blitz -h

```
=======================================
=====                           =====
=====  The BLITZ Machine Emulator  =====
=====                           =====
=======================================

Copyright 2001, Harry H. Porter III
===================================
  Original Author:
    02/05/01 - Harry H. Porter III

Command Line Options
====================
  These command line options may be given in any order.
    filename
       The input executable file.  If missing, "a.out" will be used.
    -h
       Print this help info.  Ignore other options and exit.
    -d filename
       Disk file name.  If missing, "DISK" will be used.
    -g
       Automatically begin emulation of the a.out program, bypassing
       the command line interface.
    -i filename
       Terminal input file name.  If missing, "stdin" will be used.
    -o filename
       Terminal output file name.  If missing, "stdout" will be used.
    -r integer
       Set the random seed to the given integer, which must be > 0.
    -raw
       User input for BLITZ terminal I/O will be in "raw" mode; the
       default is "cooked", in which case the running BLITZ code
       is relieved from echoing keystrokes, processing backspaces, etc.
    -wait
       This option applies only when input is coming from an interactive
       terminal and a 'wait' instruction is executed with no other pending
       interrupts.  Without this option, execution will halt; with it the
       emulator will wait for input.
```

kpl -h

```
==============================
=====                    =====
=====  The KPL Compiler  =====
=====                    =====
==============================

Copyright 2002, Harry H. Porter III
===================================
  Original Author:
    06/15/02 - Harry H. Porter III

Command Line Options
====================
  Command line options may be given in any order.
    -h
      Print this help info.  All other options are ignored.
    packageName
      Compile the package with this name.  The input will come from the files
      called "packageName.h" and "packageName.c".  No extension should be
      given on the command line.  Only one package may be compiled at once.
      The packageName is required.
    -d directoryPrefix
      When looking for header and code files, the default is to look in the
      current directory.  With this option, the current directory is first
      searched.  If that fails, then the directoryPrefix is prepended to the
      file name and the resulting file name is used.  For example:
          kpl myPack -d ~harry/BlitzLib/
      will first try to open "myPack.h" and, if that fails, will try to open
      "~harry/BlitzLib/myPack.h".
    -unsafe
      Allow unsafe language constructs.
    -o filename
      If there are no errors, an assembly code file will be created.  This
      option can be used to give the output file a specific name.  If
      missing, the name of the output file will be computed from the name of
      the package and appending ".s".  For example:
           myPackage  -->  myPackage.s
      COMPILER DEBUGGING: If packageName and output filename are missing,
      stdout will be used.
    -testLexer
      COMPILER DEBUGGING: Scan tokens only, and print tokens out.  Input may
      come from stdin.
    -testParser
      COMPILER DEBUGGING: Parse program only, and print data structures out.
      Input may come from stdin.
    -s
      COMPILER DEBUGGING: Print the symbol table on stdout.
    -p
      COMPILER DEBUGGING: Pretty-print the AST.
    -ast
      COMPILER DEBUGGING: Dump the full AST.
```

asm -h

```
================================
=====                      =====
=====  The BLITZ Assembler  =====
=====                      =====
================================

Copyright 2000, Harry H. Porter III
================================
  Original Author:
    11/12/00 - Harry H. Porter III

Command Line Options
====================
  Command line options may be given in any order.
    filename
      The input source will come from this file.  (Normally this file
      will end with ".s".)  If an input file is not given on the command
      line, the source must come from stdin.  Only one input source is allowed.
    -h
      Print this help info.  All other options are ignored.
    -l
      Print a listing on stdout.
    -s
      Print the symbol table on stdout.
    -d
      Print internal assembler info (for debugging asm.c)
    -o filename
      If there are no errors, an object file will be created.  This
      option can be used to give the object file a specific name.
      If this option is not used, then the input .s file must be named on
      the command line (i.e., the source must not come from stdin.)  In this
      case, the name of the object file will be computed from the name of
      the input file by removing the ".s" extension, if any, and appending
      ".o".  For example:
           test.s  -->  test.o
           foo     -->  foo.o

Lexical issues:
===============
  Identifiers - May contain letters, digits, and underscores.  They must
    begin with a letter or underscore.  Case is significant.  Identifiers
    are limited in length to 200 characters.
  Integers - May be specified in decimal or in hex.
    Integers must range from 0 to 2147483647.  Hex notation is, for
    example, 0x1234abcd.  0x1234ABCD is equivalent.  Shorter numbers like
    0xFFFF are not sign-extended.
  Strings - Use double quotes.  The following escape sequences are allowed:
      \0  \a  \b  \t  \n  \v  \f  \r  \"  \'  \\  \xHH
    where HH are any two hex digits.  Strings may not contain newlines directly;
    in other words, a string  may not span multiple lines.  The source file may
    not contain unprintable ASCII characters; use the escape sequences if you
    wish to include unprintable characters in string or character constants.
    String constants are limited in length to 200 characters.
  Characters - Use single quotes.  The same escape sequences are allowed.
  Comments - Begin with the exclamation mark (!) and extend thru end-of-line.
  Punctuation symbols - The following symbols have special meaning:
      ,  [  ]  :  .  +  ++  -  --  *  /  <<  >>  >>>  &  |  ^  ~  (  )  =
  Keywords - The following classes of keywords are recognized:
    BLITZ instruction op-codes (e.g., add, sub, syscall, ...)
    Synthetic instructions (e.g., mov, set, ...)
    Assembler pseudo-ops (e.g., .text, .import, .byte, ...)
```

```
    Registers (r0, r1, ... r15)
  White space - Tabs and space characters may be used between tokens.
  End-of-line - The EOL (newline) character is treated as a token, not
    as white space; the EOL is significant in syntax parsing.

Assembler pseudo-ops
====================
  .text    The following instructions and data will be placed in the
           "text" segment, which will be read-only during execution.
  .data    The following instructions and data will be placed in the
           "data" segment, which will be read-write during execution.
  .bss     The following bytes will be reserved in the "bss" segment,
           which will be initialized to zero at program load time.
  .ascii   This operand expects a single string operand.  These bytes
           will be loaded into memory.  Note that no terminating NULL
           ('\0') character will be added to the end of the string.
  .byte    This pseudo-op expects a single expression as an operand.
           This expression will be evaluated at assembly time, the value
           will be truncated to 8 bits, and the result used to initialize
           a single byte of memory.
  .word    This pseudo-op expects a single expression as an operand.
           This expression will be evaluated at assembly time to a
           32 bit value, and the result used to initialize four bytes
           of memory.  The assembler does not require alignment for .word.
  .export  This pseudo-op expects a single symbol as an operand.  This
           symbol must be given a value in this file.  This symbol with
           its value will be placed in the object file and made available
           during segment linking.
  .import  This pseudo-op expects a single symbol as an operand.  This
           symbol must not be given a value in this file; instead it will
           receive its value from another .s file during segment linking.
           All uses of this symbol in this file will be replaced by that
           value at segment-link time.
  .skip    This pseudo-op expects a single expression as an operand.
           This expression must evaluate to an absolute value.  The
           indicated number of bytes will be skipped in the current
           segment.
  .align   This instruction will insert 0, 1, 2, or 3 bytes into the
           current segment as necessary to bring the location up to an
           even multiple of 4.  No operand is used with .align.
  =        Symbols may be given values with a line of the following
           format:
                       symbol   =   expression
           These are called "equates".  Equates will be processed
           during the first pass, if possible.  If not, they will be
           processed after the program has been completely read in.
           The expression may use symbols that are defined later in the
           file, but this may cause the equate to be given a value
           slightly later in the assembly.  After the first pass, an
           attempt will be made to evaluate all the equates.  At this
           time, errors may be generated.  After the equates have been
           processed, the machine code can be generated in the final
           pass.

Segments
========
  This assembler is capable of assembling BLITZ instructions and data
  and placing them in one of three "segments":
    .text
    .data
    .bss

  At run-time, the bytes placed in the .text segment will be read-only.
  At run-time, the bytes places in the .data segment will be read-write.
  At run-time, the bytes places in the .bss segment will be read-write.
  The read-only nature of the bytes in the .text segment may or may not
  be enforced by the operating system at run-time.

  Instructions and data may be placed in either the .text or .data
```

```
  segment.  No instructions or data may be placed in the .bss segment.
  The only things that may follow the .bss pseudo-op are the following
  pseudo-ops:
    .skip
    .align
  The assembler may reserve bytes in the .bss segment but no initial
  values may be placed in these locations.  Instead, all bytes of the
  .bss segment will be initialized to zeros at program-load time.  These
  addresses may be initialized and modified during program execution.

  Segment control is done using the following pseudo-ops:
    .text
    .data
    .bss

  After any one of these pseudo-ops, all following instructions and data
  will be placed in the named segment.  A "location counter" for each of
  the three segments is maintained by the assembler.  If, for example, a
  .text pseudo-op has been used to switch to the ".text" segment, then
  all subsequent instructions will be placed in the ".text" segment.
  Any labels encountered will be be given values relative to the
  ".text" segment.  As each instruction is encountered, the location
  counter for the ".text" segment will be incremented.  If a .data
  pseudo-op is the encountered, all subsequent instructions will be placed
  in the ".data" segment.  The location counters are not reset;  if a
  .text pseudo-op is again encountered, subsequent instructions will be
  placed in the ".text" segment following the instructions encountered
  earlier, before the .data pseudo-op was seen.  Thus, we can "pick up"
  in the .text segment where we left off.

Symbols
=======
  The assembler builds a symbol table, mapping identifiers to values.
  Each symbol is given exactly one value: there is no notion of scope
  or lexical nesting levels, as in high-level languages.  Each symbol
  is given a value which will be either:
    absolute
    relative
    external
  An absolute value consists of a 32-bit quantity.  A relative value
  consists of a 32-bit (signed) offset relative to either a segment
  or to an external symbol.  An external symbol will have its value
  assigned in some other assembly file and its value will not be
  available to the code in this file until segment-linking time.  However,
  an external symbol may be used in expressions within this file; the
  actual data will not be filled in until segment-linking time.

  Symbols may be defined internally or externally.  If a symbol is used
  in this file, but not defined, then it must be "imported" using
  the .import pseudo-op.  If a symbol is defined in this file and used
  in other files, then it must be "exported" using an .export
  pseudo-op.  If a symbol is not exported, then its value will not be
  known to the linker; if this same symbol is imported in other files,
  then an "undefined symbol" error will be generated at segment-linking
  time.

  Symbols may be defined in either of two ways:
    labels
    = equates
  If a symbol is defined by being used as a label, then it is given a
  value which consists of an offset relative to the beginning of whichever
  segment is current when the label is encountered.  This is determined by
  whether a .text, .data, or .bss pseudo-op was seen last, before the label
  was encountered.  Each label occurs in a segment and names a location in
  memory.  At segment-link time, the segments are placed in their final
  positions in memory.  Only at segment-link time does the actual address of
  the location in memory become known.  At this time, the label is assigned
  an absolute value.
```

```
Expression Evaluation
=====================
  Instructions and pseudo-ops may contain expressions in their operands.
  Expressions have the form given by the following Context-Free Grammar.
  (In this grammar, the following meta-notation is used: characters
  enclosed in double quotes are terminals.  The braces { } are used to
  mean "zero or more" occurences.  The vertical bar | is used to mean
  alternation.  Parentheses are used for grouping.  The start symbol
  is "expr".)
    expr  ::= expr1 { "|" expr1 }
    expr1 ::= expr2 { "^" expr2 }
    expr2 ::= expr3 { "&" expr3 }
    expr3 ::= expr4 { ( "<<" | ">>" | ">>>" ) expr4 }
    expr4 ::= expr5 { ( "+" | "-" ) expr5 }
    expr5 ::= expr6 { ( "*" | "/" | "%" ) expr6 }
    expr6 ::= "+" expr6 | "-" expr6 | "~" expr6
              | ID | INTEGER | STRING | "(" expr ")"

  This syntax results in the following precedences and associativities:
    highest:    unary+  unary-  ~    (right associative)
                *  /  %             (left associative)
                +  -                (left associative)
                <<  >>  >>>         (left associative)
                &                   (left associative)
                ^                   (left associative)
    lowest:     |                   (left associative)

  If a string is used in an expression, it must have exactly 4 characters.
  The string will be interpreted as a 32 bit integer, based on the ASCII
  values of the 4 characters.  ("Big Endian" order is used: the first
  character will determine the most significant byte.)

  The following operators are recognized in expressions:
    unary+    nop
    unary-    32-bit signed arithmetic negation
    ~         32-bit logical negation (NOT)
    *         32-bit multiplication
    /         32-bit integer division with 32-bit integer result
    %         32-bit modulo, with 32-bit result
    binary+   32-bit signed addition
    binary-   32-bit signed subtraction
    <<        left shift logical (i.e., zeros shifted in from right)
    >>        right shift logical (i.e., zeros shifted in from left)
    >>>       right shift arithmetic (i.e., sign bit shifted in on left)
    &         32-bit logical AND
    ^         32-bit logical Exclusive-OR
    |         32-bit logical OR

  With the shift operators (<<, >>, and >>>) the second operand must
  evaluate to an integer between 0 and 31.  With the division operators
  (/ and %), the first operand must be non-negative and the second
  operand must be positive, since these operators are implemented with
  "C" operators, which are machine-dependent with negative operands.

  All operators except addition and subtraction require both operands to
  evaluate to absolute values.  All arithmetic is done with signed 32-bit
  values.  The addition operator + requires that at least one of the operands
  evaluates to an absolute value.  If one operand is relative, then the
  result will be relative to the same location.  The subtraction operator
  requires that the second operand evaluates to an absolute value.  If the
  first operand is relative, then the result will be relative to the same
  location.  Only absolute values can be negated.

  All expressions are evaluated at assembly-time.  An expression may
  evaluate to either an absolute 32-bit value, or may evaluate to a
  relative value.  A relative value is a 32-bit offset relative to some
  some symbol.  The offset will  be relative to the beginning of the .text
  segment, the .data segment, or the .bss segment, or the offset will be
  relative to some external symbol.  If the expression evaluates to a
```

```
  relative value, its value will not be determined until segment-link
  time.  At this time, the absolute locations of the .text, .data, and
  .bss segments will be determined and the absolute values of external
  symbols will be determined.  At segment-link time, the final, absolute
  values of all expressions will be determined by adding the values of the
  symbols (or locations of the segments) to the offsets.

  Expressions may be used in:
    .byte
    .word
    .skip
    =
    various BLITZ instructions
  The .skip pseudo-op requires the expression evaluates to an absolute value.
  In the case of an = (equate) pseudo-op, the expression may evaluate to
  either a relative or absolute value.  In either case, the equated symbol
  will be given a relative or absolute value (respectively).  At segment-
  linking time, when the actual value is determined, the value will be
  filled in in the byte, word, or appropriate field in the instruction.

Instruction Syntax
==================
  Each line in the assembly source file has the following general syntax:

    [ label: ]   [ opcode   operands ]    [ "!" comment ]   EOL

  The label is optional.  It need not begin in column one.  It must be
  followed by a colon token.  A label may be on a line by itself.  If
  so, it will be given an offset from the current value of the location
  counter, relative to the current segment.

  The opcode must be a legal BLITZ instruction.  The opcode is given in
  lowercase.  The exact format of the operands depends on the instruction;
  some BLITZ instructions take no operands while some require several
  operands. Operands are separated by commas.

  A comment is optional and extends to the end of the line if present.

  Each line is independent.  End-of-line (EOL) is a separate token.  An
  instruction must be on only one line, although lines may be arbitrarily long.

  Assembler pseudo-ops have the same general syntax.  Some permit labels
  and others forbid labels.

  The following formatting and spacing conventions are recommended:
    Labels should begin in column 1.
    The op-code should be indented by 1 tab stop.
    The operands, if any, should be indented by 1 additional tab stop.
    Each BLITZ instruction should be commented.
    The comment should be indented by 2 additional tab stops.
    A single space should follow the ! comment character.
    Block comments should occur before each routine.
    Comments should be indented with 2 spaces to show logical organization.

  Here is an example of the recommended style for BLITZ assembly code.
  (The next line shows standard tab stops.)
```

```
1       t       t       t       t       t       t

! main ()
!
! This routine does such and such.
!
        .text
        .export main
main:   push    r1              ! Save registers
        push    r2              ! .
loop:                           ! LOOP
        cmp     r1,10           !   IF r1>10 THEN
        ble     endif           !   .
        sub     r2,1,r2         !     r2--
endif:                          !   ENDIF
        sub     r1,r2,r3        !   r3 := r1-r2
        ...
```

Labels

A label must be followed by a colon token, but the colon is not part of the label. A label may appear on a line by itself or the label may appear on a line containing a BLITZ instruction or one of the following pseudo-ops:

.ascii .byte .word .skip

Labels are not allowed on any other assembler pseudo-ops. The label will define a new symbol, and the symbol will be given an offset relative to the beginning of the current segment. Labels defined in the current file may be exported and labels defined in other files may be imported. A label will name an address in memory, and as such a label cannot be given a final value until segment-linking time. During the assembly of the current file, labels in the file are given offsets relative to either the beginning of the .text, .data, or .bss segments.

Operand Syntax

See the BLITZ instruction reference manual for details about what operands each instruction requires. Operands are separated by commas. Registers are specified in lowercase (e.g., r4). A memory reference may be in one of the following forms, although not all forms are allowed in all instructions. (Here "R" stands for any register.)

- [R]
- [R+R]
- [R+expr]
- [expr]
- [--R]
- [R++]

Some instructions allow data to be included directly; in such cases the operand will consist of an expression. The expression may evaluate to an absolute or relative value. Certain instructions (like jmp, call, and the branch instructions) require the operand to be relative to the segment in which the instruction occurs.

Here are several sample instructions to illustrate operand syntax:

```
add     r3,r4,r5
mul     r7,size,r7
sub     r1, ((x*23) << (y+1)), r1
call    foo
push    r6,[--r14]
pop     [r14++],r6
load    [r3],r9
load    [r3+r4],r9
load    [r3+arrayBase],r9
load    [x],r9
jmp     r3
bne     loop
set     0x12ab34cd,r8
syscall 3
reti
tset    [r4],r9
```

```
            ldptbr  r5
```

Note that whenever an instruction reads or writes memory, brackets are used.

lddd -h

```
==============================
=====                    =====
=====  The BLITZ Linker  =====
=====                    =====
==============================

Copyright 2000, Harry H. Porter III
===================================
  Original Author:
    12/29/00 - Harry H. Porter III

Command Line Options
====================
  These command line options may be given in any order.
    filename1 filename2 filename3 ...
       The input object files, which will normally end with ".o".
       There must be at least one input file.
    -h
       Print this help info.  Ignore other options and exit.
    -o filename
       If there are no errors, an executable file will be created.  This
       option can be used to give the object file a specific name.
       If this option is not used, then the output file will be named
       "a.out".
    -l
       Print a listing on stdout.
    -s
       Print the symbol table on stdout.
    -p integer
       The page size.  The integer must be a multiple of 4 greater than
       zero.  (The default is 8192 = 8K.)
    -a integer
       The logical address at which to load the program at run-time.
       The integer must be a non-negative multiple of the page size.
       (The default is 0.)
```

dumpObj -h

```
================================================
=====                                      =====
=====  The BLITZ Object File Dump Program  =====
=====                                      =====
================================================

Copyright 2000, Harry H. Porter III
===================================
  Original Author:
    11/12/00 - Harry H. Porter III

Overview
========
  This program prints out a BLITZ ".o" or "a.out" file in human-readable
  form.  This program does some (very limited) error checking on the file.

Command Line Options
====================
  Command line options may be given in any order.
    -h
      Print this info.  The input source is ignored.
    filename
      The input source will come from this file.  (This file should be a
      ".o" or "a.out" file.)  If an input file is not named on the command
      line, the source must come from stdin.  Only one input source is allowed.
```

diskUtil -h

```
=======================================
=====                           =====
=====  The BLITZ Disk Utility   =====
=====                           =====
=======================================

Copyright 2004, Harry H. Porter III
===================================
  Original Author:
    10/07/04 - Harry H. Porter III

This command can be used to manipulate the BLITZ "DISK" file.

The BLITZ emulator simulates the BLITZ disk using a Unix file on the host
machine.  This program allows that file to be manipulated.  For example,
it can be used to copy an executable file containing a user program to the
BLITZ disk so that the BLITZ OS kernel can then access, load, and run it.

The BLITZ DISK is organized as follows.  The disk contains a single directory
and this is kept in sector 0.  The files are placed sequentially on the
disk, one after the other.  Each file will take up an integral number of
sectors.  Each file has an entry in the directory.  Each entry contains
     (1)  The starting sector
     (2)  The file length, in bytes (possibly zero)
     (3)  The number of characters in the file name
     (4)  The file name
The directory begins with three numbers:
     (1)  Magic Number (0x73747562 = "stub")
     (2)  Number of files (possibly zero)
     (3)  Number of the next free sector
These are followed by the entries for each file.

Once created, a BLITZ file may not have its size increased.  When a file is
removed, the free sectors become unusable; there is no compaction or any
attempt to reclaim the lost space.

Each time this program is run, it performs one of the following functions:
      Initialize  set up a new file system on the BLITZ disk
      List        list the directory on the BLITZ disk
      Create      create a new file of a given size
      Remove      remove a file
      Add         copy a file from Unix to BLITZ
      Extract     copy a file from BLITZ to Unix
      Write       write sectors from a Unix file to the BLITZ disk
 The following command line options tell which function is to be performed:

  -h
        Print this help info.  Ignore other options and exit.
  -d DiskFileName
        The file used to emulate the BLITZ disk.  If missing, "DISK" will be used.
  -i
        Initialize the file system on the BLITZ "DISK" file.  This will
        effectively remove all files on the BLITZ disk and reclaim all available
        space.
  -l
        List the directory on the BLITZ disk.
  -c BlitzFileName SizeInBytes
        Create a file of the given size on the BLITZ disk.  The BLITZ
        disk must not already contain a file with this name.  Only the
        directory will be modified; the actual data in the file will be
        whatever bytes happened to be on the disk already.
```

```
  -r BlitzFileName
        Remove the file with the given name from the directory on the BLITZ disk.
  -a UnixFilename BlitzFileName
        Copy a file from Unix to the BLITZ disk.  If BlitzFileName already
        exists, it must be large enough to accomodate the new data.
  -e BlitzFileName UnixFileName
        Extract a file from the BLITZ disk to Unix.  This command will copy
        the data from the BLITZ disk to a Unix file.  The Unix file may or may
        not already exist; its size will be shortened or lengthened as necessary.
  -w UnixFileName SectorNumber
        The UnixFileName must be an existing Unix file. The SectorNumber is an
        integer.  The Unix file data will be  written to the BLITZ disk, starting
        at sector SectorNumber.  The directory will not be modified.
  -v
        Verbose; print lots of messages.

Only one of -i, -l, -c, -r, -a, -e, or -w may be used at a time.
```