stack
```

FUNCTION

c.butob is the internal routine called by C to pack unsigned into the byte bitfield at bits. The field is specified by the two bytes offset/size, where the less significant byte offset is the number of places the bitfield must be shifted right to align it as an integer, and the more significant byte size is the number of bits in the field. offset is assumed to be in the range [0, 8), while size is in the range (0, 8].

RETURNS

c.butob inserts the unsigned into the specified bitfield at bits. All registers but af are preserved, and all arguments but the pointer to bits are popped off the stack.

SEE ALSO

c.bbtou

NAME

c.dadd - add double into double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dadd
/ pointer to left still on stack
```

FUNCTION

c.dadd is the internal routine called by C 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. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ddiv, c.dmul, c.dsub

NOTES

It doesn't check for characteristics differing by huge amounts, to save shifting. (-0 + 0) and (-0 + -0) return -0.

NAME

c.dcmp - compare two doubles

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dcmp
/ no pointers left on stack
```

FUNCTION

c.dcmp is the internal routine called by C 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 NZ set properly in f to reflect (left :: right); C is the same as N. All registers but a are preserved, and the arguments are popped off the stack.

SEE ALSO

c.dsub

NAME

c.dcpy - copy double to double

SYNOPSIS

```
/ pointer to left in bc
/ pointer to right hl
  call c.dcpy
```

FUNCTION

c.dcpy moves the double at right to the double at left.

RETURNS

Nothing. None of the volatile registers af, bc, or hl are preserved.

SEE ALSO

c.lcpy

NAME

c.ddiv - divide double into double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
call c.ddiv
/ pointer to left still on stack
```

FUNCTION

c.ddiv is the internal routine called by C 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 the 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. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dadd, c.dmul, c.dsub

NAME

c.dmul - multiply double into double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dmul
/ pointer to left still on stack
```

FUNCTION

c.dmul is the internal routine called by C 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 the 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. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dadd, c.ddiv, c.dsub

NAME

c.dneg - negate double

SYNOPSIS

```
/ pointer to left on stack
  call c.dneg
/ pointer to left still on stack
```

FUNCTION

c.dneg negates the double at left in place. If the number is normalized, an unnormalized zero will never be produced.

RETURNS

The value returned is -left stored at left. All registers but af are preserved.

NAME

c.dsub - subtract double from double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dsub
/ pointer to left still on stack
```

FUNCTION

c.dsub is the internal routine called by C 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). All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dadd, c.dcmp, c.ddiv, c.dmul

NOTES

(-0 - 0) and (-0 - -0) return -0.

NAME

c.dtd - move double to double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dtd
/ pointer to left still on stack
```

FUNCTION

c.dtd is the internal routine called by C to move a double at right into a double at left.

RETURNS

c.dtd returns a copy of the double at right in the double at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dtf, c.ftd

NAME

c.dtf – convert double to float

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dtf
/ pointer to left still on stack
```

FUNCTION

c.dtf is the internal routine called by C 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 left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dtd, c.ftd

NAME

c.dti - convert double to int

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dti
/ pointer to left still on stack
```

FUNCTION

c.dti is the internal routine called by C to convert a double at right into an integer at left. 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 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.dti returns a integer at left which is the low-order 16 bits of the integer representation of the double at right, truncated toward zero. All registers but af are preserved, and the right operand is popped off the stack.

SEE ALSO

c.dtr, c.itd

NAME

c.dtl - convert double to long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.dtl
/ pointer to left still on stack
```

FUNCTION

c.dtl is the internal routine called by C to convert a double at right into a long integer at left. 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 at left which is the low-order 32 bits of the integer representation of the double pointed at by right, truncated toward zero. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ltd

NAME

c.dtr - convert double to int on stack

SYNOPSIS

```
/ pointer to right on stack
  call c.dtr
/ integer on stack
```

FUNCTION

c.dtr is the internal routine called by C to convert a double at right into an 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 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.dtr returns a integer at left which is the low-order 16 bits of the integer representation of the double at right, truncated toward zero. All registers but af are preserved, and the right operand is popped off the stack.

SEE ALSO

c.dti, c.itd

NAME

c.fadd - add float into float

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.fadd
/ pointer to left still on stack
```

FUNCTION

c.fadd is the internal routine called by C to add the float at right into the float 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.fadd replaces its left operand with the closest internal representation to the rounded sum of its operands. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.fdiv, c.fmul, c.fsub

NOTES

It doesn't check for characteristics differing by huge amounts, to save shifting. (-0 + 0) and (-0 + -0) return -0.

NAME

c.fcmp - compare two floats

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.fcmp
/ no pointers left on stack
```

FUNCTION

c.fcmp is the internal routine called by C to compare the float at left with the float at right. The comparison involves no floating arithmetic and so is comparatively fast. -0 compares equal with +0.

RETURNS

c.fcmp returns NZ set properly in f to reflect (left :: right); C is the same as N. All registers but a are preserved, and the arguments are popped off the stack.

SEE ALSO

c.fsub

NAME

c.fcpy – copy float to float

SYNOPSIS

```
/ pointer to left in bc
/ pointer to right hl
  call c.fcpy
```

FUNCTION

c.fcpy moves the float at right to the float at left.

RETURNS

Nothing. None of the volatile registers af, bc, or hl are preserved.

SEE ALSO

c.lcpy

NAME

c.fdiv - divide float into float

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
call c.fdiv
/ pointer to left still on stack
```

FUNCTION

c.fdiv is the internal routine called by C to divide the float at right into the float 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 the result is negative if the left and right signs differ, else it is positive. The result is rounded.

RETURNS

c.fdiv replaces its left operand with the closest internal representation to the rounded quotient (left/right), or a huge number if right is zero. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.fadd, c.fmul, c.fsub

NAME

c.fmul - multiply float into float

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.fmul
/ pointer to left still on stack
```

FUNCTION

c.fmul is the internal routine called by C to multiply the float at right into the float 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 the result is negative if the left and right signs differ, else it is positive. The result is rounded.

RETURNS

c.fmul replaces its left operand with the closest internal representation to the rounded product of its operands. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.fadd, c.fdiv, c.fsub

NAME

c.fmvd - copy double to double

SYNOPSIS

```
/  hl = address of right
/  bc = address of left
   call c.fmvd
```

FUNCTION

c.fmvd moves the double (eight bytes of data) whose address is in **hl** to the double whose address is in **bc.** This routine trashes registers **bc** and **hl.**

RETURNS

Nothing.

NAME

c.fmvl – copy float to float

SYNOPSIS

```
/  hl = address of right
/  bc = address of left
   call c.fmvl
```

FUNCTION

c.fmvl moves the float (four bytes of data) whose address is in **hl** to the float whose address is in **bc.** This routine trashes registers **bc** and **hl.**

RETURNS

Nothing.

NAME

c.fneg - negate float

SYNOPSIS

```
/ pointer to left on stack
  call c.fneg
/ pointer to left still on stack
```

FUNCTION

c.fneg negates the float at left in place. If the number is normalized, an unnormalized zero will never be produced.

RETURNS

The value returned is -left stored at left. All registers but af are preserved.

NAME

c.frepk – repack a float number

SYNOPSIS

```
/ characteristic on stack
/ pointer to frac on stack
  call c.frepk
  sp => af => af
```

FUNCTION

c.frepk is the internal routine called by various floating run-time routines to pack a signed fraction at frac and a two-byte binary characteristic into a standard form float representation. The fraction occupies five bytes, starting at frac and stored least significant byte first, 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 float is all zeros. Otherwise the fraction is forced positive and shifted left or right as needed to bring the fraction into the interval [0.5, 1.0), with the characteristic being incremented or decremented as appropriate. The fraction is then rounded to 24 binary places. If the resultant characteristic can be properly represented in a float, it is put in place and the sign is set to match the original fraction sign. If the characteristic is zero or negative, the float is all zeros. Otherwise the characteristic is too large, so the float is set to the largest representable number, and is given the sign of the original fraction.

RETURNS

c.frepk replaces the first four (least significant) bytes of the fraction with the float representation, i.e., two two-byte integers, most sig- nificant integer first. The value of the function is VOID, i.e., garbage. The registers af, bc, and hl are not preserved.

SEE ALSO

c.funpk

NOTES

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

NAME

c.fsub - subtract float from float

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.fsub
/ pointer to left still on stack
```

FUNCTION

c.fsub is the internal routine called by C to subtract the float at right from the float at left. It does so without destroying any volatile registers, so the call can be used much like an ordinary machine instruction.

c.fsub copies its right operand, negates the copy, and calls c.fadd.

RETURNS

c.fsub replaces its left operand with the closest internal representation to the rounded difference (left - right). All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.fadd, c.fcmp, c.fdiv, c.fmul

NOTES

(-0 - 0) and (-0 - -0) return -0.

NAME

c.ftd - convert float to double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ftd
/ pointer to left still on stack
```

FUNCTION

c.ftd is the internal routine called by C to convert the float at right into a double at left. It does so by appending four fraction bytes of zeros to the four-byte float.

RETURNS

c.ftd returns a double in the location pointed at by left whose value matches the float at right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dtd, c.dtf

NAME

c.fti - convert float to int

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.fti
/ pointer to left still on stack
```

FUNCTION

c.fti is the internal routine called by C to convert a float at right into an integer at left. It does so by calling c.funpk, to separate the fraction from the characteristic, then shifting the fraction until the binary point is at a known fixed place. The integer immediately to the left of the binary point is delivered, with the same sign as the original float. Truncation occurs toward zero.

RETURNS

c.fti returns a integer at left which is the low-order 16 bits of the integer representation of the float at right, truncated toward zero. All registers but af are preserved, and the right operand is popped off the stack.

SEE ALSO

c.ftr, c.itf

NAME

c.ftl – convert float to long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ftl
/ pointer to left still on stack
```

FUNCTION

c.ftl is the internal routine called by C to convert a float at right into a long integer at left. 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 float. Truncation occurs toward zero.

RETURNS

c.ftl returns a long at left which is the low-order 32 bits of the integer representation of the float pointed at by right, truncated toward zero. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ltf

NAME

c.ftr - convert float to int on stack

SYNOPSIS

```
/ pointer to right on stack
  call c.ftr
/ integer on stack
```

FUNCTION

c.ftr is the internal routine called by C to convert a float at right into an 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 integer immediately to the left of the binary point is delivered, with the same sign as the original float. Truncation occurs toward zero.

RETURNS

c.ftr returns a integer at left which is the low-order 16 bits of the integer representation of the float at right, truncated toward zero. All registers but af are preserved, and the right operand is popped off the stack.

SEE ALSO

c.fti, c.itf

NAME

c.funpk - unpack a float number

SYNOPSIS

```
/ pointer to float on stack
/ pointer to frac on stack
  call c.funpk
  sp => af => af
```

FUNCTION

c.funpk is the internal routine called by various floating run-time routines to unpack a float at float into a signed fraction at frac and a characteristic. The fraction consists of five bytes at frac, stored least significant byte first; the binary point is immediately to the right of the most significant byte. If the float at float is not zero, c.unpk guarantees that the magnitude of the fraction is in the interval [0.5, 1.0). The least significant byte is guaranteed to be zero; it serves as a guard byte.

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 float (i.e., having a characteristic of zero, irrespective of other bits).

RETURNS

c.funpk writes the signed fraction as five bytes starting at frac and stored least significant byte first, and returns the characteristic in bc as the value of the function. The registers af and hl are not preserved.

SEE ALSO

c.frepk

NAME

c.ibc - jump on bc

SYNOPSIS

```
bc = &func
call c.ibc
```

FUNCTION

c.ibc is used by the C compiler to enter a function, which has one or more arguments and whose address is not known at compile time, as when calling a function given a pointer to it. It simply performs a

```
    jmp *bc
```

which presumably enters the function.

c.ibc can also be used as the target of various conditional jumps and calls, to extend the reach of these instructions.

RETURNS

c.ibc returns whatever the function at bc returns.

NAME

c.idiv - divide integer by integer

SYNOPSIS

```
/ left on stack
/ right on stack
  call c.idiv
/ quotient on stack
```

FUNCTION

c.idiv divides the integer left by the integer right to obtain the integer quotient. The sign of a nonzero 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 integer quotient of left/right on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.imod, c.udiv, c.umod

NAME

c.ihl - jump on hl

SYNOPSIS

```
hl = &func
call c.ihl
```

FUNCTION

c.ihl is used by the C compiler to enter a function, which has no arguments and whose address is not known at compile time, as when calling a function given a pointer to it. It simply performs a

```
    jmp *hl
```

which presumably enters the function.

c.ihl can also be used as the target of various conditional jumps and calls, to extend the reach of these instructions.

RETURNS

c.ihl returns whatever the function at hl returns.

NAME

c.ilsh - integer left shift

SYNOPSIS

```
/ integer val on stack
/ integer count on stack
  call c.ilsh
/ integer result on stack
```

FUNCTION

c.ilsh shifts the integer val left by the integer count. If count is negative, an arithmetic right shift occurs instead. If count is positive, the result is valid for unsigned val.

RETURNS

The value returned is the shifted integer result val<<count on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.irsh, c.ursh

NOTES

count is blindly reduced modulo 256; no checking is performed for ridiculously long shifts (16, 128), which take a long time.

NAME

c.imod - remainder of integer divided by integer

SYNOPSIS

```
/ left on stack
/ right on stack
  call c.imod
/ remainder on stack
```

FUNCTION

c.imod divides the integer left by the integer right to obtain the integer remainder. The sign of a nonzero result is the same as the sign of left. No check is made for division by zero, which currently gives a remainder equal to left.

RETURNS

The value returned is the integer remainder left%right on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.idiv, c.udiv, c.umod

NAME

c.imul - multiply integer by integer

SYNOPSIS

```
/ integer left on stack
/ integer right on stack
 call c.imul
/ integer result on stack
```

FUNCTION

c.imul multiplies the integer left by the integer right to obtain the integer product. The sign of a nonzero result is negative only if the signs of left and right differ. No check is made for overflow, which currently gives the low order 16 bits of the correct product. The result of c.imul is also valid for unsigned operands.

RETURNS

The value returned is the integer product left*right on the stack. All registers but af are preserved, and the arugments are popped off the stack.

SEE ALSO

c.idiv, c.imod, c.udiv, c.umod

NAME

c.irsh - integer right shift

SYNOPSIS

```
/ integer val on stack
/ integer count on stack
  call c.irsh
/ integer result on stack
```

FUNCTION

c.irsh shifts the integer val right by the integer count. If count is negative, a left shift occurs instead.

RETURNS

The value returned is the shifted integer result val>>count on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.ilsh, c.ursh

NOTES

count is blindly reduced modulo 256; no checking is performed for ridiculously long shifts (16, 128), which take a long time.

NAME

c.itd – convert integer to double

SYNOPSIS

```
/ pointer to left on stack
/ right on stack
  call c.itd
/ pointer to left still on stack
```

FUNCTION

c.itd is the internal routine called by C to convert the integer right into a double at left. It does so by extending the integer 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.itd replaces the operand at left with the double representation of the integer right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dti, c.utd, c.repk

NAME

c.itf - convert integer to float

SYNOPSIS

```
/ pointer to left on stack
/ right on stack
  call c.itf
/ pointer to left still on stack
```

FUNCTION

c.itf is the internal routine called by C to convert the integer right into a float at left. It does so by extending the integer to an unpacked float fraction, then calling c.frepk 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.itf replaces the operand at left with the float representation of the integer right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.fti, c.utf, c.frepk

NAME

c.jltab - perform C switch statement for long value

SYNOPSIS

```
c.r0 = value (long)
hl = &swtab
jmp c.jltab
```

FUNCTION

c.jltab is the code that branches to the appropriate case in a switch statement, when the switch value is a long. 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 integer 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 is the statement following the switch if there is no explicit default statement within the switch.

RETURNS

c.jltab exits to the appropriate case or default; it never returns. The registers af, bc, and hl are not preserved.

NAME

c.jtab - perform C switch statement

SYNOPSIS

```
bc = val
hl = &swtab
jmp c.jtab
```

FUNCTION

c.jtab 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 integer 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 is the statement following the switch if there is no explicit default statement within the switch.

RETURNS

c.jtab exits to the appropriate case or default; it never returns. The registers af, bc, and hl are not preserved.

NAME

c.ladd - add long to long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ladd
/ pointer to left still on stack
```

FUNCTION

c.ladd adds the long at right to the long at left to obtain the long sum. No check is made for overflow, which currently gives the low order 32 bits of the correct sum. The result of c.ladd is also valid for unsigned operands.

RETURNS

The value returned is the long sum left+right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.lsub

NAME

c.land - and long into long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.land
/ pointer to left still on stack
```

FUNCTION

c.land ands the long at right into the long at left to obtain the long logical intersection.

RETURNS

The value returned is the long intersection left&right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.lor, c.lxor

NAME

c.lclt - compare long to long, set NC

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.lclt
/ no pointers left on stack
```

FUNCTION

c.lclt compares the long at right to the long at left to set the N and C flags in f. No check is made for overflow, which currently gives erroneous settings for N if the arguments differ widely. The setting of C is always correct for unsigned operands, however.

RETURNS

c.lclt returns NC set properly in f to reflect (left :: right); Z is not set properly. All registers but a are preserved, and the arguments are popped off the stack.

SEE ALSO

c.lcmp

NAME

c.lcmp - compare long to long, set Z

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.lcmp
/ no pointers left on stack
```

FUNCTION

c.lcmp compares the long at right to the long at left to set the Z flag in f.

RETURNS

c.lcmp returns Z set properly in f to reflect (left :: right); N and C are not set properly. All registers but a are preserved, and the arguments are popped off the stack.

SEE ALSO

c.lclt

NAME

c.lcom – complement long

SYNOPSIS

```
/ pointer to left on stack
  call c.lcom
/ pointer to left still on stack
```

FUNCTION

c.lcom complements the long at left in place.

RETURNS

The value returned is ~left, stored at left. All registers but af are preserved.

SEE ALSO

c.lneg

NAME

c.lcpy - copy long to long

SYNOPSIS

```
/ pointer to left in bc
/ pointer to right hl
  call c.lcpy
```

FUNCTION

c.lcpy moves the long at right to the long at left.

RETURNS

Nothing. None of the volatile registers af, bc, or hl are preserved.

SEE ALSO

c.dcpy

NAME

c.ldiv - divide long by long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ldiv
/ pointer to left still on stack
```

FUNCTION

c.ldiv divides the long at left by the long at right to obtain the long quotient. The sign of a nonzero 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 long quotient of left/right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.lmod, c.uldiv, c.ulmod

NAME

c.libc - perform a far call (bank-switching)

SYNOPSIS

```
/ bc = address of far call descriptor
/ iy = address of save area (return address abd BBR of HD64180)
  call  c.libc
```

FUNCTION

c.libc is the function used to perform a far call, i.e. a call to a function which is located in another bank. c.libc is only avalaible for the **HD64180. bc** contains the address of the far call descriptor; **iy** points to a save area where c.libc will save the return address and the current **MMU** setting. Far calls are supported through use of the iy register; if there is a far call made in a function then space is allocated on the stack (and is pointed by iy) to allow for far call (that is to save the return address and the bank number). When a far call is made it will be redirected to a call to the c.libc routine with the address of a far pointer in bc. c.libc saves the proper information then calls the function really called which returns to c.libc which then restores the proper information to return to caller.

RETURNS

c.libc returns what the called function returns.

NAME

c.llsh - long left shift

SYNOPSIS

```
/ pointer to val on stack
/ integer count on stack
  call c.llsh
/ pointer to val still on stack
```

FUNCTION

c.llsh shifts the long at val left by the integer count. If count is negative, an arithmetic right shift occurs instead. If count is positive, the result is valid for unsigned long val.

RETURNS

The value returned is the shifted long result val<<count stored at val. All registers but af are preserved, and the count argument is popped off the stack.

SEE ALSO

c.lrsh, c.ulrsh

NOTES

count is blindly reduced modulo 256; no checking is performed for ridiculously long shifts (32, 128), which take a long time.

NAME

c.lmod - remainder of long divided by long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.lmod
/ pointer to left still on stack
```

FUNCTION

c.lmod divides the long at left by the long at right to obtain the long remainder. The sign of a nonzero result is the same as the sign of left. No check is made for division by zero, which currently gives a remainder equal to left.

RETURNS

The value returned is the long remainder left%right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ldiv, c.uldiv, c.ulmod

NAME

c.lmul - multiply long by long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.lmul
/ pointer to left still on stack
```

FUNCTION

c.lmul multiplies the long at left by the long at right to obtain the long product. The sign of a nonzero 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. The result of c.lmul is also valid for unsigned operands.

RETURNS

The value returned is the long product left*right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ldiv, c.lmod, c.uldiv, c.ulmod

NAME

c.lneg - negate long

SYNOPSIS

```
/ pointer to left on stack
  call c.lneg
/ pointer to left still on stack
```

FUNCTION

c.lneg negates the long at left in place. No check is made for overflow.

RETURNS

The value returned is -left stored at left. All registers but af are preserved.

SEE ALSO

c.lcom

NAME

c.lor - or long into long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.lor
/ pointer to left still on stack
```

FUNCTION

c.lor ors the long at right into the long at left to obtain the long logical union.

RETURNS

The value returned is the long union left|right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.land, c.lxor

NAME

c.lret - return from runtime function

SYNOPSIS

```
/ stack: bc, hl, pc, right, and left
  jmp c.lret
```

FUNCTION

c.lret is the code sequence used to return from several of the runtime functions. It assumes that the stack is setup as follows:

```
8(sp) left operand
6(sp) right operand
4(sp) return link
2(sp) old hl
0(sp) old bc
```

It is assumed 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 bc and hl restored and just the result left on the stack. de is preserved, but af is undefined.

SEE ALSO

c.zret

NAME

c.lrsh – long right shift

SYNOPSIS

```
/ pointer to val on stack
/ integer count on stack
  call c.lrsh
/ pointer to val still on stack
```

FUNCTION

c.lrsh shifts the long at val right by the integer count. If count is negative, a left shift occurs instead.

RETURNS

The value returned is the shifted long result val>>count stored at val. All registers but af are preserved, and the count argument is popped off the stack.

SEE ALSO

c.llsh, c.ulrsh

NOTES

count is blindly reduced modulo 256; no checking is performed for ridiculously long shifts (32, 128), which take a long time.

NAME

c.lsub - subtract long from long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.lsub
/ pointer to left still on stack
```

FUNCTION

c.lsub subtracts the long at right from the long at left to obtain the long difference. No check is made for overflow, which currently gives the low order 32 bits of the correct sum. The result of c.lsub is also valid for unsigned operands.

RETURNS

The value returned is the long difference left-right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ladd

NAME

c.ltd - convert long to double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ltd
/ pointer to left still on stack
```

FUNCTION

c.ltd is the internal routine called by C to convert the long at right 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 like an ordinary machine instruction.

RETURNS

c.ltd replaces the operand at left with the double representation of the long at right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dtl, c.repk, c.ultd

NAME

c.ltf - convert long to float

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ltf
/ pointer to left still on stack
```

FUNCTION

c.ltf is the internal routine called by C to convert the long at right into a float at left. It does so by extending the long to an unpacked float fraction, then calling c.frepk 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.ltf replaces the operand at left with the float representation of the long at right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ftl, c.frepk, c.ultf

NAME

c.lxor - exclusive or long into long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.lxor
/ pointer to left still on stack
```

FUNCTION

c.lxor exclusive ors the long at right into the long at left to obtain the long logical symmetric difference.

RETURNS

The value returned is the long symmetric difference left^right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.land, c.lor

NAME

c.movestr – copy a structure to another

SYNOPSIS

```
/ pointer to left in hl
/ pointer to right on stack
/ bc = size (of structure)
  call c.movestr
```

FUNCTION

c.movestr copies the structure at right into the structure pointed by **hl.** The length in bytes is stored in **bc.**

RETURNS

Nothing.

NAME

c.pushstr - push a structure

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
/ bc = size (of structure)
  call c.pushstr
```

FUNCTION

c.pushstr copies the structure at right into the structure at left. The length in bytes is stored in **bc.**

RETURNS

Nothing.

NAME

c.0mvd – copy double to in-core register c.r0

SYNOPSIS

```
/  hl = address of right
   call c.0mvd
```

FUNCTION

c.0mvd moves the double (eight bytes of data) whose address is in **hl** to the in-core pseudo register **c.r0.** This routine trashes registers **bc** and **hl.**

RETURNS

Nothing.

NAME

c.1mvd - copy double to in-core register c.r1

SYNOPSIS

```
/  hl = address of right
   call c.1mvd
```

FUNCTION

c.0mvd moves the double (eight bytes of data) whose address is in **hl** to the in-core pseudo registert **c.r1.** This routine trashes registers **bc** and **hl.**

RETURNS

Nothing.

NAME

c.0mvf - copy float to in-core register c.r0

SYNOPSIS

```
/  hl = address of right
   call c.0mvf
```

FUNCTION

c.0mvf moves the float (four bytes of data) whose address is in **hl** to the in-core pseudo register **c.r0.** This routine trashes registers **bc** and **hl.**

RETURNS

Nothing.

NAME

c.1mvf - copy float to in-core register c.r1

SYNOPSIS

```
/  hl = address of right
   call c.1mvf
```

FUNCTION

c.0mvf moves the float (four bytes of data) whose address is in **hl** to the in-core pseudo register **c.r1.** This routine trashes registers **bc** and **hl.**

RETURNS

Nothing.

NAME

c.r0 - the double accumulator and other pseudo registers

SYNOPSIS

```
  . := .data
 c.r0: 0; 0; 0; 0; 0; 0; 0; 0
 c.r1: 0; 0; 0; 0; 0; 0; 0; 0
 c.r2: 0; 0
 c.r3: 0; 0
```

FUNCTION

c.r0 is an eight-byte static area used for returning long and double results from C functions. It is accompanied by c.r1, another eight-byte area, and two two-byte registers c.r2 and c.r3. c.r0 and c.r1 are considered volatile, and hence may be used freely by any function; c.r2 and c.r3 must be preserved. The function entry and exit utilities c.sav, c.sav0, c.ret and c.ret0 are used to save and restore the nonvolatile pseudo registers.

The C compiler allocates up to three non-volatile registers to honor register declarations, and uses register de, c.r0 and c.r1 as long or double arithmetic accumulators.

RETURNS

c.r0 doesn't return anything; it just stands there. Doubles fill all eight-bytes, in the usual format; longs are packed into the first four bytes, also in the usual format for longs in memory.

SEE ALSO

c.sav, c.sav0, c.ret, c.ret0

NAME

c.repk - repack a double number

SYNOPSIS

```
/ characteristic on stack
/ pointer to frac on stack
  call c.repk
  sp => af => af
```

FUNCTION

c.repk is the internal routine called by various floating run-time routines to pack a signed fraction at frac and a two-byte binary characteristic into a standard form double representation. The fraction occupies nine bytes, starting at frac and stored least significant byte first, 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 to bring the fraction into the interval [0.5, 1.0), 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 first eight (least significant) bytes of the fraction with the double representation, i.e., four two-byte integers, most sig- nificant integer first. The value of the function is VOID, i.e., garbage. The registers af, bc, and hl are not preserved.

SEE ALSO

c.unpk

NOTES

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 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 ix is used 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 de, c.r2, c.r3 and leaves bc unchanged, so as not to disturb a returned value. af and hl are not preserved.

SEE ALSO

c.sav

NAME

c.ret0 - return from a C function

SYNOPSIS

```
jmp c.ret0
```

FUNCTION

c.ret0 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.sav0. The stack frame pointer ix is used 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.ret0 restores de, c.r2, c.r3 and leaves bc unchanged, so as not to disturb a returned value. af and hl are not preserved.

SEE ALSO

c.sav0

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.savs. The stack frame pointer ix is used 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 all non-volatile registers and leaves unchanged bc, c.r0, and c.r1, so as not to disturb a returned value. af and hl are not preserved.

SEE ALSO

c.savs

NAME

c.rets0 - return from a C function

SYNOPSIS

```
jmp c.rets0
```

FUNCTION

c.rets0 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.savs0. The stack frame pointer ix is used 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.rets0 restores all non-volatile registers and leaves unchanged bc, c.r0, and c.r1, so as not to disturb a returned value. af and hl are not preserved.

SEE ALSO

c.savs0

NAME

c.sav - enter a C function with one or more arguments and save registers

SYNOPSIS

```
call c.sav
```

FUNCTION

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

```
hl     holds the low 16 bits of the first argument the first
         argument
0(sp) holds the return link
```

On return fom c.sav ix matches sp and:

```
4(ix) holds first argument
2(ix) holds return link
0(ix) holds old ix
-2(ix) holds old de
-4(ix) holds old c.r3
-6(ix) holds old c.r2
```

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at -7(ix) on down.

RETURNS

c.sav alters sp and ix to make the new stack frame. af, bc, and hl are not preserved.

SEE ALSO

c.savs

NAME

c.sav0 - enter a C function with no arguments and save registers

SYNOPSIS

call c.sav0

FUNCTION

c.sav0 sets up a new stack frame and stacks de, c.r2 and c.r3. It is designed to be called on entry to a C function, at which time:

0(sp) holds the return link

On return fom c.sav ix matches sp and:

2(ix) holds return link
0(ix) holds old ix
-2(ix) holds old de
-4(ix) holds old c.r3
-6(ix) holds old c.r2

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at -7(ix) on down.

RETURNS

c.sav0 alters sp and ix to make the new stack frame. af, bc, and hl are not preserved.

SEE ALSO

c.savs0

NAME

c.savs - enter a C function with one or more arguments

SYNOPSIS

```
call c.savs
```

FUNCTION

c.savs sets up a new stack frame. It is designed to be called on entry to a C function, at which time:

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

On return fom c.ents ix holds sp+6 and:

```
4(ix) holds first argument
2(ix) holds return link
0(ix) holds old de
```

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at -7(ix) on down.

RETURNS

c.savs alters sp and ix to make the new stack frame. af, bc, and hl are not preserved.

SEE ALSO

c.rets

NAME

c.savs0 - enter a C function with no arguments

SYNOPSIS

```
call c.savs0
```

FUNCTION

c.savs0 sets up a new stack frame. It is designed to be called on entry to a C function, at which time:

0(sp) holds the return link

On return fom c.ents ix holds sp+6 and:

2(ix) holds return link
0(ix) holds old de

Automatic storage can be allocated by decrementing the stack pointer; it is addressed at -7(ix) on down.

RETURNS

c.savs0 alters sp and ix to make the new stack frame. af, bc, and hl are not preserved.

SEE ALSO

c.rets0

NAME

c.udiv - divide unsigned by unsigned

SYNOPSIS

```
/ left on stack
/ right on stack
  call c.udiv
/ quotient on stack
```

FUNCTION

c.udiv divides the unsigned left by the unsigned right to obtain the unsigned quotient. No check is made for division by zero, which currently gives a quotient of all ones.

RETURNS

The value returned is the unsigned quotient of left/right on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.imod, c.idiv, c.umod

NAME

c.uldiv - unsigned divide long by long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.uldiv
/ pointer to left still on stack
```

FUNCTION

c.uldiv divides the unsigned long at left by the unsigned long at right to obtain the unsigned long quotient. No check is made for division by zero, which currently gives a quotient of all ones.

RETURNS

The value returned is the unsigned long quotient of left/right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.lmod, c.ldiv, c.ulmod

NAME

c.ulmod - remainder of unsigned long divided by long

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ulmod
/ pointer to left still on stack
```

FUNCTION

c.ulmod divides the unsigned long at left by the unsigned long at right to obtain the unsigned long remainder. No check is made for division by zero, which currently gives a remainder equal to left.

RETURNS

The value returned is the unsigned long remainder left%right stored at left. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ldiv, c.lmod, c.uldiv

NAME

c.ulrsh - unsigned long right shift

SYNOPSIS

```
/ pointer to val on stack
/ integer count on stack
  call c.ulrsh
/ pointer to val still on stack
```

FUNCTION

c.ulrsh shifts the unsigned long at val right by the integer count. If count is negative, a left shift occurs instead.

RETURNS

The value returned is the shifted unsigned long result val>>count stored at val. All registers but af are preserved, and the count argument is popped off the stack.

SEE ALSO

c.llsh, c.lrsh

NOTES

count is blindly reduced modulo 256; no checking is performed for ridiculously long shifts (32, 128), which take a long time.

NAME

c.ultd - convert unsigned long to double

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ultd
/ pointer to left still on stack
```

FUNCTION

c.ultd is the internal routine called by C to convert the unsigned long at right 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 at right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dtl, c.ltd, c.repk

NAME

c.ultf - convert unsigned long to float

SYNOPSIS

```
/ pointer to left on stack
/ pointer to right on stack
  call c.ultf
/ pointer to left still on stack
```

FUNCTION

c.ultf is the internal routine called by C to convert the unsigned long at right into a float at left. It does so by extending the unsigned long to an unpacked float fraction, then calling c.frepk 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.ultf replaces the operand at left with the float representation of the unsigned long at right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.ftl, c.ltf, c.frepk

NAME

c.umod - remainder of unsigned divided by unsigned

SYNOPSIS

```
/ left on stack
/ right on stack
  call c.umod
/ remainder on stack
```

FUNCTION

c.umod divides the unsigned left by the unsigned right to obtain the unsigned remainder. No check is made for division by zero, which currently gives a remainder equal to left.

RETURNS

The value returned is the unsigned remainder left%right on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.idiv, c.imod, c.udiv

NAME

c.unpk – unpack a double number

SYNOPSIS

```
/ pointer to double on stack
/ pointer to frac on stack
  call c.unpk
  sp => af => af
```

FUNCTION

c.unpk is the internal routine called by various floating run-time routines to unpack a double at double into a signed fraction at frac and a characteristic. The fraction consists of nine bytes at frac, stored least significant byte first; 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 magnitude of the fraction is in the interval [0.5, 1.0). The least significant byte is guaranteed to be zero; it serves as a guard byte.

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 nine bytes starting at frac and stored least significant byte first, and returns the characteristic in bc as the value of the function. The registers af and hl are not preserved.

SEE ALSO

c.repk

NAME

c.ursh - unsigned right shift

SYNOPSIS

```
/ unsigned val on stack
/ integer count on stack
  call c.ursh
/ unsigned result on stack
```

FUNCTION

c.ursh shifts the unsigned val right by the integer count. If count is negative, a left shift occurs instead.

RETURNS

The value returned is the shifted unsigned result val>>count on the stack. All registers but af are preserved, and the arguments are popped off the stack.

SEE ALSO

c.ilsh, c.irsh

NOTES

count is blindly reduced modulo 256; no checking is performed for ridiculously long shifts (16, 128), which take a long time.

NAME

c.utd - convert unsigned to double

SYNOPSIS

```
/ pointer to left on stack
/ right on stack
  call c.utd
/ pointer to left still on stack
```

FUNCTION

c.utd is the internal routine called by C to convert the unsigned right into a double at left. It does so by extending the unsigned 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.utd replaces the operand at left with the double representation of the unsigned right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.dti, c.itd, c.repk

NAME

c.utf - convert unsigned to float

SYNOPSIS

```
/ pointer to left on stack
/ right on stack
  call c.utf
/ pointer to left still on stack
```

FUNCTION

c.utf is the internal routine called by C to convert the unsigned right into a float at left. It does so by extending the unsigned to an unpacked float fraction, then calling c.frepk 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.utf replaces the operand at left with the float representation of the unsigned right. All registers but af are preserved, and the right argument is popped off the stack.

SEE ALSO

c.fti, c.itf, c.frepk

NAME

c.utob - pack unsigned into bits

SYNOPSIS

```
/ pointer to bits on stack
/ unsigned on stack
/ offset/size on stack
  call c.utob
/ pointer to bits still on stack
```

FUNCTION

c.utob is the internal routine called by C to pack unsigned into the bitfield at bits. The field is specified by the two bytes offset/size, where the less significant byte offset is the number of places the bitfield must be shifted right to align it as an integer, and the more significant byte size is the number of bits in the field. offset is assumed to be in the range [0, 16), while size is in the range (0,16].

RETURNS

c.utob inserts the unsigned into the specified bitfield at bits. All registers but af are preserved, and all arguments but the pointer to bits are popped off the stack.

SEE ALSO

c.btou

NAME

c.zret - return from runtime compare function

SYNOPSIS

```
/ stack: bc, hl, pc, right, and left
  jmp c.zret
```

FUNCTION

c.zret is the code sequence used to return from several of the runtime compare functions. It assumes that the stack is setup as follows:

```
8(sp) left operand
6(sp) right operand
4(sp) return link
2(sp) old hl
0(sp) old bc
```

It is assumed that f is set to reflect the comparison, and so must be preserved during the stack cleanup.

RETURNS

c.zret returns with the old bc and hl restored, both operands popped off the stack, and f unchanged from the jmp to c.zret. de is preserved, but the a register is undefined.

SEE ALSO

c.lret