vasm assembler system

Volker Barthelmann
# Table of Contents

1 General ................................................................. 1
   1.1 Introduction .................................................. 1
   1.2 Legal ..................................................... 1
   1.3 Installation .................................................. 1

2 The Assembler .................................................. 3
   2.1 General Assembler Options ................................. 3
   2.2 Expressions .................................................. 5
   2.3 Symbols ....................................................... 7
   2.4 Include Files ................................................ 7
   2.5 Macros ......................................................... 7
   2.6 Structures .................................................... 7
   2.7 Conditional Assembly ..................................... 7
   2.8 Known Problems ........................................... 8
   2.9 Credits ........................................................ 8
   2.10 Error Messages ........................................... 8

3 Standard Syntax Module ......................... 11
   3.1 Legal ..................................................... 11
   3.2 Additional options for this module ...................... 11
   3.3 General Syntax ............................................. 11
   3.4 Directives ................................................... 12
   3.5 Known Problems ........................................... 18
   3.6 Error Messages ........................................... 18

4 Mot Syntax Module ................................. 21
   4.1 Legal ..................................................... 21
   4.2 Additional options for this module ...................... 21
   4.3 General Syntax ............................................. 22
   4.4 Directives ................................................... 22
   4.5 Known Problems ........................................... 32
   4.6 Error Messages ........................................... 32

5 Madmac Syntax Module ....................... 35
   5.1 Legal ..................................................... 35
   5.2 General Syntax ............................................. 35
   5.3 Directives ................................................... 35
   5.4 Known Problems ........................................... 39
   5.5 Error Messages ........................................... 39
6 Oldstyle Syntax Module ........................ 41
   6.1 Legal ........................................ 41
   6.2 Additional options for this module .......... 41
   6.3 General Syntax ................................. 41
   6.4 Directives ..................................... 42
   6.5 Structures .................................... 48
   6.6 Known Problems ................................. 49
   6.7 Error Messages .................................. 49

7 Test output module ............................... 51
   7.1 Legal ........................................ 51
   7.2 Additional options for this module .......... 51
   7.3 General ........................................ 51
   7.4 Restrictions ................................... 51
   7.5 Known Problems ................................ 51
   7.6 Error Messages .................................. 51

8 ELF output module ............................... 53
   8.1 Legal ........................................ 53
   8.2 Additional options for this module .......... 53
   8.3 General ........................................ 53
   8.4 Restrictions ................................... 53
   8.5 Known Problems ................................ 53
   8.6 Error Messages .................................. 53

9 a.out output module ............................. 55
   9.1 Legal .......................................... 55
   9.2 Additional options for this module .......... 55
   9.3 General ........................................ 55
   9.4 Restrictions ................................... 55
   9.5 Known Problems ................................ 55
   9.6 Error Messages .................................. 55

10 TOS output module ............................. 57
   10.1 Legal ......................................... 57
   10.2 Additional options for this module .......... 57
   10.3 General ........................................ 57
   10.4 Restrictions ................................... 57
   10.5 Known Problems ................................ 57
   10.6 Error Messages .................................. 57
11  Amiga output module ............................ 59
   11.1 Legal ............................................ 59
   11.2 Additional options for this module .............. 59
   11.3 General ........................................... 59
   11.4 Restrictions ...................................... 60
   11.5 Known Problems ................................. 60
   11.6 Error Messages ................................. 60

12  X68k output module .............................. 61
   12.1 Legal ............................................ 61
   12.2 Additional options for this module .............. 61
   12.3 General ........................................... 61
   12.4 Restrictions ...................................... 61
   12.5 Known Problems ................................. 61
   12.6 Error Messages ................................. 61

13  vobj output module ............................... 63
   13.1 Legal ............................................ 63
   13.2 Additional options for this module .............. 63
   13.3 General ........................................... 63
   13.4 Restrictions ...................................... 63
   13.5 Known Problems ................................. 63
   13.6 Error Messages ................................. 63

14  Simple binary output module ...................... 65
   14.1 Legal ............................................ 65
   14.2 Additional options for this module .............. 65
   14.3 General ........................................... 65
   14.4 Known Problems ................................. 65
   14.5 Error Messages ................................. 65

15  Motorola srecord output module ............... 67
   15.1 Legal ............................................ 67
   15.2 Additional options for this module .............. 67
   15.3 General ........................................... 67
   15.4 Known Problems ................................. 67
   15.5 Error Messages ................................. 67
16 m68k cpu module ............................................. 69
  16.1 Legal ...................................................... 69
  16.2 Additional options for this module .................. 69
      16.2.1 CPU selections ..................................... 69
      16.2.2 Optimization options .............................. 70
      16.2.3 Other options ....................................... 72
  16.3 General ................................................. 73
  16.4 Extensions .............................................. 74
  16.5 Optimizations ........................................... 78
  16.6 Known Problems .......................................... 81
  16.7 Error Messages .......................................... 82

17 PowerPC cpu module ............................... 85
  17.1 Legal ...................................................... 85
  17.2 Additional options for this module .................. 85
  17.3 General ................................................. 86
  17.4 Extensions .............................................. 86
  17.5 Optimizations ........................................... 86
  17.6 Known Problems .......................................... 86
  17.7 Error Messages .......................................... 87

18 c16x/st10 cpu module .................................... 89
  18.1 Legal ...................................................... 89
  18.2 Additional options for this module .................. 89
  18.3 General ................................................. 89
  18.4 Extensions .............................................. 89
  18.5 Optimizations ........................................... 90
  18.6 Known Problems .......................................... 90
  18.7 Error Messages .......................................... 90

19 6502 cpu module ............................................ 91
  19.1 Legal ...................................................... 91
  19.2 Additional options for this module .................. 91
  19.3 General ................................................. 91
  19.4 Extensions .............................................. 92
  19.5 Optimizations ........................................... 92
  19.6 Known Problems .......................................... 92
  19.7 Error Messages .......................................... 92

20 ARM cpu module ............................................ 93
  20.1 Legal ...................................................... 93
  20.2 Additional options for this module .................. 93
  20.3 General ................................................. 94
  20.4 Extensions .............................................. 94
  20.5 Optimizations ........................................... 94
  20.6 Known Problems .......................................... 95
  20.7 Error Messages .......................................... 95
# Interface

## 26.1 Introduction

## 26.2 Building vasm
- **26.2.1 Directory Structure**
- **26.2.2 Adapting the Makefile**
- **26.2.3 Building vasm**

## 26.3 General data structures
- **26.3.1 Source**
- **26.3.2 Sections**
- **26.3.3 Symbols**
- **26.3.4 Register symbols**
- **26.3.5 Atoms**
- **26.3.6 Relocations**
- **26.3.7 Errors**

## 26.4 Syntax modules
- **26.4.1 The file `syntax.h`**
- **26.4.2 The file `syntax.c`**

## 26.5 CPU modules
- **26.5.1 The file `cpu.h`**
- **26.5.2 The file `cpu.c`**

## 26.6 Output modules
1 General

1.1 Introduction

vasm is a portable and retargetable assembler able to create linkable objects in different formats as well as absolute code. Different CPU-, syntax and output-modules are supported. Many common directives/pseudo-opcodes are supported (depending on the syntax module) as well as CPU-specific extensions.

The assembler supports optimizations and relaxations (e.g. choosing the shortest possible branch instruction or addressing mode as well as converting a branch to an absolute jump if necessary).

The concept is that you get a special vasm binary for any combination of CPU- and syntax-module. All output modules, which make sense for the current CPU, are included in the vasm binary and you have to make sure to choose the output file format you need (refer to the next chapter and look for the `-F` option). The default is a test output, only useful for debugging or analyzing the output.

1.2 Legal

vasm is copyright in 2002-2019 by Volker Barthelmann.

This archive may be redistributed without modifications and used for non-commercial purposes.

An exception for commercial usage is granted, provided that the target CPU is M68k and the target OS is AmigaOS. Resulting binaries may be distributed commercially without further licensing.

In all other cases you need my written consent.

Certain modules may fall under additional copyrights.

1.3 Installation

The vasm binaries do not need additional files, so no further installation is necessary. To use vasm with vbcc, copy the binary to `vbcc/bin` after following the installation instructions for vbcc.

The vasm binaries are named `vasm<cpu>_<syntax>` with `<cpu>` representing the CPU-module and `<syntax>` the syntax-module, e.g. vasm for PPC with the standard syntax module is called `vasmppc_std`.

Sometimes the syntax-modifier may be omitted, e.g. `vasmppc`.

Detailed instructions how to build vasm can be found in the last chapter.
2 The Assembler

This chapter describes the module-independent part of the assembler. It documents the options and extensions which are not specific to a certain target, syntax or output driver. Be sure to also read the chapters on the backend, syntax- and output-module you are using. They will likely contain important additional information like data-representation or additional options.

2.1 General Assembler Options

\texttt{vasm} is run using the following syntax:

\begin{verbatim}
vasm<target>_<syntax> [options] file
\end{verbatim}

The following options are supported by the machine independent part of \texttt{vasm}:

- **-chklabels**
  Issues a warning when a label matches a mnemonic or directive name in either upper or lower case.

- **-D<name>[=expression]**
  Defines a symbol with the name <name> and assigns the value of the expression when given. The assigned value defaults to 1 otherwise.

- **-depend=<type>**
  Print all dependencies while assembling the source with the given options. No output is generated. <type> may be list for printing one file name in each new line, or make for printing a sequence of file names on a single line, suitable for Makefiles. When the output file name is given by -o then \texttt{vasm} will also print outname: in front of it. Note that unlike with -dependall only relative include file dependencies will be listed (which is the common case).

- **-dependall=<type>**
  Prints dependencies in the same way as -depend, but will also print all include files with absolute paths.

- **-depfile <filename>**
  Used together with -depend or -dependall and instructs vasm to output all dependencies to a new file, instead of stdout. Code will be generated in parallel to the dependencies output.

- **-dwarf[=<version>]**
  Automatically generate DWARF debugging sections, suitable for source level debugging. When the version specification is missing DWARF V3 will be generated. The only difference to V2 is that it creates a .debug_ranges section, with address ranges for all sections, instead of a bad workaround by specifying DW_AT_low_pc=0 and DW_AT_high_pc=0. Note that when you build vasm from source, you may have to specify your host operating system with -Dname in the Makefile to include the appropriate code which can determine the current work directory. Otherwise, the default would be to set the current work directory to an empty string. Currently supported are: UNIX, AMIGA, ATARI, MSDOS, _WIN32.
-esc  Enable escape character sequences. This will make vasm treat the escape character \ in string constants similar as in the C language.

-F<fmt> Use module <fmt> as output driver. See the chapter on output drivers for available formats and options.

-I<path> Define another include path. They are searched in the order of occurrence on the command line.

-ignore-mult-inc When the same file is included multiple times with the same path this is silently ignored, causing the file to be processed only once. Note that you can still include the same file twice when using different paths to access it.

-L <listfile> Enables generation of a listing file and directs the output into the file <listfile>.

-Ll<lines> Set the number of lines per listing file page to <lines>.

-Lnf Do not emit any form feed code into the listing file, for starting a new page.

-Lns Do not include symbols in the listing file.

-maxerrors=<n> Defines the maximum number of errors to display before assembly is aborted. When <n> is 0 then there is no limit. Defaults to 5.

-maxmacrecurs=<n> Defines the maximum of number of recursions within a macro. Defaults to 1000.

-nocase Disables case-sensitivity for everything - identifiers, directives and instructions. Note that directives and instructions may already be case-insensitive by default in some modules.

-noesc No escape character sequences. This will make vasm treat the escape character \ as any other character. Might be useful for compatibility.

-noialign Perform no automatic alignment for instructions. Note that unaligned instructions make your code crash when executed! Only set when you know what you do!

-nosym Strips all local symbols from the output file and doesn’t include any other symbols than those which are required for external linkage.

-nowarn=<n> Disable warning message <n>. <n> has to be the number of a valid warning message, otherwise an error is generated.

-o <ofile> Write the generated assembler output to <ofile> rather than a.out.

-pic Try to generate position independent code. Every relocation is flagged by an error message.
-quiet  Do not print the copyright notice and the final statistics.

-unnamed-sections
Sections are no longer distinguished by their name, but only by their attributes. This has the effect that when defining a second section with a different name but same attributes as a first one, it will switch to the first, instead of starting a new section.

-unsshift
The shift-right operator (>>) treats the value to shift as unsigned, which has the effect that 0-bits are inserted on the left side. The number of bits in a value depend on the target address type (refer to the appropriate cpu module documentation).

-w  Hide all warning messages.

-wfail  The return code of vasm will no longer be 0 (success), when there was a warning. Errors always make the return code fail.

-x  Show an error message, when referencing an undefined symbol. The default behaviour is to declare this symbol as externally defined.

Note that while most options allow an argument without any separating blank, some others require it (e.g. -o and -L).

2.2 Expressions
Standard expressions are usually evaluated by the main part of vasm rather than by one of the modules (unless this is necessary).
All expressions evaluated by the frontend are calculated in terms of target address values, i.e. the range depends on the backend.
The available operators include all those which are common in assembler as well as in C expressions.
C like operators:
- Unary: + - ! ~
- Arithmetic: + - * / % << >>
- Bitwise: & | ~
- Logical: && ||
- Comparative: < > <= >= == !=

Assembler like operators:
- Unary: + - ~
- Arithmetic: + - * / // << >>
- Bitwise: & ! ~
- Comparative: < > <= >= = !=

Up to version 1.4b the operators had the same precedence and associativity as in the C language. Newer versions have changed the operator priorities to comply with the common assembler behaviour. The expression evaluation priorities, from highest to lowest, are:
1. + - ! ~ (unary +/- sign, not, complement)
2. << >> (shift left, shift right)
3. & (bitwise and)
4. ^ ~ (bitwise exclusive-or)
5. | ! (bitwise inclusive-or)
6. * / % // (multiply, divide, modulo)
7. + - (plus, minus)
8. < > <= >= (less, greater, less or equal, greater or equal)
9. == != = <> (equality, inequality)
10. && (logical and)
11. || (logical or)

Operands are integral values of the target address type. They can either be specified as integer constants of different bases (see the documentation on the syntax module to see how the base is specified) or character constants. Character constants are introduced by ' or " and have to be terminated by the same character that started them.

Multiple characters are allowed and a constant is built according to the endianess of the target.

When the -esc option was specified, or automatically enabled by a syntax module, vasm interprets escape character sequences as in the C language:

\ Produces a single \.
\b The bell character.
\f Form feed.
\n Line feed.
\r Carriage return.
\t Tabulator.
\" Produces a single ".
\' Produces a single ‘.
\e Escape character (27).
\<octal-digits>
 One character with the code specified by the digits as octal value.
\x<hexadecimal-digits>
 One character with the code specified by the digits as hexadecimal value.
\X<hexadecimal-digits>
 Same as \x.

Note, that the default behaviour of vasm has changed since V1.7! Escape sequence handling has been the default in older versions. This has been changed to increase compatibility with other assemblers. Use -esc to assemble sources with escape character sequences. It is still the default in the std syntax module, though.
2.3 Symbols

You can define as many symbols as your available memory permits. A symbol may have any length and can be of global or local scope. Internally, there are three types of symbols:

**Expression** These symbols are usually not visible outside the source, unless they are explicitly exported.

**Label** Labels are always addresses inside a program section. By default they have local scope for the linker.

**Imported** These symbols are externally defined and must be resolved by the linker.

Beginning with vasm V1.5c one expression symbol is always defined to allow conditional assembly depending on the assembler being used: \_\_VASM. Its value depends on the selected cpu module. There may be other symbols which are pre-defined by the syntax- or by the cpu module.

2.4 Include Files

Vasm supports include files and defining include paths. Whether this functionality is available depends on the syntax module, which has to provide the appropriate directives.

On startup vasm will define at least one default include path: the current working directory, where the assembler program was launched from. When the input file is loaded from a different directory, i.e. the input file is a relative or absolute path and not a single file name, then the path to the input file name will be added as another include path.

Include paths are searched in the following order:
1. Current work directory.
2. Paths specified by \-I in the order of occurrence on the command line.
3. Path to the input source file.
4. Paths specified by directives inside the source text (in the order of occurrence).

2.5 Macros

Macros are supported by vasm, but the directives for defining them have to be implemented in the syntax module. The assembler core supports 9 macro arguments by default to be passed in the operand field, which can be extended to any number by the syntax module. They can be referenced inside the macro either by name (\name) or by number (\1 to \9), or both, depending on the syntax module. Recursions and early exits are supported. Refer to the selected syntax module for more details.

2.6 Structures

Vasm supports structures, but the directives for defining them have to be implemented in the syntax module.

2.7 Conditional Assembly

Has to be provided completely by the syntax module.
2.8 Known Problems

Some known module-independent problems of \texttt{vasm} at the moment:

- None.

2.9 Credits

All those who wrote parts of the \texttt{vasm} distribution, made suggestions, answered my questions, tested \texttt{vasm}, reported errors or were otherwise involved in the development of \texttt{vasm} (in descending alphabetical order, under work, not complete):

- Jordan Zebor
- Joseph Zatarski
- Frank Wille
- Henryk Richter
- Sebastian Pachuta
- Esben Norby
- Gunther Nikl
- George Nakos
- Timm S. Mueller
- Gareth Morris
- Dominic Morris
- Jean-Paul Mari
- Mauricio Muñoz Lucero
- Jörg van de Loo
- Robert Leffmann
- Andreas Larsson
- Miro Kropacek
- Mikael Kalms
- Matthew Hey
- Philippe Guichardon
- Romain Giot
- Francois Galea
- Tom Duin
- Karoly Balogh

2.10 Error Messages

The frontend has the following error messages:

- 1: illegal operand types
- 2: unknown mnemonic <\texttt{\%s}>
- 3: unknown section <\texttt{\%s}>
- 4: no current section specified
- 5: internal error %d in line %d of %s
- 6: symbol <\%s> redefined
- 7: \%c expected
- 8: cannot resolve section <\%s>, maximum number of passes reached
- 9: instruction not supported on selected architecture
- 10: number or identifier expected
- 11: could not initialize %s module
- 12: multiple input files
- 13: could not open <\%s> for input
- 14: could not open <\%s> for output
- 15: unknown option <\%s>
- 16: no input file specified
- 17: could not initialize output module <\%s>
- 18: out of memory
- 19: symbol <\%s> recursively defined
- 20: fail: \%s
- 21: section offset is lower than current pc
- 22: target data type overflow (%d bits)
- 23: undefined symbol <\%s>
- 24: trailing garbage after option -\%c
- 25: missing pacro parameters
- 26: missing end directive for macro "\%s"
- 27: macro definition inside macro "\%s"
- 28: maximum number of %d macro arguments exceeded
- 29: option \%s was specified twice
- 30: read error on <\%s>
- 31: expression must be constant
- 32: initialized data in bss
- 33: missing end directive in repeat-block
- 34: #\%d is not a valid warning message
- 35: relocation not allowed
- 36: illegal escape sequence \%c
- 37: no current macro to exit
- 38: internal symbol %s redefined by user
- 39: illegal relocation
- 40: label name conflicts with mnemonic
- 41: label name conflicts with directive
- 42: division by zero
- 43: illegal macro argument
- 44: reloc org is already set
- 45: reloc org was not set
- 46: address space overflow
- 47: bad file-offset argument
- 48: assertion "%s" failed: %s
- 49: cannot declare structure within structure
- 50: no structure
- 51: instruction has been auto-aligned
- 52: macro name conflicts with mnemonic
- 53: macro name conflicts with directive
- 54: non-relocatable expression in equate <%s>
- 55: initialized data in offset section
- 56: illegal structure recursion
- 57: maximum number of macro recursions (%d) reached
- 58: data has been auto-aligned
- 59: register symbol <%s> redefined
- 60: cannot evaluate constant huge integer expression
- 61: cannot evaluate floating point expression
- 62: imported symbol <%s> was not referenced
- 63: symbol <%s> already defined with %s scope
- 64: unexpected "else" without "if"
- 65: unexpected "endif" without "if"
- 66: maximum if-nesting depth exceeded (%d levels)
- 67: "endif" missing for conditional block started at %s line %d
- 68: repeatedly defined symbol <%s>
- 69: macro <%s> does not exist
- 70: register <%s> does not exist
- 71: register symbol <%s> has wrong type
- 72: cannot mix positional and keyword arguments
- 73: undefined macro argument name
- 74: required macro argument %d was left out
- 75: label <%s> redefined
3 Standard Syntax Module

This chapter describes the standard syntax module which is available with the extension std.

3.1 Legal

This module is written in 2002-2017 by Volker Barthelmann and is covered by the vasm copyright without modifications.

3.2 Additional options for this module

This syntax module provides the following additional options:

- **ac**
  Immediately allocate common symbols in .bss/.sbss section and define them as externally visible.

- **align**
  Enforces the backend’s natural alignment for all data directives (.word, .long, .float, etc.).

- **nodotneeded**
  Recognize assembly directives without a leading dot (.)

- **noesc**
  Ignore escape character sequences in string constants.

- **sdlimit=**<n>
  Put data up to a maximum size of n bytes into the small-data sections. Default is n=0, which means the function is disabled.

3.3 General Syntax

Labels have to be terminated with a colon (:). Local labels may either be preceded by a ’.’ or terminated by ’$’, and consist out of digits only. Local labels exist and keep their value between two global label definitions.

Make sure that you don’t define a label on the same line as a directive for conditional assembly (if, else, endif)! This is not supported.

The operands are separated from the mnemonic by whitespace. Multiple operands are separated by comma (,).

Comments are introduced by the comment character #. The rest of the line will be ignored. For the c16x, m68k, 650x, ARM, Z80, 6800 and Jaguar-RISC backends, the comment character is ; instead of #, although # is still allowed when being the first non-blank character on a line.

Example:

```
mylabel: inst.q1.q2 op1,op2,op3 # comment
```

In expressions, numbers starting with 0x or 0X are hexadecimal (e.g. 0xfb2c). 0b or 0B introduces binary numbers (e.g. 0b1100101). Other numbers starting with 0 are assumed to be octal numbers, e.g. 0237. All numbers starting with a non-zero digit are decimal, e.g. 1239.

C-like escape characters in string constants are allowed by default, unless disabled by -noesc.
3.4 Directives

All directives are case-insensitive. The following directives are supported by this syntax module (if the CPU- and output-module allow it):

`.2byte <exp1>[,<exp2>...]`

See `.uahalf`.

`.4byte <exp1>[,<exp2>...]`

See `.uaword`.

`.8byte <exp1>[,<exp2>...]`

See `.uaquad`.

`.ascii <exp1>[,<exp2>,"<string1>"...]`

See `.byte`.

`.abort <message>`

Print an error and stop assembly immediately.

`.asciiz"<string1>"[,"<string2>"...]`

See `.string`.

`.align <bitorbyte_count>[,<fill>][,<maxpad>]`

Depending on the current CPU backend `.align` either behaves like `.balign` (x86) or like `.p2align` (PPC).

`.balign <byte_count>[,<fill>][,<maxpad>]`

Insert as much fill bytes as required to reach an address which is dividable by `<byte_count>`. For example `.balign 2` would make an alignment to the next 16-bit boundary. The padding bytes are initialized by `<fill>`, when given. The optional third argument defines a maximum number of padding bytes to use. When more are needed then the alignment is not done at all.

`.balignl <bit_count>[,<fill>][,<maxpad>]`

Works like `.balign`, with the only difference that the optional fill value can be specified as a 32-bit word. Padding locations which are not already 32-bit aligned, will cause a warning and padded by zero-bytes.

`.bignw <bit_count>[,<fill>][,<maxpad>]`

Works like `.balign`, with the only difference that the optional fill value can be specified as a 16-bit word. Padding locations which are not already 16-bit aligned, will cause a warning and padded by zero-bytes.

`.byte <exp1>[,<exp2>,"<string1>"...]`

Assign the integer or string constant operands into successive bytes of memory in the current section. Any combination of integer and character string constant operands is permitted.

`.comm <symbol>,<size>[,<align>]`

Defines a common symbol which has a size of `<size>` bytes. The final size and alignment will be assigned by the linker, which will use the highest size and alignment values of all common symbols with the same name found. A common symbol is allocated in the .bss section in the final executable. ".comm"-areas
of less than 8 bytes in size are aligned to word boundaries, otherwise to doubleword boundaries.

.double <exp1>[,<exp2>...]  
Parse one of more double precision floating point expressions and write them into successive blocks of 8 bytes into memory using the backend’s endianess.

.endm  
Ends a macro definition.

.endr  
Ends a repetition block.

.equ <symbol>,<expression>  
See .set.

.equiv <symbol>,<expression>  
Assign the <expression> to <symbol> similar to .equ and .set, but signals an error when <symbol> has already been defined.

.err <message>  
Print a user error message. Do not create an output file.

.extern <symbol>[,<symbol>...]  
See .global.

.fail <expression>  
Cause a warning when <expression> is greater or equal 500. Otherwise cause an error.

.file "string"  
Set the filename of the input source. This may be used by some output modules. By default, the input filename passed on the command line is used.

.float <exp1>[,<exp2>...]  
Parse one of more single precision floating point expressions and write them into successive blocks of 4 bytes into memory using the backend’s endianess.

.global <symbol>[,<symbol>...]  
Flag <symbol> as an external symbol, which means that <symbol> is visible to all modules in the linking process. It may be either defined or undefined.

.globl <symbol>[,<symbol>...]  
See .global.

.half <exp1>[,<exp2>...]  
Assign the values of the operands into successive 16-bit words of memory in the current section using the backend’s endianess.

.if <expression>  
Conditionally assemble the following lines if <expression> is non-zero.

.ifeq <expression>  
Conditionally assemble the following lines if <expression> is zero.

.ifne <expression>  
Conditionally assemble the following lines if <expression> is non-zero.
.ifgt <expression>
   Conditionally assemble the following lines if <expression> is greater than zero.

.ifge <expression>
   Conditionally assemble the following lines if <expression> is greater than zero or equal.

.iflt <expression>
   Conditionally assemble the following lines if <expression> is less than zero.

.ifle <expression>
   Conditionally assemble the following lines if <expression> is less than zero or equal.

.ifb <operand>
   Conditionally assemble the following lines when <operand> is completely blank, except an optional comment.

.ifnb <operand>
   Conditionally assemble the following lines when <operand> is non-blank.

ifdef <symbol>
   Conditionally assemble the following lines if <symbol> is defined.

ifndef <symbol>
   Conditionally assemble the following lines if <symbol> is undefined.

.incbin <file>
   Inserts the binary contents of <file> into the object code at this position. The file will be searched first in the current directory, then in all paths defined by -I or .incdir in the order of occurrence.

.incdir <path>
   Add another path to search for include files to the list of known paths. Paths defined with -I on the command line are searched first.

.include <file>
   Include source text of <file> at this position. The include file will be searched first in the current directory, then in all paths defined by -I or .incdir in the order of occurrence.

.int <exp1>[,<exp2>...]  
   See .long.

.irp <symbol>[,<val>...]  
   Iterates the block between .irp and .endr for each <val>. The current <val>, which may be embedded in quotes, is assigned to \symbol. If no value is given, then the block is assembled once, with \symbol set to an empty string.

.irpc <symbol>[,<val>...]  
   Iterates the block between .irp and .endr for each character in each <val>, and assign it to \symbol. If no value is given, then the block is assembled once, with \symbol set to an empty string.
.lcomm <symbol>,<size>[,<alignment>]
Allocate <size> bytes of space in the .bss section and assign the value to that
location to <symbol>. If <alignment> is given, then the space will be aligned
to an address having <alignment> low zero bits or 2, whichever is greater.
<symbol> may be made globally visible by the .globl directive.

.list The following lines will appear in the listing file, if it was requested.

.local <symbol>[,<symbol>...] Flag <symbol> as a local symbol, which means that <symbol> is local for the
current file and invisible to other modules in the linking process.

.long <exp1>[,<exp2>...] Assign the values of the operands into successive 32-bit words of memory in the
current section using the backend’s endianess.

.macro <name> [<argname1>=[<default>]][,<argname2>...] Defines a macro which can be referenced by <name>. The macro definition is
closed by an .endm directive. The names of the arguments, which may be passed
to this macro, must be declared directly following the macro name. You can
define an optional default value in the case an argument is left out. Note that
macro names are case-insensitive while the argument names are case-sensitive.
Within the macro context arguments are referenced by \argname. The special
argument \@ inserts a unique id, useful for defining labels. \() may be used as
a separator between the name of a macro argument and the subsequent text.

.nolist The following lines will not be visible in a listing file.

.org <exp>
Before any other section directive <exp> defines the absolute start address of
the program. Within a section <exp> defines the offset from the start of this
section for the subsequent code. When <exp> starts with a current-pc symbol
followed by a plus (+) operator, then the directive behaves like .space.

.p2align <bit_count>[,[<fill>]][,<maxpad>] Insert as much fill bytes as required to reach an address where <bit_count> low
order bits are zero. For example .p2align 2 would make an alignment to the
next 32-bit boundary. The padding bytes are initialized by <fill>, when given.
The optional third argument defines a maximum number of padding bytes to
use. When more are needed then the alignment is not done at all.

.p2alignl <bit_count>[,[<fill>]][,<maxpad>] Works like .p2align, with the only difference that the optional fill value can
be specified as a 32-bit word. Padding locations which are not already 32-bit
aligned, will cause a warning and padded by zero-bytes.

.p2alignw <bit_count>[,[<fill>]][,<maxpad>] Works like .p2align, with the only difference that the optional fill value can
be specified as a 16-bit word. Padding locations which are not already 16-bit
aligned, will cause a warning and padded by zero-bytes.
.quad <exp1>[,<exp2>...]  
Assign the values of the operands into successive quadwords (64-bit) of memory in the current section using the backend’s endianess.

.rept <expression>  
Repeats the assembly of the block between .rept and .endr <expression> number of times. <expression> has to be positive.

.section <name>[,"<attributes>"][,[,<type>]|,[,%<type>]|[,<mem_flags>]]  
Starts a new section named <name> or reactivates an old one. If attributes are given for an already existing section, they must match exactly. The section’s name will also be defined as a new symbol, which represents the section’s start address. The "<attributes>" string may consist of the following characters:

Section Contents:

\begin{itemize}
  \item c section has code
  \item d section has initialized data
  \item u section has uninitialized data
  \item i section has directives (info section)
  \item n section can be discarded
  \item R remove section at link time
  \item a section is allocated in memory
\end{itemize}

Section Protection:

\begin{itemize}
  \item r section is readable
  \item w section is writable
  \item x section is executable
  \item s section is sharable
\end{itemize}

Section Alignment: A digit, which is ignored. The assembler will automatically align the section to the highest alignment restriction used within.

Memory flags (Amiga hunk format only):

\begin{itemize}
  \item C load section to Chip RAM
  \item F load section to Fast RAM
\end{itemize}

The optional <type> argument is mainly used for ELF output and may be introduced either by a ‘%’ or a ‘@’ character. Allowed are:

\begin{itemize}
  \item progbits This is the default value, which means the section data occupies space in the file and may have initialized data.
  \item nobits These sections do not occupy any space in the file and will be allocated filled with zero bytes by the OS loader.
\end{itemize}

When the optional, non-standard, <mem_flags> argument is given it defines a 32-bit memory attribute, which defines where to load the section (platform specific). The memory attributes are currently only used in the hunk-format output module.
.set <symbol>,<expression>
Create a new program symbol with the name <symbol> and assign to it the value of <expression>. If <symbol> is already assigned, it will contain a new value from now on.

.size <symbol>,<size>
Set the size in bytes of an object defined at <symbol>.

.short <exp1>[,<exp2>...]
See .half.

.single <exp1>[,<exp2>...]
Same as .float.

.skip <exp>[,<fill>]
Insert <exp> zero or <fill> bytes into the current section.

.space <exp>[,<fill>]
Insert <exp> zero or <fill> bytes into the current section.

.stabs "<name>"",<type>,<other>,<desc>,<exp>
Add an stab-entry for debugging, including a symbol-string and an expression.

.stabn <type>,<other>,<desc>,<exp>
Add an stab-entry for debugging, without a symbol-string.

.stabd <type>,<other>,<desc>
Add an stab-entry for debugging, without symbol-string and value.

.string "<string1>"[,"<string2>"...]
Like .byte, but adds a terminating zero-byte.

.swbeg <op>
Just for compatibility. Do nothing.

.type <symbol>,<type>
Set type of symbol called <symbol> to <type>, which must be one of:
1: Object
2: Function
3: Section
4: File

The predefined symbols @object and @function are available for this purpose.

.uahalf <exp1>[,<exp2>...]
Assign the values of the operands into successive 16-bit areas of memory in the current section regardless of current alignment.

.ualong <exp1>[,<exp2>...]
Assign the values of the operands into successive 32-bit areas of memory in the current section regardless of current alignment.

.uaquad <exp1>[,<exp2>...]
Assign the values of the operands into successive 64-bit areas of memory in the current section regardless of current alignment.
.uashort <exp1>[,<exp2>...]  
Assign the values of the operands into successive 16-bit areas of memory in the current section regardless of current alignment.

.uaword <exp1>[,<exp2>...]  
Assign the values of the operands into successive 16-bit areas of memory in the current section regardless of current alignment.

.weak <symbol>[,<symbol>...]  
Flag <symbol> as a weak symbol, which means that <symbol> is visible to all modules in the linking process and may be replaced by any global symbol with the same name. When a weak symbol remains undefined its value defaults to 0.

.word <exp1>[,<exp2>...]  
Assign the values of the operands into successive 16-bit words of memory in the current section using the backend’s endianess.

.zero <exp>[,<fill>]  
Insert <exp> zero or <fill> bytes into the current section.

Predefined section directives:

.bss .section ".bss","aurw"
.data .section ".data","adrw"
.rodata .section ".rodata","adr"
.sbss .section ".sbss","aurw"
.sdata .section ".sdata","adrw"
.sdata2 .section ".sdata2","adr"
.stab .section ".stab","dr"
.stabstr .section ".stabstr","dr"
.text .section ".text","acrx"
.tocd .section ".tocd","adrw"

3.5 Known Problems
Some known problems of this module at the moment:
   – None.

3.6 Error Messages
This module has the following error messages:
   – 1001: mnemonic expected
   – 1002: invalid extension
   – 1003: no space before operands
   – 1004: too many closing parentheses
- 1005: missing closing parentheses
- 1006: missing operand
- 1007: scratch at end of line
- 1008: section flags expected
- 1009: invalid data operand
- 1010: memory flags expected
- 1011: identifier expected
- 1012: assembly aborted
- 1013: unexpected "%s" without "%s"
- 1014: pointless default value for required parameter <\%s>
- 1015: invalid section type ignored, assuming progbits
- 1019: syntax error
- 1021: section name expected
- 1022: .fail %lld encountered
- 1023: .fail %lld encountered
- 1024: alignment too big
Chapter 4: Mot Syntax Module

This chapter describes the Motorola syntax module, mostly used for the M68k and ColdFire families of CPUs, which is available with the extension mot.

4.1 Legal

This module is written in 2002-2019 by Frank Wille and is covered by the wasm copyright without modifications.

4.2 Additional options for this module

This syntax module provides the following additional options:

- **-align** Enables natural alignment for data (e.g. `dc.?', `ds.?' and offset directives (`rs.?', `so.?', `fo.?').
- **-allmp** Makes all 35 macro arguments available. Default is 9 (\1 to \9). More arguments can be accessed through \a to \z which may conflict with escape characters or named arguments, therefore they are not enabled by default.
- **-cnop=<code>** Sets a two-byte code used for alignment padding with CNOP. Defaults to 0x4e71 on M68k.
- **-devpac** Devpac-compatibility mode. Only directives known to Devpac are recognized.
  - Enables natural alignment for data and structure offsets (see option -align).
  - Predefines offset symbols `__RS`, `__SO` and `__FO` as 0, which otherwise are undefined until first referenced.
  - Disable escape codes handling in strings (see -noesc).
  - Enable dots within identifiers (see -ldots).
  - Up to 35 macro arguments.
  - Do not use NOP instructions when aligning code.
- **-ldots** Allow dots (.) within all identifiers.
- **-localu** Local symbols are prefixed by '_.' instead of '. '. For Devpac compatibility, which offers a similar option.
- **-phxass** PhxAss-compatibility mode. Only directives known to PhxAss are recognized. Enables the following features:
  - `section <name>` starts a code section named `<name>` instead of a section which also has the type `<name>`.
  - Enable escape codes handling in strings (see -esc).
  - Macro names are treated as case-insensitive.
  - Up to 35 macro arguments.
  - Allow blanks in operands.
− Defines the symbol _PHXASS_ with value 2 (to differentiate from the real PhxAss with value 1).
− When no output file name is given, construct it from the input name.

-`spaces` Allow blanks in the operand field. Otherwise a blank would start the comment field there.

-`warncomm` Warn about all lines, which have comments in the operand field, introduced by a blank character. For example in: `dc.w 1 + 2`.

### 4.3 General Syntax

Labels must either start at the first column of a line or have to be terminated by a colon (:``). In the first case the mnemonic has to be separated from the label by whitespace (not required in any case, e.g., with the `=` directive). A double colon (`::`) automatically makes the label externally visible (refer to `xdef`).

Local labels are preceded by `.` or terminated by `$`. For the rest, any alphanumeric character including `_` is allowed. Local labels are valid between two global label definitions. Otherwise dots (`.`) are not allowed within a label by default, unless the option `-ldots` or `-devpac` was specified. Even then, labels ending on `.b`, `.w` or `.l` can’t be defined.

It is possible to refer to any local symbol in the source by preceding its name with the name of the last global symbol, which was defined before: `global_name\local_name`. This is for PhxAss compatibility only, and is no recommended style. Does not work in a macro, as it conflicts with macro arguments.

Make sure that you don’t define a label on the same line as a directive for conditional assembly (if, else, endif)! This is not supported.

Qualifiers are appended to the mnemonic, separated by a dot (if the CPU-module supports qualifiers). The operands are separated from the mnemonic by whitespace. Multiple operands are separated by comma (`,`).

In this syntax module, the operand field must not contain any whitespace characters, as long as the option `-spaces` was not specified.

Comments can be introduced everywhere by the characters `;` or `*`. The rest of the line will be ignored. Also everything following the operand field, separated by a whitespace, will be regarded as comment (unless `-spaces`). Be careful with `*`, which is recognized as the "current pc symbol" in any operand expression.

Example:

```c
mylabel inst.q op1,op2,op3 ;comment
```

In expressions, numbers starting with `$` are hexadecimal (e.g. `$fb2c`). `%` introduces binary numbers (e.g. `%1100101`). Numbers starting with `@` are assumed to be octal numbers, e.g. `@237`. All numbers starting with a digit are decimal, e.g. `1239`.

### 4.4 Directives

The following directives are supported by this syntax module (provided the CPU- and output-module support them):
<symbol> = <expression>
   Equivalent to <symbol> equ <expression>.

<symbol> =.s <expression>
   Equivalent to <symbol> fequ.s <expression>. PhxAss compatibility.

<symbol> =.d <expression>
   Equivalent to <symbol> fequ.d <expression>. PhxAss compatibility.

<symbol> =.x <expression>
   Equivalent to <symbol> fequ.x <expression>. PhxAss compatibility.

<symbol> =.p <expression>
   Equivalent to <symbol> fequ.p <expression>. PhxAss compatibility.

align <bitcount>
   Insert as much zero bytes as required to reach an address where <bitcount> low order bits are zero. For example align 2 would make an alignment to the next 32-bit boundary.

blk.b <exp>[<fill>]
   Equivalent to dcb.b <exp>,<fill>.

blk.d <exp>[<fill>]
   Equivalent to dcb.d <exp>,<fill>.

blk.l <exp>[<fill>]
   Equivalent to dcb.l <exp>,<fill>.

blk.q <exp>[<fill>]
   Equivalent to dcb.q <exp>,<fill>.

blk.s <exp>[<fill>]
   Equivalent to dcb.s <exp>,<fill>.

blk.w <exp>[<fill>]
   Equivalent to dcb.w <exp>,<fill>.

blk.x <exp>[<fill>]
   Equivalent to dcb.x <exp>,<fill>.

bss
   Equivalent to section bss,bss.

bss_c
   Equivalent to section bss_c,bss,chip.

bss_f
   Equivalent to section bss_f,bss,fast.

cargs [##<offset>,]<symbol1>[<size1>][<symbol2>[<size2>]]...
   Defines <symbol1> with the value of <offset>. Further symbols on the line, separated by comma, will be assigned the <offset> plus the size of the previous symbol. The size defaults to 2. Valid optional size extensions are: .b, .w, .l, where .l results in a size of 4, the others 2. The <offset> argument defaults to 4, when not given.

clrfo
   Reset stack-frame offset counter to zero. See fo directive.

clrso
   Reset structure offset counter to zero. See so directive.
cnop <offset>,<alignment>
Insert as much zero bytes as required to reach an address which can be divided by <alignment>. Then add <offset> zero bytes. May fill the padding-bytes with no-operation instructions for certain cpus.

code Equivalent to section code,code.

code_c Equivalent to section code_c,code,chip.

code_f Equivalent to section code_f,code,fast.

comm <symbol>,<size>
Create a common symbol with the given size. The alignment is always 32 bits.

comment Everything in the operand field is ignored and seen as a comment. There is only one exception, when the operand contains HEAD=. Then the following expression is passed to the TOS output module via the symbol 'TOSFLAGS', to define the Atari specific TOS flags.

cseg Equivalent to section code,code.

data Equivalent to section data,data.

data_c Equivalent to section data_c,data,chip.

data_f Equivalent to section data_f,data,fast.

dc.b <exp1>[,<exp2>,"<string1>",’<string2>’...] Assign the integer or string constant operands into successive bytes of memory in the current section. Any combination of integer and character string constant operands is permitted.

dc.d <exp1>[,<exp2>...] Assign the values of the operands into successive 64-bit words of memory in the current section. Also IEEE double precision floating point constants are allowed.

dc.l <exp1>[,<exp2>...] Assign the values of the operands into successive 32-bit words of memory in the current section.

dc.q <exp1>[,<exp2>...] Assign the values of the operands into successive 64-bit words of memory in the current section.

dc.s <exp1>[,<exp2>...] Assign the values of the operands into successive 32-bit words of memory in the current section. Also IEEE single precision floating point constants are allowed.

dc.w <exp1>[,<exp2>...] Assign the values of the operands into successive 16-bit words of memory in the current section.

dc.x <exp1>[,<exp2>...] Assign the values of the operands into successive 96-bit words of memory in the current section. Also IEEE extended precision floating point constants are allowed.
**dcb.b <exp>[,<fill>]**
Insert <exp> zero or <fill> bytes into the current section.

**dcb.d <exp>[,<fill>]**
Insert <exp> zero or <fill> 64-bit words into the current section. <fill> might also be an IEEE double precision constant.

**dcb.l <exp>[,<fill>]**
Insert <exp> zero or <fill> 32-bit words into the current section.

**dcb.q <exp>[,<fill>]**
Insert <exp> zero or <fill> 64-bit words into the current section. <fill> might also be an IEEE single precision constant.

**dcb.s <exp>[,<fill>]**
Insert <exp> zero or <fill> 32-bit words into the current section. <fill> might also be an IEEE single precision constant.

**dcb.w <exp>[,<fill>]**
Insert <exp> zero or <fill> 16-bit words into the current section.

**dcb.x <exp>[,<fill>]**
Insert <exp> zero or <fill> 96-bit words into the current section. <fill> might also be an IEEE extended precision constant.

**dr.b <exp1>[,<exp2>...]**
Calculates <expN> - <current pc value> and stores it into successive bytes of memory in the current section.

**dr.w <exp1>[,<exp2>...]**
Calculates <expN> - <current pc value> and stores it into successive 16-bit words of memory in the current section.

**dr.l <exp1>[,<exp2>...]**
Calculates <expN> - <current pc value> and stores it into successive 32-bit words of memory in the current section.

**ds.b <exp>**
Equivalent to dcb.b <exp>, 0.

**ds.d <exp>**
Equivalent to dcb.d <exp>, 0.

**ds.l <exp>**
Equivalent to dcb.l <exp>, 0.

**ds.q <exp>**
Equivalent to dcb.q <exp>, 0.

**ds.s <exp>**
Equivalent to dcb.s <exp>, 0.

**ds.w <exp>**
Equivalent to dcb.w <exp>, 0.

**ds.x <exp>**
Equivalent to dcb.x <exp>, 0.
dseg  Equivalent to section data, data.

dx.b <exp>
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to dcb.b <exp>, 0.

dx.d <exp>
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to dcb.d <exp>, 0.

dx.l <exp>
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to dcb.l <exp>, 0.

dx.q <exp>
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to dcb.q <exp>, 0.

dx.s <exp>
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to dcb.s <exp>, 0.

dx.w <exp>
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to dcb.w <exp>, 0.

dx.x <exp>
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to dcb.x <exp>, 0.

echo <string>
Prints <string> to stdout.

einline
End a block of isolated local labels, started by inline.

else
Assemble the following lines if the previous if condition was false.

end
Assembly will terminate behind this line.

endif
Ends a section of conditional assembly.

endm
Ends a macro definition.

endr
Ends a repetition block.

<symbol> equ <expression>
Define a new program symbol with the name <symbol> and assign to it the value of <expression>. Defining <symbol> twice will cause an error.

<symbol> equ.s <expression>
Equivalent to <symbol> fequ.s <expression>. PhxAss compatibility.

<symbol> equ.d <expression>
Equivalent to <symbol> fequ.d <expression>. PhxAss compatibility.

<symbol> equ.x <expression>
Equivalent to <symbol> fequ.x <expression>. PhxAss compatibility.
<symbol> equ.p <expression>
  Equivalent to <symbol> fequ.p <expression>. PhxAss compatibility.

erem
  Ends an outcommented block. Assembly will continue.

even
  Aligns to an even address. Equivalent to cnop 0,2.

fail <message>
  Show an error message including the <message> string. Do not generate an output file.

<symbol> fequ.s <expression>
  Define a new program symbol with the name <symbol> and assign to it the floating point value of <expression>. Defining <symbol> twice will cause an error. The extension is for Devpac-compatibility, but will be ignored.

<symbol> fequ.d <expression>
  Equivalent to <symbol> fequ.s <expression>.

<symbol> fequ.x <expression>
  Equivalent to <symbol> fequ.s <expression>.

<symbol> fequ.p <expression>
  Equivalent to <symbol> fequ.s <expression>.

<label> fo.<size> <expression>
  Assigns the current value of the stack-frame offset counter to <label>. Afterwards the counter is decremented by the instruction’s <size> multiplied by <expression>. Any valid M68k size extension is allowed for <size>: b, w, l, q, s, d, x, p. The offset counter can also be referenced directly under the name __FO.

idnt <name>
  Sets the file or module name in the generated object file to <name>, when the selected output module supports it. By default, the input filename passed on the command line is used.

if <expression>
  Conditionally assemble the following lines if <expression> is non-zero.

ifeq <expression>
  Conditionally assemble the following lines if <expression> is zero.

ifne <expression>
  Conditionally assemble the following lines if <expression> is zero.

ifgt <expression>
  Conditionally assemble the following lines if <expression> is non-zero.

ifge <expression>
  Conditionally assemble the following lines if <expression> is greater than zero.

iflt <expression>
  Conditionally assemble the following lines if <expression> is less than zero.
ifle <expression>
Conditionally assemble the following lines if <expression> is less than zero or equal.

ifb <operand>
Conditionally assemble the following lines when <operand> is completely blank, except an optional comment.

ifnb <operand>
Conditionally assemble the following lines when <operand> is non-blank.

ifc <string1>,<string2>
Conditionally assemble the following lines if <string1> matches <string2>.

ifnc <string1>,<string2>
Conditionally assemble the following lines if <string1> does not match <string2>.

ifd <symbol>
Conditionally assemble the following lines if <symbol> is defined.

ifnd <symbol>
Conditionally assemble the following lines if <symbol> is undefined.

ifmacro <macro>
Conditionally assemble the following line if <macro> is defined.

ifmacrond <macro>
Conditionally assemble the following line if <macro> is undefined.

iif <expression> <statement>
Conditionally assemble the <statement> following <expression>. IIF stands for Immediate IF. If the value of <expression> is non-zero then <statement> is assembled. No ENDC should be used in conjunction with this directive. The <statement> can not include a label, but a label may precede the IIF directive. For example: foo IIF bar equ 42 The foo label will be assigned with 42 if bar evaluates to true, otherwise foo will be assigned with the current program counter. The case when assigning a value in the IIF <statement> using the equal (=) operator and the option -spaces is used can’t work as the equal operator will be evaluated as part of the expression. I.e. foo IIF 1+1 = 42 will work, but foo IIF 1 + 1 = 42 when the option -spaces is specified won’t work as = 42 will be evaluated as part of the expression.

incbin <file>,[,<offset>],[,<length>]]
Inserts the binary contents of <file> into the object code at this position. When <offset> is specified, then the given number of bytes will be skipped at the beginning of the file. The optional <length> argument specifies the maximum number of bytes to be read from that file. The file will be searched first in the current directory, then in all paths defined by -I or incdir in the order of occurrence.

incdir <path>
Add another path to search for include files to the list of known paths. Paths defined with -I on the command line are searched first.
include <file>
Include source text of <file> at this position. The include file will be searched first in the current directory, then in all paths defined by -I or incdir in the order of occurrence.

inline
Local labels in the following block are isolated from previous local labels and those after einline.

list
The following lines will appear in the listing file, if it was requested.

llen <len>
Set the line length in a listing file to a maximum of <len> characters. Currently without any effect.

macro <name>
Defines a macro which can be referenced by <name>. The <name> may also appear at the left side of the macro directive, starting at the first column. Then the operand field is ignored. The macro definition is closed by an endm directive. When calling a macro you may pass up to 9 arguments, separated by comma. Those arguments are referenced within the macro context as \1 to \9. Parameter \0 is set to the macro’s first qualifier (mnemonic extension), when given. In Devpac- and PhxAss-compatibility mode, or with option -allmp, up to 35 arguments are accepted, where argument 10-35 can be referenced by \a to \z.
Special macro parameters:
\0 Insert a unique id, useful for defining labels. Every macro call gets its own unique id.
\0! Push the current unique id onto a global id stack, then insert it.
\0? Push the current unique id below the top element of the global id stack, then insert it.
\00 Pull the top element from the global id stack and insert it. The macro’s current unique id is not affected by this operation.
\# Insert the number of arguments that have been passed to this macro. Equivalent to the contents of NARG.
\?n Insert the length of the n’th macro argument.
\. Insert the argument which is selected by the current value of the CARG symbol (first argument, when CARG is 1).
\+ Works like \. , but increments the value of CARG after that.
\- Works like \. , but decrements the value of CARG after that.
\<symbolname>
Inserts the current decimal value of the absolute symbol symbolname.
\$<symbolname>
Inserts the current hexadecimal value of the absolute symbol symbolname, without leading $.
mexit Leave the current macro and continue with assembling the parent context. Note that this directive also resets the level of conditional assembly to a state before the macro was invoked; which means that it also works as a 'break' command on all new if directives.

nolist The following lines will not be visible in a listing file.
nopage Never start a new page in the listing file. This implementation will only prevent emitting the formfeed code.
nref <symbol>,<symbol>... Flag <symbol> as externally defined, similar to xref, but also indicate that references should be optimized to base-relative addressing modes, when possible. This directive is only present in PhxAss-compatibility mode.

odd Aligns to an odd address. Equivalent to cnop 1,2. Bugs: Note that this is not a real odd instruction, as it wastes two bytes when the address is already odd.

offset [<expression>] Switches to a special offset-section. The contents of such a section is not included in the output. Their labels may be referenced as absolute offset symbols. Can be used to define structure offsets. The optional <expression> gives the start offset for this section. When missing the last offset of the previous offset-section is used, or 0.

org <expression> Sets the base address for the subsequent code. Note that it is allowed to embed such an absolute ORG block into a section. Return into relocatable mode with any new section directive. Although in Devpac compatibility mode the previous section will stay absolute.

output <name> Sets the output file name to <name> when no output name was given on the command line. A special case for Devpac-compatibility is when <name> starts with a '.' and an output name was already given. Then the current output name gets <name> appended as an extension. When an extension already exists, then it is replaced.

page Start a new page in the listing file (not implemented). Make sure to start a new page when the maximum page length is reached.

plen <len> The the page length for a listing file to <len> lines. Currently ignored.

printt <string>,<string>... Prints <string> to stdout. Each additional string into a new line. Quotes are optional.

printv <expression>,<expression>... Evaluate <expression> and print it to stdout out in hexadecimal, decimal, ASCII and binary format.

public <symbol>,<symbol>... Flag <symbol> as an external symbol, which means that <symbol> is visible to all modules in the linking process. It may be either defined or undefined.
rem The assembler will ignore everything from encountering the rem directive until an erem directive was found.

rept <expression>
Repeats the assembly of the block between rept and endr <expression> number of times. <expression> has to be positive. The internal symbol REPTN always holds the iteration counter of the inner repeat loop, starting with 0. REPTN is -1 outside of any repeat block.

rorg <expression>
Sets the program counter <expression> bytes behind the start of the current section. The new program counter must not be smaller than the current one. The space will be padded with zeros.

<label> rs.<size> <expression>
Works like the so directive, with the only difference that the offset symbol is named __RS.

rsreset Equivalent to clrso, but the symbol manipulated is __RS.

rsset Equivalent to setso, but the symbol manipulated is __RS.

section [<name>,<sec_type>,<mem_type>]
Starts a new section named <name> or reactivates an old one. <sec_type> defines the section type and may be code, text (same as code), data or bss. When the output format is "tos", then a single argument will be interpreted as section type. Otherwise this argument is used as section name and the type defaults to code. When <mem_type> is given it defines a 32-bit memory attribute, which specifies where to load the section. <mem_type> is either a numerical constant or one of the keywords chip (for Chip-RAM) or fast (for Fast-RAM). Optionally it is also possible to attach the suffix _C, _F or _P to the <sec_type> argument for defining the memory type. The memory attributes are currently only used in the hunk-format output module.

<symbol> set <expression>
Create a new symbol with the name <symbol> and assign the value of <expression>. If <symbol> is already assigned, it will contain a new value from now on.

setfo <expression>
Sets the stack-frame offset counter to <expression>. See fo directive.

setso <expression>
Sets the structure offset counter to <expression>. See so directive.

<label> so.<size> <expression>
Assigns the current value of the structure offset counter to <label>. Afterwards the counter is incremented by the instruction’s <size> multiplied by <expression>. Any valid M68k size extension is allowed for <size>: b, w, l, q, s, d, x, p. The offset counter can also be referenced directly under the name __SO.

spc <lines>
Output <lines> number of blank lines in the listing file. Currently without any effect.
text          Equivalent to section code, code.

ttl <name>
          PhxAss syntax. Equivalent to idnt <name>.

<name> ttl
          Motorola syntax. Equivalent to idnt <name>.

weak <symbol>[,<symbol>...]
          Flag <symbol> as a weak symbol, which means that <symbol> is visible to all
          modules in the linking process and may be replaced by any global symbol with
          the same name. When a weak symbol remains undefined its value defaults to
          0.

xdef <symbol>[,<symbol>...]
          Flag <symbol> as a global symbol, which means that <symbol> is visible to all
          modules in the linking process. See also public.

xref <symbol>[,<symbol>...]
          Flag <symbol> as externally defined, which means it has to be important from
          another module in the linking process. See also public.

4.5 Known Problems

Some known problems of this module at the moment:
  − odd directive wastes two bytes, when address is already odd.
  − Some listing file directives have no effect.

4.6 Error Messages

This module has the following error messages:
  − 1001: mnemonic expected
  − 1002: invalid extension
  − 1003: no space before operands
  − 1004: too many closing parentheses
  − 1005: missing closing parentheses
  − 1006: missing operand
  − 1007: garbage at end of line
  − 1008: syntax error
  − 1009: invalid data operand
  − 1010: , expected
  − 1011: identifier expected
  − 1012: directive has no effect
  − 1013: unexpected "%s" without "%s"
  − 1014: illegal section type
  − 1015: macro id insert on empty stack
- 1016: illegal memory type
- 1017: macro id stack overflow
- 1018: macro id pull without matching push
- 1019: check comment
- 1020: invalid numeric expansion
- 1021: einline without inline
- 1022: missing %c
- 1023: maximum inline nesting depth exceeded (%d)
5 Madmac Syntax Module

This chapter describes the madmac syntax module, which is compatible to the MadMac assembler syntax, written by Landon Dyer for Atari and improved later to support Jaguar and JRISC. It is mainly intended for Atari’s 6502, 68000 and Jaguar systems.

5.1 Legal

This module is written in 2015 by Frank Wille and is covered by the wasm copyright without modifications.

5.2 General Syntax

A statement may contain up to four fields which are identified by order of appearance and terminating characters. The general form is:

```plaintext
label: operator operand(s) ; comment
```

Labels must not start at the first column, as they are identified by the mandatory terminating colon (:) character. A double colon (::) automatically makes the label externally visible.

Labels preceded by ‘.’ have local scope and are only valid between two global labels.

Equate directives, starting in the operator field, have a symbol without terminating colon in the first field, left of the operator. The equals-character (=) can be used as an alias for equ. A double-equals (==) automatically makes the symbol externally visible.

```plaintext
symbol equate expression ; comment
```

Identifiers, like symbols or labels, may start with any upper- or lower-case character, a dot (.), question-mark (?) or underscore (_). The remaining characters may be any alphanumeric character, a dollar-sign ($), question-mark (?) or underscore (_).

The operands are separated from the operator by whitespace. Multiple operands are separated by comma (,).

Comments are introduced by the comment character ;;. The asterisk (*) can be used at the first column to start a comment. The rest of the line will be ignored.

In expressions, numbers starting with $ are hexadecimal (e.g. $fb2c). % introduces binary numbers (e.g. %1100101). Numbers starting with @ are assumed to be octal numbers, e.g. @237. All other numbers starting with a digit are decimal, e.g. 1239.

NOTE: Unlike the original Madmac assembler all expressions are evaluated following the usual mathematical operator priorities.

C-like escape characters are supported in strings.

5.3 Directives

The following directives are supported by this syntax module (if the CPU- and output-module allow it). Note that all directives, besides the equals-character, may be optionally preceded by a dot (.).

```plaintext
<symbol> = <expression>
```

Equivalent to <symbol> equ <expression>.
### Equivalent to <symbol> equ <expression>, but declare <symbol> as externally visible.

### assert <expression>[,<expression>...]
Assert that all conditions are true (non-zero), otherwise issue a warning.

### bss
The following data (space definitions) are going into the BSS section. The BSS section cannot contain any initialized data.

### data
The following data are going into the data section, which usually contains pre-initialized data and no executable code.

### dc <exp1>[,<exp2>...]
Equivalent to dc.w.

### dc.b <exp1>[,<exp2>,"<string1>"',<string2>'...]
Assign the integer or string constant operands into successive bytes of memory in the current section. Any combination of integer and character string constant operands is permitted.

### dc.i <exp1>[,<exp2>...]
Assign the values of the operands into successive 32-bit words of memory in the current section. In contrast to dc.l the high and low half-words will be swapped as with the Jaguar-RISC movei instruction.

### dc.l <exp1>[,<exp2>...]
Assign the values of the operands into successive 32-bit words of memory in the current section.

### dc.w <exp1>[,<exp2>...]
Assign the values of the operands into successive 16-bit words of memory in the current section.

### dcb
Equivalent to dcb.w.

### dcb.b <exp>[,<fill>]
Insert <exp> zero or <fill> bytes into the current section.

### dcb.l <exp>[,<fill>]
Insert <exp> zero or <fill> 32-bit words into the current section.

### dcb.w <exp>[,<fill>]
Insert <exp> zero or <fill> 16-bit words into the current section.

### dphrase
Align the program counter to the next integral double phrase boundary (16 bytes).

### ds <exp>
Equivalent to dcb.w <exp>,0.

### ds.b <exp>
Equivalent to dcb.b <exp>,0.

### ds.l <exp>
Equivalent to dcb.l <exp>,0.
ds.w <exp>
   Equivalent to dcb.w <exp>, 0.
else
   Else-part of a conditional-assembly block. Refer to 'if'.
end
   End the assembly of the current file. Parsing of an include file is terminated here and assembling of the parent source commences. It also works to break the current conditional block, repetition or macro.
endif
   Ends a block of conditional assembly.
endm
   Ends a macro definition.
endr
   Ends a repetition block.
<symblo> equ <expression>
   Define a new program symbol with the name <symbol> and assign to it the value of <expression>. Defining <symbol> twice will cause an error.
even
   Align the program counter to an even value, by inserting a zero-byte when it is odd.
exitm
   Exit the current macro (proceed to endm) at this point and continue assembling the parent context. Note that this directive also resets the level of conditional assembly to a state before the macro was invoked (which means that it works as a 'break' command on all new if directives).
extern <symbol>[,,<symbol>...]
   Declare the given symbols as externally defined. Internally there is no difference to globl, as both declare the symbols, no matter if defined or not, as externally visible.
globl <symbol>[,,<symbol>...]
   Declare the given symbols as externally visible in the object file for the linker. Note that you can have the same effect by using a double-colon (::) on labels or a double-equal (==) on equate-symbols.
if <expression>
   Start of block of conditional assembly. If <expression> is true, the block between 'if' and the matching 'endif' or 'else' will be assembled. When false, ignore all lines until and 'else' or 'endif' directive is encountered. It is possible to leave such a block early from within an include file (with end) or a macro (with endm).
iif <expression>, <statement>
   A single-line conditional assembly. The <statement> will be parsed when <expression> evaluates to true (non-zero). <statement> may be a normal source line, including labels, operators and operands.
incbin "<file>"
   Inserts the binary contents of <file> into the object code at this position. The file will be searched first in the current directory, then in all paths defined by -I in the order of occurrence.
include "<file>"

Include source text of <file> at this position. The include file will be searched first in the current directory, then in all paths defined by -I in the order of occurrence.

list

The following lines will appear in the listing file, if it was requested.

long

Align the program counter to the next integral longword boundary (4 bytes), by inserting as many zero-bytes as needed.

macro <name> [,<argname>...]

Defines a macro which can be referenced by <name> (case-sensitive). The macro definition is terminated by an endm directive and may be exited by exitm. When calling a macro you may pass up to 64 arguments, separated by comma. The first ten arguments are referenced within the macro context as \1 to \9 and \0 for the tenth. Optionally you can specify a list of argument names, which are referenced with a leading backslash character (\) within the macro. The special code \~ inserts a unique id, useful for defining labels. \# is replaced by the number of arguments. \! writes the the size-qualifier (M68k) including the dot. \?<argname> expands to 1 when the named argument is specified and non-empty, otherwise it expands to 0. It is also allowed to enclose argument names in curly braces, which is useful in situations where the argument name is followed by another valid identifier character.

macundef <name> [,<name>...]

Undefine one or more already defined macros, making them unknown for the following source to assemble.

nlist

The following lines will not be visible in a listing file.

nolist

The following lines will not be visible in a listing file.

org <expression>

Sets the base address for the subsequent code and switch into absolute mode. Such a block is terminated by any section directive or by .68000 (Jaguar only).

phrase

Align the program counter to the next integral phrase boundary (8 bytes).

print <expression> [,<expression>...]

Prints strings and formatted expressions to the assembler’s console. <expression> is either a string in quotes or an expression, which is optionally preceded by special format flags:

Several flags can be used to format the output of expressions. The default is a 16-bit signed decimal.

/x     hexadecimal
/d     signed decimal
/u     unsigned decimal
/w     16-bit word
/l     32-bit longword
For example:

```
.print "Value: ", /d/1 xyz
```

qphrase Align the program counter to the next integral quad phrase boundary (32 bytes).

```
rept <expression>
The block between rept and endr will be repeated <expression> times, which has to be positive.
```

```
<symbol> set <expression>
Create a new symbol with the name <symbol> and assign the value of <expression>. If <symbol> is already assigned, it will contain a new value from now on.
```

```
text The following code and data is going into the text section, which usually is the first program section, containing the executable code.
```

5.4 Known Problems
Some known problems of this module at the moment:

− Not all Madmac, smac and extended Jaguar-Madmac directives are supported.
− Expressions are not evaluated left-to-right, but mathematically correct.
− Square-brackets ([ ]) are currently not supported to prioritize terms, as an alternative for parentheses.
− Functions (^func) are currently not supported.

5.5 Error Messages
This module has the following error messages:

− 1001: malformed immediate-if
− 1003: cannot export local symbol
− 1004: no space before operands
− 1005: too many closing parentheses
− 1006: missing closing parentheses
− 1007: missing operand
− 1008: garbage at end of line
− 1009: unknown print format flag '%c'
− 1010: invalid data operand
− 1011: print format corrupted
− 1012: identifier expected
− 1014: unexpected "%s" without "%s"
6 Oldstyle Syntax Module

This chapter describes the oldstyle syntax module suitable for some 8-bit CPUs (6502, 680x, 68HC1x, Z80, etc.), which is available with the extension oldstyle.

6.1 Legal

This module is written in 2002-2018 by Frank Wille and is covered by the vasm copyright without modifications.

6.2 Additional options for this module

This syntax module provides the following additional options:

- **autoexp** Automatically export all non-local symbols, making them visible to other modules during linking.
- **dotdir** Directives have to be preceded by a dot (.)
- **i** Ignore everything after a blank in the operand field and treat it as a comment. This option is only available when the backend does not separate its operands with blanks as well.
- **noc** Disable C-style constant prefixes.
- **noi** Disable intel-style constant suffixes.

6.3 General Syntax

Labels always start at the first column and may be terminated by a colon (colon), but don’t need to. In the last case the mnemonic needs to be separated from the label by whitespace (not required in any case, e.g. `=`).

Local labels are preceded by `.` or terminated by `$`. For the rest, any alphanumeric character including `_` is allowed. Local labels are valid between two global label definitions. It is allowed, but not recommended, to refer to any local symbol starting with `.` in the source, by preceding its name with the name of the last global symbol, which was defined before it: `global_name.local_name`.

The operands are separated from the mnemonic by whitespace. Multiple operands are separated by comma (`,`).

Make sure that you don’t define a label on the same line as a directive for conditional assembly (if, else, endif)! This is not supported.

Some CPU backends may supported multiple statements (directives or mnemonics) per line, separated by a special character (e.g. `:` for Z80).

Comments are introduced by the comment character `;`, or the first blank following the operand field when option `-i` was given. The rest of the line will be ignored.

Example:

```plaintext
mylabel instr op1,op2 ;comment
```

In expressions, numbers starting with `$` are hexadecimal (e.g. `$fb2c`). For Z80 also `&` may be used as a hexadecimal prefix, but make sure to avoid conflicts with the and-operator (either
by using parentheses or blanks). % introduces binary numbers (e.g. %1100101). Numbers starting with @ are assumed to be octal numbers, e.g. @237 (except for Z80, where it means binary). A special case is a digit followed by a #, which can be used to define an arbitrary base between 2 and 9 (e.g. 4#3012). Intel-style constant suffixes are supported: h for hexadecimal, d for decimal, o or q for octal and b for binary. Hexadecimal intel-style constants must start with a digit (prepend 0, when required). Also C-style prefixes are supported for hexadecimal (0x) and binary (0b). All other numbers starting with a digit are decimal, e.g. 1239.

6.4 Directives

The following directives are supported by this syntax module (if the CPU- and output-module allow it):

<symbol> = <expression>
Equivalent to <symbol> equ <expression>.

abyte <offset>,<exp1>[,<exp2>,"<string1>"...]
Write the integer or string constant operands into successive bytes of memory in the current section while adding the constant <offset> to each byte. Any combination of integer and character string constant operands is permitted.

addr <exp1>[,<exp2>...]  
Equivalent to word <exp1>[,<exp2>...].

align <bitcount>
Insert as much zero bytes as required to reach an address where <bit_count> low order bits are zero. For example align 2 would make an alignment to the next 32-bit boundary.

asc <exp1>[,<exp2>,"<string1>"...]
Equivalent to byte <exp1>[,<exp2>,"<string1>"...].

ascii <exp1>[,<exp2>,"<string1>"...]
See defm.

asciiz "<string1>"[,"<string2>"...]
See string.

assert <expression>[,<message>]
Display an error with the optional <message> when the expression is false.

binary <file>
Inserts the binary contents of <file> into the object code at this position. The file will be searched first in the current directory, then in all paths defined by -I or incdir in the order of occurrence.

blk <exp>[,<fill>]
Insert <exp> zero or <fill> bytes into the current section.

blkw <exp>[,<fill>]
Insert <exp> zero or <fill> 16-bit words into the current section, using the endianness of the target CPU.
bsz <exp>[,<fill>]
   Equivalent to blk <exp>[,<fill>].

byt
   Increases the program counter by one. Equivalent to blk 1,0.

byte <exp1>[,<exp2>,"<string1>"...]  
   Assign the integer or string constant operands into successive bytes of memory 
   in the current section. Any combination of integer and character string constant 
   operands is permitted.

data <exp1>[,<exp2>,"<string1>"...]  
   Equivalent to byte <exp1>[,<exp2>,"<string1>"...].

db <exp1>[,<exp2>,"<string1>"...]  
   Equivalent to byte <exp1>[,<exp2>,"<string1>"...].

dc <exp>[,<fill>]
   Equivalent to blk <exp>[,<fill>].

defb <exp1>[,<exp2>,"<string1>"...] 
   Equivalent to byte <exp1>[,<exp2>,"<string1>"...].

defc <symbol> = <expression> 
   Define a new program symbol with the name <symbol> and assign to it the 
   value of <expression>. Defining <symbol> twice will cause an error.

defl <exp1>[,<exp2>...]  
   Assign the values of the operands into successive 32-bit integers of memory in 
   the current section, using the endianess of the target CPU.

defp <exp1>[,<exp2>...]  
   Assign the values of the operands into successive 24-bit integers of memory in 
   the current section, using the endianess of the target CPU.

defm "string" 
   Equivalent to text "string".

defw <exp1>[,<exp2>...] 
   Equivalent to word <exp1>[,<exp2>...].

defb <exp1>[,<exp2>,"<string1>"...] 
   Equivalent to byte <exp1>[,<exp2>,"<string1>"...].

defw <exp1>[,<exp2>...] 
   Equivalent to word <exp1>[,<exp2>...].

defs <exp>[,<fill>]  
   Equivalent to blk <exp>[,<fill>].

dephase  
   Equivalent to rend.

dsb <exp>[,<fill>]  
   Equivalent to blk <exp>[,<fill>].
dsw <exp>[,<fill>]
   Equivalent to blkw <exp>[,<fill>].

dw <exp1>[,<exp2>...]
   Equivalent to word <exp1>[,<exp2>...].
end
   Assembly will terminate behind this line.
endif
   Ends a section of conditional assembly.
e1
   Equivalent to else.
else
   Assemble the following lines when the previous if-condition was false.
ei
   Equivalent to endif. (Not available for Z80 CPU)
endm
   Ends a macro definition.
endmac
   Ends a macro definition.
endmacro
   Ends a macro definition.
endr
   Ends a repetition block.
endrep
   Ends a repetition block.
endrepeat
   Ends a repetition block.
endstruct
   Ends a structure definition.
endstructure
   Ends a structure definition.
<> symbol> eq <expression>
   Equivalent to <symbol> equ <expression>.
<symbol> equ <expression>
   Define a new program symbol with the name <symbol> and assign to it the
   value of <expression>. Defining <symbol> twice will cause an error.
extern <symbol>[,<symbol>...]
   See global.
even
   Aligns to an even address. Equivalent to align 1.
fail <message>
   Show an error message including the <message> string. Do not generate an
   output file.
fill <exp>
   Equivalent to blk <exp>,0.
fcb <exp1>[,<exp2>,"<string1>"...]
   Equivalent to byte <exp1>[,<exp2>,"<string1>"...].
fcc "<string>"
   Equivalent to text.
fdb <exp1>[,<exp2>,"<string1>"...]  
Equivalent to word <exp1>[,<exp2>...].

global <symbol>[,[<symbol>...]]
Flag <symbol> as an external symbol, which means that <symbol> is visible to all modules in the linking process. It may be either defined or undefined.

if <expression>
Conditionally assemble the following lines if <expression> is non-zero.

ifdef <symbol>
Conditionally assemble the following lines if <symbol> is defined.

ifndef <symbol>
Conditionally assemble the following lines if <symbol> is undefined.

ifd <symbol>
Conditionally assemble the following lines if <symbol> is defined.

ifnd <symbol>
Conditionally assemble the following lines if <symbol> is undefined.

ifeq <expression>
Conditionally assemble the following lines if <expression> is zero.

ifne <expression>
Conditionally assemble the following lines if <expression> is non-zero.

ifgt <expression>
Conditionally assemble the following lines if <expression> is greater than zero.

ifge <expression>
Conditionally assemble the following lines if <expression> is greater than zero or equal.

iflt <expression>
Conditionally assemble the following lines if <expression> is less than zero.

ifle <expression>
Conditionally assemble the following lines if <expression> is less than zero or equal.

ifused <symbol>
Conditionally assemble the following lines if <symbol> has been previously referenced in an expression or in a parameter of an opcode. Issue a warning, when <symbol> is already defined. Note that ifused does not work, when the symbol has only been used in the following lines of the source.

incbin <file>[,[<offset>,[<nbytes>]]]
Inserts the binary contents of <file> into the object code at this position. When <offset> is specified, then the given number of bytes will be skipped at the beginning of the file. The optional <nbytes> argument specifies the maximum number of bytes to be read from that file. The file will be searched first in the current directory, then in all paths defined by -I or incdir in the order of occurrence.
incdir <path>
Add another path to search for include files to the list of known paths. Paths
defined with -I on the command line are searched first.

include <file>
Include source text of <file> at this position. The include file will be searched
first in the current directory, then in all paths defined by -I or incdir in the
order of occurrence.

mac <name>
Equivalent to macro <name>.

list
The following lines will appear in the listing file, if it was requested.

local <symbol>[,<symbol>...]
Flag <symbol> as a local symbol, which means that <symbol> is local for the
current file and invisible to other modules in the linking process.

macro <name>[,<argname>...]
Defines a macro which can be referenced by <name>. The <name> may also
appear on the left side of the macro directive, starting at the first column. The
macro definition is closed by an endm directive. When calling a macro you
may pass up to 9 arguments, separated by comma. Those arguments are refer-
enced within the macro context as \1 to \9, or optionally by named arguments,
which you have to specify in the operand. Argument \0 is set to the macro’s
first qualifier (mnemonic extension), when given. The special argument @ in-
serts an underscore followed by a six-digit unique id, useful for defining labels.
\() may be used as a separator between the name of a macro argument and
the subsequent text. \<symbolname> inserts the current decimal value of the
absolute symbol symbolname.

mdat <file>
Equivalent to incbin <file>.

nolist
The following lines will not be visible in a listing file.

org <expression>
Sets the base address for the subsequent code. This is equivalent to
*=<expression>.

phase <expression>
Equivalent to rorg <expression>.

repeat <expression>
Equivalent to rept <expression>.

rept <expression>
Repeats the assembly of the block between rept and endr <expression> number
of times. <expression> has to be positive.

reserve <exp>
Equivalent to blk <exp>,0.

rend Ends a rorg block of label relocation. Following labels will be based on org
again.
rmb <exp>[,<fill>]
   Equivalent to blk <exp>[,<fill>]. (Not available for 6502 CPU.)

roffs <expression>
   Sets the program counter <expression> bytes behind the start of the current
   section. The new program counter must not be smaller than the current one.
   The space will be padded with zeros.

rorg <expression>
   Relocate all labels between rorg and rend based on the new origin from
   <expression>.

section <name>[,"<attributes>"
   Starts a new section named <name> or reactivate an old one. If attributes are
   given for an already existing section, they must match exactly. The section’s
   name will also be defined as a new symbol, which represents the section’s start
   address. The "<attributes>" string may consist of the following characters:
   Section Contents:
   c    section has code
   d    section has initialized data
   u    section has uninitialized data
   i    section has directives (info section)
   n    section can be discarded
   R    remove section at link time
   a    section is allocated in memory
   Section Protection:
   r    section is readable
   w    section is writable
   x    section is executable
   s    section is sharable

<symbol> set <expression>
   Create a new symbol with the name <symbol> and assign the value of <expression>. If <symbol> is already assigned, it will contain a new value from now on.

spc <exp>
   Equivalent to blk <exp>,0.

string "<string1>"[,"<string2>"
   Like text, but adds a terminating zero-byte.

struct <name>
   Defines a structure which can be referenced by <name>. Labels within a structure definition can be used as field offsets. They will be defined as local labels of <name> and can be referenced through <name>.<label>. All directives are
allowed, but instructions will be ignored when such a structure is used. Data
definitions can be used as default values when the structure is used as initializer.
The structure name, \texttt{name}, is defined as a global symbol with the structure’s
size. A structure definition is ended by \texttt{endstruct}.

\textbf{structure} \texttt{name}
\begin{itemize}
\item Equivalent to \texttt{struct name}.
\end{itemize}

\textbf{text} \texttt{"string"}
\begin{itemize}
\item Places a single string constant operands into successive bytes of memory in the
current section. The string delimiters may be any printable ASCII character.
\end{itemize}

\textbf{weak} \texttt{symbol[,symbol\ldots]}
\begin{itemize}
\item Flag \texttt{symbol} as a weak symbol, which means that \texttt{symbol} is visible to all
modules in the linking process and may be replaced by any global symbol with
the same name. When a weak symbol remains undefined its value defaults to
0.
\end{itemize}

\textbf{wor} \texttt{exp1[,exp2\ldots]}
\begin{itemize}
\item Equivalent to \texttt{word exp1[,exp2\ldots]}.
\end{itemize}

\textbf{wrd}
\begin{itemize}
\item Increases the program counter by two. Equivalent to \texttt{blkw 1,0}.
\end{itemize}

\textbf{word} \texttt{exp1[,exp2\ldots]}
\begin{itemize}
\item Assign the values of the operands into successive 16-bit words of memory in the
current section, using the endianess of the target CPU.
\end{itemize}

\textbf{xdef} \texttt{symbol[,symbol\ldots]}
\begin{itemize}
\item See \texttt{global}.
\end{itemize}

\textbf{xlib} \texttt{symbol[,symbol\ldots]}
\begin{itemize}
\item See \texttt{global}.
\end{itemize}

\textbf{xref} \texttt{symbol[,symbol\ldots]}
\begin{itemize}
\item See \texttt{global}.
\end{itemize}

\textbf{zmb} \texttt{exp[,fill]}
\begin{itemize}
\item Equivalent to \texttt{blk exp[,fill]}.
\end{itemize}

\section{6.5 Structures}
The oldstyle syntax is able to manage structures. Structures can be defined in two ways:

\begin{verbatim}
mylabel struct[ure]
   <fields>
   endstruct[ure]
\end{verbatim}

or:

\begin{verbatim}
struct[ure] mylabel
   <fields>
   endstruct[ure]
\end{verbatim}

Any directive is allowed to define the structure fields. Labels can be used to define offsets
into the structure. The initialized data is used as default value, whenever no value is given
for a field when the structure is referenced.

Some examples of structure declarations:
struct point
  x    db 4
  y    db 5
  z    db 6
endstruct

This will create the following labels:

point.x ; 0   offsets
point.y ; 1
point.z ; 2
point  ; 3   size of the structure

The structure can be used by optionaly redefining the fields value:

point1 point
point2 point 1, 2, 3
point3 point ,,4

is equivalent to

point1
db 4
db 5
db 6

point2
db 1
db 2
db 3

point3
db 4
db 5
db 4

6.6 Known Problems

Some known problems of this module at the moment:

− Addresses assigned to org or to the current pc symbol ’*’ (on the z80 the pc symbol is ’$’) must be constant.
− Expressions in an if directive must be constant.

6.7 Error Messages

This module has the following error messages:

− 1001: syntax error
− 1002: invalid extension
− 1003: no space before operands
− 1004: too many closing parentheses
− 1005: missing closing parentheses
− 1006: missing operand
- 1007: garbage at end of line
- 1008: %c expected
- 1009: invalid data operand
- 1010: , expected
- 1011: identifier expected
- 1012: illegal escape sequence %c
- 1013: unexpected "%s" without "%s"
- 1021: cannot open binary file "%s"
- 1023: alignment too big
- 1024: label <%%> has already been defined
- 1025: skipping instruction in struct init
- 1026: last %d bytes of string constant have been cut
7 Test output module

This chapter describes the test output module which can be selected with the -Ftest option.

7.1 Legal
This module is written in 2002 by Volker Barthelmann and is covered by the vasm copyright without modifications.

7.2 Additional options for this module
This output module provides no additional options.

7.3 General
This output module outputs a textual description of the contents of all sections. It is mainly intended for debugging.

7.4 Restrictions
None.

7.5 Known Problems
Some known problems of this module at the moment:
   – None.

7.6 Error Messages
This module has the following error messages:
   – None.
8 ELF output module

This chapter describes the ELF output module which can be selected with the `-Felf` option.

8.1 Legal
This module is written in 2002-2016 by Frank Wille and is covered by the vasm copyright without modifications.

8.2 Additional options for this module

- `-keepempty`
  Do not delete empty sections without any symbol definition.

8.3 General
This output module outputs the ELF (Executable and Linkable Format) format, which is a portable object file format which works for a variety of 32- and 64-bit operating systems.

8.4 Restrictions
The ELF output format, as implemented in vasm, currently supports the following architectures:
- PowerPC
- M68k
- ARM
- i386
- x86_64
- Jaguar RISC

The supported relocation types depend on the selected architecture.

8.5 Known Problems
Some known problems of this module at the moment:
- None.

8.6 Error Messages
This module has the following error messages:
- 3002: output module doesn’t support cpu <name>
- 3003: write error
- 3005: reloc type <m>, size <n>, mask <mask> (symbol <sym> + <offset>) not supported
- 3006: reloc type <n> not supported
- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx
9 a.out output module

This chapter describes the a.out output module which can be selected with the -Faout option.

9.1 Legal

This module is written in 2008-2012 by Frank Wille and is covered by the vasm copyright without modifications.

9.2 Additional options for this module

-mid=<machine id>

Sets the MID field of the a.out header to the specified value. The MID defaults to 2 (Sun020 big-endian) for M68k and to 100 (PC386 little-endian) for x86.

9.3 General

This output module outputs the a.out (assembler output) format, which is an older 32-bit format for Unix-like operating systems, originally invented by AT&T.

9.4 Restrictions

The a.out output format, as implemented in vasm, currently supports the following architectures:

- M68k
- i386

The following standard relocations are supported by default:

- absolute, 8, 16, 32 bits
- pc-relative, 8, 16, 32 bits
- base-relative

Standard relocations occupy 8 bytes and don’t include an addend, so they are not suitable for most RISC CPUs. The extended relocations format occupies 12 bytes and also allows more relocation types.

9.5 Known Problems

Some known problems of this module at the moment:

- Support for stab debugging symbols is still missing.
- The extended relocation format is not supported.

9.6 Error Messages

This module has the following error messages:

- 3004: section attributes <attr> not supported
- 3008: output module doesn’t allow multiple sections of the same type
- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx
10 TOS output module

This chapter describes the TOS output module which can be selected with the \texttt{-Ftos} option.

10.1 Legal

This module is written in 2009-2014 by Frank Wille and is covered by the \texttt{vasm} copyright without modifications.

10.2 Additional options for this module

\texttt{-monst} Write Devpac "MonST"-compatible symbols.

\texttt{-tos-flags=<flags>}

Sets the flags field in the TOS file header. Defaults to 0. Overwrites a TOS flags definition in the assembler source.

10.3 General

This module outputs the TOS executable file format, which is used on Atari 16/32-bit computers with 68000 up to 68060 CPU. The symbol table uses the DRI format.

10.4 Restrictions

- All symbols must be defined, otherwise the generation of the executable fails. Unknown symbols are listed by \texttt{vasm}.
- The only relocations allowed in this format are 32-bit absolute.

Those are restrictions of the output format, not of \texttt{vasm}.

10.5 Known Problems

Some known problems of this module at the moment:
- None.

10.6 Error Messages

This module has the following error messages:
- 3004: section attributes \texttt{<attr>} not supported
- 3005: reloc type \texttt{%d}, size \texttt{%d}, mask \texttt{0x%lx} (symbol \texttt{%s + 0x%lx}) not supported
- 3006: reloc type \texttt{%d} not supported
- 3007: undefined symbol \texttt{<%s>}
- 3008: output module doesn’t allow multiple sections of the same type
- 3010: section \texttt{<%s>}: alignment padding \texttt{(%lu)} not a multiple of \texttt{%lu} at \texttt{0x%llx}
- 3011: weak symbol \texttt{<%s>} not supported by output format, treating as global
11 Amiga output module

This chapter describes the AmigaOS hunk-format output module which can be selected with the `-Fhunk` option to generate objects and with the `-Fhunkexe` option to generate executable files.

11.1 Legal

This module is written in 2002-2019 by Frank Wille and is covered by the vasm copyright without modifications.

11.2 Additional options for this module

- `hunkpad=<code>`
  Sets a two-byte code used for aligning a code hunk to the next 32-bit border. Defaults to 0x4e71 on M68k to allow linking of functions which extend over two object files.

- `keepempty`
  Do not delete empty sections without any symbol definition.

- `kick1hunks`
  Use only those hunk types and external reference types which have been valid at the time of Kickstart 1.x for compatibility with old assembler sources and old linkers. For example: no longer differentiate between absolute and relative references. In executables it will prevent the assembler from using 16-bit relocation offsets in hunks and rejects 32-bit PC-relative relocations.

- `linedebug`
  Automatically generate an SAS/C-compatible LINE DEBUG hunk for the input source. Overrides any line debugging directives from the source.

These options are valid for the `hunkexe` module only:

- `databss`
  Try to shorten sections in the output file by removing zero words without relocation from the end. This technique is only supported by AmigaOS 2.0 and higher.

11.3 General

This output module outputs the `hunk` object (standard for M68k and extended for PowerPC) and `hunkexe` executable format, which is a proprietary file format used by AmigaOS and WarpOS.

The `hunkexe` module will generate directly executable files, without the need for another linker run. But you have to make sure that there are no undefined symbols, common symbols, or unusual relocations (e.g. small data) left.

It is allowed to define sections with the same name but different attributes. They will be regarded as different entities.
11.4 Restrictions

The hunk/hunkexe output format is only intended for M68k and PowerPC cpu modules and will abort when used otherwise.

The hunk module supports the following relocation types:

- absolute, 32-bit
- absolute, 16-bit
- absolute, 8-bit
- relative, 8-bit
- relative, 14-bit (mask 0xffffc) for PPC branch instructions.
- relative, 16-bit
- relative, 24-bit (mask 0x3fffffc) for PPC branch instructions.
- relative, 32-bit
- base-relative, 16-bit
- common symbols are supported as 32-bit absolute and relative references

The hunkexe module supports absolute 32-bit relocations only.

11.5 Known Problems

Some known problems of this module at the moment:

- The hunkexe module won’t process common symbols and allocate them in a BSS section. Use a real linker for that.

11.6 Error Messages

This module has the following error messages:

- 3001: multiple sections not supported by this format
- 3002: output module doesn’t support cpu <name>
- 3003: write error
- 3004: section attributes <attr> not supported
- 3005: reloc type <m>, size <n>, mask <mask> (symbol <sym> + <offset>) not supported
- 3006: reloc type <n> not supported
- 3009: undefined symbol <%s> at %s+0x%lx, reloc type %d
- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx
- 3011: weak symbol <%s> not supported by output format, treating as global
- 3014: data definition following a databss space directive
12 X68k output module

This chapter describes the Xfile output module which can be selected with the \-Fxfile\ option.

12.1 Legal

This module is written in 2018 by Frank Wille and is covered by the Vasm copyright without modifications.

12.2 Additional options for this module

None.

12.3 General

This module outputs the Xfile executable file format, which is used on Sharp X68000 16/32-bit computer with 68000 up to 68040 CPU.

12.4 Restrictions

- The source must not define more than one code, data and bss section each. More complex sources with \.rdata\ or \.stack\ sections require a linker.
- All symbols must be defined, otherwise the generation of the executable fails. Unknown symbols are listed by Vasm.
- The only relocations allowed in this format are 32-bit absolute.

12.5 Known Problems

Some known problems of this module at the moment:
- None.

12.6 Error Messages

This module has the following error messages:
- 3004: section attributes \<attr\> not supported
- 3005: reloc type \%d, size \%d, mask 0x%lx (symbol \%s + 0x%lx) not supported
- 3006: reloc type \%d not supported
- 3007: undefined symbol \<%s\>
- 3008: output module doesn't allow multiple sections of the same type
- 3011: weak symbol \<%s\> not supported by output format, treating as global
- 3013: reloc type \%d, mask 0x%lx to symbol \%s + 0x%lx does not fit into \%u bits
13 vobj output module

This chapter describes the simple binary output module which can be selected with the -Fvobj option.

13.1 Legal

This module is written in 2002-2014 by Volker Barthelmann and is covered by the vasm copyright without modifications.

13.2 Additional options for this module

None.

13.3 General

This output module outputs the vobj object format, a simple portable proprietary object file format of vasm.

As this format is not yet fixed, it is not described here.

13.4 Restrictions

None.

13.5 Known Problems

Some known problems of this module at the moment:

- None.

13.6 Error Messages

This module has the following error messages:

- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx
14 Simple binary output module

This chapter describes the simple binary output module which can be selected with the \texttt{-Fbin} option.

14.1 Legal

This module is written in 2002-2009,2013 by Volker Barthelmann and is covered by the \texttt{vasm} copyright without modifications.

14.2 Additional options for this module

\texttt{-cbm-prg}  Writes a Commodore PRG header in front of the output file, which consists of two bytes in little-endian order, defining the load address of the program.

14.3 General

This output module outputs the contents of all sections as simple binary data without any header or additional information. When there are multiple sections, they must not overlap. Gaps between sections are filled with zero bytes. Undefined symbols are not allowed.

14.4 Known Problems

Some known problems of this module at the moment:

- None.

14.5 Error Messages

This module has the following error messages:

- 3001: sections must not overlap
- 3007: undefined symbol \texttt{<%s>}
- 3010: section \texttt{<%s>}: alignment padding (%lu) not a multiple of %lu at 0x%llx
15 Motorola srecord output module

This chapter describes the Motorola srecord output module which can be selected with the 
\texttt{-Fsrec} option.

15.1 Legal

This module is written in 2015 by Joseph Zatarski and is covered by the \texttt{vasm} copyright 
without modifications.

15.2 Additional options for this module

\texttt{-s19} \hspace{0.5cm} Writes \texttt{S1} data records and \texttt{S9} trailers with 16-bit addresses.
\texttt{-s28} \hspace{0.5cm} Writes \texttt{S2} data records and \texttt{S8} trailers with 24-bit addresses.
\texttt{-s37} \hspace{0.5cm} Writes \texttt{S3} data records and \texttt{S7} trailers with 32-bit addresses. This is the default 
setting.

\texttt{-exec[]=\langle\text{symbol}\rangle} \hspace{0.5cm} Use the given symbol \texttt{\langle\text{symbol}\rangle} as entry point of the program. This start 
address will be written into the trailer record, which is otherwise zero. When 
the symbol assignment is omitted, then the default symbol \texttt{\texttt{start}} will be used.

15.3 General

This output module outputs the contents of all sections in Motorola srecord format, which 
is a simple ASCII output of hexadecimal digits. Each record starts with \texttt{S} and a one-digit 
ID. It is followed by the data and terminated by a checksum and a newline character. Every 
section starts with a new header record.

15.4 Known Problems

Some known problems of this module at the moment:

\begin{itemize}
  \item A new header is written for every new section. This may cause compatibility issues.
\end{itemize}

15.5 Error Messages

This module has the following error messages:

\begin{itemize}
  \item 3007: undefined symbol \texttt{\langle\text{symbol}\rangle}
  \item 3010: section \texttt{\langle\text{symbol}\rangle}: alignment padding (%lu) not a multiple of %lu at 0x%llx
  \item 3012: address 0x%llx out of range for selected format
\end{itemize}
16  m68k cpu module

This chapter documents the backend for the Motorola M68k/CPU32/ColdFire micropro-cessor family.

16.1 Legal

This module is written in 2002-2019 by Frank Wille and is covered by the vasm copyright without modifications.

16.2 Additional options for this module

Note, that the order on the command line may be important when specifying options. For example, if you specify -devpac compatibility mode behind enabling some optimization options, the Devpac-mode might disable these optimizations again.

This module provides the following additional options:

16.2.1 CPU selections

-\texttt{-m68000} Generate code for the MC68000 CPU.
-\texttt{-m68008} Generate code for the MC68008 CPU.
-\texttt{-m68010} Generate code for the MC68010 CPU.
-\texttt{-m68020} Generate code for the MC68020 CPU.
-\texttt{-m68030} Generate code for the MC68030 CPU.
-\texttt{-m68040} Generate code for the MC68040 CPU.
-\texttt{-m68060} Generate code for the MC68060 CPU.
-\texttt{-m68020up} Generate code for the MC68020-68060 CPU. Be careful with instructions like PFLUSH A, which exist on 68030 and 68040/060 with a different opcode (vasm will use the 040/060 version).
-\texttt{-m68080} Generate code for the Apollo Core AC68080 FPGA CPU.
-\texttt{-mcpu32} Generate code for the CPU32 family (MC6833x, MC6834x, etc.).
-\texttt{-mcf5...}
-\texttt{-m5...} Generate code for a ColdFire family CPU. The following types are recognized: 5202, 5204, 5206, 520x, 5206e, 5207, 5208, 5210a, 5211a, 5212, 5213, 5214, 5216, 5224, 5225, 5232, 5233, 5234, 5235, 523x, 5249, 5250, 5253, 5270, 5271, 5272, 5274, 5275, 5280, 5281, 528x, 52221, 52553, 52230, 52231, 52232, 52233, 52234, 52235, 52252, 52254, 52255, 52256, 52258, 52259, 52274, 52277, 5307, 5327, 5328, 5329, 532x, 5372, 5373, 537x, 53011, 53012, 53013, 53014, 53015, 53016, 53017, 5301x, 5407, 5470, 5471, 5472, 5473, 5474, 5475, 547x, 5480, 5481, 5482, 5483, 5484, 5485, 548x, 54450, 54451, 54452, 54453, 5445x.
-\texttt{-mcfv2} Generate code for the V2 ColdFire core. This option selects ISA_A (no hard-ware division or MAC), which is the most limited ISA supported by 5202, 5204 and 5206. All other ColdFire chips are backwards compatible to V2.
-mcfv3  Generate code for the V3 ColdFire core. This option selects ISA_A+, hardware division MAC and EMAC instructions, which are supported by nearly all V3 CPUs, except the 5307.

-mcfv4  Generate code for the V4 ColdFire core. This option selects ISA_B and MAC as supported by the 5407.

-mcfv4e Generate code for the V4e ColdFire core. This option selects ISA_B, USP-, FPU-, MAC- and EMAC-instructions (no hardware division) as supported by all 547x and 548x CPUs.

-m68851 Generate code for the MC68851 MMU. May be used in combination with another -m option.

-m68881 Generate code for the MC68881 FPU. May be used in combination with another -m option.

-m68882 Generate code for the MC68882 FPU. May be used in combination with another -m option.

-no-fpu Ignore any FPU options or directives, which has the effect that no 68881/2 FPU instructions will be accepted. This option can override the default of -gas to enable the FPU.

16.2.2 Optimization options

-no-opt Disable all optimizations. Can be seen as a main switch to ignore all other optimization options on the command line and in the source.

-opt-allbra When specified the assembler will also try to optimize branch instructions which already have a valid size extension. This option is automatically enabled in -phxass mode.

-opt-brajmp Translate relative branch instructions, whose destination is in a different section, into absolute jump instructions.

-opt-clr Enables optimization from MOVE #0,<ea> into CLR <ea> for the MC68000. Note that CLR will execute a read-modify-write cycle on the 68000, so it is disabled by default. With 68010 and higher this is a generic standard optimization.

-opt-fconst Floating point constants are loaded with the lowest precision possible. This means that MOVE.D #1.0,FP0 would be optimized to MOVE.S #1.0,FP0, because it is faster and shorter at the same precision. The optimization will be performed on all FPU instructions with immediate addressing mode. When an FDIV-family instruction (FSDIV, FDDIV, FSQDIV) is detected it will additionally be checked if the immediate constant is a power of 2 and then converted into FMUL #1/c,FPn.

-opt-jbra JMP and JSR instructions to external labels will be converted into BRA.L and BSR.L, when the selected CPU is 68020 or higher (or CPU32).
Chapter 16: m68k cpu module

-opt-lsl Allows optimization of LSL #1 into ADD. It is also needed to optimize ASL #2 and LSL #2 into two ADD instructions (together with -opt-speed). These optimizations may modify the V-flag, which might not be intended.

-opt-movem Enables optimization from MOVEM <ea>,Rn into MOVE <ea>,Rn (or the other way around). This optimization will modify the flags, when the destination is no address register.

-opt-mul Immediate multiplication factors, which are a power of two (from 2 to 256), are optimized to shifts. Multiplications with zero are replaced by a MOVEQ #0,Dn, with -1 are replaced by a NEG.L Dn and with 1 by EXT.L Dn or TST.L Dn (long-form). Not all optimizations are available for all cpu types (e.g. MULU.W can only be optimized on ColdFire by using the MVZ.W instruction. This optimization will leave the flags in a different state as can normally be expected after a multiplication instruction, and the size of the optimized code may be bigger than before in a few situations (e.g. MULS.W #4,Dn). The latter will additionally require the -opt-speed flag.

-opt-div Unsigned immediate divisors, which are a power of two (from 2 to 256), are optimized to shifts. Divisions by 1 are replaced by TST.L Dn (32-bit) or MVZ.W Dn,Dn (16-bit, ColdFire only). Divisions by -1 are replaced by a combination of NEG.W Dn and MVZ.W Dn,Dn (16-bit, ColdFire only). This optimization will leave the flags in a different state as can normally be expected after a division instruction.

-opt-pea Enables optimization from MOVE #x,-(SP) into PEA x. This optimization will leave the flags unmodified, which might not be intended.

-opt-speed Optimize for speed, even if this would increase code size. For example it enables optimization of ASL.W #2,Dn into two ADD.W Dn,Dn instructions. Or MULS.W #4,Dn into EXT.L Dn + ASL.L #2,Dn + NEG.L Dn. Generally the assembler will never optimize a single into multiple instructions without this option.

-opt-st Enables optimization from MOVE.B #-1,<ea> into ST <ea>. This optimization will leave the flags unmodified, which might not be intended.

-sc All JMP and JSR instructions to external labels will be converted into 16-bit PC-relative jumps.

-sd References to absolute symbols in a small data section (named "._MERGED") are optimized into a base-relative addressing mode using the current base register set by an active NEAR directive. This option is automatically enabled in -phxass mode.

-showcrit Print all critical optimizations which have side effects. Among those are -opt-lsl, -opt-mul, -opt-st, -opt-pea, -opt-movem and -opt-clr.

-showopt Print all optimizations and translations vasm is doing (same as opt ow+).

In its default setting (no -devpac or -phxass option) vasm performs the following optimizations:
Absolute to PC-relative.
- Branches without explicit size.
- Displacements (32 to 16 bit, \(0, An\) to \(An\), etc).
- Optimize floating point constants to the lowest possible precision.
- Many instruction optimizations which are safe.

### 16.2.3 Other options

- **-conv-brackets**
  Brackets ('[' and ']') in an operand are automatically converted into parentheses ('(' and ')') as long as the CPU is 68000 or 68010. This is a compatibility option for some old assemblers.

- **-devpac**
  All options are initially set to be Devpac compatible. Which means that all optimizations are disabled, no debugging symbols will be written and vasm will warn about any optimization being done. When symbol output is enabled by \texttt{opt d+}, then the TOS symbol table defaults to standard DRI format (limited to 8 characters). Shift-right operations are performed using an unsigned 32-bit value. Other options are the same as vasm's defaults. The symbol \_\_G2 is defined, which contains information about the selected cpu type. The symbol \_\_LK reflects the type of output file generated. Which is 0 for TOS executables, 4 for Amiga executables and 3 for Amiga object files. All other formats are represented by 99, as they are unknown to Devpac. It will also automatically enable \texttt{-guess-ext}.

- **-elfregs**
  Register names are preceded by a '%' to prevent confusion with symbol names.

- **-gas**
  Enable additional GNU-as compatibility mnemonics, like \texttt{mov}, \texttt{movm} and \texttt{jra}. Also accepts | instead of ; for comments. GNU-as compatibility mode selects the 68020 CPU and 68881/2 FPU by default and enables \texttt{-opt-jbra}.

- **-guess-ext**
  Accept illegal size extensions for an instruction, as long as the instruction is unsized or there is just a single size possible. This is the default setting in PhxAss and Devpac compatibility mode.

- **-kick1hunks**
  Prevents optimization of JMP/JSR to 32-bit PC-relative (BRA/BSR).

- **-phxass**
  PhxAss-compatibility mode. The "current PC symbol" (e.g. * in mot-syntax module) is set to the instruction’s address + 2 whenever an instruction is parsed. According to the current cpu setting the symbols \_\_CPU, \_\_FPU and \_\_MMU are defined. JMP/JSR (label,PC) will never be optimized (into a branch, for example). It will also automatically enable \texttt{-opt-allbra}, \texttt{-sd} and \texttt{-guess-ext}.

- **-rangewarnings**
  Values which are out of range usually produce an error. With this option the errors 2026, 2030, 2033 and 2037 will be displayed as a warning, allowing the user to create an object file.

- **-sdreg=\textless n\textgreater**
  Set the small data base register to \(An\). \(\textless n\textgreater\) is valid between 2 and 6.
Additionally allow immediate operands to be prefixed by & instead of just by #. This syntax was used by the SGS assembler.

Allow redefining register symbols with EQUR. This should only be used for compatibility with old sources. Not many assemblers support that.

16.3 General

This backend accepts M68k and CPU32 instructions as described in Motorola’s M68000 family Programmer’s Reference Manual. Additionally it supports ColdFire instructions as described in Motorola’s ColdFire Microprocessor Family Programmer’s Reference Manual. The syntax for the scale factor in ColdFire MAC instructions is << for left- and >> for right-shift. The scale factor may be appended as an optional operand, when needed. Example: mac d0.1, d1.u,<<.

The mask flag in MAC instructions is written as & and is appended directly to the effective address operand. Example: mac d0, d1, (a0)&, d2.

The target address type is 32bit.

Default alignment for instructions is 2 bytes. The default alignment for data is 2 bytes, when the data size is larger than 8 bits.

Depending on the selected cpu type the __VASM symbol will have a value defined by the following bits:

bit 0  MC68000 instruction set. Also used by MC6830x, MC68322, MC68356.
bit 1  MC68010 instruction set.
bit 2  MC68020 instruction set.
bit 3  MC68030 instruction set.
bit 4  MC68040 instruction set.
bit 5  MC68060 instruction set.
bit 6  MC68881 or MC68882 FPU.
bit 7  MC68851 PMMU.
bit 8  CPU32. Any MC6833x or MC6834x CPU.
bit 9  ColdFire ISA_A.
bit 10 ColdFire ISA_A+.
bit 11  ColdFire ISA_B.
bit 12  ColdFire ISA_C.
bit 13 ColdFire hardware division support.
bit 14 ColdFire MAC instructions.
bit 15 ColdFire enhanced MAC instructions.
bit 16 ColdFire USP register.
bit 17  ColdFire FPU instructions.
bit 18  ColdFire MMU instructions.
bit 20  Apollo Core AC68080 instruction set.

16.4 Extensions
This backend extends the selected syntax module by the following directives:

.sdreg <An>
Equivalents to near <An>.

basereg <expression>,<An>
Starts a block of base-relative addressing through register An (remember that A7 is not allowed as a base register). The programmer has to make sure that <expression> is placed into An first, while the assembler automatically subtracts <expression>, which is usually a program label with an optional offset, from each displacement in a (d,An) addressing mode. basereg has priority over the near directive. Its effect can be suspended with the endb directive. It is allowed to use several base registers in parallel.

cpu32  Generate code for the CPU32 family.
endb <An>  Ends a basereg block and suspends its effect onto the specified base register An. It may be reused with a different base expression thereafter (refer to basereg).

far  Disables small data (base-relative) mode. All data references will be absolute.

fpu <cpID>
Enables 68881/68882 FPU code generation. The <cpID> is inserted into the FPU instructions to select the correct coprocessor. Note that <cpID> is always 1 for the on-chip FPUs in the 68040 and 68060. A <cpID> of zero will disable FPU code generation.

initnear  Initializes the selected small data register. In contrast to PhxAss, where this directive comes from, just a reference to _LinkerDB is generated, which has to be resolved by a linker.

machine <cpu_type>
Makes the assembler generate code for <cpu_type>, which can be the following: 68000, 68010, 68020, 68030, 68040, 68060, 68080, 68851, 68881, 68882, cpu32. And various ColdFire CPUs, starting with 5. . . .
mcf5... Generate code for a ColdFire CPU. The recognized models are listed in the assembler-options section.

near [<An>]
Enables small data (base-relative) mode and sets the base register to An. near without an argument will re-activate a previously defined small data mode, which might have been switched off by a far directive.

near code All JMP and JSR instructions to external labels will be converted into 16-bit PC-relative jumps. The small code mode can be switched off by a far directive.

opt <option>[,<option>...]
Sets Devpac-compatible options. When option ~phxass is given, then it will parse PhxAss options instead (which is discouraged for new code, so there is no detailed description here). Most supported Devpac2-style options are always suffixed by a + or - to enable or disable the option:

a Automatically optimize absolute to PC-relative references. Default is off in Devpac-compatibility mode, otherwise on.

c Case-sensitivity for all symbols, instructions and macros. Default is on.

d Include all symbols for debugging in the output file. May also generate line debugging information in some output formats. Default is off in Devpac-compatibility mode, otherwise on.

l Generate a linkable object file. The default is defined by the selected output format via the assembler’s -F option. This option was supported by Devpac-Amiga only.

o Enable all optimizations (o1 to o12), or disable all optimizations. The default is that all are disabled in Devpac-compatibility mode and enabled otherwise. When running in native vasm mode this option will also enable PC-relative (opt a) and the following safe vasm-specific optimizations (see below): og, of.

o1 Optimize branches without an explicit size extension.

o2 Standard displacement optimizations (e.g. (0,An) -> (An)).

o3 Optimize absolute addresses to short words.

o4 Optimize move.l to moveq.

o5 Optimize add #x and sub #x into their quick forms.

o6 No effect in vasm.

o7 Convert bra.b to nop, when branching to the next instruction.

o8 Optimize 68020+ base displacements to 16 bit.

o9 Optimize 68020+ outer displacements to 16 bit.

o10 Optimize add/sub #x,An to lea.

o11 Optimize lea (d,An),An to addq/subq.
Optimize `<op>.l #x, An` to `<op>.w #x, An`.

Show all optimizations being performed. Default is on in Devpac-compatibility mode, otherwise off.

Check if code is position independent. This will cause an error on each relocation being required. Default is off.

Include symbols in listing file. Default is on.

Check size and type of all expressions. Default is on.

Show assembler warnings. Default is on.

For Amiga hunk format objects `x+` strips local symbols from the symbol table (symbols without `xdef`). For Atari TOS executables this will enable the extended (HiSoft) DRI symbol table format, which allows symbols with up to 22 characters. DRI standard only supports 8 characters.

Devpac options without +/- suffix:

Sets the output format (Devpac Atari only). Currently without effect.

Sets the CPU type to any model vasm supports (original Devpac only allowed 68000-68040, 68332, 68881, 68882 and 68851).

Also the following Devpac3-style options are supported:

Corresponds to `a+`.

Corresponds to `c+`.

Corresponds to `p+`.

Corresponds to `d+`.

Corresponds to `s+`.

Corresponds to `t+`.

Corresponds to `w+`.

Corresponds to `x+`.

Corresponds to `a-`.

Corresponds to `c-`.

Corresponds to `p-`.

Corresponds to `d-`.

Corresponds to `s-`.

Corresponds to `t-`.

Corresponds to `w-`. 
noxdebug Corresponds to x-.

The following options are vasm specific and should not be used when writing portable source. Using opt o+ or opt o- in Devpac mode only toggles og, of and oj.

ob Convert absolute jumps to external labels into long-branches (refer to -opt-jbra).

oc Enable optimizations to CLR (refer to -opt-clr).

od Enable optimization of divisions into shifts (refer to -opt-div).

of Enable immediate float constant optimizations (refer to -opt-fconst).

og Enable generic vasm optimizations. All optimizations which cannot be controlled by another option.

oj Enable branch to jump translations (refer to -opt-brajmp).

ol Enable shift optimizations to ADD (refer to -opt-lsl).

om Enable MOVEM optimizations (refer to -opt-movem).

on Enable small data optimizations. References to absolute symbols in a small data section (named "_.MERGED") are optimized into a base-relative addressing mode (refer to -sd).

op Enable optimizations to PEA (refer to -opt-pea).

os Optimize for speed before optimizing for size (refer to -opt-speed).

ot Enable optimizations to ST (refer to -opt-st).

ox Enable optimization of multiplications into shifts (refer to -opt-mul).

The default state is 'off' for all those vasm specific options.

The following directives are only available for the Motorola syntax module:

**<symbol> equr <Rn>**

Define a new symbol named <symbol> and assign the data or address register Rn, which can be used from now on in operands. When 68080 code generation is enabled, also Bn base address registers are allowed to be assigned. Note that a register symbol must be defined before it can be used!

**<symbol> equrl <reglist>**

Equivalents to <symbol> reg <reglist>.

**<symbol> fequr <FPn>**

Define a new symbol named <symbol> and assign the FPU register FPn, which can be used from now on in operands. Note that a register symbol must be defined before it can be used!

**<symbol> fequrl <reglist>**

Equivalents to <symbol> freg <reglist>.
<symbol> freg <reglist>
Defines a new symbol named <symbol> and assign the FPU register list <reglist> to it. Registers in a list must be separated by a slash (/) and ranges or registers can be defined by using a hyphen (−). Examples for valid FPU register lists are: fp0-fp7, fp1-3/fp5/fp7, fpiar/fpcr.

<symbol> reg <reglist>
Defines a new symbol named <symbol> and assign the register list <reglist> to it. Registers in a list must be separated by a slash (/) and ranges or registers can be defined by using a hyphen (−). Examples for valid register lists are: d0-d7/a0-a6, d3-6/a0/a1/a4-5.

### 16.5 Optimizations

This backend performs the following operand optimizations:

- (0,An) optimized to (An).
- (d16,An) translated to (bd32,An,ZDn.w), when d16 is not between -32768 and 32767 and the selected CPU allows it (68020 up or CPU32).
- (d16,PC) translated to (bd32,PC,ZDn.w), when d16 is not between -32768 and 32767 and the selected CPU allows it (68020 up or CPU32).
- (d8,An,Rn) translated to (bd,An,Rn), when d8 is not between -128 and 127 and the selected CPU allows it (68020 up or CPU32).
- (d8,PC,Rn) translated to (bd,PC,Rn), when d8 is not between -128 and 127 and the selected CPU allows it (68020 up or CPU32).
- <exp>.l optimized to <exp>.w, when <exp> is absolute and between -32768 and 32767.
- <exp>.w translated to <exp>.l, when <exp> is a program label or absolute and not between -32768 and 32767.
- (0,An,...) optimized to (An,...) (which means the base displacement will be suppressed). This allows further optimization to (An), when the index is suppressed.
- (bd16,An,...) translated to (bd32,An,...), when bd16 is not between -32768 and 32767.
- (bd32,An,...) optimized to (bd16,An,...), when bd32 is between -32768 and 32767, and the index is suppressed (zero-Rn).
- (An,ZRn) optimized to (An), when the index is suppressed.
- (0,PC,...) optimized to (PC,...) (which means the base displacement will be suppressed).
- (bd16,PC,...) translated to (bd32,PC,...), when bd16 is not between -32768 and 32767.
- (bd32,PC,...) optimized to (bd16,PC,...), when bd32 is between -32768 and 32767, and the index is suppressed (zero-Rn).
- ([0,Rn,...],...) optimized to ([An,...],...) (which means the base displacement will be suppressed).
Chapter 16: m68k cpu module

- \([\text{bd16}, \text{Rn}, \ldots], \ldots\) translated to \([\text{bd32}, \text{An}, \ldots], \ldots\), when \text{bd16} is not between -32768 and 32768.
- \([\text{bd32}, \text{Rn}, \ldots], \ldots\) optimized to \([\text{bd16}, \text{An}, \ldots], \ldots\), when \text{bd32} is between -32768 and 32768.
- \([\ldots], 0\) optimized to \([\ldots]\) (which means the outer displacement will be suppressed).
- \([\ldots], \text{od16}\) translated to \([\ldots], \text{od32}\), when \text{od16} is not between -32768 and 32767.
- \([\ldots], \text{od32}\) translated to \([\ldots], \text{od16}\), when \text{od32} is between -32768 and 32767.

Note that an operand optimization will only take place when a displacement's size was not enforced by the programmer (e.g. \((4.l, a0)\))!

This backend performs the following instruction optimizations:

- \(<\text{op}\.L \#x, \text{An}>\) optimized to \(<\text{op}\.W \#x, \text{An}>\), when \(x\) is between -32768 and 32767.
- \(<\text{ADD}.? \#x, <\text{ea}>\) optimized to \(<\text{ADDQ}.? \#x, <\text{ea}>\), when \(x\) is between 1 and 8.
- \(<\text{ADD}.? \#x, <\text{ea}>\) optimized to \(<\text{SUBQ}.? \#x, <\text{ea}>\), when \(x\) is between -1 and -8.
- \(<\text{ADDA}.? \#0, \text{An}>\) and \(<\text{SUBA}.? \#0, \text{An}>\) will be deleted.
- \(<\text{ADDA}.? \#x, \text{An}>\) optimized to \(<\text{LEA} (x, \text{An}), \text{An}>\), when \(x\) is between -32768 and 32767.
- \(<\text{ANDI}.L \#fff, \text{Dn}>\) optimized to \(<\text{MVZ}.B \text{Dn}, \text{Dn}>\), for ColdFire ISA_B/C.
- \(<\text{ANDI}.L \#ffff, \text{Dn}>\) optimized to \(<\text{MVZ}.W \text{Dn}, \text{Dn}>\), for ColdFire ISA_B/C.
- \(<\text{ANDI}.? \#0, <\text{ea}>\) optimized to \(<\text{CLR}.? <\text{ea}>\), when allowed by the option -opt-clr or a different CPU than the MC68000 was selected.
- \(<\text{ANDI}.? \#-1, <\text{ea}>\) optimized to \(<\text{TST}.? <\text{ea}>\).
- \(<\text{ASL}.? \#1, \text{Dn}>\) optimized to \(<\text{ADD}.? \text{Dn}, \text{Dn}>\), for 68000 and 68010.
- \(<\text{ASL}.? \#2, \text{Dn}>\) optimized into a sequence of two \(<\text{ADD}.? \text{Dn}, \text{Dn}>\) for 68000 and 68010, when the operation size is either byte or word and the options -opt-speed and -opt-lsl are given.
- \(<\text{B<cc> <label>}>\) translated into a combination of \(<\text{B<!cc> *+8}>\) and \(<\text{JMP <label>}>\), when \(<\text{label}>\) is not defined in the same section (and option -opt-brajmp is given), or outside the range of -32768 to 32767 bytes from the current address when the selected CPU is not 68020 up, CPU32 or ColdFire ISA_B/C.
- \(<\text{B<cc> <label>}>\) is automatically optimized to 8-bit, 16-bit or 32-bit (68020 up, CPU32, MCF5407 only), whatever fits best. When the selected CPU doesn't support 32-bit branches it will try to change the conditional branch into a \(<\text{B<!cc> *+8}>\) and \(<\text{JMP <label>}>\) sequence.
- \(<\text{BRA <label>}>\) translated to \(<\text{JMP <label>}>\), when \(<\text{label}>\) is not defined in the same section (and option -opt-brajmp is given), or outside the range of -32768 to 32767 bytes from the current address when the selected CPU is not 68020 up, CPU32 or ColdFire ISA_B/C.
- \(<\text{BSR <label>}>\) translated to \(<\text{JSR <label>}>\), when \(<\text{label}>\) is not defined in the same section (and option -opt-brajmp is given), or outside the range of -32768 to 32767 bytes from the current address when the selected CPU is not 68020 up, CPU32 or ColdFire ISA_B/C.
− **<cp>B<cc> <label>** is automatically optimized to 16-bit or 32-bit, whatever fits best. **<cp>** means coprocessor and is P for the PMMU and F for the FPU.

− **CLR.L Dn** optimized to **MOVEQ #0,Dn**.

− **CMP.? #0,<ea>** optimized to **TST.? <ea>**. The selected CPU type must be MC68020 up, ColdFire or CPU32 to support address register direct as effective address (<ea>).

− **DIVS.W/DIVU.W #1,Dn** optimized to **MVZ.W Dn,Dn** for ColdFire ISA B/C (-opt-div).

− **DIVS.W #1,Dn** optimized to the sequence of **NEG.W Dn and MVZ.W Dn,Dn** (-opt-div and -opt-speed).

− **DIVS.L/DIVU.L #1,Dn** optimized to **TST.L Dn** (-opt-div).

− **DIVS.L #1,Dn** optimized to **NEG.L Dn** (-opt-div).

− **DIVU.L #2..256,Dn** optimized to **LSR.L #x,Dn** (-opt-div).

− **EORI.? #1,<ea>** optimized to **NOT.? <ea>**.

− **EORI.? 0,<ea>** optimized to **TST.? <ea>**.

− **FMOVEM.? <reglist>** is deleted, when the register list was empty.

− **FxDIV.? #m,FPn** optimized to **FxMUL.? #1/m,FPn** when m is a power of 2 and option -opt-fconst is given.

− **JMP <label>** optimized to **BRA.? <label>**, when <label> is defined in the same section and in the range of -32768 to 32767 bytes from the current address. Note that **JMP (<lab>,PC)** is never optimized.

− **JSR <label>** optimized to **BSR.? <label>**, when <label> is defined in the same section and in the range of -32768 to 32767 bytes from the current address. Note that **JSR (<lab>,PC)** is never optimized.

− **LEA 0,An** optimized to **SUBA.L An,An**.

− **LEA (0,An),An** and **LEA (An),An** will be deleted.

− **LEA (d,An),An** is optimized to **ADDQ.L #d,An** when d is between 1 and 8 and to **SUBQ.L #d,An** when d is between -1 and -8.

− **LEA (d,Am),An** will be translated into a combination of **MOVEA and ADDA.L** for 68000 and 68010, when d is lower than -32768 or higher than 32767. The **MOVEA** will be omitted when **Am and An** are identical. Otherwise -opt-speed is required.

− **LINK.L An,#x** optimized to **LINK.W An,#x**, when x is between -32768 and 32767.

− **LINK.W An,#x** translated to **LINK.L An,#x**, when x is not between -32768 and 32767 and selected CPU supports this instruction.

− **LSL.? #1,Dn** optimized to **ADD.? Dn,Dn** for 68000 and 68010, when option -opt-lsl is given.

− **LSL.? #2,Dn** optimized into a sequence of two **ADD.? Dn,Dn** for 68000 and 68010, when the operation size is either byte or word and the options -opt-speed and -opt-lsl are given.

− **MOVE.? #0,<ea>** optimized to **CLR.? <ea>**, when allowed by the option -opt-clr or a different CPU than the MC68000 was selected.

− **MOVE.? #x,-(SP)** optimized to **PEA x**, when allowed by the option -opt-pea. The move-size must not be byte (.b).

− **MOVE.B #1,<ea>** optimized to **ST <ea>**, when allowed by the option -opt-st.
Chapter 16: m68k cpu module

− MOVE.L #x,Dn optimized to MOVEQ #x,Dn, when x is between -128 and 127.
− MOVE.L #x,<ea> optimized to MOV3Q #x,<ea>, for ColdFire ISA_B and ISA_C, when x is -1 or between 1 and 7.
− MOVEA.? #0,An optimized to SUBA.L An,An.
− MOVEA.L #x,An optimized to MOVEA.W #x,An, when x is between -32768 and 32767.
− MOVEA.L #label,An optimized to LEA label,An, which could allow further optimization to LEA label(PC),An.
− MOVEM.? <reglist> is deleted, when the register list was empty.
− MOVEM.? <ea>,An optimized to MOVE.? <ea>,An, when the register list only contains a single address register.
− MOVEM.? <ea>,Rn optimized to MOVE.? <ea>,Rn and MOVEM.? Rn,<ea> optimized to MOVE.? Rn,<ea>, when allowed by the option -opt-movem or when just loading an address register.
− MOVEM.? <ea>,Rm/Rn and MOVEM.? Rm/Rn,<ea> are optimized into a sequence of two MOVE for all cpus except 68000 and 68010. Complex addressing modes with displacements or addresses are optimized for 68040 only. Has to be enabled by the options -opt-movem and -opt-speed.
− MULS.?/MULU.? #0,Dn optimized to MOVEQ #0,Dn (-opt-mul).
− MULS.?/MULU.? #1,Dn is deleted (-opt-mul).
− MULS.W #-1,Dn optimized to the sequence EXT.L Dn and NEG.L Dn (-opt-mul and -opt-speed).
− MULS.L #-1,Dn optimized to NEG.L Dn (-opt-mul).
− MULS.W #2..256,Dn optimized to the sequence EXT.L Dn and ASL.L #x,Dn (-opt-mul and -opt-speed).
− MULS.W #-2..-256,Dn optimized to the sequence EXT.L Dn, ASL.L #x,Dn and NEG.L Dn (-opt-mul and -opt-speed).
− MULS.L #2..256,Dn optimized to ASL.L #x,Dn (-opt-mul).
− MULS.L #-2..-256,Dn optimized to the sequence ASL.L #x,Dn and NEG.L Dn (-opt-mul and -opt-speed).
− MULU.W #2..256,Dn optimized to the sequence MVZ.W Dn,Dn and ASL.L #x,Dn for ColdFire ISA_B/C (-opt-mul and -opt-speed).
− MULU.L #2..256,Dn optimized to LSL.L #x,Dn (-opt-mul).
− MVZ.? #x,Dn and MVS.? #x,Dn are optimized to MOVEQ #x,Dn.
− ORI.? #0,<ea> optimized to TST.? <ea>.
− SUB.? #x,<ea> optimized to SUBQ.? #x,<ea>, when x is between 1 and 8.
− SUB.? #x,<ea> optimized to ADDQ.? #x,<ea>, when x is between -1 and -8.
− SUBA.? #x,An optimized to LEA (-x,An),An, when x is between -32767 and 32768.

16.6 Known Problems

Some known problems of this module at the moment:
In some rare cases, mainly by stupid input sources, the optimizer might oscillate forever between two states. When this happens, assembly will be terminated automatically after some time.

16.7 Error Messages

This module has the following error messages:

- 2001: instruction not supported on selected architecture
- 2002: illegal addressing mode
- 2003: invalid register list
- 2004: missing ) in register indirect addressing mode
- 2005: address register required
- 2006: bad size extension
- 2007: displacement at bad position
- 2008: base or index register expected
- 2009: missing ] in memory indirect addressing mode
- 2010: no extension allowed here
- 2011: illegal scale factor
- 2012: can’t scale PC register
- 2013: index register expected
- 2014: too many ] in memory indirect addressing mode
- 2015: missing outer displacement
- 2016: %c expected
- 2017: can’t use PC register as index
- 2018: double registers in list
- 2019: data register required
- 2020: illegal bitfield width/offset
- 2021: constant integer expression required
- 2022: value from -64 to 63 required for k-factor
- 2023: need 32 bits to reference a program label
- 2024: option expected
- 2025: absolute value expected
- 2026: operand value out of range: %ld (valid: %ld..%ld)
- 2027: label in operand required
- 2028: using signed operand as unsigned: %ld (valid: %ld..%ld), %ld to fix
- 2029: branch destination out of range
- 2030: displacement out of range
- 2031: absolute displacement expected
- 2032: unknown option %c%c ignored
- 2033: absolute short address out of range
− 2034: 8-bit branch with zero displacement was converted to 16-bit
− 2035: illegal opcode extension
− 2036: extension for unsized instruction ignored
− 2037: immediate operand out of range
− 2038: immediate operand has illegal type or size
− 2039: data objects with %d bits size are not supported
− 2040: data out of range
− 2041: data has illegal type
− 2042: illegal combination of ColdFire addressing modes
− 2043: FP register required
− 2044: unknown cpu type
− 2045: register expected
− 2046: link.w changed to link.l
− 2047: branch out of range changed to jmp
− 2048: lea-displacement out of range, changed into move/add
− 2049: translated (A%d) into (0,A%d) for movep
− 2050: operand optimized: %s
− 2051: operand translated: %s
− 2052: instruction optimized: %s
− 2053: instruction translated: %s
− 2054: branch optimized into: b<cc>.%c
− 2055: branch translated into: b<cc>.%c
− 2056: basereg A%d already in use
− 2057: basereg A%d is already free
− 2058: short-branch to following instruction turned into a nop
− 2059: not a valid small data register
− 2060: small data mode is not enabled
− 2061: division by zero
− 2062: can’t use B%d register as index
− 2063: register list on both sides
− 2064: "%s" directive was replaced by an instruction with the same name
− 2065: Addr.reg. operand at level #0 causes F-line exception
− 2066: Dr and Dq are identical, transforming DIVxL.L effectively into DIVx.L
17 PowerPC cpu module

This chapter documents the Backend for the PowerPC microprocessor family.

17.1 Legal

This module is written in 2002-2016 by Frank Wille and is covered by the vesm copyright without modifications.

17.2 Additional options for this module

This module provides the following additional options:

- **-big** Select big-endian mode.
- **-little** Select little-endian mode.
- **-many** Allow both, 32- and 64-bit instructions.
- **-mavec,** **-maltivec**
  Generate code for the Altivec unit.
- **-mcom** Allow only common PPC instructions.
- **-m601** Generate code for the PPC 601.
- **-mppc32,** **-mppc,** **-m603,** **-m604**
  Generate code for the 32-bit PowerPC 6xx family.
- **-mppc64,** **-m620**
  Generate code for the 64-bit PowerPC 600 family.
- **-m7400,** **-m7410,** **-m7455**
  Generate code for the 32-bit PowerPC 74xx (G4) family.
- **-m7450** Generate code for the 32-bit PowerPC 7450.
- **-m403,** **-m405**
  Generate code for the IBM/AMCC 32-bit embedded 40x family.
- **-m440,** **-m460**
  Generate code for the AMCC 32-bit embedded 440/460 family.
- **-m821,** **-m850,** **-m860**
  Generate code for the 32-bit MPC8xx PowerQUICC I family.
- **-mbooke** Generate code for the 32-bit Book-E architecture.
- **-me300** Generate code for the 32-bit e300 core (MPC51xx, MPC52xx, MPC83xx).
- **-me500** Generate code for the 32-bit e500 core (MPC85xx), including SPE, EFS and PMR.
- **-mpwr** Generate code for the POWER family.
- **-mpwr2**
  Generate code for the POWER2 family.
- **no-regnames**
  
  Don’t redefine any register-name symbols.

- **opt-branch**
  
  Enables translation of 16-bit branches into "B<!cc> $+8 ; B label" sequences when destination is out of range.

- **sd2reg=<n>**
  
  Sets the 2nd small data base register to Rn.

- **sdreg=<n>**
  
  Sets small data base register to Rn.

The default setting is to generate code for a 32-bit PPC G2, G3, G4 CPU with Altivec support.

### 17.3 General

This backend accepts PowerPC instructions as described in the instruction set manuals from IBM, Motorola, Freescale and AMCC.

The full instruction set of the following families is supported: POWER, POWER2, 40x, 44x, 46x, 60x, 620, 750, 74xx, 860, Book-E, e300 and e500.

The target address type is 32 or 64 bits, depending on the selected CPU model.

Default alignment for sections and instructions is 4 bytes. Data is aligned to its natural alignment by default.

### 17.4 Extensions

This backend provides the following specific extensions:

- When not disabled by the option **-no-regnames**, the registers r0 - r31, f0 - f31, v0 - v31, cr0 - cr7, vrsave, sp, rtoc, fp, fpscr, xer, lr, ctr, and the symbols lt, gt, so and un will be predefined on startup and may be referenced by the program.

This backend extends the selected syntax module by the following directives:

- **.sdreg <n>**
  
  Sets the small data base register to Rn.

- **.sd2reg <n>**
  
  Sets the 2nd small data base register to Rn.

### 17.5 Optimizations

This backend performs the following optimizations:

- 16-bit branches, where the destination is out of range, are translated into B<!cc> $+8 and a 26-bit unconditional branch.

### 17.6 Known Problems

Some known problems of this module at the moment:

- No real differentiation between 403, 750, 860 instructions at the moment.
- There may still be some unsupported PPC models.
17.7 Error Messages

This module has the following error messages:

- 2002: instruction not supported on selected architecture
- 2003: constant integer expression required
- 2004: trailing garbage in operand
- 2005: illegal operand type
- 2006: missing closing parenthesis in load/store addressing mode
- 2007: relocation does not allow hi/lo modifier
- 2008: multiple relocation attributes
- 2009: multiple hi/lo modifiers
- 2010: data size %d not supported
- 2011: data has illegal type
- 2012: relocation attribute not supported by operand
- 2013: operand out of range: %ld (allowed: %ld to %ld)
- 2014: not a valid register (0-31)
- 2015: missing base register in load/store addressing mode
- 2016: missing mandatory operand
- 2017: ignoring fake operand
18 c16x/st10 cpu module

This chapter documents the Backend for the c16x/st10 microcontroller family.
Note that this module is not yet fully completed!

18.1 Legal

This module is written in 2002-2004 by Volker Barthelmann and is covered by the vasam copyright without modifications.

18.2 Additional options for this module

This module provides the following additional options:

-no-translations
  Do not translate between jump instructions. If the offset of a jmp instruction is too large, it will not be translated to jmps but an error will be emitted.
  Also, jmpa will not be optimized to jmp.
  The pseudo-instruction jmp will still be translated.

-jmpa
  A jmp or jmp instruction that is translated due to its offset being larger than 8 bits will be translated to a jmpa rather than a jmps, if possible.

18.3 General

This backend accepts c16x/st10 instructions as described in the Infineon instruction set manuals.
The target address type is 32bit.
Default alignment for sections and instructions is 2 bytes.

18.4 Extensions

This backend provides the following specific extensions:

- There is a pseudo instruction jmp that will be translated either to a jmp or jmpa instruction, depending on the offset.
- The sfr pseudo opcode can be used to declare special function registers. It has two, three or four arguments. The first argument is the identifier to be declared as special function register. The second argument is either the 16bit sfr address or its 8bit base address (0xfe for normal sfrs and 0xf0 for extended special function registers). In the latter case, the third argument is the 8bit sfr number. If another argument is given, it specifies the bit-number in the sfr (i.e. the declaration declares a single bit).
  Example:
  .sfr zeros,0xfe,0x8e
- SEG and SOF can be used to obtain the segment or segment offset of a full address.
  Example:
  mov r3,#SEG farfunc
18.5 Optimizations

This backend performs the following optimizations:
- jmp is translated to jmpw, if possible. Also, if -no-translations was not specified, jmp and jmpa are translated.
- Relative jump instructions with an offset that does not fit into 8 bits are translated to a jmps instruction or an inverted jump around a jmps instruction.
- For instruction that have two forms gpr,#IMM3/4 and reg,#IMM16 the smaller form is used, if possible.

18.6 Known Problems

Some known problems of this module at the moment:
- Lots...

18.7 Error Messages

This module has the following error messages:
- 2001: illegal operand
- 2002: word register expected
- 2004: value does not find in %d bits
- 2005: data size not supported
- 2006: illegal use of SOF
- 2007: illegal use of SEG
- 2008: illegal use of DPP prefix
19 6502 cpu module

This chapter documents the backend for the MOS/Rockwell 6502 microprocessor family. It also supports the Rockwell/WDC 65C02 and the Hudson Soft HuC6280 instruction sets.

19.1 Legal

This module is written in 2002,2006,2008-2012,2014-2018 by Frank Wille and is covered by the vasm copyright without modifications.

19.2 Additional options for this module

This module provides the following additional options:

- **-6280** Recognize all HuC6280 instructions.
- **-c02** Recognize all 65C02 instructions. This excludes DTV (**-dtv**) and illegal (**-illegal**) instructions.
- **-dtv** Recognize the three additional C64-DTV instructions.
- **-illegal** Allow 'illegal' 6502 instructions to be recognized.
- **-opt-branch** Enables 'optimization' of B<cc> branches into "B<!cc> *+3 ; JMP label" sequences when necessary.
- **-wdc02** Recognize all 65C02 instructions and the WDC65C02 extensions (RMB,SMB,BBR,BBS,STP,WAI).

19.3 General

This backend accepts 6502 family instructions as described in the instruction set reference manuals from MOS and Rockwell, which are valid for the following CPUs: 6502 - 6518, 6570, 6571, 6702, 7501, 8500, 8502.

Optionally accepts 65C02 family instructions as described in the instruction set reference manuals from Rockwell and WDC. Also supports the WDC extensions in the W65C02 and W65C134.

Optionally accepts 65CE02 family instructions as described in the instruction set reference manuals from Commodore Semiconductor Group.

Optionally accepts HuC6280 instructions as described in the instruction set reference manuals from Hudson Soft.

The target address type is 16 bit.

Instructions consist of one up to three bytes for the standard 6502 family (up to 7 bytes for the 6280) and require no alignment. There is also no alignment requirement for sections and data.

All known mnemonics for illegal instructions are optionally recognized (e.g. **dcm** and **dcp** refer to the same instruction). Some illegal instructions (e.g. **$ab**) are known to show unpredictable behaviour, or do not always work the same on different CPUs.
19.4 Extensions

This backend provides the following specific extensions:

− The parser understands a lo/hi-modifier to select low- or high-byte of a 16-bit word. The character < is used to select the low-byte and > for the high-byte. It has to be the first character before an expression.

− When applying the operation /256, %256 or &256 on a label, an appropriate lo/hi-byte relocation will automatically be generated.

This backend extends the selected syntax module by the following directives:

\texttt{setdp <expr>}

Set the current base address of the zero/direct page for optimizations from absolute to zero-page addressing modes. Example: set it to $2000$ for the HuC6280/PC-Engine.

19.5 Optimizations

This backend performs the following operand optimizations:

− Branches, where the destination is out of range, are translated into \texttt{B<!cc> ++3} and an absolute \texttt{JMP} instruction.

19.6 Known Problems

Some known problems of this module at the moment:

− None.

19.7 Error Messages

This module has the following error messages:

− 2001: instruction not supported on selected architecture

− 2002: trailing garbage in operand

− 2003: missing closing parenthesis in addressing mode

− 2004: data size %d not supported

− 2005: relocation does not allow hi/lo modifier

− 2006: operand doesn’t fit into 8-bits

− 2007: branch destination out of range

− 2008: illegal bit number
Chapter 20: ARM cpu module

This chapter documents the backend for the Advanced RISC Machine (ARM) microprocessor family.

20.1 Legal

This module is written in 2004,2006,2010-2015 by Frank Wille and is covered by the vasm copyright without modifications.

20.2 Additional options for this module

This module provides the following additional options:

- `-a2` Generate code compatible with ARM V2 architecture.
- `-a3` Generate code compatible with ARM V3 architecture.
- `-a3m` Generate code compatible with ARM V3m architecture.
- `-a4` Generate code compatible with ARM V4 architecture.
- `-a4t` Generate code compatible with ARM V4t architecture.
- `-big` Output big-endian code and data.
- `-little` Output little-endian code and data (default).
- `-m2` Generate code for the ARM2 CPU.
- `-m250` Generate code for the ARM250 CPU.
- `-m3` Generate code for the ARM3 CPU.
- `-m6` Generate code for the ARM6 CPU.
- `-m600` Generate code for the ARM600 CPU.
- `-m610` Generate code for the ARM610 CPU.
- `-m7` Generate code for the ARM7 CPU.
- `-m710` Generate code for the ARM710 CPU.
- `-m7500` Generate code for the ARM7500 CPU.
- `-m7d` Generate code for the ARM7d CPU.
- `-m7di` Generate code for the ARM7di CPU.
- `-m7dm` Generate code for the ARM7dm CPU.
- `-m7dmi` Generate code for the ARM7dmi CPU.
- `-m7tdmi` Generate code for the ARM7tdmi CPU.
- `-m8` Generate code for the ARM8 CPU.
- `-m810` Generate code for the ARM810 CPU.
- `-m9` Generate code for the ARM9 CPU.
-m9          Generate code for the ARM9 CPU.
-m920        Generate code for the ARM920 CPU.
-m920t       Generate code for the ARM920t CPU.
-m9tdmi      Generate code for the ARM9tdmi CPU.
-msa1        Generate code for the SA1 CPU.
-mstrongarm  Generate code for the STRONGARM CPU.
-mstrongarm110 Generate code for the STRONGARM110 CPU.
-mstrongarm1100 Generate code for the STRONGARM1100 CPU.
-opt-adr     The ADR directive will be automatically converted into ADRL if required (which
inserts an additional ADD/SUB to calculate an address).
-opt-ldrpc   The maximum range in which PC-relative symbols can be accessed through
LDR and STR is extended from +/-4KB to +/-1MB (or +/-256 Bytes to +/-65536
Bytes when accessing half-words). This is done by automatically inserting an
additional ADD or SUB instruction before the LDR/STR.
-thumb       Start assembling in Thumb mode.

20.3 General
This backend accepts ARM instructions as described in various ARM CPU data sheets. Ad-
ditionally some architectures support a second, more dense, instruction set, called THUMB.
There are special directives to switch between those two instruction sets.
The target address type is 32bit.
Default alignment for instructions is 4 bytes for ARM and 2 bytes for THUMB. Sections
will be aligned to 4 bytes by default. Data is aligned to its natural alignment by default.

20.4 Extensions
This backend extends the selected syntax module by the following directives:
.arm         Generate 32-bit ARM code.
.thumb       Generate 16-bit THUMB code.

20.5 Optimizations
This backend performs the following optimizations and translations for the ARM instruction
set:
- LDR/STR Rd,symbol, with a distance between symbol and PC larger than 4KB, is
translated to ADD/SUB Rd,PC,#offset&0xff000 + LDR/STR Rd,[Rd,#offset&0xffff],
when allowed by the option -opt-ldrpc.
- ADR Rd,symbol is translated to ADD/SUB Rd,PC,#rotated_offset8.
- ADRL Rd,symbol is translated to ADD/SUB Rd,PC,#hi_rotated8 + ADD/SUB Rd,Rd,#lo_rotated8. ADR will be automatically treated as ADRL when required and when allowed by the option -opt-adr.
- The immediate operand of ALU-instructions will be translated into the appropriate 8-bit-rotated value. When rotation alone doesn't succeed the backed will try it with inverted and negated values (inverting/negating the ALU-instruction too). Optionally you may specify the rotate constant yourself, as an additional operand.

For the THUMB instruction set the following optimizations and translations are done:
- A conditional branch with a branch-destination being out of range is translated into B<!cc> .+4 + B label.
- The BL instruction is translated into two sub-instructions combining the high- and low 22 bit of the branch displacement.

20.6 Known Problems
Some known problems of this module at the moment:
- Only instruction sets up to ARM architecture V4t are supported.

20.7 Error Messages
This module has the following error messages:
- 2001: instruction not supported on selected architecture
- 2002: trailing garbage in operand
- 2003: label from current section required
- 2004: branch offset (%ld) is out of range
- 2005: PC-relative load/store (offset %ld) out of range
- 2006: cannot make rotated immediate from PC-relative offset (0x%lx)
- 2007: constant integer expression required
- 2008: constant (0x%lx) not suitable for 8-bit rotated immediate
- 2009: branch to an unaligned address (offset %ld)
- 2010: not a valid ARM register
- 2011: PC (r15) not allowed in this mode
- 2012: PC (r15) not allowed for offset register Rm
- 2013: PC (r15) not allowed with write-back
- 2014: register r%ld was used multiple times
- 2015: illegal immediate shift count (%ld)
- 2016: not a valid shift register
- 2017: 24-bit unsigned immediate expected
- 2018: data size %d not supported
- 2019: illegal addressing mode: %s
- 2020: signed/halfword ldr/str doesn't support shifts
- 2021: %d-bit immediate offset out of range (%ld)
- 2022: post-indexed addressing mode expected
- 2023: operation not allowed on external symbols
- 2024: ldc/stc offset has to be a multiple of 4
- 2025: illegal coprocessor operation mode or type: %ld
- 2026: %d-bit unsigned immediate offset out of range (%ld)
- 2027: offset has to be a multiple of %d
- 2028: instruction at unaligned address
- 2029: TSTP/TEQP/CMNP/CMPP deprecated on 32-bit architectures
- 2030: rotate constant must be an even number between 0 and 30: %ld
- 2031: %d-bit unsigned constant required: %ld
Chapter 21: 80x86 cpu module

21 80x86 cpu module

This chapter documents the Backend for the 80x86 microprocessor family.

21.1 Legal

This module is written in 2005-2006,2011,2015-2016 by Frank Wille and is covered by the vasm copyright without modifications.

21.2 Additional options for this module

This module provides the following additional options:

- `cpudebug=<n>`
  Enables debugging output.

- `m8086` Generate code for the 8086 CPU.

- `mi186` Generate code for the 80186 CPU.

- `mi286` Generate code for the 80286 CPU.

- `mi386` Generate code for the 80386 CPU.

- `mi486` Generate code for the 80486 CPU.

- `mi586` Generate code for the Pentium.

- `mi686` Generate code for the PentiumPro.

- `mpentium` Generate code for the Pentium.

- `mpentiumpro` Generate code for the PentiumPro.

- `mk6` Generate code for the AMD K6.

- `mathlon` Generate code for the AMD Athlon.

- `msledgehammer` Generate code for the Sledgehammer CPU.

- `m64` Generate code for 64-bit architectures (x86_64).

21.3 General

This backend accepts 80x86 instructions as described in the Intel Architecture Software Developer's Manual.

The target address type is 32 bits. It is 64 bits when the x86_64 architecture was selected (`-m64`).

Instructions do not need any alignment. Data is aligned to its natural alignment by default.

The backend uses MIT-syntax! This means the left operands are always the source and the right operand is the destination. Register names have to be prefixed by a '%'.

The operation size is indicated by a 'b', 'w', 'l', etc. suffix directly appended to the mnemonic. The assembler can also determine the operation size from the size of the registers being used.
21.4 Extensions

Predefined register symbols in this backend:

- 8-bit registers:  \texttt{al} cl \texttt{dl} bl ah ch dh bh axl cxl dxl spl bpl sil dil r8b r9b r10b r11b r12b r13b r14b r15b
- 16-bit registers: ax cx dx bx sp bp si di r8w r9w r10w r11w r12w r13w r14w r15w
- 32-bit registers: eax ecx edx ebx esp ebp esi edi r8d r9d r10d r11d r12d r13d r14d r15d
- 64-bit registers: rax rcx rdx rbx rsp rbp rsi rdi r8 r9 r10 r11 r12 r13 r14 r15
- Segment registers: es cs ss ds fs gs
- Control registers: cr0 cr1 cr2 cr3 cr4 cr5 cr6 cr7 cr8 cr9 cr10 cr11 cr12 cr13 cr14 cr15
- Debug registers: dr0 dr1 dr2 dr3 dr4 dr5 dr6 dr7 dr8 dr9 dr10 dr11 dr12 dr13 dr14 dr15
- Test registers: tr0 tr1 tr2 tr3 tr4 tr5 tr6 tr7
- MMX and SIMD registers: mm0 mm1 mm2 mm3 mm4 mm5 mm6 mm7 xmm0 xmm1 xmm2 xmm3 xmm4 xmm5 xmm6 xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
- FPU registers: st st(0) st(1) st(2) st(3) st(4) st(5) st(6) st(7)

This backend extends the selected syntax module by the following directives:

\texttt{.code16} Sets the assembler to 16-bit addressing mode.
\texttt{.code32} Sets the assembler to 32-bit addressing mode, which is the default.
\texttt{.code64} Sets the assembler to 64-bit addressing mode.

21.5 Optimizations

This backend performs the following optimizations:

- Immediate operands are optimized to the smallest size which can still represent the absolute value.
- Displacement operands are optimized to the smallest size which can still represent the absolute value.
- Jump instructions are optimized to 8-bit displacements, when possible.

21.6 Known Problems

Some known problems of this module at the moment:

- 64-bit operations are incomplete and experimental.

21.7 Error Messages

This module has the following error messages:

- 2001: instruction not supported on selected architecture
- 2002: trailing garbage in operand
- 2003: same type of prefix used twice
– 2004: immediate operand illegal with absolute jump
– 2005: base register expected
– 2006: scale factor without index register
– 2007: missing ’)’ in baseindex addressing mode
– 2008: redundant %s prefix ignored
– 2009: unknown register specified
– 2010: using register %%%s instead of %%%s due to ’%c’ suffix
– 2011: %%%s not allowed with ’%c’ suffix
– 2012: illegal suffix ’%c’
– 2013: instruction has no suffix and no register operands - size is unknown
– 2015: memory operand expected
– 2016: you cannot pop %%%s
– 2017: translating to %s %%%s,%%%s
– 2018: translating to %s %%%s
– 2019: absolute scale factor required
– 2020: illegal scale factor (valid: 1,2,4,8)
– 2021: data objects with %d bits size are not supported
– 2022: need at least %d bits for a relocatable symbol
– 2023: pc-relative jump destination out of range (%lld)
– 2024: instruction doesn’t support these operand sizes
– 2025: cannot determine immediate operand size without a suffix
– 2026: displacement doesn’t fit into %d bits
22 z80 cpu module

This chapter documents the backend for the 8080/z80/gbz80/64180/RCMx000 microprocessor family.

22.1 Legal

This module is copyright in 2009 by Dominic Morris.

* Redistribution and use in source and binary forms, with or without
  * modification, are permitted provided that the following conditions
  * are met:
  * 1. Redistributions of source code must retain the above copyright
  *     notice, this list of conditions and the following disclaimer.
  * 2. Redistributions in binary form must reproduce the above copyright
  *     notice, this list of conditions and the following disclaimer in the
  *     documentation and/or other materials provided with the distribution.

* THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR
* IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
* IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
* INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
* DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
* THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
* THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

22.2 Additional options for this module

This module provides the following additional options:

- **-8080**
  Turns on 8080 compatibility mode. Any use of z80 (or higher) opcodes will result in an error being generated.

- **-gbz80**
  Turns on gbz80 compatibility mode. Any use of non-supported opcodes will result in an error being generated.

- **-hd64180**
  Turns on 64180 mode supporting additional 64180 opcodes.

- **-rcm2000**

- **-rcm3000**

- **-rcm4000**
  Turns on Rabbit compatibility mode, generating the correct codes for moved opcodes and supporting the additional Rabbit instructions. In this mode, 8 bit access to the 16 bit index registers is not permitted.

- **-rcmemu**
  Turns on emulation of some instructions which aren’t available on the Rabbit processors.
- `swapixiy`  
  Swaps the usage of ix and iy registers. This is useful for compiling generic code that uses an index register that is reserved on the target machine.

- `z80asm`  
  Switches on z80asm mode. This translates ASMPC to $ and accepts some pseudo opcodes that z80asm supports. Most emulation of z80asm directives is provided by the oldsyntax syntax module.

### 22.3 General

This backend accepts z80 family instructions in standard Zilog syntax. Rabbit opcodes are accepted as defined in the publically available reference material from Rabbit Semiconductor, with the exception that the `ljp` and `lcall` opcodes need to be supplied with a 24 bit number rather than an 8 bit `xpc` and a 16 bit address.

The target address type is 16 bit.

Instructions consist of one up to six bytes and require no alignment. There is also no alignment requirement for sections and data.

### 22.4 Extensions

This backend provides the following specific extensions:

- Certain Rabbit opcodes can be prefixed by the `altd` and/or the `ioi/ioe` modifier. For details of which instructions these are valid for please see the documentation from Rabbit.

- The parser understands a lo/hi-modifier to select low- or high-byte of a 16-bit word. The character `<` is used to select the low-byte and `>` for the high-byte. It has to be the first character before an expression.

- When applying the operation `/256`, `%256` or `&256` on a label, an appropriate lo/hi-byte relocation will automatically be generated.

### 22.5 Optimisations

This backend supports the emulation of certain z80 instructions on the Rabbit/gbz80 processor. These instructions are `rld`, `rrd`, `cpi`, `cpir`, `cpd` and `cpdr`. The link stage should provide routines with the opcode name prefixed with `rcmx_` (eg `rcmx_rld`) which implements the same functionality. Example implementations are available within the z88dk CVS tree.

Additionally, for the Rabbit targets the missing call `cc`, opcodes will be emulated.

### 22.6 Known Problems

Some known problems of this module at the moment:

- Not all RCM4000 opcodes are supported (`lcall`, `ljp` are not available).
22.7 Error Messages

This module has the following error messages:

- 2001: index offset out of bounds (%d)
- 2002: Opcode not supported by %s (%s)
- 2003: Index registers not available on 8080
- 2004: out of range for 8 bit expression (%d)
- 2005: invalid bit number (%d) should be in range 0..7
- 2006: rst value out of range (%d/0x%02x)
- 2007: %s value out of range (%d)
- 2008: index offset should be a constant
- 2009: invalid branch type for jr
- 2010: Rabbit target doesn’t support rst %d
- 2011: Rabbit target doesn’t support 8 bit index registers
- 2012: z180 target doesn’t support 8 bit index registers
- 2013: invalid branch type for jre
- 2014: Opcode not supported by %s (%s) but it can be emulated (-rcmenu)
- 2015: %s specifier is only valid for Rabbit processors
- 2016: Only one of ioi and ioe can be specified at a time
- 2017: %s specifier is not valid for the opcode %s
- 2018: %s specifier redundant for the opcode %s
- 2019: %s specifier has no effect on the opcode %s
- 2020: Operand value must evaluate to a constant for opcode %s
- 2021: Unhandled operand type wanted 0x%x got 0x%x
- 2022: Missed matched index registers on %s
- 2023: Only out (c),0 is supported for the opcode %s
- 2024: Operations between different index registers are forbidden
- 2025: Operations between ix/iy/hl are forbidden
- 2026: Double indirection forbidden
Chapter 23: 6800 cpu module

23 6800 cpu module

This chapter documents the backend for the Motorola 6800 microprocessor family.

23.1 Legal

This module is written in 2013-2016 by Esben Norby and Frank Wille and is covered by the vasm copyright without modifications.

23.2 Additional options for this module

This module provides the following additional options:

- -m6800 Generate code for the 6800 CPU (default setting).
- -m6801 Generate code for the 6801 CPU.
- -m68hc11 Generate code for the 68HC11 CPU.

23.3 General

This backend accepts 6800 family instructions for the following CPUs:

- 6800 code generation: 6800, 6802, 6808.
- 6801 code generation: 6801, 6803.
- 68HC11.

The 6804, 6805, 68HC08 and 6809 are not supported, they use a similar instruction set, but are not opcode compatible.

The target address type is 16 bit.

Instructions consist of one up to five bytes and require no alignment. There is also no alignment requirement for sections and data.

23.4 Extensions

This backend provides the following specific extensions:

- When an instruction supports direct and extended addressing mode the < character can be used to force direct mode and the > character forces extended mode. Otherwise the assembler selects the best mode automatically, which defaults to extended mode for external symbols.
- When applying the operation /256, %256 or &256 on a label, an appropriate lo/hi-byte relocation will automatically be generated.

23.5 Optimizations

None.

23.6 Known Problems

Some known problems of this module at the moment:

- None?
23.7 Error Messages

This module has the following error messages:

− 2001: data size %d not supported
− 2002: operand doesn’t fit into 8-bits
− 2003: branch destination out of range
Chapter 24: Jaguar RISC cpu module

This chapter documents the backend for the Atari Jaguar GPU/DSP RISC processor.

24.1 Legal
This module is written in 2014-2015 by Frank Wille and is covered by the vasm copyright without modifications.

24.2 Additional options for this module
This module provides the following additional options:
- **-big** Output big-endian code and data (default).
- **-little** Output little-endian code and data.
- **-many** Generate code for GPU or DSP RISC. All instructions are accepted (default).
- **-mdsp**
- **-mjerry** Generate code for the DSP RISC (part of Jerry).
- **-mgpu**
- **-mtom** Generate code for the GPU RISC (part of Tom).

24.3 General
This backend accepts RISC instructions for the GPU or DSP in Atari’s Jaguar custom chip set according to the "Jaguar Technical Reference Manual for Tom & Jerry", Revision 8. Documentation bugs were fixed by using various sources on the net.
The target address type is 32 bits.
Default alignment for instructions is 2 bytes. Data is aligned to its natural alignment by default.

24.4 Optimizations
This backend performs the following optimizations and translations for the GPU/DSP RISC instruction set:
- load (Rn+0),Rm is optimized to load (Rn),Rm.
- store Rn,(Rm+0) is optimized to store Rn,(Rm).

24.5 Extensions
This backend extends the selected syntax module by the following directives (note that a leading dot is optional):

```
<symbol> ccdef <expression>
```

Allows defining a symbol for the condition codes used in jump and jr instructions. Must be constant number in the range of 0 to 31 or another condition code symbol.
ccundef <symbol>
   Undefine a condition code symbol previously defined via ccdef.

dsp
   Select DSP instruction set.

<symbol> equr <Rn>
   Define a new symbol named <symbol> and assign the address register Rn to it.
   <Rn> may also be another register symbol. Note that a register symbol must be defined before it can be used.

equrundef <symbol>
   Undefine a register symbol previously defined via equr.

gpu
   Select GPU instruction set.

<symbol> regequ <Rn>
   Equivalent to equr.

regundef <symbol>
   Undefine a register symbol previously defined via regequ.

All directives may be optionally preceded by a dot (.), for compatibility with various syntax modules.

24.6 Known Problems

Some known problems of this module at the moment:
   – Encoding of MOVEI instruction in little-endian mode is unkown.
   – The developer has to provide the necessary NOP instructions after jumps, or OR instructions to work around hardware bugs, her/himself.

24.7 Error Messages

This module has the following error messages:
   – 2001: data size %d not supported
   – 2002: value from %ld to %ld required
25 Trillek TR3200 cpu module

This chapter documents the Backend for the TR3200 cpu.

25.1 Legal
This module is written in 2014 by Luis Panadero Guadén and is covered by the vasm copyright without modifications.

25.2 General
This backend accepts TR3200 instructions as described in the TR3200 specification (https://github.com/trillek-team/trillek-computer)
The target address type is 32 bits.
Default alignment for sections is 4 bytes. Instructions alignment is 4 bytes. Data is aligned to its natural alignment by default, i.e. 2 byte wide data alignment is 2 bytes and 4 byte wide data alignment is 4 byte.
The backend uses TR3200 syntax! This means the left operands are always the destination and the right operand is the source (except for single operand instructions). Register names have to be prefixed by a '%' (%bp, %r0, etc.) This means that it should accept WaveAsm assembly files if oldstyle syntax module is being used. The instructions are lowercase, -dotdir option is being used and directives are not in the first column.

25.3 Extensions
Predefined register symbols in this backend:
- register by number: r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
- special registers by name: bp sp y ia flags

25.4 Known Problems
Some known problems of this module at the moment:
- This module need to be fully checked, but has been tested building a full program that could be found here: https://github.com/Zardoz89/trillek-firmware
- Instruction relocations are missing.

25.5 Error Messages
This module has the following error messages:
- 2001: illegal operand
- 2002: illegal qualifier <%%s>
- 2003: data size not supported
25.6 Example

It follows a little example to illustrate TR3200 assembly using the oldstyle syntax module (option -dotdir required):

```assembly
const .equ 0xBEBACAFE ; A constant
an_addr .equ 0x100 ; Other constant

; ROM code
.org 0x100000
.text
_start ; Label with or without an ending ";"
    mov %sp, 0x1000 ; Set the initial stack
    mov %r0, 0
    mov %r1, 0xA5
    mov %r2, 0
    storeb %r0, an_addr, %r1
    add %r0, %r2, %bp
    add %r0, %r2, 0
    add %r0, %r0, 10
    add %r0, %r0, 10h
    add %r0, %r0, 0x100010
    add %r0, %r0, (256 + 100) ; vasm parses math expressions
    mov %r2, 0
    mov %r3, 0x10200
    loadb %r6, 0x100200
    loadb %r1, %r2, 0x100200
    loadb %r1, %r2, %r3
    loadb %r4, var1
    push %r0
    .repeat 2 ; directives to repeat stuff!
    push const
    .endrepeat
    .repeat 2
    pop %r5
    .endrepeat
    pop %r0
    rcall foo ; Relative call/jump!
    sleep

foo: ; Subroutine
```
ifneq %r5, 0
    mul %r5, %r5, 2
    sub %r5, %r5, 1
ret

; ROM data
.org 0x100500

var1 .db 0x20 ; A byte size variable
    .even ; Enforce to align to even address
var3 .dw 0x1020 ; A word size variable
var4 .dd 0xA0B0C20 ; A double word size variable
str1 .asciiz "Hello world!" ; ASCII string with null termination
str2 .string "Hello world!" ; ASCII string with null termination
    .fill 5, 0xFF ; Fill 5 bytes with 0xFF
    .reserve 5 ; Reserves space for 5 byte
26 Interface

26.1 Introduction
This chapter is under construction!

This chapter describes some of the internals of vasm and tries to explain what has to be done to write a cpu module, a syntax module or an output module for vasm. However if someone wants to write one, I suggest to contact me first, so that it can be integrated into the source tree.

Note that this documentation may mention explicit values when introducing symbolic constants. This is due to copying and pasting from the source code. These values may not be up to date and in some cases can be overridden. Therefore do never use the absolute values but rather the symbolic representations.

26.2 Building vasm
This section deals with the steps necessary to build the typical vasm executable from the sources.

26.2.1 Directory Structure
The vasm-directory contains the following important files and directories:

`vasm/` The main directory containing the assembler sources.

`vasm/Makefile` The Makefile used to build vasm.

`vasm/syntax/<syntax-module>/` Directories for the syntax modules.

`vasm/cpus/<cpu-module>/` Directories for the cpu modules.

`vasm/obj/` Directory the object modules will be stored in.

All compiling is done from the main directory and the executables will be placed there as well. The main assembler for a combination of `<cpu>` and `<syntax>` will be called `vasm<cpu>_<syntax>`. All output modules are usually integrated in every executable and can be selected at runtime.

26.2.2 Adapting the Makefile
Before building anything you have to insert correct values for your compiler and operating system in the Makefile.

TARGET Here you may define an extension which is appended to the executable’s name. Useful, if you build various targets in the same directory.

TARGETEXTENSION Defines the file name extension for executable files. Not needed for most operating systems. For Windows it would be .exe.
CC Here you have to insert a command that invokes an ANSI C compiler you want to use to build vasm. It must support the -I option the same like e.g. vc or gcc.

COPTS Here you will usually define an option like -c to instruct the compiler to generate an object file. Additional options, like the optimization level, should also be inserted here as well. When the host operating system is different from a Unix (MacOSX and MiNT are Unix), you have to define one of the following preprocessor macros:

-DAMIGA AmigaOS (M68k or PPC), MorphOS, AROS.
-DATARI Atari TOS.
-DMSDOS CP/M, MS-DOS, Windows.

CCOUT Here you define the option which is used to specify the name of an output file, which is usually -o.

LD Here you insert a command which starts the linker. This may be the the same as under CC.

LDFLAGS Here you have to add options which are necessary for linking. E.g. some compilers need special libraries for floating-point.

LDOUT Here you define the option which is used by the linker to specify the output file name.

RM Specify a command to delete a file, e.g. rm -f.

An example for the Amiga using vbcc would be:

```
TARGET = _os3
TARGETEXTENSION =
CC = vc +aos68k
CCOUT = -o
COPTS = -c -c99 -cpu=68020 -DAMIGA -O1
LD = $(CC)
LDOUT = $(CCOUT)
LDFLAGS = -lmieee
RM = delete force quiet
```

An example for a typical Unix-installation would be:

```
TARGET =
TARGETEXTENSION =
CC = gcc
CCOUT = -o
COPTS = -c -O2
LD = $(CC)
LDOUT = $(CCOUT)
LDFLAGS = -lm
RM = rm -f
```

Open/Net/Free/Any BSD i386 systems will probably require the following an additional -D_ANSI_SOURCE in COPTS.
26.2.3 Building vasm

Note to users of Open/Free/Any BSD i386 systems: You will probably have to use GNU make instead of BSD make, i.e. in the following examples replace "make" with "gmake".

Type:

make CPU=<cpu> SYNTAX=<syntax>

For example:

make CPU=ppc SYNTAX=std

The following CPU modules can be selected:

- CPU=6502
- CPU=6800
- CPU=arm
- CPU=c16x
- CPU=jagrisc
- CPU=m68k
- CPU=ppc
- CPU=test
- CPU=tr3200
- CPU=vidcore
- CPU=x86
- CPU=z80

The following syntax modules can be selected:

- SYNTAX=std
- SYNTAX=mot
- SYNTAX=madmac
- SYNTAX=oldstyle
- SYNTAX=test

For Windows and various Amiga targets there are already Makefiles included, which you may either copy on top of the default Makefile, or call it explicitly with make’s -f option:

make -f Makefile.OS4 CPU=ppc SYNTAX=std

26.3 General data structures

This section describes the fundamental data structures used in vasm which are usually necessary to understand for writing any kind of module (cpu, syntax or output). More detailed information is given in the respective sections on writing specific modules where necessary.
26.3.1 Source

A source structure represents a source text module, which can be either the main source text, an included file or a macro. There is always a link to the parent source from where the current source context was included or called.

```c
struct source *parent;
    Pointer to the parent source context. Assembly continues there when the current source context ends.

int parent_line;
    Line number in the parent source context, from where we were called. This information is needed, because line numbers are only reliable during parsing and later from the atoms. But an include directive doesn’t create an atom.

struct source_file *srcfile;
    The source_file structure has the unique file name, index and text-pointer for this source text instance. Used for debugging output, like DWARF.

char *name;
    File name of the main source or include file, or macro name.

char *text;
    Pointer to the source text start.

size_t size;
    Size of the source text to assemble in bytes.

struct source *defsrc;
    This is a NULL-pointer for real source text files. Otherwise it is a reference to the source which defines the current macro or repetition.

int defline;
    Valid when defsrc is not NULL. Contains the starting line number of a macro or repetition in a source text file.

macro *macro;
    Pointer to macro structure, when currently inside a macro (see also num_params).

unsigned long repeat;
    Number of repetitions of this source text. Usually this is 1, but for text blocks between a rept and endr directive it allows any number of repetitions, which is decremented everytime the end of this source text block is reached.

char *irpname;
    Name of the iterator symbol in special repeat loops which use a sequence of arbitrary values, being assigned to this symbol within the loop. Example: irp directive in std-syntax.

struct macarg *irpvals;
    A list of arbitrary values to iterate over in a loop. With each iteration the frontmost value is removed from the list until it is empty.
int cond_level;
Current level of conditional nesting while entering this source text. It is automatically restored to the previous level when leaving the source prematurely through end_source().

struct macarg *argnames;
The current list of named macro arguments.

int num_params;
Number of macro parameters passed at the invocation point from the parent source. For normal source files this entry will be -1. For macros 0 (no parameters) or higher.

char *param[MAXMACPARAMS];
Pointer to the macro parameters.

int param_len[MAXMACPARAMS];
Number of characters per macro parameter.

int num_quals;
(If MAX_QUALIFIERS!=0.) Number of qualifiers for a macro. when not passed on invocation these are the default qualifiers.

char *qual[MAX_QUALIFIERS];
(If MAX_QUALIFIERS!=0.) Pointer to macro qualifiers.

int qual_len[MAX_QUALIFIERS];
(If MAX_QUALIFIERS!=0.) Number of characters per macro qualifier.

unsigned long id;
Every source has its unique id. Useful for macros supporting the special \@ parameter.

char *srcptr;
The current source text pointer, pointing to the beginning of the next line to assemble.

int line;
Line number in the current source context. After parsing the line number of the current atom is stored here.

size_t bufsize;
Current size of the line buffer (linebuf). The size of the line buffer is extended automatically, when an overflow happens.

char *linebuf;
A buffer for the current line being assembled in this source text. A child-source, like a macro, can refer to arguments from this buffer, so every source has got its own. When returning to the parent source, the linebuf is deallocated to save memory.

expr *cargexp;
(If CARGSYM was defined.) Pointer to the current expression assigned to the CARG-symbol (used to select a macro argument) in this source instance. So it can be restored when reentering this instance.
long retn;
    (If REPTNSYM was defined.) Current value of the repetition counter symbol in
this source instance. So it can be restored when reentering this instance.

26.3.2 Sections
One of the top level structures is a linked list of sections describing continuous blocks of
memory. A section is specified by an object of type section with the following members
that can be accessed by the modules:

    struct section *next;
        A pointer to the next section in the list.

    char *name;
        The name of the section.

    char *attr;
        A string describing the section flags in ELF notation (see, for example, docu-
mentation o the .section directive of the standard syntax mopdule.

    atom *first;
    atom *last;
        Pointers to the first and last atom of the section. See following sections for
information on atoms.

    taddr align;
        Alignment of the section in bytes.

    uint32_t flags;
        Flags of the section. Currently available flags are:

        HAS_SYMBOLS
            At least one symbol is defined in this section.

        RESOLVE_WARN
            The current atom changed its size multiple times, so atom_size() is
now called with this flag set in its section to make the back-
end (e.g. instruction_size()) aware of it and do less aggressive
optimizations.

        UNALLOCATED
            Section is unallocated, which means it doesn’t use any memory
space in the output file. Such a section will be removed before
creating the output file and all its labels converted into absolute
expression symbols. Used for "offset" sections. Refer to switch_
offset_section().

        LABELS_ARE_LOCAL
            As long as this flag is set new labels in a section are defined as local
labels, with the section name as global parent label.

        ABSOLUTE
            Section is loaded at an absolute address in memory.

        PREVABS
            Remembers state of the ABSOLUTE flag before entering relocated-org
mode (IN_RORG). So it can be restored later.
Section has entered relocated-org mode, which also sets the ABSOLUTE flag. In this mode code is written into the current section, but relocated to an absolute address. No relocation information are generated.

Section is marked as suitable for cpu-specific "near" addressing modes. For example, base-register relative. The cpu backend can use this information as an optimization hint when referencing symbols from this section.

taddr org;
Start address of a section. Usually zero.

taddr pc;
Current address in this section. Can be used while traversing through the section. Has to be updated by a module using it. Is set to org at the beginning.

unsigned long idx;
A member usable by the output module for private purposes.

Symbols are represented by a linked list of type symbol with the following members that can be accessed by the modules:

int type;  Type of the symbol. Available are:
#define LABSYM 1
The symbol is a label defined at a specific location.

#define IMPORT 2
The symbol is imported from another file.

#define EXPRESSION 3
The symbol is defined using an expression.

uint32_t flags;
Flags of this symbol. Available are:
#define TYPE_UNKNOWN 0
The symbol has no type information.

#define TYPE_OBJECT 1
The symbol defines an object.

#define TYPE_FUNCTION 2
The symbol defines a function.

#define TYPE_SECTION 3
The symbol defines a section.

#define TYPE_FILE 4
The symbol defines a file.

#define EXPORT (1<<3)
The symbol is exported to other files.


```c
#define INEVAL (1<<4)
   Used internally.

#define COMMON (1<<5)
   The symbol is a common symbol.

#define WEAK (1<<6)
   The symbol is weak, which means the linker may overwrite it with
   any global definition of the same name. Weak symbols may also
   stay undefined, in which case the linker would assign them a value
   of zero.

#define LOCAL (1<<7)
   Only informational. A symbol can be explicitly declared as local
   by a syntax-module directive.

#define VASMINTERN (1<<8)
   Vasm-internal symbol, which is usually not exported into an output
   file.

#define PROTECTED (1<<9)
   Used internally to protect the current-PC symbol from deletion.

#define REFERENCED (1<<10)
   Symbol was referenced in the source and a relocation entry has
   been created.

#define ABSLABEL (1<<11)
   Label was defined inside an absolute section, or during relocated-
   org mode. So it has an absolute address and will not generate a
   relocation entry when being referenced.

#define EQUATE (1<<12)
   Symbols flagged as EQUATE are constant and its value must not be
   changed.

#define REGLIST (1<<13)
   Symbol is a register list definition.

#define USED (1<<14)
   Symbol appeared in an expression. Symbols which were only de-
   fined, (as label or equate) and never used throughout the whole
   source, don’t get this flag set.

#define NEAR (1<<15)
   Symbol may be referenced by "near" addressing mode. For exam-
   ple, base register relative. Used as an optimization hint in the cpu
   backend.

#define RSRVD_S (1L<<24)
   The range from bit 24 to 27 (counted from the LSB) is reserved for
   use by the syntax module.
```
#define RSRVD_0 (1L<<28)

The range from bit 28 to 31 (counted from the LSB) is reserved for use by the output module.

The type-flags can be extracted using the TYPE() macro which expects a pointer to a symbol as argument.

char *name;
The name of the symbol.

expr *expr;
The expression in case of EXPRESSION symbols.

expr *size;
The size of the symbol, if specified.

section *sec;
The section a LABSYM symbol is defined in.

taddr pc;  The address of a LABSYM symbol.

taddr align;
The alignment of the symbol in bytes.

unsigned long idx;
A member usable by the output module for private purposes.

26.3.4 Register symbols

Optional register symbols are available when the backend defines HAVE_REGSYMS in cpu.h together with the hash table size. Example:

#define HAVE_REGSYMS
#define REGSYMHTSIZE 256

A register symbol is defined by an object of type regsym with the following members that can be accessed by the modules:

char *reg_name;
Symbol name.

int reg_type;
Optional type of register.

unsigned int reg_flags;
Optional register symbol flags.

unsigned int reg_num;
Register number or value.

Refer to symbol.h for functions to create and find register symbols.

26.3.5 Atoms

The contents of each section are a linked list built out of non-separable atoms. The general structure of an atom is:
typedef struct atom {
    struct atom *next;
    int type;
    taddr align;
    taddr lastsize;
    unsigned changes;
    source *src;
    int line;
    listing *list;
    union {
        instruction *inst;
        dblock *db;
        symbol *label;
        sblock *sb;
        defblock *defb;
        void *opts;
        int srcline;
        char *ptext;
        printexpr *pexpr;
        expr *roffs;
        taddr *rorg;
        assertion *assert;
        aoutnlist *nlist;
    } content;
} atom;

The members have the following meaning:

struct atom *next;
    Pointer to the following atom (0 if last).

int type;  The type of the atom. Can be one of

#define LABEL 1
    A label is defined here.

#define DATA 2
    Some data bytes of fixed length and constant data are put here.

#define INSTRUCTION 3
    Generally refers to a machine instruction or pseudo/opcode. These
    atoms can change length during optimization passes and will be
    translated to DATA-atoms later.

#define SPACE 4
    Defines a block of data filled with one value (byte). BSS sections
    usually contain only such atoms, but they are also sometimes useful
    as shorter versions of DATA-atoms in other sections.

#define DATADEF 5
    Defines data of fixed size which can contain cpu specific operands
    and expressions. Will be translated to DATA-atoms later.
#define LINE 6
A source text line number (usually from a high level language) is bound to the atom’s address. Useful for source level debugging in certain ABIs.

#define OPTS 7
A means to change assembler options at a specific source text line. For example optimization settings, or the cpu type to generate code for. The cpu module has to define HAVE_CPU_OPTS and export the required functions if it wants to use this type of atom.

#define PRINTTEXT 8
A string is printed to stdout during the final assembler pass. A newline is automatically appended.

#define PRINTEXPR 9
Prints the value of an expression during the final assembler pass to stdout.

#define ROFFS 10
Set the program counter to an address relative to the section’s start address. These atoms will be translated into SPACE atoms in the final pass.

#define RORG 11
Assemble this block under the given base address, while the code is still written into the original memory region.

#define RORGEND 12
Ends a RORG block and returns to the original addressing.

#define ASSERT 13
The assertion expression is checked in the final pass and an error message is generated (using the expression string and an optional message out of this atom) when it evaluates to 0.

#define NLIST 14
Defines a stab-entry for the a.out object file format. nlist-style stabs can also occur embedded in other object file formats, like ELF.

taddr align;
The alignment of this atom. Address must be dividable by align.

taddr lastsize;
The size of this atom in the last resolver pass. When the size has changed in the current pass, the assembler will request another resolver run through the section.

unsigned changes;
Number of changes in the size of this atom since pass number FASTOPTPHASE. An increasing number usually indicates a problem in the cpu backend's optimizer and will be flagged by setting RESOLVE_WARN in the Section flags, as soon as changes exceeds MAXSIZECHANGES. So the backend can choose not to optimize this atom as aggressive as before.
source *src;
   Pointer to the source text object to which this atom belongs.

int line; The source line number that created this atom.

listing *list;
   Pointer to the listing object to which this atoms belong.

instruction *inst;
   (In union content.) Pointer to an instruction structure in the case of an
   INSTRUCTION-atom. Contains the following elements:

   int code; The cpu specific code of this instruction.

   char *qualifiers[MAX_QUALIFIERS];
      (If MAX_QUALIFIERS!=0.) Pointer to the qualifiers of this instruc-
   tion.

   operand *op[MAX_OPERANDS];
      (If MAX_OPERANDS!=0.) The cpu-specific operands of this instruc-
   tion.

   instruction_ext ext;
      (If the cpu module defines HAVE_INSTRUCTION_EXTENSION.) A cpu-
   module-specific structure. Typically used to store appropriate op-
   codes, allowed addressing modes, supported cpu derivates etc.

dblock *db;
   (In union content.) Pointer to a dblock structure in the case of a DATA-atom.
   Contains the following elements:

   taddr size;
      The number of bytes stored in this atom.

   char *data;
      A pointer to the data.

   rlist *relocs;
      A pointer to relocation information for the data.

symbol *label;
   (In union content.) Pointer to a symbol structure in the case of a LABEL-atom.

sblock *sb;
   (In union content.) Pointer to a sblock structure in the case of a SPACE-atom.
   Contains the following elements:

   size_t space;
      The size of the empty/filled space in bytes.

   expr *space_exp;
      The above size as an expression, which will be evaluated during
   assembly and copied to space in the final pass.

   size_t size;
      The size of each space-element and of the fill-pattern in bytes.
unsigned char fill[MAXBYTES];
    The fill pattern, up to MAXBYTES bytes.

expr *fill_exp;
    Optional. Evaluated and copied to fill in the final pass, when not null.

rlist *relocs;
    A pointer to relocation information for the space.

taddr maxalignbytes;
    An optional number of maximum padding bytes to fulfil the atom’s alignment requirement. Zero means there is no restriction.

uint32_t flags;
    SPC_DATABSS
    The output module should not allocate any file space for this atom, when possible (example: DataBss section, as supported by the "hunkexe" output file format). It is not needed to set this flag when the output section is BSS.

defblock *defb;
    (In union content.) Pointer to a defblock structure in the case of a DATADEF-atom. Contains the following elements:

    taddr bitsize;
    The size of the definition in bits.

    operand *op;
    Pointer to a cpu-specific operand structure.

void *opts;
    (In union content.) Points to a cpu module specific options object in the case of a OPTS-atom.

int srcline;
    (In union content.) Line number for source level debugging in the case of a LINE-atom.

char *ptext;
    (In union content.) A string to print to stdout in case of a PRINTTEXT-atom.

printexpr *pexpr;
    (In union content.) Pointer to a printexpr structure in the case of a PRINTEXPR-atom. Contains the following elements:

    expr *print_exp;
    Pointer to an expression to evaluate and print.

short type;
    Format type of the printed value. We can print as hexadecimal (PEXP_HEX), signed decimal (PEXP_SDEC), unsigned decimal (PEXP_UDEC), binary (PEXP_BIN) OR ASCII (PEXP_ASC).
short size;
   Size (precision) of the printed value in bits. Excessive bits will be
   masked out, and sign-extended when requested.

expr *roffs;
   (In union content.) The expression holds the relative section offset to align to
   in case of a ROFFS-atom.

taddr *rorg;
   (In union content.) Assemble the code under the base address in rorg in case
   of a RORG-atom.

assertion *assert;
   (In union content.) Pointer to an assertion structure in the case of an ASSERT-
   atom. Contains the following elements:

   expr *assert_exp;
      Pointer to an expression which should evaluate to non-zero.

   char *exprstr;
      Pointer to the expression as text (to be used in the output).

   char *msgstr;
      Pointer to the message, which would be printed when assert_exp
      evaluates to zero.

aoutnlist *nlist;
   (In union content.) Pointer to an nlist structure, describing an aout stab entry,
   in case of an NLIST-atom. Contains the following elements:

   char *name;
      Name of the stab symbol.

   int type;  Symbol type. Refer to stabs.h for definitions.

   int other;
      Defines the nature of the symbol (function, object, etc.).

   int desc;  Debugger information.

   expr *value;
      Symbol’s value.

26.3.6 Relocations

DATA and SPACE atoms can have a relocation list attached that describes how this data must
be modified when linking/relocating. They always refer to the data in this atom only.

There are a number of predefined standard relocations and it is possible to add other cpu-
specific relocations. Note however, that it is always preferrable to use standard relocations,
if possible. Chances that an output module supports a certain relocation are much higher
if it is a standard relocation.

A relocation list uses this structure:

    typedef struct rlist {
       struct rlist *next;
    }
void *reloc;
    int type;
} rlist;

Type identifies the relocation type. All the standard relocations have type numbers between FIRST_STANDARD_RELOC and LAST_STANDARD_RELOC. Consider reloc.h to see which standard relocations are available.

The detailed information can be accessed via the pointer reloc. It will point to a structure that depends on the relocation type, so a module must only use it if it knows the relocation type.

All standard relocations point to a type nreloc with the following members:

size_t byteoffset;
    Offset in bytes, from the start of the current DATA atom, to the beginning of the relocation field. This may also be the address which is used as a basis for PC-relative relocations. Or a common basis for several separated relocation fields, which will be translated into a single relocation type by the output module.

size_t bitoffset;
    Offset in bits to the beginning of the relocation field, adds to byteoffset*bitsperbyte. Bits are counted in a bit-stream from lower to higher address bytes. But note, that inside a little-endian byte they are counted from the LSB to the MSB, while they are counted from the MSB to the LSB for big-endian targets.

int size;    The size of the relocation field in bits.

taddr mask;
    The mask defines which portion of the relocated value is set by this relocation field.

taddr addend;
    Value to be added to the symbol value.
	symbol *sym;
    The symbol referred by this relocation

To describe the meaning of these entries, we will define the steps that shall be executed when performing a relocation:

1. Extract the size bits from the data atom, starting with bit number byteoffset*bitsperbyte+bitoffset. We start counting bits from the lowest to the highest numbered byte in memory. Inside a big-endian byte we count from the MSB to the LSB. Inside a little-endian byte we count from the LSB to the MSB.

2. Determine the relocation value of the symbol. For a simple absolute relocation, this will be the value of the symbol sym plus the addend. For other relocation types, more complex calculations will be needed. For example, in a program-counter relative relocation, the value will be obtained by subtracting the address of the data atom plus byteoffset from the value of sym plus addend.

3. Calculate the bit-wise "and" of the value obtained in the step above and the mask value.
4. Normalize, i.e. shift the value above right as many bit positions as there are low order zero bits in \texttt{mask}.

5. Add this value to the value extracted in step 1.

6. Insert the low order \texttt{size} bits of this value into the data atom starting with bit \texttt{byteoffset*bitsperbyte+bitoffset}.

\subsection*{26.3.7 Errors}

Each module can provide a list of possible error messages contained e.g. in \texttt{syntax_errors.h} or \texttt{cpu_errors.h}. They are a comma-separated list of a printf-format string and error flags. Allowed flags are \texttt{WARNING}, \texttt{ERROR}, \texttt{FATAL}, \texttt{MESSAGE} and \texttt{NOLINE}. They can be combined using or (|). \texttt{NOLINE} has to be set for error messages during initialisation or while writing the output, when no source text is available. Errors cause the assembler to return false. \texttt{FATAL} causes the assembler to terminate immediately.

The errors can be emitted using the function \texttt{syntax_error(int n,...)}, \texttt{cpu_error(int n,...)} or \texttt{output_error(int n,...)}. The first argument is the number of the error message (starting from zero). Additional arguments must be passed according to the format string of the corresponding error message.

\subsection*{26.4 Syntax modules}

A new syntax module must have its own subdirectory under \texttt{vasm/syntax}. At least the files \texttt{syntax.h}, \texttt{syntax.c} and \texttt{syntax_errors.h} must be written.

\subsubsection*{26.4.1 The file syntax.h}

\texttt{#define ISIDSTART(x)/ISIDCHAR(x)}

These macros should return non-zero if and only if the argument is a valid character to start an identifier or a valid character inside an identifier, respectively. \texttt{ISIDCHAR} must be a superset of \texttt{ISIDSTART}.

\texttt{#define ISBADID(p,l)}

Even with \texttt{ISIDSTART} and \texttt{ISIDCHAR} checked, there may be combinations of characters which do not form a valid initializer (for example, a single character). This macro returns non-zero, when this is the case. First argument is a pointer to the new identifier and second is its length.

\texttt{#define ISEOL(x)}

This macro returns true when the string pointing at \texttt{x} is either a comment character or end-of-line.

\texttt{#define CHKIDEND(s,e) chkidend((s),(e))}

Defines an optional function to be called at the end of the identifier recognition process. It allows you to adjust the length of the identifier by returning a modified \texttt{e}. Default is to return \texttt{e}. The function is defined as \texttt{char *chkidend(char *startpos,char *endpos)}. 

\texttt{#define BOOLEAN(x) -(x)}

Defines the result of boolean operations. Usually this is \texttt{(x)}, as in C, or \texttt{-(x)} to return -1 for True.
#define NARGSYM "NARG"
Defines the name of an optional symbol which contains the number of arguments in a macro.

#define CARGSYM "CARG"
Defines the name of an optional symbol which can be used to select a specific macro argument with \, \+ and \-.

#define REPTNSYM "REPTN"
Defines the name of an optional symbol containing the counter of the current repeat iteration.

#define EXPSKIP() s=exp_skip(s)
Defines an optional replacement for skip() to be used in expr.c, to skip blanks in an expression. Useful to forbid blanks in an expression and to ignore the rest of the line (e.g. to treat the rest as comment). The function is defined as char *exp_skip(char *stream).

#define IGNORE_FIRST_EXTRA_OP 1
Should be defined when the syntax module wants to ignore the operand field on instructions without an operand. Useful, when everything following an operand should be regarded as comment, without a comment character.

#define MAXMACPARAMS 35
Optionally defines the maximum number of macro arguments, if you need more than the default number of 9.

#define SKIP_MACRO_ARGNAME(p) skip_identifier(p)
An optional function to skip a named macro argument in the macro definition. Argument is the current source stream pointer. The default is to skip an identifier.

#define MACRO_ARG_OPTS(m,n,a,p) NULL
An optional function to parse and skip options, default values and qualifiers for each macro argument. Returns NULL when no argument options have been found. Arguments are:
struct macro *m;
    Pointer to the macro structure being currently defined.
int n;    Argument index, starting with zero.
char *a;    Name of this argument.
char *p;    Current source stream pointer. An updated pointer will be returned.

Defaults to unused.

#define MACRO_ARG_SEP(p) (*p==',' ? skip(p+1) : NULL)
An optional function to skip a separator between the macro argument names in the macro definition. Returns NULL when no valid separator is found. Argument is the current source stream pointer. Defaults to using comma as the only valid separator.
#define MACRO_PARAM_SEP(p) (*p==',' ? skip(p+1) : NULL)
An optional function to skip a separator between the macro parameters in a
macro call. Returns NULL when no valid separator is found. Argument is
the current source stream pointer. Defaults to using comma as the only valid
separator.

#define EXEC_MACRO(s)
An optional function to be called just before a macro starts execution. Parameters and qualifiers are already parsed. Argument is the source pointer of the
new macro. Defaults to unused.

26.4.2 The file syntax.c
A syntax module has to provide the following elements (all other functions should be static
to prevent name clashes):

char *syntax_copyright;
A string that will be emitted as part of the copyright message.

hashtable *dirhash;
A pointer to the hash table with all directives.

char commentchar;
A character used to introduce a comment until the end of the line.

char *defsectname;
Name of a default section which vasm creates when a label or code occurs in
the source, but the programmer forgot to specify a section. Assigning NULL
means that there is no default and vasm will show an error in this case.

char *defsecttype;
Type of the default section (see above). May be NULL.

int init_syntax();
Will be called during startup, after argument parsing Must return zero if ini-
tializations failed, non-zero otherwise.

int syntax_args(char *);
This function will be called with the command line arguments (unless they were
already recognized by other modules). If an argument was recognized, return
non-zero.

char *skip(char *);
A function to skip whitespace etc.

char *skip_operand(char *);
A function to skip an instruction’s operand. Will terminate at end of line or
the next comma, returning a pointer to the rest of the line behind the comma.

void eol(char *);
This function should check that the argument points to the end of a line (only
comments or whitespace following). If not, an error or warning message should
be omitted.
char *const_prefix(char *,int *);
    Check if the first argument points to the start of a constant. If yes return a
    pointer to the real start of the number (i.e. skip a prefix that may indicate the
    base) and write the base of the number through the pointer passed as second
    argument. Return zero if it does not point to a number.

char *const_suffix(char *,char *);
    First argument points to the start of the constant (including prefix) and the
    second argument to first character after the constant (excluding suffix). Checks
    for a constant-suffix and skips it. Return pointer to the first character after
    that constant. Example: constants with a 'h' suffix to indicate a hexadecimal
    base.

void parse(void);
    This is the main parsing function. It has to read lines via the read_next_
    line() function, parse them and create sections, atoms and symbols. Pseudo
directives are usually handled by the syntax module. Instructions can be parsed
by the cpu module using parse_instruction().

char *parse_macro_arg(struct macro *,char *,struct namelen *,struct namelen *
);
    Called to parse a macro parameter by using the source stream pointer in the
second argument. The start pointer and length of a single passed parameter is
written to the first struct namelen, while the optionally selected named macro
argument is passed in the second struct namelen. When the len field of the
second namelen is zero, then the argument is selected by position instead by
name. Returns the updated source stream pointer after successful parsing.

int expand_macro(source *,char **,char *,int);
    Expand parameters and special commands inside a macro source. The second
argument is a pointer to the current source stream pointer, which is updated
on any sucessful expansion. The function will return the number of charac-
ters written to the destination buffer (third argument) in this case. Returning
-1 means: no expansion took place. The last argument defines the space in
characters which is left in the destination buffer.

char *get_local_label(char **);
    Gets a pointer to the current source pointer. Has to check if a valid local label
is found at this point. If yes return a pointer to the vasm-internal symbol name
representing the local label and update the current source pointer to point
behind the label.

    Have a look at the support functions provided by the frontend to help.

26.5 CPU modules
A new cpu module must have its own subdirectory under vasm/cpus. At least the files
cpu.h, cpu.c and cpu_errors.h must be written.
26.5.1 The file cpu.h

A cpu module has to provide the following elements (all other functions should be static to prevent name clashes) in cpu.h:

```c
#define MAX_OPERANDS 3
    Maximum number of operands of one instruction.
#define MAX_QUALIFIERS 0
    Maximum number of mnemonic-qualifiers per mnemonic.
#define NO_MACRO_QUALIFIERS
    Define this, when qualifiers shouldn’t be allowed for macros. For some architectures, like ARM, macro qualifiers make no sense.
typedef int32_t taddr;
    Data type to represent a target-address. Preferrably use the ones from stdint.h.
typedef uint32_t utaddr;
    Unsigned data type to represent a target-address.
#define LITTLEENDIAN 1
#define BIGENDIAN 0
    Define these according to the target endianess. For CPUs which support big- and little-endian, you may assign a global variable here. So be aware of it, and never use #if BIGENDIAN, but always if(BIGENDIAN) in your code.
#define VASM_CPU_<cpu> 1
    Insert the cpu specifier.
#define INST_ALIGN 2
    Minimum instruction alignment.
#define DATA_ALIGN(n) ...
    Default alignment for n-bit data. Can also be a function.
#define DATA_OPERAND(n) ...
    Operand class for n-bit data definitions. Can also be a function. Negative values denote a floating point data definition of -n bits.
typedef ... operand;
    Structure to store an operand.
typedef ... mnemonic_extension;
    Mnemonic extension.
```

Optional features, which can be enabled by defining the following macros:

```c
#define HAVE_INSTRUCTION_EXTENSION 1
    If cpu-specific data should be added to all instruction atoms.
typedef ... instruction_ext;
    Type for the above extension.
#define NEED_CLEARED_OPERANDS 1
    Backend requires a zeroed operand structure when calling parse_operand() for the first time. Defaults to undefined.
```
START_PARENTH(x)
Valid opening parenthesis for instruction operands. Defaults to ’(’.

END_PARENTH(x)
Valid closing parenthesis for instruction operands. Defaults to ’)’.

#define MNEMONIC_VALID(i)
An optional function with the arguments (int idx). Returns true when the
mnemonic with index idx is valid for the current state of the backend (e.g. it
is available for the selected cpu architecture).

#define MNEMOHTABSIZE 0x4000
You can optionally overwrite the default hash table size defined in vasm.h. May
be necessary for larger mnemonic tables.

#define OPERAND_OPTIONAL(p,t)
When defined, this is a function with the arguments (operand *op,int type),
which returns true when the given operand type (type) is optional. The func-
tion is only called for missing operands and should also initialize op with default
values (e.g. 0).

Implementing additional target-specific unary operations is done by defining the following
optional macros:

#define EXT_UNARY_NAME(s)
Should return True when the string in s points to an operation name we want
to handle.

#define EXT_UNARY_TYPE(s)
Returns the operation type code for the string in s. Note that the last valid
standard operation is defined as LAST_EXP_TYPE, so the target-specific types
will start with LAST_EXP_TYPE+1.

#define EXT_UNARY_EVAL(t,v,r,c)
Defines a function with the arguments (int t, taddr v, taddr *r, int c) to
handle the operation type t returning an int to indicate whether this type has
been handled or not. Your operation will by applied on the value v and the
result is stored in *r. The flag c is passed as 1 when the value is constant (no
relocatable addresses involved).

#define EXT_FIND_BASE(b,e,s,p)
Defines a function with the arguments (symbol **b, expr *e, section *s,
taddr p) to save a pointer to the base symbol of expression e into the symbol
pointer, pointed to by b. The type of this base is given by an int return code.
Further on, e->type has to checked to be one of the operations to handle.
The section pointer s and the current pc p are needed to call the standard
find_base() function.

26.5.2 The file cpu.c
A cpu module has to provide the following elements (all other functions and data should
be static to prevent name clashes) in cpu.c:
int bitsperbyte;
The number of bits per byte of the target cpu.

int bytesspertaddr;
The number of bytes per taddr.

mnemonic mnemonics[];
The mnemonic table keeps a list of mnemonic names and operand types the assembler will match against using parse_operand(). It may also include a target specific mnemonic_extension.

char *cpu_copyright;
A string that will be emitted as part of the copyright message.

char *cpuname;
A string describing the target cpu.

int init_cpu();
Will be called during startup, after argument parsing. Must return zero if initializations failed, non-zero otherwise.

int cpu_args(char *);
This function will be called with the command line arguments (unless they were already recognized by other modules). If an argument was recognized, return non-zero.

char *parse_cpu_special(char *);
This function will be called with a source line as argument and allows the cpu module to handle cpu-specific directives etc. Functions like eol() and skip() should be used by the syntax module to keep the syntax consistent.

operand *new_operand();
Allocate and initialize a new operand structure.

int parse_operand(char *text, int len, operand *out, int requires);
Parses the source at text with length len to fill the target specific operand structure pointed to by out. Returns PO_MATCH when the operand matches the operand-type passed in requires and PO_NOMATCH otherwise. When the source is definitely identified as garbage, the function may return PO_CORRUPT to tell the assembler that it is useless to try matching against any other operand types. Another special case is PO_SKIP, which is also a match, but skips the next operand from the mnemonic table (because it was already handled together with the current operand).

taddr instruction_size(instruction *ip, section *sec, taddr pc);
Returns the size of the instruction ip in bytes, which must be identical to the number of bytes written by eval_instruction() (see below).

dblock *eval_instruction(instruction *ip, section *sec, taddr pc);
Converts the instruction ip into a DATA atom, including relocations, if necessary.

dblock *eval_data(operand *op, taddr bitsize, section *sec, taddr pc);
Converts a data operand into a DATA atom, including relocations.
void init_instruction_ext(instruction_ext *);
   (If HAVE_INSTRUCTION_EXTENSION is set.) Initialize an instruction extension.

char *parse_instruction(char *,int *,char **,int *,int *);
   (If MAX_QUALIFIERS is greater than 0.) Parses instruction and saves extension locations.

int set_default_qualifiers(char **,int *);
   (If MAX_QUALIFIERS is greater than 0.) Saves pointers and lengths of default
   qualifiers for the selected CPU and returns the number of default qualifiers.
   Example: for a M680x0 CPU this would be a single qualifier, called "w". Used
   by execute_macro().

cpu_opts_init(section *);
   (If HAVE_CPU_OPTS is set.) Gives the cpu module the chance to write out OPTS
   atoms with initial settings before the first atom is generated.

cpu_opts(void *);
   (If HAVE_CPU_OPTS is set.) Apply option modifications from an OPTS atom. For
   example: change cpu type or optimization flags.

print_cpu_opts(FILE *,void *);
   (If HAVE_CPU_OPTS is set.) Called from print_atom() to print an OPTS atom’s
   contents.

26.6 Output modules

Output modules can be chosen at runtime rather than compile time. Therefore, several
output modules are linked into one vasm executable and their structure differs somewhat
from syntax and cpu modules.

Usually, an output module for some object format fmt should be contained in a file output_<fmt>.c (it may use/include other files if necessary). To automatically include this format
in the build process, the make.rules has to be extended. The module should be added to
the OBJS variable at the start of make.rules. Also, a dependency line should be added (see
the existing output modules).

An output module must only export a single function which will return pointers to necessary
data/functions. This function should have the following prototype:

   int init_output_<fmt>(
      char **copyright,
      void (**write_object)(FILE *,section *,symbol *),
      int (**output_args)(char *)
   );

In case of an error, zero must be returned. Otherwise, it should perform all necessary
initializations, return non-zero and return the following output parameters via the pointers
passed as arguments:

   copyright
      A pointer to the copyright string.
write_object
A pointer to a function emitting the output. It will be called after the assembler has completed and will receive pointers to the output file, to the first section of the section list and to the first symbol in the symbol list. See the section on general data structures for further details.

output_args
A pointer to a function checking arguments. It will be called with all command line arguments (unless already handled by other modules). If the output module recognizes an appropriate option, it has to handle it and return non-zero. If it is not an option relevant to this output module, zero must be returned.

At last, a call to the `output_init_<fmt>` has to be added in the `init_output()` function in `vasm.c` (should be self-explanatory).

Some remarks:
- Some output modules can not handle all supported CPUs. Nevertheless, they have to be written in a way that they can be compiled. If code references CPU-specifcics, they have to be enclosed in `#ifdef VASM_CPU_MYCPU ... #endif` or similar.
  Also, if the selected CPU is not supported, the init function should fail.
- Error/warning messages can be emitted with the `output_error` function. As all output modules are linked together, they have a common list of error messages in the file `output_errors.h`. If a new message is needed, this file has to be extended (see the section on general data structures for details).
- `vasm` has a mechanism to specify rather complex relocations in a standard way (see the section on general data structures). They can be extended with CPU specific relocations, but usually CPU modules will try to create standard relocations (sometimes several standard relocations can be used to implement a CPU specific relocation). An output module should try to find appropriate relocations supported by the object format. The goal is to avoid special CPU specific relocations as much as possible.

Volker Barthelmann vb@compilers.de