Red Hat Enterprise Linux 3

Using as, the Gnu Assembler

Red Hat Enterprise Linux 3: Using as, the Gnu Assembler

This documentation has been prepared by Red Hat, Inc.

Red Hat, Inc.
1801 Varsity Drive
Raleigh NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

Manual identifier:

- PDF: rhds-as-EN-3-PDF-RHI (2003-09-24T01:08)
- HTML: rhds-as-EN-3-HTML-RHI (2003-09-24T01:08)

HTML, PDF, and RPM versions of the manuals are available on the Documentation CD and online at http://www.redhat.com/docs/.
The GPG fingerprint of the security@redhat.com key is:
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Table of Contents

Chapter 1.

Using as

This file is a user guide to the gnu assembler `as` version 2.14.90.0.4.

Chapter 2.
Overview

Here is a brief summary of how to invoke `as`. For details, refer to Chapter 3 *Command-Line Options*.

```
as [-a[cdhlns][=file]] [-D] [--defsym sym=val]
 [-f] [--gstabs] [--gdwarf2] [--help] [-I dir]
 [-J] [-K] [-L]
 [--listing-lhs-width=NUM] [--listing-lhs-width2=NUM]
 [--listing-rhs-width=NUM] [--listing-cont-lines=NUM]
 [--keep-locals] [-o objfile] [-R] [--statistics] [-v]
 [-version] [--version] [-W] [--warn] [--fatal-warnings]
 [-w] [-x] [-Z] [--target-help] [target-options]
 [--|files ...]

Target Alpha options:
   [-mcpu]
   [-mdebug | -no-mdebug]
   [-relax] [-g] [-Gsize]
   [-F] [-32addr]

Target ARC options:
   [-marc[5|6|7|8]]
   [-EB|-EL]

Target ARM options:
   [-mcpu=processor[+extension...]]
   [-march=architecture[+extension...]]
   [-mfpu=floating-point-fromat]
   [-mthumb]
   [-EB|-EL]
   [-mapcs-32|-mapcs-26|-mapcs-float|
    -mapcs-reentrant]
   [-mthumb-interwork] [-moabi] [-k]

Target CRIS options:
   [--underscore | --no-underscore]
   [--pic] [-N]
   [--emulation=criself | --emulation=crisaout]

Target D10V options:
   [-O]

Target D30V options:
   [-O|-n|-N]

Target i386 options:
   [--32|--64]

Target i960 options:
   [-ACA|-ACA_A|-ACB|-ACC|-AKA|-AKB|
    -AKC|-AMC]
   [-b] [-no-relax]

Target IP2K options:
   [-mip2022|-mip2022ext]

Target M32R options:
   [--m32rx|--[no-]warn-explicit-parallel-conflicts|
```

```
   --W[n]p]

Target M680X0 options:
   [-l] [-m68000|-m68010|-m68020|...]

Target M68HC11 options:
   [-m68hc11|-m68hc12|-m68hcs12]
   [-mshort|-mlong]
   [-mshort-double|-mlong-double]
   [--force-long-branchs] [--short-branchs]
   [--strict-direct-mode] [--print-insn-syntax]
   [--print-opcodes] [--generate-example]

Target MCORE options:
   [-jsri2bsr] [-sifilter] [-relax]
   [-mcpu=[210|340]]

Target MIPS options:
   [-nocpp] [-EL] [-EB] [-n] [-O[optimization level]]
   [-g[debug level]] [-G num] [-KPIC] [-call_shared]
   [-non_shared] [-xgot] [--membedded-pic]
   [-mabi=ABI] [-32] [-n32] [-64] [-mfp32] [-mgp32]
   [-march=CPU] [-mtune=CPU] [-mips1] [-mips2]
   [-mips3] [-mips4] [-mips5] [-mips32] [-mips32r2]
   [-mips64]
   [-construct-floats] [-no-construct-floats]
   [-trap] [-no-break] [-break] [-no-trap]
   [-mfix7000] [-mno-fix7000]
   [-mips16] [-no-mips16]
   [-mips3d] [-no-mips3d]
   [-mdmx] [-no-mdmx]
   [-mdebug] [-no-mdebug]

Target MMIX options:
   [--fixed-special-register-names] [--globalize-symbols]
   [--gnu-syntax] [--relax] [--no-predefined-symbols]
   [--no-expand] [--no-merge-gregs] [-x]
   [--linker-allocated-gregs]

Target PDP11 options:
   [-mpic|-mno-pic] [-mall] [-mno-extensions]
   [-mextension|-mno-extension]
   [-mcpu] [-mmachine]

Target picoJava options:
   [-mb|-me]

Target PowerPC options:
   [-mpwrx|-mpwr2|-mpwr|-m601|-mppc|-mppc32|-m603|-m604|
    -m403|-m405|-mppc64|-m620|-mppc64bridge|-mbooke|
    -mbooke32|-mbooke64]
   [-mcom|-many|-maltivec] [-memb]
   [-mregnames|-mno-regnames]
   [-mrelocatable|-mrelocatable-lib]
   [-mlittle|-mlittle-endian|-mbig|-mbig-endian]
   [-msolaris|-mno-solaris]

Target SPARC options:
   [-Av6|-Av7|-Av8|-Asparclet|-Asparclite
    -Av8plus|-Av8plusa|-Av9|-Av9a]
   [-xarch=v8plus|-xarch=v8plusa] [-bump]
   [-32|-64]
```

```
Target TIC54X options:
 [-mcpu=54[123589]|-mcpu=54[56]lp] [-mfar-mode|-mf]
 [-merrors-to-file <filename>|-me <filename>]

Target Xtensa options:
 [--[no-]density] [--[no-]relax] [--[no-]generics]
 [--[no-]text-section-literals]
 [--[no-]target-align] [--[no-]longcalls]
```

`-a[cdhlmns]`

Turn on listings, in any of a variety of ways:

`-ac`

omit false conditionals

`-ad`

omit debugging directives

`-ah`

include high-level source

`-al`

include assembly

`-am`

include macro expansions

`-an`

omit forms processing

`-as`

include symbols

`=file`

set the name of the listing file

You may combine these options; for example, use `-aln` for assembly listing without forms processing. The `=file` option, if used, must be the last one. By itself, `-a` defaults to `-ahls`.

`-D`

Ignored. This option is accepted for script compatibility with calls to other assemblers.

`-defsym sym=value`

Define the symbol `sym` to be `value` before assembling the input file. `value` must be an integer constant. As in C, a leading `0x` indicates a hexadecimal value, and a leading `0` indicates an octal value.

`-f`

"fast"--skip whitespace and comment preprocessing (assume source is compiler output).

`-gstabs`

Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.

`-gdwarf2`

Generate DWARF2 debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it. Note--this option is only supported by some targets, not all of them.

`-help`

Print a summary of the command line options and exit.

`-target-help`

Print a summary of all target specific options and exit.

`-I dir`

Add directory `dir` to the search list for `.include` directives.

`-J`

Don't warn about signed overflow.

`-K`

Issue warnings when difference tables altered for long displacements.

`-L`
`-keep-locals`

Keep (in the symbol table) local symbols. On traditional a.out systems these start with `L`, but different systems have different local label prefixes.

`-listing-lhs-width=number`

Set the maximum width, in words, of the output data column for an assembler listing to `number`.

`-listing-lhs-width2=number`

Set the maximum width, in words, of the output data column for continuation lines in an assembler listing to `number`.

`-listing-rhs-width=number`

Set the maximum width of an input source line, as displayed in a listing, to `number` bytes.

`-listing-cont-lines=number`

Set the maximum number of lines printed in a listing for a single line of input to `number` + 1.

`-o objfile`

Name the object-file output from `as objfile`.

`-R`

Fold the data section into the text section.

`-statistics`

Print the maximum space (in bytes) and total time (in seconds) used by assembly.

`-strip-local-absolute`

Remove local absolute symbols from the outgoing symbol table.

`-v`
`-version`

Print the `as` version.

`-version`

Print the `as` version and exit.

`-W`
`-no-warn`

Suppress warning messages.

`-fatal-warnings`

Treat warnings as errors.

`-warn`

Don't suppress warning messages or treat them as errors.

`-w`

Ignored.

`-x`

Ignored.

`-Z`

Generate an object file even after errors.

`- | files ...`

Standard input, or source files to assemble.

The following options are available when as is configured for an ARC processor.

`-marc[5|6|7|8]`

This option selects the core processor variant.

`-EB | -EL`

Select either big-endian (-EB) or little-endian (-EL) output.

The following options are available when as is configured for the ARM processor family.

`-mcpu=processor[+extension...]`

Specify which ARM processor variant is the target.

`-march=architecture[+extension...]`

Specify which ARM architecture variant is used by the target.

`-mfpu=floating-point-format`

Select which Floating Point architecture is the target.

`-mthumb`

Enable Thumb only instruction decoding.

`-mapcs-32 | -mapcs-26 | -mapcs-float | -mapcs-reentrant | -moabi`

Select which procedure calling convention is in use.

`-EB | -EL`

Select either big-endian (-EB) or little-endian (-EL) output.

`-mthumb-interwork`

Specify that the code has been generated with interworking between Thumb and ARM code in mind.

`-k`

Specify that PIC code has been generated.

See the info pages for documentation of the CRIS-specific options.

The following options are available when as is configured for a D10V processor.

`-O`

Optimize output by parallelizing instructions.

The following options are available when as is configured for a D30V processor.

`-O`

Optimize output by parallelizing instructions.

`-n`

Warn when nops are generated.

`-N`

Warn when a nop after a 32-bit multiply instruction is generated.

The following options are available when as is configured for the Intel 80960 processor.

`-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC`

Specify which variant of the 960 architecture is the target.

`-b`

Add code to collect statistics about branches taken.

`-no-relax`

Do not alter compare-and-branch instructions for long displacements; error if necessary.

The following options are available when as is configured for the Ubicom IP2K series.

`-mip2022ext`

Specifies that the extended IP2022 instructions are allowed.

`-mip2022`

Restores the default behaviour, which restricts the permitted instructions to just the basic IP2022 ones.

The following options are available when as is configured for the Renesas M32R (formerly Mitsubishi M32R) series.

`-m32rx`

Specify which processor in the M32R family is the target. The default is normally the M32R, but this option changes it to the M32RX.

`-warn-explicit-parallel-conflicts or -Wp`

Produce warning messages when questionable parallel constructs are encountered.

`-no-warn-explicit-parallel-conflicts or -Wnp`

Do not produce warning messages when questionable parallel constructs are encountered.

The following options are available when as is configured for the Motorola 68000 series.

`-l`

Shorten references to undefined symbols, to one word instead of two.

`-m68000 | -m68008 | -m68010 | -m68020 | -m68030`
`| -m68040 | -m68060 | -m68302 | -m68331 | -m68332`
`| -m68333 | -m68340 | -mcpu32 | -m5200`

Specify what processor in the 68000 family is the target. The default is normally the 68020, but this can be changed at configuration time.

`-m68881 | -m68882 | -mno-68881 | -mno-68882`

The target machine does (or does not) have a floating-point coprocessor. The default is to assume a coprocessor for 68020, 68030, and cpu32. Although the basic 68000 is not compatible with the 68881, a combination of the two can be specified, since it's possible to do emulation of the coprocessor instructions with the main processor.

`-m68851 | -mno-68851`

The target machine does (or does not) have a memory-management unit coprocessor. The default is to assume an MMU for 68020 and up.

For details about the PDP-11 machine dependent features options, refer to Section 32.1 *Options*.

`-mpic | -mno-pic`

Generate position-independent (or position-dependent) code. The default is `-mpic`.

`-mall`
`-mall-extensions`

Enable all instruction set extensions. This is the default.

`-mno-extensions`

Disable all instruction set extensions.

`-mextension | -mno-extension`

Enable (or disable) a particular instruction set extension.

`-mcpu`

Enable the instruction set extensions supported by a particular CPU, and disable all other extensions.

`-mmachine`

Enable the instruction set extensions supported by a particular machine model, and disable all other extensions.

The following options are available when as is configured for a picoJava processor.

`-mb`

Generate "big endian" format output.

`-ml`

Generate "little endian" format output.

The following options are available when as is configured for the Motorola 68HC11 or 68HC12 series.

`-m68hc11 | -m68hc12 | -m68hcs12`

Specify what processor is the target. The default is defined by the configuration option when building the assembler.

`-mshort`

Specify to use the 16-bit integer ABI.

`-mlong`

Specify to use the 32-bit integer ABI.

`-mshort-double`

Specify to use the 32-bit double ABI.

`-mlong-double`

Specify to use the 64-bit double ABI.

`-force-long-branchs`

Relative branches are turned into absolute ones. This concerns conditional branches, unconditional branches and branches to a sub routine.

`-S | -short-branchs`

Do not turn relative branchs into absolute ones when the offset is out of range.

`-strict-direct-mode`

Do not turn the direct addressing mode into extended addressing mode when the instruction does not support direct addressing mode.

`-print-insn-syntax`

Print the syntax of instruction in case of error.

`-print-opcodes`

print the list of instructions with syntax and then exit.

`-generate-example`

print an example of instruction for each possible instruction and then exit. This option is only useful for testing `as`.

The following options are available when `as` is configured for the SPARC architecture:

`-Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite`
`-Av8plus | -Av8plusa | -Av9 | -Av9a`

Explicitly select a variant of the SPARC architecture.

`-Av8plus` and `-Av8plusa` select a 32 bit environment. `-Av9` and `-Av9a` select a 64 bit environment.

`-Av8plusa` and `-Av9a` enable the SPARC V9 instruction set with UltraSPARC extensions.

`-xarch=v8plus | -xarch=v8plusa`

For compatibility with the Solaris v9 assembler. These options are equivalent to -Av8plus and -Av8plusa, respectively.

`-bump`

Warn when the assembler switches to another architecture.

The following options are available when as is configured for the 'c54x architecture.

`-mfar-mode`

Enable extended addressing mode. All addresses and relocations will assume extended addressing (usually 23 bits).

`-mcpu=CPU_VERSION`

Sets the CPU version being compiled for.

`-merrors-to-file FILENAME`

Redirect error output to a file, for broken systems which don't support such behaviour in the shell.

The following options are available when as is configured for a mips processor.

`-G num`

This option sets the largest size of an object that can be referenced implicitly with the `gp` register. It is only accepted for targets that use ECOFF format, such as a DECstation running Ultrix. The default value is 8.

`-EB`

Generate "big endian" format output.

`-EL`

Generate "little endian" format output.

`-mips1`
`-mips2`
`-mips3`
`-mips4`
`-mips5`
`-mips32`
`-mips32r2`
`-mips64`

Generate code for a particular mips Instruction Set Architecture level. `-mips1` is an alias for `-march=r3000`, `-mips2` is an alias for `-march=r6000`, `-mips3` is an alias for `-march=r4000` and `-mips4` is an alias for `-march=r8000`. `-mips5`, `-mips32`, `-mips32r2`, and `-mips64` correspond to generic `MIPS V`, `MIPS32`, `MIPS32 Release 2`, and `MIPS64` ISA processors, respectively.

`-march=CPU`

Generate code for a particular mips cpu.

`-mtune=cpu`

Schedule and tune for a particular mips cpu.

`-mfix7000`
`-mno-fix7000`

Cause nops to be inserted if the read of the destination register of an mfhi or mflo instruction occurs in the following two instructions.

`-mdebug`
`-no-mdebug`

Cause stabs-style debugging output to go into an ECOFF-style .mdebug section instead of the standard ELF .stabs sections.

`-mgp32`
`-mfp32`

The register sizes are normally inferred from the ISA and ABI, but these flags force a certain group of registers to be treated as 32 bits wide at all times. `-mgp32` controls the size of general-purpose registers and `-mfp32` controls the size of floating-point registers.

`-mips16`
`-no-mips16`

Generate code for the MIPS 16 processor. This is equivalent to putting `.set mips16` at the start of the assembly file. `-no-mips16` turns off this option.

`-mips3d`
`-no-mips3d`

Generate code for the MIPS-3D Application Specific Extension. This tells the assembler to accept MIPS-3D instructions. `-no-mips3d` turns off this option.

`-mdmx`
`-no-mdmx`

Generate code for the MDMX Application Specific Extension. This tells the assembler to accept MDMX instructions. `-no-mdmx` turns off this option.

`-construct-floats`
`-no-construct-floats`

The `-no-construct-floats` option disables the construction of double width floating point constants by loading the two halves of the value into the two single width floating point registers that make up the double width register. By default `-construct-floats` is selected, allowing construction of these floating point constants.

`-emulation=name`

This option causes `as` to emulate `as` configured for some other target, in all respects, including output format (choosing between ELF and ECOFF only), handling of pseudo-opcodes which may generate debugging information or store symbol table information, and default endianness. The available configuration names are: `mipsecoff`, `mipself`, `mipslecoff`, `mipsbecoff`, `mipslelf`, `mipsbelf`. The first two do not alter the default endianness from that of the primary target for which the assembler was configured; the others change the default to little- or big-endian as indicated by the `b` or `l` in the name. Using `-EB` or `-EL` will override the endianness selection in any case.

This option is currently supported only when the primary target `as` is configured for is a mips ELF or ECOFF target. Furthermore, the primary target or others specified with `-enable-targets=...` at configuration time must include support for the other format, if both are to be available. For example, the Irix 5 configuration includes support for both.

Eventually, this option will support more configurations, with more fine-grained control over the assembler's behavior, and will be supported for more processors.

`-nocpp`

`as` ignores this option. It is accepted for compatibility with the native tools.

`-trap`
`-no-trap`
`-break`
`-no-break`

Control how to deal with multiplication overflow and division by zero. `-trap` or `-no-break` (which are synonyms) take a trap exception (and only work for Instruction Set Architecture level 2 and higher); `-break` or `-no-trap` (also synonyms, and the default) take a break exception.

`-n`

When this option is used, `as` will issue a warning every time it generates a nop instruction from a macro.

The following options are available when as is configured for an MCore processor.

`-jsri2bsr`
`-nojsri2bsr`

Enable or disable the JSRI to BSR transformation. By default this is enabled. The command line option `-nojsri2bsr` can be used to disable it.

`-sifilter`
`-nosifilter`

Enable or disable the silicon filter behaviour. By default this is disabled. The default can be overridden by the `-sifilter` command line option.

`-relax`

Alter jump instructions for long displacements.

`-mcpu=[210|340]`

Select the cpu type on the target hardware. This controls which instructions can be assembled.

`-EB`

Assemble for a big endian target.

`-EL`

Assemble for a little endian target.

See the info pages for documentation of the MMIX-specific options.

The following options are available when as is configured for an Xtensa processor.

`-density | -no-density`

Enable or disable use of instructions from the Xtensa code density option. This is enabled by default when the Xtensa processor supports the code density option.

`-relax | -no-relax`

Enable or disable instruction relaxation. This is enabled by default. Note: In the current implementation, these options also control whether assembler optimizations are performed, making these options equivalent to `-generics` and `-no-generics`.

`-generics | -no-generics`

Enable or disable all assembler transformations of Xtensa instructions. The default is `-generics`; `-no-generics` should be used only in the rare cases when the instructions must be exactly as specified in the assembly source.

`-text-section-literals | -no-text-section-literals`

With `-text-section-literals`, literal pools are interspersed in the text section. The default is `-no-text-section-literals`, which places literals in a separate section in the output file.

`-target-align | -no-target-align`

Enable or disable automatic alignment to reduce branch penalties at the expense of some code density. The default is `-target-align`.

`-longcalls | -no-longcalls`

Enable or disable transformation of call instructions to allow calls across a greater range of addresses. The default is `-no-longcalls`.

2.1. Structure of this Manual

This manual is intended to describe what you need to know to use gnu `as`. We cover the syntax expected in source files, including notation for symbols, constants, and expressions; the directives that `as` understands; and of course how to invoke `as`.

This manual also describes some of the machine-dependent features of various flavors of the assembler.

On the other hand, this manual is *not* intended as an introduction to programming in assembly language--let alone programming in general! In a similar vein, we make no attempt to introduce the machine architecture; we do *not* describe the instruction set, standard mnemonics, registers or addressing modes that are standard to a particular architecture. You may want to consult the manufacturer's machine architecture manual for this information.

2.2. The GNU Assembler

gnu `as` is really a family of assemblers. If you use (or have used) the gnu assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops*) and assembler syntax.

`as` is primarily intended to assemble the output of the gnu C compiler `gcc` for use by the linker `ld`. Nevertheless, we've tried to make `as` assemble correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (refer to Chapter 9 *Machine Dependent Features*). This doesn't mean `as` always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of 680x0 assembly language syntax.

Unlike older assemblers, `as` is designed to assemble a source program in one pass of the source file. This has a subtle impact on the **`.org`** directive (refer to Section 8.61 *`.org new-lc, fill`*).

2.3. Object File Formats

The gnu assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. Refer to Section 6.5 *Symbol Attributes*.

2.4. Command Line

After the program name `as`, the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is significant.

`--` (two hyphens) by itself names the standard input file explicitly, as one of the files for `as` to assemble.

Except for `--` any command line argument that begins with a hyphen (`-`) is an option. Each option changes the behavior of `as`. No option changes the way another option works. An option is a `-` followed by one or more letters; the case of the letter is important. All options are optional.

Some options expect exactly one file name to follow them. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (gnu standard). These two command lines are equivalent:

```
as -o my-object-file.o mumble.s
as -omy-object-file.o mumble.s
```

2.5. Input Files

We use the phrase *source program*, abbreviated *source*, to describe the program input to one run of `as`. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run `as` it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give `as` a command line that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name.

If you give `as` no file names it attempts to read one input file from the `as` standard input, which is normally your terminal. You may have to type [ctl-D] to tell `as` there is no more program to assemble.

Use `-` if you need to explicitly name the standard input file in your command line.

If the source is empty, `as` produces a small, empty object file.

2.5.1. Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. Refer to Section 2.7 *Error and Warning Messages*.

Physical files are those files named in the command line given to `as`.

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when `as` source is itself synthesized from other files. `as` understands the `#` directives emitted by the `gcc` preprocessor. Refer to Section 8.35 *`.file string`*.

2.6. Output (Object) File

Every time you run `as` it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is `a.out`, or `b.out` when `as` is configured for the Intel 80960. You can give it another name by using the `-o` option. Conventionally, object file names end with `.o`. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn't currently possible, but it can be done for the `a.out` format.)

The object file is meant for input to the linker `ld`. It contains assembled program code, information to help `ld` integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

2.7. Error and Warning Messages

`as` may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs `as` automatically. Warnings report an assumption made so that `as` could keep assembling a flawed program; errors report a grave problem that stops the assembly.

Warning messages have the format

```
file_name:NNN:Warning Message Text
```

(where *NNN* is a line number). If a logical file name has been given (refer to Section 8.35 *`.file string`*), it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (refer to Section 8.52 *`.line line-number`*), then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory.

Error messages have the format

```
file_name:NNN:FATAL:Error Message Text
```

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

Chapter 3. Command-Line Options

This chapter describes command-line options available in *all* versions of the gnu assembler. Refer to Chapter 9 *Machine Dependent Features* for options specific to particular machine architectures.

If you are invoking `as` via the gnu C compiler, you can use the `-Wa` option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the `-Wa`) by commas. For example:

```
gcc -c -g -O -Wa,-alh,-L file.c
```

This passes two options to the assembler: `-alh` (emit a listing to standard output with high-level and assembly source) and `-L` (retain local symbols in the symbol table).

Usually you do not need to use this `-Wa` mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the gnu compiler driver with the `-v` option to see precisely what options it passes to each compilation pass, including the assembler.)

3.1. Enable Listings: `-a[cdhlns]`

These options enable listing output from the assembler. By itself, `-a` requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: `-ah` requests a high-level language listing, `-al` requests an output-program assembly listing, and `-as` requests a symbol table listing. High-level listings require that a compiler debugging option like `-g` be used, and that assembly listings (`-al`) be requested also.

Use the `-ac` option to omit false conditionals from a listing. Any lines which are not assembled because of a false `.if` (or `.ifdef`, or any other conditional), or a true `.if` followed by an `.else`, will be omitted from the listing.

Use the `-ad` option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives `.list`, `.nolist`, `.psize`, `.eject`, `.title`, and `.sbttl`. The `-an` option turns off all forms processing. If you do not request listing output with one of the `-a` options, the listing-control directives have no effect.

The letters after `-a` may be combined into one option, *e.g.*, `-aln`.

Note if the assembler source is coming from the standard input (eg because it is being created by `gcc` and the `-pipe` command line switch is being used) then the listing will not contain any comments or preprocessor directives. This is because the listing code buffers input source lines from stdin only after they have been preprocessed by the assembler. This reduces memory usage and makes the code more efficient.

3.2. `-D`

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers also work with `as`.

3.3. Work Faster: `-f`

`-f` should only be used when assembling programs written by a (trusted) compiler. `-f` stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. Refer to Section 4.1 *Preprocessing*.

> *Warning:* if you use `-f` when the files actually need to be preprocessed (if they contain comments, for example), `as` does not work correctly.

3.4. `.include`Search Path: `-Ipath`

Use this option to add a `path` to the list of directories `as` searches for files specified in `.include` directives (refer to Section 8.45 *`.include "file"`*). You may use `-I` as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, `as` searches any `-I` directories in the same order as they were specified (left to right) on the command line.

3.5. Difference Tables: `-K`

`as` sometimes alters the code emitted for directives of the form `.word sym1-sym2`; Section 8.97 *`.word expressions`*. You can use the `-K` option if you want a warning issued when this is done.

3.6. Include Local Labels: `-L`

Labels beginning with `L` (upper case only) are called *local labels*. Refer to Section 6.3 *Symbol Names*. Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both `as` and `ld` discard such labels, so you do not normally debug with them.

This option tells `as` to retain those `L...` symbols in the object file. Usually if you do this you also tell the linker `ld` to preserve symbols whose names begin with `L`.

By default, a local label is any label beginning with `L`, but each target is allowed to redefine the local label prefix. On the HPPA local labels begin with `L$`.

3.7. Configuring listing output: `-listing`

The listing feature of the assembler can be enabled via the command line switch `-a` (refer to Section 3.1 *Enable Listings: `-a[cdhlns]`*). This feature combines the input source file(s) with a hex dump of the corresponding locations in the output object file, and displays them as a listing file. The format of this listing can be controlled by pseudo ops inside the assembler source (refer to Section 8.56 *`.list`* Section 8.89 *`.title "heading"`* Section 8.72 *`.sbttl "subheading"`* Section 8.67 *`.psize lines, columns`* Section 8.22 *`.eject`*) and also by the following switches:

`-listing-lhs-width=number`

: Sets the maximum width, in words, of the first line of the hex byte dump. This dump appears on the left hand side of the listing output.

`-listing-lhs-width2=number`

: Sets the maximum width, in words, of any further lines of the hex byte dump for a given input source line. If this value is not specified, it defaults to being the same as the value specified for `-listing-lhs-width`. If neither switch is used the default is to one.

`-listing-rhs-width=number`

Sets the maximum width, in characters, of the source line that is displayed alongside the hex dump. The default value for this parameter is 100. The source line is displayed on the right hand side of the listing output.

`-listing-cont-lines=number`

Sets the maximum number of continuation lines of hex dump that will be displayed for a given single line of source input. The default value is 4.

3.8. Assemble in MRI Compatibility Mode: `-M`

The `-M` or `-mri` option selects MRI compatibility mode. This changes the syntax and pseudo-op handling of `as` to make it compatible with the `ASM68K` or the `ASM960` (depending upon the configured target) assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. Note in particular that the handling of macros and macro arguments is somewhat different. The purpose of this option is to permit assembling existing MRI assembler code using `as`.

The MRI compatibility is not complete. Certain operations of the MRI assembler depend upon its object file format, and can not be supported using other object file formats. Supporting these would require enhancing each object file format individually. These are:

- global symbols in common section

  The m68k MRI assembler supports common sections which are merged by the linker. Other object file formats do not support this. `as` handles common sections by treating them as a single common symbol. It permits local symbols to be defined within a common section, but it can not support global symbols, since it has no way to describe them.

- complex relocations

  The MRI assemblers support relocations against a negated section address, and relocations which combine the start addresses of two or more sections. These are not support by other object file formats.

- `END` pseudo-op specifying start address

  The MRI `END` pseudo-op permits the specification of a start address. This is not supported by other object file formats. The start address may instead be specified using the `-e` option to the linker, or in a linker script.

- `IDNT`, `.ident` and `NAME` pseudo-ops

  The MRI `IDNT`, `.ident` and `NAME` pseudo-ops assign a module name to the output file. This is not supported by other object file formats.

- `ORG` pseudo-op

  The m68k MRI `ORG` pseudo-op begins an absolute section at a given address. This differs from the usual `as .org` pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script.

There are some other features of the MRI assembler which are not supported by `as`, typically either because they are difficult or because they seem of little consequence. Some of these may be supported in future releases.

- EBCDIC strings

  EBCDIC strings are not supported.

- packed binary coded decimal

  Packed binary coded decimal is not supported. This means that the `DC.P` and `DCB.P` pseudo-ops are not supported.

- `FEQU` pseudo-op

  The m68k `FEQU` pseudo-op is not supported.

- `NOOBJ` pseudo-op

  The m68k `NOOBJ` pseudo-op is not supported.

- `OPT` branch control options

  The m68k `OPT` branch control options--`B`, `BRS`, `BRB`, `BRL`, and `BRW`--are ignored. `as` automatically relaxes all branches, whether forward or backward, to an appropriate size, so these options serve no purpose.

- `OPT` list control options

  The following m68k `OPT` list control options are ignored: `C`, `CEX`, `CL`, `CRE`, `E`, `G`, `I`, `M`, `MEX`, `MC`, `MD`, `X`.

- other `OPT` options

  The following m68k `OPT` options are ignored: `NEST`, `O`, `OLD`, `OP`, `P`, `PCO`, `PCR`, `PCS`, `R`.

- `OPT D` option is default

  The m68k `OPT D` option is the default, unlike the MRI assembler. `OPT NOD` may be used to turn it off.

- `XREF` pseudo-op.

  The m68k `XREF` pseudo-op is ignored.

- `.debug` pseudo-op

  The i960 `.debug` pseudo-op is not supported.

- `.extended` pseudo-op

  The i960 `.extended` pseudo-op is not supported.

- `.list` pseudo-op.

  The various options of the i960 `.list` pseudo-op are not supported.

- `.optimize` pseudo-op

  The i960 `.optimize` pseudo-op is not supported.

- `.output` pseudo-op

  The i960 `.output` pseudo-op is not supported.

- `.setreal` pseudo-op

  The i960 `.setreal` pseudo-op is not supported.

3.9. Dependency Tracking: `-MD`

`as` can generate a dependency file for the file it creates. This file consists of a single rule suitable for `make` describing the dependencies of the main source file.

The rule is written to the file named in its argument.

This feature is used in the automatic updating of makefiles.

3.10. Name the Object File: `-o`

There is always one object file output when you run `as`. By default it has the name `a.out` (or `b.out`, for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, `as` overwrites any existing file of the same name.

3.11. Join Data and Text Sections: `-R`

`-R` tells `as` to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section (refer to Chapter 5 *Sections and Relocation*).

When you specify `-R` it would be possible to generate shorter address displacements (because we do not have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of `as`. In future, `-R` may work this way.

When `as` is configured for COFF or ELF output, this option is only useful if you use sections named `.text` and `.data`.

`-R` is not supported for any of the HPPA targets. Using `-R` generates a warning from `as`.

3.12. Display Assembly Statistics: `-statistics`

Use `-statistics` to display two statistics about the resources used by `as`: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in cpu seconds).

3.13. Compatible Output: `-traditional-format`

For some targets, the output of `as` is different in some ways from the output of some existing assembler. This switch requests `as` to use the traditional format instead.

For example, it disables the exception frame optimizations which `as` normally does by default on `gcc` output.

3.14. Announce Version: `-v`

You can find out what version of as is running by including the option `-v` (which you can also spell as `-version`) on the command line.

3.15. Control Warnings: `-W`, `-warn`, `-no-warn`, `-fatal-warnings`

`as` should never give a warning or error message when assembling compiler output. But programs written by people often cause `as` to give a warning that a particular assumption was made. All such warnings are directed to the standard error file.

If you use the `-W` and `-no-warn` options, no warnings are issued. This only affects the warning messages: it does not change any particular of how `as` assembles your file. Errors, which stop the assembly, are still reported.

If you use the `-fatal-warnings` option, `as` considers files that generate warnings to be in error.

You can switch these options off again by specifying `-warn`, which causes warnings to be output as usual.

3.16. Generate Object File in Spite of Errors: `-Z`

After an error message, `as` normally produces no output. If for some reason you are interested in object file output even after `as` gives an error message on your program, use the `-Z` option. If there are any errors, `as` continues anyways, and writes an object file after a final warning message of the form `n errors, m warnings, generating bad object file.`

Chapter 4. Syntax

This chapter describes the machine-independent syntax allowed in a source file. `as` syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler, except that `as` does not assemble Vax bit-fields.

4.1. Preprocessing

The `as` internal preprocessor:

- adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the `.include` directive (refer to Section 8.45 *.include "file"*). You can use the gnu C compiler driver to get other "CPP" style preprocessing by giving the input file a `.S` suffix.

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not preprocessed.

If the first line of an input file is `#NO_APP` or if you use the `-f` option, whitespace and comments are not removed from the input file. Within an input file, you can ask for whitespace and comment removal in specific portions of the by putting a line that says `#APP` before the text that may contain whitespace or comments, and putting a line that says `#NO_APP` after this text. This feature is mainly intend to support `asm` statements in compilers whose output is otherwise free of comments and whitespace.

4.2. Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Unless within character constants (refer to Section 4.6.1 *Character Constants*), any whitespace means the same as exactly one space.

4.3. Comments

There are two ways of rendering comments to `as`. In both cases the comment is equivalent to one space.

Anything from `/*` through the next `*/` is a comment. This means you may not nest these comments.

```
/*
  The only way to include a newline ('\n') in a comment
  is to use this sort of comment.
*/

/* This sort of comment does not nest. */
```

Anything from the *line comment* character to the next newline is considered a comment and is ignored. The line comment character is ; for the AMD 29K family; ; on the ARC; @ on the ARM; ; for the H8/300 family; ! for the H8/500 family; ; for the HPPA; # on the i386 and x86-64; # on the i960; ; for the PDP-11; ; for picoJava; ; for Motorola PowerPC; ! for the Renesas / SuperH SH; ! on the SPARC; # on the ip2k; # on the m32r; | on the 680x0; # on the 68HC11 and 68HC12; ; on the M880x0; # on the Vax; ! for the Z8000; # on the V850; # for Xtensa systems; see Chapter 9 *Machine Dependent Features*.

On some machines there are two different line comment characters. One character only begins a comment if it is the first non-whitespace character on a line, while the other always begins a comment.

The V850 assembler also supports a double dash as starting a comment that extends to the end of the line.

-;

To be compatible with past assemblers, lines that begin with # have a special interpretation. Following the # should be an absolute expression (refer to Chapter 7 *Expressions*): the logical line number of the *next* line. Then a string (refer to Section 4.6.1.1 *Strings*) is allowed: if present, it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
                          # This is an ordinary comment.
# 42-6 "new_file_name"    # New logical file name
                          # This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of as.

4.4. Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters _.$. On most machines, you can also use $ in symbol names; exceptions are noted in Chapter 9 *Machine Dependent Features*. No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). Chapter 6 *Symbols*.

4.5. Statements

A *statement* ends at a newline character (\n) or line separator character. (The line separator is usually ;, unless this conflicts with the comment character; Chapter 9 *Machine Dependent Features*.) The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements.

It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

An empty statement is allowed, and may include whitespace. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot . then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language *instruction*: it assembles into a machine language instruction. Different versions of as for different computers recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is a symbol immediately followed by a colon (`:`). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label's symbol and its colon. Refer to Section 6.1 *Labels*.

For HPPA targets, labels need not be immediately followed by a colon, but the definition of a label must begin in column zero. This also implies that only one label may be defined on each line.

```
label:     .directive    followed by something
another_label:           # This is an empty statement.
           instruction   operand_1, operand_2, ...
```

4.6. Constants

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.
.ascii "Ring the bell\7"                  # A string constant.
.octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
.float 0f-314159265358979323846264338327\
95028841971.693993751E-40                 # - pi, a flonum.
```

4.6.1. Character Constants

There are two kinds of character constants. A *character* stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string *literals*) are potentially many bytes and their values may not be used in arithmetic expressions.

4.6.1.1. Strings

A *string* is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to *escape* these characters: precede them with a backslash \ character. For example \\ represents one backslash: the first \ is an escape which tells `as` to interpret the second character literally as a backslash (which prevents `as` from recognizing the second \ as an escape character). The complete list of escapes follows.

\b

Mnemonic for backspace; for ASCII this is octal code 010.

\f

Mnemonic for FormFeed; for ASCII this is octal code 014.

\n

Mnemonic for newline; for ASCII this is octal code 012.

\r

Mnemonic for carriage-Return; for ASCII this is octal code 015.

\t

Mnemonic for horizontal Tab; for ASCII this is octal code 011.

\ digit digit digit

An octal character code. The numeric code is 3 octal digits. For compatibility with other Unix systems, 8 and 9 are accepted as digits: for example, `\008` has the value 010, and `\009` the value 011.

\x hex-digits...

A hex character code. All trailing hex digits are combined. Either upper or lower case `x` works.

****

Represents one \ character.

\\"

Represents one " character. Needed in strings to represent this character, because an unescaped " would end the string.

\ anything-else

Any other character when escaped by **** gives a warning, but assembles as if the \ was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However `as` has no other interpretation, so `as` knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, do not use an escape sequence.

4.6.1.2. Characters

A single character may be written as a single quote immediately followed by that character. The same escapes apply to characters as to strings. So if you want to write the character backslash, you must write **'** where the first \ escapes the second \. As you can see, the quote is an acute accent, not a grave accent. A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. `as` assumes your character code is ASCII: **'A** means 65, **'B** means 66, and so on.

4.6.2. Number Constants

`as` distinguishes three kinds of numbers according to how they are stored in the target machine. *Integers* are numbers that would fit into an `int` in the C language. *Bignums* are integers, but they are stored in more than 32 bits. *Flonums* are floating point numbers, described below.

4.6.2.1. Integers

A binary integer is `0b` or `0B` followed by zero or more of the binary digits `01`.

An octal integer is `0` followed by zero or more of the octal digits (`01234567`).

A decimal integer starts with a non-zero digit followed by zero or more digits (`0123456789`).

A hexadecimal integer is `0x` or `0X` followed by one or more hexadecimal digits chosen from `0123456789abcdefABCDEF`.

Integers have the usual values. To denote a negative integer, use the prefix operator - discussed under expressions (refer to Section 7.2.3 *Prefix Operator*).

4.6.2.2. Bignums

A *bignum* has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

4.6.2.3. Flonums

A *flonum* represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by `as` to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer's floating point format (or formats) by a portion of `as` specialized to that computer.

A flonum is written by writing (in order)

- The digit `0`. (`0` is optional on the HPPA.)
- A letter, to tell `as` the rest of the number is a flonum. **`e`** is recommended. Case is not important.

  On the H8/300, H8/500, Renesas / SuperH SH, and AMD 29K architectures, the letter must be one of the letters `DFPRSX` (in upper or lower case).

  On the ARC, the letter must be one of the letters `DFRS` (in upper or lower case).

  On the Intel 960 architecture, the letter must be one of the letters `DFT` (in upper or lower case).

  On the HPPA architecture, the letter must be `E` (upper case only).
- An optional sign: either `+` or `-`.
- An optional *integer part*: zero or more decimal digits.
- An optional *fractional part*: **`.`** followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - An `E` or `e`.
  - Optional sign: either `+` or `-`.
  - One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

`as` does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running `as`.

Chapter 5.

Sections and Relocation

5.1. Background

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purpose. For example there may be a "read only" section.

The linker `ld` reads many object files (partial programs) and combines their contents to form a runnable program. When `as` emits an object file, the partial program is assumed to start at address 0. `ld` assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how `as` uses sections.

`ld` moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses. For the H8/300 and H8/500, and for the Renesas / SuperH SH, `as` pads sections if needed to ensure they end on a word (sixteen bit) boundary.

An object file written by `as` has at least three sections, any of which may be empty. These are named *text*, *data* and *bss* sections.

When it generates COFF or ELF output, `as` can also generate whatever other named sections you specify using the `.section` directive (refer to Section 8.74 *`.section name`*). If you do not use any directives that place output in the `.text` or `.data` sections, these sections still exist, but are empty.

When `as` generates SOM or ELF output for the HPPA, `as` can also generate whatever other named sections you specify using the `.space` and `.subspace` directives. See [HP9000 Series 800 Assembly Language Reference Manual] (HP 92432-90001) for details on the `.space` and `.subspace` assembler directives.

Additionally, `as` uses different names for the standard text, data, and bss sections when generating SOM output. Program text is placed into the `$CODE$` section, data into `$DATA$`, and BSS into `$BSS$`.

Within the object file, the text section starts at address `0`, the data section follows, and the bss section follows the data section.

When generating either SOM or ELF output files on the HPPA, the text section starts at address `0`, the data section at address `0x4000000`, and the bss section follows the data section.

To let `ld` know which data changes when the sections are relocated, and how to change that data, `as` also writes to the object file details of the relocation needed. To perform relocation `ld` must know, each time an address in the object file is mentioned:

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of `(address) − (start-address of section)`?
- Is the reference to an address "Program-Counter relative"?

In fact, every address `as` ever uses is expressed as

```
(section) + (offset into section)
```

Further, most expressions `as` computes have this section-relative nature. (For some object formats, such as SOM for the HPPA, some expressions are symbol-relative instead.)

In this manual we use the notation {`secname N`} to mean "offset `N` into section `secname`."

Apart from text, data and bss sections you need to know about the *absolute* section. When `ld` mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}` is "relocated" to run-time address 0 by `ld`. Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `{absolute 239}` in one part of a program is always the same address when the program is running as address `{absolute 239}` in any other part of the program.

The idea of sections is extended to the *undefined* section. Any address whose section is unknown at assembly time is by definition rendered {undefined `U`}--where `U` is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

By analogy the word *section* is used to describe groups of sections in the linked program. `ld` puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and bss sections.

Some sections are manipulated by `ld`; others are invented for use of `as` and have no meaning except during assembly.

5.2. Linker Sections

`ld` deals with just four kinds of sections, summarized below.

named sections

section names

text section
data section

These sections hold your program. `as` and `ld` treat them as separate but equal sections. Anything you can say of one section is true of another. When the program is running, however, it is customary for the text section to be unalterable. The text section is often shared among processes: it contains instructions, constants and the like. The data section of a running program is usually alterable: for example, C variables would be stored in the data section.

bss section

This section contains zeroed bytes when your program begins running. It is used to hold uninitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.

absolute section

Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that `ld` must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.

undefined section

This "section" is a catch-all for address references to objects not in the preceding sections.

An idealized example of three relocatable sections follows. The example uses the traditional section names `.text` and `.data`. Memory addresses are on the horizontal axis.

```
                     +-----+----+--+
partial program # 1: |ttttt|dddd|00|
                     +-----+----+--+

                     text   data bss
                     seg.   seg. seg.

                     +---+---+---+
partial program # 2: |TTT|DDD|000|
                     +---+---+---+

                     +--+---+-----+--+----+---+-----+~~
linked program:      |  |TTT|ttttt|  |dddd|DDD|00000|
                     +--+---+-----+--+----+---+-----+~~

    addresses:       0 ...
```

5.3. Assembler Internal Sections

These sections are meant only for the internal use of `as`. They have no meaning at run-time. You do not really need to know about these sections for most purposes; but they can be mentioned in `as` warning messages, so it might be helpful to have an idea of their meanings to `as`. These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

ASSEMBLER-INTERNAL-LOGIC-ERROR!

An internal assembler logic error has been found. This means there is a bug in the assembler.

expr section

The assembler stores complex expression internally as combinations of symbols. When it needs to represent an expression as a symbol, it puts it in the expr section.

5.4. Sub-Sections

Assembled bytes conventionally fall into two sections: text and data. You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. `as` allows you to use *subsections* for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0` before each section of code being output, and a `.text 1` before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes. (Subsections may be padded a different amount on different flavors of `as`.)

Subsections appear in your object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) The object file contains no representation of subsections; `ld` and other programs that manipulate object files see no trace of them. They just see all your text subsections as a text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text expression` or a `.data expression` statement. When generating COFF or ELF output, you can also use an extra subsection argument with arbitrary named sections: `.section name, expression`. `Expression` should be an absolute expression (refer to Chapter 7 *Expressions*). If you just say `.text` then `.text 0` is assumed. Likewise `.data` means `.data 0`. Assembly begins in `text 0`. For instance:

```
.text 0     # The default subsection is text 0 anyway.
.ascii "This lives in the first text subsection. *"
.text 1
.ascii "But this lives in the second text subsection."
.data 0
.ascii "This lives in the data section,"
.ascii "in the first data subsection."
.text 0
.ascii "This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
```

Each section has a *location counter* incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to `as` there is no concept of a subsection location counter. There is no way to directly manipulate a location counter--but the `.align` directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the *active* location counter.

5.5. bss Section

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

The `.lcomm` pseudo-op defines a symbol in the bss section; see Section 8.50 *`.lcomm symbol, length`*.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see Section 8.16 *`.comm symbol, length`*.

When assembling for a target which supports multiple sections, such as ELF or COFF, you may switch into the `.bss` section and define symbols as usual; see Section 8.74 *`.section name`*. You may only assemble zero values into the section. Typically the section will only contain symbol definitions and `.skip` directives (refer to Section 8.80 *`.skip size, fill`*).

Chapter 6. Symbols

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug.

> *Warning:* `as` does not place symbols in the object file in the same order they were declared. This may break some debuggers.

6.1. Labels

A *label* is written as a symbol immediately followed by a colon `:`. The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

On the HPPA, the usual form for a label need not be immediately followed by a colon, but instead must start in column zero. Only one label may be defined on a single line. To work around this, the HPPA version of `as` also provides a special directive `.label` for defining labels more flexibly.

6.2. Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign =, followed by an expression (refer to Chapter 7 *Expressions*). This is equivalent to using the `.set` directive. Section 8.75 *`.set symbol, expression`*.

6.3. Symbol Names

Symbol names begin with a letter or with one of `._`. On most machines, you can also use `$` in symbol names; exceptions are noted in Chapter 9 *Machine Dependent Features*. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 9 *Machine Dependent Features*), and underscores. For the AMD 29K family, `?` is also allowed in the body of a symbol name, though not at its beginning.

Case of letters is significant: `foo` is a different symbol name than `Foo`.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

6.3.1. Local Symbol Names

Local symbols help compilers and programmers use names temporarily. They create symbols which are guaranteed to be unique over the entire scope of the input source code and which can be referred to by a simple notation. To define a local symbol, write a label of the form `N:` (where *N* represents any positive integer). To refer to the most recent previous definition of that symbol write `Nb`, using the same number as when you defined the label. To refer to the next definition of a local label, write `Nf`-- The `b` stands for"backwards" and the `f` stands for "forwards".

There is no restriction on how you can use these labels, and you can reuse them too. So that it is possible to repeatedly define the same local label (using the same number `N`), although you can only refer to the most recently defined local label of that number (for a backwards reference) or the next definition of a specific local label for a forward reference. It is also worth noting that the first 10 local labels (`0:...9:`) are implemented in a slightly more efficient manner than the others.

Here is an example:

```
1:        branch 1f
2:        branch 1b
1:        branch 2f
2:        branch 1b
```

Which is the equivalent of:

```
label_1:  branch label_3
label_2:  branch label_1
label_3:  branch label_4
label_4:  branch label_3
```

Local symbol names are only a notational device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file. The names are constructed using these parts:

`L`

All local labels begin with `L`. Normally both `as` and `ld` forget symbols that start with `L`. These labels are used for symbols you are never intended to see. If you use the `-L` option then `as` retains these symbols in the object file. If you also instruct `ld` to retain these symbols, you may use them in debugging.

`number`

This is the number that was used in the local label definition. So if the label is written `55:` then the number is `55`.

C-B

This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value of `\002` (control-B).

ordinal number

This is a serial number to keep the labels distinct. The first definition of `0:` gets the number `1`. The 15th definition of `0:` gets the number `15`, and so on. Likewise the first definition of `1:` gets the number `1` and its 15th defintion gets `15` as well.

So for example, the first `1:` is named `L1`**C-B**`1`, the 44th `3:` is named `L3`**C-B**`44`.

6.3.2. Dollar Local Labels

`as` also supports an even more local form of local labels called dollar labels. These labels go out of scope (ie they become undefined) as soon as a non-local label is defined. Thus they remain valid for only a small region of the input source code. Normal local labels, by contrast, remain in scope for the entire file, or until they are redefined by another occurrence of the same local label.

Dollar labels are defined in exactly the same way as ordinary local labels, except that instead of being terminated by a colon, they are terminated by a dollar sign. eg `55$`.

They can also be distinguished from ordinary local labels by their transformed name which uses ASCII character `\001` (control-A) as the magic character to distinguish them from ordinary labels. Thus the 5th defintion of `6$` is named `L6`**C-A**`5`.

6.4. The Special Dot Symbol

The special symbol `.` refers to the current address that `as` is assembling into. Thus, the expression `melvin: .long .` defines `melvin` to contain its own address. Assigning a value to `.` is treated the same as a `.org` directive. Thus, the expression `.=.+4` is the same as saying `.space 4`.

6.5. Symbol Attributes

Every symbol has, as well as its name, the attributes "Value" and "Type". Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, `as` assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

6.5.1. Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as `ld` changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source file, and `ld` tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a `.comm` common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

6.5.2. Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

6.5.3. Symbol Attributes: `a.out`

6.5.3.1. Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (refer to Section 8.19 *`.desc symbol, abs-expression`*). A descriptor value means nothing to `as`.

6.5.3.2. Other

This is an arbitrary 8-bit value. It means nothing to `as`.

6.5.4. Symbol Attributes for COFF

The COFF format supports a multitude of auxiliary symbol attributes; like the primary symbol attributes, they are set between `.def` and `.endef` directives.

6.5.4.1. Primary Attributes

The symbol name is set with `.def`; the value and type, respectively, with `.val` and `.type`.

6.5.4.2. Auxiliary Attributes

The `as` directives `.dim`, `.line`, `.scl`, `.size`, and `.tag` can generate auxiliary symbol table information for COFF.

6.5.5. Symbol Attributes for SOM

The SOM format for the HPPA supports a multitude of symbol attributes set with the `.EXPORT` and `.IMPORT` directives.

The attributes are described in [HP9000 Series 800 Assembly Language Reference Manual] (HP 92432-90001) under the `IMPORT` and `EXPORT` assembler directive documentation.

Chapter 7. Expressions

An *expression* specifies an address or numeric value. Whitespace may precede and/or follow an expression.

The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when `as` sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression--but the second pass is currently not implemented. `as` aborts with an error message in this situation.

7.1. Empty Expressions

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression, and `as` assumes a value of (absolute) 0. This is compatible with other assemblers.

7.2. Integer Expressions

An *integer expression* is one or more *arguments* delimited by *operators*.

7.2.1. Arguments

Arguments are symbols, numbers or subexpressions. In other contexts arguments are sometimes called "arithmetic operands". In this manual, to avoid confusing them with the "instruction operands" of the machine language, we use the term "argument" to refer to parts of expressions only, reserving the word "operand" to refer only to machine instruction operands.

Symbols are evaluated to yield {`section` `NNN`} where `section` is one of text, data, bss, absolute, or undefined. `NNN` is a signed, 2's complement 32 bit integer.

Numbers are usually integers.

A number can be a flonum or bignum. In this case, you are warned that only the low order 32 bits are used, and `as` pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Subexpressions are a left parenthesis `(` followed by an integer expression, followed by a right parenthesis `)`; or a prefix operator followed by an argument.

7.2.2. Operators

Operators are arithmetic functions, like `+` or `%`. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by whitespace.

7.2.3. Prefix Operator

`as` has the following *prefix operators*. They each take one argument, which must be absolute.

`-`

Negation. Two's complement negation.

`~`

Complementation. Bitwise not.

7.2.4. Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from `+` or `-`, both arguments must be absolute, and the result is absolute.

1. Highest Precedence

   `*`

   Multiplication.

   `/`

   Division. Truncation is the same as the C operator `/`

   `%`

   Remainder.

   `<`
   `<<`

   Shift Left. Same as the C operator `<<`.

   `>`
   `>>`

   Shift Right. Same as the C operator `>>`.

2. Intermediate precedence

   `|`

   Bitwise Inclusive Or.

   `&`

   Bitwise And.

   `^`

   Bitwise Exclusive Or.

   `!`

   Bitwise Or Not.

3. Low Precedence

`+`

Addition. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.

`-`

Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.

`==`

Is Equal To

`<>`

Is Not Equal To

`<`

Is Less Than

`>`

Is Greater Than

`>=`

Is Greater Than Or Equal To

`<=`

Is Less Than Or Equal To

The comparison operators can be used as infix operators. A true results has a value of -1 whereas a false result has a value of 0. Note, these operators perform signed comparisons.

4. Lowest Precedence

`&&`

Logical And.

`||`

Logical Or.

These two logical operations can be used to combine the results of sub expressions. Note, unlike the comparison operators a true result returns a value of 1 but a false results does still return 0. Also note that the logical or operator has a slightly lower precedence than logical and.

In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

Chapter 8. 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. Chapter 9 *Machine Dependent Features*.

8.1. `.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.

8.2. `.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.

8.3. `.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, Xtensa, and Renesas / SuperH 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, and the arm and strongarm, 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).

8.4. `.ascii "string"...`

`.ascii` expects zero or more string literals (refer to Section 4.6.1.1 *Strings*) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

8.5. `.asciz "string"...`

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

8.6. `.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.

8.7. `.byte expressions`

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

8.8. `.cfi_startproc`

`.cfi_startproc` is used at the beginning of each function that should have an entry in `.eh_frame`. It initializes some internal data structures and emits architecture dependent initial CFI instructions. Don't forget to close the function by `.cfi_endproc`.

8.9. `.cfi_endproc`

`.cfi_endproc` is used at the end of a function where it closes its unwind entry previously opened by `.cfi_startproc`. and emits it to `.eh_frame`.

8.10. `.cfi_def_cfa register, offset`

`.cfi_def_cfa` defines a rule for computing CFA as: `take address from register and add offset to it.`

8.11. `.cfi_def_cfa_register register`

`.cfi_def_cfa_register` modifies a rule for computing CFA. From now on `register` will be used instead of the old one. Offset remains the same.

8.12. `.cfi_def_cfa_offset offset`

`.cfi_def_cfa_offset` modifies a rule for computing CFA. Register remains the same, but `offset` is new. Note that it is the absolute offset that will be added to a defined register to compute CFA address.

8.13. `.cfi_adjust_cfa_offset offset`

Same as `.cfi_def_cfa_offset` but `offset` is a relative value that is added/substracted from the previous offset.

8.14. `.cfi_offset register, offset`

Previous value of `register` is saved at offset `offset` from CFA.

8.15. `.cfi_verbose [1|0]`

Switch on/off verbosity of the CFI machinery. as will print lots of useful messages to standard output if you use this directive.

8.16. `.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.

8.17. `.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.

8.18. `.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.

8.19. `.desc symbol, abs-expression`

This directive sets the descriptor of the symbol (refer to Section 6.5 *Symbol Attributes*) 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.

8.20. `.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.

8.21. `.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. Chapter 9 *Machine Dependent Features*.

8.22. `.eject`

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

8.23. `.else`

`.else` is part of the `as` support for conditional assembly; Section 8.43 *`.if absolute expression`*. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

8.24. `.elseif`

`.elseif` is part of the `as` support for conditional assembly; Section 8.43 *`.if absolute expression`*. It is shorthand for beginning a new `.if` block that would otherwise fill the entire `.else` section.

8.25. `.end`

`.end` marks the end of the assembly file. `as` does not process anything in the file past the `.end` directive.

8.26. `.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.

8.27. `.endfunc`

`.endfunc` marks the end of a function specified with `.func`.

8.28. `.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. Section 8.43 *`.if absolute expression`*.

8.29. `.equ symbol, expression`

This directive sets the value of `symbol` to `expression`. It is synonymous with `.set`; Section 8.75 *`.set symbol, expression`*.

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

8.30. `.equiv symbol, expression`

The `.equiv` directive is like `.equ` and `.set`, except that the assembler will signal an error if `symbol` is already defined. Note a symbol which has been referenced but not actually defined is considered to be undefined.

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

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

8.31. `.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.

8.32. `.exitm`

Exit early from the current macro definition. Section 8.58 *`.macro`*.

8.33. `.extern`

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

8.34. `.fail expression`

Generates an error or a warning. If the value of the `expression` is 500 or more, `as` will print a warning message. If the value is less than 500, `as` will print an error message. The message will include the value of `expression`. This can occasionally be useful inside complex nested macros or conditional assembly.

8.35. `.file string`

`.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. Chapter 9 *Machine Dependent Features*.

8.36. `.fill repeat, size, value`

`repeat`, `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.

8.37. `.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. Chapter 9 *Machine Dependent Features*.

8.38. .func name[,label]

.func emits debugging information to denote function name, and is ignored unless the file is assembled with debugging enabled. Only -gstabs is currently supported. label is the entry point of the function and if omitted name prepended with the leading char is used. leading char is usually _ or nothing, depending on the target. All functions are currently defined to have void return type. The function must be terminated with .endfunc.

8.39. .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. Section 19.5 *HPPA Assembler Directives*.

8.40. .hidden names

This one of the ELF visibility directives. The other two are .internal (refer to Section 8.47 *.internal names*) and .protected (refer to Section 8.66 *.protected names*).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to hidden which means that the symbols are not visible to other components. Such symbols are always considered to be protected as well.

8.41. .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.

8.42. .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.

8.43. .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 (refer to Section 8.28 *.endif*); optionally, you may include code for the alternative condition, flagged by .else (refer to Section 8.23 *.else*). If you have several conditions to check, .elseif may be used to avoid nesting blocks if/else within each subsequent .else block.

The following variants of .if are also supported:

`.ifdef symbol`

Assembles the following section of code if the specified `symbol` has been defined. Note a symbol which has been referenced but not yet defined is considered to be undefined.

`.ifc string1,string2`

Assembles the following section of code if the two strings are the same. The strings may be optionally quoted with single quotes. If they are not quoted, the first string stops at the first comma, and the second string stops at the end of the line. Strings which contain whitespace should be quoted. The string comparison is case sensitive.

`.ifeq absolute expression`

Assembles the following section of code if the argument is zero.

`.ifeqs string1,string2`

Another form of `.ifc`. The strings must be quoted using double quotes.

`.ifge absolute expression`

Assembles the following section of code if the argument is greater than or equal to zero.

`.ifgt absolute expression`

Assembles the following section of code if the argument is greater than zero.

`.ifle absolute expression`

Assembles the following section of code if the argument is less than or equal to zero.

`.iflt absolute expression`

Assembles the following section of code if the argument is less than zero.

`.ifnc string1,string2.`

Like `.ifc`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

`.ifndef symbol`
`.ifnotdef symbol`

Assembles the following section of code if the specified `symbol` has not been defined. Both spelling variants are equivalent. Note a symbol which has been referenced but not yet defined is considered to be undefined.

`.ifne absolute expression`

Assembles the following section of code if the argument is not equal to zero (in other words, this is equivalent to `.if`).

`.ifnes string1,string2`

Like `.ifeqs`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

8.44. `.incbin "file"[,skip[,count]]`

The `incbin` directive includes `file` verbatim at the current location. You can control the search paths used with the `-I` command-line option (refer to Chapter 3 *Command-Line Options*). Quotation marks are required around `file`.

The `skip` argument skips a number of bytes from the start of the `file`. The `count` argument indicates the maximum number of bytes to read. Note that the data is not aligned in any way, so it is the user's responsibility to make sure that proper alignment is provided both before and after the `incbin` directive.

8.45. `.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 (refer to Chapter 3 *Command-Line Options*). Quotation marks are required around `file`.

8.46. `.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.

8.47. `.internal names`

This one of the ELF visibility directives. The other two are `.hidden` (refer to Section 8.40 *`.hidden names`*) and `.protected` (refer to Section 8.66 *`.protected names`*).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to `internal` which means that the symbols are considered to be `hidden` (i.e., not visible to other components), and that some extra, processor specific processing must also be performed upon the symbols as well.

8.48. `.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@-
```

8.49. `.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@-
```

8.50. `.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 (refer to Section 8.39 *`.global symbol`, `.globl symbol`*), 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.

8.51. `.lflags`

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

8.52. `.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.

8.53. .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.

8.54. .ln line-number

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

8.55. .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. MRI mode.

8.56. .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; Chapter 3 *Command-Line Options*), the initial value of the listing counter is one.

8.57. .long expressions

`.long` is the same as `.int`, Section 8.46 *`.int expressions`*.

8.58. `.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*.

8.59. `.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.

8.60. `.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.

8.61. `.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.

8.62. `.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.

8.63. `.previous`

This is one of the ELF section stack manipulation directives. The others are `.section` (refer to Section 8.74 *.section name*), `.subsection` (refer to Section 8.85 *.subsection name*), `.pushsection` (refer to Section 8.69 *.pushsection name, subsection*), and `.popsection` (refer to Section 8.64 *.popsection*).

This directive swaps the current section (and subsection) with most recently referenced section (and subsection) prior to this one. Multiple `.previous` directives in a row will flip between two sections (and their subsections).

In terms of the section stack, this directive swaps the current section with the top section on the section stack.

8.64. `.popsection`

This is one of the ELF section stack manipulation directives. The others are `.section` (refer to Section 8.74 *.section name*), `.subsection` (refer to Section 8.85 *.subsection name*), `.pushsection` (refer to Section 8.69 *.pushsection name, subsection*), and `.previous` (refer to Section 8.63 *.previous*).

This directive replaces the current section (and subsection) with the top section (and subsection) on the section stack. This section is popped off the stack.

8.65. `.print string`

`as` will print `string` on the standard output during assembly. You must put `string` in double quotes.

8.66. `.protected names`

This one of the ELF visibility directives. The other two are `.hidden` (refer to Section 8.40 *.hidden names*) and `.internal` (refer to Section 8.47 *.internal names*).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to `protected` which means that any references to the symbols from within the components that defines them must be resolved to the definition in that component, even if a definition in another component would normally preempt this.

8.67. `.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`.

8.68. `.purgem name`

Undefine the macro `name`, so that later uses of the string will not be expanded. Section 8.58 *.macro*.

8.69. `.pushsection name, subsection`

This is one of the ELF section stack manipulation directives. The others are `.section` (refer to Section 8.74 *.section name*), `.subsection` (refer to Section 8.85 *.subsection name*), `.popsection` (refer to Section 8.64 *.popsection*), and `.previous` (refer to Section 8.63 *.previous*).

This directive is a synonym for `.section`. It pushes the current section (and subsection) onto the top of the section stack, and then replaces the current section and subsection with `name` and `subsection`.

8.70. `.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.

8.71. `.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
```

8.72. `.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.

8.73. `.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.

8.74. `.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.

8.74.1. COFF Version

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

`s`

shared section (meaningful for PE targets)

`a`

ignored. (For compatibility with the ELF version)

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. Note the `n` and `w` flags remove attributes from the section, rather than adding them, so if they are used on their own it will be as if no flags had been specified at all.

If the optional argument to the `.section` directive is not quoted, it is taken as a subsegment number (refer to Section 5.4 *Sub-Sections*).

8.74.2. ELF Version

This is one of the ELF section stack manipulation directives. The others are `.subsection` (refer to Section 8.85 *.subsection name*), `.pushsection` (refer to Section 8.69 *.pushsection name, subsection*), `.popsection` (refer to Section 8.64 *.popsection*), and `.previous` (refer to Section 8.63 *.previous*).

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

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

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

`a`

section is allocatable

`w`

section is writable

`x`

section is executable

`M`

section is mergeable

`S`

section contains zero terminated strings

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)

Note on targets where the `@` character is the start of a comment (eg ARM) then another character is used instead. For example the ARM port uses the `%` character.

If `flags` contains `M` flag, `type` argument must be specified as well as `entsize` argument. Sections with `M` flag but not `S` flag must contain fixed size constants, each `entsize` octets long. Sections with both `M` and `S` must contain zero terminated strings where each character is `entsize` bytes long. The linker may remove duplicates within sections with the same name, same entity size and same flags.

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

This directive replaces the current section and subsection. The replaced section and subsection are pushed onto the section stack. See the contents of the gas testsuite directory `gas/testsuite/gas/elf` for some examples of how this directive and the other section stack directives work.

8.75. `.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 (refer to Section 6.5 *Symbol Attributes*).

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`.

8.76. `.short expressions`

`.short` is normally the same as `.word`. Section 8.97 *`.word expressions`*.

In some configurations, however, `.short` and `.word` generate numbers of different lengths; Chapter 9 *Machine Dependent Features*.

8.77. `.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. Chapter 9 *Machine Dependent Features*.

8.78. `.size`

This directive is used to set the size associated with a symbol.

8.78.1. COFF Version

For COFF targets, the `.size` directive is only permitted inside `.def`/`.endef` pairs. It is used like this:

```
.size expression
```

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

8.78.2. ELF Version

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

```
.size name , expression
```

This directive sets the size associated with a symbol `name`. The size in bytes is computed from `expression` which can make use of label arithmetic. This directive is typically used to set the size of function symbols.

8.79. `.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. `.uleb128`.

8.80. `.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`.

8.81. `.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. Section 19.5 *HPPA Assembler Directives*, 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` Chapter 9 *Machine Dependent Features*.

8.82. `.stabd, .stabn, .stabs`

There are three directives that begin `.stab`. All emit symbols (refer to Chapter 6 *Symbols*), 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.

8.83. `.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 4.6.1.1 *Strings*.

8.84. `.struct expression`

Switch to the absolute section, and set the section offset to `expression`, which must be an absolute expression. You might use this as follows:

```
        .struct 0
field1:
        .struct field1 + 4
field2:
        .struct field2 + 4
field3:
```

This would define the symbol `field1` to have the value 0, the symbol `field2` to have the value 4, and the symbol `field3` to have the value 8. Assembly would be left in the absolute section, and you would need to use a `.section` directive of some sort to change to some other section before further assembly.

8.85. `.subsection name`

This is one of the ELF section stack manipulation directives. The others are `.section` (refer to Section 8.74 *.section name*), `.pushsection` (refer to Section 8.69 *.pushsection name, subsection*), `.popsection` (refer to Section 8.64 *.popsection*), and `.previous` (refer to Section 8.63 *.previous*).

This directive replaces the current subsection with `name`. The current section is not changed. The replaced subsection is put onto the section stack in place of the then current top of stack subsection.

8.86. `.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 can be used like this:

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

If the symbol `name` is defined within the file being assembled, the `.symver` 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.

If the symbol `name` is not defined within the file being assembled, all references to `name` will be changed to `name2@nodename`. If no reference to `name` is made, `name2@nodename` will be removed from the symbol table.

Another usage of the `.symver` directive is:

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

In this case, the symbol `name` must exist and be defined within the file being assembled. It is similar to `name2@nodename`. The difference is `name2@@nodename` will also be used to resolve references to `name2` by the linker.

The third usage of the `.symver` directive is:

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

When `name` is not defined within the file being assembled, it is treated as `name2@nodename`. When `name` is defined within the file being assembled, the symbol name, `name`, will be changed to `name2@@nodename`.

8.87. `.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.

8.88. `.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.

8.89. `.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.

8.90. `.type`

This directive is used to set the type of a symbol.

8.90.1. COFF Version

For COFF targets, this directive is permitted only within `.def`/`.endef` pairs. It is used like this:

```
.type int
```

This 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.

8.90.2. ELF Version

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

```
.type name , type description
```

This sets the type of symbol `name` to be either a function symbol or an object symbol. There are five different syntaxes supported for the `type description` field, in order to provide compatibility with various other assemblers. The syntaxes supported are:

```
.type <name>,#function
.type <name>,#object

.type <name>,@function
.type <name>,@object

.type <name>,%function
.type <name>,%object

.type <name>,"function"
.type <name>,"object"

.type <name> STT_FUNCTION
.type <name> STT_OBJECT
```

8.91. `.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. `.sleb128`.

8.92. `.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.

8.93. `.version "string"`

This directive creates a `.note` section and places into it an ELF formatted note of type NT_VERSION. The note's name is set to `string`.

8.94. `.vtable_entry table, offset`

This directive finds or creates a symbol `table` and creates a `VTABLE_ENTRY` relocation for it with an addend of `offset`.

8.95. `.vtable_inherit child, parent`

This directive finds the symbol `child` and finds or creates the symbol `parent` and then creates a `VTABLE_INHERIT` relocation for the parent whose addend is the value of the child symbol. As a special case the parent name of `0` is treated as refering the `*ABS*` section.

8.96. `.weak names`

This directive sets the weak attribute on the comma separated list of symbol `names`. If the symbols do not already exist, they will be created.

8.97. `.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; Chapter 9 *Machine Dependent Features*), you can ignore this issue.

In order to assemble compiler output into something that works, `as` occasionally 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.

8.98. Deprecated Directives

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

.abort
.line

deprecated

Chapter 9.

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.

Chapter 10. AMD 29K Dependent Features

10.1. Options

`as` has no additional command-line options for the AMD 29K family.

10.2. Syntax

10.2.1. Macros

The macro syntax used on the AMD 29K is like that described in the AMD 29K Family Macro Assembler Specification. Normal `as` macros should still work.

10.2.2. Special Characters

`;` is the line comment character.

The character `?` is permitted in identifiers (but may not begin an identifier).

10.2.3. Register Names

General-purpose registers are represented by predefined symbols of the form `GRnnn` (for global registers) or `LRnnn` (for local registers), where `nnn` represents a number between `0` and `127`, written with no leading zeros. The leading letters may be in either upper or lower case; for example, `gr13` and `LR7` are both valid register names.

You may also refer to general-purpose registers by specifying the register number as the result of an expression (prefixed with `%%` to flag the expression as a register number):

```
%%expression
```

--where `expression` must be an absolute expression evaluating to a number between `0` and `255`. The range [0, 127] refers to global registers, and the range [128, 255] to local registers.

In addition, `as` understands the following protected special-purpose register names for the AMD 29K family:

```
vab    chd    pc0
ops    chc    pc1
cps    rbp    pc2
cfg    tmc    mmu
cha    tmr    lru
```

These unprotected special-purpose register names are also recognized:

```
ipc    alu    fpe
ipa    bp     inte
ipb    fc     fps
q      cr     exop
```

10.3. Floating Point

The AMD 29K family uses ieee floating-point numbers.

10.4. AMD 29K Machine Directives

`.block 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.

In other versions of the gnu assembler, this directive is called `.space`.

`.cputype`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.file`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

> *Warning:* in other versions of the gnu assembler, `.file` is used for the directive called `.app-file` in the AMD 29K support.

`.line`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.sect`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.use section name`

Establishes the section and subsection for the following code; `section name` may be one of `.text`, `.data`, `.data1`, or `.lit`. With one of the first three `section name` options, `.use` is equivalent to the machine directive `section name`; the remaining case, `.use .lit`, is the same as `.data 200`.

10.5. Opcodes

`as` implements all the standard AMD 29K opcodes. No additional pseudo-instructions are needed on this family.

For information on the 29K machine instruction set, see [Am29000 User's Manual], Advanced Micro Devices, Inc.

Chapter 11. Alpha Dependent Features

11.1. Notes

The documentation here is primarily for the ELF object format. `as` also supports the ECOFF and EVAX formats, but features specific to these formats are not yet documented.

11.2. Options

`-mcpu`

This option specifies the target processor. If an attempt is made to assemble an instruction which will not execute on the target processor, the assembler may either expand the instruction as a macro or issue an error message. This option is equivalent to the `.arch` directive.

The following processor names are recognized: `21064`, `21064a`, `21066`, `21068`, `21164`, `21164a`, `21164pc`, `21264`, `21264a`, `21264b`, `ev4`, `ev5`, `lca45`, `ev5`, `ev56`, `pca56`, `ev6`, `ev67`, `ev68`. The special name `all` may be used to allow the assembler to accept instructions valid for any Alpha processor.

In order to support existing practice in OSF/1 with respect to `.arch`, and existing practice within `MILO` (the Linux ARC bootloader), the numbered processor names (e.g. 21064) enable the processor-specific PALcode instructions, while the "electro-vlasic" names (e.g. `ev4`) do not.

`-mdebug`
`-no-mdebug`

Enables or disables the generation of `.mdebug` encapsulation for stabs directives and procedure descriptors. The default is to automatically enable `.mdebug` when the first stabs directive is seen.

`-relax`

This option forces all relocations to be put into the object file, instead of saving space and resolving some relocations at assembly time. Note that this option does not propagate all symbol arithmetic into the object file, because not all symbol arithmetic can be represented. However, the option can still be useful in specific applications.

`-g`

This option is used when the compiler generates debug information. When `gcc` is using `mips-tfile` to generate debug information for ECOFF, local labels must be passed through to the object file. Otherwise this option has no effect.

`-Gsize`

A local common symbol larger than `size` is placed in `.bss`, while smaller symbols are placed in `.sbss`.

`-F`
`-32addr`

These options are ignored for backward compatibility.

11.3. Syntax

The assembler syntax closely follow the Alpha Reference Manual; assembler directives and general syntax closely follow the OSF/1 and OpenVMS syntax, with a few differences for ELF.

11.3.1. Special Characters

`#` is the line comment character.

`;` can be used instead of a newline to separate statements.

11.3.2. Register Names

The 32 integer registers are refered to as `$n` or `$rn`. In addition, registers 15, 28, 29, and 30 may be refered to by the symbols `$fp`, `$at`, `$gp`, and `$sp` respectively.

The 32 floating-point registers are refered to as `$fn`.

11.3.3. Relocations

Some of these relocations are available for ECOFF, but mostly only for ELF. They are modeled after the relocation format introduced in Digial Unix 4.0, but there are additions.

The format is `!tag` or `!tag!number` where `tag` is the name of the relocation. In some cases `number` is used to relate specific instructions.

The relocation is placed at the end of the instruction like so:

```
ldah  $0,a($29)    !gprelhigh
lda   $0,a($0)     !gprellow
ldq   $1,b($29)    !literal!100
ldl   $2,0($1)     !lituse_base!100
```

`!literal`
`!literal!N`

Used with an `ldq` instruction to load the address of a symbol from the GOT.

A sequence number `N` is optional, and if present is used to pair `lituse` relocations with this `literal` relocation. The `lituse` relocations are used by the linker to optimize the code based on the final location of the symbol.

Note that these optimizations are dependent on the data flow of the program. Therefore, if *any* `lituse` is paired with a `literal` relocation, then *all* uses of the register set by the `literal` instruction must also be marked with `lituse` relocations. This is because the original `literal` instruction may be deleted or transformed into another instruction.

Also note that there may be a one-to-many relationship between `literal` and `lituse`, but not a many-to-one. That is, if there are two code paths that load up the same address and feed the value to a single use, then the use may not use a `lituse` relocation.

`!lituse_base!N`

Used with any memory format instruction (e.g. `ldl`) to indicate that the literal is used for an address load. The offset field of the instruction must be zero. During relaxation, the code may be altered to use a gp-relative load.

`!lituse_jsr!N`

Used with a register branch format instruction (e.g. `jsr`) to indicate that the literal is used for a call. During relaxation, the code may be altered to use a direct branch (e.g. `bsr`).

`!lituse_bytoff!N`

Used with a byte mask instruction (e.g. `extbl`) to indicate that only the low 3 bits of the address are relevant. During relaxation, the code may be altered to use an immediate instead of a register shift.

`!lituse_addr!N`

Used with any other instruction to indicate that the original address is in fact used, and the original `ldq` instruction may not be altered or deleted. This is useful in conjunction with `lituse_jsr` to test whether a weak symbol is defined.

```
ldq  $27,foo($29)    !literal!1
beq  $27,is_undef    !lituse_addr!1
jsr  $26,($27),foo   !lituse_jsr!1
```

`!lituse_tlsgd!N`

Used with a register branch format instruction to indicate that the literal is the call to `__tls_get_addr` used to compute the address of the thread-local storage variable whose descriptor was loaded with `!tlsgd!N`.

`!lituse_tlsldm!N`

Used with a register branch format instruction to indicate that the literal is the call to `__tls_get_addr` used to compute the address of the base of the thread-local storage block for the current module. The descriptor for the module must have been loaded with `!tlsldm!N`.

`!gpdisp!N`

Used with `ldah` and `lda` to load the GP from the current address, a-la the `ldgp` macro. The source register for the `ldah` instruction must contain the address of the `ldah` instruction. There must be exactly one `lda` instruction paired with the `ldah` instruction, though it may appear anywhere in the instruction stream. The immediate operands must be zero.

```
bsr  $26,foo
ldah $29,0($26)      !gpdisp!1
lda  $29,0($29)      !gpdisp!1
```

`!gprelhigh`

Used with an `ldah` instruction to add the high 16 bits of a 32-bit displacement from the GP.

`!gprellow`

Used with any memory format instruction to add the low 16 bits of a 32-bit displacement from the GP.

`!gprel`

Used with any memory format instruction to add a 16-bit displacement from the GP.

`!samegp`

Used with any branch format instruction to skip the GP load at the target address. The referenced symbol must have the same GP as the source object file, and it must be declared to either not use `$27` or perform a standard GP load in the first two instructions via the `.prologue` directive.

`!tlsgd`
`!tlsgd!N`

Used with an `lda` instruction to load the address of a TLS descriptor for a symbol in the GOT.

The sequence number `N` is optional, and if present it used to pair the descriptor load with both the `literal` loading the address of the `__tls_get_addr` function and the `lituse_tlsgd` marking the call to that function.

For proper relaxation, both the `tlsgd`, `literal` and `lituse` relocations must be in the same extended basic block. That is, the relocation with the lowest address must be executed first at runtime.

`!tlsldm`
`!tlsldm!N`

Used with an `lda` instruction to load the address of a TLS descriptor for the current module in the GOT.

Similar in other respects to `tlsgd`.

`!gotdtprel`

Used with an `ldq` instruction to load the offset of the TLS symbol within its module's thread-local storage block. Also known as the dynamic thread pointer offset or dtp-relative offset.

`!dtprelhi`
`!dtprello`
`!dtprel`

Like `gprel` relocations except they compute dtp-relative offsets.

`!gottprel`

Used with an `ldq` instruction to load the offset of the TLS symbol from the thread pointer. Also known as the tp-relative offset.

`!tprelhi`
`!tprello`
`!tprel`

Like `gprel` relocations except they compute tp-relative offsets.

11.4. Floating Point

The Alpha family uses both ieee and VAX floating-point numbers.

11.5. Alpha Assembler Directives

`as` for the Alpha supports many additional directives for compatibility with the native assembler. This section describes them only briefly.

These are the additional directives in `as` for the Alpha:

`.arch cpu`

Specifies the target processor. This is equivalent to the `-mcpu` command-line option. Options, for a list of values for `cpu`.

`.ent function[, n]`

Mark the beginning of `function`. An optional number may follow for compatibility with the OSF/1 assembler, but is ignored. When generating `.mdebug` information, this will create a procedure descriptor for the function. In ELF, it will mark the symbol as a function a-la the generic `.type` directive.

`.end function`

Mark the end of `function`. In ELF, it will set the size of the symbol a-la the generic `.size` directive.

`.mask mask, offset`

Indicate which of the integer registers are saved in the current function's stack frame. `mask` is interpreted a bit mask in which bit `n` set indicates that register `n` is saved. The registers are saved in a block located `offset` bytes from the *canonical frame address* (CFA) which is the value of the stack pointer on entry to the function. The registers are saved sequentially, except that the return address register (normally `$26`) is saved first.

This and the other directives that describe the stack frame are currently only used when generating `.mdebug` information. They may in the future be used to generate DWARF2 `.debug_frame` unwind information for hand written assembly.

`.fmask mask, offset`

Indicate which of the floating-point registers are saved in the current stack frame. The `mask` and `offset` parameters are interpreted as with `.mask`.

`.frame framereg, frameoffset, retreg[, argoffset]`

Describes the shape of the stack frame. The frame pointer in use is `framereg`; normally this is either `$fp` or `$sp`. The frame pointer is `frameoffset` bytes below the CFA. The return address is initially located in `retreg` until it is saved as indicated in `.mask`. For compatibility with OSF/1 an optional `argoffset` parameter is accepted and ignored. It is believed to indicate the offset from the CFA to the saved argument registers.

`.prologue n`

Indicate that the stack frame is set up and all registers have been spilled. The argument `n` indicates whether and how the function uses the incoming *procedure vector* (the address of the called function) in `$27`. 0 indicates that `$27` is not used; 1 indicates that the first two instructions of the function use `$27` to perform a load of the GP register; 2 indicates that `$27` is used in some non-standard way and so the linker cannot elide the load of the procedure vector during relaxation.

`.usepv function, which`

Used to indicate the use of the `$27` register, similar to `.prologue`, but without the other semantics of needing to be inside an open `.ent`/`.end` block.

The `which` argument should be either `no`, indicating that `$27` is not used, or `std`, indicating that the first two instructions of the function perform a GP load.

One might use this directive instead of `.prologue` if you are also using dwarf2 CFI directives.

`.gprel32 expression`

Computes the difference between the address in `expression` and the GP for the current object file, and stores it in 4 bytes. In addition to being smaller than a full 8 byte address, this also does not require a dynamic relocation when used in a shared library.

`.t_floating expression`

Stores `expression` as an ieee double precision value.

`.s_floating expression`

Stores `expression` as an ieee single precision value.

`.f_floating expression`

Stores `expression` as a VAX F format value.

`.g_floating expression`

Stores `expression` as a VAX G format value.

`.d_floating expression`

Stores `expression` as a VAX D format value.

`.set feature`

Enables or disables various assembler features. Using the positive name of the feature enables while using `nofeature` disables.

`at`

Indicates that macro expansions may clobber the *assembler temporary* (`$at` or `$28`) register. Some macros may not be expanded without this and will generate an error message if `noat` is in effect. When `at` is in effect, a warning will be generated if `$at` is used by the programmer.

`macro`

Enables the expansion of macro instructions. Note that variants of real instructions, such as `br label` vs `br $31,label` are considered alternate forms and not macros.

`move`
`reorder`
`volatile`

These control whether and how the assembler may re-order instructions. Accepted for compatibility with the OSF/1 assembler, but `as` does not do instruction scheduling, so these features are ignored.

The following directives are recognized for compatibility with the OSF/1 assembler but are ignored.

```
.proc          .aproc
.reguse        .livereg
.option        .aent
.ugen          .eflag
.alias         .noalias
```

11.6. Opcodes

For detailed information on the Alpha machine instruction set, see the ftp://ftp.digital.com/pub/Digital/info/semiconductor/literature/alphaahb.pdfAlpha Architecture Handbook.

Chapter 12. ARC Dependent Features

12.1. Options

`-marc[5|6|7|8]`

This option selects the core processor variant. Using `-marc` is the same as `-marc6`, which is also the default.

`arc5`

Base instruction set.

`arc6`

Jump-and-link (jl) instruction. No requirement of an instruction between setting flags and conditional jump. For example:

```
mov.f r0,r1
beq   foo
```

`arc7`

Break (brk) and sleep (sleep) instructions.

`arc8`

Software interrupt (swi) instruction.

Note: the `.option` directive can to be used to select a core variant from within assembly code.

`-EB`

This option specifies that the output generated by the assembler should be marked as being encoded for a big-endian processor.

`-EL`

This option specifies that the output generated by the assembler should be marked as being encoded for a little-endian processor - this is the default.

12.2. Syntax

12.2.1. Special Characters

TODO

12.2.2. Register Names

TODO

12.3. Floating Point

The ARC core does not currently have hardware floating point support. Software floating point support is provided by `GCC` and uses ieee floating-point numbers.

12.4. ARC Machine Directives

The ARC version of `as` supports the following additional machine directives:

`.2byte expressions`

TODO

`.3byte expressions`

TODO

`.4byte expressions`

TODO

`.extAuxRegister name,address,mode`

TODO

```
.extAuxRegister mulhi,0x12,w
```

`.extCondCode suffix,value`

TODO

```
.extCondCode is_busy,0x14
```

`.extCoreRegister name,regnum,mode,shortcut`

TODO

```
.extCoreRegister mlo,57,r,can_shortcut
```

`.extInstruction name,opcode,subopcode,suffixclass,syntaxclass`

TODO

```
.extInstruction mul64,0x14,0x0,SUFFIX_COND,SYNTAX_3OP|OP1_MUST_BE_IMM
```

`.half expressions`

TODO

`.long expressions`

TODO

`.option arc|arc5|arc6|arc7|arc8`

The `.option` directive must be followed by the desired core version. Again `arc` is an alias for `arc6`.

Note: the `.option` directive overrides the command line option `-marc`; a warning is emitted when the version is not consistent between the two - even for the implicit default core version (arc6).

`.short expressions`

TODO

`.word expressions`

TODO

12.5. Opcodes

For information on the ARC instruction set, see [ARC Programmers Reference Manual], ARC Cores Ltd.

Chapter 13.
ARM Dependent Features

13.1. Options

`-mcpu=processor[+extension...]`

This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: `arm1`, `arm2`, `arm250`, `arm3`, `arm6`, `arm60`, `arm600`, `arm610`, `arm620`, `arm7`, `arm7m`, `arm7d`, `arm7dm`, `arm7di`, `arm7dmi`, `arm70`, `arm700`, `arm700i`, `arm710`, `arm710t`, `arm720`, `arm720t`, `arm740t`, `arm710c`, `arm7100`, `arm7500`, `arm7500fe`, `arm7t`, `arm7tdmi`, `arm8`, `arm810`, `strongarm`, `strongarm1`, `strongarm110`, `strongarm1100`, `strongarm1110`, `arm9`, `arm920`, `arm920t`, `arm922t`, `arm940t`, `arm9tdmi`, `arm9e`, `arm946e-r0`, `arm946e`, `arm966e-r0`, `arm966e`, `arm10t`, `arm10e`, `arm1020`, `arm1020t`, `arm1020e`, `ep9312` (ARM920 with Cirrus Maverick coprocessor), `i80200` (Intel XScale processor) `iwmmxt` (Intel(r) XScale processor with Wireless MMX(tm) technology coprocessor) and `xscale`. The special name `all` may be used to allow the assembler to accept instructions valid for any ARM processor.

In addition to the basic instruction set, the assembler can be told to accept various extension mnemonics that extend the processor using the co-processor instruction space. For example, `-mcpu=arm920+maverick` is equivalent to specifying `-mcpu=ep9312`. The following extensions are currently supported: `+maverick` `+iwmmxt` and `+xscale`.

`-march=architecture[+extension...]`

This option specifies the target architecture. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target architecture. The following architecture names are recognized: `armv1`, `armv2`, `armv2a`, `armv2s`, `armv3`, `armv3m`, `armv4`, `armv4xm`, `armv4t`, `armv4txm`, `armv5`, `armv5t`, `armv5txm`, `armv5te`, `armv5texp` `iwmmxt` and `xscale`. If both `-mcpu` and `-march` are specified, the assembler will use the setting for `-mcpu`.

The architecture option can be extended with the same instruction set extension options as the `-mcpu` option.

`-mfpu=floating-point-format`

This option specifies the floating point format to assemble for. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target floating point unit. The following format options are recognized: `softfpa`, `fpe`, `fpe2`, `fpe3`, `fpa`, `fpa10`, `fpa11`, `arm7500fe`, `softvfp`, `softvfp+vfp`, `vfp`, `vfp10`, `vfp10-r0`, `vfp9`, `vfpxd`, `arm1020t` and `arm1020e`.

In addition to determining which instructions are assembled, this option also affects the way in which the `.double` assembler directive behaves when assembling little-endian code.

The default is dependent on the processor selected. For Architecture 5 or later, the default is to assembler for VFP instructions; for earlier architectures the default is to assemble for FPA instructions.

`-mthumb`

This option specifies that the assembler should start assembling Thumb instructions; that is, it should behave as though the file starts with a `.code 16` directive.

`-mthumb-interwork`

This option specifies that the output generated by the assembler should be marked as supporting interworking.

`-mapcs [26|32]`

This option specifies that the output generated by the assembler should be marked as supporting the indicated version of the Arm Procedure. Calling Standard.

`-matpcs`

This option specifies that the output generated by the assembler should be marked as supporting the Arm/Thumb Procedure Calling Standard. If enabled this option will cause the assembler to create an empty debugging section in the object file called .arm.atpcs. Debuggers can use this to determine the ABI being used by.

`-mapcs-float`

This indicates the the floating point variant of the APCS should be used. In this variant floating point arguments are passed in FP registers rather than integer registers.

`-mapcs-reentrant`

This indicates that the reentrant variant of the APCS should be used. This variant supports position independent code.

`-EB`

This option specifies that the output generated by the assembler should be marked as being encoded for a big-endian processor.

`-EL`

This option specifies that the output generated by the assembler should be marked as being encoded for a little-endian processor.

`-k`

This option specifies that the output of the assembler should be marked as position-independent code (PIC).

`-moabi`

This indicates that the code should be assembled using the old ARM ELF conventions, based on a beta release release of the ARM-ELF specifications, rather than the default conventions which are based on the final release of the ARM-ELF specifications.

13.2. Syntax

13.2.1. Special Characters

The presence of a `@` on a line indicates the start of a comment that extends to the end of the current line. If a `#` appears as the first character of a line, the whole line is treated as a comment.

The `;` character can be used instead of a newline to separate statements.

Either `#` or `$` can be used to indicate immediate operands.

TODO Explain about /data modifier on symbols.

13.2.2. Register Names

TODO Explain about ARM register naming, and the predefined names.

13.3. Floating Point

The ARM family uses ieee floating-point numbers.

13.4. ARM Machine Directives

`.align expression [, expression]`

This is the generic `.align` directive. For the ARM however if the first argument is zero (ie no alignment is needed) the assembler will behave as if the argument had been 2 (ie pad to the next four byte boundary). This is for compatibility with ARM's own assembler.

`name .req register name`

This creates an alias for `register name` called `name`. For example:

```
foo .req r0
```

`.code [16|32]`

This directive selects the instruction set being generated. The value 16 selects Thumb, with the value 32 selecting ARM.

`.thumb`

This performs the same action as `.code 16`.

`.arm`

This performs the same action as `.code 32`.

`.force_thumb`

This directive forces the selection of Thumb instructions, even if the target processor does not support those instructions

`.thumb_func`

This directive specifies that the following symbol is the name of a Thumb encoded function. This information is necessary in order to allow the assembler and linker to generate correct code for interworking between Arm and Thumb instructions and should be used even if interworking is not going to be performed. The presence of this directive also implies `.thumb`

`.thumb_set`

This performs the equivalent of a `.set` directive in that it creates a symbol which is an alias for another symbol (possibly not yet defined). This directive also has the added property in that it marks the aliased symbol as being a thumb function entry point, in the same way that the `.thumb_func` directive does.

`.ltorg`

This directive causes the current contents of the literal pool to be dumped into the current section (which is assumed to be the .text section) at the current location (aligned to a word boundary). GAS maintains a separate literal pool for each section and each sub-section. The `.ltorg` directive will only affect the literal pool of the current section and sub-section. At the end of assembly all remaining, un-empty literal pools will automatically be dumped.

Note - older versions of GAS would dump the current literal pool any time a section change occurred. This is no longer done, since it prevents accurate control of the placement of literal pools.

`.pool`

This is a synonym for .ltorg.

13.5. Opcodes

`as` implements all the standard ARM opcodes. It also implements several pseudo opcodes, including several synthetic load instructions.

`NOP`

```
nop
```

This pseudo op will always evaluate to a legal ARM instruction that does nothing. Currently it will evaluate to MOV r0, r0.

`LDR`

```
ldr <register> , = <expression>
```

If expression evaluates to a numeric constant then a MOV or MVN instruction will be used in place of the LDR instruction, if the constant can be generated by either of these instructions. Otherwise the constant will be placed into the nearest literal pool (if it not already there) and a PC relative LDR instruction will be generated.

`ADR`

```
adr <register> <label>
```

This instruction will load the address of `label` into the indicated register. The instruction will evaluate to a PC relative ADD or SUB instruction depending upon where the label is located. If the label is out of range, or if it is not defined in the same file (and section) as the ADR instruction, then an error will be generated. This instruction will not make use of the literal pool.

ADRL

```
adrl <register> <label>
```

This instruction will load the address of `label` into the indicated register. The instruction will evaluate to one or two PC relative ADD or SUB instructions depending upon where the label is located. If a second instruction is not needed a NOP instruction will be generated in its place, so that this instruction is always 8 bytes long.

If the label is out of range, or if it is not defined in the same file (and section) as the ADRL instruction, then an error will be generated. This instruction will not make use of the literal pool.

For information on the ARM or Thumb instruction sets, see [ARM Software Development Toolkit Reference Manual], Advanced RISC Machines Ltd.

Chapter 14. CRIS Dependent Features

14.1. Command-line Options

The CRIS version of as has these machine-dependent command-line options.

The format of the generated object files can be either ELF or a.out, specified by the command-line options -emulation=crisaout and -emulation=criself. The default is ELF (criself), unless as has been configured specifically for a.out by using the configuration name cris-axis-aout.

There are two different link-incompatible ELF object file variants for CRIS, for use in environments where symbols are expected to be prefixed by a leading _ character and for environments without such a symbol prefix. The variant used for GNU/Linux port has no symbol prefix. Which variant to produce is specified by either of the options -underscore and -no-underscore. The default is -underscore. Since symbols in CRIS a.out objects are expected to have a _ prefix, specifying -no-underscore when generating a.out objects is an error. Besides the object format difference, the effect of this option is to parse register names differently. The -no-underscore option makes a $ register prefix mandatory.

The option -pic must be passed to as in order to recognize the symbol syntax used for ELF (SVR4 PIC) position-independent-code. This will also affect expansion of instructions. The expansion with -pic will use PC-relative rather than (slightly faster) absolute addresses in those expansions.

When -N is specified, as will emit a warning when a 16-bit branch instruction is expanded into a 32-bit multiple-instruction construct (refer to Section 14.2 *Instruction expansion*).

14.2. Instruction expansion

as will silently choose an instruction that fits the operand size for [register+constant] operands. For example, the offset 127 in move.d [r3+127],r4 fits in an instruction using a signed-byte offset. Similarly, move.d [r2+32767],r1 will generate an instruction using a 16-bit offset. For symbolic expressions and constants that do not fit in 16 bits including the sign bit, a 32-bit offset is generated.

For branches, as will expand from a 16-bit branch instruction into a sequence of instructions that can reach a full 32-bit address. Since this does not correspond to a single instruction, such expansions can optionally be warned about. Section 14.1 *Command-line Options*.

14.3. Syntax

There are different aspects of the CRIS assembly syntax.

14.3.1. Special Characters

The character # is a line comment character. It starts a comment if and only if it is placed at the beginning of a line.

A ; character starts a comment anywhere on the line, causing all characters up to the end of the line to be ignored.

A @ character is handled as a line separator equivalent to a logical new-line character (except in a comment), so separate instructions can be specified on a single line.

14.3.2. Symbols in position-independent code

When generating position-independent code (SVR4 PIC) for use in cris-axis-linux-gnu shared libraries, symbol suffixes are used to specify what kind of run-time symbol lookup will be used, expressed in the object as different *relocation types*. Usually, all absolute symbol values must be located in a table, the *global offset table*, leaving the code position-independent; independent of values of global symbols and independent of the address of the code. The suffix modifies the value of the symbol, into for example an index into the global offset table where the real symbol value is entered, or a PC-relative value, or a value relative to the start of the global offset table. All symbol suffixes start with the character `:` (omitted in the list below). Every symbol use in code or a read-only section must therefore have a PIC suffix to enable a useful shared library to be created. Usually, these constructs must not be used with an additive constant offset as is usually allowed, i.e. no 4 as in `symbol + 4` is allowed. This restriction is checked at link-time, not at assembly-time.

`GOT`

Attaching this suffix to a symbol in an instruction causes the symbol to be entered into the global offset table. The value is a 32-bit index for that symbol into the global offset table. The name of the corresponding relocation is `R_CRIS_32_GOT`. Example: `move.d [$r0+extsym:GOT],$r9`

`GOT16`

Same as for `GOT`, but the value is a 16-bit index into the global offset table. The corresponding relocation is `R_CRIS_16_GOT`. Example: `move.d [$r0+asymbol:GOT16],$r10`

`PLT`

This suffix is used for function symbols. It causes a *procedure linkage table*, an array of code stubs, to be created at the time the shared object is created or linked against, together with a global offset table entry. The value is a pc-relative offset to the corresponding stub code in the procedure linkage table. This arrangement causes the run-time symbol resolver to be called to look up and set the value of the symbol the first time the function is called (at latest; depending environment variables). It is only safe to leave the symbol unresolved this way if all references are function calls. The name of the relocation is `R_CRIS_32_PLT_PCREL`. Example: `add.d fnname:PLT,$pc`

`PLTG`

Like PLT, but the value is relative to the beginning of the global offset table. The relocation is `R_CRIS_32_PLT_GOTREL`. Example: `move.d fnname:PLTG,$r3`

`GOTPLT`

Similar to `PLT`, but the value of the symbol is a 32-bit index into the global offset table. This is somewhat of a mix between the effect of the `GOT` and the `PLT` suffix; the difference to `GOT` is that there will be a procedure linkage table entry created, and that the symbol is assumed to be a function entry and will be resolved by the run-time resolver as with `PLT`. The relocation is `R_CRIS_32_GOTPLT`. Example: `jsr [$r0+fnname:GOTPLT]`

`GOTPLT16`

A variant of `GOTPLT` giving a 16-bit value. Its relocation name is `R_CRIS_16_GOTPLT`. Example: `jsr [$r0+fnname:GOTPLT16]`

`GOTOFF`

This suffix must only be attached to a local symbol, but may be used in an expression adding an offset. The value is the address of the symbol relative to the start of the global offset table. The relocation name is `R_CRIS_32_GOTREL`. Example: `move.d [$r0+localsym:GOTOFF],r3`

14.3.3. Register names

A `$` character may always prefix a general or special register name in an instruction operand but is mandatory when the option `-no-underscore` is specified or when the `.syntax register_prefix` directive is in effect. Register names are case-insensitive.

14.3.4. Assembler Directives

There are a few CRIS-specific pseudo-directives in addition to the generic ones. Chapter 8 *Assembler Directives*. Constants emitted by pseudo-directives are in little-endian order for CRIS. There is no support for floating-point-specific directives for CRIS.

`.dword EXPRESSIONS`

The `.dword` directive is a synonym for `.int`, expecting zero or more EXPRESSIONS, separated by commas. For each expression, a 32-bit little-endian constant is emitted.

`.syntax ARGUMENT`

The `.syntax` directive takes as ARGUMENT one of the following case-sensitive choices.

`no_register_prefix`

The `.syntax no_register_prefix` directive makes a `$` character prefix on all registers optional. It overrides a previous setting, including the corresponding effect of the option `-no-underscore`. If this directive is used when ordinary symbols do not have a _ character prefix, care must be taken to avoid ambiguities whether an operand is a register or a symbol; using symbols with names the same as general or special registers then invoke undefined behavior.

`register_prefix`

This directive makes a `$` character prefix on all registers mandatory. It overrides a previous setting, including the corresponding effect of the option `-underscore`.

`leading_underscore`

This is an assertion directive, emitting an error if the `-no-underscore` option is in effect.

`no_leading_underscore`

This is the opposite of the `.syntax leading_underscore` directive and emits an error if the option `-underscore` is in effect.

Chapter 15. D10V Dependent Features

15.1. D10V Options

The Mitsubishi D10V version of `as` has a few machine dependent options.

`-O`

The D10V can often execute two sub-instructions in parallel. When this option is used, `as` will attempt to optimize its output by detecting when instructions can be executed in parallel.

`-nowarnswap`

To optimize execution performance, `as` will sometimes swap the order of instructions. Normally this generates a warning. When this option is used, no warning will be generated when instructions are swapped.

`-gstabs-packing`
`-no-gstabs-packing`

`as` packs adjacent short instructions into a single packed instruction. `-no-gstabs-packing` turns instruction packing off if `-gstabs` is specified as well; `-gstabs-packing` (the default) turns instruction packing on even when `-gstabs` is specified.

15.2. Syntax

The D10V syntax is based on the syntax in Mitsubishi's D10V architecture manual. The differences are detailed below.

15.2.1. Size Modifiers

The D10V version of `as` uses the instruction names in the D10V Architecture Manual. However, the names in the manual are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. How does the assembler pick the correct form? `as` will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, it will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either `.s` (short) or `.l` (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol that is defined later in your program, you can write `bra.s foo`. Objdump and GDB will always append `.s` or `.l` to instructions which have both short and long forms.

15.2.2. Sub-Instructions

The D10V assembler takes as input a series of instructions, either one-per-line, or in the special two-per-line format described in the next section. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one. Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary.

If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols described in the next section.

15.2.3. Special Characters

`;` and `#` are the line comment characters. Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially. To specify the executing order, use the following symbols:

`->`

Sequential with instruction on the left first.

`<-`

Sequential with instruction on the right first.

`||`

Parallel

The D10V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line. For example

`abs a1 -> abs r0`

Execute these sequentially. The instruction on the right is in the right container and is executed second.

`abs r0 <- abs a1`

Execute these reverse-sequentially. The instruction on the right is in the right container, and is executed first.

`ld2w r2,@r8+ || mac a0,r0,r7`

Execute these in parallel.

```
ld2w r2,@r8+ ||
mac a0,r0,r7
```

Two-line format. Execute these in parallel.

```
ld2w r2,@r8+
mac a0,r0,r7
```

Two-line format. Execute these sequentially. Assembler will put them in the proper containers.

```
ld2w r2,@r8+ ->
mac a0,r0,r7
```

Two-line format. Execute these sequentially. Same as above but second instruction will always go into right container.

Since `$` has no special meaning, you may use it in symbol names.

15.2.4. Register Names

You can use the predefined symbols `r0` through `r15` to refer to the D10V registers. You can also use `sp` as an alias for `r15`. The accumulators are `a0` and `a1`. There are special register-pair names that may optionally be used in opcodes that require even-numbered registers. Register names are not case sensitive.

Register Pairs

`r0-r1`
`r2-r3`
`r4-r5`
`r6-r7`
`r8-r9`
`r10-r11`
`r12-r13`
`r14-r15`

Registers

The D10V also has predefined symbols for these control registers and status bits:

`psw`
: Processor Status Word

`bpsw`
: Backup Processor Status Word

`pc`
: Program Counter

`bpc`
: Backup Program Counter

`rpt_c`
: Repeat Count

`rpt_s`
: Repeat Start address

`rpt_e`
: Repeat End address

`mod_s`
: Modulo Start address

`mod_e`
: Modulo End address

`iba`
: Instruction Break Address

`f0`

Flag 0

`f1`

Flag 1

`c`

Carry flag

15.2.5. Addressing Modes

`as` understands the following addressing modes for the D10V. `Rn` in the following refers to any of the numbered registers, but *not* the control registers.

`Rn`

Register direct

`@Rn`

Register indirect

`@Rn+`

Register indirect with post-increment

`@Rn-`

Register indirect with post-decrement

`@-SP`

Register indirect with pre-decrement

`@(disp, Rn)`

Register indirect with displacement

`addr`

PC relative address (for branch or rep).

`#imm`

Immediate data (the `#` is optional and ignored)

15.2.6. @WORD Modifier

Any symbol followed by `@word` will be replaced by the symbol's value shifted right by 2. This is used in situations such as loading a register with the address of a function (or any other code fragment). For example, if you want to load a register with the location of the function `main` then jump to that function, you could do it as follows:

```
ldi     r2, main@word
jmp     r2
```

15.3. Floating Point

The D10V has no hardware floating point, but the `.float` and `.double` directives generates ieee floating-point numbers for compatibility with other development tools.

15.4. Opcodes

For detailed information on the D10V machine instruction set, see [D10V Architecture: A VLIW Microprocessor for Multimedia Applications] (Mitsubishi Electric Corp.). `as` implements all the standard D10V opcodes. The only changes are those described in the section on size modifiers

Chapter 16. D30V Dependent Features

16.1. D30V Options

The Mitsubishi D30V version of `as` has a few machine dependent options.

`-O`

: The D30V can often execute two sub-instructions in parallel. When this option is used, `as` will attempt to optimize its output by detecting when instructions can be executed in parallel.

`-n`

: When this option is used, `as` will issue a warning every time it adds a nop instruction.

`-N`

: When this option is used, `as` will issue a warning if it needs to insert a nop after a 32-bit multiply before a load or 16-bit multiply instruction.

16.2. Syntax

The D30V syntax is based on the syntax in Mitsubishi's D30V architecture manual. The differences are detailed below.

16.2.1. Size Modifiers

The D30V version of `as` uses the instruction names in the D30V Architecture Manual. However, the names in the manual are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. How does the assembler pick the correct form? `as` will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, it will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either `.s` (short) or `.l` (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol that is defined later in your program, you can write `bra.s foo`. Objdump and GDB will always append `.s` or `.l` to instructions which have both short and long forms.

16.2.2. Sub-Instructions

The D30V assembler takes as input a series of instructions, either one-per-line, or in the special two-per-line format described in the next section. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one. Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary.

If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols described in the next section.

16.2.3. Special Characters

`;` and `#` are the line comment characters. Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially unless you use the `-O` option.

To specify the executing order, use the following symbols:

`->`

Sequential with instruction on the left first.

`<-`

Sequential with instruction on the right first.

`||`

Parallel

The D30V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line. For example

```
abs r2,r3 -> abs r4,r5
```

Execute these sequentially. The instruction on the right is in the right container and is executed second.

```
abs r2,r3 <- abs r4,r5
```

Execute these reverse-sequentially. The instruction on the right is in the right container, and is executed first.

```
abs r2,r3 || abs r4,r5
```

Execute these in parallel.

```
ldw r2,@(r3,r4) ||
mulx r6,r8,r9
```

Two-line format. Execute these in parallel.

```
mulx a0,r8,r9
stw r2,@(r3,r4)
```

Two-line format. Execute these sequentially unless `-O` option is used. If the `-O` option is used, the assembler will determine if the instructions could be done in parallel (the above two instructions can be done in parallel), and if so, emit them as parallel instructions. The assembler will put them in the proper containers. In the above example, the assembler will put the `stw` instruction in left container and the `mulx` instruction in the right container.

```
stw r2,@(r3,r4) ->
mulx a0,r8,r9
```

Two-line format. Execute the `stw` instruction followed by the `mulx` instruction sequentially. The first instruction goes in the left container and the second instruction goes into right container. The assembler will give an error if the machine ordering constraints are violated.

```
stw r2,@(r3,r4) <-
mulx a0,r8,r9
```

Same as previous example, except that the `mulx` instruction is executed before the `stw` instruction.

Since `$` has no special meaning, you may use it in symbol names.

16.2.4. Guarded Execution

`as` supports the full range of guarded execution directives for each instruction. Just append the directive after the instruction proper. The directives are:

`/tx`

Execute the instruction if flag f0 is true.

`/fx`

Execute the instruction if flag f0 is false.

`/xt`

Execute the instruction if flag f1 is true.

`/xf`

Execute the instruction if flag f1 is false.

`/tt`

Execute the instruction if both flags f0 and f1 are true.

`/tf`

Execute the instruction if flag f0 is true and flag f1 is false.

16.2.5. Register Names

You can use the predefined symbols `r0` through `r63` to refer to the D30V registers. You can also use `sp` as an alias for `r63` and `link` as an alias for `r62`. The accumulators are `a0` and `a1`.

The D30V also has predefined symbols for these control registers and status bits:

`psw`

Processor Status Word

`bpsw`

Backup Processor Status Word

`pc`

Program Counter

`bpc`

Backup Program Counter

`rpt_c`

Repeat Count

`rpt_s`

Repeat Start address

`rpt_e`
: Repeat End address

`mod_s`
: Modulo Start address

`mod_e`
: Modulo End address

`iba`
: Instruction Break Address

`f0`
: Flag 0

`f1`
: Flag 1

`f2`
: Flag 2

`f3`
: Flag 3

`f4`
: Flag 4

`f5`
: Flag 5

`f6`
: Flag 6

`f7`
: Flag 7

`s`
: Same as flag 4 (saturation flag)

`v`
: Same as flag 5 (overflow flag)

`va`
: Same as flag 6 (sticky overflow flag)

`c`
: Same as flag 7 (carry/borrow flag)

`b`

Same as flag 7 (carry/borrow flag)

16.2.6. Addressing Modes

`as` understands the following addressing modes for the D30V. `Rn` in the following refers to any of the numbered registers, but *not* the control registers.

`Rn`

Register direct

`@Rn`

Register indirect

`@Rn+`

Register indirect with post-increment

`@Rn-`

Register indirect with post-decrement

`@-SP`

Register indirect with pre-decrement

`@(disp, Rn)`

Register indirect with displacement

`addr`

PC relative address (for branch or rep).

`#imm`

Immediate data (the `#` is optional and ignored)

16.3. Floating Point

The D30V has no hardware floating point, but the `.float` and `.double` directives generates ieee floating-point numbers for compatibility with other development tools.

16.4. Opcodes

For detailed information on the D30V machine instruction set, see [D30V Architecture: A VLIW Microprocessor for Multimedia Applications] (Mitsubishi Electric Corp.). `as` implements all the standard D30V opcodes. The only changes are those described in the section on size modifiers

Chapter 17. H8/300 Dependent Features

17.1. Options

`as` has no additional command-line options for the Renesas (formerly Hitachi) H8/300 family.

17.2. Syntax

17.2.1. Special Characters

`;` is the line comment character.

`$` can be used instead of a newline to separate statements. Therefore *you may not use `$` in symbol names* on the H8/300.

17.2.2. Register Names

You can use predefined symbols of the form `rnh` and `rnl` to refer to the H8/300 registers as sixteen 8-bit general-purpose registers. `n` is a digit from `0` to `7`); for instance, both `r0h` and `r7l` are valid register names.

You can also use the eight predefined symbols `rn` to refer to the H8/300 registers as 16-bit registers (you must use this form for addressing).

On the H8/300H, you can also use the eight predefined symbols `ern` (`er0` ... `er7`) to refer to the 32-bit general purpose registers.

The two control registers are called `pc` (program counter; a 16-bit register, except on the H8/300H where it is 24 bits) and `ccr` (condition code register; an 8-bit register). `r7` is used as the stack pointer, and can also be called `sp`.

17.2.3. Addressing Modes

as understands the following addressing modes for the H8/300:

`rn`

Register direct

`@rn`

Register indirect

`@(d, rn)`
`@(d:16, rn)`
`@(d:24, rn)`

Register indirect: 16-bit or 24-bit displacement `d` from register `n`. (24-bit displacements are only meaningful on the H8/300H.)

`@rn+`

Register indirect with post-increment

`@-rn`

Register indirect with pre-decrement

`@aa`
`@aa:8`
`@aa:16`
`@aa:24`

Absolute address `aa`. (The address size `:24` only makes sense on the H8/300H.)

`#xx`
`#xx:8`
`#xx:16`
`#xx:32`

Immediate data `xx`. You may specify the `:8`, `:16`, or `:32` for clarity, if you wish; but `as` neither requires this nor uses it--the data size required is taken from context.

`@@aa`
`@@aa:8`

Memory indirect. You may specify the `:8` for clarity, if you wish; but `as` neither requires this nor uses it.

17.3. Floating Point

The H8/300 family has no hardware floating point, but the `.float` directive generates ieee floating-point numbers for compatibility with other development tools.

17.4. H8/300 Machine Directives

`as` has the following machine-dependent directives for the H8/300:

`.h8300h`

Recognize and emit additional instructions for the H8/300H variant, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

`.h8300s`

Recognize and emit additional instructions for the H8S variant, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

`.h8300hn`

Recognize and emit additional instructions for the H8/300H variant in normal mode, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

`.h8300sn`

Recognize and emit additional instructions for the H8S variant in normal mode, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

On the H8/300 family (including the H8/300H) `.word` directives generate 16-bit numbers.

17.5. Opcodes

For detailed information on the H8/300 machine instruction set, see [H8/300 Series Programming Manual]. For information specific to the H8/300H, see [H8/300H Series Programming Manual] (Renesas).

`as` implements all the standard H8/300 opcodes. No additional pseudo-instructions are needed on this family.

Four H8/300 instructions (`add`, `cmp`, `mov`, `sub`) are defined with variants using the suffixes `.b`, `.w`, and `.l` to specify the size of a memory operand. `as` supports these suffixes, but does not require them; since one of the operands is always a register, `as` can deduce the correct size.

For example, since `r0` refers to a 16-bit register,

```
mov    r0,@foo
is equivalent to           mov.w  r0,@foo
```

If you use the size suffixes, `as` issues a warning when the suffix and the register size do not match.

redhat

Chapter 18. H8/500 Dependent Features

18.1. Options

`as` has no additional command-line options for the Renesas (formerly Hitachi) H8/500 family.

18.2. Syntax

18.2.1. Special Characters

`!` is the line comment character.

`;` can be used instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

18.2.2. Register Names

You can use the predefined symbols `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, and `r7` to refer to the H8/500 registers.

The H8/500 also has these control registers:

`cp`

code pointer

`dp`

data pointer

`bp`

base pointer

`tp`

stack top pointer

`ep`

extra pointer

`sr`

status register

`ccr`

condition code register

All registers are 16 bits long. To represent 32 bit numbers, use two adjacent registers; for distant memory addresses, use one of the segment pointers (`cp` for the program counter; `dp` for `r0`-`r3`; `ep` for `r4` and `r5`; and `tp` for `r6` and `r7`.

18.2.3. Addressing Modes

as understands the following addressing modes for the H8/500:

`Rn`
: Register direct

`@Rn`
: Register indirect

`@(d:8, Rn)`
: Register indirect with 8 bit signed displacement

`@(d:16, Rn)`
: Register indirect with 16 bit signed displacement

`@-Rn`
: Register indirect with pre-decrement

`@Rn+`
: Register indirect with post-increment

`@aa:8`
: 8 bit absolute address

`@aa:16`
: 16 bit absolute address

`#xx:8`
: 8 bit immediate

`#xx:16`
: 16 bit immediate

18.3. Floating Point

The H8/500 family has no hardware floating point, but the `.float` directive generates ieee floating-point numbers for compatibility with other development tools.

18.4. H8/500 Machine Directives

`as` has no machine-dependent directives for the H8/500. However, on this platform the `.int` and `.word` directives generate 16-bit numbers.

18.5. Opcodes

For detailed information on the H8/500 machine instruction set, see [H8/500 Series Programming Manual] (Renesas M21T001).

`as` implements all the standard H8/500 opcodes. No additional pseudo-instructions are needed on this family.

Chapter 19.

HPPA Dependent Features

19.1. Notes

As a back end for gnu cc `as` has been throughly tested and should work extremely well. We have tested it only minimally on hand written assembly code and no one has tested it much on the assembly output from the HP compilers.

The format of the debugging sections has changed since the original `as` port (version 1.3X) was released; therefore, you must rebuild all HPPA objects and libraries with the new assembler so that you can debug the final executable.

The HPPA `as` port generates a small subset of the relocations available in the SOM and ELF object file formats. Additional relocation support will be added as it becomes necessary.

19.2. Options

`as` has no machine-dependent command-line options for the HPPA.

19.3. Syntax

The assembler syntax closely follows the HPPA instruction set reference manual; assembler directives and general syntax closely follow the HPPA assembly language reference manual, with a few noteworthy differences.

First, a colon may immediately follow a label definition. This is simply for compatibility with how most assembly language programmers write code.

Some obscure expression parsing problems may affect hand written code which uses the `spop` instructions, or code which makes significant use of the `!` line separator.

`as` is much less forgiving about missing arguments and other similar oversights than the HP assembler. `as` notifies you of missing arguments as syntax errors; this is regarded as a feature, not a bug.

Finally, `as` allows you to use an external symbol without explicitly importing the symbol. *Warning:* in the future this will be an error for HPPA targets.

Special characters for HPPA targets include:

`;` is the line comment character.

`!` can be used instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

19.4. Floating Point

The HPPA family uses ieee floating-point numbers.

19.5. HPPA Assembler Directives

`as` for the HPPA supports many additional directives for compatibility with the native assembler. This section describes them only briefly. For detailed information on HPPA-specific assembler directives, see [HP9000 Series 800 Assembly Language Reference Manual] (HP 92432-90001).

`as` does *not* support the following assembler directives described in the HP manual:

```
.endm           .liston
.enter          .locct
.leave          .macro
.listoff
```

Beyond those implemented for compatibility, `as` supports one additional assembler directive for the HPPA: `.param`. It conveys register argument locations for static functions. Its syntax closely follows the `.export` directive.

These are the additional directives in `as` for the HPPA:

`.block n`
`.blockz n`

Reserve `n` bytes of storage, and initialize them to zero.

`.call`

Mark the beginning of a procedure call. Only the special case with *no arguments* is allowed.

`.callinfo [ param=value, ... ] [ flag, ... ]`

Specify a number of parameters and flags that define the environment for a procedure.

`param` may be any of `frame` (frame size), `entry_gr` (end of general register range), `entry_fr` (end of float register range), `entry_sr` (end of space register range).

The values for `flag` are `calls` or `caller` (proc has subroutines), `no_calls` (proc does not call subroutines), `save_rp` (preserve return pointer), `save_sp` (proc preserves stack pointer), `no_unwind` (do not unwind this proc), `hpux_int` (proc is interrupt routine).

`.code`

Assemble into the standard section called `$TEXT$`, subsection `$CODE$`.

`.copyright "string"`

In the SOM object format, insert `string` into the object code, marked as a copyright string.

`.copyright "string"`

In the ELF object format, insert `string` into the object code, marked as a version string.

`.enter`

Not yet supported; the assembler rejects programs containing this directive.

`.entry`

Mark the beginning of a procedure.

`.exit`

Mark the end of a procedure.

`.export name [ ,typ ] [ ,param=r ]`

Make a procedure `name` available to callers. `typ`, if present, must be one of `absolute`, `code` (ELF only, not SOM), `data`, `entry`, `data`, `entry`, `millicode`, `plabel`, `pri_prog`, or `sec_prog`.

`param`, if present, provides either relocation information for the procedure arguments and result, or a privilege level. `param` may be `argwn` (where `n` ranges from `0` to `3`, and indicates one of four one-word arguments); `rtnval` (the procedure's result); or `priv_lev` (privilege level). For arguments or the result, `r` specifies how to relocate, and must be one of `no` (not relocatable), `gr` (argument is in general register), `fr` (in floating point register), or `fu` (upper half of float register). For `priv_lev`, `r` is an integer.

`.half n`

Define a two-byte integer constant `n`; synonym for the portable `as` directive `.short`.

`.import name [ ,typ ]`

Converse of `.export`; make a procedure available to call. The arguments use the same conventions as the first two arguments for `.export`.

`.label name`

Define `name` as a label for the current assembly location.

`.leave`

Not yet supported; the assembler rejects programs containing this directive.

`.origin lc`

Advance location counter to `lc`. Synonym for the `{No value for 'as'}` portable directive `.org`.

`.param name [ ,typ ] [ ,param=r ]`

Similar to `.export`, but used for static procedures.

`.proc`

Use preceding the first statement of a procedure.

`.procend`

Use following the last statement of a procedure.

`label .reg expr`

Synonym for `.equ`; define `label` with the absolute expression `expr` as its value.

`.space secname [ ,params ]`

Switch to section `secname`, creating a new section by that name if necessary. You may only use `params` when creating a new section, not when switching to an existing one. `secname` may identify a section by number rather than by name.

If specified, the list `params` declares attributes of the section, identified by keywords. The keywords recognized are `spnum=exp` (identify this section by the number `exp`, an absolute expression), `sort=exp` (order sections according to this sort key when linking; `exp` is an absolute expression), `unloadable` (section contains no loadable data), `notdefined` (this section defined elsewhere), and `private` (data in this section not available to other programs).

`.spnum secnam`

Allocate four bytes of storage, and initialize them with the section number of the section named `secnam`. (You can define the section number with the HPPA `.space` directive.)

`.string "str"`

Copy the characters in the string `str` to the object file. Section 4.6.1.1 *Strings*, for information on escape sequences you can use in `as` strings.

Warning! The HPPA version of `.string` differs from the usual `as` definition: it does *not* write a zero byte after copying `str`.

`.stringz "str"`

Like `.string`, but appends a zero byte after copying `str` to object file.

`.subspa name [ ,params ]`
`.nsubspa name [ ,params ]`

Similar to `.space`, but selects a subsection `name` within the current section. You may only specify `params` when you create a subsection (in the first instance of `.subspa` for this `name`).

If specified, the list `params` declares attributes of the subsection, identified by keywords. The keywords recognized are `quad=expr` ("quadrant" for this subsection), `align=expr` (alignment for beginning of this subsection; a power of two), `access=expr` (value for "access rights" field), `sort=expr` (sorting order for this subspace in link), `code_only` (subsection contains only code), `unloadable` (subsection cannot be loaded into memory), `common` (subsection is common block), `dup_comm` (initialized data may have duplicate names), or `zero` (subsection is all zeros, do not write in object file).

`.nsubspa` always creates a new subspace with the given name, even if one with the same name already exists.

`.version "str"`

Write `str` as version identifier in object code.

19.6. Opcodes

For detailed information on the HPPA machine instruction set, see [PA-RISC Architecture and Instruction Set Reference Manual] (HP 09740-90039).

Chapter 20. ESA/390 Dependent Features

20.1. Notes

The ESA/390 `as` port is currently intended to be a back-end for the gnu cc compiler. It is not HLASM compatible, although it does support a subset of some of the HLASM directives. The only supported binary file format is ELF; none of the usual MVS/VM/OE/USS object file formats, such as ESD or XSD, are supported.

When used with the gnu cc compiler, the ESA/390 `as` will produce correct, fully relocated, functional binaries, and has been used to compile and execute large projects. However, many aspects should still be considered experimental; these include shared library support, dynamically loadable objects, and any relocation other than the 31-bit relocation.

20.2. Options

`as` has no machine-dependent command-line options for the ESA/390.

20.3. Syntax

The opcode/operand syntax follows the ESA/390 Principles of Operation manual; assembler directives and general syntax are loosely based on the prevailing AT&T/SVR4/ELF/Solaris style notation. HLASM-style directives are *not* supported for the most part, with the exception of those described herein.

A leading dot in front of directives is optional, and the case of directives is ignored; thus for example, .using and USING have the same effect.

A colon may immediately follow a label definition. This is simply for compatibility with how most assembly language programmers write code.

`#` is the line comment character.

`;` can be used instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

Registers can be given the symbolic names r0..r15, fp0, fp2, fp4, fp6. By using thesse symbolic names, `as` can detect simple syntax errors. The name rarg or r.arg is a synonym for r11, rtca or r.tca for r12, sp, r.sp, dsa r.dsa for r13, lr or r.lr for r14, rbase or r.base for r3 and rpgt or r.pgt for r4.

`*` is the current location counter. Unlike `.` it is always relative to the last USING directive. Note that this means that expressions cannot use multiplication, as any occurrence of `*` will be interpreted as a location counter.

All labels are relative to the last USING. Thus, branches to a label always imply the use of base+displacement.

Many of the usual forms of address constants / address literals are supported. Thus,

```
.using *,r3
L r15,=A(some_routine)
LM r6,r7,=V(some_longlong_extern)
A r1,=F'12'
```

```
AH r0,=H'42'
ME r6,=E'3.1416'
MD r6,=D'3.14159265358979'
O r6,=XL4'cacad0d0'
.ltorg
```

should all behave as expected: that is, an entry in the literal pool will be created (or reused if it already exists), and the instruction operands will be the displacement into the literal pool using the current base register (as last declared with the `.using` directive).

20.4. Floating Point

The assembler generates only ieee floating-point numbers. The older floating point formats are not supported.

20.5. ESA/390 Assembler Directives

`as` for the ESA/390 supports all of the standard ELF/SVR4 assembler directives that are documented in the main part of this documentation. Several additional directives are supported in order to implement the ESA/390 addressing model. The most important of these are `.using` and `.ltorg`

These are the additional directives in `as` for the ESA/390:

`.dc`

A small subset of the usual DC directive is supported.

`.drop regno`

Stop using `regno` as the base register. The `regno` must have been previously declared with a `.using` directive in the same section as the current section.

`.ebcdic string`

Emit the EBCDIC equivalent of the indicated string. The emitted string will be null terminated. Note that the directives `.string` etc. emit ascii strings by default.

`EQU`

The standard HLASM-style EQU directive is not supported; however, the standard `as` directive .equ can be used to the same effect.

`.ltorg`

Dump the literal pool accumulated so far; begin a new literal pool. The literal pool will be written in the current section; in order to generate correct assembly, a `.using` must have been previously specified in the same section.

`.using expr,regno`

Use `regno` as the base register for all subsequent RX, RS, and SS form instructions. The `expr` will be evaluated to obtain the base address; usually, `expr` will merely be `*`.

This assembler allows two `.using` directives to be simultaneously outstanding, one in the `.text` section, and one in another section (typically, the `.data` section). This feature allows dynamically loaded objects to be implemented in a relatively straightforward way. A `.using` directive must always be specified in the `.text` section; this will specify the base register that will be used for branches in the `.text` section. A second `.using` may be specified in another section; this will specify the base register that is used for non-label address literals. When a second `.using` is specified, then the subsequent `.ltorg` must be put in the same section; otherwise an error will result.

Thus, for example, the following code uses `r3` to address branch targets and `r4` to address the literal pool, which has been written to the `.data` section. The is, the constants `=A(some_routine)`, `=H'42'` and `=E'3.1416'` will all appear in the `.data` section.

```
.data
 .using  LITPOOL,r4
.text
 BASR r3,0
 .using *,r3
        B       START
 .long LITPOOL
START:
 L r4,4(,r3)
 L r15,=A(some_routine)
 LTR r15,r15
 BNE LABEL
 AH r0,=H'42'
LABEL:
 ME r6,=E'3.1416'
.data
LITPOOL:
 .ltorg
```

Note that this dual-`.using` directive semantics extends and is not compatible with HLASM semantics. Note that this assembler directive does not support the full range of HLASM semantics.

20.6. Opcodes

For detailed information on the ESA/390 machine instruction set, see [ESA/390 Principles of Operation] (IBM Publication Number DZ9AR004).

Chapter 21. 80386 Dependent Features

The i386 version `as` supports both the original Intel 386 architecture in both 16 and 32-bit mode as well as AMD x86-64 architecture extending the Intel architecture to 64-bits.

21.1. Options

The i386 version of `as` has a few machine dependent options:

`-32 | -64`

Select the word size, either 32 bits or 64 bits. Selecting 32-bit implies Intel i386 architecture, while 64-bit implies AMD x86-64 architecture.

These options are only available with the ELF object file format, and require that the necessary BFD support has been included (on a 32-bit platform you have to add -enable-64-bit-bfd to configure enable 64-bit usage and use x86-64 as target platform).

21.2. AT&T Syntax versus Intel Syntax

`as` now supports assembly using Intel assembler syntax. `.intel_syntax` selects Intel mode, and `.att_syntax` switches back to the usual AT&T mode for compatibility with the output of `gcc`. Either of these directives may have an optional argument, `prefix`, or `noprefix` specifying whether registers require a `%` prefix. AT&T System V/386 assembler syntax is quite different from Intel syntax. We mention these differences because almost all 80386 documents use Intel syntax. Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by `$`; Intel immediate operands are undelimited (Intel `push 4` is AT&T `pushl $4`). AT&T register operands are preceded by `%`; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by `*`; they are undelimited in Intel syntax.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel `add eax, 4` is `addl $4, %eax`. The `source, dest` convention is maintained for compatibility with previous Unix assemblers. Note that instructions with more than one source operand, such as the `enter` instruction, do *not* have reversed order. Section 21.11 *AT&T Syntax bugs*.
- In AT&T syntax the size of memory operands is determined from the last character of the instruction mnemonic. Mnemonic suffixes of `b`, `w`, `l` and `q` specify byte (8-bit), word (16-bit), long (32-bit) and quadruple word (64-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (*not* the instruction mnemonics) with `byte ptr`, `word ptr`, `dword ptr` and `qword ptr`. Thus, Intel `mov al, byte ptr foo` is `movb foo, %al` in AT&T syntax.
- Immediate form long jumps and calls are `lcall/ljmp $section, $offset` in AT&T syntax; the Intel syntax is `call/jmp far section:offset`. Also, the far return instruction is `lret $stack-adjust` in AT&T syntax; Intel syntax is `ret far stack-adjust`.
- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

21.3. Instruction Naming

Instruction mnemonics are suffixed with one character modifiers which specify the size of operands. The letters `b`, `w`, `l` and `q` specify byte, word, long and quadruple word operands. If no suffix is specified by an instruction then `as` tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, `mov %ax, %bx` is equivalent to `movw %ax, %bx`; also, `mov $1, %bx` is equivalent to `movw $1, bx`. Note that this is incompatible with the AT&T Unix assembler which assumes that a missing mnemonic suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the mnemonic suffix.)

Almost all instructions have the same names in AT&T and Intel format. There are a few exceptions. The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend *from* and a size to zero extend *to*. This is accomplished by using two instruction mnemonic suffixes in AT&T syntax. Base names for sign extend and zero extend are `movs...` and `movz...` in AT&T syntax (`movsx` and `movzx` in Intel syntax). The instruction mnemonic suffixes are tacked on to this base name, the *from* suffix before the *to* suffix. Thus, `movsbl %al, %edx` is AT&T syntax for "move sign extend *from* %al *to* %edx." Possible suffixes, thus, are `bl` (from byte to long), `bw` (from byte to word), `wl` (from word to long), `bq` (from byte to quadruple word), `wq` (from word to quadruple word), and `lq` (from long to quadruple word).

The Intel-syntax conversion instructions

- `cbw` -- sign-extend byte in `%al` to word in `%ax`,
- `cwde` -- sign-extend word in `%ax` to long in `%eax`,
- `cwd` -- sign-extend word in `%ax` to long in `%dx:%ax`,
- `cdq` -- sign-extend dword in `%eax` to quad in `%edx:%eax`,
- `cdqe` -- sign-extend dword in `%eax` to quad in `%rax` (x86-64 only),
- `cdo` -- sign-extend quad in `%rax` to octuple in `%rdx:%rax` (x86-64 only),

are called `cbtw`, `cwtl`, `cwtd`, `cltd`, `cltq`, and `cqto` in AT&T naming. `as` accepts either naming for these instructions.

Far call/jump instructions are `lcall` and `ljmp` in AT&T syntax, but are `call far` and `jump far` in Intel convention.

21.4. Register Naming

Register operands are always prefixed with `%`. The 80386 registers consist of

- the 8 32-bit registers `%eax` (the accumulator), `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp` (the frame pointer), and `%esp` (the stack pointer).
- the 8 16-bit low-ends of these: `%ax`, `%bx`, `%cx`, `%dx`, `%di`, `%si`, `%bp`, and `%sp`.
- the 8 8-bit registers: `%ah`, `%al`, `%bh`, `%bl`, `%ch`, `%cl`, `%dh`, and `%dl` (These are the high-bytes and low-bytes of `%ax`, `%bx`, `%cx`, and `%dx`)
- the 6 section registers `%cs` (code section), `%ds` (data section), `%ss` (stack section), `%es`, `%fs`, and `%gs`.
- the 3 processor control registers `%cr0`, `%cr2`, and `%cr3`.
- the 6 debug registers `%db0`, `%db1`, `%db2`, `%db3`, `%db6`, and `%db7`.
- the 2 test registers `%tr6` and `%tr7`.

- the 8 floating point register stack `%st` or equivalently `%st(0)`, `%st(1)`, `%st(2)`, `%st(3)`, `%st(4)`, `%st(5)`, `%st(6)`, and `%st(7)`. These registers are overloaded by 8 MMX registers `%mm0`, `%mm1`, `%mm2`, `%mm3`, `%mm4`, `%mm5`, `%mm6` and `%mm7`.
- the 8 SSE registers registers `%xmm0`, `%xmm1`, `%xmm2`, `%xmm3`, `%xmm4`, `%xmm5`, `%xmm6` and `%xmm7`.

The AMD x86-64 architecture extends the register set by:

- enhancing the 8 32-bit registers to 64-bit: `%rax` (the accumulator), `%rbx`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rbp` (the frame pointer), `%rsp` (the stack pointer)
- the 8 extended registers `%r8`-`%r15`.
- the 8 32-bit low ends of the extended registers: `%r8d`-`%r15d`
- the 8 16-bit low ends of the extended registers: `%r8w`-`%r15w`
- the 8 8-bit low ends of the extended registers: `%r8b`-`%r15b`
- the 4 8-bit registers: `%sil`, `%dil`, `%bpl`, `%spl`.
- the 8 debug registers: `%db8`-`%db15`.
- the 8 SSE registers: `%xmm8`-`%xmm15`.

21.5. Instruction Prefixes

Instruction prefixes are used to modify the following instruction. They are used to repeat string instructions, to provide section overrides, to perform bus lock operations, and to change operand and address sizes. (Most instructions that normally operate on 32-bit operands will use 16-bit operands if the instruction has an "operand size" prefix.) Instruction prefixes are best written on the same line as the instruction they act upon. For example, the `scas` (scan string) instruction is repeated with:

```
repne scas %es:(%edi),%al
```

You may also place prefixes on the lines immediately preceding the instruction, but this circumvents checks that `as` does with prefixes, and will not work with all prefixes.

Here is a list of instruction prefixes:

- Section override prefixes `cs`, `ds`, `ss`, `es`, `fs`, `gs`. These are automatically added by specifying using the `section`:`memory-operand` form for memory references.
- Operand/Address size prefixes `data16` and `addr16` change 32-bit operands/addresses into 16-bit operands/addresses, while `data32` and `addr32` change 16-bit ones (in a `.code16` section) into 32-bit operands/addresses. These prefixes *must* appear on the same line of code as the instruction they modify. For example, in a 16-bit `.code16` section, you might write:
  ```
  addr32 jmpl *(%ebx)
  ```
- The bus lock prefix `lock` inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).
- The wait for coprocessor prefix `wait` waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387 combination.
- The `rep`, `repe`, and `repne` prefixes are added to string instructions to make them repeat `%ecx` times (`%cx` times if the current address size is 16-bits).

- The `rex` family of prefixes is used by x86-64 to encode extensions to i386 instruction set. The `rex` prefix has four bits -- an operand size overwrite (`64`) used to change operand size from 32-bit to 64-bit and X, Y and Z extensions bits used to extend the register set.

  You may write the `rex` prefixes directly. The `rex64xyz` instruction emits `rex` prefix with all the bits set. By omitting the `64`, `x`, `y` or `z` you may write other prefixes as well. Normally, there is no need to write the prefixes explicitly, since gas will automatically generate them based on the instruction operands.

21.6. Memory References

An Intel syntax indirect memory reference of the form

```
section:[base + index*scale + disp]
```

is translated into the AT&T syntax

```
section:disp(base, index, scale)
```

where `base` and `index` are the optional 32-bit base and index registers, `disp` is the optional displacement, and `scale`, taking the values 1, 2, 4, and 8, multiplies `index` to calculate the address of the operand. If no `scale` is specified, `scale` is taken to be 1. `section` specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Note that section overrides in AT&T syntax *must* be preceded by a `%`. If you specify a section override which coincides with the default section register, `as` does *not* output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

AT&T: `-4(%ebp)`, Intel: `[ebp - 4]`

: `base` is `%ebp`; `disp` is `-4`. `section` is missing, and the default section is used (`%ss` for addressing with `%ebp` as the base register). `index`, `scale` are both missing.

AT&T: `foo(,%eax,4)`, Intel: `[foo + eax*4]`

: `index` is `%eax` (scaled by a `scale` 4); `disp` is `foo`. All other fields are missing. The section register here defaults to `%ds`.

AT&T: `foo(,1)`; Intel `[foo]`

: This uses the value pointed to by `foo` as a memory operand. Note that `base` and `index` are both missing, but there is only *one* `,`. This is a syntactic exception.

AT&T: `%gs:foo`; Intel `gs:foo`

: This selects the contents of the variable `foo` with section register `section` being `%gs`.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with `*`. If no `*` is specified, `as` always chooses PC relative addressing for jump/call labels.

Any instruction that has a memory operand, but no register operand, *must* specify its size (byte, word, long, or quadruple) with an instruction mnemonic suffix (`b`, `w`, `l` or `q`, respectively).

The x86-64 architecture adds an RIP (instruction pointer relative) addressing. This addressing mode is specified by using `rip` as a base register. Only constant offsets are valid. For example:

AT&T: `1234(%rip)`, Intel: `[rip + 1234]`

Points to the address 1234 bytes past the end of the current instruction.

AT&T: `symbol(%rip)`, Intel: `[rip + symbol]`

Points to the `symbol` in RIP relative way, this is shorter than the default absolute addressing.

Other addressing modes remain unchanged in x86-64 architecture, except registers used are 64-bit instead of 32-bit.

21.7. Handling of Jump Instructions

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long displacement is used. We do not support word (16-bit) displacement jumps in 32-bit mode (i.e. prefixing the jump instruction with the `data16` instruction prefix), since the 80386 insists upon masking `%eip` to 16 bits after the word displacement is added. (See also Section 21.12 *Specifying CPU Architecture*)

Note that the `jcxz`, `jecxz`, `loop`, `loopz`, `loope`, `loopnz` and `loopne` instructions only come in byte displacements, so that if you use these instructions (`gcc` does not use them) you may get an error message (and incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding `jcxz foo` to

```
         jcxz cx_zero
         jmp cx_nonzero
cx_zero: jmp foo
cx_nonzero:
```

21.8. Floating Point

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty). These data types are 16-, 32-, and 64- bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an instruction mnemonic suffix and a constructor associated with it. Instruction mnemonic suffixes specify the operand's data type. Constructors build these data types into memory.

- Floating point constructors are `.float` or `.single`, `.double`, and `.tfloat` for 32-, 64-, and 80-bit formats. These correspond to instruction mnemonic suffixes `s`, `l`, and `t`. `t` stands for 80-bit (ten byte) real. The 80387 only supports this format via the `fldt` (load 80-bit real to stack top) and `fstpt` (store 80-bit real and pop stack) instructions.
- Integer constructors are `.word`, `.long` or `.int`, and `.quad` for the 16-, 32-, and 64-bit integer formats. The corresponding instruction mnemonic suffixes are `s` (single), `l` (long), and `q` (quad). As with the 80-bit real format, the 64-bit `q` format is only present in the `fildq` (load quad integer to stack top) and `fistpq` (store quad integer and pop stack) instructions.

Register to register operations should not use instruction mnemonic suffixes. `fstl %st, %st(1)` will give a warning, and be assembled as if you wrote `fst %st, %st(1)`, since all register to register operations use 80-bit floating point operands. (Contrast this with `fstl %st, mem`, which converts `%st` from 80-bit to 64-bit floating point format, then stores the result in the 4 byte location `mem`)

21.9. Intel's MMX and AMD's 3DNow! SIMD Operations

`as` supports Intel's MMX instruction set (SIMD instructions for integer data), available on Intel's Pentium MMX processors and Pentium II processors, AMD's K6 and K6-2 processors, Cyrix' M2 processor, and probably others. It also supports AMD's 3DNow! instruction set (SIMD instructions for 32-bit floating point data) available on AMD's K6-2 processor and possibly others in the future.

Currently, `as` does not support Intel's floating point SIMD, Katmai (KNI).

The eight 64-bit MMX operands, also used by 3DNow!, are called `%mm0`, `%mm1`, ... `%mm7`. They contain eight 8-bit integers, four 16-bit integers, two 32-bit integers, one 64-bit integer, or two 32-bit floating point values. The MMX registers cannot be used at the same time as the floating point stack.

See Intel and AMD documentation, keeping in mind that the operand order in instructions is reversed from the Intel syntax.

21.10. Writing 16-bit Code

While `as` normally writes only "pure" 32-bit i386 code or 64-bit x86-64 code depending on the default configuration, it also supports writing code to run in real mode or in 16-bit protected mode code segments. To do this, put a `.code16` or `.code16gcc` directive before the assembly language instructions to be run in 16-bit mode. You can switch `as` back to writing normal 32-bit code with the `.code32` directive.

`.code16gcc` provides experimental support for generating 16-bit code from gcc, and differs from `.code16` in that `call`, `ret`, `enter`, `leave`, `push`, `pop`, `pusha`, `popa`, `pushf`, and `popf` instructions default to 32-bit size. This is so that the stack pointer is manipulated in the same way over function calls, allowing access to function parameters at the same stack offsets as in 32-bit mode. `.code16gcc` also automatically adds address size prefixes where necessary to use the 32-bit addressing modes that gcc generates.

The code which `as` generates in 16-bit mode will not necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you must refrain from using *any* 32-bit constructs which require `as` to output address or operand size prefixes.

Note that writing 16-bit code instructions by explicitly specifying a prefix or an instruction mnemonic suffix within a 32-bit code section generates different machine instructions than those generated for a 16-bit code segment. In a 32-bit code section, the following code generates the machine opcode bytes `66 6a 04`, which pushes the value `4` onto the stack, decrementing `%esp` by 2.

```
pushw $4
```

The same code in a 16-bit code section would generate the machine opcode bytes `6a 04` (ie. without the operand size prefix), which is correct since the processor default operand size is assumed to be 16 bits in a 16-bit code section.

21.11. AT&T Syntax bugs

The UnixWare assembler, and probably other AT&T derived ix86 Unix assemblers, generate floating point instructions with reversed source and destination registers in certain cases. Unfortunately, gcc and possibly many other programs use this reversed syntax, so we're stuck with it.

For example

```
fsub %st,%st(3)
```

results in `%st(3)` being updated to `%st - %st(3)` rather than the expected `%st(3) - %st`. This happens with all the non-commutative arithmetic floating point operations with two register operands where the source register is `%st` and the destination register is `%st(i)`.

21.12. Specifying CPU Architecture

`as` may be told to assemble for a particular CPU architecture with the `.arch cpu_type` directive. This directive enables a warning when gas detects an instruction that is not supported on the CPU specified. The choices for `cpu_type` are:

i8086	i186	i286	i386
i486	i586	i686	pentium
pentiumpro	pentium4	k6	athlon
sledgehammer			

Apart from the warning, there are only two other effects on `as` operation; Firstly, if you specify a CPU other than `i486`, then shift by one instructions such as `sarl $1, %eax` will automatically use a two byte opcode sequence. The larger three byte opcode sequence is used on the 486 (and when no architecture is specified) because it executes faster on the 486. Note that you can explicitly request the two byte opcode by writing `sarl %eax`. Secondly, if you specify `i8086`, `i186`, or `i286`, *and* `.code16` or `.code16gcc` then byte offset conditional jumps will be promoted when necessary to a two instruction sequence consisting of a conditional jump of the opposite sense around an unconditional jump to the target.

Following the CPU architecture, you may specify `jumps` or `nojumps` to control automatic promotion of conditional jumps. `jumps` is the default, and enables jump promotion; All external jumps will be of the long variety, and file-local jumps will be promoted as necessary. (refer to Section 21.7 *Handling of Jump Instructions*) `nojumps` leaves external conditional jumps as byte offset jumps, and warns about file-local conditional jumps that `as` promotes. Unconditional jumps are treated as for `jumps`.

For example

```
.arch i8086,nojumps
```

21.13. Notes

There is some trickery concerning the `mul` and `imul` instructions that deserves mention. The 16-, 32-, 64- and 128-bit expanding multiplies (base opcode `0xf6`; extension 4 for `mul` and 5 for `imul`) can be output only in the one operand form. Thus, `imul %ebx, %eax` does *not* select the expanding multiply; the expanding multiply would clobber the `%edx` register, and this would confuse `gcc` output. Use `imul %ebx` to get the 64-bit product in `%edx:%eax`.

We have added a two operand form of `imul` when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying `%eax` by 69, for example, can be done with `imul $69, %eax` rather than `imul $69, %eax, %eax`.

Chapter 22.

Intel i860 Dependent Features

22.1. i860 Notes

This is a fairly complete i860 assembler which is compatible with the UNIX System V/860 Release 4 assembler. However, it does not currently support SVR4 PIC (i.e., `@GOT, @GOTOFF, @PLT`).

Like the SVR4/860 assembler, the output object format is ELF32. Currently, this is the only supported object format. If there is sufficient interest, other formats such as COFF may be implemented.

22.2. i860 Command-line Options

22.2.1. SVR4 compatibility options

`-V`

Print assembler version.

`-Qy`

Ignored.

`-Qn`

Ignored.

22.2.2. Other options

`-EL`

Select little endian output (this is the default).

`-EB`

Select big endian output. Note that the i860 always reads instructions as little endian data, so this option only effects data and not instructions.

`-mwarn-expand`

Emit a warning message if any pseudo-instruction expansions occurred. For example, a `or` instruction with an immediate larger than 16-bits will be expanded into two instructions. This is a very undesirable feature to rely on, so this flag can help detect any code where it happens. One use of it, for instance, has been to find and eliminate any place where `gcc` may emit these pseudo-instructions.

22.3. i860 Machine Directives

`.dual`

Enter dual instruction mode. While this directive is supported, the preferred way to use dual instruction mode is to explicitly code the dual bit with the `d.` prefix.

`.enddual`

Exit dual instruction mode. While this directive is supported, the preferred way to use dual instruction mode is to explicitly code the dual bit with the `d.` prefix.

`.atmp`

Change the temporary register used when expanding pseudo operations. The default register is `r31`.

22.4. i860 Opcodes

All of the Intel i860 machine instructions are supported. Please see either *i860 Microprocessor Programmer's Reference Manual* or *i860 Microprocessor Architecture* for more information.

22.4.1. Other instruction support (pseudo-instructions)

For compatibility with some other i860 assemblers, a number of pseudo-instructions are supported. While these are supported, they are a very undesirable feature that should be avoided - in particular, when they result in an expansion to multiple actual i860 instructions. Below are the pseudo-instructions that result in expansions.

- Load large immediate into general register:

  The pseudo-instruction `mov imm,%rn` (where the immediate does not fit within a signed 16-bit field) will be expanded into:
  ```
  orh large_imm@h,%r0,%rn
  or large_imm@l,%rn,%rn
  ```
- Load/store with relocatable address expression:

  For example, the pseudo-instruction `ld.b addr,%rn` will be expanded into:
  ```
  orh addr_exp@ha,%r0,%r31
  ld.l addr_exp@l(%r31),%rn
  ```
  The analogous expansions apply to `ld.x, st.x, fld.x, pfld.x, fst.x`, and `pst.x` as well.
- Signed large immediate with add/subtract:

  If any of the arithmetic operations `adds, addu, subs, subu` are used with an immediate larger than 16-bits (signed), then they will be expanded. For instance, the pseudo-instruction `adds large_imm,%rx,%rn` expands to:
  ```
  orh large_imm@h,%r0,%r31
  or large_imm@l,%r31,%r31
  adds %r31,%rx,%rn
  ```
- Unsigned large immediate with logical operations:

  Logical operations (`or, andnot, or, xor`) also result in expansions. The pseudo-instruction `or large_imm,%rx,%rn` results in:
  ```
  orh large_imm@h,%rx,%r31
  or large_imm@l,%r31,%rn
  ```
  Similarly for the others, except for `and` which expands to:

```
andnot (-1 - large_imm)@h,%rx,%r31
andnot (-1 - large_imm)@l,%r31,%rn
```

Chapter 23. Intel 80960 Dependent Features

23.1. i960 Command-line Options

`-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC`

Select the 80960 architecture. Instructions or features not supported by the selected architecture cause fatal errors.

`-ACA` is equivalent to `-ACA_A`; `-AKC` is equivalent to `-AMC`. Synonyms are provided for compatibility with other tools.

If you do not specify any of these options, `as` generates code for any instruction or feature that is supported by *some* version of the 960 (even if this means mixing architectures!). In principle, `as` attempts to deduce the minimal sufficient processor type if none is specified; depending on the object code format, the processor type may be recorded in the object file. If it is critical that the `as` output match a specific architecture, specify that architecture explicitly.

`-b`

Add code to collect information about conditional branches taken, for later optimization using branch prediction bits. (The conditional branch instructions have branch prediction bits in the CA, CB, and CC architectures.) If `BR` represents a conditional branch instruction, the following represents the code generated by the assembler when `-b` is specified:

```
        call    increment routine
        .word   0       # pre-counter
Label:  BR
        call    increment routine
        .word   0       # post-counter
```

The counter following a branch records the number of times that branch was *not* taken; the differenc between the two counters is the number of times the branch *was* taken.

A table of every such `Label` is also generated, so that the external postprocessor `gbr960` (supplied by Intel) can locate all the counters. This table is always labeled `__BRANCH_TABLE__`; this is a local symbol to permit collecting statistics for many separate object files. The table is word aligned, and begins with a two-word header. The first word, initialized to 0, is used in maintaining linked lists of branch tables. The second word is a count of the number of entries in the table, which follow immediately: each is a word, pointing to one of the labels illustrated above.

The first word of the header is used to locate multiple branch tables, since each object file may contain one. Normally the links are maintained with a call to an initialization routine, placed at the beginning of each function in the file. The gnu C compiler generates these calls automatically when you give it a `-b` option. For further details, see the documentation of `gbr960`.

`-no-relax`

Normally, Compare-and-Branch instructions with targets that require displacements greater than 13 bits (or that have external targets) are replaced with the corresponding compare (or `chkbit`) and branch instructions. You can use the `-no-relax` option to specify that `as` should generate errors instead, if the target displacement is larger than 13 bits.

This option does not affect the Compare-and-Jump instructions; the code emitted for them is *always* adjusted when necessary (depending on displacement size), regardless of whether you use `-no-relax`.

23.2. Floating Point

`as` generates ieee floating-point numbers for the directives `.float`, `.double`, `.extended`, and `.single`.

23.3. i960 Machine Directives

`.bss symbol, length, align`

Reserve `length` bytes in the bss section for a local `symbol`, aligned to the power of two specified by `align`. `length` and `align` must be positive absolute expressions. This directive differs from `.lcomm` only in that it permits you to specify an alignment. Section 8.50 *.lcomm symbol, length*.

`.extended flonums`

`.extended` expects zero or more flonums, separated by commas; for each flonum, `.extended` emits an ieee extended-format (80-bit) floating-point number.

`.leafproc call-lab, bal-lab`

You can use the `.leafproc` directive in conjunction with the optimized `callj` instruction to enable faster calls of leaf procedures. If a procedure is known to call no other procedures, you may define an entry point that skips procedure prolog code (and that does not depend on system-supplied saved context), and declare it as the `bal-lab` using `.leafproc`. If the procedure also has an entry point that goes through the normal prolog, you can specify that entry point as `call-lab`.

A `.leafproc` declaration is meant for use in conjunction with the optimized call instruction `callj`; the directive records the data needed later to choose between converting the `callj` into a `bal` or a `call`.

`call-lab` is optional; if only one argument is present, or if the two arguments are identical, the single argument is assumed to be the `bal` entry point.

`.sysproc name, index`

The `.sysproc` directive defines a name for a system procedure. After you define it using `.sysproc`, you can use `name` to refer to the system procedure identified by `index` when calling procedures with the optimized call instruction `callj`.

Both arguments are required; `index` must be between 0 and 31 (inclusive).

23.4. i960 Opcodes

All Intel 960 machine instructions are supported; Section 23.1 *i960 Command-line Options* for a discussion of selecting the instruction subset for a particular 960 architecture.

Some opcodes are processed beyond simply emitting a single corresponding instruction: `callj`, and Compare-and-Branch or Compare-and-Jump instructions with target displacements larger than 13 bits.

23.4.1. `callj`

You can write `callj` to have the assembler or the linker determine the most appropriate form of subroutine call: `call`, `bal`, or `calls`. If the assembly source contains enough information--a `.leafproc` or `.sysproc` directive defining the operand--then `as` translates the `callj`; if not, it simply emits the `callj`, leaving it for the linker to resolve.

23.4.2. Compare-and-Branch

The 960 architectures provide combined Compare-and-Branch instructions that permit you to store the branch target in the lower 13 bits of the instruction word itself. However, if you specify a branch target far enough away that its address won't fit in 13 bits, the assembler can either issue an error, or convert your Compare-and-Branch instruction into separate instructions to do the compare and the branch.

Whether `as` gives an error or expands the instruction depends on two choices you can make: whether you use the `-no-relax` option, and whether you use a "Compare and Branch" instruction or a "Compare and Jump" instruction. The "Jump" instructions are *always* expanded if necessary; the "Branch" instructions are expanded when necessary *unless* you specify `-no-relax`--in which case `as` gives an error instead.

These are the Compare-and-Branch instructions, their "Jump" variants, and the instruction pairs they may expand into:

Chapter 24. IP2K Dependent Features

24.1. IP2K Options

The Ubicom IP2K version of `as` has a few machine dependent options:

`-mip2022ext`

: `as` can assemble the extended IP2022 instructions, but it will only do so if this is specifically allowed via this command line option.

`-mip2022`

: This option restores the assembler's default behaviour of not permitting the extended IP2022 instructions to be assembled.

Chapter 25. M32R Dependent Features

25.1. M32R Options

The Renease M32R version of `as` has a few machine dependent options:

`-m32rx`

: `as` can assemble code for several different members of the Renesas M32R family. Normally the default is to assemble code for the M32R microprocessor. This option may be used to change the default to the M32RX microprocessor, which adds some more instructions to the basic M32R instruction set, and some additional parameters to some of the original instructions.

`-m32r`

: This option can be used to restore the assembler's default behaviour of assembling for the M32R microprocessor. This can be useful if the default has been changed by a previous command line option.

`-warn-explicit-parallel-conflicts`

: Instructs `as` to produce warning messages when questionable parallel instructions are encountered. This option is enabled by default, but `gcc` disables it when it invokes `as` directly. Questionable instructions are those whoes behaviour would be different if they were executed sequentially. For example the code fragment `mv r1, r2 || mv r3, r1` produces a different result from `mv r1, r2 \n mv r3, r1` since the former moves r1 into r3 and then r2 into r1, whereas the later moves r2 into r1 and r3.

`-Wp`

: This is a shorter synonym for the *-warn-explicit-parallel-conflicts* option.

`-no-warn-explicit-parallel-conflicts`

: Instructs `as` not to produce warning messages when questionable parallel instructions are encountered.

`-Wnp`

: This is a shorter synonym for the *-no-warn-explicit-parallel-conflicts* option.

25.2. M32R Warnings

There are several warning and error messages that can be produced by `as` which are specific to the M32R:

`output of 1st instruction is the same as an input to 2nd instruction - is this intentional ?`

: This message is only produced if warnings for explicit parallel conflicts have been enabled. It indicates that the assembler has encountered a parallel instruction in which the destination register of the left hand instruction is used as an input register in the right hand instruction. For example in this code fragment `mv r1, r2 || neg r3, r1` register r1 is the destination of the move instruction and the input to the neg instruction.

`output of 2nd instruction is the same as an input to 1st instruction - is this intentional ?`

This message is only produced if warnings for explicit parallel conflicts have been enabled. It indicates that the assembler has encountered a parallel instruction in which the destination register of the right hand instruction is used as an input register in the left hand instruction. For example in this code fragment `mv r1, r2 || neg r2, r3` register r2 is the destination of the neg instruction and the input to the move instruction.

`instruction ... is for the M32RX only`

This message is produced when the assembler encounters an instruction which is only supported by the M32Rx processor, and the `-m32rx` command line flag has not been specified to allow assembly of such instructions.

`unknown instruction ...`

This message is produced when the assembler encounters an instruction which it doe snot recognise.

`only the NOP instruction can be issued in parallel on the m32r`

This message is produced when the assembler encounters a parallel instruction which does not involve a NOP instruction and the `-m32rx` command line flag has not been specified. Only the M32Rx processor is able to execute two instructions in parallel.

`instruction ... cannot be executed in parallel.`

This message is produced when the assembler encounters a parallel instruction which is made up of one or two instructions which cannot be executed in parallel.

`Instructions share the same execution pipeline`

This message is produced when the assembler encounters a parallel instruction whoes components both use the same execution pipeline.

`Instructions write to the same destination register.`

This message is produced when the assembler encounters a parallel instruction where both components attempt to modify the same register. For example these code fragments will produce this message: `mv r1, r2 || neg r1, r3` `jl r0 || mv r14, r1` `st r2, @-r1 || mv r1, r3` `mv r1, r2 || ld r0, @r1+` `cmp r1, r2 || addx r3, r4` (Both write to the condition bit)

Chapter 26. M680x0 Dependent Features

26.1. M680x0 Options

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

`-l`

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.

`-register-prefix-optional`

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.

`-bitwise-or`

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.

`-base-size-default-16 -base-size-default-32`

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.

`-disp-size-default-16 -disp-size-default-32`

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.

`-pcrel`

Always keep branches PC-relative. In the M680x0 architecture all branches are defined as PC-relative. However, on some processors they are limited to word displacements maximum. When `as` needs a long branch that is not available, it normally emits an absolute jump instead. This option disables this substitution. When this option is given and no long branches are available, only word branches will be emitted. An error message will be generated if a word branch cannot reach its target. This option has no effect on 68020 and other processors that have long branches. Section 26.6.1 *Branch Improvement*.

`-m68000`

`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.

26.2. 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`.

26.3. Motorola Syntax

The standard Motorola syntax for this chip differs from the syntax already discussed (refer to Section 26.2 *Syntax*). `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.

26.4. 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.

26.5. 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.

26.6. Opcodes

26.6.1. 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, not PC-relative OK
Pseudo-Op  |BYTE    WORD    LONG            ABSOLUTE LONG JUMP    **
           +-------------------------------------------------------------
     jbsr  |bsrs    bsrw    bsrl            jsr
      jra  |bras    braw    bral            jmp
*     jXX  |bXXs    bXXw    bXXl            bNXs;jmp
*    dbXX  | N/A    dbXXw   dbXX;bras;bral  dbXX;bras;jmp
     fjXX  | N/A    fbXXw   fbXXl            N/A

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

`*`--see full description below`**`--this expansion mode is disallowed by `-pcrel`

`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. This instruction will be a byte or word branch is that is sufficient. Otherwise, a long branch will be emitted if available. If no long branches are available and the `-pcrel` option is not given, an absolute long jump will be emitted instead. If no long branches are available, the `-pcrel` option is given, and a word branch cannot reach the target, an error message is generated.

In addition to standard branch operands, `as` allows these pseudo-operations to have all operands that are allowed for jsr and jmp, substituting these instructions if the operand given is not valid for a branch instruction.

`jXX`

Here, `jXX` 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
```

Usually, each of these pseudo-operations expands to a single branch instruction. However, if a word branch is not sufficient, no long branches are available, and the `-pcrel` option is not given, `as` issues a longer code fragment in terms of `NX`, the opposite condition to `XX`. For example, under these conditions:

```
    jXX foo
```

gives

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

`dbXX`

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
```

Motorola `dbXX` instructions allow word displacements only. When a word displacement is sufficient, each of these pseudo-operations expands to the corresponding Motorola instruction. When a word displacement is not sufficient and long branches are available, when the source reads `dbXX foo`, `as` emits

```
     dbXX oo1
     bras oo2
oo1:bral foo
oo2:
```

If, however, long branches are not available and the `-pcrel` option is not given, `as` emits

```
     dbXX oo1
     bras oo2
oo1:jmp foo
oo2:
```

`fjXX`

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
```

Each of these pseudo-operations always expands to a single Motorola coprocessor branch instruction, word or long. All Motorola coprocessor branch instructions allow both word and long displacements.

26.6.2. 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.

Chapter 27.
M68HC11 and M68HC12 Dependent Features

27.1. M68HC11 and M68HC12 Options

The Motorola 68HC11 and 68HC12 version of `as` have a few machine dependent options.

`-m68hc11`

This option switches the assembler in the M68HC11 mode. In this mode, the assembler only accepts 68HC11 operands and mnemonics. It produces code for the 68HC11.

`-m68hc12`

This option switches the assembler in the M68HC12 mode. In this mode, the assembler also accepts 68HC12 operands and mnemonics. It produces code for the 68HC12. A few 68HC11 instructions are replaced by some 68HC12 instructions as recommended by Motorola specifications.

`-m68hcs12`

This option switches the assembler in the M68HCS12 mode. This mode is similar to `-m68hc12` but specifies to assemble for the 68HCS12 series. The only difference is on the assembling of the `movb` and `movw` instruction when a PC-relative operand is used.

`-mshort`

This option controls the ABI and indicates to use a 16-bit integer ABI. It has no effect on the assembled instructions. This is the default.

`-mlong`

This option controls the ABI and indicates to use a 32-bit integer ABI.

`-mshort-double`

This option controls the ABI and indicates to use a 32-bit float ABI. This is the default.

`-mlong-double`

This option controls the ABI and indicates to use a 64-bit float ABI.

`-strict-direct-mode`

You can use the `-strict-direct-mode` option to disable the automatic translation of direct page mode addressing into extended mode when the instruction does not support direct mode. For example, the `clr` instruction does not support direct page mode addressing. When it is used with the direct page mode, `as` will ignore it and generate an absolute addressing. This option prevents `as` from doing this, and the wrong usage of the direct page mode will raise an error.

`-short-branchs`

The `-short-branchs` option turns off the translation of relative branches into absolute branches when the branch offset is out of range. By default `as` transforms the relative branch (`bsr`, `bgt`, `bge`, `beq`, `bne`, `ble`, `blt`, `bhi`, `bcc`, `bls`, `bcs`, `bmi`, `bvs`, `bvs`, `bra`) into an absolute branch when the offset is out of the -128 .. 127 range. In that case, the `bsr` instruction is translated into a `jsr`, the `bra` instruction is translated into a `jmp` and the conditional branchs instructions are inverted and followed by a `jmp`. This option disables these translations and `as` will generate an error if a relative branch is out of range. This option does not affect the optimization associated to the `jbra`, `jbsr` and `jbXX` pseudo opcodes.

`-force-long-branchs`

The `-force-long-branchs` option forces the translation of relative branches into absolute branches. This option does not affect the optimization associated to the `jbra`, `jbsr` and `jbXX` pseudo opcodes.

`-print-insn-syntax`

You can use the `-print-insn-syntax` option to obtain the syntax description of the instruction when an error is detected.

`-print-opcodes`

The `-print-opcodes` option prints the list of all the instructions with their syntax. The first item of each line represents the instruction name and the rest of the line indicates the possible operands for that instruction. The list is printed in alphabetical order. Once the list is printed `as` exits.

`-generate-example`

The `-generate-example` option is similar to `-print-opcodes` but it generates an example for each instruction instead.

27.2. Syntax

In the M68HC11 syntax, the instruction name comes first and it may be followed by one or several operands (up to three). Operands are separated by comma (`,`). In the normal mode, `as` will complain if too many operands are specified for a given instruction. In the MRI mode (turned on with `-M` option), it will treat them as comments. Example:

```
inx
lda  #23
bset 2,x #4
brclr *bot #8 foo
```

The following addressing modes are understood for 68HC11 and 68HC12:

Immediate

`#number`

Address Register

`number,X`, `number,Y`

The `number` may be omitted in which case 0 is assumed.

Direct Addressing mode

`*symbol`, or `*digits`

Absolute

`symbol`, or `digits`

The M68HC12 has other more complex addressing modes. All of them are supported and they are represented below:

Constant Offset Indexed Addressing Mode

`number,reg`

The `number` may be omitted in which case 0 is assumed. The register can be either `X`, `Y`, `SP` or `PC`. The assembler will use the smaller post-byte definition according to the constant value (5-bit constant offset, 9-bit constant offset or 16-bit constant offset). If the constant is not known by the assembler it will use the 16-bit constant offset post-byte and the value will be resolved at link time.

Offset Indexed Indirect

`[number,reg]`

The register can be either `X`, `Y`, `SP` or `PC`.

Auto Pre-Increment/Pre-Decrement/Post-Increment/Post-Decrement

`number,-reg` `number,+reg` `number,reg-` `number,reg+`

The number must be in the range `-8`..`+8` and must not be 0. The register can be either `X`, `Y`, `SP` or `PC`.

Accumulator Offset

`acc,reg`

The accumulator register can be either `A`, `B` or `D`. The register can be either `X`, `Y`, `SP` or `PC`.

Accumulator D offset indexed-indirect

`[D,reg]`

The register can be either `X`, `Y`, `SP` or `PC`.

For example:

```
ldab 1024,sp
ldd [10,x]
orab 3,+x
stab -2,y-
ldx a,pc
sty [d,sp]
```

27.3. Symbolic Operand Modifiers

The assembler supports several modifiers when using symbol addresses in 68HC11 and 68HC12 instruction operands. The general syntax is the following:

```
%modifier(symbol)
```

`%addr`

This modifier indicates to the assembler and linker to use the 16-bit physical address corresponding to the symbol. This is intended to be used on memory window systems to map a symbol in the memory bank window. If the symbol is in a memory expansion part, the physical address corresponds to the symbol address within the memory bank window. If the symbol is not in a memory expansion part, this is the symbol address (using or not using the %addr modifier has no effect in that case).

`%page`

This modifier indicates to use the memory page number corresponding to the symbol. If the symbol is in a memory expansion part, its page number is computed by the linker as a number used to map the page containing the symbol in the memory bank window. If the symbol is not in a memory expansion part, the page number is 0.

`%hi`

This modifier indicates to use the 8-bit high part of the physical address of the symbol.

`%lo`

This modifier indicates to use the 8-bit low part of the physical address of the symbol.

For example a 68HC12 call to a function `foo_example` stored in memory expansion part could be written as follows:

```
call %addr(foo_example),%page(foo_example)
```

and this is equivalent to

```
call foo_example
```

And for 68HC11 it could be written as follows:

```
ldab #%page(foo_example)
stab _page_switch
jsr  %addr(foo_example)
```

27.4. Assembler Directives

The 68HC11 and 68HC12 version of `as` have the following specific assembler directives:

`.relax`

The relax directive is used by the `GNU Compiler` to emit a specific relocation to mark a group of instructions for linker relaxation. The sequence of instructions within the group must be known to the linker so that relaxation can be performed.

`.mode [mshort|mlong|mshort-double|mlong-double]`

This directive specifies the ABI. It overrides the `-mshort`, `-mlong`, `-mshort-double` and `-mlong-double` options.

`.far symbol`

This directive marks the symbol as a `far` symbol meaning that it uses a `call/rtc` calling convention as opposed to `jsr/rts`. During a final link, the linker will identify references to the `far` symbol and will verify the proper calling convention.

`.interrupt symbol`

This directive marks the symbol as an interrupt entry point. This information is then used by the debugger to correctly unwind the frame across interrupts.

`.xrefb symbol`

This directive is defined for compatibility with the `Specification for Motorola 8 and 16-Bit Assembly Language Input Standard` and is ignored.

27.5. 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.

27.6. Opcodes

27.6.1. 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 prepending `j` to the start of Motorola mnemonic. These pseudo opcodes are not affected by the `-short-branchs` or `-force-long-branchs` options.

The following table summarizes the pseudo-operations.

```
                        Displacement Width
     +-------------------------------------------------------------+
     |                          Options                            |
     |    --short-branchs            --force-long-branchs          |
     +--------------------------+----------------------------------+
  Op |BYTE             WORD     | BYTE          WORD               |
     +--------------------------+----------------------------------+
 bsr | bsr <pc-rel>    <error>  |               jsr <abs>          |
 bra | bra <pc-rel>    <error>  |               jmp <abs>          |
jbsr | bsr <pc-rel>   jsr <abs> | bsr <pc-rel>  jsr <abs>          |
jbra | bra <pc-rel>   jmp <abs> | bra <pc-rel>  jmp <abs>          |
 bXX | bXX <pc-rel>    <error>  |               bNX +3; jmp <abs>  |
jbXX | bXX <pc-rel>   bNX +3;   | bXX <pc-rel>  bNX +3; jmp <abs>  |
     |                jmp <abs> |                                  |
     +--------------------------+----------------------------------+
XX: condition
```

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

`jbsr`
`jbra`

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

`jbXX`

Here, `jbXX` 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:

```
jbcc   jbeq   jbge   jbgt   jbhi   jbvs   jbpl  jblo
jbcs   jbne   jblt   jble   jbls   jbvc   jbmi
```

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

```
    jbXX  foo
```

gives

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

Chapter 28. Motorola M88K Dependent Features

28.1. M88K Machine Directives

The M88K version of the assembler supports the following machine directives:

`.align`

This directive aligns the section program counter on the next 4-byte boundary.

`.dfloat expr`

This assembles a double precision (64-bit) floating point constant.

`.ffloat expr`

This assembles a single precision (32-bit) floating point constant.

`.half expr`

This directive assembles a half-word (16-bit) constant.

`.word expr`

This assembles a word (32-bit) constant.

`.string "str"`

This directive behaves like the standard `.ascii` directive for copying `str` into the object file. The string is not terminated with a null byte.

`.set symbol, value`

This directive creates a symbol named `symbol` which is an alias for another symbol (possibly not yet defined). This should not be confused with the mnemonic `set`, which is a legitimate M88K instruction.

`.def symbol, value`

This directive is synonymous with `.set` and is presumably provided for compatibility with other M88K assemblers.

`.bss symbol, length, align`

Reserve `length` bytes in the bss section for a local `symbol`, aligned to the power of two specified by `align`. `length` and `align` must be positive absolute expressions. This directive differs from `.lcomm` only in that it permits you to specify an alignment. Section 8.50 *.lcomm symbol, length*.

Chapter 29. MIPS Dependent Features

gnu `as` for mips architectures supports several different mips processors, and MIPS ISA levels I through V, MIPS32, and MIPS64. For information about the mips instruction set, see [MIPS RISC Architecture], by Kane and Heindrich (Prentice-Hall). For an overview of mips assembly conventions, see "Appendix D: Assembly Language Programming" in the same work.

29.1. Assembler options

The mips configurations of gnu `as` support these special options:

`-G num`

This option sets the largest size of an object that can be referenced implicitly with the `gp` register. It is only accepted for targets that use ecoff format. The default value is 8.

`-EB`
`-EL`

Any mips configuration of `as` can select big-endian or little-endian output at run time (unlike the other gnu development tools, which must be configured for one or the other). Use `-EB` to select big-endian output, and `-EL` for little-endian.

`-mips1`
`-mips2`
`-mips3`
`-mips4`
`-mips5`
`-mips32`
`-mips32r2`
`-mips64`

Generate code for a particular MIPS Instruction Set Architecture level. `-mips1` corresponds to the r2000 and r3000 processors, `-mips2` to the r6000 processor, `-mips3` to the r4000 processor, and `-mips4` to the r8000 and r10000 processors. `-mips5`, `-mips32`, `-mips32r2`, and `-mips64` correspond to generic MIPS V, MIPS32, MIPS32 Release 2, and MIPS64 ISA processors, respectively. You can also switch instruction sets during the assembly; see Directives to override the ISA level.

`-mgp32`
`-mfp32`

Some macros have different expansions for 32-bit and 64-bit registers. The register sizes are normally inferred from the ISA and ABI, but these flags force a certain group of registers to be treated as 32 bits wide at all times. `-mgp32` controls the size of general-purpose registers and `-mfp32` controls the size of floating-point registers.

On some MIPS variants there is a 32-bit mode flag; when this flag is set, 64-bit instructions generate a trap. Also, some 32-bit OSes only save the 32-bit registers on a context switch, so it is essential never to use the 64-bit registers.

`-mgp64`

Assume that 64-bit general purpose registers are available. This is provided in the interests of symmetry with -gp32.

`-mips16`
`-no-mips16`

Generate code for the MIPS 16 processor. This is equivalent to putting `.set mips16` at the start of the assembly file. `-no-mips16` turns off this option.

`-mips3d`
`-no-mips3d`

Generate code for the MIPS-3D Application Specific Extension. This tells the assembler to accept MIPS-3D instructions. `-no-mips3d` turns off this option.

`-mdmx`
`-no-mdmx`

Generate code for the MDMX Application Specific Extension. This tells the assembler to accept MDMX instructions. `-no-mdmx` turns off this option.

`-mfix7000`
`-mno-fix7000`

Cause nops to be inserted if the read of the destination register of an mfhi or mflo instruction occurs in the following two instructions.

`-mfix-vr4122-bugs`
`-no-mfix-vr4122-bugs`

Insert `nop` instructions to avoid errors in certain versions of the vr4122 core. This option is intended to be used on GCC-generated code: it is not designed to catch errors in hand-written assembler code.

`-m4010`
`-no-m4010`

Generate code for the LSI r4010 chip. This tells the assembler to accept the r4010 specific instructions (`addciu`, `ffc`, etc.), and to not schedule `nop` instructions around accesses to the `HI` and `LO` registers. `-no-m4010` turns off this option.

`-m4650`
`-no-m4650`

Generate code for the MIPS r4650 chip. This tells the assembler to accept the `mad` and `madu` instruction, and to not schedule `nop` instructions around accesses to the `HI` and `LO` registers. `-no-m4650` turns off this option.

`-m3900`
`-no-m3900`
`-m4100`
`-no-m4100`

For each option `-mnnnn`, generate code for the MIPS r`nnnn` chip. This tells the assembler to accept instructions specific to that chip, and to schedule for that chip's hazards.

`-march=cpu`

Generate code for a particular MIPS cpu. It is exactly equivalent to `-mcpu`, except that there are more value of `cpu` understood. Valid `cpu` value are:

> 2000, 3000, 3900, 4000, 4010, 4100, 4111, vr4120, vr4130, vr4181, 4300, 4400, 4600, 4650, 5000, rm5200, rm5230, rm5231, rm5261, rm5721, vr5400, vr5500, 6000, rm7000, 8000, 10000, 12000, mips32-4k, sb1

`-mtune=cpu`

Schedule and tune for a particular MIPS cpu. Valid `cpu` values are identical to `-march=cpu`.

`-mabi=abi`

Record which ABI the source code uses. The recognized arguments are: `32`, `n32`, `o64`, `64` and `eabi`.

`-nocpp`

This option is ignored. It is accepted for command-line compatibility with other assemblers, which use it to turn off C style preprocessing. With gnu `as`, there is no need for `-nocpp`, because the gnu assembler itself never runs the C preprocessor.

`-construct-floats`
`-no-construct-floats`

The `-no-construct-floats` option disables the construction of double width floating point constants by loading the two halves of the value into the two single width floating point registers that make up the double width register. This feature is useful if the processor support the FR bit in its status register, and this bit is known (by the programmer) to be set. This bit prevents the aliasing of the double width register by the single width registers.

By default `-construct-floats` is selected, allowing construction of these floating point constants.

`-trap`
`-no-break`

`as` automatically macro expands certain division and multiplication instructions to check for overflow and division by zero. This option causes `as` to generate code to take a trap exception rather than a break exception when an error is detected. The trap instructions are only supported at Instruction Set Architecture level 2 and higher.

`-break`
`-no-trap`

Generate code to take a break exception rather than a trap exception when an error is detected. This is the default.

`-n`

When this option is used, `as` will issue a warning every time it generates a nop instruction from a macro.

29.2. MIPS ECOFF object code

Assembling for a mips ecoff target supports some additional sections besides the usual `.text`, `.data` and `.bss`. The additional sections are `.rdata`, used for read-only data, `.sdata`, used for small data, and `.sbss`, used for small common objects.

When assembling for ecoff, the assembler uses the `$gp` (`$28`) register to form the address of a "small object". Any object in the `.sdata` or `.sbss` sections is considered "small" in this sense. For external objects, or for objects in the `.bss` section, you can use the `gcc -G` option to control the size of objects addressed via `$gp`; the default value is 8, meaning that a reference to any object eight bytes or smaller uses `$gp`. Passing `-G 0` to `as` prevents it from using the `$gp` register on the basis of object size (but the assembler uses `$gp` for objects in `.sdata` or `sbss` in any case). The size of an object in the `.bss` section is set by the `.comm` or `.lcomm` directive that defines it. The size of an external object may be set with the `.extern` directive. For example, `.extern sym,4` declares that the object at `sym` is 4 bytes in length, whie leaving `sym` otherwise undefined.

Using small ecoff objects requires linker support, and assumes that the `$gp` register is correctly initialized (normally done automatically by the startup code). mips ecoff assembly code must not modify the `$gp` register.

29.3. Directives for debugging information

mips ecoff `as` supports several directives used for generating debugging information which are not support by traditional mips assemblers. These are `.def`, `.endef`, `.dim`, `.file`, `.scl`, `.size`, `.tag`, `.type`, `.val`, `.stabd`, `.stabn`, and `.stabs`. The debugging information generated by the three `.stab` directives can only be read by gdb, not by traditional mips debuggers (this enhancement is required to fully support C++ debugging). These directives are primarily used by compilers, not assembly language programmers!

29.4. Directives to override the ISA level

gnu `as` supports an additional directive to change the mips Instruction Set Architecture level on the fly: `.set mipsn`. `n` should be a number from 0 to 5, or 32, 32r2, or 64. The values other than 0 make the assembler accept instructions for the corresponding isa level, from that point on in the assembly. `.set mipsn` affects not only which instructions are permitted, but also how certain macros are expanded. `.set mips0` restores the isa level to its original level: either the level you selected with command line options, or the default for your configuration. You can use this feature to permit specific r4000 instructions while assembling in 32 bit mode. Use this directive with care!

The directive `.set mips16` puts the assembler into MIPS 16 mode, in which it will assemble instructions for the MIPS 16 processor. Use `.set nomips16` to return to normal 32 bit mode.

Traditional mips assemblers do not support this directive.

29.5. Directives for extending MIPS 16 bit instructions

By default, MIPS 16 instructions are automatically extended to 32 bits when necessary. The directive `.set noautoextend` will turn this off. When `.set noautoextend` is in effect, any 32 bit instruction must be explicitly extended with the `.e` modifier (e.g., `li.e $4,1000`). The directive `.set autoextend` may be used to once again automatically extend instructions when necessary.

This directive is only meaningful when in MIPS 16 mode. Traditional mips assemblers do not support this directive.

29.6. Directive to mark data as an instruction

The `.insn` directive tells `as` that the following data is actually instructions. This makes a difference in MIPS 16 mode: when loading the address of a label which precedes instructions, `as` automatically adds 1 to the value, so that jumping to the loaded address will do the right thing.

29.7. Directives to save and restore options

The directives `.set push` and `.set pop` may be used to save and restore the current settings for all the options which are controlled by `.set`. The `.set push` directive saves the current settings on a stack. The `.set pop` directive pops the stack and restores the settings.

These directives can be useful inside an macro which must change an option such as the ISA level or instruction reordering but does not want to change the state of the code which invoked the macro.

Traditional mips assemblers do not support these directives.

29.8. Directives to control generation of MIPS ASE instructions

The directive `.set mips3d` makes the assembler accept instructions from the MIPS-3D Application Specific Extension from that point on in the assembly. The `.set nomips3d` directive prevents MIPS-3D instructions from being accepted.

The directive `.set mdmx` makes the assembler accept instructions from the MDMX Application Specific Extension from that point on in the assembly. The `.set nomdmx` directive prevents MDMX instructions from being accepted.

Traditional mips assemblers do not support these directives.

Chapter 30. MMIX Dependent Features

30.1. Command-line Options

The MMIX version of `as` has some machine-dependent options.

When `-fixed-special-register-names` is specified, only the register names specified in Section 30.3.3 *Register names* are recognized in the instructions `PUT` and `GET`.

You can use the `-globalize-symbols` to make all symbols global. This option is useful when splitting up a `mmixal` program into several files.

The `-gnu-syntax` turns off most syntax compatibility with `mmixal`. Its usability is currently doubtful.

The `-relax` option is not fully supported, but will eventually make the object file prepared for linker relaxation.

If you want to avoid inadvertently calling a predefined symbol and would rather get an error, for example when using `as` with a compiler or other machine-generated code, specify `-no-predefined-syms`. This turns off built-in predefined definitions of all such symbols, including rounding-mode symbols, segment symbols, `BIT` symbols, and `TRAP` symbols used in `mmix` "system calls". It also turns off predefined special-register names, except when used in `PUT` and `GET` instructions.

By default, some instructions are expanded to fit the size of the operand or an external symbol (refer to Section 30.2 *Instruction expansion*). By passing `-no-expand`, no such expansion will be done, instead causing errors at link time if the operand does not fit.

The `mmixal` documentation specifies that global registers allocated with the `GREG` directive and initialized to the same non-zero value, will refer to the same global register. This isn't strictly enforceable in `as` since the final addresses aren't known until link-time, but it will do an effort unless the `-no-merge-gregs` option is specified. (Register merging isn't yet implemented in `ld`.)

`as` will warn every time it expands an instruction to fit an operand unless the option `-x` is specified. It is believed that this behaviour is more useful than just mimicking `mmixal`'s behaviour, in which instructions are only expanded if the `-x` option is specified, and assembly fails otherwise, when an instruction needs to be expanded. It needs to be kept in mind that `mmixal` is both an assembler and linker, while `as` will expand instructions that at link stage can be contracted. (Though linker relaxation isn't yet implemented in `ld`.) The option `-x` also imples `-linker-allocated-gregs`.

Usually a two-operand-expression without a matching `GREG` directive is treated as an error by `as`. When the option `-linker-allocated-gregs` is in effect, they are instead passed through to the linker, which will allocate as many global registers as is needed.

30.2. Instruction expansion

When `as` encounters an instruction with an operand that is either not known or does not fit the operand size of the instruction, `as` (and `ld`) will expand the instruction into a sequence of instructions semantically equivalent to the operand fitting the instruction. Expansion will take place for the following instructions:

`GETA`

Expands to a sequence of four instructions: `SETL`, `INCML`, `INCMH` and `INCH`. The operand must be a multiple of four.

Conditional branches

A branch instruction is turned into a branch with the complemented condition and prediction bit over five instructions; four instructions setting `$255` to the operand value, which like with `GETA` must be a multiple of four, and a final `GO $255,$255,0`.

`PUSHJ`

Similar to expansion for conditional branches; four instructions set `$255` to the operand value, followed by a `PUSHGO $255,$255,0`.

`JMP`

Similar to conditional branches and `PUSHJ`. The final instruction is `GO $255,$255,0`.

The linker `ld` is expected to shrink these expansions for code assembled with `-relax` (though not currently implemented).

30.3. Syntax

The assembly syntax is supposed to be upward compatible with that described in Sections 1.3 and 1.4 of `The Art of Computer Programming, Volume 1`. Draft versions of those chapters as well as other MMIX information is located at *http://www-cs-faculty.stanford.edu/~knuth/mmix-news.html*. Most code examples from the mmixal package located there should work unmodified when assembled and linked as single files, with a few noteworthy exceptions (refer to Section 30.4 *Differences to* `mmixal`).

Before an instruction is emitted, the current location is aligned to the next four-byte boundary. If a label is defined at the beginning of the line, its value will be the aligned value.

In addition to the traditional hex-prefix `0x`, a hexadecimal number can also be specified by the prefix character `#`.

After all operands to an MMIX instruction or directive have been specified, the rest of the line is ignored, treated as a comment.

30.3.1. Special Characters

The characters `*` and `#` are line comment characters; each start a comment at the beginning of a line, but only at the beginning of a line. A `#` prefixes a hexadecimal number if found elsewhere on a line.

Two other characters, `%` and `!`, each start a comment anywhere on the line. Thus you can't use the `modulus` and `not` operators in expressions normally associated with these two characters.

A `;` is a line separator, treated as a new-line, so separate instructions can be specified on a single line.

30.3.2. Symbols

The character `:` is permitted in identifiers. There are two exceptions to it being treated as any other symbol character: if a symbol begins with `:`, it means that the symbol is in the global namespace and that the current prefix should not be prepended to that symbol. The `:` is then not considered part of the symbol. For a symbol in the label position (first on a line), a `:` at the end of a symbol is silently stripped off. A label is permitted, but not required, to be followed by a `:`, as with many other assembly formats.

The character `@` in an expression, is a synonym for `.`, the current location.

In addition to the common forward and backward local symbol formats (refer to Section 6.3 *Symbol Names*), they can be specified with upper-case `B` and `F`, as in `8B` and `9F`. A local label defined for the current position is written with a `H` appended to the number:

```
3H LDB $0,$1,2
```

This and traditional local-label formats cannot be mixed: a label must be defined and referred to using the same format.

There's a minor caveat: just as for the ordinary local symbols, the local symbols are translated into ordinary symbols using control characters are to hide the ordinal number of the symbol. Unfortunately, these symbols are not translated back in error messages. Thus you may see confusing error messages when local symbols are used. Control characters `\003` (control-C) and `\004` (control-D) are used for the MMIX-specific local-symbol syntax.

The symbol `Main` is handled specially; it is always global.

By defining the symbols `__.MMIX.start..text` and `__.MMIX.start..data`, the address of respectively the `.text` and `.data` segments of the final program can be defined, though when linking more than one object file, the code or data in the object file containing the symbol is not guaranteed to be start at that position; just the final executable.

30.3.3. Register names

Local and global registers are specified as `$0` to `$255`. The recognized special register names are `rJ`, `rA`, `rB`, `rC`, `rD`, `rE`, `rF`, `rG`, `rH`, `rI`, `rK`, `rL`, `rM`, `rN`, `rO`, `rP`, `rQ`, `rR`, `rS`, `rT`, `rU`, `rV`, `rW`, `rX`, `rY`, `rZ`, `rBB`, `rTT`, `rWW`, `rXX`, `rYY` and `rZZ`. A leading `:` is optional for special register names.

Local and global symbols can be equated to register names and used in place of ordinary registers.

Similarly for special registers, local and global symbols can be used. Also, symbols equated from numbers and constant expressions are allowed in place of a special register, except when either of the options `-no-predefined-syms` and `-fixed-special-register-names` are specified. Then only the special register names above are allowed for the instructions having a special register operand; `GET` and `PUT`.

30.3.4. Assembler Directives

`LOC`

Sets the current location to the value of the operand field

The `LOC` directive sets the current location to the value of the operand field, which may include changing sections. If the operand is a constant, the section is set to either `.data` if the value is `0x2000000000000000` or larger, else it is set to `.text`. Within a section, the current location may only be changed to monotonically higher addresses. A LOC expression must be a previously defined symbol or a "pure" constant.

An example, which sets the label `prev` to the current location, and updates the current location to eight bytes forward:

```
prev LOC @+8
```

When a LOC has a constant as its operand, a symbol `__.MMIX.start..text` or `__.MMIX.start..data` is defined depending on the address as mentioned above. Each such symbol is interpreted as special by the linker, locating the section at that address. Note that if multiple files are linked, the first object file with that section will be mapped to that address (not necessarily the file with the LOC definition).

`LOCAL`

Generates a link-time assertion that the operand does not correspond to a global register.

Example:

```
 LOCAL external_symbol
 LOCAL 42
 .local asymbol
```

This directive-operation generates a link-time assertion that the operand does not correspond to a global register. The operand is an expression that at link-time resolves to a register symbol or a number. A number is treated as the register having that number. There is one restriction on the use of this directive: the pseudo-directive must be placed in a section with contents, code or data.

`IS`

Sets the symbol `asymbol` to `an_expression`.

The `IS` directive:

```
asymbol IS an_expression
```

sets the symbol `asymbol` to `an_expression`. A symbol may not be set more than once using this directive. Local labels may be set using this directive, for example:

```
5H IS @+4
```

`GREG`

This directive reserves a global register, gives it an initial value and optionally gives it a symbolic name. Some examples:

```
areg GREG
breg GREG data_value
     GREG data_buffer
     .greg creg, another_data_value
```

The symbolic register name can be used in place of a (non-special) register. If a value isn't provided, it defaults to zero. Unless the option `-no-merge-gregs` is specified, non-zero registers allocated with this directive may be eliminated by `as`; another register with the same value used in its place. Any of the instructions `CSWAP`, `GO`, `LDA`, `LDBU`, `LDB`, `LDHT`, `LDOU`, `LDO`, `LDSF`, `LDTU`, `LDT`, `LDUNC`, `LDVTS`, `LDWU`, `LDW`, `PREGO`, `PRELD`, `PREST`, `PUSHGO`, `STBU`, `STB`, `STCO`, `STHT`, `STOU`, `STSF`, `STTU`, `STT`, `STUNC`, `SYNCD`, `SYNCID`, can have a value nearby an initial value in place of its second and third operands. Here, "nearby" is defined as within the range 0. . . 255 from the initial value of such an allocated register.

```
buffer1 BYTE 0,0,0,0,0
buffer2 BYTE 0,0,0,0,0
 ...
 GREG buffer1
 LDOU $42,buffer2
```

In the example above, the `Y` field of the `LDOUI` instruction (LDOU with a constant Z) will be replaced with the global register allocated for `buffer1`, and the `Z` field will have the value 5, the offset from `buffer1` to `buffer2`. The result is equivalent to this code:

```
buffer1 BYTE 0,0,0,0,0
buffer2 BYTE 0,0,0,0,0
 ...
tmpreg GREG buffer1
 LDOU $42,tmpreg,(buffer2-buffer1)
```

Global registers allocated with this directive are allocated in order higher-to-lower within a file. Other than that, the exact order of register allocation and elimination is undefined. For example, the order is undefined when more than one file with such directives are linked together. With the options `-x` and `-linker-allocated-gregs`, `GREG` directives for two-operand cases like the one mentioned above can be omitted. Sufficient global registers will then be allocated by the linker.

`BYTE`

The `BYTE` directive takes a series of operands separated by a comma. If an operand is a string (refer to Section 4.6.1.1 *Strings*), each character of that string is emitted as a byte. Other operands must be constant expressions without forward references, in the range 0. . . 255. If you need operands having expressions with forward references, use `.byte` (refer to Section 8.7 *`.byte expressions`*). An operand can be omitted, defaulting to a zero value.

`WYDE`
`TETRA`
`OCTA`

The directives `WYDE`, `TETRA` and `OCTA` emit constants of two, four and eight bytes size respectively. Before anything else happens for the directive, the current location is aligned to the respective constant-size bondary. If a label is defined at the beginning of the line, its value will be that after the alignment. A single operand can be omitted, defaulting to a zero value emitted for the directive. Operands can be expressed as strings (refer to Section 4.6.1.1 *Strings*), in which case each character in the string is emitted as a separate constant of the size indicated by the directive.

`PREFIX`

The `PREFIX` directive sets a symbol name prefix to be prepended to all symbols (except local symbols, Section 30.3.2 *Symbols*), that are not prefixed with `:`, until the next `PREFIX` directive. Such prefixes accumulate. For example,

```
 PREFIX a
 PREFIX b
c IS 0
```

defines a symbol `abc` with the value 0.

`BSPEC`
`ESPEC`

A pair of `BSPEC` and `ESPEC` directives delimit a section of special contents (without specified semantics). Example:

```
 BSPEC 42
 TETRA 1,2,3
 ESPEC
```

The single operand to `BSPEC` must be number in the range 0. . . 255. The `BSPEC` number 80 is used by the GNU binutils implementation.

30.4. Differences to `mmixal`

The binutils `as` and `ld` combination has a few differences in function compared to `mmixal`.

The replacement of a symbol with a GREG-allocated register is not handled the exactly same way in `as` as in `mmixal`. This is apparent in the `mmixal` example file `inout.mms`, where different registers with different offsets, eventually yielding the same address, are used in the first instruction. This type of difference should however not affect the function of any program unless it has specific assumptions about the allocated register number.

Line numbers (in the `mmo` object format) are currently not supported.

Expression operator precedence is not that of mmixal: operator precedence is that of the C programming language. It's recommended to use parentheses to explicitly specify wanted operator precedence whenever more than one type of operators are used.

The serialize unary operator `&`, the fractional division operator `//`, the logical not operator `!` and the modulus operator `%` are not available.

Symbols are not global by default, unless the option `-globalize-symbols` is passed. Use the `.global` directive to globalize symbols (refer to Section 8.39 *`.global symbol`, `.globl symbol`*).

Operand syntax is a bit stricter with `as` than `mmixal`. For example, you can't say `addu 1,2,3`, instead you must write `addu $1,$2,3`.

You can't LOC to a lower address than those already visited (i.e. "backwards").

A LOC directive must come before any emitted code.

Predefined symbols are visible as file-local symbols after use. (In the ELF file, that is--the linked mmo file has no notion of a file-local symbol.)

Some mapping of constant expressions to sections in LOC expressions is attempted, but that functionality is easily confused and should be avoided unless compatibility with `mmixal` is required. A LOC expression to `0x2000000000000000` or higher, maps to the `.data` section and lower addresses map to the `.text` section.

The code and data areas are each contiguous. Sparse programs with far-away LOC directives will take up the same amount of space as a contiguous program with zeros filled in the gaps between the LOC directives. If you need sparse programs, you might try and get the wanted effect with a linker script and splitting up the code parts into sections (refer to Section 8.74 *`.section name`*). Assembly code for this, to be compatible with `mmixal`, would look something like:

```
 .if 0
 LOC away_expression
 .else
 .section away,"ax"
 .fi
```

`as` will not execute the LOC directive and `mmixal` ignores the lines with `.`. This construct can be used generally to help compatibility.

Symbols can't be defined twice-not even to the same value.

Instruction mnemonics are recognized case-insensitive, though the `IS` and `GREG` pseudo-operations must be specified in upper-case characters.

There's no unicode support.

The following is a list of programs in `mmix.tar.gz`, available at *http://www-cs-faculty.stanford.edu/~knuth/mmix-news.html*, last checked with the version dated 2001-08-25 (md5sum c393470cfc86fac040487d22d2bf0172) that assemble with `mmixal` but do not assemble with `as`:

`silly.mms`

LOC to a previous address.

`sim.mms`

Redefines symbol `Done`.

`test.mms`

Uses the serial operator `&`.

Chapter 31. MSP 430 Dependent Features

31.1. Options

as has only -m flag which selects the mpu arch. Currently has no effect.

31.2. Syntax

31.2.1. Macros

The macro syntax used on the MSP 430 is like that described in the MSP 430 Family Assembler Specification. Normal as macros should still work.

Additional built-in macros are:

llo(exp)

Extracts least significant word from 32-bit expression 'exp'.

lhi(exp)

Extracts most significant word from 32-bit expression 'exp'.

hlo(exp)

Extracts 3rd word from 64-bit expression 'exp'.

hhi(exp)

Extracts 4rd word from 64-bit expression 'exp'.

They normally being used as an immediate source operand.

```
mov #llo(1), r10 ; == mov #1, r10
mov #lhi(1), r10 ; == mov #0, r10
```

31.2.2. Special Characters

; is the line comment character.

The character $ in jump instructions indicates current location and implemented only for TI syntax compatibility.

31.2.3. Register Names

General-purpose registers are represented by predefined symbols of the form rN (for global registers), where N represents a number between 0 and 15. The leading letters may be in either upper or lower case; for example, r13 and R7 are both valid register names.

Register names PC, SP and SR cannot be used as register names and will be treated as variables. Use r0, r1, and r2 instead.

31.2.4. Assembler Extensions

`@rN`

As destination operand being treated as `0(rn)`

`0(rN)`

As source operand being treated as `@rn`

`jCOND +N`

Skips next N bytes followed by jump instruction and equivalent to `jCOND $+N+2`

31.3. Floating Point

The MSP 430 family uses ieee 32-bit floating-point numbers.

31.4. MSP 430 Machine Directives

`.file`

This directive is ignored; it is accepted for compatibility with other MSP 430 assemblers.

> *Warning:* in other versions of the gnu assembler, `.file` is used for the directive called `.app-file` in the MSP 430 support.

`.line`

This directive is ignored; it is accepted for compatibility with other MSP 430 assemblers.

`.arch`

Currently this directive is ignored; it is accepted for compatibility with other MSP 430 assemblers.

31.5. Opcodes

`as` implements all the standard MSP 430 opcodes. No additional pseudo-instructions are needed on this family.

For information on the 430 machine instruction set, see [MSP430 User's Manual, document slau049b], Texas Instrument, Inc.

Chapter 32. PDP-11 Dependent Features

32.1. Options

The PDP-11 version of `as` has a rich set of machine dependent options.

32.1.1. Code Generation Options

`-mpic | -mno-pic`

Generate position-independent (or position-dependent) code.

The default is to generate position-independent code.

32.1.2. Instruction Set Extension Options

These options enables or disables the use of extensions over the base line instruction set as introduced by the first PDP-11 CPU: the KA11. Most options come in two variants: a `-mextension` that enables `extension`, and a `-mno-extension` that disables `extension`.

The default is to enable all extensions.

`-mall | -mall-extensions`

Enable all instruction set extensions.

`-mno-extensions`

Disable all instruction set extensions.

`-mcis | -mno-cis`

Enable (or disable) the use of the commercial instruction set, which consists of these instructions: `ADDNI`, `ADDN`, `ADDPI`, `ADDP`, `ASHNI`, `ASHN`, `ASHPI`, `ASHP`, `CMPCI`, `CMPC`, `CMPNI`, `CMPN`, `CMPPI`, `CMPP`, `CVTLNI`, `CVTLN`, `CVTLPI`, `CVTLP`, `CVTNLI`, `CVTNL`, `CVTNPI`, `CVTNP`, `CVTPLI`, `CVTPL`, `CVTPNI`, `CVTPN`, `DIVPI`, `DIVP`, `L2DR`, `L3DR`, `LOCCI`, `LOCC`, `MATCI`, `MATC`, `MOVCI`, `MOVC`, `MOVRCI`, `MOVRC`, `MOVTCI`, `MOVTC`, `MULPI`, `MULP`, `SCANCI`, `SCANC`, `SKPCI`, `SKPC`, `SPANCI`, `SPANC`, `SUBNI`, `SUBN`, `SUBPI`, and `SUBP`.

`-mcsm | -mno-csm`

Enable (or disable) the use of the `CSM` instruction.

`-meis | -mno-eis`

Enable (or disable) the use of the extended instruction set, which consists of these instructions: `ASHC`, `ASH`, `DIV`, `MARK`, `MUL`, `RTT`, `SOB SXT`, and `XOR`.

`-mfis | -mkev11`
`-mno-fis | -mno-kev11`

Enable (or disable) the use of the KEV11 floating-point instructions: `FADD`, `FDIV`, `FMUL`, and `FSUB`.

`-mfpp | -mfpu | -mfp-11`
`-mno-fpp | -mno-fpu | -mno-fp-11`

Enable (or disable) the use of FP-11 floating-point instructions: `ABSF`, `ADDF`, `CFCC`, `CLRF`, `CMPF`, `DIVF`, `LDCFF`, `LDCIF`, `LDEXP`, `LDF`, `LDFPS`, `MODF`, `MULF`, `NEGF`, `SETD`, `SETF`, `SETI`, `SETL`, `STCFF`, `STCFI`, `STEXP`, `STF`, `STFPS`, `STST`, `SUBF`, and `TSTF`.

`-mlimited-eis | -mno-limited-eis`

Enable (or disable) the use of the limited extended instruction set: `MARK`, `RTT`, `SOB`, `SXT`, and `XOR`.

The -mno-limited-eis options also implies -mno-eis.

`-mmfpt | -mno-mfpt`

Enable (or disable) the use of the `MFPT` instruction.

`-mmultiproc | -mno-multiproc`

Enable (or disable) the use of multiprocessor instructions: `TSTSET` and `WRTLCK`.

`-mmxps | -mno-mxps`

Enable (or disable) the use of the `MFPS` and `MTPS` instructions.

`-mspl | -mno-spl`

Enable (or disable) the use of the `SPL` instruction.

Enable (or disable) the use of the microcode instructions: `LDUB`, `MED`, and `XFC`.

32.1.3. CPU Model Options

These options enable the instruction set extensions supported by a particular CPU, and disables all other extensions.

`-mka11`

KA11 CPU. Base line instruction set only.

`-mkb11`

KB11 CPU. Enable extended instruction set and `SPL`.

`-mkd11a`

KD11-A CPU. Enable limited extended instruction set.

`-mkd11b`

KD11-B CPU. Base line instruction set only.

`-mkd11d`

KD11-D CPU. Base line instruction set only.

`-mkd11e`

KD11-E CPU. Enable extended instruction set, `MFPS`, and `MTPS`.

`-mkd11f | -mkd11h | -mkd11q`

KD11-F, KD11-H, or KD11-Q CPU. Enable limited extended instruction set, `MFPS`, and `MTPS`.

`-mkd11k`

KD11-K CPU. Enable extended instruction set, `LDUB`, `MED`, `MFPS`, `MFPT`, `MTPS`, and `XFC`.

`-mkd11z`

KD11-Z CPU. Enable extended instruction set, `CSM`, `MFPS`, `MFPT`, `MTPS`, and `SPL`.

`-mf11`

F11 CPU. Enable extended instruction set, `MFPS`, `MFPT`, and `MTPS`.

`-mj11`

J11 CPU. Enable extended instruction set, `CSM`, `MFPS`, `MFPT`, `MTPS`, `SPL`, `TSTSET`, and `WRTLCK`.

`-mt11`

T11 CPU. Enable limited extended instruction set, `MFPS`, and `MTPS`.

32.1.4. Machine Model Options

These options enable the instruction set extensions supported by a particular machine model, and disables all other extensions.

`-m11/03`

Same as `-mkd11f`.

`-m11/04`

Same as `-mkd11d`.

`-m11/05 | -m11/10`

Same as `-mkd11b`.

`-m11/15 | -m11/20`

Same as `-mka11`.

`-m11/21`

Same as `-mt11`.

`-m11/23 | -m11/24`

Same as `-mf11`.

`-m11/34`

Same as `-mkd11e`.

`-m11/34a`

Ame as `-mkd11e -mfpp`.

`-m11/35 | -m11/40`

Same as `-mkd11a`.

`-m11/44`

Same as `-mkd11z`.

`-m11/45 | -m11/50 | -m11/55 | -m11/70`

Same as `-mkb11`.

`-m11/53 | -m11/73 | -m11/83 | -m11/84 | -m11/93 | -m11/94`

Same as `-mj11`.

`-m11/60`

Same as `-mkd11k`.

32.2. Assembler Directives

The PDP-11 version of `as` has a few machine dependent assembler directives.

`.bss`

Switch to the `bss` section.

`.even`

Align the location counter to an even number.

32.3. PDP-11 Assembly Language Syntax

`as` supports both DEC syntax and BSD syntax. The only difference is that in DEC syntax, a `#` character is used to denote an immediate constants, while in BSD syntax the character for this purpose is `$`.

eneral-purpose registers are named `r0` through `r7`. Mnemonic alternatives for `r6` and `r7` are `sp` and `pc`, respectively.

Floating-point registers are named `ac0` through `ac3`, or alternatively `fr0` through `fr3`.

Comments are started with a `#` or a `/` character, and extend to the end of the line. (FIXME: clash with immediates?)

32.4. Instruction Naming

Some instructions have alternative names.

`BCC`
: `BHIS`

`BCS`
: `BLO`

`L2DR`
: `L2D`

`L3DR`
: `L3D`

`SYS`
: `TRAP`

32.5. Synthetic Instructions

The `JBR` and `JCC` synthetic instructions are not supported yet.

Chapter 33. picoJava Dependent Features

33.1. Options

`as` has two additional command-line options for the picoJava architecture.

`-ml`

This option selects little endian data output.

`-mb`

This option selects big endian data output.

Chapter 34. PowerPC Dependent Features

34.1. Options

The PowerPC chip family includes several successive levels, using the same core instruction set, but including a few additional instructions at each level. There are exceptions to this however. For details on what instructions each variant supports, please see the chip's architecture reference manual.

The following table lists all available PowerPC options.

`-mpwrx | -mpwr2`

Generate code for POWER/2 (RIOS2).

`-mpwr`

Generate code for POWER (RIOS1)

`-m601`

Generate code for PowerPC 601.

`-mppc, -mppc32, -m603, -m604`

Generate code for PowerPC 603/604.

`-m403, -m405`

Generate code for PowerPC 403/405.

`-m7400, -m7410, -m7450, -m7455`

Generate code for PowerPC 7400/7410/7450/7455.

`-mppc64, -m620`

Generate code for PowerPC 620/625/630.

`-mppc64bridge`

Generate code for PowerPC 64, including bridge insns.

`-mbooke64`

Generate code for 64-bit BookE.

`-mbooke, mbooke32`

Generate code for 32-bit BookE.

`-maltivec`

Generate code for processors with AltiVec instructions.

`-mpower4`

Generate code for Power4 architecture.

`-mcom`

Generate code Power/PowerPC common instructions.

`-many`

Generate code for any architecture (PWR/PWRX/PPC).

`-mregnames`

Allow symbolic names for registers.

`-mno-regnames`

Do not allow symbolic names for registers.

`-mrelocatable`

Support for GCC's -mrelocatble option.

`-mrelocatable-lib`

Support for GCC's -mrelocatble-lib option.

`-memb`

Set PPC_EMB bit in ELF flags.

`-mlittle, -mlittle-endian`

Generate code for a little endian machine.

`-mbig, -mbig-endian`

Generate code for a big endian machine.

`-msolaris`

Generate code for Solaris.

`-mno-solaris`

Do not generate code for Solaris.

Chapter 35. Renesas / SuperH SH Dependent Features

35.1. Options

`as` has following command-line options for the Renesas (formerly Hitachi) / SuperH SH family.

`-little`
: Generate little endian code.

`-big`
: Generate big endian code.

`-relax`
: Alter jump instructions for long displacements.

`-small`
: Align sections to 4 byte boundaries, not 16.

`-dsp`
: Enable sh-dsp insns, and disable sh3e / sh4 insns.

35.2. Syntax

35.2.1. Special Characters

`!` is the line comment character.

You can use `;` instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

35.2.2. Register Names

You can use the predefined symbols `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, and `r15` to refer to the SH registers.

The SH also has these control registers:

`pr`
: procedure register (holds return address)

`pc`
: program counter

`mach`
`macl`

high and low multiply accumulator registers

`sr`

status register

`gbr`

global base register

`vbr`

vector base register (for interrupt vectors)

35.2.3. Addressing Modes

`as` understands the following addressing modes for the SH. `Rn` in the following refers to any of the numbered registers, but *not* the control registers.

`Rn`

Register direct

`@Rn`

Register indirect

`@-Rn`

Register indirect with pre-decrement

`@Rn+`

Register indirect with post-increment

`@(disp, Rn)`

Register indirect with displacement

`@(R0, Rn)`

Register indexed

`@(disp, GBR)`

`GBR` offset

`@(R0, GBR)`

GBR indexed

`addr`
`@(disp, PC)`

PC relative address (for branch or for addressing memory). The `as` implementation allows you to use the simpler form `addr` anywhere a PC relative address is called for; the alternate form is supported for compatibility with other assemblers.

`#imm`

Immediate data

35.3. Floating Point

The SH family has no hardware floating point, but the `.float` directive generates ieee floating-point numbers for compatibility with other development tools.

35.4. SH Machine Directives

`uaword`
`ualong`

`as` will issue a warning when a misaligned `.word` or `.long` directive is used. You may use `.uaword` or `.ualong` to indicate that the value is intentionally misaligned.

35.5. Opcodes

For detailed information on the SH machine instruction set, see [SH-Microcomputer User's Manual] (Renesas) or [SH-4 32-bit CPU Core Architecture] (SuperH) and [SuperH (SH) 64-Bit RISC Series] (SuperH).

`as` implements all the standard SH opcodes. No additional pseudo-instructions are needed on this family. Note, however, that because `as` supports a simpler form of PC-relative addressing, you may simply write (for example)

```
mov.l  bar,r0
```

where other assemblers might require an explicit displacement to `bar` from the program counter:

```
mov.l  @(disp, PC)
```

Chapter 36. SuperH SH64 Dependent Features

36.1. Options

`-isa=shmedia | -isa=shcompact`

Specify the default instruction set. `SHmedia` specifies the 32-bit opcodes, and `SHcompact` specifies the 16-bit opcodes compatible with previous SH families. The default depends on the ABI selected; the default for the 64-bit ABI is SHmedia, and the default for the 32-bit ABI is SHcompact. If neither the ABI nor the ISA is specified, the default is 32-bit SHcompact.

Note that the `.mode` pseudo-op is not permitted if the ISA is not specified on the command line.

`-abi=32 | -abi=64`

Specify the default ABI. If the ISA is specified and the ABI is not, the default ABI depends on the ISA, with SHmedia defaulting to 64-bit and SHcompact defaulting to 32-bit.

Note that the `.abi` pseudo-op is not permitted if the ABI is not specified on the command line. When the ABI is specified on the command line, any `.abi` pseudo-ops in the source must match it.

`-shcompact-const-crange`

Emit code-range descriptors for constants in SHcompact code sections.

`-no-mix`

Disallow SHmedia code in the same section as constants and SHcompact code.

`-no-expand`

Do not expand MOVI, PT, PTA or PTB instructions.

`-expand-pt32`

With -abi=64, expand PT, PTA and PTB instructions to 32 bits only.

36.2. Syntax

36.2.1. Special Characters

`!` is the line comment character.

You can use `;` instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

36.2.2. Register Names

You can use the predefined symbols `r0` through `r63` to refer to the SH64 general registers, `cr0` through `cr63` for control registers, `tr0` through `tr7` for target address registers, `fr0` through `fr63` for single-precision floating point registers, `dr0` through `dr62` (even numbered registers only) for double-precision floating point registers, `fv0` through `fv60` (multiples of four only) for single-precision floating point vectors, `fp0` through `fp62` (even numbered registers only) for single-precision floating point pairs, `mtrx0` through `mtrx48` (multiples of 16 only) for 4x4 matrices of single-precision floating point registers, `pc` for the program counter, and `fpscr` for the floating point status and control register.

You can also refer to the control registers by the mnemonics `sr`, `ssr`, `pssr`, `intevt`, `expevt`, `pexpevt`, `tra`, `spc`, `pspc`, `resvec`, `vbr`, `tea`, `dcr`, `kcr0`, `kcr1`, `ctc`, and `usr`.

36.2.3. Addressing Modes

SH64 operands consist of either a register or immediate value. The immediate value can be a constant or label reference (or portion of a label reference), as in this example:

```
movi 4,r2
pt function, tr4
movi (function >> 16) & 65535,r0
shori function & 65535, r0
ld.l r0,4,r0
```

Instruction label references can reference labels in either SHmedia or SHcompact. To differentiate between the two, labels in SHmedia sections will always have the least significant bit set (i.e. they will be odd), which SHcompact labels will have the least significant bit reset (i.e. they will be even). If you need to reference the actual address of a label, you can use the `datalabel` modifier, as in this example:

```
.long function
.long datalabel function
```

In that example, the first longword may or may not have the least significant bit set depending on whether the label is an SHmedia label or an SHcompact label. The second longword will be the actual address of the label, regardless of what type of label it is.

36.3. SH64 Machine Directives

In addition to the SH directives, the SH64 provides the following directives:

`.mode [shmedia|shcompact]`
`.isa [shmedia|shcompact]`

Specify the ISA for the following instructions (the two directives are equivalent). Note that programs such as `objdump` rely on symbolic labels to determine when such mode switches occur (by checking the least significant bit of the label's address), so such mode/isa changes should always be followed by a label (in practice, this is true anyway). Note that you cannot use these directives if you didn't specify an ISA on the command line.

`.abi [32|64]`

Specify the ABI for the following instructions. Note that you cannot use this directive unless you specified an ABI on the command line, and the ABIs specified must match.

`.uaquad`

Like .uaword and .ualong, this allows you to specify an intenionally unaligned quadword (64 bit word).

36.4. Opcodes

For detailed information on the SH64 machine instruction set, see [SuperH 64 bit RISC Series Architecture Manual] (SuperH, Inc.).

`as` implements all the standard SH64 opcodes. In addition, the following pseudo-opcodes may be expanded into one or more alternate opcodes:

`movi`

If the value doesn't fit into a standard `movi` opcode, `as` will replace the `movi` with a sequence of `movi` and `shori` opcodes.

`pt`

This expands to a sequence of `movi` and `shori` opcode, followed by a `ptrel` opcode, or to a `pta` or `ptb` opcode, depending on the label referenced.

Chapter 37. SPARC Dependent Features

37.1. Options

The SPARC chip family includes several successive levels, using the same core instruction set, but including a few additional instructions at each level. There are exceptions to this however. For details on what instructions each variant supports, please see the chip's architecture reference manual.

By default, `as` assumes the core instruction set (SPARC v6), but "bumps" the architecture level as needed: it switches to successively higher architectures as it encounters instructions that only exist in the higher levels.

If not configured for SPARC v9 (`sparc64-*-*`) GAS will not bump passed sparclite by default, an option must be passed to enable the v9 instructions.

GAS treats sparclite as being compatible with v8, unless an architecture is explicitly requested. SPARC v9 is always incompatible with sparclite.

`-Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite`
`-Av8plus | -Av8plusa | -Av9 | -Av9a`

> Use one of the `-A` options to select one of the SPARC architectures explicitly. If you select an architecture explicitly, `as` reports a fatal error if it encounters an instruction or feature requiring an incompatible or higher level.
>
> `-Av8plus` and `-Av8plusa` select a 32 bit environment.
>
> `-Av9` and `-Av9a` select a 64 bit environment and are not available unless GAS is explicitly configured with 64 bit environment support.
>
> `-Av8plusa` and `-Av9a` enable the SPARC V9 instruction set with UltraSPARC extensions.

`-xarch=v8plus | -xarch=v8plusa`

> For compatibility with the Solaris v9 assembler. These options are equivalent to -Av8plus and -Av8plusa, respectively.

`-bump`

> Warn whenever it is necessary to switch to another level. If an architecture level is explicitly requested, GAS will not issue warnings until that level is reached, and will then bump the level as required (except between incompatible levels).

`-32 | -64`

> Select the word size, either 32 bits or 64 bits. These options are only available with the ELF object file format, and require that the necessary BFD support has been included.

37.2. Enforcing aligned data

SPARC GAS normally permits data to be misaligned. For example, it permits the `.long` pseudo-op to be used on a byte boundary. However, the native SunOS and Solaris assemblers issue an error when they see misaligned data.

You can use the `-enforce-aligned-data` option to make SPARC GAS also issue an error about misaligned data, just as the SunOS and Solaris assemblers do.

The `-enforce-aligned-data` option is not the default because gcc issues misaligned data pseudo-ops when it initializes certain packed data structures (structures defined using the `packed` attribute). You may have to assemble with GAS in order to initialize packed data structures in your own code.

37.3. Floating Point

The Sparc uses ieee floating-point numbers.

37.4. Sparc Machine Directives

The Sparc version of `as` supports the following additional machine directives:

`.align`

This must be followed by the desired alignment in bytes.

`.common`

This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.comm`, but the syntax is different.

`.half`

This is functionally identical to `.short`.

`.nword`

On the Sparc, the `.nword` directive produces native word sized value, ie. if assembling with -32 it is equivalent to `.word`, if assembling with -64 it is equivalent to `.xword`.

`.proc`

This directive is ignored. Any text following it on the same line is also ignored.

`.register`

This directive declares use of a global application or system register. It must be followed by a register name %g2, %g3, %g6 or %g7, comma and the symbol name for that register. If symbol name is `#scratch`, it is a scratch register, if it is `#ignore`, it just suppresses any errors about using undeclared global register, but does not emit any information about it into the object file. This can be useful e.g. if you save the register before use and restore it after.

`.reserve`

This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.lcomm`, but the syntax is different.

`.seg`

This must be followed by `"text"`, `"data"`, or `"data1"`. It behaves like `.text`, `.data`, or `.data 1`.

`.skip`

This is functionally identical to the `.space` directive.

`.word`

On the Sparc, the `.word` directive produces 32 bit values, instead of the 16 bit values it produces on many other machines.

`.xword`

On the Sparc V9 processor, the `.xword` directive produces 64 bit values.

redhat

Chapter 38. TIC54X Dependent Features

38.1. Options

The TMS320C54x version of `as` has a few machine-dependent options.

You can use the `-mfar-mode` option to enable extended addressing mode. All addresses will be assumed to be > 16 bits, and the appropriate relocation types will be used. This option is equivalent to using the `.far_mode` directive in the assembly code. If you do not use the `-mfar-mode` option, all references will be assumed to be 16 bits. This option may be abbreviated to `-mf`.

You can use the `-mcpu` option to specify a particular CPU. This option is equivalent to using the `.version` directive in the assembly code. For recognized CPU codes, see Section 38.9 *Directives*. The default CPU version is `542`.

You can use the `-merrors-to-file` option to redirect error output to a file (this provided for those deficient environments which don't provide adequate output redirection). This option may be abbreviated to `-me`.

38.2. Blocking

A blocked section or memory block is guaranteed not to cross the blocking boundary (usually a page, or 128 words) if it is smaller than the blocking size, or to start on a page boundary if it is larger than the blocking size.

38.3. Environment Settings

`C54XDSP_DIR` and `A_DIR` are semicolon-separated paths which are added to the list of directories normally searched for source and include files. `C54XDSP_DIR` will override `A_DIR`.

38.4. Constants Syntax

The TIC54X version of `as` allows the following additional constant formats, using a suffix to indicate the radix:

```
Binary                   000000B, 011000b
Octal                    10Q, 224q
Hexadecimal              45h, 0FH
```

38.5. String Substitution

A subset of allowable symbols (which we'll call subsyms) may be assigned arbitrary string values. This is roughly equivalent to C preprocessor #define macros. When `as` encounters one of these symbols, the symbol is replaced in the input stream by its string value. Subsym names *must* begin with a letter.

Subsyms may be defined using the `.asg` and `.eval` directives (refer to Section 38.9 *Directives* and Section 38.9 *Directives*.

Expansion is recursive until a previously encountered symbol is seen, at which point substitution stops.

In this example, x is replaced with SYM2; SYM2 is replaced with SYM1, and SYM1 is replaced with x. At this point, x has already been encountered and the substitution stops.

```
 .asg   "x",SYM1
 .asg   "SYM1",SYM2
 .asg   "SYM2",x
 add    x,a              ; final code assembled is "add  x, a"
```

Macro parameters are converted to subsyms; a side effect of this is the normal `as` '\ARG' dereferencing syntax is unnecessary. Subsyms defined within a macro will have global scope, unless the `.var` directive is used to identify the subsym as a local macro variable Section 38.9 *Directives*.

Substitution may be forced in situations where replacement might be ambiguous by placing colons on either side of the subsym. The following code:

```
 .eval  "10",x
LAB:X:  add     #x, a
```

When assembled becomes:

```
LAB10  add     #10, a
```

Smaller parts of the string assigned to a subsym may be accessed with the following syntax:

`:symbol(char_index):`

Evaluates to a single-character string, the character at `char_index`.

`:symbol(start,length):`

Evaluates to a substring of `symbol` beginning at `start` with length `length`.

38.6. Local Labels

Local labels may be defined in two ways:

- $N, where N is a decimal number between 0 and 9
- LABEL?, where LABEL is any legal symbol name.

Local labels thus defined may be redefined or automatically generated. The scope of a local label is based on when it may be undefined or reset. This happens when one of the following situations is encountered:

- .newblock directive Section 38.9 *Directives*
- The current section is changed (.sect, .text, or .data)
- Entering or leaving an included file
- The macro scope where the label was defined is exited

38.7. Math Builtins

The following built-in functions may be used to generate a floating-point value. All return a floating-point value except `$cvi`, `$int`, and `$sgn`, which return an integer value.

`$acos(expr)`

Returns the floating point arccosine of `expr`.

`$asin(expr)`

Returns the floating point arcsine of `expr`.

`$atan(expr)`

Returns the floating point arctangent of `expr`.

`$atan2(expr1,expr2)`

Returns the floating point arctangent of `expr1` / `expr2`.

`$ceil(expr)`

Returns the smallest integer not less than `expr` as floating point.

`$cosh(expr)`

Returns the floating point hyperbolic cosine of `expr`.

`$cos(expr)`

Returns the floating point cosine of `expr`.

`$cvf(expr)`

Returns the integer value `expr` converted to floating-point.

`$cvi(expr)`

Returns the floating point value `expr` converted to integer.

`$exp(expr)`

Returns the floating point value e ^ `expr`.

`$fabs(expr)`

Returns the floating point absolute value of `expr`.

`$floor(expr)`

Returns the largest integer that is not greater than `expr` as floating point.

`$fmod(expr1,expr2)`

Returns the floating point remainder of `expr1` / `expr2`.

`$int(expr)`

Returns 1 if `expr` evaluates to an integer, zero otherwise.

`$ldexp(expr1,expr2)`

Returns the floating point value `expr1` * 2 ^ `expr2`.

`$log10(expr)`

Returns the base 10 logarithm of `expr`.

`$log(expr)`

Returns the natural logarithm of `expr`.

`$max(expr1,expr2)`

Returns the floating point maximum of `expr1` and `expr2`.

`$min(expr1,expr2)`

Returns the floating point minimum of `expr1` and `expr2`.

`$pow(expr1,expr2)`

Returns the floating point value `expr1` ^ `expr2`.

`$round(expr)`

Returns the nearest integer to `expr` as a floating point number.

`$sgn(expr)`

Returns -1, 0, or 1 based on the sign of `expr`.

`$sin(expr)`

Returns the floating point sine of `expr`.

`$sinh(expr)`

Returns the floating point hyperbolic sine of `expr`.

`$sqrt(expr)`

Returns the floating point square root of `expr`.

`$tan(expr)`

Returns the floating point tangent of `expr`.

`$tanh(expr)`

Returns the floating point hyperbolic tangent of `expr`.

`$trunc(expr)`

Returns the integer value of `expr` truncated towards zero as floating point.

38.8. Extended Addressing

The `LDX` pseudo-op is provided for loading the extended addressing bits of a label or address. For example, if an address `_label` resides in extended program memory, the value of `_label` may be loaded as follows:

```
 ldx     #_label,16,a    ; loads extended bits of _label
 or      #_label,a       ; loads lower 16 bits of _label
 bacc    a               ; full address is in accumulator A
```

38.9. Directives

`.align [size]`
`.even`

Align the section program counter on the next boundary, based on `size`. `size` may be any power of 2. `.even` is equivalent to `.align` with a `size` of 2.

`1`

Align SPC to word boundary

`2`

Align SPC to longword boundary (same as .even)

`128`

Align SPC to page boundary

`.asg string, name`

Assign `name` the string `string`. String replacement is performed on `string` before assignment.

`.eval string, name`

Evaluate the contents of string `string` and assign the result as a string to the subsym `name`. String replacement is performed on `string` before assignment.

`.bss symbol, size [, [blocking_flag] [,alignment_flag]]`

Reserve space for `symbol` in the .bss section. `size` is in words. If present, `blocking_flag` indicates the allocated space should be aligned on a page boundary if it would otherwise cross a page boundary. If present, `alignment_flag` causes the assembler to allocate `size` on a long word boundary.

```
.byte value [,...,value_n]
.ubyte value [,...,value_n]
.char value [,...,value_n]
.uchar value [,...,value_n]
```

Place one or more bytes into consecutive words of the current section. The upper 8 bits of each word is zero-filled. If a label is used, it points to the word allocated for the first byte encountered.

```
.clink ["section_name"]
```

Set STYP_CLINK flag for this section, which indicates to the linker that if no symbols from this section are referenced, the section should not be included in the link. If `section_name` is omitted, the current section is used.

```
.c_mode
```

TBD.

```
.copy "filename" | filename
.include "filename" | filename
```

Read source statements from `filename`. The normal include search path is used. Normally .copy will cause statements from the included file to be printed in the assembly listing and .include will not, but this distinction is not currently implemented.

```
.data
```

Begin assembling code into the .data section.

```
.double value [,...,value_n]
.ldouble value [,...,value_n]
.float value [,...,value_n]
.xfloat value [,...,value_n]
```

Place an IEEE single-precision floating-point representation of one or more floating-point values into the current section. All but `.xfloat` align the result on a longword boundary. Values are stored most-significant word first.

```
.drlist
.drnolist
```

Control printing of directives to the listing file. Ignored.

```
.emsg string
.mmsg string
.wmsg string
```

Emit a user-defined error, message, or warning, respectively.

```
.far_mode
```

Use extended addressing when assembling statements. This should appear only once per file, and is equivalent to the -mfar-mode option Section 38.1 *Options*.

```
.fclist
.fcnolist
```

Control printing of false conditional blocks to the listing file.

`.field value [,size]`

Initialize a bitfield of `size` bits in the current section. If `value` is relocatable, then `size` must be 16. `size` defaults to 16 bits. If `value` does not fit into `size` bits, the value will be truncated. Successive `.field` directives will pack starting at the current word, filling the most significant bits first, and aligning to the start of the next word if the field size does not fit into the space remaining in the current word. A `.align` directive with an operand of 1 will force the next `.field` directive to begin packing into a new word. If a label is used, it points to the word that contains the specified field.

```
.global symbol [,...,symbol_n]
.def symbol [,...,symbol_n]
.ref symbol [,...,symbol_n]
```

`.def` nominally identifies a symbol defined in the current file and availalbe to other files. `.ref` identifies a symbol used in the current file but defined elsewhere. Both map to the standard `.global` directive.

```
.half value [,...,value_n]
.uhalf value [,...,value_n]
.short value [,...,value_n]
.ushort value [,...,value_n]
.int value [,...,value_n]
.uint value [,...,value_n]
.word value [,...,value_n]
.uword value [,...,value_n]
```

Place one or more values into consecutive words of the current section. If a label is used, it points to the word allocated for the first value encountered.

`.label symbol`

Define a special `symbol` to refer to the load time address of the current section program counter.

```
.length
.width
```

Set the page length and width of the output listing file. Ignored.

```
.list
.nolist
```

Control whether the source listing is printed. Ignored.

```
.long value [,...,value_n]
.ulong value [,...,value_n]
.xlong value [,...,value_n]
```

Place one or more 32-bit values into consecutive words in the current section. The most significant word is stored first. `.long` and `.ulong` align the result on a longword boundary; `xlong` does not.

```
.loop [count]
.break [condition]
.endloop
```

Repeatedly assemble a block of code. `.loop` begins the block, and `.endloop` marks its termination. `count` defaults to 1024, and indicates the number of times the block should be repeated.

`.break` terminates the loop so that assembly begins after the `.endloop` directive. The optional `condition` will cause the loop to terminate only if it evaluates to zero.

`macro_name .macro [param1][,...param_n]`
`[.mexit]`
`.endm`

See the section on macros for more explanation (Section 38.10 *Macros*.

`.mlib "filename" | filename`

Load the macro library `filename`. `filename` must be an archived library (BFD ar-compatible) of text files, expected to contain only macro definitions. The standard include search path is used.

`.mlist`
`.mnolist`

Control whether to include macro and loop block expansions in the listing output. Ignored.

`.mmregs`

Define global symbolic names for the 'c54x registers. Supposedly equivalent to executing `.set` directives for each register with its memory-mapped value, but in reality is provided only for compatibility and does nothing.

`.newblock`

This directive resets any TIC54X local labels currently defined. Normal `as` local labels are unaffected.

`.option option_list`

Set listing options. Ignored.

`.sblock "section_name" | section_name [,"name_n" | name_n]`

Designate `section_name` for blocking. Blocking guarantees that a section will start on a page boundary (128 words) if it would otherwise cross a page boundary. Only initialized sections may be designated with this directive. See also Section 38.2 *Blocking*.

`.sect "section_name"`

Define a named initialized section and make it the current section.

`symbol .set "value"`
`symbol .equ "value"`

Equate a constant `value` to a `symbol`, which is placed in the symbol table. `symbol` may not be previously defined.

`.space size_in_bits`
`.bes size_in_bits`

Reserve the given number of bits in the current section and zero-fill them. If a label is used with `.space`, it points to the *first* word reserved. With `.bes`, the label points to the *last* word reserved.

```
.sslist
.ssnolist
```

Controls the inclusion of subsym replacement in the listing output. Ignored.

```
.string "string" [,...,"string_n"]
.pstring "string" [,...,"string_n"]
```

Place 8-bit characters from `string` into the current section. `.string` zero-fills the upper 8 bits of each word, while `.pstring` puts two characters into each word, filling the most-significant bits first. Unused space is zero-filled. If a label is used, it points to the first word initialized.

```
[stag] .struct [offset]
[name_1] element [count_1]
[name_2] element [count_2]
[tname] .tag stagx [tcount]
...
[name_n] element [count_n]
[ssize] .endstruct
label .tag [stag]
```

Assign symbolic offsets to the elements of a structure. `stag` defines a symbol to use to reference the structure. `offset` indicates a starting value to use for the first element encountered; otherwise it defaults to zero. Each element can have a named offset, `name`, which is a symbol assigned the value of the element's offset into the structure. If `stag` is missing, these become global symbols. `count` adjusts the offset that many times, as if `element` were an array. `element` may be one of `.byte`, `.word`, `.long`, `.float`, or any equivalent of those, and the structure offset is adjusted accordingly. `.field` and `.string` are also allowed; the size of `.field` is one bit, and `.string` is considered to be one word in size. Only element descriptors, structure/union tags, `.align` and conditional assembly directives are allowed within `.struct`/`.endstruct`. `.align` aligns member offsets to word boundaries only. `ssize`, if provided, will always be assigned the size of the structure.

The `.tag` directive, in addition to being used to define a structure/union element within a structure, may be used to apply a structure to a symbol. Once applied to `label`, the individual structure elements may be applied to `label` to produce the desired offsets using `label` as the structure base.

```
.tab
```

Set the tab size in the output listing. Ignored.

```
[utag] .union
[name_1] element [count_1]
[name_2] element [count_2]
[tname] .tag utagx[,tcount]
...
[name_n] element [count_n]
[usize] .endstruct
label .tag [utag]
```

Similar to `.struct`, but the offset after each element is reset to zero, and the `usize` is set to the maximum of all defined elements. Starting offset for the union is always zero.

`[symbol] .usect "section_name", size, [,[blocking_flag] [,alignment_flag]]`

Reserve space for variables in a named, uninitialized section (similar to .bss). `.usect` allows definitions sections independent of .bss. `symbol` points to the first location reserved by this allocation. The symbol may be used as a variable name. `size` is the allocated size in words. `blocking_flag` indicates whether to block this section on a page boundary (128 words) (refer to Section 38.2 *Blocking*). `alignment flag` indicates whether the section should be longword-aligned.

`.var sym[,..., sym_n]`

Define a subsym to be a local variable within a macro. See Section 38.10 *Macros*.

`.version version`

Set which processor to build instructions for. Though the following values are accepted, the op is ignored.

```
541
542
543
545
545LP
546LP
548
549
```

Version directives

38.10. Macros

Macros do not require explicit dereferencing of arguments (i.e. \ARG).

During macro expansion, the macro parameters are converted to subsyms. If the number of arguments passed the macro invocation exceeds the number of parameters defined, the last parameter is assigned the string equivalent of all remaining arguments. If fewer arguments are given than parameters, the missing parameters are assigned empty strings. To include a comma in an argument, you must enclose the argument in quotes.

The following built-in subsym functions allow examination of the string value of subsyms (or ordinary strings). The arguments are strings unless otherwise indicated (subsyms passed as args will be replaced by the strings they represent).

`$symlen(str)`

Returns the length of `str`.

`$symcmp(str1,str2)`

Returns 0 if `str1` == `str2`, non-zero otherwise.

`$firstch(str,ch)`

Returns index of the first occurrence of character constant `ch` in `str`.

`$lastch(str,ch)`

Returns index of the last occurrence of character constant `ch` in `str`.

`$isdefed(symbol)`

Returns zero if the symbol `symbol` is not in the symbol table, non-zero otherwise.

`$ismember(symbol,list)`

Assign the first member of comma-separated string `list` to `symbol`; `list` is reassigned the remainder of the list. Returns zero if `list` is a null string. Both arguments must be subsyms.

`$iscons(expr)`

Returns 1 if string `expr` is binary, 2 if octal, 3 if hexadecimal, 4 if a character, 5 if decimal, and zero if not an integer.

`$isname(name)`

Returns 1 if `name` is a valid symbol name, zero otherwise.

`$isreg(reg)`

Returns 1 if `reg` is a valid predefined register name (AR0-AR7 only).

`$structsz(stag)`

Returns the size of the structure or union represented by `stag`.

`$structacc(stag)`

Returns the reference point of the structure or union represented by `stag`. Always returns zero.

38.11. Memory-mapped Registers

The following symbols are recognized as memory-mapped registers:

Chapter 39.
Z8000 Dependent Features

The Z8000 as supports both members of the Z8000 family: the unsegmented Z8002, with 16 bit addresses, and the segmented Z8001 with 24 bit addresses.

When the assembler is in unsegmented mode (specified with the `unsegm` directive), an address takes up one word (16 bit) sized register. When the assembler is in segmented mode (specified with the `segm` directive), a 24-bit address takes up a long (32 bit) register. Section 39.3 *Assembler Directives for the Z8000*, for a list of other Z8000 specific assembler directives.

39.1. Options

`as` has no additional command-line options for the Zilog Z8000 family.

39.2. Syntax

39.2.1. Special Characters

`!` is the line comment character.

You can use `;` instead of a newline to separate statements.

39.2.2. Register Names

The Z8000 has sixteen 16 bit registers, numbered 0 to 15. You can refer to different sized groups of registers by register number, with the prefix `r` for 16 bit registers, `rr` for 32 bit registers and `rq` for 64 bit registers. You can also refer to the contents of the first eight (of the sixteen 16 bit registers) by bytes. They are named `rnh` and `rnl`.

```
byte registers
     r0l r0h r1h r1l r2h r2l r3h r3l r4h r4l r5h r5l r6h r6l r7h r7l

word registers
     r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15

long word registers
     rr0 rr2 rr4 rr6 rr8 rr10 rr12 rr14

quad word registers
     rq0 rq4 rq8 rq12
```

39.2.3. Addressing Modes

as understands the following addressing modes for the Z8000:

`rn`

Register direct

`@rn`

Indirect register

`addr`

Direct: the 16 bit or 24 bit address (depending on whether the assembler is in segmented or unsegmented mode) of the operand is in the instruction.

`address(rn)`

Indexed: the 16 or 24 bit address is added to the 16 bit register to produce the final address in memory of the operand.

`rn(#imm)`

Base Address: the 16 or 24 bit register is added to the 16 bit sign extended immediate displacement to produce the final address in memory of the operand.

`rn(rm)`

Base Index: the 16 or 24 bit register `rn` is added to the sign extended 16 bit index register `rm` to produce the final address in memory of the operand.

`#xx`

Immediate data `xx`.

39.3. Assembler Directives for the Z8000

The Z8000 port of as includes these additional assembler directives, for compatibility with other Z8000 assemblers. As shown, these do not begin with `.` (unlike the ordinary as directives).

`segm`

Generates code for the segmented Z8001.

`unsegm`

Generates code for the unsegmented Z8002.

`name`

Synonym for `.file`

`global`

Synonym for `.global`

`wval`

Synonym for `.word`

`lval`

Synonym for `.long`

`bval`

Synonym for `.byte`

`sval`

Assemble a string. `sval` expects one string literal, delimited by single quotes. It assembles each byte of the string into consecutive addresses. You can use the escape sequence `%xx` (where `xx` represents a two-digit hexadecimal number) to represent the character whose ascii value is `xx`. Use this feature to describe single quote and other characters that may not appear in string literals as themselves. For example, the C statement `char *a = "he said \"it's 50% off\"";` is represented in Z8000 assembly language (shown with the assembler output in hex at the left) as

```
68652073    sval    'he said %22it%27s 50%25 off%22%00'
61696420
22697427
73203530
25206F66
662200
```

`rsect`

synonym for `.section`

`block`

synonym for `.space`

`even`

special case of `.align`; aligns output to even byte boundary.

39.4. Opcodes

For detailed information on the Z8000 machine instruction set, see [Z8000 Technical Manual].

Chapter 40. VAX Dependent Features

40.1. VAX Command-Line Options

The Vax version of `as` accepts any of the following options, gives a warning message that the option was ignored and proceeds. These options are for compatibility with scripts designed for other people's assemblers.

`-D (Debug)`
`-S (Symbol Table)`
`-T (Token Trace)`

These are obsolete options used to debug old assemblers.

`-d (Displacement size for JUMPs)`

This option expects a number following the `-d`. Like options that expect filenames, the number may immediately follow the `-d` (old standard) or constitute the whole of the command line argument that follows `-d` (gnu standard).

`-V (Virtualize Interpass Temporary File)`

Some other assemblers use a temporary file. This option commanded them to keep the information in active memory rather than in a disk file. `as` always does this, so this option is redundant.

`-J (JUMPify Longer Branches)`

Many 32-bit computers permit a variety of branch instructions to do the same job. Some of these instructions are short (and fast) but have a limited range; others are long (and slow) but can branch anywhere in virtual memory. Often there are 3 flavors of branch: short, medium and long. Some other assemblers would emit short and medium branches, unless told by this option to emit short and long branches.

`-t (Temporary File Directory)`

Some other assemblers may use a temporary file, and this option takes a filename being the directory to site the temporary file. Since `as` does not use a temporary disk file, this option makes no difference. `-t` needs exactly one filename.

The Vax version of the assembler accepts additional options when compiled for VMS:

`-h n`

External symbol or section (used for global variables) names are not case sensitive on VAX/VMS and always mapped to upper case. This is contrary to the C language definition which explicitly distinguishes upper and lower case. To implement a standard conforming C compiler, names must be changed (mapped) to preserve the case information. The default mapping is to convert all lower case characters to uppercase and adding an underscore followed by a 6 digit hex value, representing a 24 digit binary value. The one digits in the binary value represent which characters are uppercase in the original symbol name.

The `-h n` option determines how we map names. This takes several values. No `-h` switch at all allows case hacking as described above. A value of zero (`-h0`) implies names should be upper case, and inhibits the case hack. A value of 2 (`-h2`) implies names should be all lower case, with no case hack. A value of 3 (`-h3`) implies that case should be preserved. The value 1 is unused. The `-H` option directs `as` to display every mapped symbol during assembly.

Symbols whose names include a dollar sign `$` are exceptions to the general name mapping. These symbols are normally only used to reference VMS library names. Such symbols are always mapped to upper case.

`-+`

The `-+` option causes `as` to truncate any symbol name larger than 31 characters. The `-+` option also prevents some code following the `_main` symbol normally added to make the object file compatible with Vax-11 "C".

`-1`

This option is ignored for backward compatibility with `as` version 1.x.

`-H`

The `-H` option causes `as` to print every symbol which was changed by case mapping.

40.2. VAX Floating Point

Conversion of flonums to floating point is correct, and compatible with previous assemblers. Rounding is towards zero if the remainder is exactly half the least significant bit.

`D`, `F`, `G` and `H` floating point formats are understood.

Immediate floating literals (*e.g.* `S`$6.9`) are rendered correctly. Again, rounding is towards zero in the boundary case.

The `.float` directive produces `f` format numbers. The `.double` directive produces `d` format numbers.

40.3. Vax Machine Directives

The Vax version of the assembler supports four directives for generating Vax floating point constants. They are described in the table below.

`.dfloat`

This expects zero or more flonums, separated by commas, and assembles Vax `d` format 64-bit floating point constants.

`.ffloat`

This expects zero or more flonums, separated by commas, and assembles Vax `f` format 32-bit floating point constants.

`.gfloat`

This expects zero or more flonums, separated by commas, and assembles Vax `g` format 64-bit floating point constants.

`.hfloat`

This expects zero or more flonums, separated by commas, and assembles Vax `h` format 128-bit floating point constants.

40.4. VAX Opcodes

All DEC mnemonics are supported. Beware that `case...` instructions have exactly 3 operands. The dispatch table that follows the `case...` instruction should be made with `.word` statements. This is compatible with all unix assemblers we know of.

40.5. VAX Branch Improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that reaches the target. Generally these mnemonics are made by substituting `j` for `b` at the start of a DEC mnemonic. This feature is included both for compatibility and to help compilers. If you do not need this feature, avoid these opcodes. Here are the mnemonics, and the code they can expand into.

`jbsb`

`Jsb` is already an instruction mnemonic, so we chose `jbsb`.

(byte displacement)

`bsbb ...`

(word displacement)

`bsbw ...`

(long displacement)

`jsb ...`

`jbr`
`jr`

Unconditional branch.

(byte displacement)

`brb ...`

(word displacement)

`brw ...`

(long displacement)

`jmp ...`

`jCOND`

`COND` may be any one of the conditional branches `neq`, `nequ`, `eql`, `eqlu`, `gtr`, `geq`, `lss`, `gtru`, `lequ`, `vc`, `vs`, `gequ`, `cc`, `lssu`, `cs`. `COND` may also be one of the bit tests `bs`, `bc`, `bss`, `bcs`, `bsc`, `bcc`, `bssi`, `bcci`, `lbs`, `lbc`. `NOTCOND` is the opposite condition to `COND`.

(byte displacement)

`bCOND ...`

(word displacement)

`bNOTCOND foo ; brw ... ; foo:`

(long displacement)

`bNOTCOND foo ; jmp ... ; foo:`

`jacbX`

`X` may be one of `b d f g h l w`.

(word displacement)

`OPCODE ...`

(long displacement)

```
    OPCODE ..., foo ;
brb bar ;
foo: jmp ... ;
bar:
```

`jaobYYY`

`YYY` may be one of `lss leq`.

`jsobZZZ`

`ZZZ` may be one of `geq gtr`.

(byte displacement)

`OPCODE ...`

(word displacement)

```
    OPCODE ..., foo ;
brb bar ;
foo: brw destination ;
bar:
```

(long displacement)

```
    OPCODE ..., foo ;
brb bar ;
foo: jmp destination ;
bar:
```

```
aobleq
aoblss
sobgeq
sobgtr
```

(byte displacement)

```
OPCODE ...
```

(word displacement)

```
    OPCODE ..., foo ;
brb bar ;
foo: brw destination ;
bar:
```

(long displacement)

```
    OPCODE ..., foo ;
brb bar ;
foo: jmp destination ;
bar:
```

40.6. VAX Operands

The immediate character is `$` for Unix compatibility, not `#` as DEC writes it.

The indirect character is `*` for Unix compatibility, not `@` as DEC writes it.

The displacement sizing character is ` (an accent grave) for Unix compatibility, not `^` as DEC writes it. The letter preceding ` may have either case. `G` is not understood, but all other letters (`b i l s w`) are understood.

Register names understood are `r0 r1 r2 ... r15 ap fp sp pc`. Upper and lower case letters are equivalent.

For instance

```
tstb *w`$4(r5)
```

Any expression is permitted in an operand. Operands are comma separated.

40.7. Not Supported on VAX

Vax bit fields can not be assembled with `as`. Someone can add the required code if they really need it.

Chapter 41. v850 Dependent Features

41.1. Options

`as` supports the following additional command-line options for the V850 processor family:

`-wsigned_overflow`

Causes warnings to be produced when signed immediate values overflow the space available for then within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

`-wunsigned_overflow`

Causes warnings to be produced when unsigned immediate values overflow the space available for then within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

`-mv850`

Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

`-mv850e`

Specifies that the assembled code should be marked as being targeted at the V850E processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

`-mv850any`

Specifies that the assembled code should be marked as being targeted at the V850 processor but support instructions that are specific to the extended variants of the process. This allows the production of binaries that contain target specific code, but which are also intended to be used in a generic fashion. For example libgcc.a contains generic routines used by the code produced by GCC for all versions of the v850 architecture, together with support routines only used by the V850E architecture.

`-mrelax`

Enables relaxation. This allows the .longcall and .longjump pseudo ops to be used in the assembler source code. These ops label sections of code which are either a long function call or a long branch. The assembler will then flag these sections of code and the linker will attempt to relax them.

41.2. Syntax

41.2.1. Special Characters

`#` is the line comment character.

41.2.2. Register Names

`as` supports the following names for registers:

`general register 0`
: r0, zero

`general register 1`
: r1

`general register 2`
: r2, hp

`general register 3`
: r3, sp

`general register 4`
: r4, gp

`general register 5`
: r5, tp

`general register 6`
: r6

`general register 7`
: r7

`general register 8`
: r8

`general register 9`
: r9

`general register 10`
: r10

`general register 11`
: r11

`general register 12`
: r12

`general register 13`
: r13

`general register 14`
: r14

`general register 15`
: r15

`general register 16`
: r16

`general register 17`
: r17

`general register 18`
: r18

`general register 19`
: r19

`general register 20`
: r20

`general register 21`
: r21

`general register 22`
: r22

`general register 23`
: r23

`general register 24`
: r24

`general register 25`
: r25

`general register 26`
: r26

`general register 27`
: r27

`general register 28`
: r28

`general register 29`
: r29

`general register 30`
: r30, ep

`general register 31`
: r31, lp

`system register 0`
: eipc

`system register 1`
: eipsw

`system register 2`
: fepc

`system register 3`
: fepsw

`system register 4`
: ecr

`system register 5`
: psw

`system register 16`
: ctpc

`system register 17`
: ctpsw

`system register 18`
: dbpc

`system register 19`
: dbpsw

`system register 20`
: ctbp

41.3. Floating Point

The V850 family uses ieee floating-point numbers.

41.4. V850 Machine Directives

`.offset <expression>`

Moves the offset into the current section to the specified amount.

`.section "name", <type>`

This is an extension to the standard .section directive. It sets the current section to be <type> and creates an alias for this section called "name".

`.v850`

Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

`.v850e`

Specifies that the assembled code should be marked as being targeted at the V850E processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

41.5. Opcodes

`as` implements all the standard V850 opcodes.

`as` also implements the following pseudo ops:

`hi0()`

Computes the higher 16 bits of the given expression and stores it into the immediate operand field of the given instruction. For example:

```
mulhi hi0(here - there), r5, r6
```

computes the difference between the address of labels 'here' and 'there', takes the upper 16 bits of this difference, shifts it down 16 bits and then mutliplies it by the lower 16 bits in register 5, putting the result into register 6.

`lo()`

Computes the lower 16 bits of the given expression and stores it into the immediate operand field of the given instruction. For example:

```
addi lo(here - there), r5, r6
```

computes the difference between the address of labels 'here' and 'there', takes the lower 16 bits of this difference and adds it to register 5, putting the result into register 6.

`hi()`

Computes the higher 16 bits of the given expression and then adds the value of the most significant bit of the lower 16 bits of the expression and stores the result into the immediate operand field of the given instruction. For example the following code can be used to compute the address of the label 'here' and store it into register 6:

```
movhi hi(here), r0, r6 movea lo(here), r6, r6
```

The reason for this special behaviour is that movea performs a sign extension on its immediate operand. So for example if the address of 'here' was 0xFFFFFFFF then without the special behaviour of the hi() pseudo-op the movhi instruction would put 0xFFFF0000 into r6, then the movea instruction would takes its immediate operand, 0xFFFF, sign extend it to 32 bits, 0xFFFFFFFF, and then add it into r6 giving 0xFFFEFFFF which is wrong (the fifth nibble is E). With the hi() pseudo op adding in the top bit of the lo() pseudo op, the movhi instruction actually stores 0 into r6 (0xFFFF + 1 = 0x0000), so that the movea instruction stores 0xFFFFFFFF into r6 - the right value.

`hilo()`

Computes the 32 bit value of the given expression and stores it into the immediate operand field of the given instruction (which must be a mov instruction). For example:

```
mov hilo(here), r6
```

computes the absolute address of label 'here' and puts the result into register 6.

`sdaoff()`

Computes the offset of the named variable from the start of the Small Data Area (whoes address is held in register 4, the GP register) and stores the result as a 16 bit signed value in the immediate operand field of the given instruction. For example:

```
ld.w sdaoff(_a_variable)[gp],r6
```

loads the contents of the location pointed to by the label '_a_variable' into register 6, provided that the label is located somewhere within +/- 32K of the address held in the GP register. [Note the linker assumes that the GP register contains a fixed address set to the address of the label called '__gp'. This can either be set up automatically by the linker, or specifically set by using the `-defsym __gp=<value>` command line option].

`tdaoff()`

Computes the offset of the named variable from the start of the Tiny Data Area (whoes address is held in register 30, the EP register) and stores the result as a 4,5, 7 or 8 bit unsigned value in the immediate operand field of the given instruction. For example:

```
sld.w tdaoff(_a_variable)[ep],r6
```

loads the contents of the location pointed to by the label '_a_variable' into register 6, provided that the label is located somewhere within +256 bytes of the address held in the EP register. [Note the linker assumes that the EP register contains a fixed address set to the address of the label called '__ep'. This can either be set up automatically by the linker, or specifically set by using the `-defsym __ep=<value>` command line option].

`zdaoff()`

Computes the offset of the named variable from address 0 and stores the result as a 16 bit signed value in the immediate operand field of the given instruction. For example:

```
movea zdaoff(_a_variable),zero,r6
```

puts the address of the label '_a_variable' into register 6, assuming that the label is somewhere within the first 32K of memory. (Strictly speaking it also possible to access the last 32K of memory as well, as the offsets are signed).

`ctoff()`

Computes the offset of the named variable from the start of the Call Table Area (whoes address is helg in system register 20, the CTBP register) and stores the result a 6 or 16 bit unsigned value in the immediate field of then given instruction or piece of data. For example:

```
callt ctoff(table_func1)
```

will put the call the function whoes address is held in the call table at the location labeled 'table_func1'.

`.longcall name`

Indicates that the following sequence of instructions is a long call to function `name`. The linker will attempt to shorten this call sequence if `name` is within a 22bit offset of the call. Only valid if the `-mrelax` command line switch has been enabled.

`.longjump name`

Indicates that the following sequence of instructions is a long jump to label `name`. The linker will attempt to shorten this code sequence if `name` is within a 22bit offset of the jump. Only valid if the `-mrelax` command line switch has been enabled.

For information on the V850 instruction set, see [V850 Family 32-/16-Bit single-Chip Microcontroller Architecture Manual] from NEC. Ltd.

Chapter 42. Xtensa Dependent Features

This chapter covers features of the gnu assembler that are specific to the Xtensa architecture. For details about the Xtensa instruction set, please consult the [Xtensa Instruction Set Architecture (ISA) Reference Manual].

42.1. Command Line Options

The Xtensa version of the gnu assembler supports these special options:

`-density | -no-density`

Enable or disable use of the Xtensa code density option (16-bit instructions). Section 42.3.1 *Using Density Instructions*. If the processor is configured with the density option, this is enabled by default; otherwise, it is always disabled.

`-relax | -no-relax`

Enable or disable relaxation of instructions with immediate operands that are outside the legal range for the instructions. Section 42.4 *Xtensa Relaxation*. The default is `-relax` and this default should almost always be used. If relaxation is disabled with `-no-relax`, instruction operands that are out of range will cause errors. Note: In the current implementation, these options also control whether assembler optimizations are performed, making these options equivalent to `-generics` and `-no-generics`.

`-generics | -no-generics`

Enable or disable all assembler transformations of Xtensa instructions, including both relaxation and optimization. The default is `-generics`; `-no-generics` should only be used in the rare cases when the instructions must be exactly as specified in the assembly source. As with `-no-relax`, using `-no-generics` causes out of range instruction operands to be errors.

`-text-section-literals | -no-text-section-literals`

Control the treatment of literal pools. The default is `-no-text-section-literals`, which places literals in a separate section in the output file. This allows the literal pool to be placed in a data RAM/ROM, and it also allows the linker to combine literal pools from separate object files to remove redundant literals and improve code size. With `-text-section-literals`, the literals are interspersed in the text section in order to keep them as close as possible to their references. This may be necessary for large assembly files.

`-target-align | -no-target-align`

Enable or disable automatic alignment to reduce branch penalties at some expense in code size. Section 42.3.2 *Automatic Instruction Alignment*. This optimization is enabled by default. Note that the assembler will always align instructions like `LOOP` that have fixed alignment requirements.

`-longcalls | -no-longcalls`

Enable or disable transformation of call instructions to allow calls across a greater range of addresses. Section 42.4.2 *Function Call Relaxation*. This option should be used when call targets can potentially be out of range, but it degrades both code size and performance. The default is `-no-longcalls`.

42.2. Assembler Syntax

Block comments are delimited by `/*` and `*/`. End of line comments may be introduced with either `#` or `//`.

Instructions consist of a leading opcode or macro name followed by whitespace and an optional comma-separated list of operands:

```
opcode [operand, ...]
```

Instructions must be separated by a newline or semicolon.

42.2.1. Opcode Names

See the [Xtensa Instruction Set Architecture (ISA) Reference Manual] for a complete list of opcodes and descriptions of their semantics.

The Xtensa assembler distinguishes between *generic* and *specific* opcodes. Specific opcodes correspond directly to Xtensa machine instructions. Prefixing an opcode with an underscore character (_) identifies it as a specific opcode. Opcodes without a leading underscore are generic, which means the assembler is required to preserve their semantics but may not translate them directly to the specific opcodes with the same names. Instead, the assembler may optimize a generic opcode and select a better instruction to use in its place (refer to Section 42.3 *Xtensa Optimizations*), or the assembler may relax the instruction to handle operands that are out of range for the corresponding specific opcode (refer to Section 42.4 *Xtensa Relaxation*).

Only use specific opcodes when it is essential to select the exact machine instructions produced by the assembler. Using specific opcodes unnecessarily only makes the code less efficient, by disabling assembler optimization, and less flexible, by disabling relaxation.

Note that this special handling of underscore prefixes only applies to Xtensa opcodes, not to either built-in macros or user-defined macros. When an underscore prefix is used with a macro (e.g., `_NOP`), it refers to a different macro. The assembler generally provides built-in macros both with and without the underscore prefix, where the underscore versions behave as if the underscore carries through to the instructions in the macros. For example, `_NOP` expands to `_OR a1,a1,a1`.

The underscore prefix only applies to individual instructions, not to series of instructions. For example, if a series of instructions have underscore prefixes, the assembler will not transform the individual instructions, but it may insert other instructions between them (e.g., to align a `LOOP` instruction). To prevent the assembler from modifying a series of instructions as a whole, use the `no-generics` directive. Section 42.5.4 *generics*.

42.2.2. Register Names

An initial `$` character is optional in all register names. General purpose registers are named `a0`...`a15`. Additional registers may be added by processor configuration options. In particular, the mac16 option adds a mr register bank. Its registers are named `m0`...`m3`.

As a special feature, `sp` is also supported as a synonym for `a1`.

42.3. Xtensa Optimizations

The optimizations currently supported by `as` are generation of density instructions where appropriate and automatic branch target alignment.

42.3.1. Using Density Instructions

The Xtensa instruction set has a code density option that provides 16-bit versions of some of the most commonly used opcodes. Use of these opcodes can significantly reduce code size. When possible, the assembler automatically translates generic instructions from the core Xtensa instruction set into equivalent instructions from the Xtensa code density option. This translation can be disabled by using specific opcodes (refer to Section 42.2.1 *Opcode Names*), by using the `-no-density` command-line option (refer to Section 42.1 *Command Line Options*), or by using the `no-density` directive (refer to Section 42.5.1 *density*).

It is a good idea *not* to use the density instuctions directly. The assembler will automatically select dense instructions where possible. If you later need to avoid using the code density option, you can disable it in the assembler without having to modify the code.

42.3.2. Automatic Instruction Alignment

The Xtensa assembler will automatically align certain instructions, both to optimize performance and to satisfy architectural requirements.

When the `-target-align` command-line option is enabled (refer to Section 42.1 *Command Line Options*), the assembler attempts to widen density instructions preceding a branch target so that the target instruction does not cross a 4-byte boundary. Similarly, the assembler also attempts to align each instruction following a call instruction. If there are not enough preceding safe density instructions to align a target, no widening will be performed. This alignment has the potential to reduce branch penalties at some expense in code size. The assembler will not attempt to align labels with the prefixes `.Ln` and `.LM`, since these labels are used for debugging information and are not typically branch targets.

The `LOOP` family of instructions must be aligned on either a 1 or 2 mod 4 byte boundary. The assembler knows about this restriction and inserts the minimal number of 2 or 3 byte no-op instructions to satisfy it. When no-op instructions are added, any label immediately preceding the original loop will be moved in order to refer to the loop instruction, not the newly generated no-op instruction.

Similarly, the `ENTRY` instruction must be aligned on a 0 mod 4 byte boundary. The assembler satisfies this requirement by inserting zero bytes when required. In addition, labels immediately preceding the `ENTRY` instruction will be moved to the newly aligned instruction location.

42.4. Xtensa Relaxation

When an instruction operand is outside the range allowed for that particular instruction field, `as` can transform the code to use a functionally-equivalent instruction or sequence of instructions. This process is known as *relaxation*. This is typically done for branch instructions because the distance of the branch targets is not known until assembly-time. The Xtensa assembler offers branch relaxation and also extends this concept to function calls, `MOVI` instructions and other instructions with immediate fields.

42.4.1. Conditional Branch Relaxation

When the target of a branch is too far away from the branch itself, i.e., when the offset from the branch to the target is too large to fit in the immediate field of the branch instruction, it may be necessary to replace the branch with a branch around a jump. For example,

```
beqz    a2, L
```

may result in:

```
    bnez.n  a2, M
    j L
M:
```

(The BNEZ.N instruction would be used in this example only if the density option is available. Otherwise, BNEZ would be used.)

42.4.2. Function Call Relaxation

Function calls may require relaxation because the Xtensa immediate call instructions (CALL0, CALL4, CALL8 and CALL12) provide a PC-relative offset of only 512 Kbytes in either direction. For larger programs, it may be necessary to use indirect calls (CALLX0, CALLX4, CALLX8 and CALLX12) where the target address is specified in a register. The Xtensa assembler can automatically relax immediate call instructions into indirect call instructions. This relaxation is done by loading the address of the called function into the callee's return address register and then using a CALLX instruction. So, for example:

```
call8 func
```

might be relaxed to:

```
    .literal .L1, func
    l32r    a8, .L1
    callx8  a8
```

Because the addresses of targets of function calls are not generally known until link-time, the assembler must assume the worst and relax all the calls to functions in other source files, not just those that really will be out of range. The linker can recognize calls that were unnecessarily relaxed, but it can only partially remove the overhead introduced by the assembler.

Call relaxation has a negative effect on both code size and performance, so this relaxation is disabled by default. If a program is too large and some of the calls are out of range, function call relaxation can be enabled using the --longcalls command-line option or the longcalls directive (refer to Section 42.5.3 *longcalls*).

42.4.3. Other Immediate Field Relaxation

The MOVI machine instruction can only materialize values in the range from -2048 to 2047. Values outside this range are best materalized with L32R instructions. Thus:

```
movi a0, 100000
```

is assembled into the following machine code:

```
    .literal .L1, 100000
    l32r a0, .L1
```

The L8UI machine instruction can only be used with immediate offsets in the range from 0 to 255. The L16SI and L16UI machine instructions can only be used with offsets from 0 to 510. The L32I machine instruction can only be used with offsets from 0 to 1020. A load offset outside these ranges can be materalized with an L32R instruction if the destination register of the load is different than the source address register. For example:

```
    l32i a1, a0, 2040
```

is translated to:

```
.literal .L1, 2040
l32r a1, .L1
addi a1, a0, a1
l32i a1, a1, 0
```

If the load destination and source address register are the same, an out-of-range offset causes an error.

The Xtensa `ADDI` instruction only allows immediate operands in the range from -128 to 127. There are a number of alternate instruction sequences for the generic `ADDI` operation. First, if the immediate is 0, the `ADDI` will be turned into a `MOV.N` instruction (or the equivalent `OR` instruction if the code density option is not available). If the `ADDI` immediate is outside of the range -128 to 127, but inside the range -32896 to 32639, an `ADDMI` instruction or `ADDMI`/`ADDI` sequence will be used. Finally, if the immediate is outside of this range and a free register is available, an `L32R`/`ADD` sequence will be used with a literal allocated from the literal pool.

For example:

```
addi    a5, a6, 0
addi    a5, a6, 512
addi    a5, a6, 513
addi    a5, a6, 50000
```

is assembled into the following:

```
.literal .L1, 50000
mov.n   a5, a6
addmi   a5, a6, 0x200
addmi   a5, a6, 0x200
addi    a5, a5, 1
l32r    a5, .L1
add     a5, a6, a5
```

42.5. Directives

The Xtensa assember supports a region-based directive syntax:

```
.begin directive [options]
    ...
    .end directive
```

All the Xtensa-specific directives that apply to a region of code use this syntax.

The directive applies to code between the `.begin` and the `.end`. The state of the option after the `.end` reverts to what it was before the `.begin`. A nested `.begin`/`.end` region can further change the state of the directive without having to be aware of its outer state. For example, consider:

```
    .begin no-density
L:  add a0, a1, a2
    .begin density
M:  add a0, a1, a2
    .end density
N:  add a0, a1, a2
    .end no-density
```

The generic `ADD` opcodes at `L` and `N` in the outer `no-density` region both result in `ADD` machine instructions, but the assembler selects an `ADD.N` instruction for the generic `ADD` at `M` in the inner `density` region.

The advantage of this style is that it works well inside macros which can preserve the context of their callers.

When command-line options and assembler directives are used at the same time and conflict, the one that overrides a default behavior takes precedence over one that is the same as the default. For example, if the code density option is available, the default is to select density instructions whenever possible. So, if the above is assembled with the `-no-density` flag, which overrides the default, all the generic `ADD` instructions result in `ADD` machine instructions. If assembled with the `-density` flag, which is already the default, the `no-density` directive takes precedence and only one of the generic `ADD` instructions is optimized to be a `ADD.N` machine instruction. An underscore prefix identifying a specific opcode always takes precedence over directives and command-line flags.

The following directives are available:

42.5.1. density

The `density` and `no-density` directives enable or disable optimization of generic instructions into density instructions within the region. Section 42.3.1 *Using Density Instructions*.

```
.begin [no-]density
.end [no-]density
```

This optimization is enabled by default unless the Xtensa configuration does not support the code density option or the `-no-density` command-line option was specified.

42.5.2. relax

The `relax` directive enables or disables relaxation within the region. Section 42.4 *Xtensa Relaxation*. Note: In the current implementation, these directives also control whether assembler optimizations are performed, making them equivalent to the `generics` and `no-generics` directives.

```
.begin [no-]relax
.end [no-]relax
```

Relaxation is enabled by default unless the `-no-relax` command-line option was specified.

42.5.3. longcalls

The `longcalls` directive enables or disables function call relaxation. Section 42.4.2 *Function Call Relaxation*.

```
.begin [no-]longcalls
.end [no-]longcalls
```

Call relaxation is disabled by default unless the `-longcalls` command-line option is specified.

42.5.4. generics

This directive enables or disables all assembler transformation, including relaxation (refer to Section 42.4 *Xtensa Relaxation*) and optimization (refer to Section 42.3 *Xtensa Optimizations*).

```
.begin [no-]generics
.end [no-]generics
```

Disabling generics is roughly equivalent to adding an underscore prefix to every opcode within the region, so that every opcode is treated as a specific opcode. Section 42.2.1 *Opcode Names*. In the current implementation of `as`, built-in macros are also disabled within a `no-generics` region.

42.5.5. literal

The `.literal` directive is used to define literal pool data, i.e., read-only 32-bit data accessed via `L32R` instructions.

```
.literal label, value[, value...]
```

This directive is similar to the standard `.word` directive, except that the actual location of the literal data is determined by the assembler and linker, not by the position of the `.literal` directive. Using this directive gives the assembler freedom to locate the literal data in the most appropriate place and possibly to combine identical literals. For example, the code:

```
    entry sp, 40
    .literal .L1, sym
    l32r    a4, .L1
```

can be used to load a pointer to the symbol `sym` into register `a4`. The value of `sym` will not be placed between the `ENTRY` and `L32R` instructions; instead, the assembler puts the data in a literal pool.

By default literal pools are placed in a separate section; however, when using the `-text-section-literals` option (refer to Section 42.1 *Command Line Options*), the literal pools are placed in the current section. These text section literal pools are created automatically before `ENTRY` instructions and manually after `.literal_position` directives (refer to Section 42.5.6 *literal_position*). If there are no preceding `ENTRY` instructions or `.literal_position` directives, the assembler will print a warning and place the literal pool at the beginning of the current section. In such cases, explicit `.literal_position` directives should be used to place the literal pools.

42.5.6. literal_position

When using `-text-section-literals` to place literals inline in the section being assembled, the `.literal_position` directive can be used to mark a potential location for a literal pool.

```
.literal_position
```

The `.literal_position` directive is ignored when the `-text-section-literals` option is not used.

The assembler will automatically place text section literal pools before `ENTRY` instructions, so the `.literal_position` directive is only needed to specify some other location for a literal pool. You may need to add an explicit jump instruction to skip over an inline literal pool.

For example, an interrupt vector does not begin with an `ENTRY` instruction so the assembler will be unable to automatically find a good place to put a literal pool. Moreover, the code for the interrupt vector must be at a specific starting address, so the literal pool cannot come before the start of the code. The literal pool for the vector must be explicitly positioned in the middle of the vector (before any uses of the literals, of course). The `.literal_position` directive can be used to do this. In the following code, the literal for `M` will automatically be aligned correctly and is placed after the unconditional jump.

```
    .global M
code_start:
    j continue
    .literal_position
```

```
    .align 4
continue:
    movi    a4, M
```

42.5.7. literal_prefix

The `literal_prefix` directive allows you to specify different sections to hold literals from different portions of an assembly file. With this directive, a single assembly file can be used to generate code into multiple sections, including literals generated by the assembler.

```
    .begin literal_prefix [name]
    .end literal_prefix
```

For the code inside the delimited region, the assembler puts literals in the section `name.literal`. If this section does not yet exist, the assembler creates it. The `name` parameter is optional. If `name` is not specified, the literal prefix is set to the "default" for the file. This default is usually `.literal` but can be changed with the `-rename-section` command-line argument.

42.5.8. freeregs

This directive tells the assembler that the given registers are unused in the region.

```
    .begin freeregs ri[,ri...]
    .end freeregs
```

This allows the assembler to use these registers for relaxations or optimizations. (They are actually only for relaxations at present, but the possibility of optimizations exists in the future.)

Nested `freeregs` directives can be used to add additional registers to the list of those available to the assembler. For example:

```
    .begin freeregs a3, a4
    .begin freeregs a5
```

has the effect of declaring `a3`, `a4`, and `a5` all free.

42.5.9. frame

This directive tells the assembler to emit information to allow the debugger to locate a function's stack frame. The syntax is:

```
   .frame reg, size
```

where `reg` is the register used to hold the frame pointer (usually the same as the stack pointer) and `size` is the size in bytes of the stack frame. The `.frame` directive is typically placed immediately after the `ENTRY` instruction for a function.

In almost all circumstances, this information just duplicates the information given in the function's `ENTRY` instruction; however, there are two cases where this is not true:

1. The size of the stack frame is too big to fit in the immediate field of the `ENTRY` instruction.
2. The frame pointer is different than the stack pointer, as with functions that call `alloca`.

Chapter 43. Reporting Bugs

Your bug reports play an essential role in making `as` reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of `as` work better. Bug reports are your contribution to the maintenance of `as`.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

43.1. Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the assembler gets a fatal signal, for any input whatever, that is a `as` bug. Reliable assemblers never crash.
- If `as` produces an error message for valid input, that is a bug.
- If `as` does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be our idea of "an extension" or "support for traditional practice".
- If you are an experienced user of assemblers, your suggestions for improvement of `as` are welcome in any case.

43.2. How to Report Bugs

A number of companies and individuals offer support for gnu products. If you obtained `as` from a support organization, we recommend you contact that organization first.

You can find contact information for many support companies and individuals in the file `etc/SERVICE` in the gnu Emacs distribution.

In any event, we also recommend that you send bug reports for `as` to `bug-binutils@gnu.org`.

The fundamental principle of reporting bugs usefully is this: *report all the facts*. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of a symbol you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the assembler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug if it is new to us. Therefore, always write your bug reports on the assumption that the bug has not been reported previously.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with.

To enable us to fix the bug, you should include all these things:

- The version of `as`. `as` announces it if you start it with the `-version` argument.

  Without this, we will not know whether there is any point in looking for the bug in the current version of `as`.

- Any patches you may have applied to the `as` source.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile `as`--e.g. "`gcc-2.7`".
- The command arguments you gave the assembler to assemble your example and observe the bug. To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient.

  If we were to try to guess the arguments, we would probably guess wrong and then we might not encounter the bug.

- A complete input file that will reproduce the bug. If the bug is observed when the assembler is invoked via a compiler, send the assembler source, not the high level language source. Most compilers will produce the assembler source when run with the `-S` option. If you are using `gcc`, use the options `-v -save-temps`; this will save the assembler source in a file with an extension of `.s`, and also show you exactly how `as` is being run.
- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal."

  Of course, if the bug is that `as` gets a fatal signal, then we will certainly notice it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong. You might as well not give us a chance to make a mistake.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of `as` is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.

- If you wish to suggest changes to the `as` source, send us context diffs, as generated by `diff` with the `-u`, `-c`, or `-p` option. Always send diffs from the old file to the new file. If you even discuss something in the `as` source, refer to it by context, not by line number.

  The line numbers in our development sources will not match those in your sources. Your line numbers would convey no useful information to us.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

  However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

Sometimes with a program as complicated as `as` it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

Chapter 44. Acknowledgements

If you have contributed to `as` and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send mail to the maintainer, and we'll correct the situation. Currently the maintainer is Ken Raeburn (email address `raeburn@cygnus.com`).

Dean Elsner wrote the original gnu assembler for the VAX.[1]

Jay Fenlason maintained GAS for a while, adding support for GDB-specific debug information and the 68k series machines, most of the preprocessing pass, and extensive changes in `messages.c`, `input-file.c`, `write.c`.

K. Richard Pixley maintained GAS for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking GAS up to handle multiple object file format back ends (including heavy rewrite, testing, an integration of the coff and b.out back ends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted GAS to strictly ANSI C including full prototypes, added support for m680[34]0 and cpu32, did considerable work on i960 including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, rs6000, and hp300hpux host ports, updated "know" assertions and made them work, much other reorganization, cleanup, and lint.

Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.

The original VMS support was contributed by David L. Kashtan. Eric Youngdale has done much work with it since.

The Intel 80386 machine description was written by Eliot Dresselhaus.

Minh Tran-Le at IntelliCorp contributed some AIX 386 support.

The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish Institute of Computer Science.

Keith Knowles at the Open Software Foundation wrote the original MIPS back end (`tc-mips.c`, `tc-mips.h`), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support a.out format.

Support for the Zilog Z8k and Renesas H8/300 and H8/500 processors (tc-z8k, tc-h8300, tc-h8500), and IEEE 695 object file format (obj-ieee), was written by Steve Chamberlain of Cygnus Support. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.

John Gilmore built the AMD 29000 support, added `.include` support, and simplified the configuration of which versions accept which directives. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g., `jsr`), while synthetic instructions remained shrinkable (`jbsr`). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the proverbial one-bit fix.

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

Steve Chamberlain made `as` able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

1. Any more details?

Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah and Cygnus Support.

Support for ELF format files has been worked on by Mark Eichin of Cygnus Support (original, incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner of the Open Software Foundation (i386 mainly), and Ken Raeburn of Cygnus Support (sparc, and some initial 64-bit support).

Linas Vepstas added GAS support for the ESA/390 "IBM 370" architecture.

Richard Henderson rewrote the Alpha assembler. Klaus Kaempf wrote GAS and BFD support for openVMS/Alpha.

Timothy Wall, Michael Hayes, and Greg Smart contributed to the various tic* flavors.

David Heine, Sterling Augustine, Bob Wilson and John Ruttenberg from Tensilica, Inc. added support for Xtensa processors.

Several engineers at Cygnus Support have also provided many small bug fixes and configuration enhancements.

Many others have contributed large or small bugfixes and enhancements. If you have contributed significant work and are not mentioned on this list, and want to be, let us know. Some of the history has been lost; we are not intentionally leaving anyone out.

Appendix A.

GNU Free Documentation License

Version 1.1, March 2000

1. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

   A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five). C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version. N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section. If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties-for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

7. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

9. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

10. TERMINATION

    You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

11. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.1. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled "GNU
Free Documentation License."
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being `list`"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

A

B

C

D

E

F

G

H

J

L

M

N

O

R

T

U

V

W

X

Z