The BLITZ Assembler

Harry H. Porter III
Department of Computer Science
Portland State University

September 25, 2007

Introduction

This document describes the BLITZ Assembler tool. The assembler takes a single input source program, written in BLITZ assembly code, and produces an "object file". The input assembly code program may be produced as the output of a compiler or may be hand-coded by an assembly language programmer. Later, one or more object files are combined to produce an "executable file" by the linker.

The material in this document closely resembles the "help" info printed by

```
asm –h
```

However, the material in this document is more inclusive. This document includes corrections, clarifications, elaborations, and additional information.

Command Line Options

The assembler tool is called "`asm`". Like any Unix/Linux command, it accepts command line options. Here are the command line options, which 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 help info. All other options will be ignored and no assembly will be performed.

-l

Print a detailed listing on stdout, showing what machine code was generated.

-s

Print the symbol table on stdout, showing the values of all symbols appearing in the program.

-d

Print internal assembler info, for debugging the assembler tool itself.

-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 option is not used, 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

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

Integers may be specified in decimal or in hex. Integers must range from 0 to 2147483647. Hex notation may be used, and begin with `0x`, for example, `0x1234abcd`. The hex numerals `a-f` can be capitalized so, for example `0x1234ABCD` is equivalent to `0x1234abcd`. Shorter integers like `0xFFFF` are not sign-extended.

Floating-Point Constants (Real Numbers)

Real numbers are specified in decimal notation, for example:

`123.456`	
`456e10`	Either "e" or "." is required.
`456.789e10`	Both fraction and exponent can appear.
`+123.456`	
`-123.456`	A leading "+" or "-" is optionally allowed.
`456.789e10`	The "e" may be upper or lower case.
`456.789E-10`	The exponent can be negative.

`456.789e+10` If the exponent is positive, the "+" is optional.

Reals are represented using the IEEE double precision floating point representation. Each real number is represented with 64 bits. The leading sign is optional. The number must contain either a decimal point (".") or an "e" to indicate that it is a real number and not an integer. The "e" may be upper or lower case and must be followed by at least one digit. A sign may be included after the "e" and before the exponent.

Strings

Strings use double quotes. For example,

```
"Hello, world!\n"
```

The following escape sequences are allowed within strings:

```
\0      00    0     NUL               control-@
\a      07    7     BEL (alert)       control-G
\b      08    8     BS (backspace)    control-H
\t      09    9     HT (tab)          control-I
\n      0A    10    LF/NL (newline)   control-J
\v      0B    11    VT                control-K
\f      0C    12    FF (form feed)    control-L
\r      0D    13    CR (return)       control-M
\"      22    34    "
\'      27    39    '
\\      5C    92    \
\xHH    HH          (other)
```

The last form, in which HH are any two hex digits, can be used to include any byte within a string. For example, these two strings include the same bytes:

```
"Hello, world!\n"
"Hello, world!\h0A"
```

Strings may not contain newlines directly; in other words, a string may not span multiple lines. However, the following string is okay:

```
"First\nSecond\nThird\n"
```

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.

Note that—unlike in the "C" language—strings do not include the NUL character. However, it can be included where desired, as in:

```
"Hello, world!\n\0"
```

Characters

Character constants use single quotes. The same escape sequences are allowed. For example:

```
'x'
'\n'
```

A character constant must specify exactly one byte.

Comments

Comments begin with the exclamation mark (!) and extend through the 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, ...)
Register names (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 Commands

An number of assembler command may appear in the assembler source code. (Assembler commands are often called "pseudo-ops".) Syntactically, these command are used like normal machine instructions. For example, some of them expect operands and some of them may be prefixed with a label. For example:

```
          set       r1,mydata
          jmp       label43
          .data
mydata:   .word     123
```

However, these are commands to the assembler, telling it how to produce the machine code. These commands are processed directly by the assembler and, unlike normal instructions, are not translated into machine instructions.

To distinguish the assembler commands from normal instructions, most all of the commands start with a period.

.text

The instructions and data following the **.text** command (up to the next **.data** or **.bss** command) will be placed in the "**.text**" segment. The **.text** segment will be marked by the operating system as "read-only" during execution.

.data

The instructions and data following the **.data** command (up to the next **.text** or **.bss** command) will be placed in the "**.data**" segment. The **.data** segment will be marked by the operating system as "readable" and "writable" during execution.

.bss

The bytes following the **.bss** command (up to the next **.text** or **.data** command) will be placed in the "**.bss**" segment. The **.bss** segment will be initialized to zero by the operating system at program load time and will be flagged as "readable" and "writable".

.ascii

This command 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**, although many instructions do.

.double

This pseudo-op expects a single floating-point constant as an operand. Here are examples of allowable floating point constants:

```
1.2
-3.4E-21
+4.5e+21
```

.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*

For example:

```
            mov     MyStrLen,r1
            .data
MyStr:      .ascii  "hello, world"
MyStrEnd:
MyStrLen    =       MyStrEnd-MyStr
```

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 separate "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 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 32-bit integer value, although the assembler may not know the exact value. In some cases, the value cannot be computed until the linker is invoked and code is relocated to its final position in memory. In other cases, the assembler can determine the symbol's value.

Symbols may either be defined within the file being assembled or externally, in some other .s file. If the symbol is defined in this file, it will either be given an absolute value that can be determined during assembly, or it will be defined relative to values that will not be known until the linker is run.

Thus, each symbol used in a given .s file will be either

external
absolute
relative

An external symbol will have its value assigned in some other assembly source file so its value will not be available to the code in this file until segment-linking time. In other words, the assembler will not know the value, but the value will be determined later by the linker. An external symbol may be used in expressions within this file. However, the actual value will not be computed and the data will not be filled in until segment-linking time.

An absolute symbol has its value defined within the file being assembled. This quantity will be computed by the assembler and its value will be known while this file is assembled.

A relative symbol is defined relative to some other value that the assembler does not know. Just like an external symbol, the value of a relative symbol cannot be determined until the linker runs. The value of a relative symbol is an offset relative to some memory location or to an external symbol, neither of which can be determined until link time.

If a symbol is used in this file, but not defined in this file, then the symbol 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 by the linker at segment-linking time. In other words, if a symbol is ever imported by any .s file, then it must be exported by exactly one .s file.

More precisely, absolute symbols must be defined within this file, possibly as a function of other absolute symbols and constant values, but not in terms of external or relative symbols. External symbols are defined, not in the file being assembled, but in some other file. Any symbol defined within this file will be "relative" if it is defined in terms of an external symbol, or if it is defined relative to some position in the **.text**, **.data**, or **.bss** segments, or if it is defined as a function of another relative symbol.

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

Some 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, and appear exactly as shown. The braces { } are used to mean "zero or more" occurrences. 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 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 to evaluate 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. The label 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. The 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 first 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
.double
.skip
```

When an instruction is labeled, the value of the label will be the address of the first byte of the instruction. This address will be determined later by the linker, when the segments are placed in memory.

When the label applies to an **.ascii**, **.byte**, **.word**, or **.double** pseudo-op, the value will be the address of the first byte of the data. When the label applies to a **.skip** pseudo-op, the value will be the address of the first byte of the uninitialized data.

For example, consider the following code:

```
Lab1:   add       r1,0x1234,r2
Lab2:   .skip     2
Lab3:   .ascii    "abc"
```

Assume that the linker places this material at address `0x00010000`. Note that the `add` instruction is assembled to `0x80211234` and that `"abc"` is `0x616263` in ASCII.

The linker would load memory with the following data:

address	data	
00010000	80	← Lab1
00010001	21	
00010002	12	
00010003	34	
00010004	00	← Lab2
00010005	00	
00010006	61	← Lab3
00010007	62	
00010008	63	

Addresses would be assigned as follows:

```
Lab1 = 00010000
Lab2 = 00010004
Lab3 = 00010006
```

Each label definition 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 defined in the file are given offsets relative to either the beginning of the **.text**, **.data**, or **.bss** segments.

Note that labels are not allowed on the following pseudo-ops:

```
.text
.data
.bss
.export
.import
.align
```

The **.text**, **.data**, and **.bss** segments are not named, therefore these pseudo-ops permit neither a label nor any operands. The **.export** and **.import** pseudo-ops refer to symbols, but these symbols are specified in the operand field, not as labels. Finally, the **.align** pseudo-op may or may not place bytes in memory; it makes no sense to label these padding bytes, especially if there are zero of them.

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 "*Reg*" stands for any register and "*Expr*" stands for any expression.)

```
[ Reg]
[ Reg + Reg ]
[ Reg + Expr ]
[ Expr ]
[ --Reg ]
[ Reg++ ]
```

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.

An Example Program

Below is a example program written in BLITZ assembly code:

```
        .text
!
! Imported Symbols:
!
        .import putChar
!
! Exported Symbols:
!
        .export main
!
! main
!
! This routine prints "Hello, world!"
!
main:   set     hello,r1                ! print message
        call    putString               ! .
        ret                             ! return
!
! putString
!
! This routine is passed r1, a pointer to a string
! of characters, terminated by '\0'.  It sends all
! of them except the final '\0' to the output by
! calling 'putChar' repeatedly.
!
! Registers modified: none
!
putString:
        push    r1                      ! save registers
        push    r2                      ! .
        mov     r1,r2                   ! r2 := ptr into string
putStLoop:                              ! loop
        loadb   [r2],r1                 !   r1 := next char
        add     r2,1,r2                 !   incr ptr
        cmp     r1,0                    !   if (r1 == '\0')
        be      putStExit               !     then break
```

```
          call    putChar                 !   putChar (r1)
          jmp     putStLoop               ! end
   putStExit:                             ! .
          pop     r2                      ! restore regs
          pop     r1                      ! .
          ret                             ! return
   !
   ! The data area
   !
          .data
   hello:  .ascii  "Hello, world!\n\0"
```

Listings

If the above program is assembled with using the "-l" command line option:

```
asm MyProgram.s -l
```

then the following "listing" will be printed on stdout.

```
000000                  .text
                !
                ! Imported Symbols:
                !
                        .import putChar
                !
                ! Exported Symbols:
                !
                        .export main
                !
                ! main
                !
                ! This routine prints "Hello, world!"
                !
000000 c0100000 main:   set     hello,r1                ! print message
000004 c1100000
000008 a0000008         call    putString               ! .
00000c 09000000         ret                             ! return
                !
                ! putString
                !
                ! This routine is passed r1, a pointer to a string
                ! of characters, terminated by '\0'.  It sends all
                ! of them except the final '\0' to the output by
                ! calling 'putChar' repeatedly.
                !
                ! Registers modified: none
                !
000010          putString:
000010 541f0000         push    r1                      ! save registers
000014 542f0000         push    r2                      ! .
000018 67210000         mov     r1,r2                   ! r2 := ptr into string
```

```
00001c           putStLoop:                              ! loop
00001c 6c120000          loadb   [r2],r1                 !   r1 := next char
000020 80220001          add     r2,1,r2                 !   incr ptr
000024 81010000          cmp     r1,0                    !   if (r1 == '\0')
000028 a200000c          be      putStExit               !     then break
00002c a0000000          call    putChar                 !   putChar (r1)
000030 a1ffffec          jmp     putStLoop               ! end
000034           putStExit:                              ! .
000034 552f0000          pop     r2                      ! restore regs
000038 551f0000          pop     r1                      ! .
00003c 09000000          ret                             ! return
                 !
                 ! The data area
                 !
000000                   .data
000000 48656c6c hello:   .ascii  "Hello, world!\n\0"
```

In the leftmost column, you see the addresses and in the next column you see the data to be placed in memory. Finally, the actual corresponding line from the source file is shown. Notice that the addresses are all relative, with the **.text** and **.data** segments (and the **.bss** segment) beginning at zero.

If the above program is assembled with using the "-s" command line option:

```
asm MyProgram.s -s
```

then the following information about the symbols will be printed:

```
STRING             IMPORT  EXPORT  OFFSET  RELATIVE-TO
===========        ======  ======  ======  ===========
putStLoop                          28      .text
hello                              0       .data
putString                          16      .text
putStExit                          52      .text
Hello, world!\n\0                  0
main                       export  0       .text
_entry                             0
putChar            import          0
```

The symbol information shows all strings and symbols appearing in the program, as well as information about whether they are imported or exported. For each symbol we also see an offset. This value is always 0 for imported symbols. For relative symbols, the last column shows what this offset is relative to. For absolute symbols, the final column is blank and the offset column shows the absolute value.

Warnings and Error Messages

Warnings are printed in the form:

```
Warning on line <number>: <message>
```

Below is a list of all the warning messages that may be produced. Each one almost certainly indicates a programmer error.

```
Immediate value (0xHHHHHHHH) exceeds 16-bit limit.
In SETLO, the data exceeds 16 bits in length
In SETHI, the data appears to be in the form 0x1234 instead
                                                    of 0x12340000 as expected
Instruction not on aligned address
Relative branch offset (HHHHHHHH) exceeds 24-bit limit.
```

Errors are printed in the form:

`Error on line` *<number>*`:` *<message>*

Here, in alphabetical order, are all the error messages that may be produced:

```
.align takes no operands
.bss takes no operands
.data takes no operands
.skip expression may not use symbols defined after it
.text takes no operands
A label is not allowed on .align
A label is not allowed on .bss
A label is not allowed on .data
A label is not allowed on .export
A label is not allowed on .import
A label is not allowed on .text
A lone < is not a valid token
A lone > is not a valid token
At least one digit is required after decimal
Attempt to export a symbol which is not defined in this file: <XXXXX>
Attempt to import a symbol which is also defined in this file
Both operands to binary + may not be relative
Call, jump, or branch has an absolute value as an operand
EOF encountered within a string
End-of-file encountered after a \\ escape
End-of-file encountered after a \\x escape
End-of-file encountered within a comment
End-of-line (CR) encountered after a \\ escape
End-of-line (CR) encountered after a \\x escape
End-of-line (CR) encountered within a string
End-of-line (NL) encountered after a \\ escape
End-of-line (NL) encountered after a \\x escape
End-of-line (NL) encountered within a string
Expecting ')' in expression")) return NULL;
Expecting + after reg Ra
Expecting + or ]
Expecting ++ in [Ra++],Rc
Expecting -- in Rc,[--Ra]
Expecting Ra or Ra+Rc
Expecting Rc or Rc+data16
Expecting Register Fa
Expecting Register Fb
Expecting Register Fc
```

```
Expecting Register Ra
Expecting Register Rb
Expecting Register Rc
Expecting [ after comma
Expecting [ after op-code
Expecting [ in Rc,[--Ra]
Expecting [ in [Ra],Rc
Expecting ]
Expecting ] in Rc,[--Ra]
Expecting ] in Rc,[Ra+data16]
Expecting ] in [Ra++],Rc
Expecting ] in [Ra+data16],Ra
Expecting ] in [Ra],Rc
Expecting ] or + after Rc,[Ra...
Expecting ] or + after [Ra...
Expecting a floating point constant
Expecting closing quote in character constant
Expecting comma
Expecting comma after expression
Expecting comma after reg Fc
Expecting comma after reg Rc
Expecting comma in Fc,Ra
Expecting comma in Ra,Rb
Expecting comma in Ra,Rb or Ra,data16
Expecting comma in Ra,Rc
Expecting comma in Ra,Rc or [Ra+data16],Rc
Expecting comma in Rb,Rc
Expecting comma in Rc,Ra or Rc,[Ra+data16]
Expecting comma in [Ra++],Rc
Expecting comma in [Ra],Rc
Expecting comma in data16,Rb
Expecting comma in data16,Rc
Expecting comma in data32,Rc
Expecting either Rc or Rc,[--Ra]
Expecting exponent numerals
Expecting expression
Expecting string after .ascii
Expecting symbol after .export
Expecting symbol after .import
Exponent is out of range
Hex constants must be 8 or fewer digits
Illegal escape (only \\0, \\a, \\b, \\t, \\n, \\v, \\f, \\r, \\\", \\\', \\\\,
                                                          and \\xHH allowed)
Integer out of range (0..2147483647); use 0x80000000 for -2147483648
Invalid op-code or missing colon after label
Invalid or missing op-code
Maximum string length exceeded
Must have a hex digit after 0x
Must have a hex digit after \\x
Must have two hex digits after \\x
No legal instructions encountered
Not currently in a .text, .data, or .bss segment
Not in .text, .data, or .bss segment
Operands to % must be positive
Operands to / must be positive
Operands to binary - are relative to different symbols
Real number is out of range
Shift amount must be within 0..31
```

```
The % operator requires operands to be absolute values
The .skip expression must evaluate to an absolute value
The / operator requires operands to be absolute values
The << operator requires operands to be absolute values
The >> operator requires operands to be absolute values
The >>> operator requires operands to be absolute values
The ^ operator requires operands to be absolute values
The unary - operator requires operand to be an absolute value
The | operator requires operands to be absolute values
The ~ operator requires its operand to be an absolute value
This symbol is already defined
Undefined symbol: <XXXXX>
Unexpected material after [Ra],Rc
Unexpected material after op-code
Unexpected material after operand Rc
Unexpected material after operands
Unexpected period within identifier
Unexpected tokens after expression
Unexpected tokens after floating constant
Unexpected tokens after operands
Unexpected tokens after string
Unexpected tokens after symbol
We are not currently in the .text or .data segment
When strings are used in places expecting an integer, the string must be
                                                        exactly 4 chars long
"_entry" is not in the .text segment
"_entry" not at beginning of .text segment
```

When errors are detected, no object file will be created.