vasm assembler system

Volker Barthelmann

Table of Contents

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

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

`vasm` is run using the following syntax:

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

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

'`-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.

'`-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 occurence 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.

'`-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.

'`-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 independant 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.

‘`-w`’
: Hide all warning messages.

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

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. `* / % //` (multiply, divide, modulo)
4. `&` (bitwise and)
5. `^ ~` (bitwise exclusive-or)
6. `| !` (bitwise inclusive-or)
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.

Inside character contants, the following escape sequences are allowed (unless '`-noesc`' was specified):

`\\` 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 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 explicitely 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.

2.5 Macros

Macros are supported by vasm, but the directives for defining them have to be implemented in the syntax module. In any case the assembler core allows up to 9 macro parameters by default (extendable to up to 36 parameters) to be passed in the operand field. They can be referenced inside the macro by `\1` to `\9`. Additionally there is a special argument `\0` which is set to the first qualifier (mnemonic extension) of the macro invocation when given.

A macro parameter which is enclosed inside `<` and `>` characters is treated as a single parameter, even when it contains commas. `>` characters within such a parameter may be specified by `>>`.

`\@` inserts a unique id of the form `_nnnnnn` (where 'n' is a digit between 0 and 9) per macro invocation. Useful for defining reusable labels in a macro.

`\@!` does the same, but pushes the id onto an internal stack.

`\@@` insert the id from top of the id-stack.

`\#` is replaced by the number of parameters given to the current macro invocation.

`\?n` represents the total length of parameter n in bytes. Note that the quotes in a string parameter are included.

2.6 Conditional Assembly

Has to be provided completely by the syntax module.

2.7 Known Problems

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

– None.

2.8 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
- Sebastian Pachuta
- Gunther Nikl
- George Nakos
- Timm S. Mueller
- Gareth Morris
- Dominic Morris
- Mauricio Muñoz Lucero
- Jörg van de Loo
- Robert Leffmann
- Miro Kropacek
- Mikael Kalms
- Matthew Hey
- Tom Duin
- Karoly Balogh

2.9 Error Messages

The frontend has the following error messages:

– 1: illegal operand types
– 2: unknown mnemonic <%s>
– 3: unknown section <%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: division by zero
– 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: rorg is lower than current pc

- 22: character constant too long
- 23: undefined local symbol
- 24: trailing garbage after option -%c
- 25: undefined macro parameter '\%d'
- 26: missing %s directive for macro "%s"
- 27: macro definition inside macro "%s"
- 28: maximum number of %d macro arguments exceeded
- 29: option -%c was specified twice
- 30: read error on <%s>
- 31: expression must be constant
- 32: initialized data in bss
- 33: missing %s 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: macro id stack overflow
- 41: macro id pull without matching push

3 Standard Syntax Module

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

3.1 Legal

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 (`:`). Local labels are preceded by '`.`' and have to contain digits only. Local labels are valid 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 and ARM 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. Note that this directive

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

`.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 highes 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, other- wise to doubleword boundaries.

`.endm`
: Ends a macro definition.

`.endr`
: Ends a repetition block.

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

`.extern <symbol>[,<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>[,<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>...]`
: 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.

`.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.

`.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 occurence.

`.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 occurence.

`.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.

`.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>...]`
: See `.word`.

`.macro <name>`
: Defines a macro which can be referenced by <name>. 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`. Argument `\0` is set to the macro's first qualifier (mnemonic extension), when given. The special argument `\@` inserts a unique id, useful for defining labels.

`.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.

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

`.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

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

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.

`.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.

`.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`.

`.uaquad <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.

`.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>...]`
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.

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: \" expected
- 1009: invalid data operand
- 1010: , expected
- 1011: identifier expected
- 1012: illegal escape sequence \\%c
- 1013: expression must be constant
- 1014: unexpected endm without macro
- 1015: endif without if
- 1016: if without endif
- 1017: maximum if-nesting depth exceeded (%d levels)
- 1018: else without if
- 1019: syntax error
- 1020: unexpected endr without rept
- 1021: symbol <%s> already defined with %s scope

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

4.2 Additional options for this version

This syntax module provides the following additional options:

'`-align`' Enables 16-bit alignment for constant declaration (`dc.?`, except `dc.b`) directives.

'`-devpac`' Devpac-compatibility mode.

−Enables 16-bit alignment for constant declaration (`dc.?`, except `dc.b`) directives.

−Predefines offset symbols `__RS`, `__SO` and `__FO` as `0`, which in vasm are undefined until first referenced.

−Disable escape codes in strings (see '`-noesc`').
−Enable dots within identifiers (see '`-ldots`').
−Up to 35 macro arguments.

'`-ldots`' Allow dots (`.`) within all identifiers.

'`-localu`' Local symbols are introduced by `'_'` instead of `'.'`. For Devpac compatbility, which offers a similar option.

'`-phxass`' PhxAss-compatibilty mode. Enables the following features:

−`section <name>` starts a code section named `<name>` instead of a section which also has the type `<name>`.

−Macro names are treated as case-insensitive.
−Up to 35 macro arguments.
−Defines the symbol `_PHXASS_`.

'`-spaces`' Allow blanks in operands.

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 has to be separated from the label by whitespace (not required in any case, e.g. with `=`). 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 (`,`).

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` are never possible.

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.

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

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

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 (provided the CPU- and output-module support them):

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

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

`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`.

`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.

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

`code_c`
: Equivalent to `section code_c,code,chip`.

`code_f`
: Equivalent to `section code_f,code,fast`.

`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.

`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`.

`echo <string>`
: Prints <string> to stdout.

`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.

`erem`
: Ends an outcommented block. Assembly will continue.

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

`fail <message>`
: Immediately break assembly with a fatal error, showing the <message> from the operand field.

`<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 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.

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

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

`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>.

`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 occurence.

`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 occurence.

`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`. Argument `\0` is set to the macro's first qualifier (mnemonic extension), when given. In Devpac- and PhxAss-compatibility mode up to 35 arguments are accepted, where argument 10-35 can be referenced by `\a` to `\z`. The special argument `\@` inserts a unique id, useful for defining labels. `\#` is replaced by the number of arguments (also stored in `NARG`) and `\?n` is replaced by the length of argument `n`. `\.` selects the argument indexed by the current value of `CARG`. `\+` and `\-` do the same, but additionally post-increment and post-decrement `CARG`.

`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 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.

`odd`
: Aligns to an odd address. Equivalent to `cnop 1,2`.

`org <expression>`
: Sets the base address for the subsequent code.

`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>`
: Prints `<string>` to stdout.

`printv <expression>`
: Evaluate `<expression>` and print it to stdout out in decimal and hexadecimal form.

`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.

`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`. <sec_type> defaults to `code` in Phxass mode. Otherwise a single argument will start a section with the type and name of <sec_type>. 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. Optionally it is also possible to attach the suffix `_C`, `_F` or `_P` to the <sec_type> argument for defining the memory type.

`<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 <expresion>. See `fo` directive.

`setso <expression>`
Sets the structure offset counter to <expresion>. 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>`.

`xdef <symbol>[,<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>[,<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 oprands
- 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: expression must be a constant
- 1014: illegal section type
- 1015: repeatedly defined symbol
- 1016: illegal memory type
- 1017: unexpected %s without %s
- 1020: maximum if-nesting depth exceeded (%d levels)
- 1022: missing %c

5 Oldstyle Syntax Module

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

5.1 Legal

5.2 Additional options for this version

This syntax module provides the following additional options:

'`-dotdir`' Directives have to be preceded by a dot (`.`).

'`-autoexp`'
: Automatically export all non-local symbols, making them visible to other modules during linking.

5.3 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. `=`).

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.

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.

Comments are introduced by the comment character `;`. The rest of the line will be ignored.

Example:

```
mylabel instr op1,op2 ;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` (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`). Also C-style prefixes are supported for hexadecimal (`0x`) and binary (`0b`). All other numbers starting with a digit are decimal, e.g. `1239`.

5.4 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>`.

`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`.

`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 occurence.

`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 endianess of the target CPU.

`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>...]`.

`dfb <exp1>[,<exp2>,"<string1>"...]`
: Equivalent to `byte <exp1>[,<exp2>,"<string1>"...]`.

`dfw <exp1>[,<exp2>...]`
: Equivalent to `word <exp1>[,<exp2>...]`.

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

`ds <exp>[,<fill>]`
: Equivalent to `blk <exp>[,<fill>]`.

`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.

`el` 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.

`<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`.

`fill <exp>`
: Equivalent to `blk <exp>,0`.

`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.

`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.

`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 occurence.

`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 occurence.

`mac <name>`
: Equivalent to `macro <name>`.

`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>`
: 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. 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`. Argument `\0` is set to the macro's first qualifier (mnemonic extension), when given. The special argument `\@` inserts a unique id, useful for defining labels.

`mdat <file>`
: Equivalent to `incbin <file>`.

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

`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

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

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

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

`text "string"`
Places a single string constant operands into successive bytes of memory in the current section.

`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.

`wor <exp1>[,<exp2>...]`
Equivalent to `word <exp1>[,<exp2>...]`.

`wrd` Increases the program counter by two. Equivalent to `blkw 1,0`.

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

`xdef <symbol>[,<symbol>...]`
See `global`.

`xlib <symbol>[,<symbol>...]`
: See `global`.

`xref <symbol>[,<symbol>...]`
: See `global`.

5.5 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.

5.6 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: expression must be a constant
- 1014: repeatedly defined symbol
- 1015: endif without if
- 1016: if without endif
- 1017: maximum if-nesting depth exceeded (%d levels)
- 1018: else without if
- 1019: unexpected endr without macro
- 1020: unexpected endr without rept
- 1021: cannot open binary file "%s"
- 1022: symbol <%s> already defined with %s scope

6 Simple binary output module

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

6.1 Legal

This module is copyright in 2002,2008 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

'`-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.

6.3 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.

6.4 Known Problems

Some known problems of this module at the moment:

- None.

6.5 Error Messages

This module has the following error messages:

- 3001: sections must not overlap
- 3007: undefined symbol <%s>

7 Test output module

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

7.1 Legal

7.2 Additional options for this version

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 copyright in 2002-2011 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.

8.2 Additional options for this version

None.

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

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

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 copyright in 2008-2010 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

'`-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 suppported
- 3008: output module doesn't allow multiple sections of the same type

10 TOS output module

This chapter describes the TOS output module which can be selected with the '`-Ftos`' option.

10.1 Legal

This module is copyright in 2009-2010 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 version

'`-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.

10.4 Restrictions

– 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.

Those are restrictions of the output format, not of 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 <attr> not suppported
– 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

11 vobj output module

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

11.1 Legal

This module is copyright in 2002-2011 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.

11.2 Additional options for this version

None.

11.3 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.

11.4 Restrictions

None.

11.5 Known Problems

Some known problems of this module at the moment:

- None.

11.6 Error Messages

This module has the following error messages:

- None.

12 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.

12.1 Legal

This module is copyright in 2002-2010 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.

12.2 Additional options for this version

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.

12.3 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.

12.4 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 0xfffc) for PPC branch instructions.
- relative, 16-bit
- relative, 24-bit (mask 0x3ffffc) 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.

12.5 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.

12.6 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 suppported
- 3005: reloc type <m>, size <n>, mask <mask> (symbol <sym> + <offset>) not supported
- 3006: reloc type <n> not supported

13 m68k cpu module

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

13.1 Legal

13.2 Additional options for this module

This module provides the following additional options:

13.2.1 CPU selections

‘`-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.

‘`-m68020up`’
Generate code for the MC68020-68060 CPU. Be careful with instructions like PFLUSHA, which exist on 68030 and 68040/060 with a different opcode (vasm will use the 040/060 version).

‘`-mcpu32`’ Generate code for the CPU32 family (MC6833x, MC6834x, etc.).

‘`-mcf5...`’
‘`-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.

‘`-mcfv2`’ Generate code for the V2 ColdFire core. This option selects ISA_A (no hardware 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.

13.2.2 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>`. Note that `CLR` will execute a read-modify-write cycle on the MC68000.

'`-opt-fconst`'
Floating point constants are loaded with the lowest precision possible. This means that `FMOVE.D #1.0,FP0` would be optimized to `FMOVE.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`, `FSGLDIV`) 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-lsl`'
Allows optimization of `LSL` into `ADD`. This optimization 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 multplication 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-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. 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`.

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

'`-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).
−Many instruction optimizations which are safe.

13.2.3 Other options

'`-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.

'`-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.

'`-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.

'`-phxass`'
: PhxAss-compatibilty 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 '`-opt-allbra`'.

'`-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. Other options are the same as vasm's defaults. The symbol `__G2` is defined, which contains information about the selected cpu type.

13.3 General

This backend accepts M68k and CPU32 instructions as described in Mototola'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.l,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. 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.

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.

13.4 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`, `68851`, `68881`, `68882`, `cpu32`. And various ColdFire CPUs, starting with `5...`.

`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.

`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 reactivate a previously defined small data mode, which might 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 (which is discouraged, so there is no detailed description here). The 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-comptability 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. Default is off in Devpac-comptability mode, otherwise on.
- `o` Enable all optimizations (o1 to o12), or disable all optimizations. Default is that all are disabled in Devpac-compatibility mode and enabled otherwise. When running in native vasm mode this option will also control the following safe vasm-specific optimizations (see below): `og`, `of`, `oj`.
- `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`.

`o12` Optimize `<op>.l #x,An` to `<op>.w #x,An`.

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

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

`s` Include symbols in listing file. Default is on.

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

`w` Show assembler warnings. Default is on.

`x` Only include xdefs in the output file (no symbols). This flag has always the inverted state of `d`.

Also the following Devpac3-style options are supported:

`autopc` Corresponds to `a+`.

`case` Corresponds to `c+`.

`chkpc` Corresponds to `p+`.

`debug` Corresponds to `d+`.

`symtab` Corresponds to `s+`.

`type` Corresponds to `t+`.

`warn` Corresponds to `w+`.

`xdebug` Corresponds to `x+`.

`noautopc` Corresponds to `a-`.

`nocase` Corresponds to `c-`.

`nochkpc` Corresponds to `p-`.

`nodebug` Corresponds to `d-`.

`nosymtab` Corresponds to `s-`.

`notype` Corresponds to `t-`.

`nowarn` Corresponds to `w-`.

`noxdebug` Corresponds to `x-`.

`p=<type>[/<type>]`
Sets the CPU type to any model vasm supports (original Devpac3 only allowed 68000-68040, 68332, 68881, 68882 and 68851).

The following options are vasm specific and should not be used when writing portable source. Using `opt o+` in Devpac mode does not enable any of these options.

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

`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 optimizations (refer to `-opt-brajmp`).

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

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

`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. 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`.

13.5 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 `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 32768.
- `([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.

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:

- `<op>.L #x,An` optimized to `<op>.W #x,An`, when `x` is between -32768 and 32767.
- `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.

- `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`).
- `MVZ.? #x,Dn` and `MVS.? #x,Dn` are optimized to `MOVEQ #x,Dn`.
- `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 option `-opt-movem`.
- `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.
- `ASL.? #1,Dn` optimized to `ADD.? Dn,Dn` for 68000 and 68010.
- `LSL.? #1,Dn` optimized to `ADD.? Dn,Dn` for 68000 and 68010, when option `-opt-lsl` is given.
- `ASL.? #2,Dn` optimized into a sequence of two `ADD.? Dn,Dn` for 68000 and 68010, when the operation size is either byte or word.
- `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 option `-opt-lsl` is given.
- `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`).
- `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`).
- `MULS.W #-2..-256,Dn` optimized to the sequence `EXT.L Dn`, `ASL.L #x,Dn` and `NEG.L Dn` (`-opt-mul`).
- `MULS.L #2..256,Dn` optimized to `ASL.L #x,Dn` (`-opt-mul`).
- `MULS.L #-2..-256,Dn` optimized to the sequence codeASL.L #x,Dn and `NEG.L Dn` (`-opt-mul`).
- `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`).
- `MULU.L #2..256,Dn` optimized to `LSL.L #x,Dn` (`-opt-mul`).
- `FxDIV.? #m,FPn` optimized to `FxMUL.? #1/m,FPn` when m is a power of 2 and option `-opt-fconst` is given.
- `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.
- `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.
- `LINK An,#0` optimized into a combination of `PEA (An)` and `MOVE.L A7,An`.
- `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 (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.
- `BSR <label>` translated to `JSR <label>`, when <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.
- `B<cc> <label>` translated into a combination of `B!<cc> *+8` and `JMP <label>`, when <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.
- `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 `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.

13.6 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.

13.7 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 outside parentheses ignored
– 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: illegal relocation
– 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
- 2051: 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

14 PowerPC cpu module

This chapter documents the Backend for the PowerPC microprocessor family.

14.1 Legal

This module is written in 2002-2006,2008,2011 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.

14.2 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.

'`-sdreg=<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.

14.3 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.

14.4 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`.

14.5 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.

14.6 Known Problems

Some known problems of this module at the moment:

- None?

14.7 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: illegal relocation
- 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

15 c16x/st10 cpu module

This chapter documents the Backend for the c16x/st10 microcontroller family.

Note that this module is not yet fully completed!

15.1 Legal

15.2 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 `jmpr` instruction is too large, it will not be translated to `jmps` but an error will be emitted.

  Also, `jmpa` will not be optimized to `jmpr`.

  The pseudo-instruction `jmp` will still be translated.

'`-jmpa`'
: A `jmp` or `jmpr` 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.

15.3 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.

15.4 Extensions

This backend provides the following specific extensions:

– There is a pseudo instruction `jmp` that will be translated either to a `jmpr` or `jmpa` instruction, depending on the offset.

– The `sfr` pseudo opcode can be used to declare special function registers. It has two, three of 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
```

15.5 Optimizations

This backend performs the following optimizations:

– `jmp` is translated to `jmpr`, if possible. Also, if '`-no-translations`' was not specified, `jmpr` 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.

15.6 Known Problems

Some known problems of this module at the moment:

– Lots...

15.7 Error Messages

This module has the following error messages:

FIXME

16 6502 cpu module

This chapter documents the backend for the MOS/Rockwell 6502 microprocessor family.

16.1 Legal

16.2 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.

'`-illegal`'
: Allow 'illegal' 6502 instructions to be recognized.

'`-dtv`'
: Recognize the three additional C64-DTV instructions.

16.3 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, 65C02, 65CE02, 65C102, 65C112, 6503, 6504, 6505, 6507, 6508, 6509, 6510, 6511, 65F11, 6512 - 6518, 65C00/21, 65C29, 6570, 6571, 6280, 6702, 740, 7501, 8500, 8502, 65802, 65816.

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.

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

16.4 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.

16.5 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.

16.6 Known Problems

Some known problems of this module at the moment:

– None?

16.7 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 relocation

17 ARM cpu module

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

17.1 Legal

17.2 Additional options for this module

This module provides the following additional options:

‘`-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.

'`-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.

'`-little`' Output little-endian code and data (default).

'`-big`' Output big-endian code and data.

'`-thumb`' Start assembling in Thumb mode.

'`-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`.

'`-opt-adr`'
The `ADR` directive will be automatically converted into `ADRL` if required (which inserts an additional `ADD`/`SUB` to calculate an address).

17.3 General

This backend accepts ARM instructions as described in various ARM CPU data sheets. Additionally 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.

17.4 Extensions

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

`.arm` Generate 32-bit ARM code.

`.thumb` Generate 16-bit THUMB code.

17.5 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&0xfff]`, 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).

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.

17.6 Known Problems

Some known problems of this module at the moment:

- Only instruction sets up to ARM architecture V4t are supported.

17.7 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 exptected
- 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\n
- 2026: %d-bit unsigned immediate offset out of range (%ld)
- 2027: offset has to be a multiple of %d
- 2028: instruction at unaligned address

18 80x86 cpu module

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

18.1 Legal

This module is written in 2005-2006,2011 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.

18.2 Additional options for this module

This module provides the following additional options:

‘`-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).

‘`-debug=<n>`’
Enables debugging output.

18.3 General

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

The target address type is 32bit.

Default alignment for sections is 4 bytes. 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. Also register names have to be prefixed by a '%'. Operation size is indicated by a 'b', 'w', 'l', etc. suffix behind the mnemonic.

18.4 Extensions

Predefined register symbols in this backend:

−8-bit registers: `al cl 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 ebp 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:

`.code16` Sets the assembler into 16-bit addressing mode.

`.code32` Sets the assembler into 32-bit addressing mode, which is the default.

`.code64` Sets the assembler into 64-bit addressing mode.

18.5 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.

18.6 Known Problems

Some known problems of this module at the moment:

- 64-bit operations are incomplete and experimental.

18.7 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
- 2014: illegal relocation
- 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

19 z80 cpu module

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

19.1 Legal

This module is copyright in 2009 by Dominic Morris

19.2 Additional options for this module

This module provides the following additional options:

'`-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.

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

'`-hd64180`'
: Turns on 64180 mode supporting additional 64180 opcodes.

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

'`-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.

'`-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.

19.3 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.

19.4 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.

19.5 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.

19.6 Known Problems

Some known problems of this module at the moment:

- Not all RCM4000 opcodes are supported (`llcall`, `lljp` are not available).

19.7 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 (-rcmemu)
- 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

20 Interface

20.1 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.

20.2 Building vasm

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

20.2.1 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.

20.2.2 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. E.g. if you are compiling for the Amiga with `vbcc` you should add '`-DAMIGA`'.

`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`.

20.2.3 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=arm`
- `CPU=c16x`
- `CPU=m68k`
- `CPU=ppc`
- `CPU=test`
- `CPU=x86`
- `CPU=z80`

The following syntax modules can be selected:

- `SYNTAX=std`
- `SYNTAX=mot`
- `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 explicitely with `make`'s '`-f`' option:

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

20.3 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.

20.3.1 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.

`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.

`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.

`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.

`int cond_level;`
: Current level of conditional nesting while calling this macro. The level is provided by the syntax module through `execute_macro()`. The syntax module may use this information to restore the last valid level when exiting a macro in the middle.

`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. Parameter 0 is usually reserved for a special purpose, like an extension.

`int param_len[MAXMACPARAMS];`
: Number of characters per macro parameter.

`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.

`char *linebuf;`
: A `MAXLINELENGTH` 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.

20.3.2 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, documentation 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.

`taddr org;`
: Start address of a section. Usually zero.

`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. Is set to `org` at the beginning.

`uint32_t idx;`
: A member usable by the output module for private purposes.

20.3.3 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 expressing.

`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 symbl defines a section.

`#define TYPE_FILE 4`
: The symbol defines a file.

`#define EXPORT (1<<3)`
: The symbol is exported to other files.

`#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 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_O (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.

`uint32_t idx;`
: A member usable by the output module for private purposes.

20.3.4 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;
  source *src;
  int line;
  listing *list;
  union {
    instruction *inst;
    dblock *db;
    symbol *label;
    sblock *sb;
    defblock *defb;
    void *opts;
    int srcline;
    char *ptext;
    expr *pexpr;
  } 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 RORG 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.

`taddr align;`
: The alignment of this atom.

`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 instruction.

  `operand *op[MAX_OPERANDS];`
  : (If `MAX_OPERANDS!=0`.) The cpu-specific operands of this instruction.

  `instruction_ext ext;`
  : (If the cpu module defines `HAVE_INSTRUCTION_EXTENSION`.) A cpu-module-specific structure. Typically used to store appropriate opcodes, 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.

  `expr *space_exp;`
  : The above size as an expression, which will be evaluated during assembly and copied to `space` in the final pass.

  `int 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.

`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.

`expr *pexpr;`
: (In union `content`.) An expression to evaluate and print to stdout in case of a `PRINTEXPR`-atom.

`expr *roffs;`
: (In union `content`.) The expression holds the relative section offset to align to in case of a `RORG`-atom.

20.3.5 Relocations

`DATA` and `SPACE` 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 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:

`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.

`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 performed when performing a relocation:

1. Extract the <size> bits from the data atom, starting with bit number <offset>. <offset> zero means to start from the first bit (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 (possibly offset by a target specific value) 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. Shift the value above right as many bit positions as there are low order zero bits in <mask>.
5. Add this value to the value extracted in step 1.
6. Insert the low order <size> bits of this value into the data atom starting with bit <offset>.

20.3.6 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`, `FATAL` and `NOLINE`. They can be combined using or (`|`). `NOLINE` has to be set for error messages during initialiation or while writing the output, when no source text is available. 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.

20.4 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.

20.4.1 The file '`syntax.h`'

`#define 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`.

`#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 `e`. Default is to return `e`. The function is defined as `char *chkidend(char *startpos,char *endpos)`.

`#define NARGSYM "NARG"`
: Defines the name of an optional symbol which contains the number of arguments in a macro.

`#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.

20.4.2 The file 'syntax.c'

A syntax module has to provide the following elements (all other funtions should be `static` to prevent name clashes):

`char *syntax_copyright;`
: A string that will be emitted as part of the copyright message.

`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. 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.

`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.

`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 *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.

20.5 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.

20.5.1 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 CPU_CHECKS_OPCNT 0`
: When non-zero, `parse_operand()` is called with an arbitrary number of operands, and it is the task of the cpu module to check it.

`#define MAX_QUALIFIERS 0`
: Maximum number of mnemonic-qualifiers per mnemonic.

`typedef long taddr;`
: Data type to represent a target-address. Preferrably use the ones from '`stdint.h`'.

`#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 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. 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:

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

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

`NEED_CLEARED_OPERANDS`
: 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 `')'`.

`CHECK_ATOMSIZE`
: Usually vasm will start another assembler pass when a label changed its value. With complex optimizers (e.g. M68k) it can happen that the effect from optimizations and translations of instructions is nullified, so the labels do not change. `CHECK_ATOMSIZE` will remember the sizes of all atoms from the previous pass and check them for changes, which is much safer but requires more resources.

Implementing additional target-specific unary operations is done by defining the following optional macros:

`EXT_UNARY_NAME(s)`
: Should return True when the string in `s` points to an operation name we want to handle.

`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`.

`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).

`EXT_FIND_BASE(e,s,p)`
: Defines a function with the arguments `(expr *e, section *s, taddr p)` to return a pointer to the base symbol of expression `e`. The type in `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.

20.5.2 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 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);`
: 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.

`mnemonic mnemonics[];`
: The mnemonic table is usually defined in '`opcodes.h`' and 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`.

`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.

20.6 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-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