The as12 assembler

Version 1.2: 8 July 1996 by Karl Lunt, http://www.seanet.com/~karllunt

Version 1.2a: 7 April 1999 by Tom Almy, http://www.aracnet.com/~tomalmy

Version 1.2b: 18 January 2003 by Eric Engler, http://www.geocities.com/SiliconValley/Network/2114/

Version 1.2c: 29 January 2003 by Eric Engler, http://www.geocities.com/SiliconValley/Network/2114/

Version 1.2d: 29 March 2003 by Eric Engler, http://www.geocities.com/SiliconValley/Network/2114/

Version 1.2e June 19, 2005 by Matthew Kincaid, Eric Engler, Tom Almy http://www.ericengler.com/AsmIDE.aspx

The current version of as12 is available here: http://www.ericengler.com/AsmIDE.aspx

---

as12 constructs

Invoking the as12 assembler

To start the as12 assembler, enter the following command at the prompt:

```
as12 foo.asm
```

Where as12 is the name of the assembler you want to use, and foo.asm is the path, name, and extension of the file you want to assemble.
The as12 assembler will assemble the file, sending the listing output to the console and writing the S19 object output to a file named file.ext.

To save the listing output in a text file for later review, use the -L option. For example:

```
as12 foo.asm -Lfoo.lst
```

will assemble the file foo.asm and write the output listing to a file named foo.lst.
Entering **as12** with no arguments will display a short help file describing the available commandline options.

Commandline Options

Commandline options allow you to specify input file names, define global symbols, and declare paths for library files. For example:

```
as12 foo.asm -dTIMERS -l\mylib\hc12 -Lfoo.lst
```

will assemble the file foo.asm, using the library files found in the directory \mylib\hc12. Additionally, the label

TIMERS will be defined with a value of 1 and the listing output will be written to file foo.lst.

When specifying the assembler source file, the extension ".asm" is assumed if no extension is explicitly given.

The full set of commandline options includes:

-o<filename>	Define object file (default extension is .s19)
-d<symbol>	Define the symbol 'name' with a value of 1
-l<dir>	Define a library directory with path 'lib'
-L<filename>	Define listing file (default extension is .lst)
-D	Turn on debugging printout
-s<filename>	Create a symbol table file (use dflt: filename.sym)
-p<part #>	Define MCU part number, such as 68hc12a4 (see #ifp)
--list	Display list file to console
--cycles	Display the cycle count
--line-numbers	Display line numbers in list file
--no-warns	Suppress warnings being displayed to console and list file

---

-d Define a label

The -d option allows you to define a label on the commandline. Labels defined in this manner are treated by the as12 assembler as if they had been defined within your source file and assigned a value of 1. Your source file can then refer to these labels in #ifdef tests. For example:

```
as12 foo.asm -dMY_ALGORITHM
```

causes the as12 assembler to behave as if your source file began with the line:

#define MY_ALGORITHM

This ability to define labels from the commandline adds great power to the as12 assembler. You can use this feature to selectively assemble blocks of source code based on arguments you specify in the commandline, without first having to edit the source code before each assembly.

-l Define a library path

Normally, as12 first checks in the current directory for needed include files. If as12 cannot find a needed file in the current directory, it will try the library path specified with the -l option on the commandline, if any.

For example:

```
as12 foo.asm -lc:\mypath
```

-p Define the target processor

as12 allows you to pass in a specific processor or board to identify the part you are compiling the program for. When combined with the #ifpart conditional assembly directive, can give users a powerful way to compile source code which may depend upon what part is being targeted.

For example:

```
as12 foo.asm -pDragon12
```

-D Debug the as12 assembler

Turns on the internal debug features of as12. Mostly for the developer of as12, but might be helpful if you are having

a problem understanding what as12 is doing.

For example:

```
as12 foo.asm -D
```

-L Listing file

Specifies a listing file. If no filename extension is given, ".lst" is assumed. If no file name is given, then the file name will be that of the first source file, with an extension ".lst".

For example:

```
as12 foo.asm -L
```

Pound Sign (#) Operators

#include

The include directive allows other assembly source files to be inserted in the code immediately after the include statement, **as if** the contents of the included file were actually in the file that contained the include statement. Stated differently, the include statement works as you might expect. The syntax of the include statement is shown below...

```
#include \my_dir\myfile.asm
```

In linux, it is important to note that the filename expansion will only be as good as the filename expansion as the shell that you are operating in. For example, if you are running shell (/bin/sh) then the tilde username (~user) lookup may not work correctly. It is best to put in relative or absolute filepath specifications that are not shell dependent.

Include statements may be used within #ifdef statements.

We support quoted filenames within #include. This lets us use filenames that might have embedded spaces, or the directory name may have embedded spaces:

```
#include "c:\program files\as12\my defs.h"
```

#define

The define statement allows labels to be defined. This statement is simply an alternate form for an equ assembler directive. The alternate form is provided so that users will be alerted to the opportunities to write more sophisticated code that the #ifeq, and related statements allow. The proper use of the define statement is:

#define MY_LABEL expression

The define statement is as if the user had typed the following:

MY_LABEL: EQU expression

Both forms are equally valid, and both forms are implemented internally thee same way. The EQU is probably more portable of the two constructs.

#ifeq

The ifeq command allows for the user to conditionally compile different sections of assembly language based on whether or not a label is equal to a value. Example:

```
#ifeq MY_SYMBOL expression_to_compare_to
  ...
  (this code will be assembled if MY_SYMBOL has the same value as expression_to_compare_to)
  ...
#endif
```

I show the #endif statement because for every form of **#if** there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the expression resolves to the same value as the label in an #ifeq directive, then every line between the #ifeq and the #endif is executed. If the expression resolves to a different value than the label, all of the lines between the #ifeq and the #endif are ignored.

#ifndef

The ifndef command allows for the user to conditionally compile different sections of assembly language based on whether or not a label is defined. Example:

```
#ifndef MY_LABEL
  ...
  (this code will be assembled if MY_LABEL has not been defined)
  ...
#endif
```

I show the #endif statement because for every form of **#if** there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

#ifdef

The ifdef command allows for the user to conditionally compile different sections of assembly language based on whether or not a label is defined (via a #define or an EQU. Example...

```
#ifdef MY_LABEL
  ...
  (this code will be assembled if MY_LABEL has been defined)
  ...
#endif
```

I show the #endif statement because for every form of **#if** there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the label in an #ifdef directive is defined, then every line between the #ifeq and the #endif is executed. If the label is not defined, all of the lines between the #ifdef and the #endif are ignored.

#ifpart

This is the only directive that allows for a string comparison. A special internal variable is the only variable which is a string variable. The only way to set that variable is with the -p commandline option. The sole purpose of this directive is to allow for conditional assembly based upon the value of the string. This seemed natural for handling the different part types. Example..

```
#ifpart b32
  ...
```

```
  (this code will be assembled if the string <b32> is same as string in -p commandline option
  ...
#endif
```

I show the #endif statement because for every form of **#if** there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the string that follows the #ifpart directive matches the string that was passed in via the -p option, then the lines betwee n the #ifpart and the #endif will be executed. If the strings do not match, the lines between the #ifpart and the #endif will be ignored.

#else

This directive must be coupled with any of the if directives. This allows either or compilation and performs just like you expect an else to perform. Example..

```
#ifdef MY_LABEL
  ...
  (this code will be assembled if MY_LABEL is defined)
  ...
#else
  ...
  (this code will be assembled if MY_LABEL is NOT defined)
  ...
#endif
```

I show the #endif statement because for every form of **#if** there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every **#if** there needs to be an **#endif**.

If the **#if** statement that goes with the **#else** statement is true, the statements between the **#if** and the **#else** will be assembled, and the statements between the **#else** and the **#endif** will be ignored. If the **#if** statement is false, the statements between the **#if** and the **#else** will be ignored and the statements between the **#else** and the **#endif** will be executed

There can only be one **#else** for each **#if** statement. But **#else** is optional, so you do not have to use it.

#endif

The **#endif** statement tells the assembler when the conditional assembly section of the code is finished. Otherwise the assembler would have no way of knowing when to quit.

For every **#if** statement there needs to be one **#endif**. If there is an **#if** and an **#else**, then there should be one **#endif** statement also.

Examples:

```
#ifpart part_name
  ...
#else
  ...
#endif

#ifndef MY_LABEL>
  ...
#endif

#ifndef MY_LABEL
  ...
#else
  ...
#endif
```

Typical Conditional Assembly Examples

- Use to handle parts starting in different modes. You can automate this and keep from modifying your source code my defining the label by invoking the assembler using the -d commandline option.

```
#ifdef EXPANDED_MODE
  org START_OF_EXTERNAL_RAM_TESTS
#else
  org START_OF_FLASH_RAM
#endif
```

- Use to handle configuring software so that your code will operate regardless of what part might be used. You can keep from changing your source code by passing in the parttype using the -p commandline option.

```
#ifpart b32
RAM_START:  EQU  $800
FEE_START:  EQU  $8000
REG_START:  EQU  $0000
PWM_START:  EQU  $c7
#endif

#ifpart a4
RAM_START:  EQU  $600
REG_START:  EQU  $0100
#endif
```

Notice how easy you could build a library of different parts and make your source code compile accordingly.

Files

- Assembler executables, as12
- Motorola machine code, *.s19
- Source files, *.asm
- Listing files, *.lst

Assembler executable, as12

Filename: as12.exe.

NOTE: On linux the .exe extension is not typically used.

These are the cross assemblers that allow you to convert your Motorola source code to Motorola machine code on your PC.

Motorola machine code, *.s19

A file with the same name as the first source file but with the extension ".s19" is used to hold the binary machine code instructions. This is produced by the assembler: the assembler translates text commands into binary commands. The binary data is stored in the s-record format, which is contained in the .s19 file.

This is the file that is sent to the embedded board, where your program will be executed.

For additional information regarding s-records visit Seattle Encoder's s-record article

---

Listing files, *.lst

The listing file is useful for debugging. Simply add the commandline option "-L" to create the listing file.

Source files, *.asm

Standard ASCII source files. These should be created with the extension ".asm" since that is the default used by the

assembler, but is not required.

Features

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: +-*/%&|^. You may nest expressions using parentheses up to 10 levels deep. The operators are the same as in C:

```
+       add
-       subtract
*       multiply
/       divide
%       remainder after division
&       bitwise and
|       bitwise or
^       bitwise exclusive-or
```

In addition, the unary minus (-) and complement (~) operators are allowed when preceding a symbol, constant, or character '*' only.

Examples of valid expressions...

```
(5*8)
(my_val-10+20*(16-label)/10)
10
$10
*
%10010
my_value
~$20
```

Starting with version 1.2e you can **NOT** have spaces in an expression:

```
ldaa foo + 1
```

will produce erronous assembly. The correct way to write this expression is:

```
ldaa foo+1
```

Note: When the asterisk (*) is used in a context where the as12 is expecting a label, the asterisk (*) represents the value of the current program counter.

Symbols

Symbols consist of one or more characters where the first character is alphabetic and any remaining characters are alphanumeric. Symbols **ARE** case sensitive.

Constants

```
'       followed by ASCII character
!       followed by a decimal constant (decimal is assumed, so ! is optional)
$       followed by hexadecimal constant
@       followed by octal constant
%       followed by binary constant
digit   decimal constant
```

Examples:

```
'A
46
$2E
@07
%10001001
```

Labels

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 then interpreted as:

```
        Label   EQU     *
```

Note that labels are **NOT** case sensitive. You can use labels named LABEL interchangebly with LaBeL.

Comments

Here are some notes about comments...

- Any line beginning with an ***** in column 1 is a comment line
- Any text beginning with a **;** is a comment - does not have to begin in column 1

AS12 Directives (or pseudo-opcodes)

---

bsz Pseudo Opcode - Block Set Zeros

Sets a block of memory to zero values. Same as zmb.

---

db Pseudo Opcode - Define Byte

Syntax and examples:

```
        db      Byte_Definition[,Byte_Definition]
        db      $55,$66,%11000011
        db      10
half    db      0.5*100
```

db Defines the value of a byte or bytes that will be placed at a given address.

The db directive assigns the value of the expression to the current program counter. Then the program counter is incremented.

Multiple bytes can be defined at a time by comma separating the arguments. Each comma separated argument can be a separate expression that the as 12 will evaluate.

Notes:

- This is probably a more universally accepted pseudo-op than the fcb. However, the selection of a pseudo op does have implications on portability. I provide as many as I can to enhance OUR ability to read other peoples code.
- This should be used for memory that is not considered volatile (ROM/EE/FLASH) or memory that will be boot-loaded or similar. For defining RAM memory for variables and scratchpad memory the ds directive is more appropriate.

Related To:

- fcb
- fdb
- dw
- ds

Useful With:

- Defining Data Tables/Structures
- Defining ASCII phrases (strings)
- Defining Constants

Things to look out for:

- Be careful not to define values that are **larger** than 8 bits. as12 truncates the left most bits to make the byte fit into a byte.
- A label is usually used so there is a reference to this memory. In the last example in the Syntax section, it can be seen that the label **half** will refer to the byte with a decimal value of 50. (Not really fixed point math but I'm only demonstrating the use of a label)

---

dc.b Pseudo Opcode - Define Constant Byte - declare a byte of memory

Identical to db

---

dc.w Pseudo Opcode - Define Constant Word - declare a word of memory

Identical to dw.

---

ds Pseudo Opcode - Define Storage

Syntax and examples:

```
ds      Number_of_Bytes_To_Advance_Program_Counter
```

The ds increments the program counter by the value indicated in the Number of Bytes argument.

Notes:

- This is the preferred method of defining a memory location whose value...
  - is changing
  - is generally not known

- In other words, this is optimal for defining RAM or REGISTER spaces. The reason for this is the ease in which a ds based region can be relocated.

Related To:

- rmb

Useful With:

- RAM definitions
- REGISTER definitions

Things to look out for:

- Inappropriate for non-volatile memory definitions

---

ds.b Pseudo Opcode - Define Storage Bytes - declare bytes of storage

Identical to ds.

---

ds.w Pseudo Opcode - Define Storage Word

Syntax and examples:

```
        ds.w    Number_of_Words_To_Advance_Program_Counter
```

The ds.w increments the program counter by the value indicated in the argument multiplied by two. In other words, if the ds.w expression evaluates to 4 then the program counter is advanced by 8.

Notes:

- Good for defining RAM and REGISTERS

Related To:

- ds

Useful With:

- labels

Things to look out for:

- Inappropriate for non-volatile memory.

---

dw Pseudo Opcode - Define Word

Syntax and examples:

```
        dw      Word_Definition[,Word_Definition]
        dw      $55aa,$66,%11000011
        dw      10
half    dw      0.5*65536
```

Defines the value of a word or words that will be placed at a given address.

The dw directive assigns the value of the expression to the current program counter. Then the program counter is incremented by 2.

Multiple words can be defined at a time by comma separating the arguments. Each comma separated argument can be a separate expression that the as 12 will evaluate.

Notes:

- This is probably a more universally accepted pseudo-op than the fdb. However, the selection of a pseudo op does have implications on portability. I provide as many as I can to enhance OUR ability to read other peoples code.
- This should be used for memory that is not considered volatile (ROM/EE/FLASH) or memory that will be boot-loaded or similar. For defining RAM memory for variables and scratchpad memory the ds directive is more appropriate.
- Words are right justified and left filled with zero's.

Related To:

- fdb
- dc.w

Useful With:

- Defining Data Tables/Structures
- Defining Constants

Things to look out for:

- Be careful not to define values that are **larger** than 16 bits. as12 truncates the left most bits to make the word fit into a word.

---

equ Pseudo Opcode - Equate

Syntax and examples:

```
Label   EQU     Value_To_Assign_To_The_Label
```

Directly assigns a numeric value to a label.

Notes:

- assigns a meaningful name to constants

Related To:

- #define
- -d commandline option

Useful With:

- #ifeq and related options

Things to look out for:

- Be careful of how many bits your label can take. The as12 internally uses anywhere from 32 bits for the label value with the Win32 version. It is very easy to get bigger than 8 or 16 bits.
- Inappropriate for defining memory locations. I would recommend only using for defining constants. Otherwise relocation can be made very difficult.

---

fcb Pseudo Opcode - Form Constant Byte - declare bytes of storage

Identical to db.

---

fcc Pseudo Opcode - Form Constant Characters

Syntax and examples :

```
fcc     delim_characterstring_to_encodedelim_character
fcc     /my_string/
fcc     *// string with slashes //*
fcc     'best to use single quotes'
```

FCC allows the encoding of a string.

The first character is the delimiter. By allowing the flexibility of selecting delimiters, you can easily make strings which have slashes and tick marks in them. The **only** catch is that if you choose a delimiter, it

- must also be used to mark the end of the string
- it cannot appear in the string as a character.

In the second example, my_string will be encoded as an ASCII string. The /'s simply mark the ending and beginning of the string. This also lets you put spaces in the string.

In the third example, the * (asterisk) is the delimiter and the slashes will be encoded with their ASCII values into the ASCII string.

Notes:

- You cannot have the space as a delimiter
- you can also define strings using FCB except that you have to encode them one character at a time and comma delimit them.

Related To:

- fcb

Useful With:

- Defining strings for displays and such.

---

fdb Pseudo Opcode - Form Double Byte - declare words of storage

Identical to dw.

---

fill Pseudo Opcode - Fill Memory

Syntax and examples:

```
fill    byte_to_fill_memory_with,num_of_bytes_to_fill
```

FILL allows a user to fill memory with a byte. See my comments in zmb about the value of these Pseudo Opcodes.

Notes:

- Nice for initializing memory.

Related To:

- zmb

Useful With:

- Debugging
- Filling unused non-volatile memory with a **safe** opcode when the processor gets lost.

Things to look out for:

- Since RAM memory, by definition, cannot be initialized, this command has little use. This is because you must DOWNLOAD the s-records to make the clearing take place. Only in systems which have some sort of

bootstrapping (where s-records are downloaded) would this be very useful. If you are clearing memory, you should probably count on routines to do it for you.

---

loc Pseudo Opcode - creates automatically incrementing labels

WARNING: Some people do not like to see this command used in your programs

Increments and produces an internal counter used in conjunction with the backwards tick mark (`). By using LOC's and the ` mark you can write code like the following without worrying about thinking up new labels.

```
        LOC
        ldaa    #1
loop`
        deca
        bra     loop`
        LOC
loop`
        brset   0,x $55 loop`
```

This code will work perfectly fine because the second loops label is really loop002 and the first ones is loop001. The assembler really sees this:

```
        LOC
        ldaa    #1
loop001
        deca
        bra     loop001
        LOC
loop002
        brset   0,x $55 loop002
```

You may also seed the LOC with a valid expression or number by putting that expression or number in the operand field. This gives you the ability to over ride the automatic numbering. This is also sometimes handy if you need to keep track of what your local variable is. (you lose track in the source if you aren't careful, because the tick ' mark is the only thing you see).

---

opt Pseudo Opcode - Assembler List Options

There are five permissible operands for this instruction:

- l - enable listing after **opt nol**
- nol - disable listing until **opt l** or end of source code
- c - calculate execution time (clock cycles)
- noc - stop calculating execution time
- contc - continue calculating execution time

The org Pseudo Opcode - Origin

Specify the address in memory where the following code should be located.

Syntax and examples:

```
        org     value_to_set_program_counter_to
        org     $800
        org     MY_PROGRAM_START ; use a symbol defined elsewhere with EQU
        org     LAST_MEMORY_LOCATION-(LAST_PROGRAM_BYTE-FIRST_PROGRAM_BYTE) ; calculate a value
```

The org Pseudo Opcode allows the assembler's program counter to be set to a value. This is useful for locating your software and its elements (tables, ram, constants, etc) in useful (intelligent) locations within the memory space of the microcontroller.

In better multi-pass assemblers (not as12), the org statement is rarely used because the code is located at the link, and

not during compilation. Since as12 is a **simple** two-pass assembler, orgs must be used so that the code is compiled where it is supposed to.

Notes:

- When starting a new region of code, you can examine the s-record file and see how org affects the construction of that file.
- It is better to use the form **org label** than **org constant** because the more constants that are buried within your code, the more difficult it is to reuse.
- The less orgs you use, the more reusable your code is.

Related To:

- program counter because this sets its value
- rmb and its cousins because they change the program counter

Things to look out for:

- Always find out where the orgs are in a program. This is the first key to understanding the program.

---

redef Pseudo Opcode - Redefine

WARNING: Some people do not like to see this command used in your programs

Used to redefine first operand (which must be a label) to value of second operand (an expression)

Example:

```
foo     equ     10
        ldaa    #foo        ; Accumulator A gets value 10
        redef   foo 12
        ldab    #foo      ; Accumulator B gets value 12
```

---

rmb Pseudo Opcode - Reserve Memory Bytes

Equivalent to **ds.b** and **ds**.

---

rmw Pseudo Opcode - Reserve Memory Words

Equivalent to **ds.w**.

---

zmb Pseudo Opcode - Zero Memory Bytes

Operand specifies number of bytes to allocate and fill with zero. Similar to bss on some assemblers.

AS12 Opcode Mnemonics - Names for Machine Code Instructions

- **ABA** - add accumulator B to accumulator A
- **ABX** - add accumulator B to index reg. X
- **ABY** - add accumulator B to index reg. Y
- **ADCA** - add with carry to A
- **ADCB** - add with carry to B
- **ADDA** - add without carry to A
- **ADDB** - add without carry to B
- **ADDD** - add double accumulator
- **ANDA** - logical and A
- **ANDB** - logical and B

- **ANDCC** - logical and CCR with mask *
- **ASL** - arithmetic shift left memory
- **ASLA** - arithmetic shift left A
- **ASLB** - arithmetic shift left B
- **ASLD** - arithmetic shift left double acc.
- **ASR** - arithmetic shift right memory
- **ASRA** - arithmetic shift right A
- **ASRB** - arithmetic shift right B
- **BCC** - branch if carry clear
- **BCLR** - clear bit(s) in memory
- **BCS** - branch if carry set
- **BEQ** - branch if equal
- **BGE** - branch if >= zero
- **BGND** - enter background debug mode *
- **BGT** - branch if > zero
- **BHI** - branch if higher
- **BHS** - branch if higher or same
- **BITA** - bit test A
- **BITB** - bit test B
- **BLE** - branch if <= zero
- **BLO** - branch if lower
- **BLS** - branch if lower or same
- **BLT** - branch if < zero
- **BMI** - branch if minus
- **BNE** - branch if not equal to zero
- **BPL** - branch if plus
- **BRA** - branch always
- **BRCLR** - branch if bit(s) clear
- **BRN** - branch never
- **BRSET** - branch if bit(s) set
- **BSET** - set bit(s) in memory
- **BSR** - branch to subroutine
- **BVC** - branch if overflow clear
- **BVS** - branch if overflow set
- **CALL** - call subroutine in extended memory *
- **CBA** - compare accumulators
- **CLC** - clear carry
- **CLI** - clear interrupt mask
- **CLR** - clear memory
- **CLRA** - clear A
- **CLRB** - clear B
- **CLV** - clear two's complement overflow bit
- **CMPA** - compare A
- **CMPB** - compare B
- **COM** - complement memory
- **COMA** - complement A
- **COMB** - complement B
- **CPD** - compare accumulator D
- **CPS** - compare stack pointer *
- **CPX** - compare index reg. X
- **CPY** - compare index reg. Y
- **DAA** - decimal adjust A
- **DBEQ** - decrement and branch if equal to zero *
- **DBNE** - decrement and branch if not equal to zero *
- **DEC** - decrement memory
- **DECA** - decrement A
- **DECB** - decrement B
- **DES** - decrement stack pointer
- **DEX** - decrement index register X
- **DEY** - decrement index register Y

- **EDIV** - extended divide 32-bit by 16-bit (unsigned) *
- **EDIVS** - extended divide 32-bit by 16-bit (signed) *
- **EMACS** - extended multiply and accumulate (signed) *
- **EMAXD** - max of 2 unsigned 16-bit values (result in D) *
- **EMAXM** - max of 2 unsigned 16-bit values (result in mem) *
- **EMIND** - min of 2 unsigned 16-bit values (result in D) *
- **EMINM** - min of 2 unsigned 16-bit values (result in mem) *
- **EMUL** - extended multiply 16-bit by 16-bit (unsigned) *
- **EMULS** - extended multiply 16-bit by 16-bit (signed) *
- **EORA** - exclusive or A
- **EORB** - exclusive or B
- **ETBL** - extended table lookup and interpolate *
- **EXG** - exchange register contents *
- **FDIV** - fractional divide
- **IBEQ** - increment and branch if equal to zero *
- **IBNE** - increment and branch if not equal to zero *
- **IDIV** - integer divide
- **IDIVS** - integer divide (signed) *
- **INC** - increment memory
- **INCA** - increment A
- **INCB** - increment B
- **INS** - increment stack pointer
- **INX** - increment index register X
- **INY** - increment index register Y
- **JMP** - jump
- **JSR** - jump to subroutine
- **LBCC** - long branch if carry clear *
- **LBCS** - long branch if carry set *
- **LBEQ** - long branch if equal *
- **LBGE** - long branch if greater than or equal to zero *
- **LBGT** - long branch if greater than zero *
- **LBHI** - long branch if higher *
- **LBHS** - long branch if higher or same *
- **LBLE** - long branch if less than or equal to zero *
- **LBLO** - long branch if lower *
- **LBLS** - long branch if lower or same *
- **LBLT** - long branch if less than zero *
- **LBMI** - long branch if minus *
- **LBNE** - long branch if not equal to zero *
- **LBPL** - long branch if plus *
- **LBRA** - long branch always *
- **LBRN** - long branch never *
- **LBVC** - long branch if overflow clear *
- **LBVS** - long branch if overflow set *
- **LDAA** - load accumulator A
- **LDAB** - load accumulator B
- **LDD** - load double accumulator
- **LDS** - load stack pointer
- **LDX** - load index register X
- **LDY** - load index register Y
- **LEAS** - load stack pointer with effective address *
- **LEAX** - load X with effective address *
- **LEAY** - load Y with effective address *
- **LSL** - logical shift left memory
- **LSLA** - logical shift left A
- **LSLB** - logical shift left B
- **LSLD** - logical shift left double
- **LSR** - logical shift right memory
- **LSRA** - logical shift right A
- **LSRB** - logical shift right B

- **LSRD** - logical shift right double accumulator
- **MAXA** - max of 2 unsigned 8-bit values (result in A) *
- **MAXM** - max of 2 unsigned 8-bit values (result in mem) *
- **MEM** - determine grade of membership *
- **MINA** - min of 2 unsigned 8-bit values (result in A) *
- **MINM** - min of 2 unsigned 8-bit values (result in mem) *
- **MOVB** - move data from one memory byte to another *
- **MOVW** - move data from one memory word to another *
- **MUL** - multiply unsigned
- **NEG** - negate memory
- **NEGA** - negate A
- **NEGB** - negate B
- **NOP** - no operation
- **ORAA** - inclusive or A
- **ORAB** - inclusive or B
- **ORCC** - logical or CCR with mask *
- **PSHA** - push A onto stack
- **PSHB** - push B onto stack
- **PSHC** - push CCR onto stack *
- **PSHD** - push double accumulator onto stack *
- **PSHX** - push index reg. X onto stack
- **PSHY** - push index reg. Y onto stack
- **PULA** - pull A from stack
- **PULB** - pull B from stack
- **PULC** - pull CCR from stack *
- **PULD** - pull double accumulator from stack *
- **PULX** - pull index reg. X from stack
- **PULY** - pull index reg. Y from stack
- **REV** - fuzzy logic rule evaluation *
- **REVW** - fuzzy logic rule evaluation (weighted) *
- **ROL** - rotate left memory
- **ROLA** - rotate left A
- **ROLB** - rotate left B
- **ROR** - rotate right memory
- **RORA** - rotate right A
- **RORB** - rotate right B
- **RTC** - return from call *
- **RTI** - return from interrupt
- **RTS** - return from subroutine
- **SBA** - subtract accumulators
- **SBCA** - subtract with carry from A
- **SBCB** - subtract with carry from B
- **SEC** - set carry
- **SEI** - set interrupt mask
- **SEV** - set two's complement overflow bit
- **SEX** - sign extend into 16-bit register *
- **STAA** - store accumulator A
- **STAB** - store accumulator B
- **STD** - store double accumulator
- **STOP** - stop processing
- **STS** - store stack pointer
- **STX** - store index register X
- **STY** - store index register Y
- **SUBA** - subtract A
- **SUBB** - subtract B
- **SUBD** - subtract double accumulator
- **SWI** - software interrupt
- **TAB** - transfer from acc. A to acc. B
- **TAP** - transfer from acc. A to CCR
- **TBA** - transfer from acc. B to acc. A

- **TBEQ** - test and branch if equal to zero *
- **TBL** - table lookup and interpolate *
- **TBNE** - test and branch if not equal to zero *
- **TFR** - transfer register content to another register *
- **TPA** - transfer from CCR to accumulator A
- **TRAP** - unimplemented opcode trap
- **TST** - test memory
- **TSTA** - test A
- **TSTB** - test B
- **TSX** - transfer from SP to index reg. X
- **TSY** - transfer from SP to index reg. Y
- **TXS** - transfer from index reg. X to SP
- **TYS** - transfer from index reg. Y to SP
- **WAI** - wait for interrupts
- **WAV** - weighted average *
- **XGDX** - exchange double acc. and index reg. X
- **XGDY** - exchange double acc. and index reg. Y

* means a new opcode that was not supported on the 68hc11

Non-standard Opcode Mnemonics

These mnemonics are defined in as12, but are not considered standard.

I recommend that you do not use these in your programs. This is mostly an issue for people who might want to migrate their code to a different assembler in the future - that other assembler won't understand these opcodes (although in many cases you can get around it by defining a macro for each of these):

- **bkgnd** - - alias for bgnd
- **cbnz** - - alias for dbeq
- **cmpd** - - alias for cpd
- **cmps** - - alias for cps
- **cmpx** - - alias for cpx
- **cmpy** - - alias for cpy
- **lbsr** - - alias for jsr
- **lda** - - alias for ldaa
- **ldad** - - alias for ldd
- **ldb** - - alias for ldab
- **ora** - - alias for oraa
- **orb** - - alias for orab
- **pshbyte** - - alias for movb
- **pshword** - - alias for movw
- **pulbyte** - - alias for movb
- **pulword** - - alias for movw
- **sta** - - alias for staa
- **stb** - - alias for stab
- **swpb** - - alias for tap
- **wavr** - - alias for wav