7 Assembler Directives

All assembler directives have names that begin with a period ('.'). The rest of the name is letters, usually in lower case.

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See Chapter 8 [Machine Dependencies], page 51.

7.1 .abort

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells `as` to quit also. One day `.abort` will not be supported.

7.2 .ABORT

When producing COFF output, `as` accepts this directive as a synonym for '`.abort`'.

When producing `b.out` output, `as` accepts this directive, but ignores it.

7.3 .align *abs-expr*, *abs-expr*, *abs-expr*

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system. For the a29k, hppa, m68k, m88k, w65, sparc, and Hitachi SH, and i386 using ELF format, the first expression is the alignment request in bytes. For example '`.align 8`' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

For other systems, including the i386 using a.out format, it is the number of low-order zero bits the location counter must have after advancement. For example '`.align 3`' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides `.balign` and `.p2align`

directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).

7.4 `.app-file` *string*

`.app-file` (which may also be spelled '`.file`') tells `as` that we are about to start a new logical file. *string* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '"'; but if you wish to specify an empty file name is permitted, you must give the quotes–"". This statement may go away in future: it is only recognized to be compatible with old `as` programs.

7.5 `.ascii "`*string*`"...`

`.ascii` expects zero or more string literals (see Section 3.6.1.1 [Strings], page 17) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

7.6 `.asciz "`*string*`"...`

`.asciz` is just like `.ascii`, but each string is followed by a zero byte. The "z" in '`.asciz`' stands for "zero".

7.7 `.balign[wl]` *abs-expr*, *abs-expr*, *abs-expr*

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example '`.balign 8`' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two byte word value. The `.balignl` directives treats the fill pattern as a four byte longword value. For example, `.balignw 4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

7.8 .byte *expressions*

`.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

7.9 .comm *symbol , length*

`.comm` declares a common symbol named *symbol*. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If `ld` does not see a definition for the symbol–just one or more common symbols–then it will allocate *length* bytes of uninitialized memory. *length* must be an absolute expression. If `ld` sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

When using ELF, the `.comm` directive takes an optional third argument. This is the desired alignment of the symbol, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero). The alignment must be an absolute expression, and it must be a power of two. If `ld` allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, `as` will set the alignment to the largest power of two less than or equal to the size of the symbol, up to a maximum of 16.

The syntax for `.comm` differs slightly on the HPPA. The syntax is '*symbol* `.comm,` *length*'; *symbol* is optional.

7.10 .data *subsection*

`.data` tells `as` to assemble the following statements onto the end of the data subsection numbered *subsection* (which is an absolute expression). If *subsection* is omitted, it defaults to zero.

7.11 .def *name*

Begin defining debugging information for a symbol *name*; the definition extends until the `.endef` directive is encountered.

This directive is only observed when `as` is configured for COFF format output; when producing `b.out`, '`.def`' is recognized, but ignored.

7.12 .desc *symbol, abs-expression*

This directive sets the descriptor of the symbol (see Section 5.5 [Symbol Attributes], page 28) to the low 16 bits of an absolute expression.

The '`.desc`' directive is not available when `as` is configured for COFF output; it is only for `a.out` or `b.out` object format. For the sake of compatibility, `as` accepts it, but produces no output, when configured for COFF.

7.13 .dim

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs.

'`.dim`' is only meaningful when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

7.14 .double *flonums*

`.double` expects zero or more flonums, separated by commas. It assembles floating point numbers. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 [Machine Dependencies], page 51.

7.15 .eject

Force a page break at this point, when generating assembly listings.

7.16 .else

`.else` is part of the `as` support for conditional assembly; see Section 7.29 [`.if`], page 38. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.17 .endef

This directive flags the end of a symbol definition begun with `.def`.

'`.endef`' is only meaningful when generating COFF format output; if `as` is configured to generate `b.out`, it accepts this directive but ignores it.

7.18 .endif

`.endif` is part of the `as` support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See Section 7.29 [`.if`], page 38.

7.19 .equ *symbol, expression*

This directive sets the value of *symbol* to *expression*. It is synonymous with '`.set`'; see Section 7.53 [`.set`], page 46.

The syntax for `equ` on the HPPA is '*symbol* `.equ` *expression*'.

7.20 .equiv *symbol*, *expression*

The `.equiv` directive is like `.equ` and `.set`, except that the assembler will signal an error if *symbol* is already defined.

Except for the contents of the error message, this is roughly equivalent to

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

7.21 .err

If `as` assembles a `.err` directive, it will print an error message and, unless the `-Z` option was used, it will not generate an object file. This can be used to signal error an conditionally compiled code.

7.22 .extern

`.extern` is accepted in the source program—for compatibility with other assemblers—but it is ignored. `as` treats all undefined symbols as external.

7.23 .file *string*

`.file` (which may also be spelled '`.app-file`') tells `as` that we are about to start a new logical file. *string* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '`"`'; but if you wish to specify an empty file name, you must give the quotes–`""`. This statement may go away in future: it is only recognized to be compatible with old `as` programs. In some configurations of `as`, `.file` has already been removed to avoid conflicts with other assemblers. See Chapter 8 [Machine Dependencies], page 51.

7.24 .fill *repeat* , *size* , *value*

result, *size* and *value* are absolute expressions. This emits *repeat* copies of *size* bytes. *Repeat* may be zero or more. *Size* may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each *repeat* bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are *value* rendered in the byte-order of an integer on the computer `as` is assembling for. Each *size* bytes in a repetition is taken from the lowest order *size* bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

size and *value* are optional. If the second comma and *value* are absent, *value* is assumed zero. If the first comma and following tokens are absent, *size* is assumed to be 1.

7.25 `.float` *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.single`. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 [Machine Dependencies], page 51.

7.26 `.global` *symbol*, `.globl` *symbol*

`.global` makes the symbol visible to `ld`. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *symbol* takes its attributes from a symbol of the same name from another file linked into the same program.

Both spellings ('`.globl`' and '`.global`') are accepted, for compatibility with other assemblers.

On the HPPA, `.global` is not always enough to make it accessible to other partial programs. You may need the HPPA-only `.EXPORT` directive as well. See Section 8.7.5 [HPPA Assembler Directives], page 67.

7.27 `.hword` *expressions*

This expects zero or more *expressions*, and emits a 16 bit number for each.

This directive is a synonym for '`.short`'; depending on the target architecture, it may also be a synonym for '`.word`'.

7.28 `.ident`

This directive is used by some assemblers to place tags in object files. `as` simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.

7.29 `.if` *absolute expression*

`.if` marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an *absolute expression*) is non-zero. The end of the conditional section of code must be marked by `.endif` (see Section 7.18 [`.endif`], page 36); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.16 [`.else`], page 36).

The following variants of `.if` are also supported:

`.ifdef` *symbol*
: Assembles the following section of code if the specified *symbol* has been defined.

`.ifndef` *symbol*
`.ifnotdef` *symbol*
: Assembles the following section of code if the specified *symbol* has not been defined. Both spelling variants are equivalent.

7.30 `.include "`*file*`"`

This directive provides a way to include supporting files at specified points in your source program. The code from *file* is assembled as if it followed the point of the `.include`; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the '`-I`' command-line option (see Chapter 2 [Command-Line Options], page 9). Quotation marks are required around *file*.

7.31 `.int` *expressions*

Expect zero or more *expressions*, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

7.32 `.irp` *symbol*, *values*...

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the `.irp` directive, and is terminated by an `.endr` directive. For each *value*, *symbol* is set to *value*, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with *symbol* set to the null string. To refer to *symbol* within the sequence of statements, use *symbol*.

For example, assembling

```
.irp    param,1,2,3
move    d\param,sp@-
.endr
```

is equivalent to assembling

```
move    d1,sp@-
move    d2,sp@-
move    d3,sp@-
```

7.33 `.irpc` *symbol*, *values*...

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the `.irpc` directive, and is terminated by an `.endr` directive. For each character in *value*, *symbol* is set to the character, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with *symbol* set to the null string. To refer to *symbol* within the sequence of statements, use *symbol*.

For example, assembling

```
.irpc    param,123
move    d\param,sp@-
.endr
```

is equivalent to assembling

```
move    d1,sp@-
move    d2,sp@-
move    d3,sp@-
```

7.34 `.lcomm` *symbol* , *length*

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *Symbol* is not declared global (see Section 7.26 [`.global`], page 38), so is normally not visible to `ld`.

Some targets permit a third argument to be used with `.lcomm`. This argument specifies the desired alignment of the symbol in the bss section.

The syntax for `.lcomm` differs slightly on the HPPA. The syntax is '*symbol* `.lcomm,` *length*'; *symbol* is optional.

7.35 `.lflags`

`as` accepts this directive, for compatibility with other assemblers, but ignores it.

7.36 `.line` *line-number*

Change the logical line number. *line-number* must be an absolute expression. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number *line-number* − 1. One day `as` will no longer support this directive: it is recognized only for compatibility with existing assembler programs.

Warning: In the AMD29K configuration of as, this command is not available; use the synonym `.ln` in that context.

Even though this is a directive associated with the `a.out` or `b.out` object-code formats, `as` still recognizes it when producing COFF output, and treats '`.line`' as though it were the COFF '`.ln`' *if* it is found outside a `.def`/`.endef` pair.

Inside a `.def`, '`.line`' is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.

7.37 `.linkonce` [*type*]

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensure that the linker will only include it once in the final output file. The `.linkonce` pseudo-op must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique.

This directive is only supported by a few object file formats; as of this writing, the only object file format which supports it is the Portable Executable format used on Windows NT.

The *type* argument is optional. If specified, it must be one of the following strings. For example:

```
.linkonce same_size
```

Not all types may be supported on all object file formats.

`discard` Silently discard duplicate sections. This is the default.

`one_only` Warn if there are duplicate sections, but still keep only one copy.

`same_size`
Warn if any of the duplicates have different sizes.

`same_contents`
Warn if any of the duplicates do not have exactly the same contents.

7.38 `.ln` *line-number*

'`.ln`' is a synonym for '`.line`'.

7.39 `.mri` *val*

If *val* is non-zero, this tells `as` to enter MRI mode. If *val* is zero, this tells `as` to exit MRI mode. This change affects code assembled until the next `.mri` directive, or until the end of the file. See Section 2.7 [MRI mode], page 10.

7.40 `.list`

Control (in conjunction with the `.nolist` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled. When you enable them (with the '`-a`' command line option; see Chapter 2 [Command-Line Options], page 9), the initial value of the listing counter is one.

7.41 `.long` *expressions*

`.long` is the same as '`.int`', see Section 7.31 [`.int`], page 39.

7.42 `.macro`

The commands `.macro` and `.endm` allow you to define macros that generate assembly output. For example, this definition specifies a macro `sum` that puts a sequence of numbers into memory:

```
.macro  sum from=0, to=5
.long   \from
.if     \to-\from
sum     "(\from+1)",\to
.endif
.endm
```

With that definition, '`SUM 0,5`' is equivalent to this assembly input:

```
        .long   0
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
```

`.macro` *macname*
`.macro` *macname macargs ...*

Begin the definition of a macro called *macname*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with '`=`*deflt*'. For example, these are all valid `.macro` statements:

`.macro comm`

Begin the definition of a macro called `comm`, which takes no arguments.

`.macro plus1 p, p1`
`.macro plus1 p p1`

Either statement begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write '`\p`' or '`\p1`' to evaluate the arguments.

`.macro reserve_str p1=0 p2`

Begin the definition of a macro called `reserve_str`, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as '`reserve_str` *a*`,`*b*' (with '`\p1`' evaluating to *a* and '`\p2`' evaluating to *b*), or as '`reserve_str ,`*b*' (with '`\p1`' evaluating as the default, in this case '`0`', and '`\p2`' evaluating to *b*).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, '`sum 9,17`' is equivalent to '`sum to=17, from=9`'.

`.endm` Mark the end of a macro definition.

`.exitm` Exit early from the current macro definition.

`\@` `as` maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with '`\@`', but *only within a macro definition*.

7.43 `.nolist`

Control (in conjunction with the `.list` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

7.44 `.octa` *bignums*

This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer.

The term "octa" comes from contexts in which a "word" is two bytes; hence *octa*-word for 16 bytes.

7.45 `.org` *new-lc* , *fill*

Advance the location counter of the current section to *new-lc*. *new-lc* is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use `.org` to cross sections: if *new-lc* has the wrong section, the `.org` directive is ignored. To be compatible with former assemblers, if the section of *new-lc* is absolute, `as` issues a warning, then pretends the section of *new-lc* is the same as the current subsection.

`.org` may only increase the location counter, or leave it unchanged; you cannot use `.org` to move the location counter backwards.

Because `as` tries to assemble programs in one pass, *new-lc* may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler.

Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with *fill* which should be an absolute expression. If the comma and *fill* are omitted, *fill* defaults to zero.

7.46 `.p2align[wl]` *abs-expr*, *abs-expr*, *abs-expr*

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example '`.p2align 3`' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example, `.p2alignw`

`2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

7.47 `.psize` *lines , columns*

Use this directive to declare the number of lines—and, optionally, the number of columns—to use for each page, when generating listings.

If you do not use `.psize`, listings use a default line-count of 60. You may omit the comma and *columns* specification; the default width is 200 columns.

`as` generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

If you specify *lines* as `0`, no formfeeds are generated save those explicitly specified with `.eject`.

7.48 `.quad` *bignums*

`.quad` expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum.

The term "quad" comes from contexts in which a "word" is two bytes; hence *quad*-word for 8 bytes.

7.49 `.rept` *count*

Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive *count* times.

For example, assembling

```
.rept   3
.long   0
.endr
```

is equivalent to assembling

```
.long   0
.long   0
.long   0
```

7.50 `.sbttl "`*subheading*`"`

Use *subheading* as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

7.51 `.scl` *class*

Set the storage-class value for a symbol. This directive may only be used inside a `.def`/`.endef` pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

The '`.scl`' directive is primarily associated with COFF output; when configured to generate `b.out` output format, `as` accepts this directive but ignores it.

7.52 `.section` *name*

Use the `.section` directive to assemble the following code into a section named *name*.

This directive is only supported for targets that actually support arbitrarily named sections; on `a.out` targets, for example, it is not accepted, even with a standard `a.out` section name.

For COFF targets, the `.section` directive is used in one of the following ways:

```
.section name[, "flags"]
.section name[, subsegment]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

`b` bss section (uninitialized data)

`n` section is not loaded

`w` writable section

`d` data section

`r` read-only section

`x` executable section

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable.

If the optional argument to the `.section` directive is not quoted, it is taken as a subsegment number (see Section 4.4 [Sub-Sections], page 23).

For ELF targets, the `.section` directive is used like this:

```
.section name[, "flags"[, @type]]
```

The optional *flags* argument is a quoted string which may contain any combintion of the following characters:

`a` section is allocatable

`w` section is writable

`x` section is executable

The optional *type* argument may contain one of the following constants:

`@progbits`
section contains data

`@nobits` section does not contain data (i.e., section only occupies space)

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to have none of the above flags: it will not be allocated in memory, nor writable, nor executable. The section will contain data.

For ELF targets, the assembler supports another type of `.section` directive for compatibility with the Solaris assembler:

```
.section "name"[, flags...]
```

Note that the section name is quoted. There may be a sequence of comma separated flags:

`#alloc` section is allocatable

`#write` section is writable

`#execinstr`
section is executable

7.53 `.set` *symbol, expression*

Set the value of *symbol* to *expression*. This changes *symbol*'s value and type to conform to *expression*. If *symbol* was flagged as external, it remains flagged (see Section 5.5 [Symbol Attributes], page 28).

You may `.set` a symbol many times in the same assembly.

If you `.set` a global symbol, the value stored in the object file is the last value stored into it.

The syntax for `set` on the HPPA is '*symbol* `.set` *expression*'.

7.54 `.short` *expressions*

`.short` is normally the same as '`.word`'. See Section 7.69 [`.word`], page 50.

In some configurations, however, `.short` and `.word` generate numbers of different lengths; see Chapter 8 [Machine Dependencies], page 51.

7.55 `.single` *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.float`. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 [Machine Dependencies], page 51.

7.56 `.size`

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs.

'`.size`' is only meaningful when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

7.57 `.sleb128` *expressions*

sleb128 stands for "signed little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See Section 7.68 [Uleb128], page 49.

7.58 `.skip` *size* , *fill*

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. This is the same as '`.space`'.

7.59 `.space` *size* , *fill*

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. This is the same as '`.skip`'.

> *Warning:* `.space` has a completely different meaning for HPPA targets; use `.block` as a substitute. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for the meaning of the `.space` directive. See Section 8.7.5 [HPPA Assembler Directives], page 67, for a summary.

On the AMD 29K, this directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

> *Warning:* In most versions of the GNU assembler, the directive `.space` has the effect of `.block` See Chapter 8 [Machine Dependencies], page 51.

7.60 `.stabd, .stabn, .stabs`

There are three directives that begin '`.stab`'. All emit symbols (see Chapter 5 [Symbols], page 27), for use by symbolic debuggers. The symbols are not entered in the `as` hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

string This is the symbol's name. It may contain any character except '`\000`', so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.

type An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but `ld` and debuggers choke on silly bit patterns.

other An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.

desc An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.

value An absolute expression which becomes the symbol's value.

If a warning is detected while reading a `.stabd`, `.stabn`, or `.stabs` statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

`.stabd` *type* , *other* , *desc*
: The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

  The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the `.stabd` was assembled.

`.stabn` *type* , *other* , *desc* , *value*
: The name of the symbol is set to the empty string `""`.

`.stabs` *string* , *type* , *other* , *desc* , *value*
: All five fields are specified.

7.61 `.string "str"`

Copy the characters in *str* to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in Section 3.6.1.1 [Strings], page 17.

7.62 `.symver`

Use the `.symver` directive to bind symbols to specific version nodes within a source file. This is only supported on ELF platforms, and is typically used when assembling files to be linked into a shared library. There are cases where it may make sense to use this in objects to be bound into an application itself so as to override a versioned symbol from a shared library.

For ELF targets, the `.symver` directive is used like this:

```
.symver name, name2@nodename
```

In this case, the symbol *name* must exist and be defined within the file being assembled. The `.versym` directive effectively creates a symbol alias with the name *name2@nodename*, and in fact the main reason that we just don't try and create a regular alias is that the `@` character isn't permitted in symbol names. The *name2* part of the name is the actual name of the symbol by which it will be externally referenced. The name *name* itself is merely a name of convenience that is used so that it is possible to have definitions for multiple versions of a function within a single source file, and so that the compiler can unambiguously know which version of a function is being mentioned. The *nodename* portion of the alias should be the name of a node specified in the version script supplied to the linker when building a shared library. If you are attempting to override a versioned symbol from a shared library, then *nodename* should correspond to the nodename of the symbol you are trying to override.

7.63 `.tag` *structname*

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs. Tags are used to link structure definitions in the symbol table with instances of those structures.

'`.tag`' is only used when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

7.64 `.text` *subsection*

Tells `as` to assemble the following statements onto the end of the text subsection numbered *subsection*, which is an absolute expression. If *subsection* is omitted, subsection number zero is used.

7.65 `.title` "*heading*"

Use *heading* as the title (second line, immediately after the source file name and pagenumber) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

7.66 `.type` *int*

This directive, permitted only within `.def`/`.endef` pairs, records the integer *int* as the type attribute of a symbol table entry.

'`.type`' is associated only with COFF format output; when `as` is configured for `b.out` output, it accepts this directive but ignores it.

7.67 `.val` *addr*

This directive, permitted only within `.def`/`.endef` pairs, records the address *addr* as the value attribute of a symbol table entry.

'`.val`' is used only for COFF output; when `as` is configured for `b.out`, it accepts this directive but ignores it.

7.68 `.uleb128` *expressions*

uleb128 stands for "unsigned little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See Section 7.57 [Sleb128], page 47.

7.69 `.word` *expressions*

This directive expects zero or more *expressions*, of any section, separated by commas.

The size of the number emitted, and its byte order, depend on what target computer the assembly is for.

Warning: Special Treatment to support Compilers

Machines with a 32-bit address space, but that do less than 32-bit addressing, require the following special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it; see Chapter 8 [Machine Dependencies], page 51), you can ignore this issue.

In order to assemble compiler output into something that works, `as` occasionlly does strange things to '`.word`' directives. Directives of the form '`.word sym1-sym2`' are often emitted by compilers as part of jump tables. Therefore, when `as` assembles a directive of the form '`.word sym1-sym2`', and the difference between `sym1` and `sym2` does not fit in 16 bits, `as` creates a *secondary jump table*, immediately before the next label. This secondary jump table is preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table is a long-jump to `sym2`. The original '`.word`' contains `sym1` minus the address of the long-jump to `sym2`.

If there were several occurrences of '`.word sym1-sym2`' before the secondary jump table, all of them are adjusted. If there was a '`.word sym3-sym4`', that also did not fit in sixteen bits, a long-jump to `sym4` is included in the secondary jump table, and the `.word` directives are adjusted to contain `sym3` minus the address of the long-jump to `sym4`; and so on, for as many entries in the original jump table as necessary.

7.70 Deprecated Directives

One day these directives won't work. They are included for compatibility with older assemblers.

`.abort`

`.app-file`

`.line`

8 Machine Dependent Features

The machine instruction sets are (almost by definition) different on each machine where `as` runs. Floating point representations vary as well, and `as` often supports a few additional directives or command-line options for compatibility with other assemblers on a particular platform. Finally, some versions of `as` support special pseudo-instructions for branch optimization.

This chapter discusses most of these differences, though it does not include details on any machine's instruction set. For details on that subject, see the hardware manufacturer's manual.

8.10 M680x0 Dependent Features

8.10.1 M680x0 Options

The Motorola 680x0 version of `as` has a few machine dependent options.

You can use the '`-l`' option to shorten the size of references to undefined symbols. If you do not use the '`-l`' option, references to undefined symbols are wide enough for a full `long` (32 bits). (Since `as` cannot know where these symbols end up, `as` can only allocate space for the linker to fill in later. Since `as` does not know how far away these symbols are, it allocates as much space as it can.) If you use this option, the references are only one word wide (16 bits). This may be useful if you want the object file to be as small as possible, and you know that the relevant symbols are always less than 17 bits away.

For some configurations, especially those where the compiler normally does not prepend an underscore to the names of user variables, the assembler requires a '`%`' before any use of a register name. This is intended to let the assembler distinguish between C variables and functions named '`a0`' through '`a7`', and so on. The '`%`' is always accepted, but is not required for certain configurations, notably '`sun3`'. The '`--register-prefix-optional`' option may be used to permit omitting the '`%`' even for configurations for which it is normally required. If this is done, it will generally be impossible to refer to C variables and functions with the same names as register names.

Normally the character '`|`' is treated as a comment character, which means that it can not be used in expressions. The '`--bitwise-or`' option turns '`|`' into a normal character. In this mode, you must either use C style comments, or start comments with a '`#`' character at the beginning of a line.

If you use an addressing mode with a base register without specifying the size, `as` will normally use the full 32 bit value. For example, the addressing mode '`%a0@(%d0)`' is equivalent to '`%a0@(%d0:l)`'. You may use the '`--base-size-default-16`' option to tell `as` to default to using the 16 bit value. In this case, '`%a0@(%d0)`' is equivalent to '`%a0@(%d0:w)`'. You may use the '`--base-size-default-32`' option to restore the default behaviour.

If you use an addressing mode with a displacement, and the value of the displacement is not known, `as` will normally assume that the value is 32 bits. For example, if the symbol '`disp`' has not been defined, `as` will assemble the addressing mode '`%a0@(disp,%d0)`' as though '`disp`' is a 32 bit value. You may use the '`--disp-size-default-16`' option to tell `as` to instead assume that the displacement is 16 bits. In this case, `as` will assemble '`%a0@(disp,%d0)`' as though '`disp`' is a 16 bit value. You may use the '`--disp-size-default-32`' option to restore the default behaviour.

`as` can assemble code for several different members of the Motorola 680x0 family. The default depends upon how `as` was configured when it was built; normally, the default is to assemble code for the 68020 microprocessor. The following options may be used to change the default. These options control which instructions and addressing modes are permitted. The members of the 680x0 family are very similar. For detailed information about the differences, see the Motorola manuals.

'-m68000'
'-m68ec000'
'-m68hc000'
'-m68hc001'
'-m68008'
'-m68302'
'-m68306'
'-m68307'
'-m68322'
'-m68356' Assemble for the 68000. '-m68008', '-m68302', and so on are synonyms for '-m68000', since the chips are the same from the point of view of the assembler.

'-m68010' Assemble for the 68010.

'-m68020'
'-m68ec020'
Assemble for the 68020. This is normally the default.

'-m68030'
'-m68ec030'
Assemble for the 68030.

'-m68040'
'-m68ec040'
Assemble for the 68040.

'-m68060'
'-m68ec060'
Assemble for the 68060.

'-mcpu32'
'-m68330'
'-m68331'
'-m68332'
'-m68333'
'-m68334'
'-m68336'
'-m68340'
'-m68341'
'-m68349'
'-m68360' Assemble for the CPU32 family of chips.

'-m5200' Assemble for the ColdFire family of chips.

'-m68881'
'-m68882' Assemble 68881 floating point instructions. This is the default for the 68020, 68030, and the CPU32. The 68040 and 68060 always support floating point instructions.

'`-mno-68881`'
: Do not assemble 68881 floating point instructions. This is the default for 68000 and the 68010. The 68040 and 68060 always support floating point instructions, even if this option is used.

'`-m68851`'
: Assemble 68851 MMU instructions. This is the default for the 68020, 68030, and 68060. The 68040 accepts a somewhat different set of MMU instructions; '`-m68851`' and '`-m68040`' should not be used together.

'`-mno-68851`'
: Do not assemble 68851 MMU instructions. This is the default for the 68000, 68010, and the CPU32. The 68040 accepts a somewhat different set of MMU instructions.

8.10.2 Syntax

This syntax for the Motorola 680x0 was developed at MIT.

The 680x0 version of `as` uses instructions names and syntax compatible with the Sun assembler. Intervening periods are ignored; for example, '`movl`' is equivalent to '`mov.l`'.

In the following table *apc* stands for any of the address registers ('`%a0`' through '`%a7`'), the program counter ('`%pc`'), the zero-address relative to the program counter ('`%zpc`'), a suppressed address register ('`%za0`' through '`%za7`'), or it may be omitted entirely. The use of *size* means one of '`w`' or '`l`', and it may be omitted, along with the leading colon, unless a scale is also specified. The use of *scale* means one of '`1`', '`2`', '`4`', or '`8`', and it may always be omitted along with the leading colon.

The following addressing modes are understood:

Immediate
: '`#`*number*'

Data Register
: '`%d0`' through '`%d7`'

Address Register
: '`%a0`' through '`%a7`'
'`%a7`' is also known as '`%sp`', i.e. the Stack Pointer. `%a6` is also known as '`%fp`', the Frame Pointer.

Address Register Indirect
: '`%a0@`' through '`%a7@`'

Address Register Postincrement
: '`%a0@+`' through '`%a7@+`'

Address Register Predecrement
: '`%a0@-`' through '`%a7@-`'

Indirect Plus Offset
: '*apc*`@(`*number*`)`'

Index
: '*apc*`@(`*number*`,`*register*`:`*size*`:`*scale*`)`'
The *number* may be omitted.

Postindex '*apc*`@(`*number*`)@(`*onumber*`,`*register*`:`*size*`:`*scale*`)`'

The *onumber* or the *register*, but not both, may be omitted.

Preindex '*apc*`@(`*number*`,`*register*`:`*size*`:`*scale*`)@(`*onumber*`)`'

The *number* may be omitted. Omitting the *register* produces the Postindex addressing mode.

Absolute '*symbol*', or '*digits*', optionally followed by '`:b`', '`:w`', or '`:l`'.

8.10.3 Motorola Syntax

The standard Motorola syntax for this chip differs from the syntax already discussed (see Section 8.10.2 [Syntax], page 81). `as` can accept Motorola syntax for operands, even if MIT syntax is used for other operands in the same instruction. The two kinds of syntax are fully compatible.

In the following table *apc* stands for any of the address registers ('`%a0`' through '`%a7`'), the program counter ('`%pc`'), the zero-address relative to the program counter ('`%zpc`'), or a suppressed address register ('`%za0`' through '`%za7`'). The use of *size* means one of '`w`' or '`l`', and it may always be omitted along with the leading dot. The use of *scale* means one of '`1`', '`2`', '`4`', or '`8`', and it may always be omitted along with the leading asterisk.

The following additional addressing modes are understood:

Address Register Indirect

'`(%a0)`' through '`(%a7)`'

'`%a7`' is also known as '`%sp`', i.e. the Stack Pointer. `%a6` is also known as '`%fp`', the Frame Pointer.

Address Register Postincrement

'`(%a0)+`' through '`(%a7)+`'

Address Register Predecrement

'`-(%a0)`' through '`-(%a7)`'

Indirect Plus Offset

'*number*`(%a0)`' through '*number*`(%a7)`', or '*number*`(%pc)`'.

The *number* may also appear within the parentheses, as in '`(`*number*`,%a0)`'. When used with the *pc*, the *number* may be omitted (with an address register, omitting the *number* produces Address Register Indirect mode).

Index '*number*`(`*apc*`,`*register*`.`*size*`*`*scale*`)`'

The *number* may be omitted, or it may appear within the parentheses. The *apc* may be omitted. The *register* and the *apc* may appear in either order. If both *apc* and *register* are address registers, and the *size* and *scale* are omitted, then the first register is taken as the base register, and the second as the index register.

Postindex '`([`*number*`,`*apc*`],`*register*`.`*size*`*`*scale*`,`*onumber*`)`'

The *onumber*, or the *register*, or both, may be omitted. Either the *number* or the *apc* may be omitted, but not both.

Preindex '`([`*number*`,`*apc*`,`*register*`.`*size*`*`*scale*`],`*onumber*`)`'

The *number*, or the *apc*, or the *register*, or any two of them, may be omitted. The *onumber* may be omitted. The *register* and the *apc* may appear in either order. If both *apc* and *register* are address registers, and the *size* and *scale* are omitted, then the first register is taken as the base register, and the second as the index register.

8.10.4 Floating Point

Packed decimal (P) format floating literals are not supported. Feel free to add the code!

The floating point formats generated by directives are these.

`.float` `Single` precision floating point constants.

`.double` `Double` precision floating point constants.

`.extend`
`.ldouble` `Extended` precision (`long double`) floating point constants.

8.10.5 680x0 Machine Directives

In order to be compatible with the Sun assembler the 680x0 assembler understands the following directives.

`.data1` This directive is identical to a `.data 1` directive.

`.data2` This directive is identical to a `.data 2` directive.

`.even` This directive is a special case of the `.align` directive; it aligns the output to an even byte boundary.

`.skip` This directive is identical to a `.space` directive.

8.10.6 Opcodes

8.10.6.1 Branch Improvement

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that reach the target. Generally these mnemonics are made by substituting '`j`' for '`b`' at the start of a Motorola mnemonic.

The following table summarizes the pseudo-operations. A `*` flags cases that are more fully described after the table:

```
             Displacement
             +-------------------------------------------------
             |                68020   68000/10
Pseudo-Op    |BYTE    WORD    LONG    LONG      non-PC relative
             +-------------------------------------------------
     jbsr    |bsrs    bsr     bsrl    jsr       jsr
      jra    |bras    bra     bral    jmp       jmp
```

```
*     jXX |bXXs    bXX     bXXl    bNXs;jmpl bNXs;jmp
*    dbXX |dbXX    dbXX        dbXX; bra; jmpl
*    fjXX |fbXXw   fbXXw   fbXXl             fbNXw;jmp

XX: condition
NX: negative of condition XX
```

*—see full description below

`jbsr`
`jra` These are the simplest jump pseudo-operations; they always map to one particular machine instruction, depending on the displacement to the branch target.

`j`*XX* Here, '`j`*XX*' stands for an entire family of pseudo-operations, where *XX* is a conditional branch or condition-code test. The full list of pseudo-ops in this family is:

```
jhi   jls   jcc   jcs   jne   jeq   jvc
jvs   jpl   jmi   jge   jlt   jgt   jle
```

For the cases of non-PC relative displacements and long displacements on the 68000 or 68010, `as` issues a longer code fragment in terms of *NX*, the opposite condition to *XX*. For example, for the non-PC relative case:

```
        jXX foo
```

gives

```
        bNXs oof
        jmp foo
oof:
```

`db`*XX* The full family of pseudo-operations covered here is

```
dbhi   dbls   dbcc   dbcs   dbne   dbeq   dbvc
dbvs   dbpl   dbmi   dbge   dblt   dbgt   dble
dbf    dbra   dbt
```

Other than for word and byte displacements, when the source reads '`db`*XX* `foo`', `as` emits

```
        dbXX oo1
        bra oo2
oo1:jmpl foo
oo2:
```

`fj`*XX* This family includes

```
fjne   fjeq   fjge   fjlt   fjgt   fjle   fjf
fjt    fjgl   fjgle  fjnge  fjngl  fjngle fjngt
fjnle  fjnlt  fjoge  fjogl  fjogt  fjole  fjolt
fjor   fjseq  fjsf   fjsne  fjst   fjueq  fjuge
fjugt  fjule  fjult  fjun
```

For branch targets that are not PC relative, `as` emits

```
        fbNX oof
        jmp foo
```

`oof:`

when it encounters '`fj`*XX* `foo`'.

8.10.6.2 Special Characters

The immediate character is '`#`' for Sun compatibility. The line-comment character is '`|`' (unless the '`--bitwise-or`' option is used). If a '`#`' appears at the beginning of a line, it is treated as a comment unless it looks like '`# line file`', in which case it is treated normally.