UNIX Assembler Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey

0. Introduction

This document describes the usage and input syntax of the UNIX PDP-11 assembler *as*. The details of the PDP-11 are not described; consult the DEC documents "PDP-11/20 Handbook" and "PDP-11/45 Handbook."

The input syntax of the UNIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to *as*.

As is a rather ordinary two-pass assembler without macro capabilities. It produces an output file which contains relocation information and a complete symbol table; thus the output is acceptable to the UNIX link-editor *ld*, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

1. Usage

as is used as follows:

as [−] $file_1$ . . .

If the optional "−" argument is given, all undefined symbols in the current assembly will be made undefined-external. See the **.globl** directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is placed on the file *a.out* in the current directory. If there were no unresolved external references, and no errors detected, *a.out* is made executable; otherwise, if it is produced at all, it is made non-executable.

2. Lexical conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), temporary symbols, constants, and operators.

2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period "**.**", underscore "_", and tilde "~" as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place names of local variables in the output symbol table without having to worry about making them unique.

2.2 Temporary symbols

A temporary symbol consists of a digit followed by "f" or "b". Temporary symbols are discussed fully in §5.1.

2.3 Constants

An octal constant consists of a sequence of digits; "8" and "9" are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two's complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point "**.**". The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote "**′**" followed by an ASCII character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote "**"**" followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

2.4 Operators

There are several single- and double-character operators; see §6.

2.5 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.

2.6 Comments

The character "**/**" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

3. Segments

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The UNIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor *ld* (using its "**−**" flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment; if the text segment is pure, the data segment begins at the lowest 8K byte boundary after the text segment.

The bss segment may not contain any explicitly initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by

```
lab: . = .+10
```

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also *Location counter* and *Assignment statements* below.

4. The location counter

One special symbol, "**.**", is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of "**.**" may not decrease. If the effect of the assignment is to increase the value of “ `.` ”, the required number of null bytes are generated (but see *Segments* above).

5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

Any kind of statement may be preceded by one or more labels.

5.1 Labels

There are two kinds of label: name labels and numeric labels. A name 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 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 “ `.` ” to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form “*n* f ” refer to the first numeric label “*n* :” *f*orward from the reference; “*n*b” symbols refer to the first “*n* :” label *b*ackward from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol I: Fundamental Algorithms*]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

5.2 Null statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

5.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

5.4 Assignment statements

An assignment statement consists of an identifier, an equals sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter “ `.` ”. It is required, however, that the type of the expression assigned be of the same type as “ `.` ”, and it is forbidden to decrease the value of “ `.` ”. In practice, the most common assignment to “ `.` ” has the form “`.=.+`*n*” for some number *n;* this has the effect of generating *n* null bytes.

5.5 String statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote “<” followed by a sequence of ASCII characters not including newline, followed by a right string quote “>”. Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

\n	NL	(012)
\t	HT	(011)
\e	EOT	(004)
\0	NUL	(000)
\r	CR	(015)
\a	ACK	(006)
\p	PFX	(033)
\\	\	
\>	>	

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see §2.3 above).

5.6 Keyword statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

6. Expressions

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

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two's complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

6.1 Expression operators

The operators are:

(blank) — when there is no operator between operands, the effect is exactly the same as if a "+" had appeared.

+ — addition

− — subtraction

* — multiplication

\/ — division (note that plain " / " starts a comment)

& — bitwise **and**

| — bitwise **or**

>> — logical right shift

<< — logical left shift

% — modulo

! — *a* ! *b* is *a* **or** (**not** *b*); i.e., the **or** of the first operand and the one's complement of the second; most common use is as a unary.

^ — result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets " [ ] ". (Round parentheses are reserved for address modes.)

6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. 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 one 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. After the first **.bss** statement, the value of "**.**" is bss 0.

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 . . . **r5**
fr0 . . . **fr5**
sp
pc

are predefined as register symbols. Either they or symbols defined from them must be used to refer to the six general-purpose, six floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names; the former are provided as a mnemonic aid.

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: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the ‘‘other types’’ mentioned above, or with a register expression, the result has the register or other type. As a consequence, one can refer to r3 as ‘‘r0+3’’. If two operands of ‘‘other type’’ are combined, the result has the numerically larger type (not that this fact is very useful, since the values are not made public). An ‘‘other type’’ combined with an explicitly discussed type other than absolute 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.

ˆ This operator follows no other rule than that the result has the value of the first operand and the type of the second.

others It is illegal to apply these operators to any but absolute symbols.

7. Pseudo-operations

The keywords listed below introduce statements which generate data in unusual forms or influence the later operations of the assembler. The metanotation

[stuff] . . .

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

7.1 **.byte** *expression* [**,** *expression*] . . .

The *expression*s in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones which assemble data one byte at at time.

7.2 **.even**

If the location counter ‘‘ **.** ’’ is odd, it is advanced by one so the next statement will be assembled at a word boundary.

7.3 **.if** *expression*

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the **.if** is ignored; if zero, the statements between the **.if** and the matching **.endif** (below) are ignored. **.if** may be nested. The effect of **.if** cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: **.if**s and **.endif**s are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an **.if** will show up as undefined if the symbol table is listed.)

7.4 **.endif**

This statement marks the end of a conditionally-assembled section of code. See **.if** above.

7.5 **.globl** *name* [**,** *name*] . . .

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the **.globl** statement were not given; however, the link editor *ld* may be used to combine this routine with other routines that refer these symbols.

-

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols.

As discussed in §1, it is possible to force the assembler to make all otherwise undefined symbols external.

7.6 **.text**
7.7 **.data**
7.8 **.bss**

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and ‘‘ **.** ’’ moved about by assignment.

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

Provided the *name* is not defined elsewhere, this statement is equivalent to

```
.globl  name
name = expression ^ name
```

That is, the type of *name* is ‘‘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. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

8. Machine instructions

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the 11/20 and 11/45 handbooks should be consulted on the semantics.

8.1 Sources and Destinations

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

syntax	words	mode
reg	0	0+*reg*
(*reg* **)+**	0	2+*reg*
–(*reg* **)**	0	4+*reg*
expr **(** *reg* **)**	1	6+*reg*
(*reg* **)**	0	1+*reg*
***** *reg*	0	1+*reg*
*****(** *reg* **)+**	0	3+*reg*
*****–(** *reg* **)**	0	5+*reg*
*****(** *reg* **)**	1	7+*reg*
***** *expr* **(** *reg* **)**	1	7+*reg*
expr	1	67
$ *expr*	1	27
***** *expr*	1	77
*****$** *expr*	1	37

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that ‘‘*’’ has been substituted for ‘‘@’’ and ‘‘$’’ for ‘‘#’’; the UNIX typing conventions make ‘‘@’’ and ‘‘#’’ rather inconvenient.

Notice that mode ‘‘*reg’’ is identical to ‘‘(reg)’’; that ‘‘*(reg)’’ generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form ‘‘*$expr’’ can be used, but notice that further indirection is impossible.

8.3 Simple machine instructions

The following instructions are defined as absolute symbols:

clc
clv
clz
cln
sec
sev
sez
sen

They therefore require no special syntax. The PDP-11 hardware allows more than one of the ''clear'' class, or alternatively more than one of the ''set'' class to be **or**-ed together; this may be expressed as follows:

```
clc | clv
```

8.4 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of ''**.**'' by more than 254 bytes:

br	**blos**	
bne	**bvc**	
beq	**bvs**	
bge	**bhis**	
blt	**bec**	(= **bcc**)
bgt	**bcc**	
ble	**blo**	
bpl	**bcs**	
bmi	**bes**	(= **bcs**)
bhi		

bes (''branch on error set'') and **bec** (''branch on error clear'') are intended to test the error bit returned by system calls (which is the c-bit).

8.5 Extended branch instructions

The following symbols are followed by an expression representing an address in the same segment as ''**.**''. If the target address is close enough, a branch-type instruction is generated; if the address is too far away, a **jmp** will be used.

jbr	**jlos**
jne	**jvc**
jeq	**jvs**
jge	**jhis**
jlt	**jec**
jgt	**jcc**
jle	**jlo**
jpl	**jcs**
jmi	**jes**
jhi	

jbr turns into a plain **jmp** if its target is too remote; the others (whose names are contructed by replacing the ''b'' in the branch instruction's name by ''j'') turn into the converse branch over a **jmp** to the target address.

8.6 Single operand instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in §8.1 above.

-

clr	**sbcb**
clrb	**ror**
com	**rorb**
comb	**rol**
inc	**rolb**
incb	**asr**
dec	**asrb**
decb	**asl**
neg	**aslb**
negb	**jmp**
adc	**swab**
adcb	**tst**
sbc	**tstb**

8.7 Double operand instructions

The following instructions take a general source and destination (§8.1), separated by a comma, as operands.

mov
movb
cmp
cmpb
bit
bitb
bic
bicb
bis
bisb
add
sub

8.8 Miscellaneous instructions

The following instructions have more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination (§8.1), and *expr* is an expression:

jsr	*reg,dst*	
rts	*reg*	
sys	*expr*	
ash	*src , reg*	(or, **als**)
ashc	*src , reg*	(or, **alsc**)
mul	*src , reg*	(or, **mpy**)
div	*src , reg*	(or, **dvd**)
xor	*reg , dst*	
sxt	*dst*	
mark	*expr*	
sob	*reg , expr*	

sys is another name for the **trap** instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The alternative forms for **ash**, **ashc**, **mul**, and **div** are provided to avoid conflict with EAE register names should they be needed.

The expression in **mark** must be expressible in six bits, and the expression in **sob** must be in the same segment as “**.**”, must not be external-undefined, must be less than “**.**”, and must be within 510 bytes of “**.**”.

8.9 Floating-point unit instructions

The following floating-point operations are defined, with syntax as indicated:

cfcc
setf
setd

-

seti		
setl		
clrf	*fdst*	
negf	*fdst*	
absf	*fdst*	
tstf	*fsrc*	
movf	*fsrc, freg*	(= ldf)
movf	*freg, fdst*	(= stf)
movif	*src, freg*	(= ldcif)
movfi	*freg, dst*	(= stcfi)
movof	*fsrc, freg*	(= ldcdf)
movfo	*freg, fdst*	(= stcfd)
movie	*src, freg*	(= ldexp)
movei	*freg, dst*	(= stexp)
addf	*fsrc, freg*	
subf	*fsrc, freg*	
mulf	*fsrc, freg*	
divf	*fsrc, freg*	
cmpf	*fsrc, freg*	
modf	*fsrc, freg*	
ldfps	*src*	
stfps	*dst*	
stst	*dst*	

fsrc, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0—3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

9. Other symbols

9.1 `..`

The symbol "`..`" is the *relocation counter*. Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word which refers to an absolute location, the value of "`..`" is subtracted.

Thus the value of "`..`" can be taken to mean the starting core location of the program. In UNIX systems with relocation hardware, the initial value of "`..`" is 0.

The value of "`..`" may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change "`..`" midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of "`..`".

9.2 System calls

The following absolute symbols may be used to code calls to the UNIX system (see the **sys** instruction above).

break	**nice**
chdir	**open**
chmod	**read**
chown	**seek**
close	**setuid**
creat	**signal**
exec	**stat**
exit	**stime**
fork	**stty**
fstat	**tell**
getuid	**time**
gtty	**umount**

link
makdir
mdate
mount
unlink
wait
write

Warning: the **wait** system call is not the same as the **wait** instruction, which is not defined in the assembler.

10. Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	parentheses error
]	parentheses error
>	string not terminated properly
*	indirection (*) used illegally
.	illegal assignment to “**.**”
A	error in address
B	branch address is odd or too remote
E	error in expression
F	error in local (“f” or “b”) type symbol
G	garbage (unknown) character
I	end of file inside an **.if**
M	multiply defined symbol as label
O	word quantity assembled at odd address
P	phase error—“**.**” different in pass 1 and 2
R	relocation error
U	undefined symbol
X	syntax error