Assembler Internals

This chapter describes the internals of the assembler. It is incomplete, but it may help a bit.

This chapter is not updated regularly, and it may be out of date.

GAS versions

GAS has acquired layers of code over time. The original GAS only supported the a.out object file format, with three sections. Support for multiple sections has been added in two different ways.

The preferred approach is to use the version of GAS created when the symbol `BFD_ASSEMBLER` is defined. The other versions of GAS are documented for historical purposes, and to help anybody who has to debug code written for them.

The type `segT` is used to represent a section in code which must work with all versions of GAS.

Original GAS

The original GAS only supported the a.out object file format with three sections: '`.text`', '`.data`', and '`.bss`'. This is the version of GAS that is compiled if neither `BFD_ASSEMBLER` nor `MANY_SEGMENTS` is defined. This version of GAS is still used for the m68k-aout target, and perhaps others.

This version of GAS should not be used for any new development.

There is still code that is specific to this version of GAS, notably in '`write.c`'. There is no way for this code to loop through all the sections; it simply looks at global variables like `text_frag_root` and `data_frag_root`.

The type `segT` is an enum.

MANY_SEGMENTS gas version

The `MANY_SEGMENTS` version of gas is only used for COFF. It uses the BFD library, but it writes out all the data itself using `bfd_write`. This version of gas supports up to 40 normal sections. The section names are stored in the `seg_name` array. Other information is stored in the `segment_info` array.

The type `segT` is an enum. Code that wants to examine all the sections can use a `segT` variable as loop index from `SEG_E0` up to but not including `SEG_UNKNOWN`.

Most of the code specific to this version of GAS is in the file '`config/obj-coff.c`', in the portion of that file that is compiled when `BFD_ASSEMBLER` is not defined.

This version of GAS is still used for several COFF targets.

BFD_ASSEMBLER gas version

The preferred version of GAS is the `BFD_ASSEMBLER` version. In this version of GAS, the output file is a normal BFD, and the BFD routines are used to generate the output.

`BFD_ASSEMBLER` will automatically be used for certain targets, including those that use the ELF, ECOFF, and SOM object file formats, and also all Alpha, MIPS, PowerPC, and SPARC targets. You can force the use of `BFD_ASSEMBLER` for other targets with the configure option '`--enable-bfd-assembler`'; however, it has not been tested for many targets, and can not be assumed to work.

Data types

This section describes some fundamental GAS data types.

Symbols

The definition for the symbol structure, `symbolS`, is located in '`struc-symbol.h`'.

In general, the fields of this structure may not be referred to directly. Instead, you must use one of the accessor functions defined in '`symbol.h`'. These accessor functions should work for any GAS version.

Symbol structures contain the following fields:

`sy_value`
: This is an `expressionS` that describes the value of the symbol. It might refer to one or more other symbols; if so, its true value may not be known until `resolve_symbol_value` is called with *finalize_syms* non-zero in `write_object_file`.

  The expression is often simply a constant. Before `resolve_symbol_value` is called with *finalize_syms* set, the value is the offset from the frag (see [Frags], page 9). Afterward, the frag address has been added in.

`sy_resolved`
: This field is non-zero if the symbol's value has been completely resolved. It is used during the final pass over the symbol table.

`sy_resolving`
: This field is used to detect loops while resolving the symbol's value.

`sy_used_in_reloc`
: This field is non-zero if the symbol is used by a relocation entry. If a local symbol is used in a relocation entry, it must be possible to redirect those relocations to other symbols, or this symbol cannot be removed from the final symbol list.

`sy_next`
`sy_previous`
: These pointers to other `symbolS` structures describe a singly or doubly linked list. (If `SYMBOLS_NEED_BACKPOINTERS` is not defined, the `sy_previous` field will be omitted; `SYMBOLS_NEED_BACKPOINTERS` is always defined if `BFD_ASSEMBLER`.) These fields should be accessed with the `symbol_next` and `symbol_previous` macros.

`sy_frag`
: This points to the frag (see [Frags], page 9) that this symbol is attached to.

`sy_used`
: Whether the symbol is used as an operand or in an expression. Note: Not all of the backends keep this information accurate; backends which use this bit are responsible for setting it when a symbol is used in backend routines.

`sy_mri_common`
: Whether the symbol is an MRI common symbol created by the `COMMON` pseudo-op when assembling in MRI mode.

`bsym`
: If `BFD_ASSEMBLER` is defined, this points to the BFD `asymbol` that will be used in writing the object file.

`sy_name_offset`
: (Only used if `BFD_ASSEMBLER` is not defined.) This is the position of the symbol's name in the string table of the object file. On some formats, this will start at position 4, with position 0 reserved for unnamed symbols. This field is not used until `write_object_file` is called.

`sy_symbol`
: (Only used if `BFD_ASSEMBLER` is not defined.) This is the format-specific symbol structure, as it would be written into the object file.

`sy_number`
: (Only used if `BFD_ASSEMBLER` is not defined.) This is a 24-bit symbol number, for use in constructing relocation table entries.

`sy_obj`
: This format-specific data is of type `OBJ_SYMFIELD_TYPE`. If no macro by that name is defined in '`obj-format.h`', this field is not defined.

`sy_tc`
: This processor-specific data is of type `TC_SYMFIELD_TYPE`. If no macro by that name is defined in '`targ-cpu.h`', this field is not defined.

Here is a description of the accessor functions. These should be used rather than referring to the fields of `symbolS` directly.

`S_SET_VALUE`
: Set the symbol's value.

`S_GET_VALUE`
: Get the symbol's value. This will cause `resolve_symbol_value` to be called if necessary.

`S_SET_SEGMENT`
: Set the section of the symbol.

`S_GET_SEGMENT`
: Get the symbol's section.

`S_GET_NAME`
: Get the name of the symbol.

`S_SET_NAME`
: Set the name of the symbol.

`S_IS_EXTERNAL`
: Return non-zero if the symbol is externally visible.

`S_IS_EXTERN`
: A synonym for `S_IS_EXTERNAL`. Don't use it.

`S_IS_WEAK`
: Return non-zero if the symbol is weak.

`S_IS_COMMON`
: Return non-zero if this is a common symbol. Common symbols are sometimes represented as undefined symbols with a value, in which case this function will not be reliable.

`S_IS_DEFINED`
: Return non-zero if this symbol is defined. This function is not reliable when called on a common symbol.

`S_IS_DEBUG`
: Return non-zero if this is a debugging symbol.

`S_IS_LOCAL`
: Return non-zero if this is a local assembler symbol which should not be included in the final symbol table. Note that this is not the opposite of `S_IS_EXTERNAL`. The '`-L`' assembler option affects the return value of this function.

`S_SET_EXTERNAL`
: Mark the symbol as externally visible.

`S_CLEAR_EXTERNAL`
: Mark the symbol as not externally visible.

`S_SET_WEAK`
: Mark the symbol as weak.

`S_GET_TYPE`
`S_GET_DESC`
`S_GET_OTHER`
: Get the `type`, `desc`, and `other` fields of the symbol. These are only defined for object file formats for which they make sense (primarily a.out).

`S_SET_TYPE`
`S_SET_DESC`
`S_SET_OTHER`
: Set the `type`, `desc`, and `other` fields of the symbol. These are only defined for object file formats for which they make sense (primarily a.out).

`S_GET_SIZE`
: Get the size of a symbol. This is only defined for object file formats for which it makes sense (primarily ELF).

`S_SET_SIZE`
: Set the size of a symbol. This is only defined for object file formats for which it makes sense (primarily ELF).

`symbol_get_value_expression`
: Get a pointer to an `expressionS` structure which represents the value of the symbol as an expression.

`symbol_set_value_expression`
: Set the value of a symbol to an expression.

`symbol_set_frag`
: Set the frag where a symbol is defined.

`symbol_get_frag`
: Get the frag where a symbol is defined.

`symbol_mark_used`
: Mark a symbol as having been used in an expression.

`symbol_clear_used`
: Clear the mark indicating that a symbol was used in an expression.

`symbol_used_p`
: Return whether a symbol was used in an expression.

`symbol_mark_used_in_reloc`
: Mark a symbol as having been used by a relocation.

`symbol_clear_used_in_reloc`
: Clear the mark indicating that a symbol was used in a relocation.

`symbol_used_in_reloc_p`
: Return whether a symbol was used in a relocation.

`symbol_mark_mri_common`
: Mark a symbol as an MRI common symbol.

`symbol_clear_mri_common`
: Clear the mark indicating that a symbol is an MRI common symbol.

`symbol_mri_common_p`
: Return whether a symbol is an MRI common symbol.

`symbol_mark_written`
: Mark a symbol as having been written.

`symbol_clear_written`
: Clear the mark indicating that a symbol was written.

`symbol_written_p`
: Return whether a symbol was written.

`symbol_mark_resolved`
: Mark a symbol as having been resolved.

`symbol_resolved_p`
: Return whether a symbol has been resolved.

`symbol_section_p`
: Return whether a symbol is a section symbol.

`symbol_equated_p`
: Return whether a symbol is equated to another symbol.

`symbol_constant_p`
: Return whether a symbol has a constant value, including being an offset within some frag.

`symbol_get_bfdsym`
: Return the BFD symbol associated with a symbol.

`symbol_set_bfdsym`
: Set the BFD symbol associated with a symbol.

`symbol_get_obj`
: Return a pointer to the `OBJ_SYMFIELD_TYPE` field of a symbol.

`symbol_set_obj`
: Set the `OBJ_SYMFIELD_TYPE` field of a symbol.

`symbol_get_tc`
: Return a pointer to the `TC_SYMFIELD_TYPE` field of a symbol.

`symbol_set_tc`
: Set the `TC_SYMFIELD_TYPE` field of a symbol.

When `BFD_ASSEMBLER` is defined, GAS attempts to store local symbols–symbols which will not be written to the output file–using a different structure, `struct local_symbol`. This structure can only represent symbols whose value is an offset within a frag.

Code outside of the symbol handler will always deal with `symbolS` structures and use the accessor functions. The accessor functions correctly deal with local symbols. `struct local_symbol` is much smaller than `symbolS` (which also automatically creates a bfd `asymbol` structure), so this saves space when assembling large files.

The first field of `symbolS` is `bsym`, the pointer to the BFD symbol. The first field of `struct local_symbol` is a pointer which is always set to NULL. This is how the symbol accessor functions can distinguish local symbols from ordinary symbols. The symbol accessor functions automatically convert a local symbol into an ordinary symbol when necessary.

Expressions

Expressions are stored in an `expressionS` structure. The structure is defined in '`expr.h`'.

The macro `expression` will create an `expressionS` structure based on the text found at the global variable `input_line_pointer`.

A single `expressionS` structure can represent a single operation. Complex expressions are formed by creating *expression symbols* and combining them in `expressionS` structures. An expression symbol is created by calling `make_expr_symbol`. An expression symbol should naturally never appear in a symbol table, and the implementation of `S_IS_LOCAL` (see [Symbols], page 3) reflects that. The function `expr_symbol_where` returns non-zero if a symbol is an expression symbol, and also returns the file and line for the expression which caused it to be created.

The `expressionS` structure has two symbol fields, a number field, an operator field, and a field indicating whether the number is unsigned.

The operator field is of type `operatorT`, and describes how to interpret the other fields; see the definition in '`expr.h`' for the possibilities.

An `operatorT` value of `O_big` indicates either a floating point number, stored in the global variable `generic_floating_point_number`, or an integer too large to store in an `offsetT` type, stored in the global array `generic_bignum`. This rather inflexible approach makes it impossible to use floating point numbers or large expressions in complex expressions.

Fixups

A *fixup* is basically anything which can not be resolved in the first pass. Sometimes a fixup can be resolved by the end of the assembly; if not, the fixup becomes a relocation entry in the object file.

A fixup is created by a call to `fix_new` or `fix_new_exp`. Both take a frag (see [Frags], page 9), a position within the frag, a size, an indication of whether the fixup is PC relative, and a type. In a `BFD_ASSEMBLER` GAS, the type is nominally a `bfd_reloc_code_real_type`, but several targets use other type codes to represent fixups that can not be described as relocations.

The `fixS` structure has a number of fields, several of which are obsolete or are only used by a particular target. The important fields are:

`fx_frag` The frag (see [Frags], page 9) this fixup is in.

`fx_where` The location within the frag where the fixup occurs.

`fx_addsy` The symbol this fixup is against. Typically, the value of this symbol is added into the object contents. This may be NULL.

`fx_subsy` The value of this symbol is subtracted from the object contents. This is normally NULL.

`fx_offset`
A number which is added into the fixup.

`fx_addnumber`
Some CPU backends use this field to convey information between `md_apply_fix3` and `tc_gen_reloc`. The machine independent code does not use it.

`fx_next` The next fixup in the section.

`fx_r_type`
The type of the fixup. This field is only defined if `BFD_ASSEMBLER`, or if the target defines `NEED_FX_R_TYPE`.

`fx_size` The size of the fixup. This is mostly used for error checking.

`fx_pcrel` Whether the fixup is PC relative.

`fx_done` Non-zero if the fixup has been applied, and no relocation entry needs to be generated.

`fx_file`
`fx_line` The file and line where the fixup was created.

`tc_fix_data`
This has the type `TC_FIX_TYPE`, and is only defined if the target defines that macro.

Frags

The `fragS` structure is defined in '`as.h`'. Each frag represents a portion of the final object file. As GAS reads the source file, it creates frags to hold the data that it reads. At the end of the assembly the frags and fixups are processed to produce the final contents.

`fr_address`
: The address of the frag. This is not set until the assembler rescans the list of all frags after the entire input file is parsed. The function `relax_segment` fills in this field.

`fr_next`
: Pointer to the next frag in this (sub)section.

`fr_fix`
: Fixed number of characters we know we're going to emit to the output file. May be zero.

`fr_var`
: Variable number of characters we may output, after the initial `fr_fix` characters. May be zero.

`fr_offset`
: The interpretation of this field is controlled by `fr_type`. Generally, if `fr_var` is non-zero, this is a repeat count: the `fr_var` characters are output `fr_offset` times.

`line`
: Holds line number info when an assembler listing was requested.

`fr_type`
: Relaxation state. This field indicates the interpretation of `fr_offset`, `fr_symbol` and the variable-length tail of the frag, as well as the treatment it gets in various phases of processing. It does not affect the initial `fr_fix` characters; they are always supposed to be output verbatim (fixups aside). See below for specific values this field can have.

`fr_subtype`
: Relaxation substate. If the macro `md_relax_frag` isn't defined, this is assumed to be an index into `TC_GENERIC_RELAX_TABLE` for the generic relaxation code to process (see [Relaxation], page 27). If `md_relax_frag` is defined, this field is available for any use by the CPU-specific code.

`fr_symbol`
: This normally indicates the symbol to use when relaxing the frag according to `fr_type`.

`fr_opcode`
: Points to the lowest-addressed byte of the opcode, for use in relaxation.

`tc_frag_data`
: Target specific fragment data of type TC_FRAG_TYPE. Only present if `TC_FRAG_TYPE` is defined.

`fr_file`
`fr_line`
: The file and line where this frag was last modified.

`fr_literal`
: Declared as a one-character array, this last field grows arbitrarily large to hold the actual contents of the frag.

These are the possible relaxation states, provided in the enumeration type `relax_stateT`, and the interpretations they represent for the other fields:

`rs_align`
`rs_align_code`
: The start of the following frag should be aligned on some boundary. In this frag, `fr_offset` is the logarithm (base 2) of the alignment in bytes. (For example, if alignment on an 8-byte boundary were desired, `fr_offset` would have a value of 3.) The variable characters indicate the fill pattern to be used. The `fr_subtype` field holds the maximum number of bytes to skip when doing this alignment. If more bytes are needed, the alignment is not done. An `fr_subtype` value of 0 means no maximum, which is the normal case. Target backends can use `rs_align_code` to handle certain types of alignment differently.

`rs_broken_word`
: This indicates that "broken word" processing should be done (see [Broken words], page 30). If broken word processing is not necessary on the target machine, this enumerator value will not be defined.

`rs_cfa`
: This state is used to implement exception frame optimizations. The `fr_symbol` is an expression symbol for the subtraction which may be relaxed. The `fr_opcode` field holds the frag for the preceding command byte. The `fr_offset` field holds the offset within that frag. The `fr_subtype` field is used during relaxation to hold the current size of the frag.

`rs_fill`
: The variable characters are to be repeated `fr_offset` times. If `fr_offset` is 0, this frag has a length of `fr_fix`. Most frags have this type.

`rs_leb128`
: This state is used to implement the DWARF "little endian base 128" variable length number format. The `fr_symbol` is always an expression symbol, as constant expressions are emitted directly. The `fr_offset` field is used during relaxation to hold the previous size of the number so that we can determine if the fragment changed size.

`rs_machine_dependent`
: Displacement relaxation is to be done on this frag. The target is indicated by `fr_symbol` and `fr_offset`, and `fr_subtype` indicates the particular machine-specific addressing mode desired. See [Relaxation], page 27.

`rs_org`
: The start of the following frag should be pushed back to some specific offset within the section. (Some assemblers use the value as an absolute address; GAS does not handle final absolute addresses, but rather requires that the linker set them.) The offset is given by `fr_symbol` and `fr_offset`; one character from the variable-length tail is used as the fill character.

A chain of frags is built up for each subsection. The data structure describing a chain is called a `frchainS`, and contains the following fields:

`frch_root`
: Points to the first frag in the chain. May be NULL if there are no frags in this chain.

`frch_last`
: Points to the last frag in the chain, or NULL if there are none.

`frch_next`
: Next in the list of `frchainS` structures.

`frch_seg`
: Indicates the section this frag chain belongs to.

`frch_subseg`
: Subsection (subsegment) number of this frag chain.

`fix_root, fix_tail`
: (Defined only if `BFD_ASSEMBLER` is defined). Point to first and last `fixS` structures associated with this subsection.

`frch_obstack`
: Not currently used. Intended to be used for frag allocation for this subsection. This should reduce frag generation caused by switching sections.

`frch_frag_now`
: The current frag for this subsegment.

A `frchainS` corresponds to a subsection; each section has a list of `frchainS` records associated with it. In most cases, only one subsection of each section is used, so the list will only be one element long, but any processing of frag chains should be prepared to deal with multiple chains per section.

After the input files have been completely processed, and no more frags are to be generated, the frag chains are joined into one per section for further processing. After this point, it is safe to operate on one chain per section.

The assembler always has a current frag, named `frag_now`. More space is allocated for the current frag using the `frag_more` function; this returns a pointer to the amount of requested space. The function `frag_room` says by how much the current frag can be extended. Relaxing is done using variant frags allocated by `frag_var` or `frag_variant` (see [Relaxation], page 27).

What GAS does when it runs

This is a quick look at what an assembler run looks like.

- The assembler initializes itself by calling various init routines.
- For each source file, the `read_a_source_file` function reads in the file and parses it. The global variable `input_line_pointer` points to the current text; it is guaranteed to be correct up to the end of the line, but not farther.
- For each line, the assembler passes labels to the `colon` function, and isolates the first word. If it looks like a pseudo-op, the word is looked up in the pseudo-op hash table `po_hash` and dispatched to a pseudo-op routine. Otherwise, the target dependent `md_assemble` routine is called to parse the instruction.
- When pseudo-ops or instructions output data, they add it to a frag, calling `frag_more` to get space to store it in.
- Pseudo-ops and instructions can also output fixups created by `fix_new` or `fix_new_exp`.
- For certain targets, instructions can create variant frags which are used to store relaxation information (see [Relaxation], page 27).
- When the input file is finished, the `write_object_file` routine is called. It assigns addresses to all the frags (`relax_segment`), resolves all the fixups (`fixup_segment`), resolves all the symbol values (using `resolve_symbol_value`), and finally writes out the file (in the `BFD_ASSEMBLER` case, this is done by simply calling `bfd_close`).

Porting GAS

Each GAS target specifies two main things: the CPU file and the object format file. Two main switches in the '`configure.in`' file handle this. The first switches on CPU type to set the shell variable `cpu_type`. The second switches on the entire target to set the shell variable `fmt`.

The configure script uses the value of `cpu_type` to select two files in the '`config`' directory: '`tc-`*`CPU`*`.c`' and '`tc-`*`CPU`*`.h`'. The configuration process will create a file named '`targ-cpu.h`' in the build directory which includes '`tc-`*`CPU`*`.h`'.

The configure script also uses the value of `fmt` to select two files: '`obj-`*`fmt`*`.c`' and '`obj-`*`fmt`*`.h`'. The configuration process will create a file named '`obj-format.h`' in the build directory which includes '`obj-`*`fmt`*`.h`'.

You can also set the emulation in the configure script by setting the `em` variable. Normally the default value of '`generic`' is fine. The configuration process will create a file named '`targ-env.h`' in the build directory which includes '`te-`*`em`*`.h`'.

There is a special case for COFF. For historical reason, the GNU COFF assembler doesn't follow the documented behavior on certain debug symbols for the compatibility with other COFF assemblers. A port can define `STRICTCOFF` in the configure script to make the GNU COFF assembler to follow the documented behavior.

Porting GAS to a new CPU requires writing the '`tc-`*`CPU`*' files. Porting GAS to a new object file format requires writing the '`obj-`*`fmt`*' files. There is sometimes some interaction between these two files, but it is normally minimal.

The best approach is, of course, to copy existing files. The documentation below assumes that you are looking at existing files to see usage details.

These interfaces have grown over time, and have never been carefully thought out or designed. Nothing about the interfaces described here is cast in stone. It is possible that they will change from one version of the assembler to the next. Also, new macros are added all the time as they are needed.

Writing a CPU backend

The CPU backend files are the heart of the assembler. They are the only parts of the assembler which actually know anything about the instruction set of the processor.

You must define a reasonably small list of macros and functions in the CPU backend files. You may define a large number of additional macros in the CPU backend files, not all of which are documented here. You must, of course, define macros in the '`.h`' file, which is included by every assembler source file. You may define the functions as macros in the '`.h`' file, or as functions in the '`.c`' file.

`TC_`*`CPU`*
: By convention, you should define this macro in the '`.h`' file. For example, '`tc-m68k.h`' defines `TC_M68K`. You might have to use this if it is necessary to add CPU specific code to the object format file.

`TARGET_FORMAT`
: This macro is the BFD target name to use when creating the output file. This will normally depend upon the `OBJ_`*`FMT`* macro.

`TARGET_ARCH`
: This macro is the BFD architecture to pass to `bfd_set_arch_mach`.

`TARGET_MACH`
: This macro is the BFD machine number to pass to `bfd_set_arch_mach`. If it is not defined, GAS will use 0.

`TARGET_BYTES_BIG_ENDIAN`
: You should define this macro to be non-zero if the target is big endian, and zero if the target is little endian.

`md_shortopts`
`md_longopts`
`md_longopts_size`
`md_parse_option`
`md_show_usage`
`md_after_parse_args`
: GAS uses these variables and functions during option processing. `md_shortopts` is a `const char *` which GAS adds to the machine independent string passed to `getopt`. `md_longopts` is a `struct option []` which GAS adds to the machine independent long options passed to `getopt`; you may use `OPTION_MD_BASE`, defined in '`as.h`', as the start of a set of long option indices, if necessary. `md_longopts_size` is a `size_t` holding the size `md_longopts`. GAS will call `md_parse_option` whenever `getopt` returns an unrecognized code, presumably indicating a special code value which appears in `md_longopts`. GAS will call `md_show_usage` when a usage message is printed; it should print a description of the machine specific options. `md_after_pase_args`, if defined, is called after all options are processed, to let the backend override settings done by the generic option parsing.

`md_begin`
: GAS will call this function at the start of the assembly, after the command line arguments have been parsed and all the machine independent initializations have been completed.

`md_cleanup`
: If you define this macro, GAS will call it at the end of each input file.

`md_assemble`
: GAS will call this function for each input line which does not contain a pseudo-op. The argument is a null terminated string. The function should assemble the string as an instruction with operands. Normally `md_assemble` will do this by calling `frag_more` and writing out some bytes (see [Frags], page 9). `md_assemble` will call `fix_new` to create fixups as needed (see [Fixups], page 8). Targets which need to do special purpose relaxation will call `frag_var`.

`md_pseudo_table`
: This is a const array of type `pseudo_typeS`. It is a mapping from pseudo-op names to functions. You should use this table to implement pseudo-ops which are specific to the CPU.

`tc_conditional_pseudoop`
: If this macro is defined, GAS will call it with a `pseudo_typeS` argument. It should return non-zero if the pseudo-op is a conditional which controls whether code is assembled, such as '`.if`'. GAS knows about the normal conditional pseudo-ops, and you should normally not have to define this macro.

`comment_chars`
: This is a null terminated `const char` array of characters which start a comment.

`tc_comment_chars`
: If this macro is defined, GAS will use it instead of `comment_chars`.

`tc_symbol_chars`
: If this macro is defined, it is a pointer to a null terminated list of characters which may appear in an operand. GAS already assumes that all alphanumberic characters, and '`$`', '`.`', and '`_`' may appear in an operand (see '`symbol_chars`' in '`app.c`'). This macro may be defined to treat additional characters as appearing in an operand. This affects the way in which GAS removes whitespace before passing the string to '`md_assemble`'.

`line_comment_chars`
: This is a null terminated `const char` array of characters which start a comment when they appear at the start of a line.

`line_separator_chars`
: This is a null terminated `const char` array of characters which separate lines (null and newline are such characters by default, and need not be listed in this array). Note that line_separator_chars do not separate lines if found in a comment, such as after a character in line_comment_chars or comment_chars.

`EXP_CHARS`
: This is a null terminated `const char` array of characters which may be used as the exponent character in a floating point number. This is normally `"eE"`.

`FLT_CHARS`
: This is a null terminated `const char` array of characters which may be used to indicate a floating point constant. A zero followed by one of these characters is assumed to be followed by a floating point number; thus they operate the way that `0x` is used to indicate a hexadecimal constant. Usually this includes '`r`' and '`f`'.

`LEX_AT`
: You may define this macro to the lexical type of the `@` character. The default is zero.

  Lexical types are a combination of `LEX_NAME` and `LEX_BEGIN_NAME`, both defined in '`read.h`'. `LEX_NAME` indicates that the character may appear in a name. `LEX_BEGIN_NAME` indicates that the character may appear at the beginning of a name.

`LEX_BR`
: You may define this macro to the lexical type of the brace characters `{`, `}`, `[`, and `]`. The default value is zero.

`LEX_PCT`
: You may define this macro to the lexical type of the `%` character. The default value is zero.

`LEX_QM` You may define this macro to the lexical type of the `?` character. The default value it zero.

`LEX_DOLLAR`
: You may define this macro to the lexical type of the *$* character. The default value is `LEX_NAME | LEX_BEGIN_NAME`.

`NUMBERS_WITH_SUFFIX`
: When this macro is defined to be non-zero, the parser allows the radix of a constant to be indicated with a suffix. Valid suffixes are binary (B), octal (Q), and hexadecimal (H). Case is not significant.

`SINGLE_QUOTE_STRINGS`
: If you define this macro, GAS will treat single quotes as string delimiters. Normally only double quotes are accepted as string delimiters.

`NO_STRING_ESCAPES`
: If you define this macro, GAS will not permit escape sequences in a string.

`ONLY_STANDARD_ESCAPES`
: If you define this macro, GAS will warn about the use of nonstandard escape sequences in a string.

`md_start_line_hook`
: If you define this macro, GAS will call it at the start of each line.

`LABELS_WITHOUT_COLONS`
: If you define this macro, GAS will assume that any text at the start of a line is a label, even if it does not have a colon.

`TC_START_LABEL`
`TC_START_LABEL_WITHOUT_COLON`
: You may define this macro to control what GAS considers to be a label. The default definition is to accept any name followed by a colon character.

`TC_START_LABEL_WITHOUT_COLON`
: Same as TC_START_LABEL, but should be used instead of TC_START_LABEL when LABELS_WITHOUT_COLONS is defined.

`NO_PSEUDO_DOT`
: If you define this macro, GAS will not require pseudo-ops to start with a `.` character.

`TC_EQUAL_IN_INSN`
: If you define this macro, it should return nonzero if the instruction is permitted to contain an `=` character. GAS will call it with two arguments, the character before the `=` character, and the value of `input_line_pointer` at that point. GAS uses this macro to decide if a `=` is an assignment or an instruction.

`TC_EOL_IN_INSN`
: If you define this macro, it should return nonzero if the current input line pointer should be treated as the end of a line.

`TC_CASE_SENSITIVE`
: Define this macro if instruction mnemonics and pseudos are case sensitive. The default is to have it undefined giving case insensitive names.

`md_parse_name`
: If this macro is defined, GAS will call it for any symbol found in an expression. You can define this to handle special symbols in a special way. If a symbol always has a certain value, you should normally enter it in the symbol table, perhaps using `reg_section`.

`md_undefined_symbol`
: GAS will call this function when a symbol table lookup fails, before it creates a new symbol. Typically this would be used to supply symbols whose name or value changes dynamically, possibly in a context sensitive way. Predefined symbols with fixed values, such as register names or condition codes, are typically entered directly into the symbol table when `md_begin` is called. One argument is passed, a `char *` for the symbol.

`md_operand`
: GAS will call this function with one argument, an `expressionS` pointer, for any expression that can not be recognized. When the function is called, `input_line_pointer` will point to the start of the expression.

`tc_unrecognized_line`
: If you define this macro, GAS will call it when it finds a line that it can not parse.

`md_do_align`
: You may define this macro to handle an alignment directive. GAS will call it when the directive is seen in the input file. For example, the i386 backend uses this to generate efficient nop instructions of varying lengths, depending upon the number of bytes that the alignment will skip.

`HANDLE_ALIGN`
: You may define this macro to do special handling for an alignment directive. GAS will call it at the end of the assembly.

`TC_IMPLICIT_LCOMM_ALIGNMENT (`*`size`*`,` *`p2var`*`)`
: An `.lcomm` directive with no explicit alignment parameter will use this macro to set *p2var* to the alignment that a request for *size* bytes will have. The alignment is expressed as a power of two. If no alignment should take place, the macro definition should do nothing. Some targets define a `.bss` directive that is also affected by this macro. The default definition will set *p2var* to the truncated power of two of sizes up to eight bytes.

`md_flush_pending_output`
: If you define this macro, GAS will call it each time it skips any space because of a space filling or alignment or data allocation pseudo-op.

`TC_PARSE_CONS_EXPRESSION`
: You may define this macro to parse an expression used in a data allocation pseudo-op such as `.word`. You can use this to recognize relocation directives that may appear in such directives.

`BITFIELD_CONS_EXPRESSION`
: If you define this macro, GAS will recognize bitfield instructions in data allocation pseudo-ops, as used on the i960.

`REPEAT_CONS_EXPRESSION`
: If you define this macro, GAS will recognize repeat counts in data allocation pseudo-ops, as used on the MIPS.

`md_cons_align`
: You may define this macro to do any special alignment before a data allocation pseudo-op.

`TC_CONS_FIX_NEW`
: You may define this macro to generate a fixup for a data allocation pseudo-op.

`TC_INIT_FIX_DATA (`*`fixp`*`)`
: A C statement to initialize the target specific fields of fixup *fixp*. These fields are defined with the `TC_FIX_TYPE` macro.

`TC_FIX_DATA_PRINT (`*`stream`*`, `*`fixp`*`)`
: A C statement to output target specific debugging information for fixup *fixp* to *stream*. This macro is called by `print_fixup`.

`TC_FRAG_INIT (`*`fragp`*`)`
: A C statement to initialize the target specific fields of frag *fragp*. These fields are defined with the `TC_FRAG_TYPE` macro.

`md_number_to_chars`
: This should just call either `number_to_chars_bigendian` or `number_to_chars_littleendian`, whichever is appropriate. On targets like the MIPS which support options to change the endianness, which function to call is a runtime decision. On other targets, `md_number_to_chars` can be a simple macro.

`md_atof (`*`type`*`,`*`litP`*`,`*`sizeP`*`)`
: This function is called to convert an ASCII string into a floating point value in format used by the CPU. It takes three arguments. The first is *type* which is a byte describing the type of floating point number to be created. Possible values are *'f'* or *'s'* for single precision, *'d'* or *'r'* for double precision and *'x'* or *'p'* for extended precision. Either lower or upper case versions of these letters can be used.

  The second parameter is *litP* which is a pointer to a byte array where the converted value should be stored. The third argument is *sizeP*, which is a pointer to a integer that should be filled in with the number of *LITTLENUM*s emitted into the byte array. (*LITTLENUM* is defined in gas/bignum.h). The function should return NULL upon success or an error string upon failure.

`TC_LARGEST_EXPONENT_IS_NORMAL`
: This macro is used only by '`atof-ieee.c`'. It should evaluate to true if floats of the given precision use the largest exponent for normal numbers instead of NaNs and infinities. *precision* is '`F_PRECISION`' for single precision, '`D_PRECISION`' for double precision, or '`X_PRECISION`' for extended double precision.

The macro has a default definition which returns 0 for all cases.

`md_reloc_size`
: This variable is only used in the original version of gas (not `BFD_ASSEMBLER` and not `MANY_SEGMENTS`). It holds the size of a relocation entry.

`WORKING_DOT_WORD`
`md_short_jump_size`
`md_long_jump_size`
`md_create_short_jump`
`md_create_long_jump`
`TC_CHECK_ADJUSTED_BROKEN_DOT_WORD`
: If `WORKING_DOT_WORD` is defined, GAS will not do broken word processing (see [Broken words], page 30). Otherwise, you should set `md_short_jump_size` to the size of a short jump (a jump that is just long enough to jump around a number of long jumps) and `md_long_jump_size` to the size of a long jump (a jump that can go anywhere in the function). You should define `md_create_short_jump` to create a short jump around a number of long jumps, and define `md_create_long_jump` to create a long jump. If defined, the macro TC_CHECK_ADJUSTED_BROKEN_DOT_WORD will be called for each adjusted word just before the word is output. The macro takes two arguments, an `addressT` with the adjusted word and a pointer to the current `struct broken_word`.

`md_estimate_size_before_relax`
: This function returns an estimate of the size of a `rs_machine_dependent` frag before any relaxing is done. It may also create any necessary relocations.

`md_relax_frag`
: This macro may be defined to relax a frag. GAS will call this with the segment, the frag, and the change in size of all previous frags; `md_relax_frag` should return the change in size of the frag. See [Relaxation], page 27.

`TC_GENERIC_RELAX_TABLE`
: If you do not define `md_relax_frag`, you may define `TC_GENERIC_RELAX_TABLE` as a table of `relax_typeS` structures. The machine independent code knows how to use such a table to relax PC relative references. See '`tc-m68k.c`' for an example. See [Relaxation], page 27.

`md_prepare_relax_scan`
: If defined, it is a C statement that is invoked prior to scanning the relax table.

`LINKER_RELAXING_SHRINKS_ONLY`
: If you define this macro, and the global variable '`linkrelax`' is set (because of a command line option, or unconditionally in `md_begin`), a '`.align`' directive will cause extra space to be allocated. The linker can then discard this space when relaxing the section.

`TC_LINKRELAX_FIXUP (`*`segT`*`)`
: If defined, this macro allows control over whether fixups for a given section will be processed when the *linkrelax* variable is set. The macro is given the

N_TYPE bits for the section in its *segT* argument. If the macro evaluates to a non-zero value then the fixups will be converted into relocs, otherwise they will be passed to *md_apply_fix3* as normal.

`md_convert_frag`
: GAS will call this for each rs_machine_dependent fragment. The instruction is completed using the data from the relaxation pass. It may also create any necessary relocations. See [Relaxation], page 27.

`TC_FINALIZE_SYMS_BEFORE_SIZE_SEG`
: Specifies the value to be assigned to `finalize_syms` before the function `size_segs` is called. Since `size_segs` calls `cvt_frag_to_fill` which can call `md_convert_frag`, this constant governs whether the symbols accessed in `md_convert_frag` will be fully resolved. In particular it governs whether local symbols will have been resolved, and had their frag information removed. Depending upon the processing performed by `md_convert_frag` the frag information may or may not be necessary, as may the resolved values of the symbols. The default value is 1.

`TC_VALIDATE_FIX (`*`fixP`*`, `*`seg`*`, `*`skip`*`)`
: This macro is evaluated for each fixup (when *linkrelax* is not set). It may be used to change the fixup in `struct fix *`*`fixP`* before the generic code sees it, or to fully process the fixup. In the latter case, a `goto` *`skip`* will bypass the generic code.

`md_apply_fix3 (`*`fixP`*`, `*`valP`*`, `*`seg`*`)`
: GAS will call this for each fixup that passes the `TC_VALIDATE_FIX` test when *linkrelax* is not set. It should store the correct value in the object file. `struct fix *`*`fixP`* is the fixup `md_apply_fix3` is operating on. `valueT *`*`valP`* is the value to store into the object files, or at least is the generic code's best guess. Specifically, **valP* is the value of the fixup symbol, perhaps modified by `MD_APPLY_SYM_VALUE`, plus *`fixP`*`->fx_offset` (symbol addend), less `MD_PCREL_FROM_SECTION` for pc-relative fixups. `segT` *`seg`* is the section the fix is in. `fixup_segment` performs a generic overflow check on **valP* after `md_apply_fix3` returns. If the overflow check is relevant for the target machine, then `md_apply_fix3` should modify **valP*, typically to the value stored in the object file.

`TC_FORCE_RELOCATION (`*`fix`*`)`
: If this macro returns non-zero, it guarantees that a relocation will be emitted even when the value can be resolved locally, as `fixup_segment` tries to reduce the number of relocations emitted. For example, a fixup expression against an absolute symbol will normally not require a reloc. If undefined, a default of `(S_FORCE_RELOC ((`*`fix`*`)->fx_addsy))` is used.

`TC_FORCE_RELOCATION_ABS (`*`fix`*`)`
: Like `TC_FORCE_RELOCATION`, but used only for fixup expressions against an absolute symbol. If undefined, `TC_FORCE_RELOCATION` will be used.

`TC_FORCE_RELOCATION_LOCAL (`*`fix`*`)`
: Like `TC_FORCE_RELOCATION`, but used only for fixup expressions against a symbol in the current section. If undefined, fixups that are not `fx_pcrel` or `fx_plt` or for which `TC_FORCE_RELOCATION` returns non-zero, will emit relocs.

`TC_FORCE_RELOCATION_SUB_SAME (`*`fix`*`, `*`seg`*`)`
: This macro controls resolution of fixup expressions involving the difference of two symbols in the same section. If this macro returns zero, the subtrahend will be resolved and `fx_subsy` set to `NULL` for `md_apply_fix3`. If undefined, the default of `! SEG_NORMAL (`*`seg`*`) || TC_FORCE_RELOCATION (`*`fix`*`)` will be used.

`TC_FORCE_RELOCATION_SUB_ABS (`*`fix`*`)`
: Like `TC_FORCE_RELOCATION_SUB_SAME`, but used when the subtrahend is an absolute symbol. If the macro is undefined a default of `0` is used.

`TC_FORCE_RELOCATION_SUB_LOCAL (`*`fix`*`)`
: Like `TC_FORCE_RELOCATION_SUB_ABS`, but the subtrahend is a symbol in the same section as the fixup.

`TC_VALIDATE_FIX_SUB (`*`fix`*`)`
: This macro is evaluated for any fixup with a `fx_subsy` that `fixup_segment` cannot reduce to a number. If the macro returns `false` an error will be reported.

`MD_APPLY_SYM_VALUE (`*`fix`*`)`
: This macro controls whether the symbol value becomes part of the value passed to `md_apply_fix3`. If the macro is undefined, or returns non-zero, the symbol value will be included. For ELF, a suitable definition might simply be `0`, because ELF relocations don't include the symbol value in the addend.

`S_FORCE_RELOC (`*`sym`*`, `*`strict`*`)`
: This macro (or function, for `BFD_ASSEMBLER` gas) returns true for symbols that should not be reduced to section symbols or eliminated from expressions, because they may be overridden by the linker. ie. for symbols that are undefined or common, and when *strict* is set, weak, or global (for ELF assemblers that support ELF shared library linking semantics).

`EXTERN_FORCE_RELOC`
: This macro controls whether `S_FORCE_RELOC` returns true for global symbols. If undefined, the default is `true` for ELF assemblers, and `false` for non-ELF.

`tc_gen_reloc`
: A `BFD_ASSEMBLER` GAS will call this to generate a reloc. GAS will pass the resulting reloc to `bfd_install_relocation`. This currently works poorly, as `bfd_install_relocation` often does the wrong thing, and instances of `tc_gen_reloc` have been written to work around the problems, which in turns makes it difficult to fix `bfd_install_relocation`.

`RELOC_EXPANSION_POSSIBLE`
: If you define this macro, it means that `tc_gen_reloc` may return multiple relocation entries for a single fixup. In this case, the return value of `tc_gen_reloc` is a pointer to a null terminated array.

`MAX_RELOC_EXPANSION`
: You must define this if `RELOC_EXPANSION_POSSIBLE` is defined; it indicates the largest number of relocs which `tc_gen_reloc` may return for a single fixup.

`tc_fix_adjustable`
: You may define this macro to indicate whether a fixup against a locally defined symbol should be adjusted to be against the section symbol. It should return a non-zero value if the adjustment is acceptable.

`MD_PCREL_FROM_SECTION (`*`fixp`*`, `*`section`*`)`
: If you define this macro, it should return the position from which the PC relative adjustment for a PC relative fixup should be made. On many processors, the base of a PC relative instruction is the next instruction, so this macro would return the length of an instruction, plus the address of the PC relative fixup. The latter can be calculated as *fixp*->fx_where + *fixp*->fx_frag->fr_address .

`md_pcrel_from`
: This is the default value of `MD_PCREL_FROM_SECTION`. The difference is that `md_pcrel_from` does not take a section argument.

`tc_frob_label`
: If you define this macro, GAS will call it each time a label is defined.

`md_section_align`
: GAS will call this function for each section at the end of the assembly, to permit the CPU backend to adjust the alignment of a section. The function must take two arguments, a `segT` for the section and a `valueT` for the size of the section, and return a `valueT` for the rounded size.

`md_macro_start`
: If defined, GAS will call this macro when it starts to include a macro expansion. `macro_nest` indicates the current macro nesting level, which includes the one being expanded.

`md_macro_info`
: If defined, GAS will call this macro after the macro expansion has been included in the input and after parsing the macro arguments. The single argument is a pointer to the macro processing's internal representation of the macro (macro_entry *), which includes expansion of the formal arguments.

`md_macro_end`
: Complement to md_macro_start. If defined, it is called when finished processing an inserted macro expansion, just before decrementing macro_nest.

`DOUBLEBAR_PARALLEL`
: Affects the preprocessor so that lines containing '`||`' don't have their whitespace stripped following the double bar. This is useful for targets that implement parallel instructions.

`KEEP_WHITE_AROUND_COLON`
: Normally, whitespace is compressed and removed when, in the presence of the colon, the adjoining tokens can be distinguished. This option affects the preprocessor so that whitespace around colons is preserved. This is useful when

colons might be removed from the input after preprocessing but before assembling, so that adjoining tokens can still be distinguished if there is whitespace, or concatenated if there is not.

`tc_frob_section`
: If you define this macro, a `BFD_ASSEMBLER` GAS will call it for each section at the end of the assembly.

`tc_frob_file_before_adjust`
: If you define this macro, GAS will call it after the symbol values are resolved, but before the fixups have been changed from local symbols to section symbols.

`tc_frob_symbol`
: If you define this macro, GAS will call it for each symbol. You can indicate that the symbol should not be included in the object file by defining this macro to set its second argument to a non-zero value.

`tc_frob_file`
: If you define this macro, GAS will call it after the symbol table has been completed, but before the relocations have been generated.

`tc_frob_file_after_relocs`
: If you define this macro, GAS will call it after the relocs have been generated.

`md_post_relax_hook`
: If you define this macro, GAS will call it after relaxing and sizing the segments.

`LISTING_HEADER`
: A string to use on the header line of a listing. The default value is simply `"GAS LISTING"`.

`LISTING_WORD_SIZE`
: The number of bytes to put into a word in a listing. This affects the way the bytes are clumped together in the listing. For example, a value of 2 might print '`1234 5678`' where a value of 1 would print '`12 34 56 78`'. The default value is 4.

`LISTING_LHS_WIDTH`
: The number of words of data to print on the first line of a listing for a particular source line, where each word is `LISTING_WORD_SIZE` bytes. The default value is 1.

`LISTING_LHS_WIDTH_SECOND`
: Like `LISTING_LHS_WIDTH`, but applying to the second and subsequent line of the data printed for a particular source line. The default value is 1.

`LISTING_LHS_CONT_LINES`
: The maximum number of continuation lines to print in a listing for a particular source line. The default value is 4.

`LISTING_RHS_WIDTH`
: The maximum number of characters to print from one line of the input file. The default value is 100.

`TC_COFF_SECTION_DEFAULT_ATTRIBUTES`
: The COFF `.section` directive will use the value of this macro to set a new section's attributes when a directive has no valid flags or when the flag is `w`. The default value of the macro is `SEC_LOAD | SEC_DATA`.

`DWARF2_FORMAT ()`
: If you define this, it should return one of `dwarf2_format_32bit`, `dwarf2_format_64bit`, or `dwarf2_format_64bit_irix` to indicate the size of internal DWARF section offsets and the format of the DWARF initial length fields. When `dwarf2_format_32bit` is returned, the initial length field will be 4 bytes long and section offsets are 32 bits in size. For `dwarf2_format_64bit` and `dwarf2_format_64bit_irix`, section offsets are 64 bits in size, but the initial length field differs. An 8 byte initial length is indicated by `dwarf2_format_64bit_irix` and `dwarf2_format_64bit` indicates a 12 byte initial length field in which the first four bytes are 0xffffffff and the next 8 bytes are the section's length.

  If you don't define this, `dwarf2_format_32bit` will be used as the default.

  This define only affects `.debug_info` and `.debug_line` sections generated by the assembler. DWARF 2 sections generated by other tools will be unaffected by this setting.

`DWARF2_ADDR_SIZE (`*`bfd`*`)`
: It should return the size of an address, as it should be represented in debugging info. If you don't define this macro, the default definition uses the number of bits per address, as defined in *bfd*, divided by 8.

Writing an object format backend

As with the CPU backend, the object format backend must define a few things, and may define some other things. The interface to the object format backend is generally simpler; most of the support for an object file format consists of defining a number of pseudo-ops.

The object format '`.h`' file must include '`targ-cpu.h`'.

This section will only define the `BFD_ASSEMBLER` version of GAS. It is impossible to support a new object file format using any other version anyhow, as the original GAS version only supports a.out, and the `MANY_SEGMENTS` GAS version only supports COFF.

`OBJ_`*`format`*
: By convention, you should define this macro in the '`.h`' file. For example, '`obj-elf.h`' defines `OBJ_ELF`. You might have to use this if it is necessary to add object file format specific code to the CPU file.

`obj_begin`
: If you define this macro, GAS will call it at the start of the assembly, after the command line arguments have been parsed and all the machine independent initializations have been completed.

`obj_app_file`
: If you define this macro, GAS will invoke it when it sees a `.file` pseudo-op or a '`#`' line as used by the C preprocessor.

`OBJ_COPY_SYMBOL_ATTRIBUTES`
: You should define this macro to copy object format specific information from one symbol to another. GAS will call it when one symbol is equated to another.

`obj_sec_sym_ok_for_reloc`
: You may define this macro to indicate that it is OK to use a section symbol in a relocation entry. If it is not, GAS will define a new symbol at the start of a section.

`EMIT_SECTION_SYMBOLS`
: You should define this macro with a zero value if you do not want to include section symbols in the output symbol table. The default value for this macro is one.

`obj_adjust_symtab`
: If you define this macro, GAS will invoke it just before setting the symbol table of the output BFD. For example, the COFF support uses this macro to generate a `.file` symbol if none was generated previously.

`SEPARATE_STAB_SECTIONS`
: You may define this macro to a nonzero value to indicate that stabs should be placed in separate sections, as in ELF.

`INIT_STAB_SECTION`
: You may define this macro to initialize the stabs section in the output file.

`OBJ_PROCESS_STAB`
: You may define this macro to do specific processing on a stabs entry.

`obj_frob_section`
: If you define this macro, GAS will call it for each section at the end of the assembly.

`obj_frob_file_before_adjust`
: If you define this macro, GAS will call it after the symbol values are resolved, but before the fixups have been changed from local symbols to section symbols.

`obj_frob_symbol`
: If you define this macro, GAS will call it for each symbol. You can indicate that the symbol should not be included in the object file by defining this macro to set its second argument to a non-zero value.

`obj_frob_file`
: If you define this macro, GAS will call it after the symbol table has been completed, but before the relocations have been generated.

`obj_frob_file_after_relocs`
: If you define this macro, GAS will call it after the relocs have been generated.

`SET_SECTION_RELOCS (`*`sec, relocs, n`*`)`
: If you define this, it will be called after the relocations have been set for the section *sec*. The list of relocations is in *relocs*, and the number of relocations is in *n*. This is only used with `BFD_ASSEMBLER`.

Writing emulation files

Normally you do not have to write an emulation file. You can just use ‘`te-generic.h`’.

If you do write your own emulation file, it must include ‘`obj-format.h`’.

An emulation file will often define `TE_`*`EM`*; this may then be used in other files to change the output.

Relaxation

Relaxation is a generic term used when the size of some instruction or data depends upon the value of some symbol or other data.

GAS knows to relax a particular type of PC relative relocation using a table. You can also define arbitrarily complex forms of relaxation yourself.

Relaxing with a table

If you do not define `md_relax_frag`, and you do define `TC_GENERIC_RELAX_TABLE`, GAS will relax `rs_machine_dependent` frags based on the frag subtype and the displacement to some specified target address. The basic idea is that several machines have different addressing modes for instructions that can specify different ranges of values, with successive modes able to access wider ranges, including the entirety of the previous range. Smaller ranges are assumed to be more desirable (perhaps the instruction requires one word instead of two or three); if this is not the case, don't describe the smaller-range, inferior mode.

The `fr_subtype` field of a frag is an index into a CPU-specific relaxation table. That table entry indicates the range of values that can be stored, the number of bytes that will have to be added to the frag to accommodate the addressing mode, and the index of the next entry to examine if the value to be stored is outside the range accessible by the current addressing mode. The `fr_symbol` field of the frag indicates what symbol is to be accessed; the `fr_offset` field is added in.

If the `TC_PCREL_ADJUST` macro is defined, which currently should only happen for the NS32k family, the `TC_PCREL_ADJUST` macro is called on the frag to compute an adjustment to be made to the displacement.

The value fitted by the relaxation code is always assumed to be a displacement from the current frag. (More specifically, from `fr_fix` bytes into the frag.)

The end of the relaxation sequence is indicated by a "next" value of 0. This means that the first entry in the table can't be used.

For some configurations, the linker can do relaxing within a section of an object file. If call instructions of various sizes exist, the linker can determine which should be used in each instance, when a symbol's value is resolved. In order for the linker to avoid wasting space and having to insert no-op instructions, it must be able to expand or shrink the section contents while still preserving intra-section references and meeting alignment requirements.

For the i960 using b.out format, no expansion is done; instead, each '`.align`' directive causes extra space to be allocated, enough that when the linker is relaxing a section and removing unneeded space, it can discard some or all of this extra padding and cause the following data to be correctly aligned.

For the H8/300, I think the linker expands calls that can't reach, and doesn't worry about alignment issues; the cpu probably never needs any significant alignment beyond the instruction size.

The relaxation table type contains these fields:

`long rlx_forward`
: Forward reach, must be non-negative.

`long rlx_backward`
: Backward reach, must be zero or negative.

`rlx_length`
: Length in bytes of this addressing mode.

`rlx_more`
: Index of the next-longer relax state, or zero if there is no next relax state.

The relaxation is done in `relax_segment` in '`write.c`'. The difference in the length fields between the original mode and the one finally chosen by the relaxing code is taken as the size by which the current frag will be increased in size. For example, if the initial relaxing mode has a length of 2 bytes, and because of the size of the displacement, it gets upgraded to a mode with a size of 6 bytes, it is assumed that the frag will grow by 4 bytes. (The initial two bytes should have been part of the fixed portion of the frag, since it is already known that they will be output.) This growth must be effected by `md_convert_frag`; it should increase the `fr_fix` field by the appropriate size, and fill in the appropriate bytes of the frag. (Enough space for the maximum growth should have been allocated in the call to frag_var as the second argument.)

If relocation records are needed, they should be emitted by `md_estimate_size_before_relax`. This function should examine the target symbol of the supplied frag and correct the `fr_subtype` of the frag if needed. When this function is called, if the symbol has not yet been defined, it will not become defined later; however, its value may still change if the section it is in gets relaxed.

Usually, if the symbol is in the same section as the frag (given by the *sec* argument), the narrowest likely relaxation mode is stored in `fr_subtype`, and that's that.

If the symbol is undefined, or in a different section (and therefore movable to an arbitrarily large distance), the largest available relaxation mode is specified, `fix_new` is called to produce the relocation record, `fr_fix` is increased to include the relocated field (remember, this storage was allocated when `frag_var` was called), and `frag_wane` is called to convert the frag to an `rs_fill` frag with no variant part. Sometimes changing addressing modes may also require rewriting the instruction. It can be accessed via `fr_opcode` or `fr_fix`.

If you generate frags separately for the basic insn opcode and any relaxable operands, do not call `fix_new` thinking you can emit fixups for the opcode field from the relaxable frag. It is not guaranteed to be the same frag. If you need to emit fixups for the opcode field from inspection of the relaxable frag, then you need to generate a common frag for both the basic opcode and relaxable fields, or you need to provide the frag for the opcode to pass to `fix_new`. The latter can be done for example by defining `TC_FRAG_TYPE` to include a pointer to it and defining `TC_FRAG_INIT` to set the pointer.

Sometimes `fr_var` is increased instead, and `frag_wane` is not called. I'm not sure, but I think this is to keep `fr_fix` referring to an earlier byte, and `fr_subtype` set to `rs_machine_dependent` so that `md_convert_frag` will get called.

General relaxing

If using a simple table is not suitable, you may implement arbitrarily complex relaxation semantics yourself. For example, the MIPS backend uses this to emit different instruction sequences depending upon the size of the symbol being accessed.

When you assemble an instruction that may need relaxation, you should allocate a frag using `frag_var` or `frag_variant` with a type of `rs_machine_dependent`. You should store some sort of information in the `fr_subtype` field so that you can figure out what to do with the frag later.

When GAS reaches the end of the input file, it will look through the frags and work out their final sizes.

GAS will first call `md_estimate_size_before_relax` on each `rs_machine_dependent` frag. This function must return an estimated size for the frag.

GAS will then loop over the frags, calling `md_relax_frag` on each `rs_machine_dependent` frag. This function should return the change in size of the frag. GAS will keep looping over the frags until none of the frags changes size.

Broken words

Some compilers, including GCC, will sometimes emit switch tables specifying 16-bit `.word` displacements to branch targets, and branch instructions that load entries from that table to compute the target address. If this is done on a 32-bit machine, there is a chance (at least with really large functions) that the displacement will not fit in 16 bits. The assembler handles this using a concept called *broken words*. This idea is well named, since there is an implied promise that the 16-bit field will in fact hold the specified displacement.

If broken word processing is enabled, and a situation like this is encountered, the assembler will insert a jump instruction into the instruction stream, close enough to be reached with the 16-bit displacement. This jump instruction will transfer to the real desired target address. Thus, as long as the `.word` value really is used as a displacement to compute an address to jump to, the net effect will be correct (minus a very small efficiency cost). If `.word` directives with label differences for values are used for other purposes, however, things may not work properly. For targets which use broken words, the '`-K`' option will warn when a broken word is discovered.

The broken word code is turned off by the `WORKING_DOT_WORD` macro. It isn't needed if `.word` emits a value large enough to contain an address (or, more correctly, any possible difference between two addresses).

Internal functions

This section describes basic internal functions used by GAS.

Warning and error messages

{} **int** `had_warnings` (`void`) [Function]
{} **int** `had_errors` (`void`) [Function]

> Returns non-zero if any warnings or errors, respectively, have been printed during this invocation.

{} **void** `as_perror` (`const char *`*gripe*`, const char *`*filename*) [Function]

> Displays a BFD or system error, then clears the error status.

{} **void** `as_tsktsk` (`const char *`*format*`, ...`) [Function]
{} **void** `as_warn` (`const char *`*format*`, ...`) [Function]
{} **void** `as_bad` (`const char *`*format*`, ...`) [Function]
{} **void** `as_fatal` (`const char *`*format*`, ...`) [Function]

> These functions display messages about something amiss with the input file, or internal problems in the assembler itself. The current file name and line number are printed, followed by the supplied message, formatted using `vfprintf`, and a final newline.
>
> An error indicated by `as_bad` will result in a non-zero exit status when the assembler has finished. Calling `as_fatal` will result in immediate termination of the assembler process.

{} **void** `as_warn_where` (`char *`*file*`, unsigned int `*line*`, const char *`*format*`, ...`) [Function]
{} **void** `as_bad_where` (`char *`*file*`, unsigned int `*line*`, const char *`*format*`, ...`) [Function]

> These variants permit specification of the file name and line number, and are used when problems are detected when reprocessing information saved away when processing some earlier part of the file. For example, fixups are processed after all input has been read, but messages about fixups should refer to the original filename and line number that they are applicable to.

{} **void** `fprint_value` (`FILE *`*file*`, valueT `*val*) [Function]
{} **void** `sprint_value` (`char *`*buf*`, valueT `*val*) [Function]

> These functions are helpful for converting a `valueT` value into printable format, in case it's wider than modes that `*printf` can handle. If the type is narrow enough, a decimal number will be produced; otherwise, it will be in hexadecimal. The value itself is not examined to make this determination.

Hash tables

{} **{struct** `hash_control *}` `hash_new` (`void`) [Function]

> Creates the hash table control structure.

{} **void** `hash_die` (`struct hash_control *`) [Function]

Destroy a hash table.

{} **PTR** `hash_delete` (`struct hash_control *, const char *`) [Function]

Deletes entry from the hash table, returns the value it had.

{} **PTR** `hash_replace` (`struct hash_control *, const char *, PTR`) [Function]

Updates the value for an entry already in the table, returning the old value. If no entry was found, just returns NULL.

{} {**const** `char *`} `hash_insert` (`struct hash_control *, const char *, PTR`) [Function]

Inserting a value already in the table is an error. Returns an error message or NULL.

{} {**const** `char *`} `hash_jam` (`struct hash_control *, const char *, PTR`) [Function]

Inserts if the value isn't already present, updates it if it is.

Test suite

The test suite is kind of lame for most processors. Often it only checks to see if a couple of files can be assembled without the assembler reporting any errors. For more complete testing, write a test which either examines the assembler listing, or runs `objdump` and examines its output. For the latter, the TCL procedure `run_dump_test` may come in handy. It takes the base name of a file, and looks for '*file*`.d`'. This file should contain as its initial lines a set of variable settings in '`#`' comments, in the form:

```
#varname: value
```

The *varname* may be `objdump`, `nm`, or `as`, in which case it specifies the options to be passed to the specified programs. Exactly one of `objdump` or `nm` must be specified, as that also specifies which program to run after the assembler has finished. If *varname* is `source`, it specifies the name of the source file; otherwise, '*file*`.s`' is used. If *varname* is `name`, it specifies the name of the test to be used in the `pass` or `fail` messages.

The non-commented parts of the file are interpreted as regular expressions, one per line. Blank lines in the `objdump` or `nm` output are skipped, as are blank lines in the `.d` file; the other lines are tested to see if the regular expression matches the program output. If it does not, the test fails.

Note that this means the tests must be modified if the `objdump` output style is changed.