IV. Machine Support Library for 8080

IV - 49	c.switch	perform C switch statement
IV - 50	c.udiv	divide unsigned by unsigned
IV - 51	c.uldiv	unsigned divide long by long
IV - 52	c.ulmod	remainder of unsigned long divided by long
IV - 53	c.ulrsh	unsigned long right shift
IV - 54	c.ultd	convert unsigned long to double
IV - 55	c.umod	remainder of unsigned divided by unsigned
IV - 56	c.unpk	unpack a double number
IV - 57	c.ursh	unsigned right shift
IV - 58	c.utd	convert unsigned to double
IV - 59	c.utob	pack unsigned into bits
IV - 60	c.zret	return from runtime compare function
IV - 61	in	input from port
IV - 62	out	output to port

NAME

Conventions - using the 8080 Machine Support Library

FUNCTION

The 8080 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 8080. To use it, however, requires at least a basic knowledge of how C does business. All the information you need to make use of the facilities in this section has been provided earlier; it is summarized here to assist the A-Natural programmer.

Functions are described in terms of their A-Natural interface, since they operate outside the conventional C calling protocol. Unless explicitly stated otherwise, 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. Never passed between functions. A char may also be unsigned.

short - or two-byte integer, also known simply as int or integer. Stored less significant byte first, as the 8080 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. Never passed between functions.

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 exponent 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 A-Natural identifiers, the compiler prepends an underscore '_' to each symbol. Thus, the function name "func" becomes "_func" in A-Natural.

It is also important to understand the rules for calling C functions, and for being called:

A function is called by first pushing its arguments onto the stack in reverse order, so that the location of the first argument is not a function of how many arguments are actually passed. char values are widened to int, float to double. Then the function is called via:

```
call _func
```

It is the responsibility of the calling function to pop the arguments off the stack; it is acceptable for the called function to modify the arguments, since a fresh copy is expected on each call.

A C function may return one of the data types listed above. If the return value is char, it is widened to int and placed in bc; int and unsigned also are returned in bc. long values are returned in the first four bytes of the static area labelled c.r0. float values are widened to double, which is returned in the eight bytes of the static area labelled c.r0.

A called function may otherwise clobber af, bc, hl, and the eight-byte area labelled c.r1; de is used by C as a stack frame pointer and must be carefully preserved, as must the state of the stack as described above. There are also three two-byte static areas labelled c.r2, c.r3, and c.r4, which C frequently uses; these must be meticulously saved and restored.

See the manual pages for c.ent, c.ents, c.r0, c.ret, and c.rets for some assistance in the use of this calling sequence.

SEE ALSO

Techniques(I), for coding tips

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.count - counter for profiler

SYNOPSIS

```
. := .data
L: &0
. := .text
    hl = &L
    call c.count
```

FUNCTION

c.count is the function called on entry to each C function when code is compiled using the profiling option. hl points at a data word, initially zero, which is used by c.count to record where counts for that entry point are being maintained; hence, there should be a unique data word reserved for each separate call on c.count.

RETURNS

Nothing. af, bc, and hl are not preserved.

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

BUGS

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

BUGS

(-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.ent - enter a C function

SYNOPSIS

```
call c.ent
```

FUNCTION

c.ent sets up a new stack frame. 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.ent de matches sp and:

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

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

RETURNS

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

EXAMPLE

The C function:

```
COUNT idiot()
    {
    COUNT i;

    return (i);
    }
```

can be written:

```
idiot:
    call c.ent
    sp <= af <= af <= af <= af  / space for i
    bc =^ (hl = -8 + de)        / return value
    jmp c.ret
```

SEE ALSO

c.ents, c.ret, c.rets

NAME

c.ents - save registers on entering a C function

SYNOPSIS

```
call c.ents
```

FUNCTION

c.ents sets up a new stack frame and stacks c.r4, c.r3, and c.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.ents de holds sp+6 and:

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

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

RETURNS

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

EXAMPLE

The C function:

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

    return (i);
    }
```

can be written:

```
idiot:
    call c.ents
    bc =^ (hl = c.r4)    / return value
    jmp c.rets
```

SEE ALSO

c.ent, c.r0, c.ret, c.rets

NAME

c.entx - save registers and check stack on entering a C function

SYNOPSIS

```
hl = &nautos
call c.entx
```

FUNCTION

c.entx sets up a new stack frame, stacks c.r4, c.r3, and c.r2, and ensures that nautos+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 fom c.entx de holds sp+6 and:

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

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

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

RETURNS

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

EXAMPLE

The C function:

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

    return (i);
    }
```

can be written:

```
idiot:
    hl = &-24
    call c.entx
    bc =^ (hl = c.r4)   / return value
    jmp c.rets
```

SEE ALSO

c.ent, c.ents, c.r0, c.ret, c.rets

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.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 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

BUGS

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

BUGS

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.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.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

BUGS

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

BUGS

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.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.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
c.r4:   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 three two-byte registers c.r2, c.r3, and c.r4. c.r0 and c.r1 are considered volatile, and hence may be used freely by any function; c.r2, c.r3, and c.r4 must be preserved. The function entry and exit utilities c.ents and c.rets are used to save and restore the nonvolatile pseudo registers.

Since the 8080 is short on registers, and since it is better at directly addressing memory than addressing relative to a base register, the use of the pseudo registers can significantly reduce the size of a function. The C compiler allocates up to three non-volatile registers to honor register declarations, and uses 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.ents, c.rets

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 runtime 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 significant 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

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 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.ent. The stack frame pointer de 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 and leaves bc unchanged, so as not to disturb a returned value. af and hl are not preserved.

EXAMPLE
The C function:

```
COUNT idiot()
    {
    COUNT i;

    return (i);
    }
```

can be written:

```
idiot:
    call c.ent
    sp <= af <= af <= af <=af   / space for i
    bc =^ (hl = -8 + de)        / return value
    jmp c.ret
```

SEE ALSO
c.ent, c.ents, c.rets

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.ents. The stack frame pointer de is used to locate the stored c.r2, c.r3, c.r4, and de 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.

EXAMPLE
The C function:

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

    return (i);
    }
```

can be written:

```
idiot:
    call c.ents
    bc = (hl = c.r0)     / return value
    jmp c.rets
```

SEE ALSO
c.ent, c.ents, c.r0, c.ret

NAME

c.switch - perform C switch statement

SYNOPSIS

```
bc = val
hl = &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 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.switch exits to the appropriate case or default; it never returns. The registers af, bc, and hl are not preserved.

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

BUGS

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.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 runtime 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

BUGS

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.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

NAME

in - input from port

SYNOPSIS

```
COUNT in(port)
    TEXT port;
```

FUNCTION

in is a C callable function to input a byte from an arbitrary port, which is specified by the less significant byte of port. To avoid modifying pure program text, in builds an instruction sequence on the stack and calls it to input the byte.

RETURNS

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

SEE ALSO

out

NAME

out - output to port

SYNOPSIS

```
VOID out(port, out)
    TEXT port, out;
```

FUNCTION

out is a C callable function to output a byte to an arbitrary port, which is specified by the less significant byte of port. The data to be output is the less significant byte of out. To avoid modifying pure program text, out builds an instruction sequence on the stack and calls it to output the byte.

RETURNS

Nothing.

SEE ALSO

in

Whitesmiths, Ltd.
97 Lowell Road, Concord, Massachusetts 01742
Telephone (617) 369-8499 Telex 951708 SOFTWARE CNCM