Cfasm ColdFire Assembler

rev. 2.64

Contents

1. General

The ColdFire Assembler, cfasm, is a freeware assembler for ColdFire processors.

Instruction set supported for the following processors: 52xx, 5307 and 5407.

Command line arguments specify the filenames to assemble. Only one object file is output even though several files can be specified and assembled together. The assembler can output in Motorola SREC, raw binary or NuOS object code formats.

The object file is placed in the file <inputfilename>.o' where 'inputfilename' was the name of the actual assembler file (missing any extensions). The listing and error messages are written to the standard output.

The listing file contains the address and bytes assembled for each line of input followed by the original input line (unchanged, but moved over to the right some). If an input line causes more than 6 bytes to be output (e.g. a long dc.b directive), additional bytes are listed on succeeding lines with no address preceding them.

Equates cause the value of the expression to replace the address field in the listing. Equates that have forward references cause phasing Errors in Pass 2.

It is unwise to have more than one assembly in progress per directory since the object file would be the same for all assemblies running.

1.1 Expressions

Expressions may consist of symbols, constants or the character '*' (denoting the current value of the program counter) joined together by one of the operators: -

```
+    add
-    subtract
*    multiply
/    divide
%    modulo (remainder after division)
&    bitwise and
|    bitwise or
^    bitwise exclusive-or
<<   bitwise left shift
>>   bitwise right shift
~    ones complement
```

Expressions are evaluated left to right and there is no provision for parenthesized expressions. Arithmetic is carried out in signed 32-bit twos-complement integer precision.

1.2 Constants

Constants are constructed as follows:

0x	followed by a hexadecimal constant
$	followed by a hexadecimal constant
@	followed by an octal constant
%	followed by a binary constant
digit	decimal or floating-point constant

String constants are specified by enclosing the string in single or double quotes. Strings are recognized by the **dc** (define constants) and **equ** (equate) pseudo ops. For string equates, the value returned depends on the length of the string. Since a label is a 32-bit reference (four bytes), a string equate will convert up to the last four characters of a string. As an example:-

```
filesystem    equ  “nufs”        ; filesystem = 0x6e756673
test          equ  “hello0000”   ; test = 0x30303030
alpha_a       equ  “a”           ; alpha_a = 0x00000061
```

Floating-point numbers can be used with **equ** (equates) and immediate (#) addressing modes in addition to **dc** (define constants). Although not all ColdFire processors contain floating-point hardware, values can be manipulated in software. As an example:-

```
pi_equ        equ.s 3.1415926
pi_single     dc.s 3.1415926
pi_double     dc.d 3.1415926

    move.l   #3.1415926,d0          ; single precision
    move.l   #pi_equ,d0             ; single precision
    fmove.s  (pi_single,pc),fp0     ; single precision
    fmove.d  (pi_double,pc),fp1     ; double precision
```

1.3 Errors

Error diagnostics are placed in the listing file just before the line containing the error.

Format of the error line is:

```
Line_number: Module: Description of error
                    or
Line_number: Warning — Description of error
```

Errors of the first type in pass one cause cancellation of pass two. Warnings do not cause cancellation of pass two but should cause you to wonder where they came from.

Error messages are meant to be self-explanatory. If more than one file is being assembled, the file name precedes the error:

```
File_name,Line_number: Module: Description of error
```

Finally, some errors are classed as fatal and cause an immediate termination of the assembly. Generally these errors occur when a temporary file cannot be created or is lost during the assembly.

Notes:

Phasing error	This error usually occurs because of a difference between the program counter from assembly phase(pass) 1 and phase 2.

As a general rule, try to state all equate definitions before your code section.

1.4 Pseudo Ops

All pseudo-ops are treated as mnemonics, and must be preceded by at least one white space character in the assembly file. Some require a symbol preceeding the operation on the same line.

org *<value>* Origin of code in memory (also checks for word alignment).

align *<value>* Align the program counter to the boundary specified (e.g. align 4 would align to a longword boundary).

equ *(<extension>) <expression>*
Equates. An expression is evaluated and assigned to a symbol. The only extension allowed is ".s" for single precision floating-point equates. No extension defaults to longword. String constants are allowed.

dc *<extension> <value>, ...*
Define constant(s). Extensions may be one of the following:-
.b = byte (1 byte)
.w = word (2 bytes)
.l = longword (4 bytes)
.s = single precision floating-point (4 bytes)
.d = double precision floating-point (8 bytes)

dr *<extension> <value>, ...*
Define relative(s) (usually used for library offset tables). Extensions can be .b, .w or .l types.

ds *<extension> <value>(,<data>)*
Define storage. Extensions can be .b, .w or .l types. Total storage size (in bytes) equals extension size multiplied by value. The storage area will be filled with zero, or the data value if supplied.

opt *<options>* Options. (see section 1.4.2).

cpu *<processor>* ColdFire processor selection(5102, 5202, 5204, 5206, 5206e, 5249, 5272, 5280, 5282, 5307, 5407, any). Certain restrictions are enforced when a processor is selected. Defaults to 5206. A value of "any" will select all processor capabilities.

section *<name>,(<align>,<type>)*
Name can be a unique string for each section. The assembler will merge code or data for each named section. Reserved names such as code, data, bss, basedata or basebss can be used to automatically assemble to those sections. Use optional align for boundary alignment. Type can be C(code), D(data).
Sections basedata and basebss are used with base register

addressing. Only labels in these sections will be allowed with indexed addressing. When producing object files, a new block is created containing the data for base register indexing and can be loaded anywhere into memory. The linker will usually take care of this however. To simply usage of these sections, a special xdef is produced by the linker, called "_base". For example:-

```
xref  _base

lea.l _base,a4  ; indexed addressing register a4.
```

idnt <*name*>,<*version*>,<*date*>
Identification strings. Each string will cause the generation of individual NuOS blocks of type BLOCK_NAME, BLOCK_VERSION and BLOCK_DATE (only if object code option enabled).

xref <*symbol*> External symbol reference. The symbol exists in another file.

xdef <*symbol*> External symbol definition. This symbol can be referenced from another file.

include <*file*> Filename of include file to be processed.

incdir <*directory*> Include directory. Include files are first searched in the current directory, then in the incdir directory, and finally the environment variable cfasmincl (if it exists).

incbin <*file*> Include a binary file into the assembly. This file is not assembled, but added as if it were raw data. Useful for tables, graphic images etc...

lvoreset Reset library vector offset counter. An internal counter value is set to -6 (which is the code size of "jmp <longaddress>" used by library vectors). Note: lvorst can also be used.

lvo <*symbol*> Library vector offset entry. A symbol is created by prefixing the string "_lvo" to the symbol name passed and an internal counter value assigned to it. This counter value is then decremented by 6 (which is the code size of "jmp <longaddress>" used by library vectors).

call <*symbol*> Call a library function using _lvo values, register a6 points to a library base. The code generated is:-

```
jsr    (_lvo<symbol>,a6)
```

callres <*symbol*>,<*base*> Call a library function using _lvo values from a base offset, register a6 points to a library base. The code generated is:-

```
move.l a6,-(a7)
move.l (<base>,a6),a6
jsr    (_lvo<symbol>,a6)
move.l (a7)+,a6
```

callusr <*symbol*>,<*base*> Call a library function using _lvo values from a base offset, register a4 points to a user base. The code generated is:-

```
move.l a6,-(a7)
move.l (<base>,a4),a6
jsr    (_lvo<symbol>,a6)
move.l (a7)+,a6
```

push *{reglist}* Performs register pushes to the stack. If only one register is passed, then this operation performs a simple move.l {register}, -(a7).

```
lea.l   (-size,a7),a7
movem.l {reglist},(a7)
(where size is the number of registers*4)
```

pull *{reglist}* Performs register pulls from the stack. If only one register is passed, then this operation performs a simple move.l (a7)+,{register}

```
movem.l (a7),{reglist}
lea.l   (size,a7),a7
(where size is the number of registers*4)
```

structure <*symbol*>,<*value*>
Creates a new symbol set to value. An internal variable is also set to the value.

stkstruct <*symbol*>,<*value*>
Creates a new symbol set to value. An internal variable is also set to the value. The stack structure defines all further components to

struct <*symbol*>,<*value*>
Creates a new symbol set to current internal value and the value is then added to the internal.

byte <*symbol*> Creates a new symbol set to current internal value and increments the internal value by one except if defined as a stack structure in which case the internal value is decremented first by one and then set to the symbol.

word <*symbol*> Creates a new symbol set to current internal value and increments the internal value by two except if defined as a stack structure in which case the internal value is decremented first by two and then set to the symbol.

long <*symbol*> Creates a new symbol set to current internal value and increments the internal value by four except if defined as a stack structure in which case the internal value is decremented first by four and then set to the symbol.

label <*symbol*> The symbol is set to the current internal value.

aptr <*symbol*> Absolute pointer, same as long.

rptr <*symbol*> Relative pointer, same as long.

ulong <*symbol*> Unsigned long, same as long.

fixed <*symbol*> Fixed point value, same as long.

float <*symbol*> Floating point value, same as long.

short <*symbol*> Short integer, same as word.

ushort <*symbol*> Unsigned short, same as word.

uword <*symbol*> Unsigned word, same as word.

char <*symbol*> Character, same as byte.

ubyte <*symbol*> Unsigned byte, same as byte.

bitdef <*prefix*>,<*symbol*>,<*bit*>

Creates two symbols based on the prefix. The first is a bit-symbol and the second is a fieldmask-symbol. The symbols are created by taking the prefix, concatenating a “b” (for bit) and an “f” (for field mask), then adding the passed symbol name. The bit-symbol can be used for bit-wise operations (btst, bset, bclr etc...) and the fieldmask-symbol can be used with logical operations (and, or etc...)

The bit-symbol value is equal to the passed <bit> value and the fieldmask-symbol is equal to 1 left-shifted by the <bit> value.

Example 1: bitdef pa,status,2

produces the equivalent of two defined symbols:-

```
pab_status          equ  2      ; = bit
paf_status          equ  4      ; = 1<<bit
```

Example 2: bitdef pre,data,7

produces the equivalent of two defined symbols:-

```
preb_data           equ  7      ; = bit
pref_data           equ  128    ; = 1<<bit
```

classtagbegin *(<symbol>)*

Start of classtag arrayitems. A symbol called ct_<symbol> will be created and set to the number of items in the array. This will occur during the second assembler pass, since further items will not be known during pass one. A longword is emitted containing the number of items in the array, unless the symbol is not supplied. In this case, the classtag starts at zero. This can be used in include files to only create reference symbols.

classtag <*symbol>, <value>*

Use this to set each tag value. Each item in the array is set to the value supplied. A symbol called ct_<symbol> is created and set to the index within the array. If this is the first item, it will be set to zero. A longword containing <value> will be emitted only if a symbol was suppied in classtagbegin. The programmer can use the ct_ symbols for passing to routines that handle classtag arrays. Each symbol value is incremented by four (4).

classtagend *(<symbol>)*

This MUST be set to the same symbol as classtagbegin.

enumreset This resets the internal enumeration variable to zero.

enumset <*value*> This sets the internal enumeration variable to the value supplied.

enum *(<extension>,)<symbol>*

Enumeration provides a convient method of symbol definition without having to provide equate statements. Without extensions, the enumeration increment is one (1), however you can change this by using word (2) or long (4).

```
enumreset
enum     command_null          ; = 0
enum     command_reset         ; = 1
enum     command_read          ; = 2  etc...
```

boundary This can be used to place a special segment between the boundary of code and data. Since the ColdFire processor prefetches instructions (when the cache is enabled), it may attempt to access locations that could cause an error under certain circumstances. To solve this, the boundary operation will place the following segment between code and data:-

```
.label   bra.b .label
```

```
        illegal
        illegal
        illegal
        illegal
        illegal
        illegal
```

In addition to this segment, the program counter will be longword aligned.

end The assembly ends when there is no more input.

pag[e] Sends form feed to output.

1.4 Pseudo Ops (continued)

1.4.1 Conditional

endc	Terminate a conditional assembly block.
else	Assemble code based on previous condition statement.
ifd <*symbol*>	If symbol defined, then assemble the next block of code until an **endc** pseudo-op is encountered.
ifnd <*symbol*>	If symbol not defined, then assemble the next block of code until an **endc** pseudo-op is encountered.
ifeq <*expr*>	If expression evalutes to zero, then assemble next code block.
ifne <*expr*>	If expression evalutes to non-zero, then assemble next code block.
ifge <*expr*>	If expression evalutes to >=0, then assemble next code block.
ifgt <*expr*>	If expression evalutes to >0, then assemble next code block.
ifle <*expr*>	If expression evalutes to <=0, then assemble next code block.
iflt <*expr*>	If expression evalutes to <0, then assemble next code block.

1.4.2 OPT (options)

These options follow an opt pseudo-op. For example:- opt list.

list	Turn on output listing.
nolist	Turn off output listing (default).
object	Produce object code output (NuOS format).
binary	Produce binary output (overridden by object flag, raw format).
rtxnop	Adds NOP instruction after RTS or RTE instructions.
optimise=<?>	Enables/Disables various optimisations.
optimise=all	Enables all optimisations (default).
optimise=none	Disables all optimisations.
optimise=moveq	Attempts to convert move.l #<data>,Dx values to moveq.l #<data>,Dx (values -128 to 127).
optimise=addq	Attempts to convert add.l #<data>,<ea>x values to addq.l #<data>,<ea>x (values 1 to 8).
optimise=subq	Attempts to convert sub.l #<data>,<ea>x values to subq.l #<data>,<ea>x (values 1 to 8).
optimise=lea	Attempts to convert lea.l <expr>.l,ax values to lea.l <expr>.w,ax. If expr is zero, then converts to sub.l ax,ax.
optimise=branch	Attempts to improve backward branches from word to byte offsets.
optimise=index	Attempts to convert (0,ax) to (ax).
optimise=nomoveq	Disables moveq optimsation.
optimise=noaddq	Disables addq optimsation.
optimise=nosubq	Disables subq optimsation.
optimise=nolea	Disables lea optimsation.
optimise=nobranch	Disables branch optimsation.
optimise=noindex	Disables index optimsation.

unsizemode Allows cpu specific unsized opcode mode. By default, unsized opcodes will produce code that is compatible with the V2 core. For example, an unsized compare instruction will use longword sizing to remain compatible with V2. Word sizing on unsized opcodes will therefore be used wherever possible when unsizemode is enabled. For a V4 core and unsizemode, an unsized compare instruction will be treated as word sized. Unsized move instructions default to word sizing on all cores.

Old pseudo-ops (recognized, but ignored):-

ttl use pr' to get headings and page numbers

spc use blank line instead.

nam[e]

1.4.3 CPU (processor selection)

CPU	Core[1]	Multiply Accumulate[2]	Hardware Divide[3]	ISA Revision[4]	Memory Management Unit	Floating Point Unit[5]
5102	1	-	-	A	-	-
5202	2	-	-	A	-	-
5204	2	-	-	A	-	-
5206	2	-	-	A	-	-
5206e	2	mac	yes	A	-	-
5249	2	emac	yes	A	-	-
5272	2	mac	yes	A	-	-
5280	2	emac	yes	A+	-	-
5281	2	emac	yes	A+	-	-
5282	2	emac	yes	A+	-	-
5307	3	mac	yes	A	-	-
5407	4	mac	yes	B	-	-
5470	4e	emac	yes	B	yes	yes
5471	4e	emac	yes	B	yes	yes
5472	4e	emac	yes	B	yes	yes
5473	4e	emac	yes	B	yes	yes
5474	4e	emac	yes	B	yes	yes
5475	4e	emac	yes	B	yes	yes
5480	4e	emac	yes	B	yes	yes
5481	4e	emac	yes	B	yes	yes
5482	4e	emac	yes	B	yes	yes
5483	4e	emac	yes	B	yes	yes
5484	4e	emac	yes	B	yes	yes
5485	4e	emac	yes	B	yes	yes
any	4e	emac	yes	B	yes	yes

1. Core is the version number of the internal processor core.
2. MAC instructions are available on processors with a multiply accumulate unit.
3. DIV instructions are available on processors with hardware divide capability.
4. ISA Revision is the Instruction Set Addition revision. The 5407 (any Rev B) has additional instructions. ISA A+ instructions include bitrev, byterev, ff1 and stldsr.
5. Floating-point instructions are available on processors with a floating-point unit.

2. Details

Symbol: A string of characters with a non-initial digit. The string of characters may be from the set: [a-z][A-Z]_[0-9]$ (_ counts as a non-digit). The $' counts as a digit to avoid confusion with hexadecimal constants. All characters of a symbol are significant, with upper and lower case characters being distinct. The maximum number of characters in a symbol is currently set at ninety (90). The symbol table can grow until the assembler runs out of memory.

Label: A symbol starting in the first column is a label and may optionally be ended with a ':'. A label may appear on a line by itself and is interpreted as:-

```
label   equ *
```

Local Label: A local label is defined as any label ending in a '$'. This label is special in that it belongs only between two normal labels. For example:-

```
start:   nop
10$      bra 10$
mid:     nop
10$      bra 10$
endcode:
```

Mnemonic: A symbol preceded by at least one white space character. Upper case characters in this field are converted to lower case before being checked as a legal mnemonic. Thus nop', NOP' and even NoP' are recognized as the same mnemonic.

Operand: Follows mnemonic, separated by at least one white space character. The contents of the operand field is interpreted by each instruction.

White space: A blank or a tab character.

Comment: Any text after all operands for a given mnemonic have been processed or, a line beginning with semicolon, asterix or hash (; * #) characters up to the end of line or, an empty line.

Continuations: If a line ends with a backslash (\ ) then the next line is fetched and added to the end of the first line. This continues until a line is seen which doesn't end in backslash or until the character buffer is full (256 characters).

Strings:

Strings may be enclosed by either the double or single quote (“ ‘) characters. For example:-

```
mymessage   dc.b  “It’s a great day”,0
filesystem   equ   “nufs”
```

Note: The end of string should be enclosed with the same quote character that was used to define the string.

2.1 Branches

Branches can contain byte,short,word or long (version 4 core or greater) offsets. Word offset is the default for unspecified branches.

```
bra    10$       ; Defaults to word offset
bra.b  10$       ; Byte offset
bra.s  10$       ; Short offset (same as byte)
bra.w  10$       ; Word offset
bra.l  10$       ; Longword offset (version 4 or greater core only)
```

Branch optimisation can be implemented for backward branches, so that a word or longword offset could be reduced to a byte offset if possible (see section 1.4.2).

3. Command Line Options

-a<buf>	allocate forward reference buffers (minimum 8192).
-b	Generate binary ROM file.
-c<cpu>	CPU selection (e.g. 5407, 5206e ...)
-d	Turn off '.' local labels (use with cfcc).
-f	Forward branches default to byte offset.
-l	Listing output enabled.
-n	Disable incbin auto-alignment.
-o	Generate object code file.
-r	Turn on RTS/RTE post NOP generation.
-q	Quiet flag. Final error count or success suppressed.
-u	Allow cpu specific unsized opcode mode. (see section 1.4.2)
-x<val>	Debug Options (see section 4).

4. Debug Options

The debug option (-x) uses a weighted number to turn on one or more print statements:

1	Parser.
2	Template matches.
4	Expression Evaluation.
8	Symbol table lookup/install.
16	Forward references.
32	Indexed Indirect information.
64	Dump all important tables.

For example, -x10 displays template match operation and symbol table info.

5. Files

Cfasm assembles files to an output file, usually post-fixed with ".o". All assembly information and error messages are sent to standard output (stdout), allowing possible redirection.

Support for a global include path is provided with the *cfasmincl* environment variable.

<inputfilename>.o	File output (SREC, object or binary).
stdout	Listing and errors.
cfasmincl	Environment variable to enable a global include path. For example:- *setenv cfasmincl "prog:coldfire/projects/include"* or for other systems try set instead of setenv.

6. Implementation Notes

This is a classic two-pass assembler. Pass one establishes the symbol table and pass two generates the code.

7. Known Bugs

This assembler is still under development.

1. The assembler may crash if compiling non-ascii files accidentally.

If you find any program bugs, then please send reports to:-

bugs@austexsoftware.com

8. Changes

2.64 Added new v4e processors.
2.62 Added mcf5281.
2.60 Increased forward reference handling size and can be modified via new command line option. Fixed problem with evaluation of sectioned variables and generation of linker data.
2.56 Fixed problem with classtags and label references not generating relocatable blocks.
2.54 Modified internal register references (other_a7, flashbar).
2.52 Additional parameters on pseudo-op 'call' now produce an error.
2.50 Added ISA+ instructions. Some EMAC instruction extensions were not reporting errors on non-EMAC processors. S-Record end output corrected.
2.40 Added enumeration pseudo ops.
2.32 Call function zero offset bug fixed.
2.30 Removed emit debug code. CMPI instruction no longer allows non data register compares.
2.28 LEA optimization phasing error fixed.
2.26 Bumped for new release.
2.24 Added ColdFire 5280/5282 processors.
2.22 More work on optimizations.
2.20 Modified binary handling of sections. Data is now exported from sections other than code. Added processor block.
2.05 Updated blocks. Fixed problem with s-record synchronization.
2.04 Modified source code handling for higher compatibility. Added command line option -c for cpu selection.
2.03 Fixed opcode extension bug with v2.02 optimisation. Added mode to allow cpu specific unsized-opcode sizing. Optimise functions can be individually disabled.
2.02 Added quiet flag. Fixed bug with addi and subi undefined size code generation. These opcodes now support optimisation.
2.01 Fixed floating-point exponent handling.
2.00 Added floating-point and MMU support. Modified various pseudo-op code. Fixed bugs with comments after some pseudo-ops. String equates added. Define storage enhanced. Data section alignment fixed. ColdFire 5249 added.
1.92 Modified conditional assembly handling.
1.90 Bad branch destinations no longer produce phasing errors. Some modifications to error messages.
1.88 Modification of allowed type combinations has fixed some phasing errors and improved code optimisation.
1.84 Fixed bug with macl "upper" generation.
1.82 Fixed minor end-of-file conditions. This only applies to source files that terminated in a non line-feed character.
1.80 Error on comments after labels now fixed.
1.78 Added callres and callusr pseudo-ops for calling resource functions from within resources or user code.
1.74 Classtag values in conditional blocks now displayed in listing.
1.72 Fixed bug with conditional assembly and forward references. Equates that don't evaluate in pass one now produce an error.
1.70 Added new opcode handling for MAC V4e core instructions. Finally fixed the annoying last line of listing sometimes containing previous characters.

1.60 Forward references within conditional blocks now produce an error. Fixed error filename reporting mechanism.
1.52 Fixed generation of mov3q opcode for ColdFire 5407 processor.
1.50 Added index addressing optimisation. New boundary pseudo-op added.
1.40 Implemented classtag system.
1.38 Better addq/subq optimisation code. Unrecognised optimise flag now gives a warning.
1.36 Fixed branch optimisation.
1.34 Increased the parsed line items from 16 to 32.
1.32 Added environment variable cfasmincl, so that include files can be easily located.
1.30 Enhancements to XDEF handling.
1.28 Fixed bug with end-of-file termination regarding single TAB characters on a line by itself at the end of an assembly file.
1.26 TPF instruction added (same as TRAPF inst). CPUSHL syntax modified.
1.24 Fixed bit manipulation instructions. An error is now reported when an xdef could not be found. Added cpu option. Added " character for string enclosure. Enhanced code generation (optimisations).