as

The GNU Assembler

Dean Elsner, Jay Fenlason & friends

The Free Software Foundation Inc. thanks The Nice Computer Company of Australia for loaning Dean Elsner to write the first (Vax) version of `as` for Project GNU. The proprietors, management and staff of TNCCA thank FSF for distracting the boss while they got some work done.

1 Overview, Usage

This document describes the GNU assembler `as`. This document does *not* describe what an assembler does, or how it works. This document also does *not* describe the opcodes, registers or addressing modes that `as` uses on any paticular computer that `as` runs on. Consult a good book on assemblers or the machine's architecture if you need that information.

This document describes the pseudo-ops that `as` understands, and their syntax. This document also describes some of the machine-dependent features of various flavors of the assembler. This document also describes how the assembler works internally, and provides some information that may be useful to people attempting to port the assembler to another machine.

Throughout this document, we assume that you are running *GNU*, the portable operating system from the *Free Software Foundation, Inc.*. This restricts our attention to certain kinds of computer (in paticular, the kinds of computers that GNU can run on); once this assumption is granted examples and definitions need less qualification.

Readers should already comprehend:

- Central processing unit
- registers
- memory address
- contents of memory address
- bit
- 8-bit byte
- 2's complement arithmetic

`as` is part of a team of programs that turn a high-level human-readable series of instructions into a low-level computer-readable series of instructions. Different versions of `as` are used for different kinds of computer. In paticular, at the moment, `as` only works for the DEC Vax, the Motorola 68020, the Intel 80386 and the National Semiconductor 32xxx.

1.1 Notation

GNU and `as` assume the computer that will run the programs it assembles will obey these rules.

A (memory) *address* is 32 bits. The lowest address is zero.

The *contents* of any memory address is one *byte* of exactly 8 bits.

A *word* is 16 bits stored in two bytes of memory. The addresses of the bytes differ by exactly 1. Notice that the interpretation of the bits in a word and of how to address a word depends on which particular computer you are assembling for.

A *long word*, or *long*, is 32 bits composed of four bytes. It is stored in 4 bytes of memory; these bytes have contiguous addresses. Again the interpretation and addressing of those bits is machine dependent. National Semiconductor 32xxx computers say *double word* where we say *long*.

Numeric quantities are usually *unsigned* or *2's complement*. Bytes, words and longs may store numbers. `as` manipulates integer expressions as 32-bit numbers in 2's complement

format. When asked to store an integer in a byte or word, the lowest order bits are stored. The order of bytes in a word or long in memory is determined by what kind of computer will run the assembled program. We won't mention this important *caveat* again.

The meaning of these terms has changed over time. Although *byte* used to mean any length of contiguous bits, *byte* now pervasively means exactly 8 contiguous bits. A *word* of 16 bits made sense for 16-bit computers. Even on 32-bit computers, a *word* still means 16 bits (to machine language programmers). To many other programmers of GNU a *word* means 32 bits, so beware. Similarly *long* means 32 bits: from "long word". National Semiconductor 32xxx machine language calls a 32-bit number a "double word".

Names for integers of different sizes: some conventions

length (bits)	as	vax	32xxx	68020	GNU C
8	byte	byte	byte	byte	char
16	word	word	word	word	short (int)
32	long	long(-word)	double-word	long(-word)	long (int)
64	quad	quad(-word)			
128	octa	octa-word			

1.2 as, the GNU Assembler

As is an assembler; it is one of the team of programs that 'compile' your programs into the binary numbers that a computer uses to 'run' your program. Often `as` reads a *source* program written by a compiler and writes an *object* program for the linker (sometimes referred to as a *loader*) `ld` to read.

The source program consists of *statements* and comments. Each statement might *assemble* to one (and only one) machine language instruction or to one very simple datum.

Mostly you don't have to think about the assembler because the compiler invokes it as needed; in that sense the assembler is just another part of the compiler. If you write your own assembly language program, then you must run the assembler yourself to get an object file suitable for linking. You can read below how to do this.

`as` is only intended to assemble the output of the C compiler `cc` for use by the linker `ld`. `as` (vax and 68020 versions) tries to assemble correctly everything that the standard assembler would assemble, with a few exceptions (described in the machine-dependent chapters.)

Each version of the assembler knows about just one kind of machine language, but much is common between the versions, including object file formats, (most) assembler directives (often called *pseudo-ops)* and assembler syntax.

Unlike older assemblers, `as` tries to assemble a source program in one pass of the source file. This subtly changes the meaning of the `.org` directive (See Section 6.24 [Org], page 25.).

If you want to write assembly language programs, you must tell `as` what numbers should be in a computer's memory, and which addresses should contain them, so that the program

may be executed by the computer. Using symbols will prevent many bookkeeping mistakes that can occur if you use raw numbers.

1.3 Command Line Synopsis

```
as [ options ] [ -G GDB_symbol_file ] [ -o object_file ][ input1 ... ]
```

After the program name `as` the command line may contain switches and file names in any order. The order of switches doesn't matter but the order of file names is significant. Only the assembler's name `as` is compulsory and it must (of course) be first.

1.3.1 Switches

Except for '`--`' any command line argument that begins with a hyphen ('`-`') is a switch. Each switch changes the behavior of `as`. No switch changes the way another switch works. A switch is a '`-`' followed by a letter; the case of the letter is important. No switch (letter) should be used twice on the same command line. (Nobody has decided what two copies of the same switch should mean.) All switches are optional.

Some switches expect exactly one file name to follow them. The file name may either immediately follow the switch'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
as -omy-object-file.o mumble
```

Always, `--` (that's two hyphens, not one) by itself names the standard input file.

1.4 Input File(s)

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

The source text is a catenation of the text in each file.

Each time you run `as` it assembles exactly one source program. A source program text is made of one or more GNU 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 `as` is given no file names it attempts to read one input file from `as`'s standard input.

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

It is OK to assemble an empty source. You get a small harmless object (output) file.

If you try to assemble no files then `as` will try to read standard input, which is normally your terminal. You may have to type `ctl-D` to tell `as` there is no more program to assemble.

1.4.1 Input Filenames and Line-numbers

A line is text up to and including the next newline. The first line of a file is numbered **1**, the next **2** and so on.

There are two ways of locating a line in the input file(s) and both are 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.

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

Logical files are "pretend" files which bear no relation to physical files. Logical file names help error messages reflect the proper source file. Often they are used when `as`' source is itself synthesized from other files.

1.5 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; named `a.out` unless you tell `as` to give it another name by using the `-o` switch. Conventionally, object file names end with `.o`. The default name of `a.out` is used for historical reasons. Older assemblers were capable of assembling self-contained programs directly into a runnable program. This may still work, but hasn't been tested.

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

1.6 Error and Warning Messages

`as` may write warnings and error messages to the standard error file (usually your terminal). This should not happen when `as` is run automatically by a compiler. Error messages are useful for those (few) people who still write in assembly language.

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:line_number:Warning Message Text
```

If a logical file name has been given (See Section 6.10 [File], page 23.) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (See Section 6.20 [Line], page 24.) 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 (In the grand UN*X tradition).

Error messages have the format

```
file_name:line_number:FATAL:Error Message Text
```

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

1.7 Optional Switches

1.7.1 -f Works Faster

'`-f`' should only be used when assembling programs written by a (trusted) compiler. '`-f`' causes the assembler to not bother pre-processing the input file(s) before assembling them. Needless to say, if the files actually need to be pre-processed (if the contain comments, for example), `as` will not work correctly if '`-f`' is used.

1.7.2 -G Includes GDB Symbolic Information

(This option is depreciated, and may stop working without warning. GNU is abandoning the GDB symbolic information. It doesn't speed things up by much, and is difficult to maintain.)

The C compiler may produce (apart from an assembler source file of your program) symbolic information for the `gdb` program, in a file. Certain assembler statements manipulate this information, and `as` can include the symbolic information in the object file that is the result of your assembly.

Use this switch to say which file contains the symbolic information. The switch needs exactly one filename.

`as` directives that begin with '`.gdb...`' manipulate this `gdb` symbolic information. Unless you use a '`-G`' switch all '`.gdb...`' assembler statements are ignored.

The `gdb` notes file is described elsewhere.

1.7.3 -l Shortens Long Undefined Symbols

If this switch is not given, references to undefined symbols will be a full long (32 bits) wide. (Since `as` cannot know where these symbols will end up being, `as` can only allocate space for the linker to fill in later. Since `as` doesn't know how far away these symbols will be, it allocates as much space as it can.) If this option is given, the references will only be 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 will be less than 17 bits away.

This switch only works with the MC68020 version of `as`.

1.7.4 -L Includes Local Labels

For historical reasons, labels beginning with '`L`' (upper case only) are called *local labels*. Normally you don't see such labels 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 don't normally debug with them.

This switch 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`'.

1.7.5 -m{c}680{0,1,2}0 Different Kinds of 68000

The 68020 version of `as` is usually used to assemble programs for the Motorola MC68020 microprocessor. Occasionally it is used to assemble programs for the mostly-similar-but-slightly-different MC68000 or MC68010 microprocessors. You can give `as` the switches '`-m68000`', '`-mc68000`', '`-m68010`', '`-mc68010`', '`-m68020`', and '`-mc68020`' to tell it what processor it should be assembling for. Unfortunately, these switches are essentially ignored.

1.7.6 -o Names the Object File

There is always one object file output when you run `as`. By default it has the name `a.out`. You use this switch (which takes exactly one filename) to give the object file a different name.

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

1.7.7 -R Folds Data Segment into Text Segment

`-R` tells `as` to write the object file as if all data-segment data lives in the text segment. This is only done at the very last moment: your binary data are the same, but data segment parts are relocated differently. The data segment part of your object file is zero bytes long because all it bytes are appended to the text segment. (See Chapter 3 [Segments], page 13.)

When you use `-R` it would be nice to generate shorter address displacements (possible because we don't have to cross segments) between text and data segment. We don't do this simply for compatibility with older versions of `as`. `-R` may work this way in future.

1.7.8 -W Represses 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 this switch, any warning is repressed. This switch only affects warning messages: it cannot change any detail of how `as` assembles your file. Errors, which stop the assembly, are still reported.

1.7.9 Useless (but Compatible) Switches

`As` accepts any of these switches, gives a warning message that the switch was ignored and proceeds. These switches are for compatibility with scripts designed for other people's assemblers.

`-D` (Debug)
`-S` (Symbol Table)
`-T` (Token Trace)
: Obsolete switches used to debug old assemblers.

`-V` (Virtualize Interpass Temporary File)
: Other assemblers use a temporary file. This switch commanded them to keep the information in active memory rather than in a disk file. `as` always does this, so this switch 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. Other assemblers would emit short and medium branches, unless told by this switch to emit short and long branches. This is an archaic machine-dependent switch.

`-d` (Displacement size for JUMPs)
: Like the `-J` switch, this is archaic. It expects a number following the `-d`. Like switches 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).

`-t` (Temporary File Directory)
: Other assemblers may use a temporary file, and this switch takes a filename being the directory to site the temporary file. `as` does not use a temporary disk file, so this switch makes no difference. `-t` needs exactly one filename.

1.8 Special Features to support Compilers

In order to assemble compiler output into something that will work, `as` will occasionlly do strange things to '`.word`' pseudo-ops. In particular, when `gas` assembles a pseudo-op of the form '`.word sym1-sym2`', and the difference between `sym1` and `sym2` does not fit in 16 bits, `as` will create a *secondary jump table*, immediately before the next label. This *secondary jump table* will be preceeded by a short-jump to the first byte after the table. The short-jump prevents the flow-of-control from accidentally falling into the table. Inside the table will be a long-jump to `sym2`. The original '`.word`' will contain `sym1` minus (the address of the long-jump to sym2) If there were several '`.word sym1-sym2`' before the secondary jump table, all of them will be adjusted. If ther was a '`.word sym3-sym4`', that also did not fit in sixteen bits, a long-jump to `sym4` will be included in the secondary jump table, and the `.word`(s), will be adjusted to contain `sym3` minus (the address of the long-jump to sym4), etc.

This feature may be disabled by compiling `as` *with the* '`-DWORKING_DOT_WORD`' *option.* This feature is likely to confuse assembly language programmers.

2 Syntax

This chapter informally defines the machine-independent syntax allowed in a source file. `as` has ordinary syntax; it tries to be upward compatible from BSD 4.2 assembler except `as` does not assemble Vax bit-fields.

2.1 The Pre-processor

The preprocess phase handles several aspects of the syntax. The pre-processor will be disabled by the '`-f`' option, or if the first line of the source file is `#NO_APP`. The option to disable the pre-processor was designed to make compiler output assemble as fast as possible.

The pre-processor 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.

The pre-processor removes all comments, replacing them with a single space (for /* ... */ comments), or an appropriate number of newlines.

The pre-processor converts character constants into the appropriate numeric values.

This means that excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not pre-processed.

If the first line of an input file is `#NO_APP` or the '`-f`' option is given, the input file will not be pre-processed. Within such an input file, parts of the file can be pre-processed by putting a line that says `#APP` before the text that should be pre-processed, and putting a line that says `#NO_APP` after them. This feature is mainly intend to support asm statements in compilers whose output normally does not need to be pre-processed.

2.2 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 (See Section 2.6.1 [Characters], page 10.), any whitespace means the same as exactly one space.

2.3 Comments

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

Anything from '`/*`' to the next '`*/`' inclusive is a comment.

```
/*
  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 considered a comment and is ignored. The line comment character is '`#`' on the Vax, and '`|`' on the 68020. See Chapter 7 [MachineDependent], page 28.

To be compatible with past assemblers a special interpretation is given to lines that begin with '`#`'. Following the '`#`' an absolute expression (see Chapter 5 [Expressions], page 20) is expected: this will be the logical line number of the **next** line. Then a string (See Section 2.6.1.1 [Strings], page 10.) 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`.

2.4 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 '`_.$`'. 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 begin/end-of-file. (See Chapter 4 [Symbols], page 17.)

2.5 Statements

A *statement* ends at a newline character ('`\n`') or at a semicolon ('`;`'). The newline or semicolon is considered part of the preceding statement. Newlines and semicolons within character constants are an exception: they don't 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.

You may write a statement on more than one line if you put a backslash (\) immediately in front of any newlines within the statement. When `as` reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

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

Statements begin with zero or more labels, 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 will assemble into a machine language instruction. Different versions of `as` for different computers will recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is usually a symbol immediately followed by a colon (`:`). Whitespace before a label or after a colon is OK. You may not have whitespace between a label's symbol and its colon. Labels are explained below. See Section 4.1 [Labels], page 17.

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

2.6 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.
```

2.6.1 Character Constants

There are two kinds of character constants. *Characters* stand for one character in one byte and their values 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.

2.6.1.1 Strings

A *string* is written between double-quotes. It may contain double-quotes or null characters. The way to get weird 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.

`\`*`EOF`*
: A \ followed by end-of-file erroneous. It is treated just like an end-of-file without a preceding backslash.

`\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 Un*x systems, 8 and 9 are legal digits with values 010 and 011 respectively.

`\\`
: 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 \ will give a warning, but assemble 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 BSD 4.2 `as` recognizes, and is a subset of what most C compilers recognize. If you are in doubt, don't use an escape sequence.

2.6.1.2 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 accent acute, not an accent grave. A newline (or semicolon (`;`)) immediately following an accent acute 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. GNU assumes your character code is ASCII: `'A` means 65, `'B` means 66, and so on.

2.6.2 Number Constants

`as` distinguishes 3 flavors 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 a more than 32 bits. *Flonums* are floating point numbers, described below.

2.6.2.1 Integers

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 obvious values. To denote a negative integer, use the unary operator '`-`' discussed under expressions (See Section 5.2.4 [Unops], page 20.).

2.6.2.2 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.

2.6.2.3 Flonums

A *flonum* represents a floating point number. The translation is complex: 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 the particular computer's floating point format(s) by a portion of `as` specialized to that computer.

A flonum is written by writing (in order)

- The digit '`0`'.
- A letter, to tell `as` the rest of the number is a flonum. `e` is recommended. Case is not important. (Any otherwise illegal letter will work here, but that might be changed. VAX BSD 4.2 assembler seems to allow any of '`defghDEFGH`'.)
- An optional sign: either '`+`' or '`-`'.

- An optional integer part: zero or more decimal digits.
- An optional fraction part: '`.`' followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - A letter; the exact significance varies according to the computer that executes the program. `as` accepts any letter for now. Case is not important.
  - Optional sign: either '`+`' or '`-`'.
  - One or more decimal digits.

At least one of *integer part* or *fraction part* must be present. The floating point number has the obvious value.

The computer running `as` needs no floating point hardware. `as` does all processing using integers.

3 (Sub)Segments & Relocation

Roughly, a *segment* is a range of addresses, with no gaps, with all data "in" those addresses being treated the same. For example there may be a "read only" segment.

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` will assign the final addresses the partial program occupies, so that different partial programs don't overlap. That explanation is too simple, but it will suffice to explain how `as` works.

`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 *segment*. Assigning run-time addresses to segments is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses.

An object file written by `as` has three segments, any of which may be empty. These are named *text*, *data* and *bss* segments. Within the object file, the text segment starts at address 0, the data segment follows, and the bss segment follows the data segment.

To let `ld` know which data will change when the segments 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 for each mention of an address in the object file:

- At what address in the object file does this mention of an address begin?
- How long (in bytes) is this mention?
- Which segment does the address refer to? What is the numeric value of (*address* - *start-address of segment*)?
- Is the mention of an address "Program counter relative"?

In fact, every address `as` ever thinks about is expressed as (*segment* + *offset into segment*). Further, every expression `as` computes is of this segmented nature. So *absolute expression* means an expression with segment "absolute" (See Section 3.1.1 [LdSegs], page 14.). A *pass1 expression* means an expression with segment "pass1" (See Section 3.1.2 [MythSegs], page 15.). In this document "(segment, offset)" will be written as { segment-name (offset into segment) }.

Apart from text, data and bss segments you need to know about the *absolute* segment. When `ld` mixes partial programs, addresses in the absolute segment remain unchanged. That is, address {absolute 0} is "relocated" to run-time address 0 by `ld`. Although two partial programs' data segments will not overlap addresses after linking, **by definition** their absolute segments will overlap. Address {absolute 239} in one partial program will always be the same address when the program is running as address {absolute 239} in any other partial program.

The idea of segments is extended to the *undefined* segment. Any address whose segment is unknown at assembly time is by definition rendered {undefined (something, unknown yet)}. 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 segment *undefined*.

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

3.1 Segments

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

3.1.1 ld segments

`ld` deals with just 5 kinds of segments, summarized below.

text segment
data segment

These segments hold your program bytes. `as` and `ld` treat them as separate but equal segments. Anything you can say of one segment is true of the other. When the program is running however it is customary for the text segment to be unalterable: it will contain instructions, constants and the like. The data segment of a running program is usually alterable: for example, C variables would be stored in the data segment.

bss segment

This segment contains zeroed bytes when your program begins running. It is used to hold unitialized variables or common storage. The length of each partial program's bss segment 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 segment was invented to eliminate those explicit zeros from object files.

absolute segment

Address 0 of this segment 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 don't change during relocation.

undefined segment

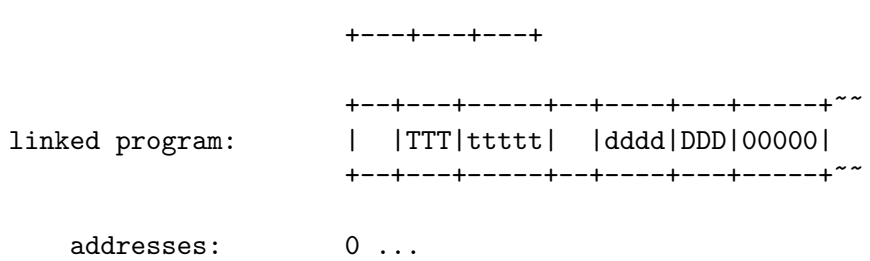
This "segment" is a catch-all for address references to objects not in the preceding segments. See the description of `a.out` for details.

An idealized example of the 3 relocatable segments follows. 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|
```



3.1.2 Mythical Segments

These segments are invented for the internal use of `as`. They have no meaning at run-time. You don't need to know about these segments except that they might be mentioned in `as`' warning messages. These segments are invented to permit the value of every expression in your assembly language program to be a segmented address.

absent segment
: An expression was expected and none was found.

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

grand segment
: A *grand number* is a bignum or a flonum, but not an integer. If a number can't be written as a C `int` constant, it is a grand number. `as` has to remember that a flonum or a bignum does not fit into 32 bits, and cannot be a primary (See Section 5.2.1 [Primary], page 20.) in an expression: this is done by making a flonum or bignum be of type "grand". This is purely for internal `as` convenience; grand segment behaves similarly to absolute segment.

pass1 segment
: The expression was impossible to evaluate in the first pass. The assembler will attempt a second pass (second reading of the source) to evaluate the expression. Your expression mentioned an undefined symbol in a way that defies the one-pass (segment `+` offset in segment) assembly process. No compiler need emit such an expression.

difference segment
: As an assist to the C compiler, expressions of the forms

  - (undefined symbol) `-` (expression)
  - (something) `-` (undefined symbol)
  - (undefined symbol) `-` (undefined symbol)

  are permitted to belong to the "difference" segment. `as` re-evaluates such expressions after the source file has been read and the symbol table built. If by that time there are no undefined symbols in the expression then the expression assumes a new segment. The intention is to permit statements like '`.word label - base_of_table`' to be assembled in one pass where both `label` and `base_of_table` are undefined. This is useful for compiling C and Algol switch statements, Pascal case statements, FORTRAN computed goto statements and the like.

3.2 Sub-Segments

Assembled bytes fall into two segments: text and data. Because you may have groups of text or data that you want to end up near to each other in the object file, `as`, allows you to use *subsegments*. Within each segment, there can be numbered subsegments with values from 0 to 8192. Objects assembled into the same subsegment will be grouped with other objects in the same subsegment when they are all put into the object file. For example, a compiler might want to store constants in the text segment, but might not want to have them intersperced 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.

Subsegments are optional. If you don't used subsegments, everything will be stored in subsegment number zero.

Each subsegment is zero-padded up to a multiple of four bytes. (Subsegments may be padded a different amount on different flavors of `as`.) Subsegments 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, `ld` *etc.* have no concept of subsegments. They just see all your text subsegments as a text segment, and all your data subsegments as a data segment.

To specify which subsegment you want subsequent statements assembled into, use a '`.text` *expression*' or a '`.data` *expression*' statement. *Expression* should be an absolute expression. (See Chapter 5 [Expressions], page 20.) 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 subsegment is text 0 anyway.
.ascii "This lives in the first text subsegment. *"
.text 1
.ascii "But this lives in the second text subsegment."
.data 0
.ascii "This lives in the data segment,"
.ascii "in the first data subsegment."
.text 0
.ascii "This lives in the first text segment,"
.ascii "immediately following the asterisk (*)."
```

Each segment has a *location counter* incremented by one for every byte assembled into that segment. Because subsegments are merely a convenience restricted to `as` there is no concept of a subsegment location counter. There is no way to directly manipulate a location counter. The location counter of the segment that statements are being assembled into is said to be the *active* location counter.

3.3 Bss Segment

The `bss` segment is used for local common variable storage. You may allocate address space in the `bss` segment, 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` segment are zeroed bytes. Addresses in the bss segment are allocated with a special statement; you may not assemble anything directly into the bss segment. Hence there are no bss subsegments.

4 Symbols

Because the linker uses symbols to link, the debugger uses symbols to debug and the programmer uses symbols to name things, symbols are a central concept. Symbols do not appear in the object file in the order they are declared. This may break some debuggers.

4.1 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.

4.2 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 (see Chapter 5 [Expressions], page 20). This is equivalent to using the `.set` directive. (See Section 6.26 [Set], page 25.)

4.3 Symbol Names

Symbol names begin with a letter or with one of ‘`$._`’. That character may be followed by any string of digits, letters, underscores and dollar signs. Case of letters is significant: `foo` is a different symbol name than `Foo`.

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

4.3.1 Local Symbol Names

Local symbols help compilers and programmers use names temporarily. There are ten *local* symbol names, which are re-used throughout the program. Their names are ‘`0`’ ‘`1`’ . . . ‘`9`’. To define a local symbol, write a label of the form *digit*`:`. To refer to the most recent previous definition of that symbol write *digit*`b`, using the same digit as when you defined the label. To refer to the next definition of a local label, write *digit*`f` where *digit* gives you a choice of 10 forward references. The ‘`b`’ stands for “backwards” and the ‘`f`’ stands for “forwards”.

Local symbols are not used by the current C compiler.

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler thinks about them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have 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 give the ‘`-L`’ switch then `as` will retain these symbols in the object file. By instructing `ld` to also retain these symbols, you may use them in debugging.

`a digit` If the label is written '`0:`' then the digit is '`0`'. If the label is written '`1:`' then the digit is '`1`'. And so on up through '`9:`'.

`control-A`
This unusual character is included so you don't accidentally invent a symbol of the same name. The character has ASCII value '`\001`'.

`an ordinal number`
This is like a serial number to keep the labels distinct. The first '`0:`' gets the number '`1`'; The 15th '`0:`' gets the number '`15`'; *etc.*. Likewise for the other labels '`1:`' through '`9:`'.

For instance, the first `1:` is named `L1^A1`, the 44th `3:` is named `L3^A44`.

4.4 Symbol Attributes

Every symbol has the attributes discussed below. The detailed definitions are in <a.out.h>.

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.

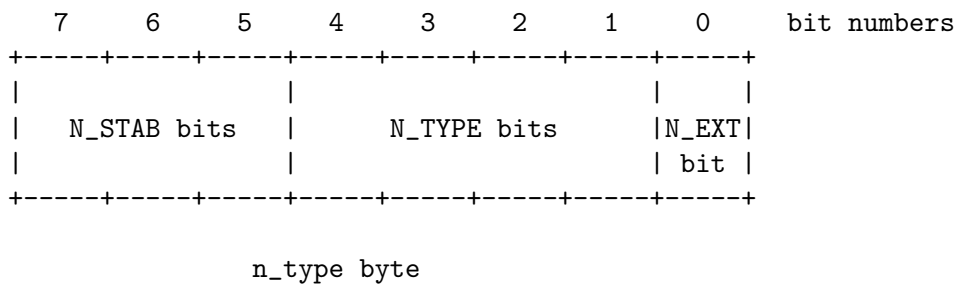
4.4.1 Value

The value of a symbol is (usually) 32 bits, the size of one C `int`. For a symbol which labels a location in the `text`, `data`, `bss` or `Absolute` segments the value is the number of addresses from the start of that segment to the label. Naturally for `text data` and `bss` segments the value of a symbol changes as `ld` changes segment 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 program, and `ld` will try to determine its value from other programs it is linked with. 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 (*i.e.* addresses). The symbol refers to the first address of the allocated storage.

4.4.2 Type

The type attribute of a symbol is 8 bits encoded in a devious way. We kept this coding standard for compatibility with older operating systems.



n_type byte

4.4.2.1 N_EXT bit

This bit is set if `ld` might need to use the symbol's value and type bits. If this bit is re-set then `ld` can ignore the symbol while linking. It is set in two cases. If the symbol is undefined, then `ld` is expected to find the symbol's value elsewhere in another program module. Otherwise the symbol has the value given, but this symbol name and value are revealed to any other programs linked in the same executable program. This second use of the `N_EXT` bit is most often done by a `.globl` statement.

4.4.2.2 N_TYPE bits

These establish the symbol's "type", which is mainly a relocation concept. Common values are detailed in the manual describing the executable file format.

4.4.2.3 N_STAB bits

Common values for these bits are described in the manual on the executable file format..

4.4.3 Desc(riptor)

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (See Section 6.8 [Desc], page 22.). A descriptor value means nothing to `as`.

4.4.4 Other

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

4.5 The Special Dot Symbol

The special symbol `.` refers to the current address that `as` is assembling into. Thus, the expression '`melvin: .long .`' will cause *melvin* to contain its own address. Assigning a value to `.` is treated the same as a `.org` pseudo-op. Thus, the expression '`.=.+4`' is the same as saying '`.space 4`'.

5 Expressions

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

5.1 Empty Expressions

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

5.2 Integer Expressions

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

5.2.1 Primaries

Primaries are symbols, numbers or subexpressions. Other languages might call primaries "arithmetic operands" but we don't want them confused with "instruction operands" of the machine language so we give them a different name.

Symbols are evaluated to yield {*segment value*} where *segment* is one of **text**, **data**, **bss**, **absolute**, or **undefined**. *value* 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 unary operator followed by an primary.

5.2.2 Operators

Operators are arithmetic marks, like `+` or `%`. Unary operators are followed by an primary. Binary operators appear between primaries. Operators may be preceded and/or followed by whitespace.

5.2.3 Unary Operators

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

`-` Hyphen. *Negation*. Two's complement negation.

`~` Tilde. *Complementation*. Bitwise not.

5.2.4 Binary Operators

Binary operators are infix. Operators are prioritized, but equal priority operators are performed left to right. Apart from '`+`' or '`-`', both primaries must be absolute, and the result is absolute, else one primary can be either undefined or pass1 and the result is pass1.

1. Highest Priority

   `*` *Multiplication*.

`/` *Division.* Truncation is the same as the C operator '`/`' of the compiler that compiled `as`.

`%` *Remainder.*

`<`
`<<` *Shift Left.* Same as the C operator '`<<`' of the compiler that compiled `as`.

`>`
`>>` *Shift Right.* Same as the C operator '`>>`' of the compiler that compiled `as`.

2. Intermediate priority

`|` *Bitwise Inclusive Or.*

`&` *Bitwise And.*

`^` *Bitwise Exclusive Or.*

`!` *Bitwise Or Not.*

3. Lowest Priority

`+` *Addition.* If either primary is absolute, the result has the segment of the other primary. If either primary is pass1 or undefined, result is pass1. Otherwise `+` is illegal.

`-` *Subtraction.* If the right primary is absolute, the result has the segment of the left primary. If either primary is pass1 the result is pass1. If either primary is undefined the result is difference segment. If both primaries are in the same segment, the result is absolute; provided that segment is one of text, data or bss. Otherwise `-` is illegal.

The sense of the rules is that you can't add or subtract quantities from two different segments. If both primaries are in one of these segments, they must be in the same segment: **text**, **data** or **bss**, and the operator must be '`-`'.

6 Assembler Directives

All assembler directives begin with a symbol that begins with a period (‘`.`’). The rest of the symbol is letters: their case does not matter.

6.1 .abort

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

6.2 .align *absolute-expression* , *absolute-expression*

Pad the location counter (in the current subsegment) to a word, longword or whatever boundary. The first expression is the number of low-order zero bits the location counter will have after advancement. For example ‘`.align 3`’ will advance 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 gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zeroed.

6.3 .ascii *strings*

Expects zero or more string literals (See Section 2.6.1.1 [Strings], page 10.) separated by commas. Assembles each string (with no automatic trailing zero byte) into consecutive addresses.

6.4 .asciz *strings*

Just like .ascii, but each string is followed by a zero byte. The ‘z’ in ‘.asciz’ stands for ‘zero’.

6.5 .byte *expressions*

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

6.6 .comm *symbol* , *length*

Declares a named common area in the bss segment. Normally `ld` reserves memory addresses for it during linking, so no partial program defines the location of the symbol. Tell `ld` that it must be at least *length* bytes long. `ld` will allocate space that is at least as long as the longest `.comm` request in any of the partial programs linked. *length* is an absolute expression.

6.7 .data *subsegment*

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

6.8 .desc *symbol*, *absolute-expression*

Set `n_desc` of the symbol to the low 16 bits of *absolute-expression*.

6.9 .double *flonums*

Expect zero or more flonums, separated by commas. Assemble floating point numbers. The exact kind of floating point numbers emitted depends on what computer `as` is assembling for. See the machine-specific part of the manual for the machine the assembler is running on for more information.

6.10 .file *string*

Tells `as` that we are about to start a new logical file. *String* is the new file name. An empty file name is OK, but you must still give the quotes: `""`. This statement may go away in future: it is only recognized to be compatible with old `as` programs.

6.11 .fill *repeat* , *size* , *value*

result, *size* and *value* are absolute expressions. Emit *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.

6.12 .float *flonums*

Expect zero or more flonums, separated by commas. Assemble floating point numbers. The exact kind of floating point numbers emitted depends on what computer `as` is assembling for. See the machine-specific part of the manual for the machine the assembler is running on for more information.

6.13 .gdbbeg *absolute-expression*

(This pseudo-op may go away without warning.) *Absolute-expression* must be at least zero. `as` will remember that a block numbered *absolute-expression* began where the location count is when this statement is read.

6.14 .gdbblock *block-number* , *offset*

(This pseudo-op may go away without warning.) *Block-number* is a `gdb` block number, at least zero, an absolute expression. *Offset* is an offset into the `gdb` symbolic file named in the '`-G`' switch; an absolute expression; the lowest offset written by this directive. Two C `int`s are written in the symbolic file: first the object-file address of the `.gdbbeg` statement of *block number*; then the object-file address of the `.gdbend` statement of *block number*.

6.15 .gdbend *absolute-expression*

(This pseudo-op may go away without warning.) *Absolute-expression* must be at least zero. `as` will remember that a block numbered *absolute-expression* ended where the location count is when this statement is read.

6.16 .gdbsym *symbol* , *offset*

(This pseudo-op may go away without warning.) If the '`-G`' switch named a file of `gdb` symbolic information then the `n_value` of *symbol* is written as a C `int` starting at *offset* in the symbolic file. *Offset* is an absolute expression. *Symbol* may be defined after the `.gdbsym` statement.

6.17 .global *symbol*

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* will take its attributes from a symbol of the same name from another partial program it is linked with.

This is done by setting the `N_EXT` bit of that symbol's `n_type` to 1.

6.18 .int *expressions*

Expect zero or more *expressions*, of any segment, separated by commas. For each expression, emit a 32-bit number that will, at run time, be the value of that expression. The byte order of the expression depends on what kind of computer will run the program.

6.19 .lcomm *symbol* , *length*

Reserve *length* (an absolute expression) bytes for a local common and denoted by *symbol*, whose segment and value are those of the new local common. The addresses are allocated in the `bss` segment, so at run-time the bytes will start off zeroed. *Symbol* is not declared global (See Section 6.17 [Global], page 24.), so is normally not visible to `ld`.

6.20 .line *logical line number*

This tells `as` to change the logical line number. *logical line number* is an absolute expression. The next line will have that logical line number. So any other statements on the current line (after a `;`) will be reported as on logical line number *logical line number* - 1. One day this directive will be unsupported: it is used only for compatibility with existing assembler programs.

6.21 .long *expressions*

The same as '`.int`', see Section 6.18 [Int], page 24.

6.22 .lsym *symbol*, *expression*

Create a new symbol named *symbol*, but do not put it in the hash table, ensuring it cannot be referenced by name during the rest of the assembly. Set the attributes of the symbol to

be the same as the expression value. `n_other` = `n_desc` = 0. `n_type` = (whatever segment the expression has); the `N_EXT` bit of `n_type` is zero. `n_value` = (expression's value).

6.23 .octa *bignums*

Expect zero or more bignums, separated by commas. For each bignum, emit an 16-byte (**octa**-word) integer.

6.24 .org *new-lc* , *fill*

This will advance the location counter of the current segment to *new-lc*. *new-lc* is either an absolute expression or an expression with the same segment as the current subsegment. That is, you can't use `.org` to cross segments. Because `as` tries to assemble programs in one pass *new-lc* must be defined. If you really detest this restriction we eagerly await a chance to share your improved assembler. To be compatible with former assemblers, if the segment of *new-lc* is absolute then we pretend the segment of *new-lc* is the same as the current subsegment.

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

If the location counter (of the current subsegment) 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.

6.25 .quad *bignums*

Expect zero or more bignums, separated by commas. For each bignum, emit an 8-byte (**quad**-word) integer. If the bignum won't fit in a quad-word, warn; just take the lowest order 8 bytes of the bignum.

6.26 .set *symbol*, *expression*

Set the value of *symbol* to expression. This will change `n_value` and `n_type` to conform to the *expression*.

It is OK to `.set` a symbol many times in the same assembly. If the expression's segment is unknowable during pass 1, a second pass over the source program will be forced. The second pass is currently not implemented. `as` will abort with an error message if one is required.

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

6.27 .short *expressions*

The same as '`.word`'. See Section 6.31 [Word], page 26.

6.28 .space *size* , *fill*

Emit *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.

6.29 .stabd, .stabn, .stabs

There are three directives that begin `.stab...`. All emit symbols, for use by symbolic debuggers. The symbols are not entered in `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. Old debuggers used to code arbitrarily complex structures into symbol names using this technique.

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

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

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

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

If a warning is detected while reading the `.stab...` statement the symbol has probably already been created and you will 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 `n_value` is set to the location counter, relocatably. When your program is linked, the value of this symbol will be where the location counter was 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*

6.30 .text *subsegment*

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

6.31 .word *expressions*

Expect zero or more *expressions*, of any segment, separated by commas. For each expression, emit a 16-bit number that will, at run time, be the value of that expression. The byte order of the expression depends on what kind of computer will run the program.

6.32 Deprecated Directives

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

`.abort`

`.file`

`.line`

7 Machine Dependent Features

7.1 Vax

7.1.1 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 floating point formats generated by directives are these.

`.float`
`.ffloat` `F` format floating point numbers.

`.double`
`.dfloat` `D` format floating point numbers.

`.gfloat` `G` format floating point numbers.

`.hfloat` `H` format floating point numbers.

7.1.2 Machine Directives

The Vax version of the assembler supports four pseudo-ops for generating Vax floating point constants.

7.1.2.1 .dfloat *flonums*

Expect zero or more flonums, separated by commas. Assemble Vax d format floating point constants.

7.1.2.2 .ffloat *flonums*

Expect zero or more flonums, separated by commas. Assembles Vax f format floating point constants.

7.1.2.3 .gfloat *flonums*

Expect zero or more flonums, separated by commas. Assembles Vax g format floating point constants.

7.1.2.4 .hfloat *flonums*

Expect zero or more flonums, separated by commas. Assembles Vax h format floating point constants.

7.1.3 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 un*x assemblers we know of.

7.1.4 Branch Improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that will reach 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 don't need this feature, don't use 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 ...
```

`j`*COND* *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)
```
bUNCOND foo ; brw ... ; foo:
```

(long displacement)
```
bUNCOND foo ; jmp ... ; foo:
```

`jacb`*X* *X* may be one of `b d f g h l w`.

(word displacement)
```
OPCODE ...
```

(long displacement)
```
OPCODE ..., foo ; brb bar ; foo: jmp ... ; bar:
```

`jaob`*YYY* *YYY* may be one of `lss leq`.

`jsob`*ZZZ* *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:

7.1.5 operands

The immediate character is '`$`' for Un*x compatibility, not '`#`' as DEC writes it.

The indirect character is '`*`' for Un*x compatibility, not '`@`' as DEC writes it.

The displacement sizing character is '`` ` ``' (an accent grave) for Un*x 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`. Any case of letters will do.

For instance

```
tstb *w`$4(r5)
```

Any expression is permitted in an operand. Operands are comma separated.

7.2 68020

7.2.1 Syntax

The 68020 version of `as` uses syntax similar to the Sun assembler. Size modifieres are appended directly to the end of the opcode without an intervening period. Thus, '`move.l`' is written '`movl`', etc. Explicit size modifiers for branch instructions are ignored; `as` automatically picks the smallest size that will reach the destination.

If `as` is compiled with SUN_ASM_SYNTAX defined, it will also allow Sun-style local labels of the form '`1$`' through '`$9`'.

In the following table *apc* stands for any of the address registers ('`a0`' through '`a7`'), nothing, ('`'`'), the Program Counter ('`pc`'), or the zero-address relative to the program counter ('`zpc`').

The following addressing modes are understood:

Immediate
'`#digits`'

Data Register
'`d0`' through '`d7`'

Address Register
'`a0`' through '`a7`'

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

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

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

Indirect Plus Offset
'*`apc`*`@(`*`digits`*`)`'

Index '*`apc`*`@(`*`digits`*`,`*`register`*`:`*`size`*`:`*`scale`*`)`' or '*`apc`*`@(`*`register`*`:`*`size`*`:`*`scale`*`)`'

Postindex '*`apc`*`@(`*`digits`*`)@(`*`digits`*`,`*`register`*`:`*`size`*`:`*`scale`*`)`' or '*`apc`*`@(`*`digits`*`)@(`*`register`*`:`*`size`*`:`*`scale`*`)`'

Preindex '*`apc`*`@(`*`digits`*`,`*`register`*`:`*`size`*`:`*`scale`*`)@(`*`digits`*`)`' or '*`apc`*`@(`*`register`*`:`*`size`*`:`*`scale`*`)@(`*`digits`*`)`'

Memory Indirect
'*`apc`*`@(`*`digits`*`)@(`*`digits`*`)`'

Absolute '*`symbol`*', or '*`digits`*', or either of the above followed by '`:b`', '`:w`', or '`:l`'.

7.2.2 Floating Point

The floating point code is not well tested, and may have subtle bugs in it.

X and P format floating literals are not supported. Feel free to add the code yourself.

The floating point formats generated by directives are these.

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

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

7.2.3 Machine Directives

In order to be compatible with the Sun assembler the 68020 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 identical to a `.align 2` directive.

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

7.2.4 Opcodes

Danger: Several bugs have been found in the opcode table (and fixed). More bugs may exist. The floating point code is especially untested.

The assembler automatically chooses the proper size for branch instructions. Any attempt to force a short displacement will be silently ignored.

The immediate character is '`#`' for Sun compatibility. The line-comment character is '`|`'. 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.

7.3 32xxx

`as` for the 32xxx computer family has not been written yet.

7.4 Intel 80386

7.4.1 AT&T Syntax versus Intel Syntax

In order to maintain compatibility with the output of `GCC`, `as` supports AT&T System V/386 assembler syntax. This is quite different from Intel syntax. We mention these differences because almost all 80386 documents used only 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 Un*x assemblers.
- In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of '`b`', '`w`', and '`l`' specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixes memory operands (*not* the opcodes themselves) with '`byte ptr`', '`word ptr`', and '`dword 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 $segment, $offset`' in AT&T syntax; the Intel syntax is '`call/jmp far segment: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 segment programs. Un*x style systems expect all programs to be single segments.

7.4.2 Opcode Naming

Opcode names are suffixed with one character modifiers which specify the size of operands. The letters '`b`', '`w`', and '`l`' specify byte, word, and long operands. If no suffix is specified by an instruction and it contains no memory operands 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 Un*x assembler which assumes that a missing opcode suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the opcode suffix.)

Almost all opcodes 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 opcode 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 opcode 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), and '`wl`' (from word to long).

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`',

are called '`cbtw`', '`cwtl`', '`cwtd`', and '`cltd`' 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.

7.4.3 Register Naming

Register operands are always prefixes 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 segment registers '`%cs`' (code segment), '`%ds`' (data segment), '`%ss`' (stack segment), '`%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)`'.

7.4.4 Opcode Prefixes

Opcode prefixes are used to modify the following opcode. They are used to repeat string instructions, to provide segment overrides, to perform bus lock operations, and to give operand and address size (16-bit operands are specified in an instruction by prefixing what would normally be 32-bit operands with a "operand size" opcode prefix). Opcode prefixes are usually given as single-line instructions with no operands, and must directly precede the instruction they act upon. For example, the '`scas`' (scan string) instruction is repeated with:

```
repne
scas
```

Here is a list of opcode prefixes:

- Segment override prefixes '`cs`', '`ds`', '`ss`', '`es`', '`fs`', '`gs`'. These are automatically added by specifying using the *segment*:*memory-operand* form for memory references.
- Operand/Address size prefixes '`data16`' and '`addr16`' change 32-bit operands/addresses into 16-bit operands/addresses. Note that 16-bit addressing modes (i.e. 8086 and 80286 addressing modes) are not supported (yet).
- 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.

7.4.5 Memory References

An Intel syntax indirect memory reference of the form

segment`:[`*base* `+` *index*`*`*scale* `+` *disp*`]`

is translated into the AT&T syntax

segment`:`*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. *segment* specifies the optional segment register for the memory operand, and may override the default segment register (see a 80386 manual for segment register defaults). Note that segment overrides in AT&T syntax *must* have be preceded by a '`%`'. If you specify a segment override which coincides with the default segment register, `as` will *not* output any segment register override prefixes to assemble the given instruction. Thus, segment overrides can be specified to emphasize which segment 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`'. *segment* is missing, and the default segment 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 segment 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 segment register *segment* being '`%gs`'.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with '`*`'. If no '`*`' is specified, `as` will always choose PC relative addressing for jump/call labels.

Any instruction that has a memory operand *must* specify its size (byte, word, or long) with an opcode suffix ('`b`', '`w`', or '`l`', respectively).

7.4.6 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 (32-bit) displacement is used. We do not support word (16-bit) displacement jumps (i.e. prefixing the jump instruction with the '`addr16`' opcode prefix), since the 80386 insists upon masking '`%eip`' to 16 bits after the word displacement is added.

Note that the '`jcxz`', '`jecxz`', '`loop`', '`loopz`', '`loope`', '`loopnz`' and '`loopne`' instructions only come in byte displacements, so that it is possible that use of these instructions (`GCC` does not use them) will cause the assembler to print an error message (and generate 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:
```

7.4.7 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 opcode suffix and a constructor associated with it. Opcode suffixes specify operand's data types. 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 opcode suffixes '`s`', '`l`', and '`t`'. '`t`' stands for temporary real, and that the 80387 only supports this format via the '`fldt`' (load temporary real to stack top) and '`fstpt`' (store temporary 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 opcode suffixes are '`s`' (single), '`l`' (long), and '`q`' (quad). As with the temporary 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 do not require opcode suffixes, so that '`fst %st, %st(1)`' is equivalent to '`fstl %st, %st(1)`'.

Since the 80387 automatically synchronizes with the 80386 '`fwait`' instructions are almost never needed (this is not the case for the 80286/80287 and 8086/8087 combinations). Therefore, `as` supresses the '`fwait`' instruction whenever it is implicitly selected by one of the '`fn...`' instructions. For example, '`fsave`' and '`fnsave`' are treated identically. In

general, all the '`fn...`' instructions are made equivalent to '`f...`' instructions. If '`fwait`' is desired it must be explicitly coded.

7.4.8 Notes

There is some trickery concerning the '`mul`' and '`imul`' instructions that deserves mention. The 16-, 32-, and 64-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`'.

8 Maintaining the Assembler

[[this chapter is still being built]]

8.1 Design

We had these goals, in descending priority:

Accuracy. For every program composed by a compiler, `as` should emit "correct" code. This leaves some latitude in choosing addressing modes, order of `relocation_info` structures in the object file, *etc.*

Speed, for usual case.
By far the most common use of `as` will be assembling compiler emissions.

Upward compatibility for existing assembler code.
Well ... we don't support bit fields but everything else seems to be upward compatible. Bit fields could be implemented if someone really cared.

Readability.
The code should be maintainable with few surprises.

We assumed that disk I/O was slow and expensive while memory was fast and access to memory was cheap. We expect the in-memory data structures to be less than 10 times the size of the emitted object file. (Contrast this with the C compiler where in-memory structures might be 100 times object file size!) This suggests:

- Try to read the source file from disk only one time. For other reasons, we do keep the entire source file in memory during assembly so this is not a problem. Also the assembly algorithm should only scan the source text once if the compiler composed the text according to a few simple rules.
- Emit the object code bytes only once. Don't store values and then backpatch later.
- Build the object file in memory and do direct writes to disk of large buffers.

RMS suggested a one-pass algorithm which seems to work well. By not parsing text during a second pass considerable time is saved on large programs (*e.g.* the sort of C program `yacc` would emit).

It happened that the data structures needed to emit relocation information to the object file were neatly subsumed into the data structures that do backpatching of addresses after pass 1.

Many of the functions began life as re-usable modules, loosely connected. RMS changed this to gain speed. For example, input parsing routines which used to work on pre-sanitized strings now must parse raw data. Hence they have to import knowledge of the assemblers' comment conventions *etc.*

8.2 Deprecated Feature(?)s

We have stopped supporting some features:

- `.org` statements must have **defined** expressions.
- VAX Bit fields (`:` operator) are entirely unsupported.

It might be a good idea to not support these features in a future release:

- `#` should begin a comment, even in column 1.
- Why support the logical line & file concept any more?
- *`.gdb...`* directives will be abandoned in favor of *`.stab...`* directives.
- Subsegments are a good candidate for flushing. Depends on which compilers need them I guess.

8.3 Bugs, Ideas, Further Work

Clearly the major improvement is DON'T USE A TEXT-READING ASSEMBLER for the back end of a compiler. It is much faster to interpret binary gobbledygook from a compiler's tables than to ask the compiler to write out human-readable code just so the assembler can parse it back to binary.

Assuming you use `as` for human written programs: here are some ideas:

- Document (here) `APP`.
- Take advantage of knowing no spaces except after opcode to speed up `as`. (Modify `app.c` to flush useless spaces: only keep space/tabs at begin of line or between 2 symbols.)
- Put pointers in this documentation to `a.out` documentation.
- Split the assembler into parts so it can gobble direct binary from *e.g.* `cc`. It is silly for`cc` to compose text just so `as` can parse it back to binary.
- Rewrite hash functions: I want a more modular, faster library.
- Clean up LOTS of code.
- Include all the non-`.c` files in the maintenance chapter.
- Document flonums.
- Implement flonum short literals.
- Change all talk of expression operands to expression quantities, or perhaps to expression primaries.
- Implement pass 2.
- Whenever a `.text` or `.data` statement is seen, we close of the current frag with an imaginary `.fill 0`. This is because we only have one obstack for frags, and we can't grow new frags for a new subsegment, then go back to the old subsegment and append bytes to the old frag. All this nonsense goes away if we give each subsegment its own obstack. It makes code simpler in about 10 places, but nobody has bothered to do it because C compiler output rarely changes subsegments (compared to ending frags with relaxable addresses, which is common).

8.4 Sources

Here is a list of the source files in the `as` directory.

`app.c` The pre-processing phase, which deletes comments, handles whitespace, etc. This was recently re-written, since app used to be a separate program, but RMS wanted it to be inline.

`append.c` A subroutine to append a string to another string returning a pointer just after the last `char` appended. (JF: All these little routines should probably all be put in one file.)

`as.c` Main program of the assembler `as`.

`expr.c` A branch office of `read.c`. Understands expressions, primaries. Inside `as`, primaries are called (expression) *operands*. This is confusing, because we also talk (elsewhere) about instruction *operands*. Also, expression operands are called *quantities* explicitly to avoid confusion with instruction operands. What a mess.

`frags.c` Implements the **frag** concept. Without frags, finding the right size for branch instructions would be a lot harder.

`gdb_blocks.c`
Implement `.gdbbeg`, `.gdbend`, `.gdbblock` statements. This file should go away when '`-G`' is flushed.

`gdb_file.c`
Operating system dependent functions to read the file named in a '`-G`' switch. This file should go away someday.

`gdb_symbols.c`
Implement the `.gdbsym` statement. Remembers all `.gdbsym` statements then executes them after assembly when gdb symbols are being built. This file should go away someday.

`gdb.c` Some more functions for the GDB dependent stuff. This file should go away someday.

`hash.c` The symbol table, opcode table *etc.* hashing functions.

`hex_value.c`
Table of values of digits, for use in atoi() type functions. Could probably be flushed by using calls to strtol(), or something similar.

`input-file.c`
Operating system dependent source file reading routines. Since error messages often say where we are in reading the source file, they live here too. Since Gas is intended to run under GNU and UN*X only, this might be worth flushing. Anyway, almost all C compilers support stdio.

`input-scrub.c`
Deals with calling the pre-processor (if needed) and feeding the chunks back to the rest of the assembler the right way.

`messages.c`
Operating system independent parts of fatal and warning message reporting.

`output-file.c`
Operating system dependent functions that write an object file for `as`. See `input-file.c` above.

`read.c` Implements all the directives of `as`. Also passing input lines to the machine dependent part of the assembler.

`strstr.c` A C library function that isn't in my C library yet.

`subsegs.c`
Implements subsegments.

`symbols.c`
Implements symbols.

`write.c` Operating system independent functions to emit an object file for `as`.

`xmalloc.c`
Implements `malloc()` or bust. Should be combined into some other file somewhere. (misc.c?)

`xrealloc.c`
Implements `realloc()` or bust. See `xmalloc.c`.

`atof-generic.c`
The following files were taken from a machine-independent subroutine library for manipulating floating point numbers and very large integers.

`atof-generic.c` turns a string into a flonum internal format floating-point number.

`flonum-const.c`
Some potentially useful floating point numbers in flonum format.

`flonum-copy.c`
Copies a flonum.

`flonum-multip.c`
Multiplies two flonums together.

`bignum-copy.c`
Copies a bignum.

Here is a table of all the machine-specific files (this includes both source and header files). Typically, there is a *machine*.c file, a *machine*-opcode.h file, and an atof-*machine*.c file. The *machine*-opcode.h file should be identical to the one used by gdb (which uses it for disassembly.)

`m-generic.h`
generic 68020 header file. To be linked to m68k.h on a non-sun3, non-hpux system.

`m-sun3.h` 68020 header file for Sun3 workstations. To be linked to m68k.h before compiling on a Sun3 system. This also works (somewhat) on a sun2 system, if you call the assembler with '`-m68010`'.

`m-hpux.h` 68020 header file for a HPUX (system 5?) box. Which box, which version of HPUX, etc? I don't know.

`m68k.h` A hard- or symbolic- link to either m-generic.h, m-hpux.h or m-sun3.h depending on which kind of 68020 you are compiling for.

`m68k-opcode.h`
Opcode table for 68020. Should be identical to the one used by `gdb`, but may contain more mnemonics.

`pmmu.h` Information for the M68851 Memory-managment-unit which is a companion chip to the 68020. 68851 support can be optionally compiled into the assembler. Check the code for details.

`m68k.c` All the mc68020 code, in one huge, slow-to-compile file.

`atof-m68k.c`
Turns a flonum into a 68020 literal constant.

`vax-inst.h`
Vax specific file for describing Vax operands and other Vax-ish things.

`vax-opcode.h`
Vax opcode table.

`vax.c` Vax specific parts of `as`. Also includes the former files `vax-ins-parse.c`, `vax-reg-parse.c` and `vip-op.c`.

`atof-vax.c`
Turns a flonum into a Vax constant.

Here is a list of the header files in the source directory. (Warning: This section may not be very accurate. I didn't write the header files; I just report them.) Also note that I think many of these header files could be cleaned up or eliminated.

`a.out.h` Describes the structures used to create the binary header data inside the object file. Perhaps we should use the one in `/usr/include`?

`as.h` Defines all the globally useful things, and pulls in <stdio.h> and <assert.h>.

`bignum.h` Macros useful for dealing with bignums.

`expr.h` Structure and macros for dealing with expression()

`flonum.h` Structure for dealing with floating point numbers. Includes bignum.h

`frags.h` Macro for appending a byte to the current frag.

`hash.h` Structures and function definitions for the hashing functions.

`input-file.h`
Function headers for the input-file.c functions.

`md.h` structures and function headers for things defined in the machine dependent part of the assembler.

`obstack.h`
GNU systemwide include file for manipulating obstacks. Since nobody is running under real GNU yet, we include this file.

`read.h` Macros and function headers for reading in source files.

`struct-symbol.h`
: Structure definition and macros for dealing with the gas internal form of a symbol.

`subsegs.h`
: structure definition for dealing with the numbered subsegments of the text and data segments.

`symbols.h`
: Macros and function headers for dealing with symbols.

`write.h`
: Structure for doing segment fixups.

9 Teaching the Assembler about a New Machine

This chapter describes the steps required in order to make the assembler work with another machine's assembly language. This chapter is not complete, and only describes the steps in the broadest terms. You should look at the source for the currently supported machine in order to discover some of the details that aren't mentioned here.

You should create a new file called *machine*`.c`, and add the appropriate lines to the file `Makefile` so that you can compile your new version of the assembler. This should be straighforward; simply add lines similar to the ones there for the four current versions of the assembler.

If you want to be compatable with GDB, (and the current machine-dependent versions of the assembler), you should create a file called *machine*`-opcode.h` which should contain all the information about the names of the machine instructions, their opcodes, and what addressing modes they support. If you do this right, the assembler and GDB can share this file, and you'll only have to write it once.

9.1 Functions You will Have to Write

Your file *machine*`.c` should contain definitions for the following functions and variables. It will need to include some header files in order to use some of the structures defined in the machine-independent part of the assembler. The needed header files are mentioned in the descriptions of the functions that will need them.

`long omagic;`
: This long integer holds the value to place at the beginning of the `a.out` file. It is usually '`OMAGIC`', except on machines that store additional information in the magic-number.

`char comment_chars[];`
: This character array holds the values of the characters that start a comment anywhere in a line. Comments are stripped off automatically by the machine independent part of the assembler. Note that the '`/*`' will always start a comment, and that only '`*/`' will end a comment started by '`*/`'.

`char line_comment_chars[];`
: This character array holds the values of the chars that start a comment only if they are the first (non-whitespace) character on a line. If the character '`#`' does not appear in this list, you may get unexpected results. (Various machine-independent parts of the assembler treat the comments '`#APP`' and '`#NO_APP`' specially, and assume that lines that start with '`#`' are comments.)

`char EXP_CHARS[];`
: This character array holds the letters that can separate the mantissa and the exponent of a floating point number. Typical values are '`e`' and '`E`'.

`char FLT_CHARS[];`
: This character array holds the letters that–when they appear immediately after a leading zero–indicate that a number is a floating-point number. (Sort of how 0x indicates that a hexadecimal number follows.)

`pseudo_typeS md_pseudo_table[];`

(*pseudo_typeS* is defined in `md.h`) This array contains a list of the machine_dependent pseudo-ops the assembler must support. It contains the name of each pseudo op (Without the leading '`.`'), a pointer to a function to be called when that pseudo-op is encountered, and an integer argument to be passed to that function.

`void md_begin(void)`

This function is called as part of the assembler's initialization. It should do any initialization required by any of your other routines.

`int md_parse_option(char **optionPTR, int *argcPTR, char ***argvPTR)`

This routine is called once for each option on the command line that the machine-independent part of `as` does not understand. This function should return non-zero if the option pointed to by *optionPTR* is a valid option. If it is not a valid option, this routine should return zero. The variables *argcPTR* and *argvPTR* are provided in case the option requires a filename or something similar as an argument. If the option is multi-character, *optionPTR* should be advanced past the end of the option, otherwise every letter in the option will be treated as a separate single-character option.

`void md_assemble(char *string)`

This routine is called for every machine-dependent non-pseudo-op line in the source file. It does all the real work involved in reading the opcode, parsing the operands, etc. *string* is a pointer to a null-terminated string, that comprises the input line, with all excess whitespace and comments removed.

`void md_number_to_chars(char *outputPTR,long value,int nbytes)`

This routine is called to turn a C long int, short int, or char into the series of bytes that represents that number on the target machine. *outputPTR* points to an array where the result should be stored; *value* is the value to store; and *nbytes* is the number of bytes in 'value' that should be stored.

`void md_number_to_imm(char *outputPTR,long value,int nbytes)`

This routine is identical to `md_number_to_chars`, except on NS32K machines.

`void md_number_to_disp(char *outputPTR,long value,int nbytes)`

This routine is identical to `md_number_to_chars`, except on NS32K machines.

`void md_number_to_field(char *outputPTR,long value,int nbytes)`

This routine is identical to `md_number_to_chars`, except on NS32K machines.

`void md_ri_to_chars(struct relocation_info *riPTR,ri)`

(`struct relocation_info` is defined in `a.out.h`) This routine emits the relocation info in *ri* in the appropriate bit-pattern for the target machine. The result should be stored in the location pointed to by *riPTR*.

`char *md_atof(char type,char *outputPTR,int *sizePTR)`

This routine turns a series of digits into the appropriate internal representation for a floating-point number. *type* is a character from *FLT_CHARS[]* that describes what kind of floating point number is wanted; *outputPTR* is a pointer to an array that the result should be stored in; and *sizePTR* is a pointer to an

integer where the size (in bytes) of the result should be stored. This routine should return an error message, or an empty string (not (char *)0) for success.

`int md_short_jump_size;`
: This variable holds the (maximum) size in bytes of a short (16 bit or so) jump created by `md_create_short_jump()`. This variable is used as part of the broken-word function, and isn't needed if the assembler is compiled with '`-DWORKING_DOT_WORD`'.

`int md_long_jump_size;`
: This variable holds the (maximum) size in bytes of a long (32 bit or so) jump created by `md_create_long_jump()`. This variable is used as part of the broken-word function, and isn't needed if the assembler is compiled with '`-DWORKING_DOT_WORD`'.

`void md_create_short_jump(char *resultPTR,long from_addr,`
`long to_addr,fragS *frag,symbolS *to_symbol)` This function creates (stores) a jump from *from_addr* to *to_addr* in the array of bytes pointed to by *resultPTR*. If this uses a type of jump that must be relocated, this function should call `fix_new()` with *frag* and *to_symbol*. The jump created by this function may be smaller than *md_short_jump_size*, but it must never create a larger one. This function is used as part of the broken-word function, and isn't needed if the assembler is compiled with '`-DWORKING_DOT_WORD`'.

`void md_create_long_jump(char *ptr,long from_addr,`
`long to_addr,fragS *frag,symbolS *to_symbol)` This function is similar to the previous function, `md_create_short_jump()`, except that it creates a long jump instead of a short one. This function is used as part of the broken-word function, and isn't needed if the assembler is compiled with '`-DWORKING_DOT_WORD`'.

`int md_estimate_size_before_relax(fragS *fragPTR,int segment_type)`
: This function does the initial setting up for relaxation. This includes forcing references to still-undefined symbols to the appropriate addressing modes.

`relax_typeS md_relax_table[];`
: (relax_typeS is defined in md.h) This array describes the various machine dependent states a frag may be in before relaxation. You will need one group of entries for each type of addressing mode you intend to relax.

`void md_convert_frag(fragS *fragPTR)`
: (*fragS* is defined in `as.h`) This routine does the required cleanup after relaxation. Relaxation has changed the type of the frag to a type that can reach its destination. This function should adjust the opcode of the frag to use the appropriate addressing mode. *fragPTR* points to the frag to clean up.

`void md_end(void)`
: This function is called just before the assembler exits. It need not free up memory unless the operating system doesn't do it automatically on exit. (In which case you'll also have to track down all the other places where the assembler allocates space but never frees it.)

9.2 External Variables You will Need to Use

You will need to refer to or change the following external variables from within the machine-dependent part of the assembler.

`extern char flagseen[];`
: This array holds non-zero values in locations corresponding to the options that were on the command line. Thus, if the assembler was called with '`-W`', *flagseen['W']* would be non-zero.

`extern fragS *frag_now;`
: This pointer points to the current frag–the frag that bytes are currently being added to. If nothing else, you will need to pass it as an argument to various machine-independent functions. It is maintained automatically by the frag-manipulating functions; you should never have to change it yourself.

`extern LITTLENUM_TYPE generic_bignum[];`
: (*LITTLENUM_TYPE* is defined in `bignum.h`. This is where *bignums*–numbers larger than 32 bits–are returned when they are encountered in an expression. You will need to use this if you need to implement pseudo-ops (or anything else) that must deal with these large numbers. `Bignums` are of `segT SEG_BIG` (defined in `as.h`, and have a positive `X_add_number`. The `X_add_number` of a `bignum` is the number of `LITTLENUMS` in *generic_bignum* that the number takes up.

`extern FLONUM_TYPE generic_floating_point_number;`
: (*FLONUM_TYPE* is defined in `flonum.h`. The is where *flonums*–floating-point numbers within expressions–are returned. `Flonums` are of `segT SEG_BIG`, and have a negative `X_add_number`. `Flonums` are returned in a generic format. You will have to write a routine to turn this generic format into the appropriate floating-point format for your machine.

`extern int need_pass_2;`
: If this variable is non-zero, the assembler has encountered an expression that cannot be assembled in a single pass. Since the second pass isn't implemented, this flag means that the assembler is punting, and is only looking for additional syntax errors. (Or something like that.)

`extern segT now_seg;`
: This variable holds the value of the segment the assembler is currently assembling into.

9.3 External functions will you need

You will find the following external functions useful (or indispensable) when you're writing the machine-dependent part of the assembler.

`char *frag_more(int bytes)`
: This function allocates *bytes* more bytes in the current frag (or starts a new frag, if it can't expand the current frag any more.) for you to store some object-file bytes in. It returns a pointer to the bytes, ready for you to store data in.

`void fix_new(fragS *frag, int where, short size, symbolS *add_symbol, symbolS *sub_symbol, long offset, int pcrel)`

This function stores a relocation fixup to be acted on later. *frag* points to the frag the relocation belongs in; *where* is the location within the frag where the relocation begins; *size* is the size of the relocation, and is usually 1 (a single byte), 2 (sixteen bits), or 4 (a longword). The value *add_symbol* − *sub_symbol* + *offset*, is added to the byte(s) at *frag->literal[where]*. If *pcrel* is non-zero, the address of the location is subtracted from the result. A relocation entry is also added to the `a.out` file. *add_symbol*, *sub_symbol*, and/or *offset* may be NULL.

`char *frag_var(relax_stateT type, int max_chars, int var, relax_substateT subtype, symbolS *symbol, char *opcode)` This function creates a machine-dependent frag of type *type* (usually `rs_machine_dependent`). *max_chars* is the maximum size in bytes that the frag may grow by; *var* is the current size of the variable end of the frag; *subtype* is the sub-type of the frag. The sub-type is used to index into *md_relax_table[]* during `relaxation`. *symbol* is the symbol whose value should be used to when relax-ing this frag. *opcode* points into a byte whose value may have to be modified if the addressing mode used by this frag changes. It typically points into the *fr_literal[]* of the previous frag, and is used to point to a location that `md_convert_frag()`, may have to change.

`void frag_wane(fragS *fragPTR)`

This function is useful from within `md_convert_frag`. It changes a frag to type rs_fill, and sets the variable-sized piece of the frag to zero. The frag will never change in size again.

`segT expression(expressionS *retval)`

(*segT* is defined in `as.h`; *expressionS* is defined in `expr.h`) This function parses the string pointed to by the external char pointer *input_line_pointer*, and returns the segment-type of the expression. It also stores the results in the *expressionS* pointed to by *retval*. *input_line_pointer* is advanced to point past the end of the expression. (*input_line_pointer* is used by other parts of the assembler. If you modify it, be sure to restore it to its original value.)

`as_warn(char *message,...)`

If warning messages are disabled, this function does nothing. Otherwise, it prints out the current file name, and the current line number, then uses `fprintf` to print the *message* and any arguments it was passed.

`as_fatal(char *message,...)`

This function prints out the current file name and line number, prints the word '`FATAL:`', then uses `fprintf` to print the *message* and any arguments it was passed. Then the assembler exits. This function should only be used for serious, unrecoverable errors.

`void float_const(int float_type)`

This function reads floating-point constants from the current input line, and calls `md_atof` to assemble them. It is useful as the function to call for the pseudo-ops '`.single`', '`.double`', '`.float`', etc. *float_type* must be a character from *FLT_CHARS*.

`void demand_empty_rest_of_line(void);`
: This function can be used by machine-dependent pseudo-ops to make sure the rest of the input line is empty. It prints a warning message if there are additional characters on the line.

`long int get_absolute_expression(void)`
: This function can be used by machine-dependent pseudo-ops to read an absolute number from the current input line. It returns the result. If it isn't given an absolute expression, it prints a warning message and returns zero.

9.4 The concept of Frags

This assembler works to optimize the size of certain addressing modes. (e.g. branch instructions) This means the size of many pieces of object code cannot be determined until after assembly is finished. (This means that the addresses of symbols cannot be determined until assembly is finished.) In order to do this, `as` stores the output bytes as *frags*.

Here is the definition of a frag (from `as.h`)

```
struct frag
{
        long int fr_fix;
        long int fr_var;
        relax_stateT fr_type;
        relax_substateT fr_substate;
        unsigned long fr_address;
        long int fr_offset;
        struct symbol *fr_symbol;
        char *fr_opcode;
        struct frag *fr_next;
        char fr_literal[];
}
```

fr_fix is the size of the fixed-size piece of the frag.

fr_var is the maximum (?) size of the variable-sized piece of the frag.

fr_type is the type of the frag. Current types are: rs_fill rs_align rs_org rs_machine_dependent

fr_substate
: This stores the type of machine-dependent frag this is. (what kind of addressing mode is being used, and what size is being tried/will fit/etc.

fr_address *fr_address* is only valid after relaxation is finished. Before relaxation, the only way to store an address is (pointer to frag containing the address) plus (offset into the frag).

fr_offset This contains a number, whose meaning depends on the type of the frag. for machine_dependent frags, this contains the offset from fr_symbol that the frag wants to go to. Thus, for branch instructions it is usually zero. (unless the instruction was '`jba foo+12`' or something like that.)

fr_symbol for machine_dependent frags, this points to the symbol the frag needs to reach.

fr_opcode This points to the location in the frag (or in a previous frag) of the opcode for the instruction that caused this to be a frag. *fr_opcode* is needed if the actual opcode must be changed in order to use a different form of the addressing mode. (For example, if a conditional branch only comes in size tiny, a large-size branch could be implemented by reversing the sense of the test, and turning it into a tiny branch over a large jump. This would require changing the opcode.)

fr_literal is a variable-size array that contains the actual object bytes. A frag consists of a fixed size piece of object data, (which may be zero bytes long), followed by a piece of object data whose size may not have been determined yet. Other information includes the type of the frag (which controls how it is relaxed),

fr_next This is the next frag in the singly-linked list. This is usually only needed by the machine-independent part of `as`.

[end of manual]

Short Contents

Table of Contents