Berkeley VAX/UNIX Assembler Reference Manual

John F. Reiser
Bell Laboratories,
Holmdel, NJ

and

Robert R. Henry[1]
Electronics Research Laboratory
University of California
Berkeley, CA 94720

November 5, 1979

Revised
February 9, 1983

1. Introduction

This document describes the usage and input syntax of the UNIX VAX-11 assembler *as*. *As* is designed for assembling the code produced by the ''C'' compiler; certain concessions have been made to handle code written directly by people, but in general little sympathy has been extended. This document is intended only for the writer of a compiler or a maintainer of the assembler.

1.1. Assembler Revisions since November 5, 1979

There has been one major change to *as* since the last release. *As* has been updated to assemble the new instructions and data formats for ''G'' and ''H'' floating point numbers, as well as the new queue instructions.

1.2. Features Supported, but No Longer Encouraged as of February 9, 1983

These feature(s) in *as* are supported, but no longer encouraged.

- The colon operator for field initialization is likely to disappear.

2. Usage

As is invoked with these command arguments:

as [**–LVWJR**] [**–d***n*] [**–DTS**] [**–t** *directory*] [**–o** *output*] [*name*$_1$] · · · [*name*$_n$]

The **–L** flag instructs the assembler to save labels beginning with a ''L'' in the symbol table portion of the *output* file. Labels are not saved by default, as the default action of the link editor *ld* is to discard them anyway.

The **–V** flag tells the assembler to place its interpass temporary file into virtual memory. In normal circumstances, the system manager will decide where the temporary file should lie. Our experiments with very large temporary files show that placing the temporary file into virtual memory will save about 13% of the assembly time, where the size of the temporary file is about 350K bytes. Most assembler sources will not be this long.

The **–W** turns of all warning error reporting.

The **–J** flag forces UNIX style pseudo–branch instructions with destinations further away than a byte displacement to be turned into jump instructions with 4 byte offsets. The **–J** flag buys you nothing if **–d2**

[1]Preparation of this paper supported in part by the National Science Foundation under grant MCS #78-07291.

is set. (See §8.4, and future work described in §11)

The **−R** flag effectively turns ‘‘**.data** *n*’’ directives into ‘‘**.text** *n*’’ directives. This obviates the need to run editor scripts on assembler source to ‘‘read−only’’ fix initialized data segments. Uninitialized data (via **.lcomm** and **.comm** directives) is still assembled into the data or bss segments.

The **−d** flag specifies the number of bytes which the assembler should allow for a displacement when the value of the displacement expression is undefined in the first pass. The possible values of *n* are 1, 2, or 4; the assembler uses 4 bytes if **-d** is not specified. See §8.2.

Provided the **−V** flag is not set, the **−t** flag causes the assembler to place its single temporary file in the *directory* instead of in */tmp*.

The **−o** flag causes the output to be placed on the file *output*. By default, the output of the assembler is placed in the file *a.out* in the current directory.

The input to the assembler is normally taken from the standard input. If file arguments occur, then the input is taken sequentially from the files $name_1$, $name_2$ $\cdots$ $name_n$ This is not to say that the files are assembled separately; $name_1$ is effectively concatenated to $name_2$, so multiple definitions cannot occur amongst the input sources.

The −**D** (debug), −**T** (token trace), and the −**S** (symbol table) flags enable assembler trace information, provided that the assembler has been compiled with the debugging code enabled. The information printed is long and boring, but useful when debugging the assembler.

3. Lexical conventions

Assembler tokens include identifiers (alternatively, ‘‘symbols’’ or ‘‘names’’), constants, and operators.

3.1. Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ‘‘ **.** ’’, underscore ‘‘_’’, and dollar ‘‘$’’). The first character may not be numeric. Identifiers may be (practically) arbitrary long; all characters are significant.

3.2. Constants

3.2.1. Scalar constants

All scalar (non floating point) constants are (potentially) 128 bits wide. Such constants are interpreted as two’s complement numbers. Note that 64 bit (quad words) and 128 bit (octal word) integers are only partially supported by the VAX hardware. In addition, 128 bit integers are only supported by the extended VAX architecture. *As* supports 64 and 128 bit integers only so they can be used as immediate constants or to fill initialized data space. *As* can not perform arithmetic on constants larger than 32 bits.

Scalar constants are initially evaluated to a full 128 bits, but are pared down by discarding high order copies of the sign bit and categorizing the number as a long, quad or octal integer. Numbers with less precision than 32 bits are treated as 32 bit quantities.

The digits are ‘‘0123456789abcdefABCDEF’’ with the obvious values.

An octal constant consists of a sequence of digits with a leading zero.

A decimal constant consists of a sequence of digits without a leading zero.

A hexadecimal constant consists of the characters ‘‘0x’’ (or ‘‘0X’’) followed by a sequence of digits.

A single-character constant consists of a single quote ‘‘ ′ ’’ followed by an ASCII character, including ASCII newline. The constant’s value is the code for the given character.

3.2.2. Floating Point Constants

Floating point constants are internally represented in the VAX floating point format that is specified by the lexical form of the constant. Using the meta notation that [dec] is a decimal digit (‘‘0123456789’’), [expt] is a type specification character (‘‘fFdDhHgG’’), [expe] is a exponent delimiter and type specification character (‘‘eEfFdDhHgG’’), x^* means 0 or more occurences of x, x^+ means 1 or more occurences of x, then the general lexical form of a floating point number is:

$$0[expe]([+-])[dec]^+(.)([dec]^*)([expt]([+-])(dec]^+))$$

The standard semantic interpretation is used for the signed integer, fraction and signed power of 10 exponent. If the exponent delimiter is specified, it must be either an ‘‘e’’ or ‘‘E’’, or must agree with the initial type specification character that is used. The type specification character specifies the type and representation of the constructed number, as follows:

type character	floating representation	size (bits)
f, F	F format floating	32
d, D	D format floating	64
g, G	G format floating	64
h, H	H format floating	128

Note that ‘‘G’’ and ‘‘H’’ format floating point numbers are not supported by all implementations of the VAX architecture. *As* does not require the augmented architecture in order to run.

The assembler uses the library routine *atof()* to convert ‘‘F’’ and ‘‘D’’ numbers, and uses its own conversion routine (derived from *atof*, and believed to be numerically accurate) to convert ‘‘G’’ and ‘‘H’’ floating point numbers.

Collectively, all floating point numbers, together with quad and octal scalars are called *Bignums*. When *as* requires a Bignum, a 32 bit scalar quantity may also be used.

3.2.3. String Constants

A string constant is defined using the same syntax and semantics as the ‘‘C’’ language uses. Strings begin and end with a ‘‘"’’ (double quote). The DEC MACRO-32 assembler conventions for flexible string quoting is not implemented. All ‘‘C’’ backslash conventions are observed; the backslash conventions peculiar to the PDP-11 assembler are not observed. Strings are known by their value and their length; the assembler does not implicitly end strings with a null byte.

3.3. Operators

There are several single-character operators; see §6.1.

3.4. Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

3.5. Scratch Mark Comments

The character ‘‘#’’ introduces a comment, which extends through the end of the line on which it appears. Comments starting in column 1, having the format ‘‘# *expression string*’’, are interpreted as an indication that the assembler is now assembling file *string* at line *expression*. Thus, one can use the ‘‘C’’ preprocessor on an assembly language source file, and use the *#include* and *#define* preprocessor directives. (Note that there may not be an assembler comment starting in column 1 if the assembler source is given to the ‘‘C’’ preprocessor, as it will be interpreted by the preprocessor in a way not intended.) Comments are otherwise ignored by the assembler.

3.6. ‘‘C’’ Style Comments

The assembler will recognize ‘‘C’’ style comments, introduced with the prologue **/*** and ending with the epilogue ***/**. ‘‘C’’ style comments may extend across multiple lines, and are the preferred comment style to use if one chooses to use the ‘‘C’’ preprocessor.

4. Segments and Location Counters

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The UNIX operating system makes some assumptions about the content of these segments; the assembler does not. Within the text and data segments there are a number of sub-segments, distinguished by number (‘‘**text** 0’’, ‘‘**text** 1’’, · · · ‘‘**data** 0’’, ‘‘**data** 1’’, · · ·). Currently there are four subsegments each in text and data. The subsegments are for programming convenience only.

Before writing the output file, the assembler zero-pads each text subsegment to a multiple of four bytes and then concatenates the subsegments in order to form the text segment; an analogous operation is done for the data segment. Requesting that the loader define symbols and storage regions is the only action allowed by the assembler with respect to the bss segment. Assembly begins in ‘‘**text** 0’’.

Associated with each (sub)segment is an implicit location counter which begins at zero and is incremented by 1 for each byte assembled into the (sub)segment. There is no way to explicitly reference a location counter. Note that the location counters of subsegments other than ‘‘**text** 0’’ and ‘‘**data** 0’’ behave peculiarly due to the concatenation used to form the text and data segments.

5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are two kinds of statements: null statements and keyword statements. Either kind of statement may be preceded by one or more labels.

5.1. Named Global Labels

A global label consists of a name followed by a colon. The effect of a name label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the value assigned changes the definition of the label.

A global label is referenced by its name.

Global labels beginning with a ‘‘ L ’’ are discarded unless the **–L** option is in effect.

5.2. Numeric Local Labels

A numeric label consists of a digit 0 to 9 followed by a colon. Such a label serves to define temporary symbols of the form ‘‘*n* b’’ and ‘‘*n* f’’, where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of the location counter to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References to symbols of the form ‘‘*n* b’’ refer to the first numeric label ‘‘*n* :’’ *b*ackwards from the reference; ‘‘*n* f’’ symbols refer to the first numeric label ‘‘*n* :’’ *f*orwards from the reference. Such numeric labels conserve the inventive powers of the human programmer.

For various reasons, *as* turns local labels into labels of the form L*n* .$*m* . Although unlikely, these generated labels may conflict with programmer defined labels.

5.3. Null statements

A null statement is an empty statement ignored by the assembler. A null statement may be labeled, however.

5.4. Keyword statements

A keyword statement begins with one of the many predefined keywords known to *as*; the syntax of the remainder of the statement depends on the keyword. All instruction opcodes are keywords. The remaining keywords are assembler pseudo-operations, also called *directives*. The pseudo-operations are listed in §8, together with the syntax they require.

6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, operators, and parentheses. Each expression has a type.

All operators in expressions are fundamentally binary in nature. Arithmetic is two's complement and has 32 bits of precision. *As* can not do arithmetic on floating point numbers, quad or octal precision scalar numbers. There are four levels of precedence, listed here from lowest precedence level to highest:

<table>
<tr><th>precedence</th><th>operators</th></tr>
<tr><td>binary</td><td>+, −</td></tr>
<tr><td>binary</td><td>| , & , ˆ , !</td></tr>
<tr><td>binary</td><td>* , /, %,</td></tr>
<tr><td>unary</td><td>−, ˜</td></tr>
</table>

All operators of the same precedence are evaluated strictly left to right, except for the evaluation order enforced by parenthesis.

6.1. Expression Operators

The operators are:

<table>
<tr><th>operator</th><th>meaning</th></tr>
<tr><td>+</td><td>addition</td></tr>
<tr><td>−</td><td>(binary) subtraction</td></tr>
<tr><td>*</td><td>multiplication</td></tr>
<tr><td>/</td><td>division</td></tr>
<tr><td>%</td><td>modulo</td></tr>
<tr><td>−</td><td>(unary) 2's complement</td></tr>
<tr><td>&</td><td>bitwise and</td></tr>
<tr><td>|</td><td>bitwise or</td></tr>
<tr><td>ˆ</td><td>bitwise exclusive or</td></tr>
<tr><td>!</td><td>bitwise or not</td></tr>
<tr><td>˜</td><td>bitwise 1's complement</td></tr>
<tr><td>></td><td>logical right shift</td></tr>
<tr><td>≫</td><td>logical right shift</td></tr>
<tr><td><</td><td>logical left shift</td></tr>
<tr><td>≪</td><td>logical left shift</td></tr>
</table>

Expressions may be grouped by use of parentheses, ‘‘('' and ‘‘)''.

6.2. Data Types

The assembler manipulates several different types of expressions. The types likely to be met explicitly are:

undefined — Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external — A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute — An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text — The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of ‘‘**.**’’ is ‘‘text 0’’.

data The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first **.data** statement, the value of ‘‘**.**’’ is ‘‘data 0’’.

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments.

external absolute, text, data, or bss
Symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register The symbols

r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 ap fp sp pc

are predefined as register symbols. In addition, the ‘‘%’’ operator converts the following absolute expression whose value is between 0 and 15 into a register reference.

other types
Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

6.3. Type Propagation in Expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then

(1) If one of the operands is undefined, the result is undefined.

(2) If both operands are absolute, the result is absolute.

(3) If an absolute is combined with one of the ‘‘other types’’ mentioned above, the result has the other type. An ‘‘other type’’ combined with an explicitly discussed type other than absolute it acts like an absolute.

Further rules applying to particular operators are:

\+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

− If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

othersIt is illegal to apply these operators to any but absolute symbols.

7. Pseudo-operations (Directives)

The keywords listed below introduce directives or instructions, and influence the later behavior of the assembler for this statement. The metanotation

[stuff]

means that 0 or more instances of the given ‘‘stuff’’ may appear.

Boldface tokens must appear literally; words in *italic* words are substitutable.

The pseudo–operations listed below are grouped into functional categories.

7.1. Interface to a Previous Pass

.ABORT

As soon as the assembler sees this directive, it ignores all further input (but it does read to the end of file), and aborts the assembly. No files are created. It is anticipated that this would be used in a pipe interconnected version of a compiler, where the first major syntax error would cause the compiler to issue this directive, saving unnecessary work in assembling code that would have to be discarded anyway.

.file *string*

This directive causes the assembler to think it is in file *string*, so error messages reflect the proper source file.

.line *expression*

This directive causes the assembler to think it is on line *expression* so error messages reflect the proper source file.

The only effect of assembling multiple files specified in the command string is to insert the *file* and *line* directives, with the appropriate values, at the beginning of the source from each file.

*expression string*
*expression*

This is the only instance where a comment is meaningful to the assembler. The ‘‘#’’ *must* be in the first column. This meta comment causes the assembler to believe it is on line *expression*. The second argument, if included, causes the assembler to believe it is in file *string*, otherwise the current file name does not change.

7.2. Location Counter Control

.data [*expression*]
.text [*expression*]

These two pseudo-operations cause the assembler to begin assembling into the indicated text or data subsegment. If specified, the *expression* must be defined and absolute; an omitted *expression* is treated as zero. The effect of a **.data** directive is treated as a **.text** directive if the **–R** assembly flag is set. Assembly starts in the **.text** 0 subsegment.

The directives **.align** and **.org** also control the placement of the location counter.

7.3. Filled Data

.align *align_expr* [**,** *fill_expr*]

The location counter is adjusted so that the *expression* lowest bits of the location counter become zero. This is done by assembling from 0 to 2^{align_expr} bytes, taken from the low order byte of *fill_expr*. If present, *fill_expr* must be absolute; otherwise it defaults to 0. Thus ‘‘.align 2’’ pads by null bytes to make the location counter evenly divisible by 4. The *align_expr* must be defined, absolute, nonnegative, and less than 16.

Warning: the subsegment concatenation convention and the current loader conventions may not preserve attempts at aligning to more than 2 low-order zero bits.

.org *org_expr* [**,** *fill_expr*]

The location counter is set equal to the value of *org_expr*, which must be defined and absolute. The value of the *org_expr* must be greater than the current value of the location counter. Space between the current value of the location counter and the desired value are filled with bytes taken from the low order byte of *fill_expr*, which must be absolute and defaults to 0.

.space *space_expr* [**,** *fill_expr*]

The location counter is advanced by *space_expr* bytes. *Space_expr* must be defined and absolute. The space is filled in with bytes taken from the low order byte of *fill_expr*, which must be defined and absolute. *Fill_expr* defaults to 0. The **.fill** directive is a more general way to accomplish the **.space** directive.

.fill *rep_expr* **,** *size_expr* **,** *fill_expr*

All three expressions must be absolute. *fill_expr*, treated as an expression of size *size_expr* bytes, is assembled and replicated *rep_expr* times. The effect is to advance the current location counter *rep_expr* ∗ *size_expr* bytes. *size_expr* must be between 1 and 8.

7.4. Symbol Definitions

7.5. Initialized Data

.byte *expr* [**,** *expr*]
.word *expr* [**,** *expr*]
.int *expr* [**,** *expr*]
.long *expr* [**,** *expr*]

The *expression*s in the comma-separated list are truncated to the size indicated by the key word:

keyword	length (bits)
.byte	8
.word	16
.int	32
.long	32

and assembled in successive locations. The *expression*s must be absolute.

Each *expression* may optionally be of the form:

$$expression_1 : expression_2$$

In this case, the value of $expression_2$ is truncated to $expression_1$ bits, and assembled in the next $expression_1$ bit field which fits in the natural data size being assembled. Bits which are skipped because a field does not fit are filled with zeros. Thus, ‘‘**.byte** 123’’ is equivalent to ‘‘**.byte** 8:123’’, and ‘‘**.byte** 3:1,2:1,5:1’’ assembles two bytes, containing the values 9 and 1.

NB: Bit field initialization with the colon operator is likely to disappear in future releases of the assembler.

.quad *number* [, *number*]
.octa *number* [, *number*]
.float *number* [, *number*]
.double *number* [, *number*]
.ffloat *number* [, *number*]
.dfloat *number* [, *number*]
.gfloat *number* [, *number*]
.hfloat *number* [, *number*]

These initialize Bignums (see §3.2.2) in successive locations whose size is a function on the key word. The type of the Bignums (determined by the exponent field, or lack thereof) may not agree with type implied by the key word. The following table shows the key words, their size, and the data types for the Bignums they expect.

keyword	format	length (bits)	valid *number* (s)
.quad	quad scalar	64	scalar
.octa	octal scalar	128	scalar
.float	F float	32	F, D and scalar
.ffloat	F float	32	F, D and scalar
.double	D float	64	F, D and scalar
.dfloat	D float	64	F, D and scalar
.gfloat	G float	64	G scalar
.hfloat	H float	128	H scalar

As will correctly perform other floating point conversions while initializing, but issues a warning message. *As* performs all floating point initializations and conversions using only the facilities defined in the original (native) architecture.

.ascii *string* [, *string*]
.asciz *string* [, *string*]

Each *string* in the list is assembled into successive locations, with the first letter in the string being placed into the first location, etc. The **.ascii** directive will not null pad the string; the **.asciz** directive will null pad the string. (Recall that strings are known by their length, and need not be terminated with a null, and that the ‘‘C’’ conventions for escaping are understood.) The **.ascii** directive is identical to:
.byte $string_0$ **,** $string_1$ **,** $\cdots$

.comm *name* **,** *expression*

Provided the *name* is not defined elsewhere, its type is made ‘‘undefined external’’, and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes.

.lcomm *name* **,** *expression*

expression bytes will be allocated in the bss segment and *name* assigned the location of the first byte, but the *name* is not declared as global and hence will be unknown to the link editor.

.globl *name*

This statement makes the *name* external. If it is otherwise defined (by **.set** or by appearance as a label) it acts within the assembly exactly as if the **.globl** statement were not given; however, the link editor may be used to combine this object module with other modules referring to this symbol.

Conversely, if the given symbol is not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbol. The assembler makes all otherwise undefined symbols external.

.set *name* **,** *expression*

The (*name* , *expression*) pair is entered into the symbol table. Multiple **.set** statements with the same name are legal; the most recent value replaces all previous values.

.lsym *name* **,** *expression*

A unique and otherwise unreferencable instance of the (*name* , *expression*) pair is created in the symbol table. The Fortran 77 compiler uses this mechanism to pass local symbol definitions to the link editor and debugger.

.stabs *string* , $expr_1$, $expr_2$, $expr_3$, $expr_4$
.stabn $expr_1$, $expr_2$, $expr_3$, $expr_4$
.stabd $expr_1$, $expr_2$, $expr_3$

The *stab* directives place symbols in the symbol table for the symbolic debugger, *sdb*[2]. A ‘‘stab’’ is a *s*ymbol *tab*le entry. The **.stabs** is a string stab, the **.stabn** is a stab not having a string, and the **.stabd** is a ‘‘dot’’ stab that implicitly references ‘‘dot’’, the current location counter.

The *string* in the **.stabs** directive is the name of a symbol. If the symbol name is zero, the **.stabn** directive may be used instead.

The other expressions are stored in the name list structure of the symbol table and preserved by the loader for reference by *sdb*; the value of the expressions are peculiar to formats required by *sdb*.

$expr_1$ is used as a symbol table tag (nlist field *n_type*).

$expr_2$ seems to always be zero (nlist field *n_other*).

$expr_3$ is used for either the source line number, or for a nesting level (nlist field *n_desc*).

$expr_4$ is used as tag specific information (nlist field *n_value*). In the case of the **.stabd** directive, this expression is nonexistent, and is taken to be the value of the location counter at the following instruction. Since there is no associated name for a **.stabd** directive, it can only be used in circumstances where the name is zero. The effect of a **.stabd** directive can be achieved by one of the other **.stab**x directives in the following manner:
.stabn $expr_1$, $expr_2$, $expr_3$, LL*n*
LL*n* **:**

The **.stabd** directive is preferred, because it does not clog the symbol table with labels used only for the stab symbol entries.

8. Machine instructions

The syntax of machine instruction statements accepted by *as* is generally similar to the syntax of DEC MACRO-32. There are differences, however.

8.1. Character set

As uses the character ‘‘$’’ instead of ‘‘#’’ for immediate constants, and the character ‘‘*’’ instead of ‘‘@’’ for indirection. Opcodes and register names are spelled with lower-case rather than upper-case letters.

[2]Katseff, H.P. *Sdb: A Symbol Debugger.* Bell Laboratories, Holmdel, NJ. April 12, 1979.
Katseff, H.P. *Symbol Table Format for Sdb*, File 39394, Bell Laboratories, Holmdel, NJ. March 14, 1979.

8.2. Specifying Displacement Lengths

Under certain circumstances, the following constructs are (optionally) recognized by *as* to indicate the number of bytes to allocate for the displacement used when constructing displacement and displacement deferred addressing modes:

primary	alternate	length
B`	**Bˆ**	byte (1 byte)
W`	**Wˆ**	word (2 bytes)
L`	**Lˆ**	long word (4 bytes)

One can also use lower case **b**, **w** or **l** instead of the upper case letters. There must be no space between the size specifier letter and the ‘‘ˆ’’ or ‘‘`’’. The constructs **Sˆ** and **Gˆ** are not recognized by *as*, as they are by the DEC MACRO-32 assembler. It is preferred to use the ‘‘`’’displacement so that the ‘‘ˆ’’ is not misinterpreted as the **xor** operator.

Literal values (including floating-point literals used where the hardware expects a floating-point operand) are assembled as short literals if possible, hence not needing the **Sˆ** DEC MACRO-32 directive.

If the displacement length modifier is present, then the displacement is **always** assembled with that displacement, even if it will fit into a smaller field, or if significance is lost. If the length modifier is not present, and if the value of the displacement is known exactly in *as*’s first pass, then *as* determines the length automatically, assembling it in the shortest possible way, Otherwise, *as* will use the value specified by the **−d** argument, which defaults to 4 bytes.

8.3. case*x* Instructions

As considers the instructions **caseb**, **casel**, **casew** to have three operands. The displacements must be explicitly computed by *as*, using one or more **.word** statements.

8.4. Extended branch instructions

These opcodes (formed in general by substituting a ‘‘j’’ for the initial ‘‘b’’ of the standard opcodes) take as branch destinations the name of a label in the current subsegment. It is an error if the destination is known to be in a different subsegment, and it is a warning if the destination is not defined within the object module being assembled.

If the branch destination is close enough, then the corresponding short branch ‘‘b’’ instruction is assembled. Otherwise the assembler choses a sequence of one or more instructions which together have the same effect as if the ‘‘b’’ instruction had a larger span. In general, *as* chooses the inverse branch followed by a **brw**, but a **brw** is sometimes pooled among several ‘‘j’’ instructions with the same destination.

As is unable to perform the same long/short branch generation for other instructions with a fixed byte displacement, such as the **sob**, **aob** families, or for the **acbx** family of instructions which has a fixed word displacement. This would be desirable, but is prohibitive because of the complexity of these instructions.

If the **−J** assembler option is given, a **jmp** instruction is used instead of a **brw** instruction for **ALL** ‘‘j’’ instructions with distant destinations. This makes assembly of large (>32K bytes) programs (inefficiently) possible. *As* does not try to use clever combinations of **brb**, **brw** and **jmp** instructions. The **jmp** instructions use PC relative addressing, with the length of the offset given by the **−d** assembler option.

These are the extended branch instructions *as* recognizes:

jeql	**jeqlu**	**jneq**
jgeq	**jgequ**	**jgtr**
jleq	**jlequ**	**jlss**
jbcc	**jbsc**	**jbcs**
jlbc	**jlbs**	
jcc	**jcs**	
jvc	**jvs**	
jbc	**jbs**	
jbr		

Note that **jbr** turns into **brb** if its target is close enough; otherwise a **brw** is used.

9. Diagnostics

Diagnostics are intended to be self explanatory and appear on the standard output. Diagnostics either report an *error* or a *warning.* Error diagnostics complain about lexical, syntactic and some semantic errors, and abort the assembly.

The majority of the warnings complain about the use of VAX features not supported by all implementations of the architecture. *As* will warn if new opcodes are used, if ''G'' or ''H'' floating point numbers are used and will complain about mixed floating conversions.

10. Limits

limit	what
Arbitrary[3]	Files to assemble
BUFSIZ	Significant characters per name
Arbitrary	Characters per input line
Arbitrary	Characters per string
Arbitrary	Symbols
4	Text segments
4	Data segments

11. Annoyances and Future Work

Most of the annoyances deal with restrictions on the extended branch instructions.

As only uses a two level algorithm for resolving extended branch instructions into short or long displacements. What is really needed is a general mechanism to turn a short conditional jump into a reverse conditional jump over one of **two** possible unconditional branches, either a **brw** or a **jmp** instruction. Currently, the **–J** forces the **jmp** instruction to *always* be used, instead of the shorter **brw** instruction when needed.

The assembler should also recognize extended branch instructions for **sob**, **aob**, and **acbx** instructions. **Sob** instructions will be easy, **aob** will be harder because the synthesized instruction uses the index operand twice, so one must be careful of side effects, and the **acbx** family will be much harder (in the general case) because the comparison depends on the sign of the addend operand, and two operands are used more than once. Augmenting *as* with these extended loop instructions will allow the peephole optimizer to produce much better loop optimizations, since it currently assumes the worst case about the size of the loop body.

The string temporary file is not put in memory when the -V flag is set. The string table in the generated a.out contains some strings and names that are never referenced from the symbol table; the loader removes these unreferenced strings, however.

[3] Although the number of characters available to the *argv* line is restricted by UNIX to 10240.