

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	3
2.3	Known Problems	4
2.4	Credits	4
2.5	Error Messages	4
3	Standard Syntax Module	5
3.1	Legal	5
3.2	Additional options for this version	5
3.3	General Syntax	5
3.4	Directives	5
3.5	Known Problems	9
3.6	Error Messages	9
4	Mot Syntax Module	11
4.1	Legal	11
4.2	Additional options for this version	11
4.3	General Syntax	11
4.4	Directives	11
4.5	Known Problems	14
4.6	Error Messages	15
5	Simple binary output module	17
5.1	Legal	17
5.2	Additional options for this version	17
5.3	General	17
5.4	Restrictions	17
5.5	Known Problems	17
5.6	Error Messages	17
6	Test output module	19
6.1	Legal	19
6.2	Additional options for this version	19
6.3	General	19
6.4	Restrictions	19
6.5	Known Problems	19
6.6	Error Messages	19

7	ELF output module	21
7.1	Legal	21
7.2	Additional options for this version	21
7.3	General	21
7.4	Restrictions	21
7.5	Known Problems	21
7.6	Error Messages	21
8	vobj output module	23
8.1	Legal	23
8.2	Additional options for this version	23
8.3	General	23
8.4	Restrictions	23
8.5	Known Problems	23
8.6	Error Messages	23
9	Amiga output module	25
9.1	Legal	25
9.2	Additional options for this version	25
9.3	General	25
9.4	Restrictions	25
9.5	Known Problems	25
9.6	Error Messages	25
10	m68k cpu module	27
10.1	Legal	27
10.2	Additional options for this module	27
10.3	General	28
10.4	Extensions	28
10.5	Optimizations	29
10.6	Known Problems	32
10.7	Error Messages	32
11	PowerPC cpu module	33
11.1	Legal	33
11.2	Additional options for this module	33
11.3	General	33
11.4	Extensions	34
11.5	Optimizations	34
11.6	Known Problems	34
11.7	Error Messages	34

12	c16x/st10 cpu module	35
12.1	Legal	35
12.2	Additional options for this module	35
12.3	General	35
12.4	Extensions	35
12.5	Optimizations	36
12.6	Known Problems	36
12.7	Error Messages	36
13	6502 cpu module	37
13.1	Legal	37
13.2	Additional options for this module	37
13.3	General	37
13.4	Extensions	37
13.5	Optimizations	37
13.6	Known Problems	37
13.7	Error Messages	38
14	Interface	39
14.1	Introduction	39
14.2	Building vasm	39
14.2.1	Directory Structure	39
14.2.2	Adapting the Makefile	39
14.2.3	Building vasm	40
14.3	General data structures	40
14.3.1	Sections	40
14.3.2	Symbols	41
14.3.3	Atoms	42
14.3.4	Relocations	44
14.3.5	Errors	45
14.4	Syntax modules	45
14.5	CPU modules	46
14.5.1	The file 'cpu.h'	46
14.5.2	The file 'cpu.c'	47
14.6	Output modules	48

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).

1.2 Legal

vasm is copyright in 2002 by Volker Barthelmann.

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

Distributing modified versions and commercial usage needs 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`.

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

`vasm` expects the following syntax:

```
vasm<target>_<syntax> [options] file
```

The following options are supported by the machine independent part of `vasm`:

- '-quiet' Do not print the copyright notice.
- '-o ofile' Write the generated assembler output to <ofile> rather than 'a.out'
- '-F<fmt>' Use module <fmt> as output driver. See the chapter on output drivers for available formats and options.

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 operators available are similar to the ones available in C expressions and have the same precedence and associativity as in the C language. All the standard arithmetic (+, -, *, /, %), bitwise (&, |, ^), logical (&&, ||) and conditional operators (?:, !, <, >, <=, >=, ==, !=) are allowed.

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 endianness of the target.

Inside character constants, the following escape sequences are allowed:

- \\ 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`.

2.3 Known Problems

Some known module-independent problems of `vasm` at the moment:

- None.

2.4 Credits

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

Frank Wille

2.5 Error Messages

The frontend has the following error messages:

FIXME

3 Standard Syntax Module

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

3.1 Legal

This module is copyright in 2002 by Volker Barthelmann.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

3.2 Additional options for this version

This syntax module provides the following additional options:

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

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

`'-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 (`:`). 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 (`,`).

Comments are introduced by the comment character `#`. The rest of the line will be ignored. For the `c16x` and `m68k` backends, the comment character is `;` instead of `#`.

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`.

3.4 Directives

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`.

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

See `.string`.

`.align <bit_count>[,<fill>]`

Insert as much fill 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.

`.bss` Equivalent to `.section .bss,"aurw4"`.

`.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.

`.data` Equivalent to `.section .data,"adrw4"`.

`.equ <symbol>,<expression>`

See `.set`.

`.extern <symbol>`

See `.global`.

`.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.

`.global <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>`

See `.global`.

`.half <exp1>[,<exp2>...]`

If the current section location counter is not on a halfword boundary, advance it to the next halfword boundary. Then, assign the values of the operands into successive halfwords of memory in the current section.

`.int <exp1>[,<exp2>...]`

See `.word`.

`.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.

`.long <exp1>[,<exp2>...]`

See `.word`.

`.quad <exp1>[,<exp2>...]`

If the current section location counter is not on a quadword boundary, advance it to the next quadword boundary. Then, assign the values of the operands into successive quadwords of memory in the current section.

`.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:

<code>c</code>	section has code
<code>d</code>	section has initialized data
<code>u</code>	section has uninitialized data
<code>i</code>	section has directives (info section)
<code>n</code>	section can be discarded
<code>R</code>	remove section at link time
<code>a</code>	section is allocated in memory

Section Protection:

<code>r</code>	section is readable
<code>w</code>	section is writable
<code>x</code>	section is executable
<code>s</code>	section is sharable

Section Alignment (only one):

0	align to byte boundary
1	align to halfword boundary
2	align to word boundary
3	align to doubleword boundary
4	align to quadword boundary
5	align to 32 byte boundary
6	align to 64 byte boundary

Memory flags:

C	load section to Chip RAM
F	load section to Fast RAM

`.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`.

`.space <exp>[,<fill>]`

See `.space`.

`.space <exp>[,<fill>]`

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

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

Like `.byte`, but adds a terminating zero-byte.

`.text` Equivalent to `.section .text,"acrx4"`.

`.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 two-byte areas of memory in the current section regardless of section alignment.

`.ualong <exp1>[,<exp2>...]`

See `.uaword`.

`.uquad <exp1>[,<exp2>...]`

Assign the values of the operands into successive eight-byte areas of memory in the current section regardless of section alignment.

`.uashort <exp1>[,<exp2>...]`

See `.uahalf`.

`.uaword <exp1>[,<exp2>...]`

Assign the values of the operands into successive four-byte areas of memory in the current section regardless of section alignment.

`.word <exp1>[,<exp2>...]`

If the current section location counter is not on a word boundary advance it to the next word boundary. Then assign the values of the operands into successive words of memory in the current section.

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:

FIXME

4 Mot Syntax Module

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

4.1 Legal

This module is copyright in 2002 by Frank Wille.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

4.2 Additional options for this version

This syntax module provides the following additional options:

- '`-nocase`' Disables case-sensitivity for everything - identifiers, directives and instructions.
- '`-align`' Enables 16-bit alignment for constant declaration (`dc.?`, except `dc.b`) directives.
- '`-phxass`' PhxAss-compatibility mode. At the moment it has only the effect that the "current PC symbol" (*) is set to the instruction's address + 2.

4.3 General Syntax

Labels always start at the first column and may be terminated by a 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. `=`). 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 (,).

Comments are introduced by the comment character `;` or `*`. The rest of the line will be ignored. `*` should only be used at the beginning of a line to avoid conflicts with the "current pc symbol".

Example:

```
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 (if the CPU- and output-module allow it):

`<symbol> = <expression>`

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

`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. Equivalent to `cnop 0,1<bitcount>`.

`blk.b <exp>[,<fill>]`

Equivalent to `dc.b <exp>,<fill>`.

`blk.d <exp>[,<fill>]`

Equivalent to `dc.d <exp>,<fill>`.

`blk.l <exp>[,<fill>]`

Equivalent to `dc.l <exp>,<fill>`.

`blk.q <exp>[,<fill>]`

Equivalent to `dc.q <exp>,<fill>`.

`blk.s <exp>[,<fill>]`

Equivalent to `dc.s <exp>,<fill>`.

`blk.w <exp>[,<fill>]`

Equivalent to `dc.w <exp>,<fill>`.

`blk.x <exp>[,<fill>]`

Equivalent to `dc.x <exp>,<fill>`.

`bss` Equivalent to `section bss,bss`.

`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.

`code` Equivalent to `section code,code`.

`cseg` Equivalent to `section code,code`.

`data` Equivalent to `section data,data`.

`dc.b <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.

`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 32-bit words of memory in the current section.
- `dc.s <exp1>[,<exp2>...]`
Assign the values of the operands into successive 64-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 64-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.
- `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> 86-bit words into the current section. <fill> might also be an IEEE extended precision constant.
- `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`.
- `end` Assembly will terminate behind this line.
- `<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.
- `even` Aligns to an even address. Equivalent to `cnop 0,2`.
- `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.
- `public <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.
- `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`, `data` or `bss`. `<sec_type>` defaults to `code`. When `<mem_type>` is given it defines the type of memory, where the section can be loaded. This is Amiga-specific and allowed identifiers are `chip` for Chip-RAM and `fast` for Fast-RAM.
- `<symbol> set <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.
- `ttl <name>`
PhxAss syntax. Equivalent to `idnt <name>`.
- `<name> ttl`
Motorola syntax. Equivalent to `idnt <name>`.
- `xdef <symbol>`
Flag `<symbol>` as an global symbol, which means that `<symbol>` is visible to all modules in the linking process. See also `public`.
- `xref <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:

- None.

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: \ expected
- 1009: invalid data operand
- 1010: , expected
- 1011: identifier expected
- 1012: illegal escape sequence <c>
- 1013: expression must be a constant
- 1014: illegal section type
- 1015: repeatedly defined symbol
- 1016: illegal memory type

5 Simple binary output module

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

5.1 Legal

This module is copyright in 2002 by Volker Barthelmann.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

5.2 Additional options for this version

This output module provides no additional options.

5.3 General

This output module outputs the contents of the first section as simple binary data without any header or additional information.

5.4 Restrictions

This module supports only one section. Section types, section alignment, filename, type-names, etc. are ignored.

5.5 Known Problems

Some known problems of this module at the moment:

- None.

5.6 Error Messages

This module has the following error messages:

FIXME

6 Test output module

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

6.1 Legal

This module is copyright in 2002 by Volker Barthelmann.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

6.2 Additional options for this version

This output module provides no additional options.

6.3 General

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

6.4 Restrictions

None.

6.5 Known Problems

Some known problems of this module at the moment:

- None.

6.6 Error Messages

This module has the following error messages:

FIXME

7 ELF output module

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

7.1 Legal

This module is copyright in 2002 by Frank Wille.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

7.2 Additional options for this version

This output module provides no additional options.

7.3 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.

7.4 Restrictions

The **ELF** output format, as implemented in `vasm`, currently supports the following architectures:

- PowerPC
- M68k

The supported relocation types depend on the selected architecture.

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:

FIXME

8 vobj output module

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

8.1 Legal

This module is copyright in 2002 by Volker Barthelmann.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

8.2 Additional options for this version

This output module provides no additional options.

8.3 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.

8.4 Restrictions

None.

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:

FIXME

9 Amiga output module

This chapter describes the AmigaOS hunk-format output module which can be selected with the ‘-Fhunk’ option.

9.1 Legal

This module is copyright in 2002 by Frank Wille.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

9.2 Additional options for this version

‘-nosym’ Don’t write HUNK_SYMBOL for debugging symbols.

9.3 General

This output module outputs the `hunk` object format, which is a proprietary object file format used by AmigaOS and WarpOS.

9.4 Restrictions

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

It supports the following relocation types:

- absolute, 32-bit
- relative, 8-bit
- relative, 14-bit (mask 0xffff) for PPC branch instructions.
- relative, 16-bit
- relative, 24-bit (mask 0x3ffff) for PPC branch instructions.
- relative, 32-bit
- base-relative, 16-bit
- common symbols are supported as 32-bit absolute and relative references

9.5 Known Problems

Some known problems of this module at the moment:

- None.

9.6 Error Messages

This module has the following error messages:

FIXME

10 m68k cpu module

This chapter documents the backend for the Motorola M68k/CPU32/ColdFire microprocessor family.

10.1 Legal

This module is copyright in 2002 by Frank Wille.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

10.2 Additional options for this module

This module provides the following additional options:

- '-m68000' Generate code for the MC68000 CPU.
- '-m68008' Generate code for the MC68008 CPU.
- '-m68010' Generate code for the MC68010 CPU.
- '-m68020' Generate code for the MC68020 CPU.
- '-m68030' Generate code for the MC68030 CPU.
- '-m68040' Generate code for the MC68040 CPU.
- '-m68060' Generate code for the MC68060 CPU.
- '-mcpu32' Generate code for the CPU32 family (MC6833x etc.).
- '-mcf5200'
Generate code for the MCF5200 ColdFire CPU.
- '-mcf5206'
Generate code for the MCF5206e ColdFire CPU.
- '-mcf5307'
Generate code for the MCF5307 ColdFire CPU.
- '-mcf5407'
Generate code for the MCF5407 ColdFire CPU.
- '-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.

- '-opt-clr'
Enables optimization from `MOVE #0,<ea>` into `CLR <ea>`. Note that `CLR` will execute a read-modify-write cycle on the MC68000.
- '-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-pea'
Enables optimization from `MOVE #x,-(SP)` into `PEA x`. This optimization will leave the flags unmodified, which might not be intended.
- '-opt-st'
Enables optimization from `MOVE.B #-1,<ea>` into `ST <ea>`. This optimization will leave the flags unmodified, which might not be intended.
- '-sdreg=<n>'
Set the small data base register to `An`. `<n>` is valid between 2 and 6.
- '-elfregs'
Register names are preceded by a `'%'` to prevent confusion with symbol names.

10.3 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 target address type is 32bit.

Default alignment for instructions is 2 bytes. Sections will be aligned to 8 bytes by default. The default alignment for data is 2 bytes, when the data size is larger than 8 bits.

10.4 Extensions

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

- `.sdreg <An>`
Equivalent to `near <An>`.
- `cpu32`
Generate code for the CPU32 family.
- `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.
- `machine <cpu_type>`
Makes the assembler generate code for `<cpu_type>`, which can be the following: 68000, 68010, 68020, 68030, 68040, 68060, 68851, 68881, 68882, 5200, 5206, 5307, 5407 and `cpu32`.

mc68000 Generate code for the MC68000 CPU.
 mc68010 Generate code for the MC68010 CPU.
 mc68020 Generate code for the MC68020 CPU.
 mc68030 Generate code for the MC68030 CPU.
 mc68040 Generate code for the MC68040 CPU.
 mc68060 Generate code for the MC68060 CPU.
 mcf5200 Generate code for the MCF5200 ColdFire CPU.
 mcf5206 Generate code for the MCF5206e ColdFire CPU.
 mcf5307 Generate code for the MCF5307 ColdFire CPU.
 mcf5407 Generate code for the MCF5407 ColdFire CPU.

near [**<An>**]

Enables small data (base-relative) mode and sets the base register to **An**. **near** without an argument will reactivate a previously defined small data mode, which might be switched off by a **far** directive.

opt [...] This directive is ignored and is present only for compatibility.

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

<symbol> equ **<Rn>**

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

<symbol> equ **l** **<reglist>**

Equivalent to **<symbol> reg <reglist>**.

<symbol> fequ **<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> fequ **l** **<reglist>**

Equivalent 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**, **fp1ar/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**.

10.5 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>.1 optimized to <exp>.w, when <exp> is absolute and between -32768 and 32767.
- <exp>.w translated to <exp>.1, 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 bd16 is between -32768 and 32767.
- (bd32,An,ZRn) optimized to (d16,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 bd16 is between -32768 and 32767.
- (bd32,PC,ZRn) optimized to (d16,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).
- ([bd16,Rn,...],...) translated to ([bd32,An,...],...), when bd16 is not between -32768 and 32767.

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

This backend performs the following instruction optimizations:

- MOVE.L #x,Dn optimized to MOVEQ #x,Dn, when x is between -128 and 127.
- MOVE.? #0,<ea> optimized to CLR.? <ea>, when allowed by the option `-opt-clr` or a different CPU than the MC68000 was selected.
- MOVE.B #-1,<ea> optimized to ST <ea>, when allowed by the option `-opt-st`.
- MOVE.? #x,-(SP) optimized to PEA x, when allowed by the option `-opt-pea`. The move-size must not be byte (.b).
- 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` and the register list only contains a single register.
- FMOVEM.? <reglist> is deleted, when the register list was empty.
- CLR.L Dn optimized to MOVEQ #0,Dn.
- EORI.? #-1,<ea> optimized to NOT.? <ea>.
- ADD.? #x,<ea> optimized to ADDQ.? #x,<ea>, when x is between 1 and 8.
- SUB.? #x,<ea> optimized to SUBQ.? #x,<ea>, when x is between 1 and 8.
- ADD.? #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.
- ADDA.? #0,An and SUBA.? #0,An will be deleted.
- ADDA.? #x,An optimized to LEA (x,An), (An), when x is between -32768 and 32767.

- SUBA.? #x,An optimized to LEA (-x,An), (An), when x is between -32767 and 32768.
- LEA (0,An),An will be deleted.
- 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.
- 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>).
- 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.
- 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.
- BRA.? <label> translated to JMP <label>, when <label> is not defined in the same section or outside the range from -32768 to 32767 bytes from the current address.
- BSR.? <label> translated to JSR <label>, when <label> is not defined in the same section or outside the range from -32768 to 32767 bytes from the current address.
- B<cc>.? <label> is automatically optimized to 8-bit, 16-bit or 32-bit (68020 up, CPU32, CF5407 only), whatever fits best. When the selected CPU doesn't support 32-bit branches it will try to change the conditional branch into a B!<cc>.? *+8 and JMP <label> sequence.
- <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.

10.6 Known Problems

Some known problems of this module at the moment:

- None?

10.7 Error Messages

This module has the following error messages:

FIXME

11 PowerPC cpu module

This chapter documents the Backend for the PowerPC microprocessor family.

Note that this module is not yet completed and stable!

11.1 Legal

This module is copyright in 2002 by Frank Wille.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

11.2 Additional options for this module

This module provides the following additional options:

- '**-big**' Select big-endian mode.
- '**-no-regnames**'
 Don't predefine any register-name symbols.
- '**-little**' Select little-endian mode.
- '**-mpwrx, -mpwr2**'
 Generate code for the POWER2 family.
- '**-mpwr**' Generate code for the POWER family.
- '**-m601**' Generate code for the 601.
- '**-mppc, -mppc32, -m403, -m603, -m604**'
 Generate code for the 32-bit PowerPC family.
- '**-mppc, -mppc64, -m620**'
 Generate code for the 64-bit PowerPC family.
- '**-mavec**' Generate code for the AltiVec unit.
- '**-mcom**' Allow common PPC instructions.
- '**-many**' Allows any PPC instruction.
- '**-sereg=<n>**'
 Sets small data base register to Rn.
- '**-sd2reg=<n>**'
 Sets the 2nd small data base register to Rn.
- '**-opt-branch**'
 Enables 'optimization' of 16-bit branches into "B<!cc> \$+8 ; B label" sequences when necessary.

11.3 General

This backend accepts PowerPC instructions as described in the instruction set manuals from IBM and Motorola (e.g. the PowerPC Programming Environments).

The target address type is 32bit.

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

11.4 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`.

11.5 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.

11.6 Known Problems

Some known problems of this module at the moment:

- None?

11.7 Error Messages

This module has the following error messages:

FIXME

12 c16x/st10 cpu module

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

12.1 Legal

This module is copyright in 2002 by Volker Barthelmann.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

12.2 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.

12.3 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.

12.4 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
```

12.5 Optimizations

This backend performs the following optimizations:

- `jmp` is translated to `jmp`, 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.

12.6 Known Problems

Some known problems of this module at the moment:

- Lots...

12.7 Error Messages

This module has the following error messages:

FIXME

13 6502 cpu module

This chapter documents the backend for the Rockwell/MOS 650x/651x microprocessor family.

13.1 Legal

This module is copyright in 2002 by Frank Wille.

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

Distributing modified versions and commercial usage needs my written consent.

Certain modules may fall under additional copyrights.

13.2 Additional options for this module

This module provides the following additional options:

`-opt-branch`

Enables 'optimization' of `B<cc>` branches into "`B<!cc> *+3 ; JMP label`" sequences when necessary.

13.3 General

This backend accepts 650x/651x family instructions as described in the instruction set reference manual from Rockwell.

The target address type is 16 bit.

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

13.4 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.

13.5 Optimizations

This backend performs the following operand optimizations:

- Branches where the destination is out of range are translated into `B<!cc> *+3` and an absolute `JMP` instruction.

13.6 Known Problems

Some known problems of this module at the moment:

- None?

13.7 Error Messages

This module has the following error messages:

FIXME

14 Interface

14.1 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.

14.2 Building `vasm`

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

14.2.1 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.

14.2.2 Adapting the Makefile

Before building anything you have to insert correct values for `CC`, `LD_FLAGS` and `RM` in the `'Makefile'`.

- CC** Here you have to insert a command that invokes an ANSI C compiler you want to use to build vasm. It must support '-D', '-I', '-c' and '-o' the same like e.g. `vc` or `gcc`. Additional options should also be inserted here. E.g. if you are compiling for the Amiga with `vbcc` you should add '-DAMIGA'.
- LDFLAGS** Here you have to add options which are necessary for linking. E.g. some compilers need special libraries for floating-point.
- RM** Specify a command to delete a file, e.g. `rm -f`.

An example for the Amiga using `vbcc` would be:

```
CC = vc -DAMIGA -c99
LDFLAGS = -lmieee
RM = delete quiet
```

An example for a typical Unix-installation would be:

```
CC = cc
LDFLAGS = -lm
RM = rm -f
```

Open/Net/Free/Any BSD i386 systems will probably require the following settings:

```
CC = gcc -D_ANSI_SOURCE
LDFLAGS = -lm
RM = rm -f
```

14.2.3 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
```

14.3 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.

14.3.1 Sections

One of the top level structures is 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 module.

atom *first;
atom *last;
    Pointers to the first and last atom of the section. See following sections for
    information on atoms.

int align;
    Alignment of the section in bytes.

int flags;
    Flags of the section. Currently available flags are:
    HAS_SYMBOLS
        At least one symbol is defined in this section.

taddr pc;
    Current offset/program counter in this section. Can be used while traversing
    through section. Has to be updated by a module using it.

size_t idx;
    A member usable by the output module for private purposes.

```

14.3.2 Symbols

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 express-
        ing.

int 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 8
    The symbol is exported to other files.

#define INEVAL 16
    Used internally.

#define COMMON 32
    The symbol is a common symbol.

```

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 offset of a LABSYM symbol in bytes relative to the beginning of the section.

taddr align;
    The alignment of the symbol in bytes.

size_t idx;
    A member usable by the output module for private purposes.

```

14.3.3 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;
    int align;
    int line;
    union {
        instruction *inst;
        dblock *db;
        symbol *label;
    };
};

```

```

    sblock *sb;
    defblock *defb;
} 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.
int align;
    The alignment of this atom.
int line; The source line number that created this atom.
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:

    taddr space;
        The size of the empty/filled space in bytes.

    char fill;
        The fill value.

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.

```

14.3.4 Relocations

DATA-atoms can have a relocations 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 preferable to use standard relocations, if possible. Chanced 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:

```
int offset;
    The offset (from the start of the DATA-atom in bits.
int size; The size of the relocation in bits.
taddr mask;
    A mask value.
symbol *sym;
    The symbol referred by this relocation
FIXME: description of masking etc.
```

14.3.5 Errors

Each module can provide a list of possible error messages contained e.g. in `'syntax_errors.h'` or `'cpu_errors.h'`. They are a comma-separated list of a printf-format string and error flags. Allowed flags are `WARNING`, `ERROR` and `FATAL`. They can be combined using `or` (`|`). Errors cause the assembler to return false. `FATAL` causes the assembler to terminate immediately.

The errors can be emitted using the function `syntax_error(int n, ...)`, `cpu_error(int n, ...)` or `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.

14.4 Syntax modules

A new syntax module must have its own subdirectory under `'vasm/syntax'`. At least the files `'syntax.h'`, `'syntax.c'` and `'syntax_errors.h'` must be written.

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.
int init_syntax();
    Will be called during startup. Must return zero if initializations 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.

```
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.

```
ISIDSTART(x)/ISIDCHAR(x)
```

These macros should return non-zero if and only if the argument is a valid character to start and identifier/inside an identifier respectively. `ISIDCHAR` must be a superset of `ISIDSTART`.

```
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.

```
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.

Have a look at the support functions provided by the frontend to help.

14.5 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.

14.5.1 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`:

```
#define MAX_OPERANDS 3
```

Maximum number of operands of one instruction.

```
#define MAX_QUALIFIERS 0
```

Maximum number of mnemonic-qualifiers per mnemonic.

```
typedef long taddr;
```

Data type to represent a target-address. Preferably use the ones from ‘`stdint.h`’.

```
#define LITTLEENDIAN 1
```

```
#define BIGENDIAN 0
```

Define these according to the target endianness.

```
#define VASM_CPU_<cpu> 1
```

Insert the cpu specifier.

```

#define INST_ALIGN 2
    Minimum instruction alignment.

#define SECTION_ALIGN 2
    Default section 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.

typedef ... operand;
    Structure to store an operand.

typedef ... mnemonic_extension;
    Mnemonic extension.

#define HAVE_INSTRUCTION_EXTENSION 1
    If cpu-specific data should be added to all instruction atoms.

typedef ... instruction_ext;
    Type for the above extension.

```

14.5.2 The file ‘cpu.c’

A cpu module has to provide the following elements (all other functions should be `static` to prevent name clashes) in `cpu.c`:

```

int bitsperbyte;
    The number of bits per byte of the target cpu.

int bytespertaddr;
    The number of bytes per taddr.

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. 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.

void free_operand(operand *);
    Free an operand.

int parse_operand(char *text,int len,operand *out,int requires);
    FIXME

mnemonic mnemonics[];
    FIXME

taddr instruction_size(instruction *,section *,taddr);
    FIXME

dblock *eval_instruction(instruction *,section *,taddr);
    FIXME

dblock *eval_data(operand *,taddr,section *,taddr);
    FIXME

void init_instruction_ext(instruction_ext *);
    (If HAVE_INSTRUCTION_EXTENSION is set.) Initialize an instruction extension.

```

14.6 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 `'Makefile'` has to be extended. The module should be added to the `OBJS` variable at the start of `'Makefile'`. 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-specifics, 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

