A Programming Tool for
Cirrus Logic 32-Bit Audio DSPs

Cirrus Logic 32-bit DSP Assembly Programmer's Guide

Preliminary Product Information

This document contains information for a new product.
Cirrus Logic reserves the right to modify this product without notice.

Copyright 2013 Cirrus Logic, Inc.

SEP 2013
DS795UM11

http://www.cirrus.com

Contacting Cirrus Logic Support

For all product questions and inquiries contact a Cirrus Logic Sales Representative.
To find one nearest you go to **www.cirrus.com**

IMPORTANT NOTICE

"Preliminary" product information describes products that are in production, but for which full characterization data is not yet available.

Cirrus Logic, Inc. and its subsidiaries ("Cirrus") believe that the information contained in this document is accurate and reliable. However, the information is subject to change without notice and is provided "AS IS" without warranty of any kind (express or implied). Customers are advised to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, indemnification, and limitation of liability. No responsibility is assumed by Cirrus for the use of this information, including use of this information as the basis for manufacture or sale of any items, or for infringement of patents or other rights of third parties. This document is the property of Cirrus and by furnishing this information, Cirrus grants no license, express or implied under any patents, mask work rights, copyrights, trademarks, trade secrets or other intellectual property rights. Cirrus owns the copyrights associated with the information contained herein and gives consent for copies to be made of the information only for use within your organization with respect to Cirrus integrated circuits or other products of Cirrus. This consent does not extend to other copying such as copying for general distribution, advertising or promotional purposes, or for creating any work for resale.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). CIRRUS PRODUCTS ARE NOT DESIGNED, AUTHORIZED OR WARRANTED FOR USE IN AIRCRAFT SYSTEMS, MILITARY APPLICATIONS, PRODUCTS SURGICALLY IMPLANTED INTO THE BODY, LIFE SUPPORT PRODUCTS OR OTHER CRITICAL APPLICATIONS (INCLUDING MEDICAL DEVICES, AIRCRAFT SYSTEMS OR COMPONENTS AND PERSONAL OR AUTOMOTIVE SAFETY OR SECURITY DEVICES). INCLUSION OF CIRRUS PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK AND CIRRUS DISCLAIMS AND MAKES NO WARRANTY, EXPRESS, STATUTORY OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE, WITH REGARD TO ANY CIRRUS PRODUCT THAT IS USED IN SUCH A MANNER. IF THE CUSTOMER OR CUSTOMER'S CUSTOMER USES OR PERMITS THE USE OF CIRRUS PRODUCTS IN CRITICAL APPLICATIONS, CUSTOMER AGREES, BY SUCH USE, TO FULLY INDEMNIFY CIRRUS, ITS OFFICERS, DIRECTORS, EMPLOYEES, DISTRIBUTORS AND OTHER AGENTS FROM ANY AND ALL LIABILITY, INCLUDING ATTORNEYS' FEES AND COSTS, THAT MAY RESULT FROM OR ARISE IN CONNECTION WITH THESE USES.

Cirrus Logic, Cirrus, and the Cirrus Logic logo designs are trademarks of Cirrus Logic, Inc. All other brand and product names in this document may be trademarks or service marks of their respective owners.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Microwire is a trademark of National Semiconductor Corp. National Semiconductor is a registered trademark of National Semiconductor Corp.

Texas Instruments is a registered trademark of Texas Instruments, Inc.

Motorola is a registered trademark of Motorola, Inc.

LINUX is a registered trademark of Linus Torvalds.

Contents

Copyright 2013 Cirrus Logic, Inc.

Copyright 2013 Cirrus Logic, Inc.

Copyright 2013 Cirrus Logic, Inc.

Copyright 2013 Cirrus Logic, Inc.

Copyright 2013 Cirrus Logic, Inc.

Figures

Tables

Copyright 2013 Cirrus Logic, Inc.

Copyright 2013 Cirrus Logic, Inc.

Chapter 1
Cirrus Logic Assembly Program (CASM)

1.1 Welcome to CASM

The Cirrus Logic Cross-assembler (CASM) application was originally used by software developers at Cirrus Logic for over 10 years to implement custom DSP audio algorithms on the Cirrus Logic 32-bit DSP core-based platforms such as the CS4953xx, CS4970x4, CS485xx, CS470xx, and the CS498xx multicore DSPs.

Cirrus Logic offers CASM as part of the Cirrus Logic Integrated Development Environment (CLIDE) that is available to Cirrus Logic customers to develop their own custom audio algorithms to run on Cirrus Logic DSPs.

Note: The Cirrus Device Manager (CDM) must be running to use the CLIDE tool set. After the SDK for the Cirrus DSP used in the customer's design is launched, CDM should launch automatically. CDM provides communication between CLIDE and the board and simulator.

The CLIDE tool set includes:

- CLIDE's graphical user interface (GUI) is described in the *CLIDE User's Manual* and allows the user to access the following applications from CLIDE:
  - *CASM*—described in this manual.
  - *Cirrus Logic C-Compiler*—described in the *Cirrus Logic C-Compiler User's Manual.*
  - *CLIDE* debugger—described in the *CLIDE User's Manual*; debugs both Assembly and C language source files, and replaces the Hydra debugger.
  - *Cirrus Linker (CLINK)*—described in the *CLINK User's Manual;* takes one or more object files as input and creates binary file(s) suitable for loading.
  - *Primitive Elements Wizard*—described in the *CLIDE User's Manual* and is an XML file wizard used to create custom primitives that are debugged within CLIDE.
  - Simulator—described in the *CLIDE User's Manual*.
  - Source editor—described in the *CLIDE User's Manual*.

Copyright 2013 Cirrus Logic

1.2 Accessing CASM Through the CLIDE GUI

Most users access CASM through the CLIDE GUI, which is described fully in the *CLIDE User's Manual*, available from the main CLIDE window in Help→Help Contents. To access CLIDE, follow these steps:

1. Install the SDK for the Cirrus DSP used in your system design.
2. From the Windows Start menu, select Cirrus Logic DSP→Programming Tools→CLIDE.
3. After CLIDE has opened, select **File →New →Project**.
4. Select one of the wizard-based templates and develop your DSP software project by following the instructions contained in the *CLIDE User's Manual*. Users can also use CLIDE's on-line Help system for assistance.

CLIDE has an Assembler tab in Project properties that can be used to set some of the options when assembling within CLIDE.

1.3 Accessing CASM Through the Assembler Command Line

Users can also access the assembler software by launching the casm.exe application. The casm.exe application can be run from a console window opened from within the Cirrus Device Manager (CDM) application, a batch file, or from a "make" utility.

Open the Cirrus Device Manager by clicking the CDM icon in the system tray, and then open a console window by selecting the menu option **File→Open build console**. The command line specifies the source file and control directives for processing the file.

1.3.1 Command Line Format

The format of the assembler command line is as follows:

casm <source file> <options>

Note: These can be in any order.

<source file> is a single valid file path representing the file containing assembler code. The <source file> parameter must be specified. Otherwise, CASM exits with an error message. If <source file> has no extension, CASM will append the default extension '.A' to <source file> prior to searching for the file. If the assembler does not find <source file>, it exits with an error message.

If the assembler finds <source file> and the environment variable CASMSPEC is included in a command, the environment string of the CAMSPEC variable is used as the default assembler option. The format of the options defined in CASMSPEC must follow the format described in Section 1.3.2.

Copyright 2013 Cirrus Logic

1.3.2 Command Line Options

<options> contains zero or more control directives to the assembler. Each control directive starts with the character '-'(dash or minus) and ends with un-quoted white space. A single dash '-' is used for single character options and a double dash '--' is used for multiple character options. The text after the initial dash indicates the specific control directive for the assembler to employ.

The formats of the various options are as follows. Command lines are case-insensitive. See Section 1.4.3 for details. All alphabetic characters are transformed to upper case prior to passing the options to the assembler.

Table 1-1 Command Line Options

Option	Description
-a<file_name>	Macro preprocessed source code output.
-c	Enable case sensitivity; CASM is case insensitive by default.
--casmspec	CASMSPEC is an environment variable that the user can set to a default value for CASM to use when given the –-casmspec option. For example to set up CASMSPEC, the user could select the following options: ? The user should begin with the CASM requirement that the –i<macro include file> be included in every invocation. ? The user might want CASM to be case sensitive with the -c directive. ? The user might want a listing file to always be generated with the `-l` directive. To accomplish the user's wishes, the CASMSPEC environmental variable in this manner: `CASMSPEC=-iC:\CirrusDSP\bin\athena.h –c -l` After CASMSPEC is configured using the `--casmspec` option ensures preset options are used whenever CASM is called. **Note:** Environment strings do not allow the '=' character, so ':' must be used for definitions of the environment variable CASMSPEC. See Section 1.4.4 for more information on defining and referencing symbols in the Cirrus Logic 32-bit DSP assembler.
--cdl	Emit dependency file .adf
--help	Display all valid command line options and switches.
-d<symbol>[:<value>]	The -D or -d directive instructs the assembler to define a symbol with label <symbol> prior to assembling the source code. A symbol defined in this manner can be referenced in the assembler source as if it were defined in the source. A replacement string <value> can optionally be associated with the symbol by using either the '=' or ':' characters.
--debug	The --debug directive instructs the assembler to add symbol debugging information used by CLIDE to the object file and instructs the assembler to create line-level debug information for the CLIDE debugger.

-e	The -e directive instructs the assembler to produce error output with alternative formatting.
	Example:
	With the -e directive:
	`"<sourcefile> (<line #>) <macro line #>:Error"`
	Without the -e directive:
	`"Error in <sourcefile>:<line #.<macro Line #>"`

-f<file name>	Used for the compatibility with the old CLINK.
-I<include folder>	Set include folder. If you include a header file in your source file such as `.include "i.h"`, then adding `-Ic:\example\inc\` tells CASM to search for `i.h` in the `c:\example\inc` folder.
-i<macro include file>	Platform dependent macros.
-l<lst file>	The -l directive instructs the assembler to create a listing file. If -l<lst file> is given in the option, the listing file will be given this path, otherwise the source code root path will be used with the default extension .LST. Similarly, if -l<lst file> has no extension, the default extension will be appended to the path. If a file at the listing file path exists prior to the assembler run, the old file contents will be lost.

Copyright 2013 Cirrus Logic

1

Table 1-1 Command Line Options (continued)

Option	Description
-o<object file>	The -o directive specifies the location of the object file to the assembler. If this option is not specified in the command line, the assembler will make an object file using the root of the source file path and the default extension .O. Similarly, if <object file> has no extension, the default extension will be appended to the path. If a file at the object file path exists prior to the assembler run, the old file contents will be lost. The object file is used exclusively by the CLINK linker.
-s	Add local symbols in the object file.

1.3.3 Command Line Examples

Example 1:

```
casm maketab.a -l --debug -dTABSIZE=128 -
i%tools%\CS498xx\common\inc\base.h
```

This example does the following:

- Assembles the file *maketab.a* in the working directory.
- Employs the definition file *base.h* that is appropriate for the CS498xx DSP.
- Define the symbol TABSIZE in the assembler and set the symbol to 128.
- Make listing file *maketab.lst*.
- Include debug information in the output file.
- Implicit make object file *maketab.o.*

Example 2:

```
casm c:\mycode\hiworld –ibase.h –dBYTEALIGN –ogreet.obj
```

This example does the following:

- Assembles file `c:\mycode\hiworld.a`.
- Employs definition file *base.h.*
- Defines the symbol BYTEALIGN in the assembler with no associated value.
- Makes the object file *greet.obj* in the working directory.

Copyright 2013 Cirrus Logic

1.4 Assembly Language Format

An assembly language file is a text file parsed as a series of lines. Each line is terminated with a new line character. Each line contains at most one instruction or assembler directive. A line may contain all white space characters or consist entirely of comments.

1.4.1 Code Line Format

A line containing an instruction or assembler directive must adhere to the following format:

```
<symbol or white space> <instruction or directive>
```

The first character of a code line is significant. If the initial character of <symbol or white space> is not white space, a symbol is defined and its value is set to either the current value of the address counter or the value of the assignment assembler directive. A symbol definition ends with either a white space character or the special character ':'. See Section 1.4.4 for more on symbol definitions. If the initial character of <symbol or white space> is white space, no symbol is defined for the current address.

<instruction or directive> follows the valid syntax of an operation or assembler directive. Executable operations are covered in Chapter 2, Chapter 3, Chapter 4, and Chapter 5 of this manual. Assembler directives are covered in detail in Section 1.4.16 of this chapter.

1.4.2 Comment Character

The comment character in this assembler is '#'. The assembler ignores the text from the comment character to the end of the line. The comment string appears in the listing file.

1.4.3 Case Sensitivity

The assembler is not case sensitive unless the -c directive is used to enable case sensitivity. The single exception to this rule is text between string delimiters. Case sensitivity should be used whenever C-language source files are involved in a project. Even if there is a single C-language file in a project, it is a good idea to enable case sensitivity for every assembler file, too. See Section 1.4.7 on page 1-7 for more information on string definition.

1.4.4 Symbol Definition

A valid assembler symbol must start with either an alphabetic character or the special character '_'. Each symbol character after the first is either alphabetic (['a'...'z'] or ['A'...'Z']), numeric (['0'...'9']), or '_'. Placing the symbol string at the beginning of a line registers the string in the assembler symbol table and associates the symbol with either the address counter value or the value of the assignment assembler directive on the code line. Placing the symbol string anywhere else in a line causes the assembler to search for the symbol in its symbol table. If the assembler finds a match in the table, the value of the symbol is evaluated in the context of the code line. If the assembler does not find a match, an assembler error is generated.

There are a few assembler keywords that cannot be used as symbols. All register names and memory reference labels are reserved (such as a0 and xmem).

Copyright 2013 Cirrus Logic

1.4.5 Local Symbol Definition and Use

Symbols beginning with '%', '>', or '<' characters are called local symbols. These are used in situations where the value of such a symbol is valid only for a brief period. Local symbols are not recorded in the assembler symbol table. These symbols are not visible to the linker or debugger.

To reference a defined local symbol, the local symbol name must be prefixed with either '<' or '>', depending on where the local symbol is defined. A local symbol reference starting with the special character '>' instructs the assembler that the value of the local symbol will be defined later on in the code (a forward reference). A local symbol reference starting with the special character '<' instructs the assembler that the value of the local symbol has already been defined (a backward reference).

The following code example employs local variables to implement a C-like While loop without having to define any permanent symbols.

```
# while (clause) {
#body
# }
%whiletop:
# calculate value of clause (zero or non-zero)
if (a==0) jmp>whileend
#loop body
jmp<whiletop
%whileend:
```

If there are multiple (non-nested) instances of this while structure in the same assembler file, the local variables can be reused without causing a symbol redefinition error.

References to local symbols are different inside of macro bodies. See Section 1.4.17 for more information on local variables in macro definitions.

1.4.6 Expressions

Quantities to be evaluated at assembly time, such as addresses, conditionals, and numerical values, are cast as expressions. Expressions in the assembler are composed using infix notation, with binary operators between their operands. Therefore, a hierarchy of operator precedence is necessary to establish an order of operator evaluation. Parenthetical characters '(' and ')' provide an escape from evaluation difficulties. See Table 1-5 for examples.

Externally defined symbols (*.extern*) can be used in expressions.

```
extern FOO
.xdata
BAR .dw FOO * 5
```


Copyright 2013 Cirrus Logic

1.4.6.1 Floating-point Expressions

In an expression, any operation involving a floating point operand is promoted to float type.

Use of a floating point value in an address expression or a context requiring an integer expression is disallowed.

Assembly time intermediate floating point computations are performed in double precision float format or higher.

Float values are implicitly emitted as 32-bit fixed point.

An error is posted if the final value of a floating point expression is not in the [–1.0, 1.0] range.

1.4.6.2 Address Expressions

Any expression involving a relocatable symbol, such as a label, is termed an address expression. The value of an address expression must be less than 2^16. Address expressions are limited in a fashion similar to C pointer arithmetic. For example:

- Legal address expressions:

```
<address> + <integer>
<address> - <integer>
<address> - <address>
```

- Illegal address expressions:

```
<address> + <address>
<address> <any-op> <float>
```

1.4.7 Constants

There are three types of literal values in the assembler: floating point, integer, and string.

1.4.7.1 Floating Point Literals

A floating point literal is expressed as:

```
<mantissa>['E'|'e'<exponent>]
```

where

```
<mantissa> ::= <digits>['.'<digits>]
<exponent> ::= [[sign]<digits>]
```

Examples:

1.0

0.2e-10

99e20

Copyright 2013 Cirrus Logic

1.4.7.2 Integer Literals

Integer literals may be specified in any of the four commonly-used radices: hexadecimal, decimal, octal, and binary. Hexadecimal radix integer literals are composed of the digits 0-9 and the characters A-F or a-f. Decimal literals may use the digits 0-9. Octal literals may use the digits 0-7. Binary literals may use only the digits 0 and 1.

There are both prefix and postfix methods of radix specification for integer literals, absence of a radix specification defaults to decimal.

1.4.7.2.1 Prefix Radix Specification

hexadecimal: 0x or 0X followed by <hex-digits>

binary: 0b or 0B followed by <binary-digits>

Examples:

```
0xffff
0b010101010
12345
```

1.4.7.2.2 Postfix Radix Specification

Hexadecimal, decimal, octal, and binary numbers can be specified with a postfix radix specification character as follows:

```
hexadecimal: <hex-digits>('H'|'h'|'X'|'x')
decimal: <digits>('D'|'d'|nil)
octal: <octal-digits>('Q'|'q'|'O'|'o')
binary: <binary-digits>('B'|'b')
Examples:
0ffffH
ffffX
777Q
101010B
```

1.4.7.3 String Literals

Strings are consecutive text characters between the string delimiter character, which can be either a single quote or double quote. The ending delimiter character must match the starting delimiter character. Two quotes in a row, within a string, are treated as a single quote character to be added to the contents of the string.

Copyright 2013 Cirrus Logic

1.4.8 Unary Operators

Unary operators apply to only one operand. The target operand is the value or expression to the immediate right of the operator. The unary operator characters for this assembler are as follows.

Table 1-2 Unary Operators

Operator	Description
+	Unary plus, operand unchanged
-	Unary minus, operand arithmetically negated
~	Complement, operand logically negated
!	Not, operand boolean-wise (zero for false, non-zero for true) negated

1.4.9 Binary Operators

Binary operators apply to two operands, the values or expressions to the immediate left and right of the operator. The binary operators for this assembler are as follows.

Table 1-3 Binary Operators

Operator Type	Operator	Description
Arithmetic	+	Add
	-	Subtract
	*	Multiply
	/	Divide
Logical	&	And
	|	Or
	^	Exclusive or
Comparison (evaluates to boolean (zero for false, -1 for true)	=	Is equal to
	!=	Is not equal to
	<>	Is not equal to
	>	Is greater than
	>=	Is greater than or equal to
	<	Is less than
	<=	Is less than or equal to

NOTE: String operands that do not evaluate to numbers can only have the comparison operators applied to them.

Copyright 2013 Cirrus Logic

1.4.9.1 Precedence of Operators

An expression is evaluated by first evaluating any parenthetical sub-expressions encountered. Then all operators are evaluated in the order of precedence, the highest precedence operators performed first, and the lowest precedence operators performed last. The assembler precedence of operators is summarized in Table 1-4.

Table 1-4 Precedence of Operators

Precedence	Description
6 (highest)	unary +, unary -, ~, !
5	*, /
4	binary +, binary -
3	=, !=, <>, >, >=, <, <=
2	&
1 (lowest)	|, ^

1.4.10 Expression Examples

Table 1-5 shows examples for expressions, precedence of operators, and what the expressions evaluate to.

Table 1-5 Expression Examples

Expression	Evaluation
32+3*(20-4)	`32+(3*16) ’ 32+48 ’ 80`
77/6*6+mod(77,6)	`(77/6)*6+mod(77,6) ’ (12*6)+mod(77,6) ’ 72+5 ’ 77`
strcat(“moo”,‘cow’)=“moocow”	`“moocow”=”moocow” ’-1 (true)`
14>=0&14<10	`(14>=0)&14<10 ’ -1&(14<10) ’ -1&0 ’ 0 (false)`
240&0x3f | 240&0x7f00	`(240&0x3f) | 240&0x7f00 ’ 48 | (240&0x7f00) ’ 48 | 0 ’ 48`
!0&9-12<>0^-64	`(!0)&9-12<>0^(-64) ’ -1&(9-12)<>0^-64 ’ -1&(-3<>0)^-64 ’ (-1&-1)^-64 ’ -1^-64 ’ 63`

1.4.11 Built-in Functions

There are a number of built-in functions in the assembler that assist the programmer in the configurability of code. These functions are presented in Table 1-6.

Table 1-6 Built-in Functions

Function	Description
.defined(<expression>)	This function returns zero (false) if the argument expression contains an undefined symbol, non-zero (true) otherwise.
.isabsolute(<expression>)	This function returns non-zero (true) if the expression evaluates to a numeric value or an absolute address, zero (false) otherwise.
.isfloat(<expression >)	Returns non-zero (true) if *expression* is a floating point quantity, zero (false) otherwise.
.isint(<expression>)	Returns non-zero (true) if *expression* is an integer quantity, zero (false) otherwise.
.isstring(<expression>)	This function returns non-zero (true) if the argument expression evaluates to a character string, zero (false) otherwise.

Copyright 2013 Cirrus Logic

Table 1-6 Built-in Functions (continued)

Function	Description
.classname(<address expression>)	This function returns a string representing the memory type in which the input address expression resides. The possible outputs are "X" for X memory, "Y" for Y memory, "L" for XY memory, and "CODE" for code memory.
.typename(<expression>)	This function returns the string representing the type of the expression. Where 'typename' is one of "FLOAT", "NUMBER", "STRING", "ADDRESS", "UNDEFINED", "ERROR", "EXTERNAL" or an enumerated type name.
.segname(<address expression>)	This function returns the name of the segment in which the input address expression resides. Segment names are defined when segments are declared. See Section 1.4.16.2, "Memory Segments" on page 16.
.segaddr(<address expression>)	This function returns the base address of the memory segment in which the input address expression resides. **Note:** .segaddr(<address expression>) + .offset(<address expression>) = <address expression>).
.offset(<address expression>)	This function returns the offset from the segment base address in which the input address expression resides, zero if undefined.
.filename()	This function returns a character string representing the full path of the file being assembled.
.linenumber()	This function returns the line number in the assembler file in which this function call resides.
.timestamp()	This function returns a character string representing the time this assembler run began, in format "MM-DD-YY HH:MM:SS".
.linecount()	Returns the total number of source lines read.

1.4.12 Mathematical Functions

There are a number of mathematical functions in the assembler. These functions are presented in Table 1-7.

Table 1-7 Mathematical Functions

Function	Description
.mod(<expression1>, <expression2>)	This function returns <expression1> modulo <expression2>, or the remainder after division of <expression1> by <expression2>.
.shl(<expression1>, <expression2>)	This function returns <expression1> left-shifted by <expression2>, or <expression1> multiplied by 2-to-the-power-of-<expression2>. <expression2> must be in the range [0...31].
.shr(<expression1>, <expression2>)	This function returns <expression1> arithmetically right-shifted by <expression2>, or <expression1> multiplied by 2-to-the-power-of-minus-<expression2>. <expression2> must be in the range [0...31]. An arithmetic right shift implies that the arithmetic sign of <expression1> is preserved.
.abs(<expression>)	Returns the absolute value of expression. Return datatype is same as expression datatype.
.acos(<expression>)	Returns the arc cosine of *expression* in radians as a floating point value in the range zero to pi. <*expression*> must be in the range [–1…1].
.asin(<expression>)	Returns the arc sine of <*expression*> in radians as a floating point value in the range [–pi/2...pi/2]. <*expression*> must be in the range [–1…1].
.atan(<expression>)	Returns the arc tangent of <*expression*> in radians as a floating point value in the range [–pi/2...pi/2].
.cos(<expression>)	Returns the cosine of <*expression*> (given in radians) as a floating point value.

Copyright 2013 Cirrus Logic

1

Table 1-7 Mathematical Functions (continued)

Function	Description

.exp(<expression>)	Returns the natural exponential (base e raised to the power of <*expression*>) as a floating point value.
	Example:
	`EXP1 .EQU .EXP(1.0) # EXP1 = 2.718282`
	`.log(<expression>)`
	Returns <*expr1*> raised to the power <*expr2*> as a floating point value.
	Example:
	`BUF .EQU .CVI(.POW(2.0,3.0)) # BUF = 8`

.log(<expression>)	Returns the natural logarithm of <*expression*> as a floating point value. <*expression*> must evaluate to an integer or floating point value greater than zero.
	Example:
	`LOG .set .LOG(100.0) # LOG = 4.605170`
	`.log10(<expression>)`

.log10(<expression>)	Returns the base 10 logarithm of <*expression*> as a floating point value. <*expression*> must evaluate to an integer or floating point value greater than zero.
	Example:
	`LOG .EQU .LOG10(100.0) # LOG = 2.0`
	`.log2(<expression>)`

.log2(<expression>)	Returns the base 2 logarithm of <*expression*> as a floating point value. <*expression*> must evaluate to an integer or floating point value greater than zero.
	Example:
	`LOG .EQU .LOG2(8.0) # LOG = 3`
	`.max(<expr1>,<expr2>[,...,<exprN>])`

.max(<expression>)	Returns the greatest of <*expr1*>,...,<*exprN*>. Expressions must be numerical. If all expressions are integral type, the return value is an integer. Otherwise a floating point value is returned.
	Example:
	`MAX .set .MAX(1.0,5,-3) # MAX = 5.0 (floating point)`
	`.min(<expr1>,<expr2>[,...,<exprN>])`

.min(<expression>)	Returns the least of <*expr1*>,...,<*exprN*>. Expressions must be numerical. If all expressions are integral type, the return value is an integer. Otherwise a floating point value is returned.
	Example:
	`MIN .set .MIN(1.0,5,-3.25) # MIN = -3.25`
	`.pow(<expr1>,<expr2>)`

.sign(<expression>)	Returns the sign of <*expression*> as an integer: -1 if <*expression*> is negative, 0 if zero, 1 if positive. Example: `.if .SIGN(INPUT)>=0 # is INPUT non-negative?`
.sin(<expression>)	Returns the sine of <*expression*> (given in radians) as a floating point value.
.sqrt(<expression>)	Returns the square root of <*expression*> as a floating point value. <*expression*> must be positive. Example: `SQRT .EQU .SQRT(3.5) # SQRT = 1.870829`

Copyright 2013 Cirrus Logic

Table 1-7 Mathematical Functions (continued)

Function	Description
.tan(<expression>)	Returns the tangent of <*expression*> (given in radians) as a floating point value. Example: `TANGENT .set .TAN(1.0) # TANGENT = 1.557408`

1.4.13 Conversion Functions

There are a number of conversion functions in the assembler. These functions are presented in Table 1-8.

Table 1-8 Conversion Functions

Function	Description
.b2f(<expression>)	Converts <*expression*> to a floating point value. <*expression*> should represent a binary fraction. Example: `FRC .EQU .B2F(0x40000000) # FRC = 0.5`

.f2b(<expression>)	Obtain the fractional representation of the floating point <*expression*> as an 32-bit integer.
	Example:
	`FRAC1 .set .F2B(0.5) # FRAC1 = 0x40000000`
	`FRAC2 .set .F2B(1.5) # error!`
	Example:
	`THREE_PT_98_IN_9_23 .equ .F2B(3.98*.pow(2,-8.0))`
	`#THREE_PT_98_IN_9_23 = 0x01fd70a4`

.f2i(<expression>)	Converts <*expression*> to an integer value. Any fractional portion of <*expression*> is removed (truncated). Example: `INT .set .F2I(-1.05) # INT = -1`
.i2f(<expression>)	Converts <*expression*> to a floating point value. Example: `FLOAT .set .I2F(5) # FLOAT = 5.0`
.ceil(<expression>)	Returns an integer value that represents the smallest integer greater than or equal to <*expression*>.
.floor(<expression>)	Returns an integer value which represents the largest integer less than or equal to <*expression*>. Example: `FLOOR .SET .FLOOR(2.5) # FLOOR = 2`

.round(<expression>)	Adds 0.5 to <*expression*> then converts to an integer as per .F2I or .FLOOR.
	Example:
	`round1 .set .ROUND(1.5) # round1 = 2`
	`round2 .set .ROUND(1.48) # round2 = 1`

Copyright 2013 Cirrus Logic

1.4.14 Loading Immediate Values to Registers

The Cirrus Logic Assembly Program (CASM) allows the developer to load 16 bits at a time into a register. When the developer needs to load more than 16 bits into a register, the developer must use multiple lines of code. Below are some examples of loading more than 16 bits into a register. The first example uses two processing cycles and no data memory, and the second example uses one processing cycle and one word of data memory. It is often more desirable to conserve processing cycles than it is to conserve memory. Therefore, the second example is generally, but not always, recommended over the first example.

Example 1: Load 3.98 as a Q.N(9.23) fixed-point number into the into accumulator, a1.

1. Define the symbol "THREE_PT_98_IN_9_23" as shown below:

```
THREE_PT_98_IN_9_23 .equ .f2b(3.98*.pow(2,–8.0))
```

2. Load 16 bits at a time into the high part of the accumulator:

```
a1 = (THREE_PT_98_IN_9_23>>16) # 3.98 in Q9.23 #a1 = 00 01fd0000 00000000
lo16(a1) = (THREE_PT_98_IN_9_23 & 0x0000FFFF) #a1 = 00 01fd70a4 00000000
```

Note: Example 1 uses two processing cycles and no data memory. The following macro could be used to implement the first example.

```
.macro
FLOAT_2_FIXEDQNM_REG_LOAD %intval, %fracval, %n, %m, %reg
# inputs:
# intval = integer part of float value
# fracval = fractional part of float value
# n = number of bits for (signed) integer part
# m = number of bits for fractional part
# reg = destination register (32 bit data register or accumulator)
%reg = (.shr(.f2b(%intval.%fracval*.pow(2,–(%n.0–1.0))),16))
lo16(%reg) = (.f2b(%intval.%fracval*.pow(2,–(%n.0–1.0)))&0xffff)
.endm

.code_ovly
FLOAT_2_FIXEDQNM_REG_LOAD 3, 98, 9, 23, a1
#### The above macro expands to
# a1 = (.SHR(.F2B(3.98*.POW(2,–(9.0–1.0))),16))
# LO16(a1) = (.F2B(3.98*.POW(2,–(9.0–1.0)))&0XFFFF)
```

Example 2: Load 3.98 as a Q.N (9.23) fixed-point number into the accumulator, a0.

```
.xdata_ovly
THREE_PT_98_IN_9_23 .equ .f2b(3.98*.pow(2,-8.0))
X_THREE_PT_98_IN_9_23 .dw (THREE_PT_98_IN_9_23)
.code_ovly
a0 = xmem[X_THREE_PT_98_IN_9_23] #a0 = 00 01fd70a4 00000000
```

Note: Example 2 uses one processing cycle and one word of data memory.

Copyright 2013 Cirrus Logic

1.4.15 String Functions

There are a number of conversion functions in the assembler. These functions are presented in Table 1-9.

Table 1-9 String Functions

Function	Description
.strpos(<str1>,<str2>[,<start>])	Returns the 0 based position (0 = first character of the <*str1*>) of string <*str2*> in <*str1*> as an integer, starting at position <*start*>. If <*start*> is not given the search begins at the first character of <*str1*>. If the <*start*> argument is specified it must be a positive integer. If <*str2*> is not found in <*str1*>, the length of <*str1*> is returned. Example: `ID .EQU .STRPOS('CS18101','18') # ID = 2`
.strcmp(<str1>,<str2>)	Lexicographical string comparison (case sensitive). Returns an integer 0 if the two strings are the same, 1 if <*str1*> is greater than <*str2*>, -1 if <*str1*> is less than <*str2*>. Example: `.IF .STRCMP(STR,'MAIN') # does STR differ from “MAIN”?`
.strcat(<string1>, <string2>[, … <stringN>])	This function returns the string arguments concatenated into a single string.
.strlen(<string>)	This function returns the length of the argument character string (<*string*>).
.streval(<string>)	This function takes its character string argument and evaluates <*string*> as an expression. The function returns the value of the expression.
.substr<string>, <startexpression>[,<length-expression>])	Return the substring starting at <*start-expression*> until the end of the string, or <*length-expression*>+<*start-expression*>. <*start-expression*> is a 1-based index into the string. If <*length-expression*> is omitted, the substring from <*start-expression*> to the end of the string is returned.
.numtostr(<expression>) or .str(<expression>)	This function evaluates <*expression*> and returns its numerical value as a decimal character string.
.numtostrx(<expression>) or .strx(<expression>)	This function evaluates <*expression*> and returns its numerical value as a hexadecimal character string.

1.4.16 Assembler Directives

Assembler directives have a format similar to instructions, but they are instructions to the assembler, not intended for the target DSP. Instructions or data memory locations may be generated as a side effect, but not necessarily. The assembler directives assist the programmer in controlling and configuring code.

There are directives that allow separate assembler files to share objects. Other directives allow for allocation and initialization of data memory. There are directives for conditional assembly allowing for optional features without writing multiple versions of the same code. Finally, there are macros, which help the programmer encapsulate often-used or iterated blocks of code.

Copyright 2013 Cirrus Logic

1.4.16.1 Code Modularity

Macros used to enable the programmers to build modularity into their code are presented in Table 1-10:

Table 1-10 Macros

Macro	Description
.include <file string>	This directive opens the file with path <file string>, inserts, and assembles the file contents where the directive resides. There is no default extension to an include file. If there is no extension to <file string>, the assembler will attempt to open the file path with no extension. The include file can itself contain .include directives with no nesting limit. An include file can be shared among several assembler files to define common constants, expressions, and macros.
.public <symbols>	This directive allows the specified list of symbols to be referenced by other assembler files. <symbols> contains one or more symbols separated by commas, that are defined in the course of assembly of this file.
.extern <symbols>	This directive tells the assembler that the specified list of symbols is defined with .public directives in other assembler files. <symbols> contains one or more symbols separated by commas, defined in other assembler files. If an .extern symbol is defined in the course of assembly of this file and declared to be .public, the .public declaration takes precedent. If an .extern symbol is not defined in any object file, the linker will produce an unresolved external error at link time. An .extern symbol cannot be used in an expression.
.end	This directive tells the assembler that there are no more lines to assemble in this file. Further assembly of this file is halted and the file is closed. Use of this directive is optional, as this directive is equivalent to a physical end of file. The.end directive may be used to hide text below the directive from the assembler.
.export <symbols>	This directive allows the specified list of symbols to be exported to a *.xo file at link time and to be referenced by other assembler files. <symbols> contains one or more symbols separated by commas that are defined in the same assembly file. Casm does not report an error if both .public and .export macros are used for the same symbol. External symbols cannot be exported, so an error is reported in this case.

1.4.16.2 Memory Segments

This segment directive instructs the assembler to put subsequent instructions or data declarations in the memory specified by <segment-class>.

```
[<name>] segment <class_name> [at <address> | align <modulo>] OR
[<name>] .<segment-class> [at <address> | align <modulo>]
```

Use .xdata_ovly, .ydata_ovly, .data_ovly, and .code_ovly. For example:

MAIN_XDATA_ALIGN16 segment "X_OVLY" align 16

MAIN_XDATA_ALIGN32 .xdata_ovly align 32

SAMPLE_MCV .ydata_ovly

MAINCODE .code_ovly

MY_64_BIT_XY_DATA_SEGMENT .data_ovly

The segment definition ends with either the next segment directive or the end of assembly. All consecutive declarations within a segment will appear contiguously in the final memory map produced by the linker.

<name> is an optional parameter. <name> may be the same as used on other segment declarations in which case the new declaration will be concatenated with the previous one(s) with like <name> parameters. The ordering of these concatenations is indeterminate.

If the *at* form of the directive is used, the first memory location following the directive will be placed at the absolute <address> specified. Only one declaration with *at* form is allowed for a given segment <name>. If such a declaration exists, it determines the starting point for the

Copyright 2013 Cirrus Logic

segment. All segments using the same <name> specification will be concatenated and follow the segment generated with the *at* specification.

Use of the *align* form of the segment declaration will instruct the linker to align the first memory location following the directive to the given address <modulo>. Only one declaration with *align* form is allowed for a given segment <name>. All segments using the same <name> specification will be concatenated and follow the segment generated with *align* specification.

The *at* and *align* forms of the segment declaration are mutually exclusive and may not be used together.

1.4.16.3 Symbol Assignment

The following directives are used to assign values to symbols.

Table 1-11 Symbol Assignment

Directive	Description
<symbol> .equ <expression>	The .equ directive can only be applied to a unique symbol one time in an assembly, meaning that the symbol must not be previously defined or redefined in the assembly. There may be no forward references to symbols in <*expression*>. `Example: uyequ .equ (0x1)`
<symbol> .set <expression>	This directive assigns the value of <*expression*> to the symbol defined in this code line. The symbol can be redefined later in the assembly with other instances of the .set directive. <*expression*> in this directive array contains forward references.

1.4.16.4 Data Memory Assignment

Data memory directives are described Table 1-12:

Table 1-12 Data Memory Assignment

Directive	Description

.bss <expression>	No allocation performed. This directive instructs the linker that <*expression*> words are reserved.
	Example:
	`.xdata`
	`.bss (0xff)`

.bsc <count>,<value>?	This directive instructs the assembler to allocate <*count*> words in the current memory segment. All words allocated are initialized to the value of <*value*>. If <value> is not used, then there is no allocation. Memory location is reserved.
	Example:
	`.data`
	`.bsc (0xf),0x8`

Copyright 2013 Cirrus Logic

1

Table 1-12 Data Memory Assignment (continued)

Directive	Description

.dcb <arg>[, <arg>, …]	Define constant byte(s). <*Arg*>(s) may be strings or integers. Strings are not implicitly null terminated. Data is stored in big endian order and zero padded to the next 32-bit word boundary.
	Example:
	`.dcb 0x30,0x31,0x32,0x33,0x34`
	`.dcb 0x35,0x36,0x37,0x38,0x40`
	`.dcb 'R'`
	`.dcb 'CirrusLogic',0 # explicit null termination`
	`.dcb 'four' # no null termination`
	`#Generates the following memory image:`
	`X 0000 30313233 34000000 35363738 40000000`
	`X 0004 52000000 43697272 75734C6F 67696300`
	`X 0008 666F7572`

.dh <arg>[,<arg, …]	Define constant 16-bit halfword(s). <*Arg*>(s) must be in the range 0…65535 to fit in 16-bit storage. Data is stored in big endian order and zero padded to the next 32-bit word boundary.
	This construct is useful for packing a 16-bit address expression and a 16-bit integer expression in the same 32-bit word.
	Example
	`.dh 0x2d,0x123,0x2d,0x1111`
	`#Generates the following memory image:`
	`0x002d0123`
	`0x002d1111`

.dw <expression>
.dw <expression1>,<expression2>
	These directives instruct the assembler to allocate one word in the current memory segment. If the memory is X or Y, the first form must be used, and the word allocated is initialized to the value of <expression>. If the memory is X-Y, the second form must be used. The X word allocated is initialized to the value of <expression1>, and the Y word allocated is initialized to the value of <expression2>.
	Example 1:
	`.xdata`
	`.dw (0x12345678)`
	
	Example 2:
	`.data`
	`.dw (0x1),(0x2)`

.dd <expression>	This directive instructs the assembler to allocate one word in the current memory segment, which must be in XY. The X word allocated is initialized to the most significant bits of <expression>, and the Y word allocated is initialized to the least significant bits of <expression>.
	Example:
	`.data`
	`.dd (0x12345678ABCDEF00)`

Copyright 2013 Cirrus Logic

1.4.16.5 Conditional Assembly

Conditional assembly directives are described Table 1-13.

Table 1-13 Conditional Assembly Directives

Directive	Description
.if <expression>	This directive evaluates <expression> (<expression> must not contain a forward reference). If the value is true (non-zero), the lines just below this directive are assembled to the next instance of the .elseif, .else, or .endif directive. If the directive is .elseif or .else, the assembler will skip to the following .endif directive. Normal assembly continues after the .endif directive. If the value is false (zero), the assembler will skip to the corresponding .elseif, .else, or .endif directive. The directive encountered is then executed. An .if directive begins an .if block. A corresponding .endif directive must follow the .if directive. An .if block does not require either ann .elseif directive or an .else directive.
.elseif <expression>	This directive functions in the same manner as the .if directive above. However, this directive must be contained within an .if block, that is, after an .if directive and before its corresponding .endif directive. In addition, the .elseif directives in an .if block must all occur before a corresponding .else directive.
.else	This directive, if encountered when the .if directive and all .elseif directives in the .if block have evaluated to false (zero), instructs the assembler to assemble lines just below this directive until an .endif directive is encountered.
.endif	This directive indicates the termination of an .if block. Normal assembly continues after this line.

Copyright 2013 Cirrus Logic

1

Table 1-13 Conditional Assembly Directives (continued)

Directive	Description

.if	There are four basic configurations of an .if block:
	? if Configuration I
	`.if <exp>`
	`<code>`
	`.endif`
	? if Configuration II
	`.if <exp>`
	`<code0>`
	`.else`
	`<code1>`
	`.endif`
	? if Configuration III
	`.if <exp0>`
	`<code0>`
	`.elseif <exp1>`
	`<code1>`
	`# optional additional .elseif clauses`
	`.endif`
	? if Configuration IV
	`.if <exp0>`
	`<code0>`
	`.elseif <exp1>`
	`<code1>`
	`.dw (0x1),(0x2)optional additional`
	`.elseif clauses`
	`.else`
	`<codeN>`
	`.endif`
	The conditional expressions are evaluated until one is found to be true. The code segment corresponding to this conditional is assembled. If no conditional expression evaluates to true, and an .else directive exists in this block, the code segment corresponding to the .else directive is assembled.
	It is possible to nest .if blocks. The entire nested .if block, from the .if directive to the .endif directive, must reside in one <code> section, meaning that the .if directive cannot reside in <codeN> and its corresponding .endif directive reside in <codeN+1>.

1.4.16.6 Token Substitution

.def <token> <substitution>

Defines a token substitution. All occurrences of <*token*> are replaced with <*substitution*> for the remainder of the source file or until a matching .undef statement is encountered.

<*token*> may be any text that is not already known as a symbol to the assembler.

<*substitution*> may be any arbitrary text delimited by end of line or start of comment.

Example:

```
.def STACK_PTR i7
x0 = xmem [ STACK_PTR ]
```


Copyright 2013 Cirrus Logic

.undef <token>

Undefines previously defined token (defined with .def). The token is undefined until end of source code or until it is redefined with .def.

Example:

```
.def STACK_PTR i7
if .defined(STACK_PTR)
x0 = xmem[STACK_PTR]
.endif
.undef STACK_PTR
x0 = xmem[STACK_PTR] # <- CASM report error!
```

1.4.16.7 Listing and Message Control

The following directives have no effect if the –l switch does not appear in the command line. Omitting the –l switch specifies that no listing file will be created.

- .list <switches>

This directive controls the format of the listing file. <switches> contains one or more listing control switches, separated by spaces. The formats of the switches are as follows:

Table 1-14 Listing Control Switches

Directive	Description
off	This switch inhibits listing output.
on	This switch activates listing output, or reverses the action of the off switch.
cond +cond –cond	These switches control the listing of code blocks skipped over during .if block processing. '+' or no prefix will list the skipped blocks (the default behavior), and the '-' prefix will not list the skipped blocks.
mac +mac –mac	These switches control the listing of macro expansion. '+' or no prefix will list the expansion lines (the default behavior), and the '-' prefix will not list the expansion lines.
inc +inc –inc	These switches control the listing of include files. '+' or no prefix will list the contents of include files (the default behavior), and the '-' prefix will not list the include files.
sym +sym –sym	These switches control the listing of the symbol table. '+' or no prefix will list the symbol table (the default behavior), and the '-' prefix will not list the symbol table.
gensym +gensym –gensym	These switches control the listing of internally generated symbols in the symbol table. '+' or no prefix will list internal symbols in the symbol table, and the '-' prefix will not list internal symbols in the symbol table (the default behavior).
allsym +allsym –allsym	These switches control the listing of internally reserved symbols in the symbol table. '+' or no prefix will list reserved symbols in the symbol table, and the '-' prefix will not list reserved symbols in the symbol table (the default behavior).
.page .page <expression>	This directive controls pagination of the listing file and the page size. The first form, without an argument, causes a page advance in the listing. The second form establishes <expression> as the number of lines per page. The default number of lines per page is 60.
.title <string>	This directive instructs the assembler to print <string> at the top of every listing page. This directive must be employed only per assembly.
.subtitle <string>	This directive instructs the assembler to print <string> under the title line of subsequent listing pages. <string> will be printed on the subtitle line of the current listing page if the directive is encountered within the first four lines of the page. This directive may be employed more than once or not at all.

Copyright 2013 Cirrus Logic

Table 1-14 Listing Control Switches

Directive	Description
.error <strings>	This directive instructs the assembler to print an error message on both the console output and the listing file. <strings> consists of one or more string expressions, separated by commas, that are concatenated together to create the message.
.message <strings>	This directive instructs the assembler to print an message on the console output but not the listing file. <strings> consists of one or more string expressions, separated by commas, that are concatenated together to create the message.

1.4.16.8 Assembler Warning/Error Control

The assembler emits warnings and errors for certain combinations of target DSP instructions. The *.pragma* directive allows some control over what conditions are considered errors and warnings. The syntax of the *.pragma* directive is:

```
.pragma enable:<condition> [,<condition>...]
.pragma disable:<condition> [,<condition>...]
```

The possible condition values are:

- OSp

If enabled, the use of registers *i8-i11*, *nm8-nm11*, *iic_mask*, and *iic_addr* is allowed. Otherwise, such use produces an error because these registers are reserved for use by the DSP operating system.

The errors or warnings produced are:

- LOAD_DELAY_AS_WARNINGp

  If enabled, the use of an index register immediately after it is loaded with a constant will be treated as a warning condition. Otherwise, such an instruction sequence produces an error.

- GLOBAL_MEMp

  If enabled, the standard memory location directives (*.data*, *.code*, *.ydata*, *.xdata*) are allowed. Otherwise use of those directives is an error, and code should use the application-specific memory segment directives (such as *.xdata_ppm*)

- CODE_IN_DATAp

  If enabled, executable instructions found in data segments are allowed. Otherwise, such instructions cause an error.

- DATA_IN_CODEp

  If enabled, data definition directives (such as *.dw*) found in code segments are allowed. Otherwise, such instructions cause an error.

Copyright 2013 Cirrus Logic

1.4.16.9 Data Structure Types

CASM provides a way of grouping variables into structures, similar to structures in the C programming language. Elements of structures are referred to as "members". Structure members can be initialized at structure type definition time or at structure instantiation time. Values specified for structure members at structure type definition time propagate to each instance of the type unless overridden by the instance definition. Not all structure members need to be initialized, but a member initialized at instantiation time may not be preceded by an uninitialized member. Any of the data memory directives described in Section 1.4.16.4 may be used to define structure type members.

Structure type definition syntax:

```
<struct type name>     .struct
<element1>             <data memory directive>
<element2>             <data memory directive>
<element3>             <data memory directive>
...
<elementn>             <data memory directive>
                       .endstruct
```

Structure instantiation syntax:

```
<label>         <struct type name> (<initial_value1>,...,<initial valuen>)
```

Accessing the structure members from code:

- Direct structure member access:

```
 a0 = xmem[<struct symbol>.<element>]
```

- Indirect structure member access may be performed by adding the offset of the member inside the structure to the structure's address:

```
i0 = (<struct symbol 1>)
i1 = (<struct symbol 2>)
a0 - a1
if (a > 0) jmp >
         anyreg (i0, i1)
%
i0 = i0 + (<struct type>.<element>)
x0 = xmem[i0]
```

- Accessing members of an externally-defined structure:

```
<struct type> .struct
...
...
              .endstruct
        .extern <symbol> (<struct type>)
  a0 = xmem[<symbol>.element]
```

Example 1:

```
# define a structure type labeled Y_OS_GLOBAL_VARS_T
Y_OS_GLOBAL_VARS_T       .struct
_X_BY_IOBUFFER_PTRS            .bsc NUM_IOBUFFER_CHANNELS,0
```


Copyright 2013 Cirrus Logic

```
_X_VY_HOST_SAMPLERATE          .dw 0
_X_VY_Host_Output_Mode_Control .dw 0
_X_VY_IO_Free                  .dw 0
_X_VY_Autodet_config           .dw 0
_X_VY_IO_Processed             .dw 0
_X_VY_ODT_PTR                  .bsc NUM_OVLY_SLOTS,0
                           .endstruct


          .ydata_dec
# define an instance of the Y_OS_GLOBAL_VARS_T structure type:
Y_OS_GLOBAL              Y_OS_GLOBAL_VARS_T


          .code_dec
# read the value of the _X_VY_HOST_SAMPLERATE member of the Y_OS_GLOBAL
structure:
          y0=ymem[Y_OS_GLOBAL._X_VY_HOST_SAMPLERATE]
```

Example 2:

```
# define a structure type labeled EXAMPLE_STRUCT_T:
EXAMPLE_STRUCT_T      .struct
INITIALIZED_ARRAY_1   .bsc 2,0
UNINITIALIZED_WORD_2  .bss 1
INITIALIZED_ARRAY_3   .bsc 3, 1
INITIALIZED_WORD_4    .dw 0x7FFFFFFF
UNINITIALIZED_ARRAY_5 .bss 5
                        .endstruct


         .ydata_dec
# define an instance of the EXAMPLE_STRUCT_T structure type:
EXAMPLE_STRUCT         EXAMPLE_STRUCT_T


         .code_dec
# read the value of the UNINITIALIZED_WORD_2 member of the EXAMPLE_STRUCT
structure:
         y0 = ymem[EXAMPLE_STRUCT.UNINITIALIZED_WORD_2]
```

Arbitrary structure nesting is supported. All the rules regarding initialization that apply to structures without nested structures apply here as well.

Structure nesting syntax:

```
<struct type name>     .struct
<element1>             <data memory directive> or <struct type name>
<element2>             <data memory directive> or <struct type name>
<element3>             <data memory directive> or <struct type name>
...
<elementn>             <data memory directive> or <struct type name>
                       .endstruct
```


Copyright 2013 Cirrus Logic

Nested structure initialization syntax:

```
<label>     <super struct type name> (<initial_value1>,...,<initial valuen>)
```

Accessing the nested structure members from code:

- Direct nested structure member access:

```
a0=xmem[<superstruct symbol>.<substruct symbol>.<element>]
```

- Indirect nested structure member access may be performed by adding the offset of the member inside the superstructure to the structure's address:

```
i0 = (<superstruct symbol 1>)
i1 = (<superstruct symbol 2>)
a0 - a1
if (a > 0) jmp >
            anyreg (i0, i1)
%
i0 = i0 + (<superstruct type>.<substruct type>.<element>)
x0 = xmem[i0]
```

Example 1:

```
Y_OS_GLOBAL_VARS_T       .struct
_X_BY_IOBUFFER_PTRS            .bsc NUM_IOBUFFER_CHANNELS,0
_X_VY_HOST_SAMPLERATE          .dw 0
_X_VY_Host_Output_Mode_Control .dw 0
_X_VY_IO_Free                  .dw 0
_X_VY_Autodet_config           .dw 0
_X_VY_IO_Processed             .dw 0
_X_VY_ODT_PTR                  .bsc NUM_OVLY_SLOTS,0
                         .endstruct
DEMO_STRUCT_TYPE_T       .struct
_ELEMENT_0                     .dw 0
_ELEMENT_1                     .dw 0
_ELEMENT_2                    Y_OS_GLOBAL_VARS_T
_ELEMENT_3                     .dw 0
                         .endstruct
         .ydata_dec
Y_OS_GLOBAL              Y_OS_GLOBAL_VARS_T
Y_DEMO_STRUCT            DEMO_STRUCT_TYPE_T (0x1,0x2,0x3,0x4,0x5,0x6,0x7,
0x8,0x9,0xA,0xB,0xC)
```

CASM supports struct initialization with brackets. For example, Y_DEMO_STRUCT can be initialized the following way:

```
Y_DEMO_STRUCT            DEMO_STRUCT_TYPE_T
(0x1,0x2,(0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xA,0xB),0xC)
```

or

```
Y_DEMO_STRUCT            DEMO_STRUCT_TYPE_T
(0x1,0x2,(0x3,0x4,0x5,0x6,0x7,0x8),0xC)
```

In the second example, _X_VY_IO_Processed and _X_VY_ODT_PTR will have pre-defined values (0 and NUM_OVLY_SLOTS,0).

Copyright 2013 Cirrus Logic

Example 2:

```
# define various nested structures:
STRUCT_1_T              .struct
member_1                .dw 0x101
member_2                .dw 0x102
member_3                .dw 0x103
                         .endstruct

STRUCT_2_T              .struct
member_1                .dw 0x201
member_2               .dw 0x202
member_3                .dw 0x203
                         .endstruct
STRUCT_3_T                 .struct
member_1                   .dw 0x301
member_2                   .dw 0x302
member_3                   .dw 0x303
                            .endstruct

STRUCT_23_T                .struct
STRUCT_2                   STRUCT_2_T
STRUCT_3                   STRUCT_3_T
                            .endstruct

STRUCT_4_T                 .struct
STRUCT_1                   STRUCT_1_T
STRUCT_23                  STRUCT_23_T
                            .endstruct

         .ydata_dec
# define an instance of the STRUCT_4_T structure type:
STRUCT_4                  STRUCT_4_T               (0x401, 0x402, 0x403)

        .code_dec
# read the value of the member_1 member of the STRUCT_2 nested structure:
         y0 = ymem[STRUCT_4.STRUCT_23.STRUCT_2.member_1]
```

1.4.16.10 Sizeof Function

This function returns the number of words allocated for a structure or a symbol.

```
sizeof(<struct type name>)
sizeof(<struct symbol>)
sizeof(<symbol>)
```


Copyright 2013 Cirrus Logic

Examples:

```
Y_OS_GLOBAL_VARS_T       .struct
_X_BY_IOBUFFER_PTRS            .bsc NUM_IOBUFFER_CHANNELS,0
_X_VY_HOST_SAMPLERATE          .dw 0
_X_VY_Host_Output_Mode_Control .dw 0
_X_VY_IO_Free                  .dw 0
_X_VY_Autodet_config           .dw 0
_X_VY_IO_Processed             .dw 0
_X_VY_ODT_PTR                  .bsc NUM_OVLY_SLOTS,0
                         .endstruct

SYMBOL                         .equ 10 + sizeof(Y_OS_GLOBAL_VARS_T)

         .ydata_dec
Y_OS_GLOBAL Y            Y_OS_GLOBAL_VARS_T
SYMBOL_0                       .dw 12
SYMBOL_1                       .bss 10
         .code_dec

i0=(sizeof(Y_OS_GLOBAL_VARS_T))
i0=(sizeof(Y_OS_GLOBAL))
i0=(sizeof(SYMBOL_0))
i0=(sizeof(SYMBOL_1))
```

1.4.16.11 Assert Directive

This macro definition is used for debug purposes. Expression in the assert macro is evaluated as true or false. If the value of the assert expression is fale, CASM reports the error.

Usage:

```
.assert <expression>
```

Example:

```
a_t                      .struct
channel_address                .dw 0
channel_stride                 .dw 0
channel_buffer                 .bss 16
                         .endstruct
         .xdata_ovly
my_data                  a_t

         .code_ovly
...
i4 = (0)+(my_data)
# use a single index register to access the members of the structure
# make sure the type declaration doesn't violate the coding assumptions
i0 = xmem[i4]; i4+ = 1 # i0 = channel_ptr
.assert (a_t.channel_stride = (a_t.channel_address+1))
```


Copyright 2013 Cirrus Logic

```
nm4 = xmem[i4]; i4+ = 1 # nm4 = channel stride
.assert (a_t.channel_buffer = (a_t.chanel_stride+1))
#i4 = buffer
...
```


1.4.17 Macro Definition and Calling

This directive instructs the assembler to begin defining a macro:

- macro
  .macro nolist

This directive instructs the assembler to begin macro definition. Unlike other assemblers, the macro defined is not associated with a symbol defined at the beginning of this line, so there should be no symbol defined on this line. The second form of this directive inhibits the lines of the macro definition from appearing in the listing file.

The line immediately following the .macro directive contains the calling prototype. The format of the prototype line is as follows.

```
[<%symbolarg>] <name> <%args>
```

<symbolarg>, if defined, must begin in the first column of the line. This local symbol allows for the macro call to pass an argument as a symbol definition. The symbol in the macro call is not defined or re-defined, but passed as an argument to the macro. <name> is the macro name. <name> cannot begin at the first column of the line.

<%args> is an optional list of local symbols that serve as arguments to the macro. These arguments should be separated by commas. There are two ways to use commas:

- ROMCMD {"ABC",7}, ABCROUTINE
- ROMCMD "ABC"%,7, ABCROUTINE

In the first example, CASM understands that the curly braces enclose an entire parameter. In the second example, the percent sign “escapes” the comma and causes CASM to accept it as text rather than a parameter delimiter. A percent sign can also be used to “escape” a curly brace or another percent sign to get these characters accepted as text rather than special parameter syntax.

The lines after the calling prototype, up to but not including the closing .endm directive, constitute the macro body. It is possible to define symbols within the macro body, but like the arguments, all symbols in a macro must be local symbols. All references to both arguments and body symbols must start with the ‘%’ character, employing the same format as their definition. No ‘>’ or ‘<’ references are allowed in a macro body. A local symbol outside of the macro body cannot be referenced.

The ‘%’ character has other special functions within a macro body. The special functions are described in Table 1-15.

Copyright 2013 Cirrus Logic

Table 1-15 Special Characters Used in Macros

Characters	Description
%%:	Replaced with a single '%' character in the macro expansion.
%#:	Delineates a comment that will not appear in the macro expansion.
%&:	Replaced with no characters, used as a concatenation operator in conjunction with a symbol or argument reference.
%(<string>):	Replaced with the contents of <string>.

The following directives are used in defining macros:

- .endm

This directive marks the end of a .rept block or .macro body. Normal assembly resumes after this line.

- .exitm

This directive, if encountered during macro expansion, will terminate macro expansion at this point, essentially skipping to the corresponding .endm directive.

An example of a macro is given in the code example in Section 1.4.19.

1.4.18 Macro Replication

The following expression is used to replicate macros:

```
.rept <expression>
.rept %<variable>=<start>,<stop>[,<step>]
.endm
```

Create duplicates of enclosed block of source lines. <*expression*> must be a non-negative integer.

<*variable*> is assigned a value of <*start*> for the first iteration and is incremented by <*step*> (or by 1 if <*step*> is not supplied) on each successive iteration. <*variable*> does not need to be previously defined. On the last iteration, <*variable*> is equal to or less than <*stop*>.

Example 1:

```
count .set 3
    .rept count
    .dw 0x20
    .endm
```

Generates:

```
count .set 3
    .dw 0x20
    .dw 0x20
    .dw 0x20
```


Copyright 2013 Cirrus Logic

Example 2:

```
.rept %v=1,5,2
.dw %v*5+3
.endm
```

Generates:

```
.dw 8
.dw 18
.dw 28
```

Example 3:

```
    .macro
    MapDef %index,%offset
DEF_AUDIOMAP%&%(.numtostr(%index)) .set %(.numtostr(%offset))
    .endm

AUDIO_INDEX .set 1
    .rept %m=1,4
        .rept %%n=1,3
MapDef AUDIO_INDEX,%m*4+%%n
AUDIO_INDEX .set AUDIO_INDEX+1
% .endm
.endm
```

Generates:

```
DEF_AUDIOMAP1 .set 5
DEF_AUDIOMAP2 .set 6
DEF_AUDIOMAP3 .set 7
DEF_AUDIOMAP4 .set 9
...
DEF_AUDIOMAP12 .set 19
```


Copyright 2013 Cirrus Logic

1.4.19 Assembly Language Example

The following code example employs many of the assembler directives in this section, concentrating on macros and conditional assembly, to create a generic FIR filter macro.

```
.list-cond

.macro
% labelfir%tapptr, %tapbuf, %tapcir, %wtbuf, %order
# tapptr: pointer into tap buffer, string reference
# tapbuf: tap buffer
# tapcir: tap buffer circular length
# wtbuf: tap weight buffer
# order: FIR order

# make sure buffer memory is configured properly
 .ifclassname(%tapbuf) = "XYMEM" | classname(%wtbuf) = "XYMEM"
 .error "fir: tap and/or weight buffer cannot be in x-y memory"
 .exitm
 .elseifclassname(%tapbuf) = "XMEM" & classname(%wtbuf) = "XMEM"
 .error "fir: tap and weight buffer cannot both be in x memory"
 .exitm
 .elseifclassname(%tapbuf) = "YMEM" & classname(%wtbuf) = "YMEM"
 .error "fir: tap and weight buffer cannot both be in y memory"
 .exitm
 .endif

# load address registers and perform convolution depending on where circular
taps and weights are
 .ifclassname(%tapbuf) = "YMEM"
 i0 = (%wtbuf)
 i4 = %(%tapptr)
# recall the %(<string>) operator
 nm4 = (%tapcir)
 a0 = 0;x0 = xmem[i0]; i0 += 1;y0 = ymem[i4]; i4 += 1
 do (%order), %loop
%loop:a0 += x0*y0; x0 = xmem[i0]; i0 += 1; y0 = ymem[i4]; i4 += 1
 nm4 = (0)
 .else
 i4 = (%wtbuf)
 i0 = %(%tapptr)
 nm0 = (%tapcir)
 a0 = 0;x0 = xmem[i0]; i0 += 1;y0 = ymem[i4]; i4 += 1
 do (%order), %loop
%loop:a0 += x0*y0; x0 = xmem[i0]; i0 += 1; y0 = ymem[i4]; i4 += 1
 nm0 = (0)
 .endif
# new value of index register is not put back into tapptr
 .endm
```

The above macro does a simple FIR one time and checks the memory locations of the taps and tap weights to assure a fast FIR can be performed. This macro can be called as follows.

```
Mylabelfir“xmem[inptr]”, inbuffer, 128, lpfilter, 65
```


Copyright 2013 Cirrus Logic

Chapter 2

32-Bit DSP Internal Architecture and Programming Model

2

2.1 Overview

The Cirrus Logic 32-bit DSP core is a fixed point, fully programmable digital signal processor which achieves high performance through an efficient instruction set and highly parallel architecture. This device uses two's complement fractional number representation. The block diagram of the internal architecture is shown in Figure 2-1. The device has busses for two data memory spaces and one program memory space.

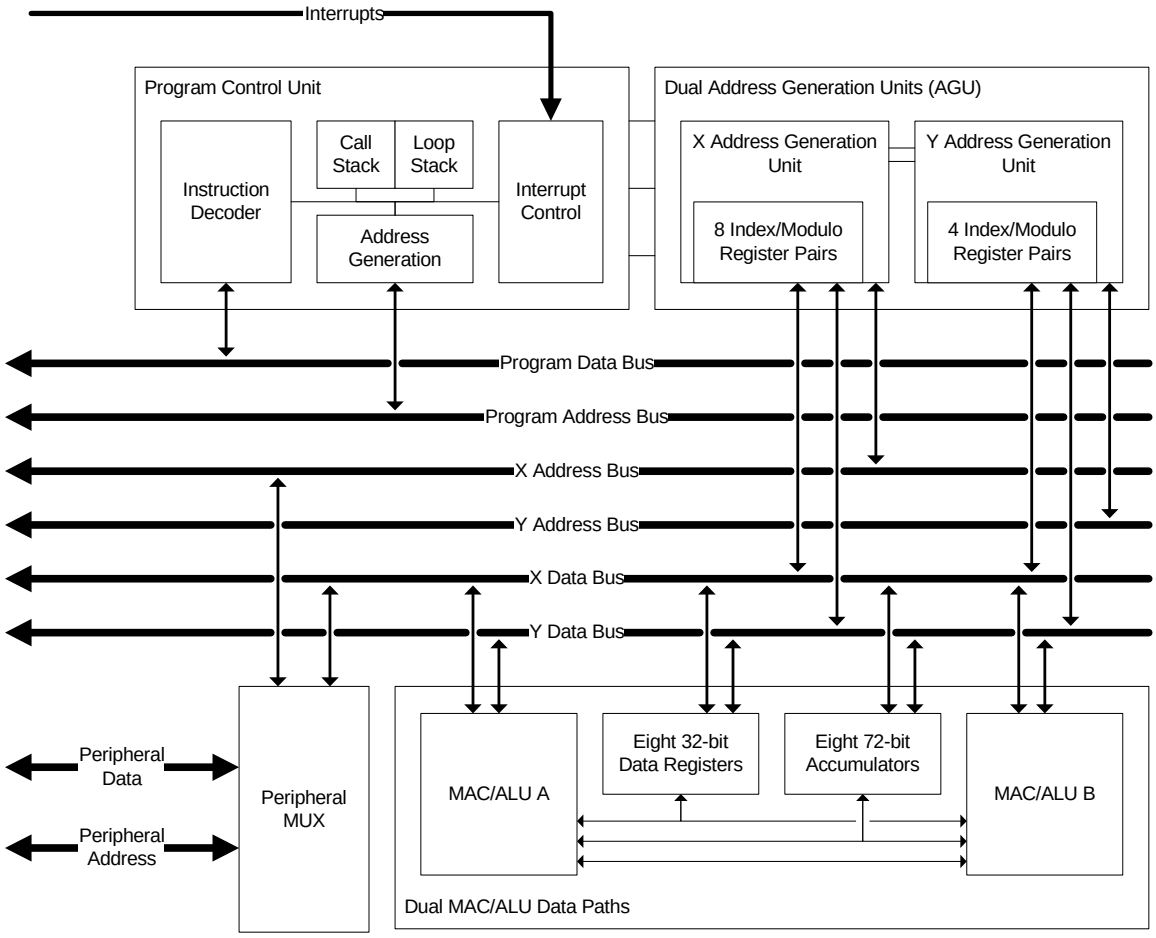

Figure 2-1. Cirrus Logic 32-Bit Architecture

The Cirrus Logic 32-bit DSP core consists of the following modules:

Copyright 2013 Cirrus Logic

2

- Program control unit
- Parallel Data Paths (A and B).
- Parallel Address Generation Units (AGUs) The AGUs contain:
  - Eight 16-bit registers for address generation
  - Eight 16-bit registers that work in conjunction with the index registers to provide different addressing modes.

2.2 Data Path and Accumulators Unit

Figure 2-2 shows the data flow within the Data Path and Accumulator Unit. Each data path has four 32-bit general-purpose registers and four 72-bit accumulators (eight each, total). Each 72-bit accumulator is the concatenation of three registers: Guard, High, and Low.

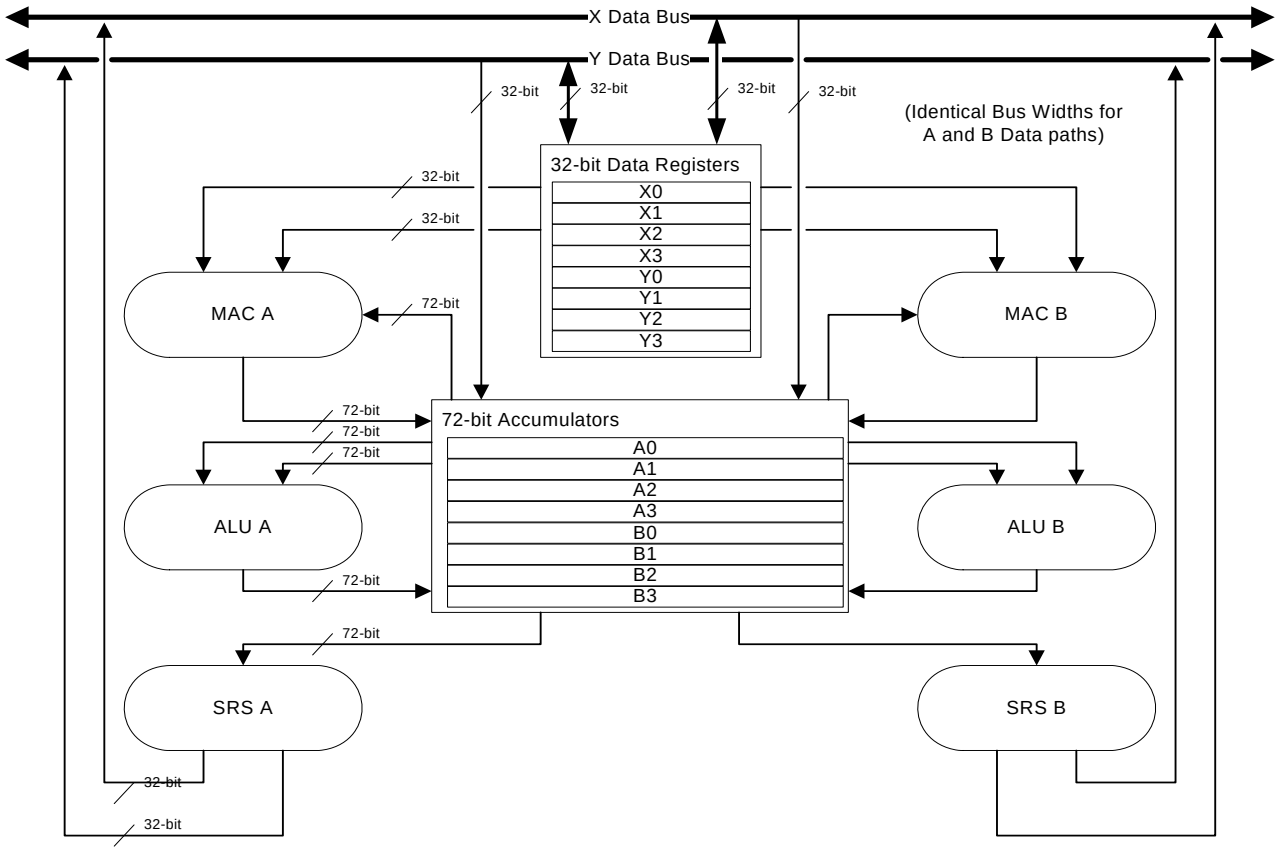

Figure 2-2. Data Flow within Data Path and Accumulators Unit

Copyright 2013 Cirrus Logic

The Guard registers are 8 bits, High and Low registers are 32 bits, and all parts can be addressed independently. See Figure 2-3. Each data path also has one Multiply-Accumulate unit (MAC), Shifter/Rounder/Saturator (SRS) and Arithmetic Logic Unit (ALU). The ALU is responsible for all the logical operations performed on the accumulators. The way the SRS handles data in the accumulator and transfers it to the data bus is explained later in this chapter.

X Data Registers	
31	0
x0	
x1	
x2	
x3	

Y Data Registers	
31	0
y0	
y1	
y2	
y3	

A Accumulators

71 64	63 32	31 0
a0g	a0h	a0l
a1g	a1h	a1l
a2g	a2h	a2l
a3g	a3h	a3l

B Accumulators

71 64	63 32	31 0
b0g	b0h	b0l
b1g	b1h	b1l
b2g	b2h	b2l
b3g	b3h	b3l

Figure 2-3. Data Path Registers

Copyright 2013 Cirrus Logic

For successive additions to a particular accumulator, it is possible for the sum being greater than maximum value that can be represented by 32 fixed-point bits. The guard bits allow for temporary accommodation of this overflow. This is useful when you are adding a bunch of numbers that will sum to be less than the maximum, but can overflow all the additions have completed.

Example 2-1

```
For example
Consider this sum:  0.5 + 0.75 - 0.3 - 0.8  - 0.6 - 0.4  - 0.25 + 0.5 = -0.6
The intermediate sums are:

0.5 + 0.75 = 1.25 (over-flow occurs. Can't be represented by 32 fixed-point
bits – one more bit is needed)

0.5 + 0.75 - 0.3 = 0.95

0.5 + 0.75 - 0.3 - 0.8 = 0.15

0.5 + 0.75 - 0.3 - 0.8  - 0.6 = -0.45

0.5 + 0.75 - 0.3 - 0.8  - 0.6 - 0.4 = -0.85

0.5 + 0.75 - 0.3 - 0.8  - 0.6 - 0.4  - 0.25 = -1.1 (Overflow occurs. Can't be
represented by 32 fixed-point bits – one more bit is needed)

0.5 + 0.75 - 0.3 - 0.8  - 0.6 - 0.4  - 0.25 + 0.5 = -0.6
```

With these 8 guard bits the numeric range of the accumulator is extended by 256 extra levels of precision.

2.2.1 Data Representation

The data representation used in this processor is the two's complement fractional notation. The 32-bit, 64-bit, and 72-bit fractional representations are shown in Figure 2-4, Figure 2-5, and Figure 2-6. The S bit is the sign bit. The X and Y data registers contain 32-bit operands, and the accumulators contain 72-bit operands which may be read out through the SRS as 32-bit or 64-bit operands. All internal ALU operations in the data path are 72 bits.

The 32-bit operand represents Twos complement form with the left most bit is the sign bit, followed by the radix point and the 31-bit fractional part. The largest positive number that can be represented is `0x7fffffff` ($1-2^{-31}$ in decimal), and the largest negative number is `0x80000000` (-1.0 in decimal).

Copyright 2013 Cirrus Logic

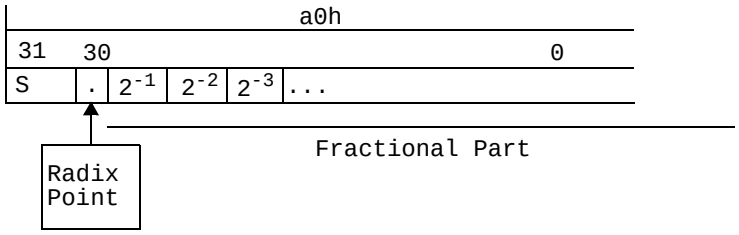

Figure 2-4. 32-bit Fractional Representation

The 64-bit accumulator has the sign bit as the left most bit followed by the radix point and the 63-bit fractional part.

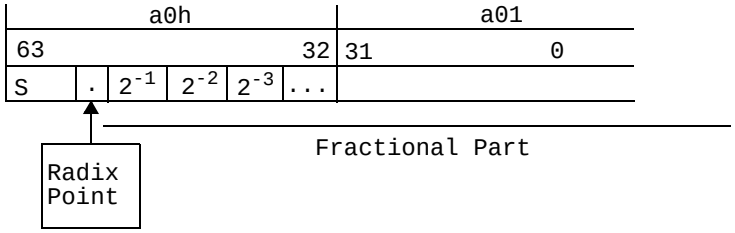

Figure 2-5. 64-bit Fractional Representation

A 72-bit accumulator has the sign bit as the left most bit followed by 8 integer bits, the radix point, and the 63-bit fractional part. The largest positive number that can be represented in an accumulator is `0x7f ffff ffff ffff ffff` ($256-2^{-63}$ in decimal) and the largest negative number is `0x80 0000 0000 0000 0000` (-256.0 in decimal).

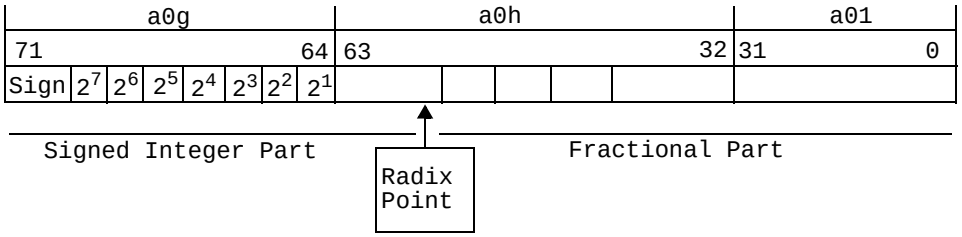

Figure 2-6. 72-bit Fractional Representation

A comparison can be made between integer and fractional number representation. The range for integer representation is $+/-2^{N-1}$, and for fractional representation is +/-1. To convert from an integer to a fraction the integer is multiplied by a scaling factor. The representation of a result from an addition or subtraction for both integer and fractional numbers is the same. This is not true when the arithmetic operation is a multiplication or a division. The difference is that the extra bit obtained in integer multiplication acts as a duplicate sign bit in fractional multiplication. See Figure 2-7.

Copyright 2013 Cirrus Logic

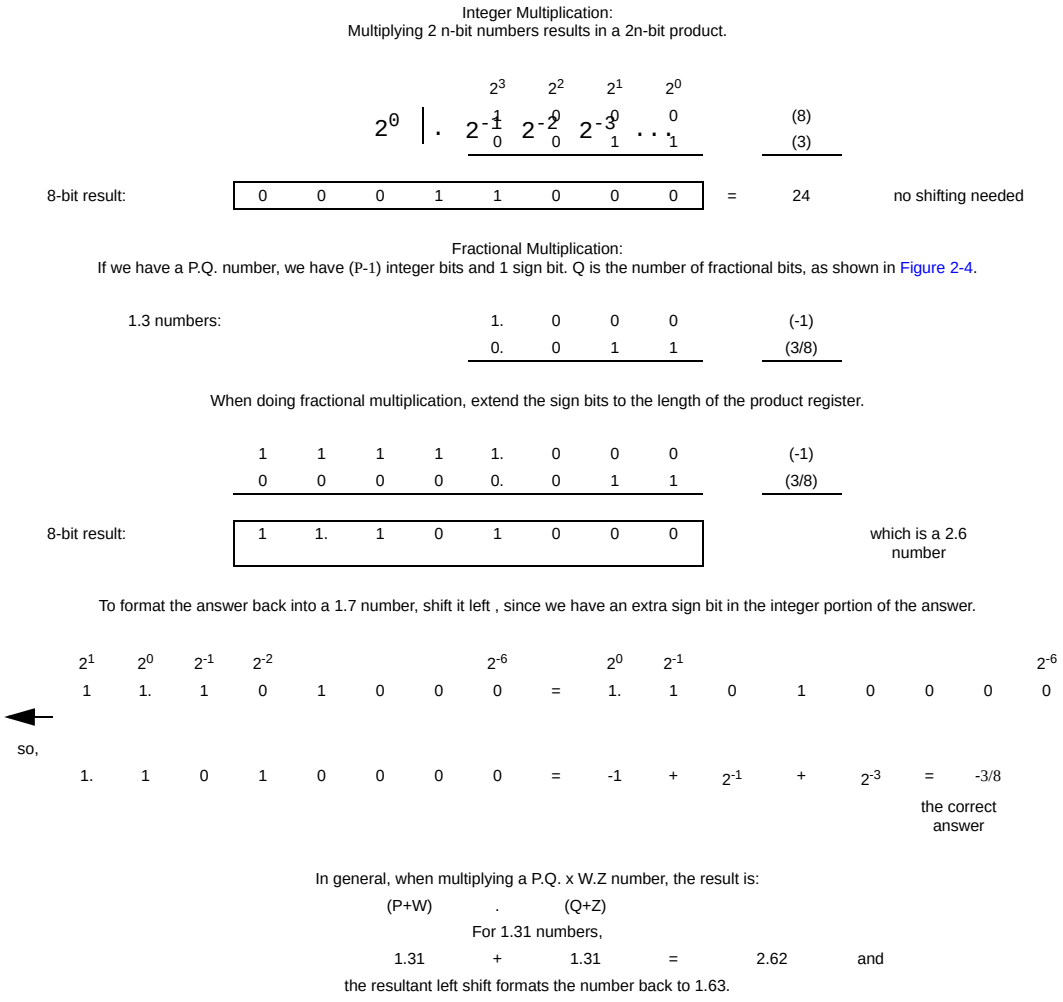

Figure 2-7. Integer vs. Fractional Multiplication

2.2.2 Accumulator Data Transfers

A 32-bit value may be transferred from the X data bus or the Y data bus to an accumulator. The 32-bit value will be loaded into the high portion of the accumulator and sign extended into the guard. The low portion of the accumulator will be zeroed.

A 64-bit value may be transferred from the X data bus and Y data bus to an accumulator. The 32-bit value from the X data bus will be loaded into the high portion of the accumulator and sign extended into the guard. The 32-bit value from the Y data bus will be loaded into the low portion of the accumulator.

Copyright 2013 Cirrus Logic

2.2.2.1 Move to Accumulator

Move from data register into 72-bit accumulator (a0=x0). See Figure 2-8 and Figure 2-9.

Before Execution

x0							
31							0
1	2	3	4	5	6	7	8

a0g		a0h								a0l							
71	64	63							32	31							0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

After Execution

x0							
31							0
1	2	3	4	5	6	7	8

a0g		a0h								a0l							
71	64	63							32	31							0
0	0	1	2	3	4	5	6	7	8	0	0	0	0	0	0	0	0

Figure 2-8. Positive 32-bit Value

Before Execution

x0							
31							0
8	2	3	4	5	6	7	8

a0g		a0h								a0l							
71	64	63							32	31							0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

After Execution

x0							
31							0
8	2	3	4	5	6	7	8

a0g		a0h								a0l							
71	64	63							32	31							0
f	f	8	2	3	4	5	6	7	8	0	0	0	0	0	0	0	0

Figure 2-9. Negative 32-bit Value

Copyright 2013 Cirrus Logic

2.2.2.2 Moving from Accumulator

Each data path (A and B) has its own independent SRS unit as shown in Figure 2-2. The SRSs are the only interface to move a value from an accumulator in a data path to the internal X and Y data busses.

The SRS units are named in the same order that they process data – Shift first, Round second, Saturate last.

When an accumulator is transferred to a data bus, data saturation will occur and the limit bit in the Condition Code register is set. Examples of saturation are shown in Section 2.2.2.3.

The data shifters can shift the data coming from an accumulator one or two bits to the right, one bit to the left, or pass the data without shifting. The data in the accumulator remains unchanged. The shifts are controlled by the shift bits in the Mode register. Shifting facilitates the scaling of fixed point data which is useful in implementing the block floating point algorithms. Examples of shifting are shown in Section 2.2.2.5.

The Rounder has four modes affecting how the data in the low register of the accumulator (i.e. a0l) is handled when an accumulator is moved onto the X or Y data bus:

- Truncate - The data in the low register is ignored.
- Add ½ then truncate - One-half of the least significant bit of the high register of the accumulator (`0x00.00000000.80000000`) is added to the data in the low register before truncation.
- Round to zero - Positive accumulators are simply truncated, but if the value of the accumulator is negative the high register is incremented by 1 before truncation. This exists for removing limit-cycle operations in IIR filters.
- Add dither then truncate - If the top 4 bits of the low register is larger (unsigned comparison) than a 4-bit random number, the high register is incremented by 1 before truncation. The 4-bit random number is actually bits 15, 13, 12, and 10 of a 16-bit random number that is seeded at reset. The A and B SRS units have individual 16-bit random numbers that are seeded differently at reset. The 16-bit random numbers are post-updated after each use by the individual SRS. In other words, moving data out of an A accumulator onto the X or Y data bus with rounding in this mode updates only the A SRS 16-bit random number after it has been used for that comparison. Examples of rounding are shown in Section 2.2.2.5.

Any move from a full 72-bit accumulator to a 32-bit destination (X or Y data register, X or Y memory, peripheral space, etc.) is appropriately shifted, rounded, and saturated. Moves from any portion of an accumulator (for example, `a0h, a3l, b2g`, etc.) are not affected by the SRS unit. Additionally, the Bitwise Accumulator Move instruction (`a0=+b3`) does not utilize the SRS.

Copyright 2013 Cirrus Logic

2.2.2.3 Saturation Examples

Figure 2-10, Figure 2-11, and Figure 2-12 are examples of saturation:

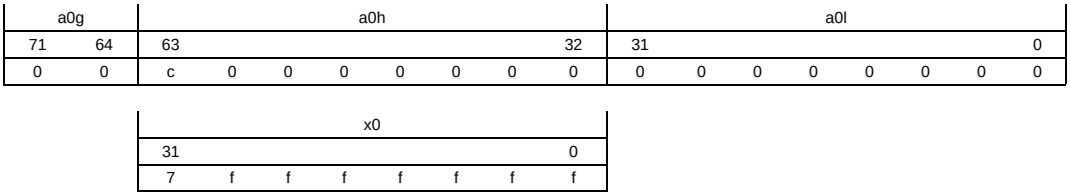

Figure 2-10. Positive Saturation: x0=a0

Note: `0x00c000000000000000` (1.5) is limited to `0x7fffffff` (.99999999953).

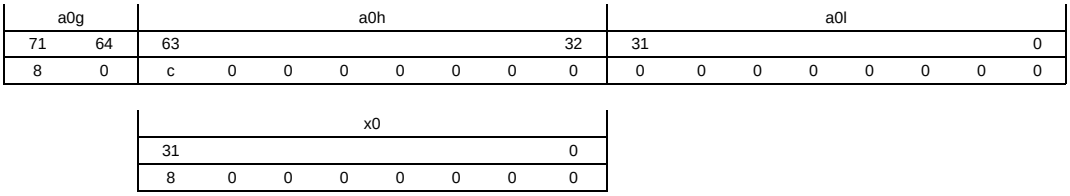

Figure 2-11. Rounding Example: Negative Saturation: x0=a0

Note: `0x80c000000000000000` (-1.5) is limited to `0x80000000` (-1).

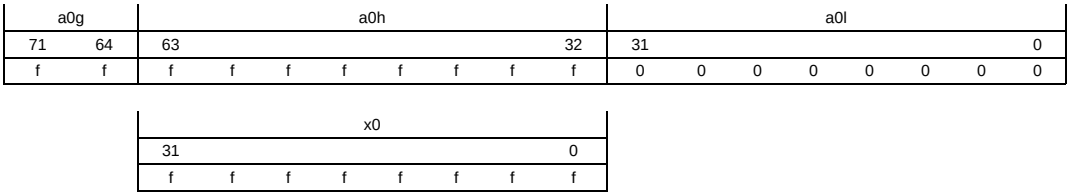

Figure 2-12. No Saturation: x0=a0

Note: `0xffffffffff00000000` (-.99999999953) remains unchanged as `0xffffffff` (-.99999999953).

2.2.2.4 Rounding Examples

Table 2-1 is an example of rounding.

Table 2-1. Result of x0=a0 for a Given Rounding Mode (Shifting Off)

a0 Contents			x0 Result Given Rounding Mode			
a0g	a0h	a0l	Truncate	add .5	round to 0	dither
00	00000001	80000000	00000001	00000002	00000001	00000001 or 00000002
00	00000001	00000001	00000001	00000001	00000001	00000001 or 00000002
ff	80000000	00000001	80000000	80000000	80000001	80000000 or 80000001
ff	fffffff	80000000	fffffff	00000000	00000000	fffffff or 00000000

Copyright 2013 Cirrus Logic

2.2.2.5 Shifting Examples

Table 2-2, Table 2-3, and Table 2-4 are examples of shifting.

Table 2-2. Result of x0=a0 for a Given Shifting Mode with Rounding Set to Truncate (off)

a0 Contents			x0 Result Given Rounding Mode			
a0g	a0h	a0l	No shift	Right shift 1	Right shift 2	Left shift 1
00	7fffffff	00000000	7fffffff	3fffffff	1fffffff	7fffffff
01	80000001	80000000	7fffffff	7fffffff	60000000	7fffffff
ff	00000000	00000000	80000000	80000000	c0000000	80000000
40	00000000	40000000	7fffffff	7fffffff	7fffffff	80000000

Table 2-3. Result of x0=a0 for a Given Shifting Mode with Rounding Set to Add ½ then Truncate

a0 Contents			x0 Result Given Rounding Mode			
a0g	a0h	a0l	No shift	Right shift 1	Right shift 2	Left shift 1
00	7fffffff	00000000	7fffffff	40000000	20000000	7fffffff
01	80000001	80000000	7fffffff	7fffffff	60000000	7fffffff
ff	00000000	00000000	80000000	80000000	c0000000	80000000
40	00000000	40000000	7fffffff	7fffffff	7fffffff	80000000

Table 2-4. Result of x0=a0 for a Given Shifting Mode with Rounding Set to Round to Zero

a0 Contents			x0 Result Given Rounding Mode			
a0g	a0h	a0l	No shift	Right shift 1	Right shift 2	Left shift 1
00	7fffffff	00000000	7fffffff	3fffffff	1fffffff	7fffffff
01	80000001	80000000	7fffffff	7fffffff	60000000	7fffffff
ff	00000000	00000000	80000000	80000001	c0000001	80000000
40	00000000	40000000	7fffffff	7fffffff	7fffffff	80000000

2.3 Parallel Address Generation Unit

This unit consists of two sets of 12 registers - the 16-bit Index (I) registers i0-i11 and the 16-bit modulo-offset registers nm0-nm11. The data flow for the Address Generation Unit (AGU) is shown in Figure 2-13. A modulo-offset register consists of a modulo portion, bits [15:12], and an offset portion, bits [11:0]. See Table 2-5 and Table 2-6. The offset portion is used to update the index register and the modulo portion to specify the type of addressing:

- Linear
- Reverse binary
- Modulo

The offset portion is treated as a signed 12-bit number, and as such can update the address in the corresponding index register with any value from -2048 to 2047 (`0x800-0x7ff`).

Copyright 2013 Cirrus Logic

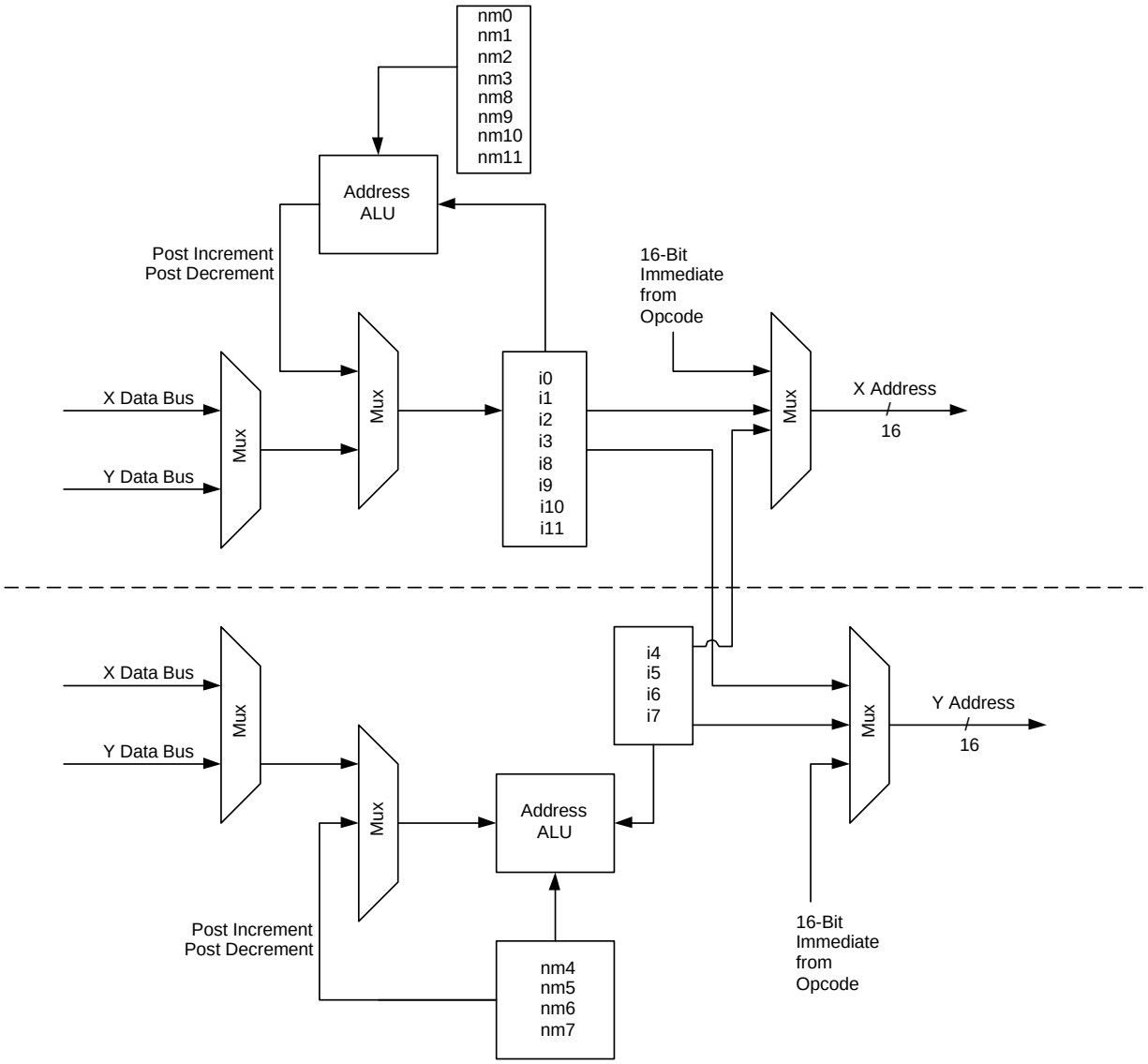

Figure 2-13. Data Flow for the Parallel Address Generation Unit

Copyright 2013 Cirrus Logic

2

Table 2-5. Index Registers

Register Names	Bits	
i0	15	0
i1	15	0
i2	15	0
i3	15	0
i4	15	0
i5	15	0
i6	15	0
i7	15	0
i8	15	0
i9	15	0
i10	15	0
i11	15	0

Table 2-6. Increment-Modulo Registers

Register Name	Modulo			Increment		
	Field Name	Bits		Field Name	Bits	
nm0	m0	15	12	n0	11	0
nm1	m1	15	12	n1	11	0
nm2	m2	15	12	n2	11	0
nm3	m3	15	12	n3	11	0
nm4	m4	15	12	n4	11	0
nm5	m5	15	12	n5	11	0
nm6	m6	15	12	n6	11	0
nm7	m7	15	12	n7	11	0
nm8	m8	15	12	n8	11	0
nm9	m9	15	12	n9	11	0
nm10	m10	15	12	n10	11	0
nm11	m11	15	12	n11	11	0

2.3.1 Addressing Modes

2.3.1.1 Modulo Addressing

Modulo addressing can be used to implement circular buffers whose size is a power of 2, ranging from 4 to 32768. When incrementing an index register with the corresponding NM register set for modulo addressing the index register wraps around to the beginning of the buffer when the end of the buffer is reached. The most significant 4 bits of the NM register control whether and how modulo addressing is used. If set to a value between `0x1` and `0xe`, modulo addressing is used with an address boundary of 2^(m+1). If set to `0x0`, then linear addressing is used. If set to `0xf`, reverse binary addressing is used. See Table 2-7.

Copyright 2013 Cirrus Logic

Table 2-7. Addressing Modes, Defined by the NM Registers

MS 4 bits of NM	Addressing Mode
0x0	Linear Addressing
0x1	Modulo 4
0x2	Modulo 8
0x3	Modulo 16
0x4	Modulo 32
0x5	Modulo 64
0x6	Modulo 128
0x7	Modulo 256
0x8	Modulo 512
0x9	Modulo 1024
0xa	Modulo 2048
0xb	Modulo 4096
0xc	Modulo 8192
0xd	Modulo 16384
0xe	Modulo 32768
0xf	Reverse Binary Addressing

To use modulo addressing, circular buffers must be placed in memory such that their base address is a multiple of their size. For example, to use modulo addressing on a 1024-sample (`0x400`) circular buffer the base address of the buffer must be `0x0000`, `0x0400`, `0x0800`, `0x0c00`, `0x1000`, etc. In modulo addressing mode, all index register updates (+/-1, +/-2, +/-n) will result in an address that is within the boundaries of the buffer, except for +/-n when n is greater than or equal to the buffer size, in which case the index register will jump out of the circular buffer.

2.3.1.2 Reverse Binary Addressing

Reverse binary addressing is useful for implementing Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) algorithms to switch the signals from time to frequency and frequency to time domain. In writing the code for an FFT it is necessary either to get the input data in a reverse binary (bit reverse order) or to extract the correct output data in a reverse binary order. The number of data points or a block of data that can be reverse binary addressed will always be a power of 2.

The reverse binary addressing is implemented by setting the value in the M register to `0xf`. Suppose the data block is 2^k locations. The N register should be initialized to a value 2^{k-1}. The index register i is initialized to any address between the lower and upper boundary. The lower boundary is $k*2^t$ where t is any integer. The upper boundary is $(k*2^t) + (2^t-1)$. The mode of addressing must be `i1+=n`.

Copyright 2013 Cirrus Logic

2.3.1.3 Immediate Addressing

This addressing mode has both long word instruction (32 bits) and short word instruction (16 bits) versions. In the long word instruction, the address field is 16 bits, which allows access to the X and Y memory of up to 64k locations. This addressing is used to transfer data from memory to registers.

For example:

```
a0 = xmem[0x6540]
```

In the short word instruction, the address field is 6 bits, which allows access to the first 64 locations in X, Y, and XY memory. XY memory is the concatenation of X and Y memory with the same address as indicated by 6-bit address field. When XY memory is used as the source or destination of a data transfer, the destination/source should be either a pair of data path registers or an accumulator.

For example:

```
x0,y0 = xymem[12] or a0 = xymem[12]
```

The short word instruction can be used in conjunction with an arithmetic or logic instruction.

2.3.1.4 Indexed Addressing

This addressing mode uses long (32 bits), short (16 bits), and 8-bit instructions. Two instructions using 8-bits can be used simultaneously along with an arithmetic or logic instruction, but one move must use the X memory field and the other the Y memory field. One instruction using 16 bits of the program word can be used along with an arithmetic or logical instruction.

In the long word instruction (see Section 4.5), X, Y, and P memory can be addressed using indexed addressing. Index registers `i0-i11` are used and they can be post- incremented or post-decremented. The updates available are +/-1, +/-2 and +/-n. The value of n is stored in the corresponding NM Register.

In the short word instruction (see Section 4.1), X, Y, and XY memory can be addressed. XY memory is the concatenation of the X and Y memory. XY memory is used for complex and double moves. When XY memory is used as the source or destination of a data transfer, the destination/source should be either a pair of data path registers or an accumulator.

The index registers used here are `i0-i11,` and the updates available are +/-1, +/-2 and +/-n.

When performing parallel moves (see Section 4.2), use X memory with X data registers, A accumulators and i0 or i1 index registers. Use Y memory with Y data registers, B accumulators and i4 or i5 index registers. Index register updates are limited to +/-1 and +n.

Example of a valid instruction is:

```
a0=a0+b0; x1=xmem[i0]; i0+=n; ymem[i4]=b1; i4-=1
```


Copyright 2013 Cirrus Logic

2.3.1.4.1 Index Register Updates

All the standard index register updates can be used without an associated move. For example:

```
i0+=n
```

These are parallel instructions that can be paired with a MAC/ALU instruction:

```
a0 = x0*y0;i4-=2
```

Additional index register update instructions exist that are not available for use with moves. These instructions add an immediate value to an index register and place the result in a second index register:

```
i0 = i3 + (0x1234)
```

The target index register can be the same as the first argument:

```
i7 = i7 + (0x1234)
```

There are two forms of these instructions. One is a full word instruction that cannot be used with any parallel instruction; this form uses a full 16-bit operand for the immediate value. The second is a 16-bit instruction that can be paired with a MAC/ALU instruction, and is limited to a 6-bit immediate operand and a source index register of i8-i11:

```
a0 = x0*y0; i0 = i9 + (0x3f)
```

These instructions are affected by the NM register associated with the source index register:

```
i2 = (0x3f)# last address of a modulo 16 buffer
nm2 = (0x3000)# set to modulo 16
nop# see “Index Register Loading Restrictions”
i3 = i2 + (0x1)# after execution, i3==0x30
# (nm3 is ignored)
```

Example 2-2 Syntax for Index Register Updates

```
<index register> += 1 or 2 or n
<index register> -= 1 or 2 or n
```

2.3.1.4.2 Parallel Index Register Updates

Index updates for parallel instructions can be executed in parallel with other parallel instructions. References in syntax statements to “`update`” almost always have the same update options:

```
+1
-1
+2
-2
+n
-n
```

When an instruction has fewer options than noted here, the available update options are noted under the **Restrictions** subsection for each instruction. Index updates are always optional.

Copyright 2013 Cirrus Logic

2

2.3.1.4.3 Index Register Loading

Index registers can be loaded by using register-to-register moves:

```
i0 = x0
```

or immediate data loads:

```
i5 = (0x1234)
```

Additionally, a specialized version of the full-word immediate value index update instruction (see previous section) can be used to load data into an index register:

```
i0 = (0) + (0x1234)
```

As stated, this is a full-word instruction that takes a full 16-bit immediate value, and as such it cannot be paired with any parallel instructions.

2.3.1.4.4 Index Register Loading Restrictions

Due to the pipelined nature of the AGU, instructions that utilize the AGU update index registers during the decode phase of the pipeline, which is the second of the three phases (Fetch - Decode - Execute). This implies that any modification to an index register that occurs during the execute phase will be undefined for any AGU operations in the subsequent instruction. The main impact on programming is that an index register that is modified through a register-to-register move or an immediate load is unavailable for use or update by the AGU in the next instruction. In this example:

```
i0 = (0x40)
nop# this is necessary
x0 = xmem[i0]
```

A one-instruction buffer is required between loading and using i0. A *nop* was used here, but any instruction that does not require i0 would have sufficed (and is usually preferable to avoid wasting cycles.) If an index register is used before it is ready, the assembler will warn the user.

Instructions that do not use the AGU are unaffected by this pipeline effect:

```
i0 = x0
x2 = i0# no problem here...
```

Note that the immediate value index update instructions use the AGU to load/add the immediate value into the index register, so the result can be used immediately:

```
i0 = (0) + (0x40)
x0 = xmem[i0]# no waiting necessary
```

Operations performed by an instruction during the *Decode* phase of the instruction pipeline can be lost if another instruction performs the same operation but in the *Execute* phase of the same cycle. In the example below, the second `i0` assignment is not performed because the previous instruction performs an `i0` assignment during its *Execute* phase. See Figure 2-14.

```
BitSet (i0), (0xEEEE)
i0 = (0) + (0xDDDD)
```


Copyright 2013 Cirrus Logic

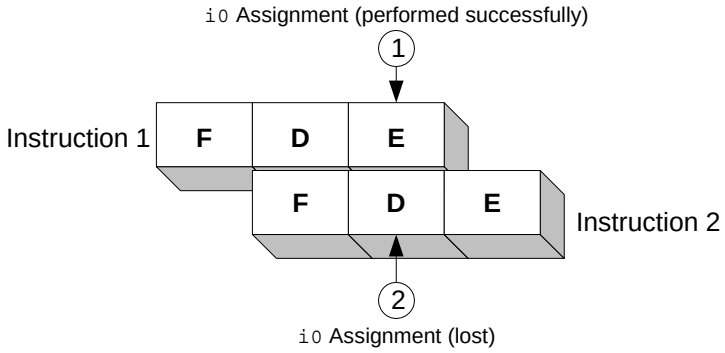

Figure 2-14. Execute Phase vs. Decode Phase Assignments

2.4 Program Control Unit

The program control unit consists of a Program Counter (PC), two system stacks and two control registers: the Mode register and the Condition Code register. The Mode Register is the MR, the Condition Codes Register is the CCR.

2.4.1 Program Counter

The PC is a 16-bit pointer used to indicate the location of the next instruction to fetch from program memory.

2.4.2 Subroutine Stack

The subroutine stack is used to store the return PC for subroutine calls. It is 16-bits wide and implemented as a 16-entry, circular buffer with overflow and underflow interrupts. Each time there is a call instruction the current PC is stored on the top of the stack and the call stack pointer is auto-incremented or auto-decremented depending on the configuration of the *jsr_mode* register. Conversely for a return instruction the entry at the top of the stack is popped and is used as the next PC value.

2.4.3 Loop Stack

The loop stack is used to store the current do-loop state (last address, first address, and count) or do_patch state (patch length, last address, return address) when a new do-loop or do_patch is encountered prior to completing any preceding do-loop or do_patch. It is 49-bits wide and is implemented as an 8-entry circular buffer.

When a do-loop or do_patch is encountered, the state required to manage software flow control is kept in a 49-bit register that appears to software as two registers *lp_data1 (*Table 2-14*)* and *lp_data2 (*Table 2-15*)*. If software is executing a do-loop or do_patch and encounters another do-loop or do_patch, the state of the current do-loop or do_patch is pushed from *lp_data* to the loop stack, and the loop stack pointer is auto-incremented or auto-decremented depending on the configuration of the *lst_mode* register. And, the state of the new do-loop or do_patch is placed in *lp_data*. Conversely, when a do-loop or do_patch

Copyright 2013 Cirrus Logic

has completed, the entry at the top of the stack is popped to *lp_data* which restores the previous do-loop or do_patch state.

2.4.4 Subroutine Stack and Loop Stack Common Implementations

The two stacks operate independently of each other and consist of circular buffers. Each stack has programmable thresholds for both overflow and underflow which can be set up to cause interrupts. Each stack can grow either up or down. Most configuration is via the registers *jsr_mode*, *jsr_ovf*, and *jsr_unf* for the subroutine stack or *lst_mode*, *lst_ovf*, and *lst_unf* for the loop stack. Software may directly read or write subroutine stack data via *jsr_data* and loop stack data via *lst_data1* and *lst_data2*. Note that if auto-update bits in *jsr_mode* or *lst_mode* are not cleared, then software reads and writes of the stacks will modify the respective stack pointers *mr_jsr_ptr* and *mr_lst_ptr*.

There are a total of five interrupt masks that are important to the proper operation of the stack interrupts. Each stack has two maskable interrupts, one each for overflow and underflow. All stack interrupts can be globally disabled by a global interrupt enable bit. Each individual stack interrupt has a mask which prevents the interrupt from getting queued or recorded. On the other hand, clearing the global stack interrupt enable bit prevents the core from taking an interrupt request.if the individual interrupt mask bits are clear, then the interrupts are still queued up and will be serviced the global stack interrupt enable bit is set.

Each of the stacks can be modified by software. This is done through reads and writes to *jsr_mode (*Table 2-8*)*, *lst_mode (*Table 2-9*), mr_jsr_ptr* (Table 2-11), *jsr_data (*Table 2-12*)*, *mr_lst_ptr (*Table 2-13*)*, *lst_data1 (*Table 2-16*)*, and *lst_data2 (*Table 2-17*)* registers. Note that the stack pointer auto-increment and auto-decrement bits should be disabled (in *jsr_mode* or *lst_mode*) before attempting to modify the contents of the stack, unless that behavior is desired. Though the pointers for each stack, *mr_jsr_ptr* and *mr_lst_ptr*, are fields in the Mode register, never modify the Mode register directly. The following Mode register fields should be accessed through separate registers: *mr_jsr_ptr*, *mr_lst_ptr*, *mr_r*, *mr_s*, or *mr_sr*.

Several additional registers are useful for handling overflow and underflow of the stacks: *jsr_ovf (*Table 2-18*), jsr_unf (*Table 2-15*), lst_ovf (*Table 2-16*), lst_unf (*Table 2-17*)*. When an underflow or an overflow condition occurs and the appropriate interrupt mask is set, interrupts are queued but held for execution. As long as the stack interrupt enable mask is set the core will fetch an ISI (interrupt service instruction) from the stack ISR (interrupt service routine) table and execute it. By default the stack ISR table is located immediately after the PIC (peripheral interrupt controller) ISR table. Typically the 32 ISIs for the PIC ISR table will be located at `0x0000-0x001f` and the 4 ISIs for the stack ISR table will be located at `0x0020-0x0023`. If the stack ISR needs to be relocated, simply modify bits [15:2] of the *stq_base* (Table 2-10) with the desired address. Bits [1:0] of the *stq_base* register always read as 0.

Copyright 2013 Cirrus Logic

2.4.5 jsr_mode Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								*	*	*	*	*	*	*	*
x	x	x	x	x	x	x	x	1	0	0	0	0	0	1	0

Table 2-8. jsr_mode Bit Definitions

Bits	Field/Flag Name	Description
15:8	Reserved	Reserved.
7	jsr_wr_inc_dec	Auto increment(1) / auto-decrement(0) on write.
6	jsr_wr_ptr_en	Enable pointer auto-update on write.
5	jsr_rd_inc_dec	Auto incremnt(1) / auto decrement(0) on read.
4	jsr_rd_ptr_en	Enable pointer auto-update on read.
3	jsr_ovf_imask	Overflow interrupt mask. (Disable overflow interrupt.)
2	jsr_unf_imask	Underflow interrupt mask. (Disable underflow interrupt.)
1	jsr_int_en	Call-stack interrupt enable.
0	jsr_auto_stq	Auto-stack mode enable. (Reserved)

jsr_wr_inc_dec — This feature is only available when the "auto-update on write" bit, *jsr_wr_ptr_en* is set. If so, auto-increment or auto-decrement the respective stack pointer when the top of the stack is written.

jsr_wr_pt_en — When set, a write to the respective stack will update (auto-increment or auto-decrement) the stack pointer.

jsr_rd_inc_dec — This feature is only available when the "auto-update on read" bit, *jsr_rd_ptr_en* is set. If so, auto-increment or auto-decrement the respective stack pointer when the top of the stack is read.

jsr_rd_ptr_en — When set, a read of the respective stack will update (auto-increment or auto-decrement) the stack pointer.

jsr_ovf_imask — When zero, this bit disables interrupts that would otherwise be generated by an overflow condition.

jsr_unf_imask — When zero, this bit disables interrupts that would otherwise be generated by an underflow condition.

jsr-int-en — This bit is equivalent in function to the MR[7] bit, except it is used for stack interrupts. Clearing this bit prevents the DSP from taking requests for either underflow or overflow interrupts. However, interrupts are still queued, assuming the corresponding mask bits are set.

auto-stq — This bit enables an un-supported stack mode. It should always be kept clear (0).

Copyright 2013 Cirrus Logic

2.4.6 lst_mode Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								*	*	*	*	*	*	*	*
x	x	x	x	x	x	x	x	1	0	0	0	0	0	1	0

Table 2-9. lst_mode Bit Definitions

Bits	Field/Flag Name	Description
15:8	Reserved	Reserved.
7	lst_wr_inc_dec	Auto increment(1) / auto-decrement(0) on write.
6	lst_wr_ptr_en	Enable pointer auto-update on write.
5	lst_rd_inc_dec	Auto incremnt(1) / auto decrement(0) on read.
4	lst_rd_ptr_en	Enable pointer auto-update on read.
3	lst_ovf_imask	Overflow interrupt mask. (Disable overflow interrupt.)
2	lst_unf_imask	Underflow interrupt mask. (Disable underflow interrupt.)
1	lst_int_en	Call-stack interrupt enable.
0	lst_auto_stq	Auto-stack mode enable. (Reserved)

lst_wr_inc_dec — This feature is only available when the "auto-update on write" bit, *lsr_wr_ptr_en* is set. If so, auto-increment or auto-decrement the respective stack pointer when the top of the stack is written.

lst_wr_pt_en — When set, a write to the respective stack will update (auto-increment or auto-decrement) the stack pointer.

lst_rd_inc_dec — This feature is only available when the "auto-update on read" bit, *lsr_rd_ptr_en* is set. If so, auto-increment or auto-decrement the respective stack pointer when the top of the stack is read.

lst_rd_ptr_en — This bit controls whether a read of the respective stack will update (auto-increment or auto-decrement) the stack pointer.

lst_ovf_imask — When zero, this bit disables interrupts that would otherwise be generated by an overflow condition.

lst_unf_imask — When zero, this bit disables interrupts that would otherwise be generated by an underflow condition.

lst-int-en — This bit is equivalent in function to the MR[7] bit, except it is used for stack interrupts. Clearing this bit prevents the DSP from taking requests for either underflow or overflow interrupts. However, interrupts are still queued, assuming the corresponding mask bits are zero.

auto-stq — This bit enables an un-supported stack mode. It should always be kept clear (0).

Copyright 2013 Cirrus Logic

2.4.7 stq_base Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
stq_isr_base_addr														Rsvd.	
0	0	0	0	0	0	0	0	0	0	1	0	0	0	x	x

Table 2-10. stq_base Bit Definitions

Bits	Field/Flag Name	Description
31:16	Reserved	Reserved.
15:2	stq_isr_base_addr	ISR base address.
1:0	Reserved	Reserved.

2.4.8 mr_jsr_ptr Register

This is the index of the next entry to which data will be pushed and/or the index of the last entry popped. It appears as a field in Mode Register but should be modified here.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												mr_jsr_ptr			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2-11. mr_jsr_ptr Bit Definitions

Bits	Field/Flag Name	Description
3:0	mr_jsr_ptr	Call stack pointer

2.4.9 jsr_data Register

The top of the call stack is **(mr_jsr_ptr - 1) mod 16**.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
jsr_data															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2-12. jsr_data Bit Definitions

Bits	Field/Flag Name	Description
15:0	jsr_data	PC value at top of call stack.

Copyright 2013 Cirrus Logic

2.4.10 mr_lst_ptr Register

This is the index of the next entry to which data will be pushed and/or the index of the last entry popped. It appears as a field in Mode Register but should be modified here.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													mr_lst_ptr		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2-13. mr_lst_ptr Bit Definitions

Bits	Field/Flag Name	Description
2:0	mr_lst_ptr	Loop stack pointer

2.4.11 lp_data1 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
lp_lad															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
lp_fad															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2-14. lp_data1 Bit Definitions

Bits	Field/Flag Name	Description	
		do loop	do_patch
31:16	lp_lad	Top of loop stack last address	Top of loop stack last address
15:0	lp_fad	Top of loop stack first address	Return address: just after do_patch instruction

2.4.12 lp_data2 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
lp_cnt															
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Table 2-15. lp_data2 Bit Definitions

Bits	Field/Flag Name	Description	
16	type	0 for do loop	1 for do_patch
15:0	lp_cnt	Top of loop stack count	Length of patch

Copyright 2013 Cirrus Logic

2.4.13 lst_data1 Register

The top of the loop stack is **(mr_lst_ptr - 1) mod 8**.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
lp_lad															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
lp_fad															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2-16. lst_data1 Bit Definitions

Bits	Field/Flag Name	Description	
		do loop	do_patch
31:16	lp_lad	Top of loop stack last address	Top of loop stack last address
15:0	lp_fad	Top of loop stack first address	Return address: just after do_patch instruction

2.4.14 lst_data2 Register

The top of the loop stack is **(mr_lst_ptr - 1) mod 8**.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
lp_cnt															
0	0	0	0	0	0	0	0	0	0	1	0	0	0	x	x

Table 2-17. lst_data2 Bit Definitions

Bits	Field/Flag Name	Description	
16	type	0 for do loop	1 for do_patch
15:0	lp_cnt	Top of loop stack count	Length of patch

2.4.15 jsr_ovf Register

An exception occurs when *mr_jsr_ptr* is incremented past *jsr_ovf* and the exception is enabled in the *jsr_mode* register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												jsr_ovf			
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Table 2-18. jsr_ovf Bit Definitions

Bits	Field/Flag Name	Description
3:0	jsr_ovf	Subroutine stack overflow threshold

Copyright 2013 Cirrus Logic

2.4.16 jsr_unf Register

An exception occurs when *mr_jsr_ptr* is decremented to *jsr_unf* and the exception is enabled in the *jsr_mode* register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												jsr_unf			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2-19. jsr_unf Bit Definitions

Bits	Field/Flag Name	Description
3:0	jsr_ovf	Subroutine stack underflow threshold

2.4.17 lst_ovf Register

An exception occurs when *mr_lst_ptr* is incremented past *lst_ovf* and the exception is enabled in the *lst_mode* register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													jsr_ovf		
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Table 2-20. lst_ovf Bit Definitions

Bits	Field/Flag Name	Description
2:0	lst_ovf	Loop stack overflow threshold

2.4.18 lst_unf Register

An exception occurs when *mr_lst_ptr* is decremented to *lst_unf* and the exception is enabled in the *lst_mode* register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													lst_unf		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2-21. lst_unf Bit Definitions

Bits	Field/Flag Name	Description
2:0	lst_unf	Loop stack underflow threshold

2.4.19 Mode Register

The Mode register is a 16-bit register defined as follows. Specific bits of the Mode register can be accessed for reading and writing through designated bitfield registers, shown in Table 2-22.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mr_jsr_ptr				mr_int_p	mr_lst_ptr			mr_int	reserved		Ls	R1	R0	S1	S0

Copyright 2013 Cirrus Logic

Table 2-22. Mode Register Bit Definitions

Bits	Field/Flag Name	Description
19	mr_jsr_ovf	Set whenever a call stack overflow occurs. **Note:** Applicable only for CS48L20.
18	mr_jsr_unf	Set whenever a call stack underflow occurs. **Note:** Applicable only for CS48L20.
17	mr_lst_ovf	Set whenever a loop stack overflow occurs. **Note:** Applicable only for CS48L20.
16	mr_lst_unf	Set whenever a loop stack underflow occurs. **Note:** Applicable only for CS48L20.
15:12	mr_jsr_ptr	Call stack pointer.
11	mr_int_p	Set to previous value of mr_int whenever an interrupt of any kind occurs; can be written by firmware but need not be (CS48L20 only).
10:8	mr_lst_ptr	Loop stack pointer.
7	mr_int	Interrupt enable/disable bit.
6:5	Reserved	Reserved.
4	Ls	Least significant bit - If set to one, data moved from the low part of an accumulator (such as a0l) will be logically shifted right one bit.
3:2	mr_r R1, R0	Round mode bits. Defined as: R1 R0 Round Mode 00 No round 01 Add 0.5 then truncate 10 Round to 0 11 Add dither then truncate **Note:** When setting these bits using the mr_r register, bits [3:2] of the mr_r register must be set to affect R1, R0.
1:0	mr_s S1, S0	Shift mode bits. Defined as: S1 S0 Shift Mode 00 No shift 01 Shift right 10 Shift right twice 11 Shift left

Note: Register mr_sr can be used to set bits [3:0] of the mode register with a single constant.

[19:16] mr_stq_queue_sticky is used to know whether two stack exceptions occurred at the same time so that after one is serviced, the other may also be. Otherwise, under certain conditions, a stack interrupt may be lost.

Note: These are all sticky and must be cleared by firmware.

[11] mr_int_p is used to know whether to enable interrupts or not when returning from a stack ISR.

Copyright 2013 Cirrus Logic

2.4.20 Condition Code Register

The Condition Code register contains flags that are affected by various instructions in the DSP. The bits of the Condition Code register are defined in Table 2-23.

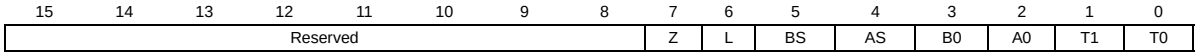

Table 2-23. Condition Code Register Bit Definitions

Bits	Field/Flag Name	Description
15:8	Reserved	Reserved.
7	Z	Zero bit - Set by the bit manipulation instructions.
6	L	Limit bit - Set when saturation occurs: after it is set, it must be cleared by software.
5	BS	B sign bit - Set when the B accumulator result is negative.
4	AS	A sign bit - Set when the A accumulator result is negative.
3	B0	B zero bit - Set when the B accumulator result is zero.
2	A0	A zero bit - Set when the A accumulator result is zero.
1:0	T1, T0	Shift mode status bits. T1 and T0 are set depending on the [63:59] bits of the accumulator and the value of s1 and s0 in the MR. See example in Table 2-24.

Example of how T1 and T0 are affected by various accum + shift values:

Table 2-24. T1, T0 with Various Accum + Shift Values

Accum values [63:59]			T1	T0		Shift
00000	or	11111	0	0		No shift
00001		11110	0	0		No shift
00010		11101	0	0		No shift
00011		11100	0	1		Shift
00100		11011	0	1		Shift
00101		11010	0	1		Shift
0011x		1100x	1	0		Shift twice
01xxx		10xxx	1	0		Shift twice
			1	1		Not used

2.4.21 Loop Stack Example

The operation of the loop stack is best illustrated by working through an example. Though the loop stack contains only eight entries, it is possible to extend it using software to initialize and manage a much larger software stack. In this way, the loop stack appears larger to the software that uses it. In this example, the software consists of some initialization code and a stack overflow and stack underflow exception handler. The code that follows represents one possible way to implement a software loop stack. Better and more complex approaches may be used. In particular, more error handling might be added or some hysteresis put in to the underflow/overflow interaction to minimize stack exceptions in certain cases.

Note that the subroutine stack operates similarly to the loop stack, except that it has 16 entries and its entries are each 16 bits.

Copyright 2013 Cirrus Logic

For the loop stack, several factors do not vary:

- **mr_lst_ptr** always points to the next stack entry to be pushed and/or the last entry popped.
- **lp_state** is the next data to be pushed and/or the last entry popped.
- **lst_data** is always the data at entry **(mr_lst_ptr - 1) mod 8**.
- An overflow exception occurs when data is pushed onto the stack such that **mr_lst_ptr** is incremented past the overflow threshold and the exception is enabled, that is, all of the following are true:
  - **lst_mode.lst_ovf_imask = 1**
  - **lst_mode.lst_int_en = 1**
  - **(mr_lst_ptr - 1) mod 8 = lsf_ovf**
  - previous clock cycle **mr_lst_ptr = lst_ovf**
- An underflow exception occurs when data is popped from the stack such that **mr_lst_ptr** is decremented to the underflow threshold, and the exception is enabled, that is, all of the following are true:
  - **lst_mode.lst_unf_imask = 1**
  - **lst_mode.lst_int_en = 1**
  - **mr_lst_ptr = lsf_unf**
  - previous clock cycle **(mr_lst_ptr - 1) mod 8 = lst_unf**
- At hardware reset, **lp_state** is 0x0000000. This means that the first data pushed onto the loop stack is always meaningless and should not be used.

Consider Figure 2-15 and Figure 2-16, which show the hardware related to the loop stack as it changes over time. Each figure shows:

- The **loop stack**, an eight entry circular buffer for holding 49-bit data. The entries are numbered 0 through 7.
- The **lp_state** register for holding 4- bit data.
- the **lst_data** register, which is just one of the entries in the **loop stack**.
- the **lst_unf**, **lst_ovf**, and **mr_lst_ptr** values, represented by arrows. **mr_lst_ptr** is the unlabeled one.

The 49-bit data is flow control management information for either a **do** loop or **do_patch** instruction. The particular values are not important here. Instead, this data is represented by unique capital letters in the figures.

To make use of the loop stack, it is assumed in this example that, during normal operation, **lst_unf = (lst_ovf + 1) mod 8**. This setting allows overflow and underflow to be detected properly. The left half of each diagram represents states before an overflow or underflow ISR, while the right half shows states after an overflow or underflow ISR.

Copyright 2013 Cirrus Logic

2

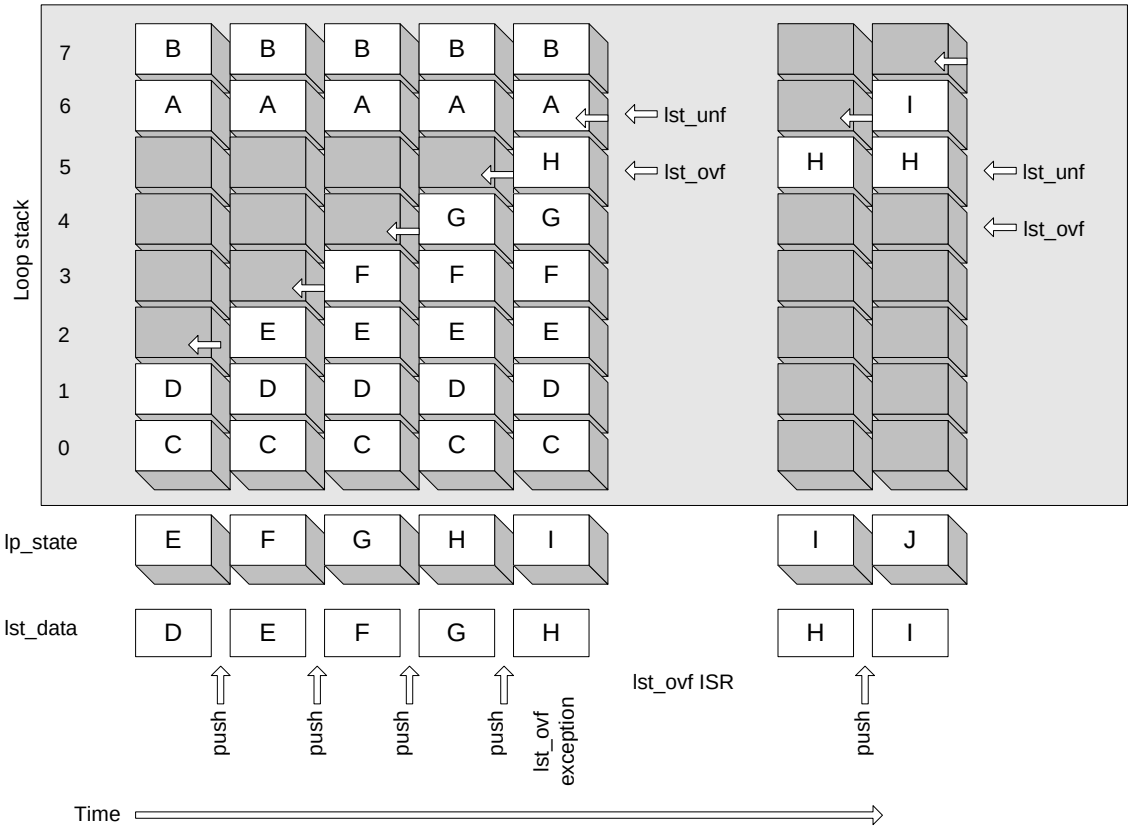

Figure 2-15. Loop Stack Overflow Example

Figure 2-15 illustrates how the hardware loop stack could overflow, causing seven entries to be pushed to the software stack. The loop stack initially contains four valid entries, A through D. Over time, four more entries, E through H, are pushed onto the stack. At this point, an overflow exception occurs. The overflow exception handler pushes seven entries in the hardware stack, A through G, onto the software stack. It also updates the overflow and underflow thresholds. Upon return from the ISR, the loop stack has seven free spaces instead of none, and **lp_state** and **lst_data** are unchanged. Then, over time, another entry, I, is pushed onto the stack.

Copyright 2013 Cirrus Logic

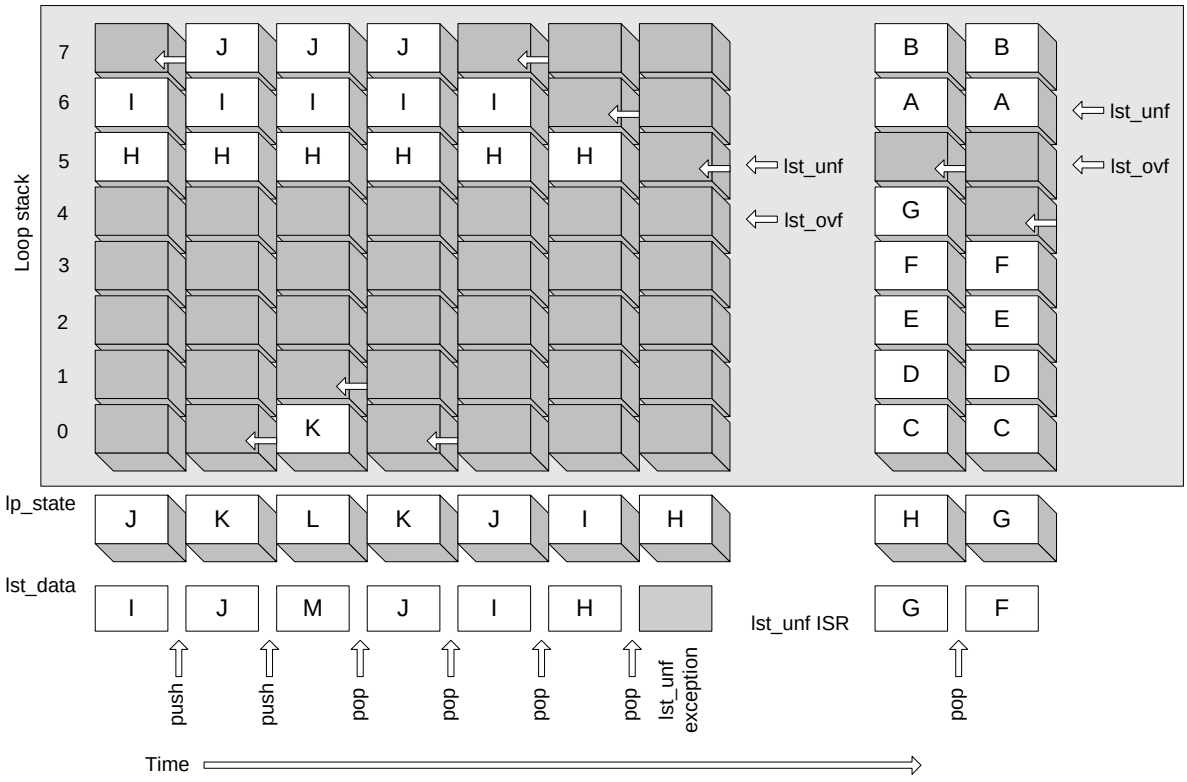

Figure 2-16. Loop Stack Underflow Example

Figure 2-16 shows how the hardware loop stack could then underflow, causing the same seven entries to be removed from the software stack and restored to the hardware stack. The loop stack initially contains two valid entries, H and I. Over time, two more entries, J and K, are pushed onto the stack. Then, four entries, H through K, are popped. At this point, an underflow exception occurs. The underflow exception handler pops seven entries from the software stack, A through G, and places them back in the hardware loop stack. It also updates the overflow and underflow thresholds. Upon return from the ISR, the loop stack has one free space instead of eight, **lp_state** is unchanged, and **lst_data** contains the next entry to be popped. Over time, another entry, G, is popped from the stack.

The examples below provide example code to set up and control the hardware and software loop stack. First consider the set-up code. The code sets the stack ISR base, and initializes the list stack hardware configuration such that:

- The **mr_lst_ptr** auto increments on read and write.
- Overflow interrupt is enabled.
- Underflow interrupt is not enabled for now since the stack is assumed to be empty.
- The overflow threshold is set such that on the eighth push, an exception occurs.
- The underflow threshold is set such that if a pop empties the stack, an exception would occur if enabled.

Copyright 2013 Cirrus Logic

If the underflow interrupt is enabled when the stack is empty, every time the stack becomes empty again, an underflow interrupt would occur, though underflow has actually not occurred.

The **regs_stack** is a small X-memory area for storing at least DSP register **i11** in all stack ISRs. The **isr_stack** is a small X-memory area for storing more DSP registers within all stack ISRs. The **soft_lst_ptr** is an X-memory address whose value points to the top of the software list stack, initially at address 0x100 in both X and Y memory. The software stack will grow upward from address 0x100 in both X and Y memory as the hardware stack overflows, and shrink downward as the hardware stack underflows.

Example 2-3

```
# Initialization
regs_stack    .bss  (8)         # Stores i11 throughout stack ISRs
isr_stack     .bss  (8)         # Stores other registers during all stack ISRs
soft_lst_ptr  .bsc  1, (0x100)  # Stores list stack entries
stq_base  =  (0x0024)   # Location os ISR table
lst_mode  =  (0xfa)     # Enable auto inc on rd/wr, ovf exception
lst_unf   =  (0x00)     #
lst_ovf   =  (0x07)     #
```

The code starting at address 0x0024 should be something like the following:

Example 2-4

```
# ISR table 0x0024 through 0x0025
        callint_stq ISR_lst_unf
        callint_stq ISR_lst_ovf
```

This set-up insures that the list stack underflow and overflow ISRs will be called properly.

The overflow ISR takes the following actions:

- Saves some registers so they can be reused locally.
- Updates the overflow and underflow thresholds by decrementing each by one.
- Saves seven stack entries by copying them from the hardware stack to the software stack.
- Updates the software list stack pointer in memory by incrementing by seven.
- Enables underflow interrupts.
- Restores saved registers.
- Determine how to finish the ISR.

After the ISR, the **lp_state**, **mr_lst_ptr**, and **lst_data** registers are unchanged, but seven entries are now available for pushing onto the hardware list stack. Looking at Figure 2-15, the value of *N* as used in the code example is 5.

Copyright 2013 Cirrus Logic

Example 2-5

```
# loop stack overflow ISR
#
ISR_lst_ovf:
        xmem[regs_stack] = i11      # Push i11
        i11 = xmem[isr_stack]       # i11 is ISR stack pointer
        xmem[i11] = ccr; i11+=1     # Push registers to isr_stack
        xmem[i11] = x0; i11+=1

ISR_lst_ovf_jmp:                    # Entry point from call stack ISRs
        xmem[i11] = i0; i11+=1      # Push register to isr_stack

        i0 = xmem[soft_lst_ptr]     # i0 is software list stack pointer

        # Save seven entries from hardware stack to software stack.
        # Update lst_unf and lst_ovf by decrementing each by 1.
        # Assume lsf_ovf = N, lst_unf = mr_lst_ptr = (N+1) mod 8
        x0 = lst_data2              # increment stack pointer to (N+2) mod 8
        xmem[i0] = lst_data1        # save stack entry (N+2) mod 8
        ymem[i0] = lst_data2;i0+=1
        xmem[i0] = lst_data1        # save stack entry (N+3) mod 8
        ymem[i0] = lst_data2;i0+=1
        xmem[i0] = lst_data1        # save stack entry (N+4) mod 8
        ymem[i0] = lst_data2;i0+=1
        xmem[i0] = lst_data1        # save stack entry (N+5) mod 8
        ymem[i0] = lst_data2;i0+=1
        xmem[i0] = lst_data1        # save stack entry (N+6) mod 8
        ymem[i0] = lst_data2;i0+=1
        x0 = mr_lst_ptr             # x0 is (N+7) mod 8
        lst_ovf = x0                # decrement overflow threshold to (N+7) mod 8
        xmem[i0] = lst_data1        # save stack entry (N+7) mod 8
        ymem[i0] = lst_data2;i0+=1
        x0 = mr_lst_ptr             # x0 is N
        lst_unf = x0                # decrement underflow threshold to N
        xmem[i0] = lst_data1        # save stack entry (N+7) mod 8
        ymem[i0] = lst_data2;i0+=1  # stack pointer becomes (N+1) mod 8
        xmem[soft_lst_ptr] = i0     # update software list stack pointer by
                                    # incrementing by 7

        i11-=1                      # Restore register
        i0 = xmem[i11]

        lst_mode = (0x00fc)         # Enable list stack underflow interrupt

        x0 = mr                     # Clear list stack overflow request
        bitclr hi(x0), (0x0002)
        mr = x0
        jmp ISR_lst_finish          # Finishing is same for both list ISRs
```


Copyright 2013 Cirrus Logic

The underflow ISR takes these actions:

- Saves some registers so they can be reused locally.
- Updates the software list stack pointer in memory by decrementing by seven.
- Updates the overflow and underflow thresholds by incrementing each by one.
- Restores seven stack entries by copying them from the software stack to the hardware stack.
- Checks if the software stack is empty; if so, disable underflow interrupts.
- Restores saved registers.
- Determine how to finish the ISR.

After the ISR, the **lp_state** register and **mr_lst_ptr** register are unchanged, and the value apparent in **lst_data** is what is expected, as seven entries have just been restored to the hardware list stack. Looking at Figure 2-16, the value of *N* as used in the code example is 5.

Copyright 2013 Cirrus Logic

Example 2-6

```
# loop stack underflow ISR
#
ISR_lst_unf:
        xmem[regs_stack] = i11      # Push i11
        i11 = xmem[isr_stack]       # i11 is ISR stack pointer
        xmem[i11] = ccr; i11+=1     # Push registers to isr_stack
        xmem[i11] = x0; i11+=1

ISR_lst_unf_jmp:                    # Entry point from call stack ISRs
        xmem[i11] = x1; i11+=1      # Push registers to isr_stack
        xmem[i11] = i0; i11+=1
        xmem[i11] = b0; i11+=1
        xmem[i11] = b1; i11+=1

        i0 = xmem[soft_lst_ptr]     # i0 is software list stack pointer
        nop
        i0 = i0-(7)
        xmem[soft_lst_ptr] = i0

        # Restore seven entries from software stack to hardware stack.
        # Update lst_unf and lst_ovf by incrementing each by 1.
        # Asume lst_ovf = (N-1) mod 8, lst_unf = mr_lst_ptr = N.
        x0 = mr_lst_ptr              # x0 is entry just popped
        lst_ovf  = x0                # increment overflow threshold to N
        x1 = lst_data2               # increment stack pointer to (N+1) mod 8
        x1 = mr_lst_ptr              # x1 is (N+1) mod 8
        lst_unf = x1                 # increment underflow threshold to (N+1) mod 8
        lst_data1 = xmem[i0]         # restore stack entry (N+1) mod 8
        lst_data2 = ymem[i0];i0+=1
        lst_data1 = xmem[i0]         # restore stack entry (N+2) mod 8
        lst_data2 = ymem[i0];i0+=1
        lst_data1 = xmem[i0]         # restore stack entry (N+3) mod 8
        lst_data2 = ymem[i0];i0+=1
        lst_data1 = xmem[i0]         # restore stack entry (N+4) mod 8
        lst_data2 = ymem[i0];i0+=1
        lst_data1 = xmem[i0]         # restore stack entry (N+5) mod 8
        lst_data2 = ymem[i0];i0+=1
        lst_data1 = xmem[i0]         # restore stack entry (N+6) mod 8
        lst_data2 = ymem[i0];i0+=1
        lst_data1 = xmem[i0]         # restore stack entry (N+7) mod 8
        lst_data2 = ymem[i0];i0+=1   # stack pointer becomes N

        b0 = 0                       # Disable underflow interrupts
        lo16(b0) = (soft_lst1 + 7)   # if software loop stack is empty
        b1 = i0
        b0 - b1
        if (b != 0) jmp ISR_lst_unf_finish
        lst_mode = (0x00f8)
```


Copyright 2013 Cirrus Logic

```
ISR_lst_unf_finish:
        i11-=1                          # Restore some saved registers
        b1 = xmem[i11]; i11-=1
        b0 = xmem[i11]; i11-=1
        i0 = xmem[i11]; i11-=1
        x1 = xmem[i11]

        x0 = mr                         # Clear list stack underflow request
        bitclr hi(x0), (0x0001)
        mr = x0

        jmp ISR_lst_finish              # Finishing is same for both list ISRs
```

Each list stack ISR ends the same way–a decision must be made between several options:

- **jmp** directly to a call stack interrupt handler:
    - overflow
    - underflow
- Return to interrupted code with stack interrupts enabled and:
    - Interrupts enabled
    - Interrupts disabled

To determine if it is necessary to jump directly to a call stack interrupt, it is only necessary to check the state of the **mr_jsr_ovf** and **mr_jsr_unf** bits. Otherwise, to determine whether to re-enable interrupts when returning to interrupted code, it is only necessary to check the **mr_int_p** bit. Example 2-7 illustrates these decisions as well as restoring registers as needed.

Copyright 2013 Cirrus Logic

Example 2-7

```
ISR_lst_finish:
        bitchg hi(x0), (0x000f)      # If call overflow/underflow, handle ...
        nop
        bittst hi(x0), (0x000c)
        if (z==0) jmp ISR_lst_pending_stq

        x0 = mr                      # No other stack interrupt.
        bittst lo(x0), (0x0800)      # If called with interrupts enabled, handle ...
        if (z==0) jmp ISR_lst_ret

ISR_lst_retint_stq:                  # Return with interrupts enabled.
        i11-=1                       # Restore more registers
        x0 = xmem[i11]; i11-=1
        ccr = xmem[i11]
        xmem[isr_stack] = i11        # Remember isr_stack pointer
        i11 = xmem[regs_stack]       # Restore i11
        retint_stq

ISR_lst_ret:                         # Return with interrupts disabled
        x0 = lst_mode                # Enable list stack interrupts.
        bitset lo(x0), 0x0002
        lst_mode = x0
        x0 = jsr_mode                # Enable call stack interrupts
        bitset lo(x0), 0x0002
        jsr_mode = x0
        i11-=1                       # Restore more registers
        x0 = xmem[i11]; i11-=1
        ccr = xmem[i11]
        xmem[isr_stack] = i11        # Remember isr_stack pointer
        i11 = xmem[regs_stack]       # Restore i11
        ret

ISR_lst_pending_stq:                 # Call stack interrupt pending.
        bittst hi(x0), (0x0008)          # Which one?
        if (z==0) jmp ISR_jsr_ovf_jmp    # If call overflow, do it.
        jmp ISR_jsr_unf_jmp              # else do call underflow.
```

Note that both list stack ISRs share the same startup code. They save **i11**, **ccr**, and **x0**, and update **i11** to point to the **isr_stack**. Similarly, on return from either handler, these registers are restored.

The call stack ISRs should be very similar to the list stack ISRs, differing in the size of the hardware stack and the location of relevant status and control bits. Hence, it should be assumed in this example that **i11**, **ccr**, and **x0** are still saved when jumping directly from a call stack ISR to a jump stack ISR or vice versa.

In the interrupted code, only one list stack interrupt and one call stack interrupt should ever occur at the same time. The stack ISRs themselves do not utilize the list stack or call stack directly, that is, there are no **calls**, **do_patch**, or **do** loops within these ISRs. Hence, stack ISRs do not ever trigger more stack ISRs.

Copyright 2013 Cirrus Logic

2.5 Master State Registers (MSREGS)

The Master State Registers are registers that exist within the core, but are separate from the AGU and ALU/MAC registers. These registers control internal configuration, provide visibility into the current state. Specific full-word instructions exist for reading and writing the Master State Registers to and from memory, peripheral space, and other registers. Immediate data loads and the Bit Manipulation instructions also work with the Master State Registers. In all instructions, the Master State Registers are referred to in the syntax by their name as specified in Table 2-25:

```
x0 = page_p
bittst (mr), (0x0002)
search_latch = xmem[0x1234]
 ]
```

Table 2-25. Master State Registers

Register	Syntactical Name
Shift bits of Mode Register (S1 S0)	mr_s
Round bits and right shift bits (Ls R1 R0)	mr_r
Round bits and shift bits (Ls R1 R0 S1 S0)	mr_sr
Condition Code Register	ccr
Stack base address	stq_base
Call stack mode register	jsr_mode
Loop stack mode register	lst_mode
Search Count	search_cnt
Call stack pointer from the Mode Register	mr_jsr_ptr
Call stack overflow value	jsr_ovf
Call stack underflow value	jsr_unf
Search Latch register	search_latch
Loop stack pointer from the Mode Register	mr_lst_ptr
Loop stack overflow value	lst_ovf
Loop stack underflow value	lst_unf
Reserved	N/A
P Page for external memory	page_p
X Page for external memory	page_x
Y Page for external memory	page_y
Random Number Register	rand
See Section 2.5.2 on page 37.	rand_reset
Dither register A	rand_a
Dither register B	rand_b
Top of loop stack; 31:16 last address, 15:0 first address	lst_data1
Top of loop stack; 15:0 cnt	lst_data2
Current loop value; 31:16 last address, 15:0 first address	lp_data1
Current loop value; 15:0 cnt	lp_data2
Program Counter	pc
Program Counter for Breakpoints	pc_bp
PC value at top of call stack	jsr_data

Copyright 2013 Cirrus Logic

2.5.1 Search Registers

When finding the maximum or minimum value of a buffer in memory, it is often desirable not only to know what that value is, but where it was located in the buffer. The *Search Count Register*, `search_cnt`, and *Search Latch Register*, `search_latch`, can be used to accomplish this task. Whenever a "Conditional Operation" is performed, the *Search Count Register* is incremented, and whenever a MAX or MIN instruction results in the accumulator move the search count register is copied into the search latch register. Consider the following code fragment:

Example 2-8

```
        i0  = (X_BX_data1)          # i0 set to the beginning of a buffer
        search_cnt = i0             # Search Count Register and search
        search_latch = i0           # Latch Register set to i0
        a0  = xmem[i0]; i0+=1       # find minimum of buffer, leave in b0
        b0  = a0
       do (64),>
      %: if (a0<b0) b0=a0;  a0 = xmem[i0]; i0+=1
```

At the end of the loop, the search latch register contains the address of the minimum value, such that if you then execute:

Example 2-9

```
      i1 = search_latch
      nop
      x0 = xmem[i1]
# x0 now equals b0.
```

2.5.2 Random Number Generator

The DSP core has three hardware-based random number generators. The first one, called the PSR, generates 32-bit random data from a 16-bit seed which is updated each time a random number is generated. The PSR register is the only one of the three that is readable (from a programmer's perspective). The other two 4-bit random number generators are called Dither A and Dither B and each are generated independently from their own 16-bit seeds. They are only used when the Mode Register bits MR[3:2], also known as MR[R1: R0], are both set and data is moved through the respective SRS A/B (Shift Round Saturate). The purpose of setting MR[3:2] is to select the "add dither and truncate" mode so dither can be added to the lower-order bits of the accumulator as it passes through the SRS.

Copyright 2013 Cirrus Logic

New random numbers from the PSR are generated by reading the MSREG rand. By default, the seed for the PSR will be 0x0000 unless the MSREG rand_reset is written. The MSREG rand_reset is limited to 16-bits. Any higher-order bits that are written to it will be ignored. The current PSR seed can be obtained by reading the MSREG rand_reset. Each time the PSR is read and a new random number is generated a new PSR seed will be written to the MSREG rand_reset.

The 16-bit Dither A and Dither B registers are updated with new values when data is moved through the respective SRS in the "add dither and truncate" mode. For Dither A, the move must use an An accumulator. For Dither B, the move must be use a Bn accumulator. It is not possible to use SRS A or Dither A on a Bn accumulator. Likewise, it is not possible to use SRS B or Dither B in conjunction with an accumulator. By default the seed for Dither A and Dither B will be 0x0010 and 0x0030 respectively. Each dither seed is limited to 16-bits. Any higher-order bits that are written will be ignored. The current Dither A and B registers can be read from MSREG rand_a and MSREG rand_b respectively. The values will not be updated when read by the programmer.

2.6 Interrupt Controller

The Interrupt Controller prioritizes 32 peripheral interrupt requests. The interrupt priority is fixed; 0 is highest and 32 is lowest. For a given interrupt service routine (ISR), a single instruction corresponding to that interrupt is inserted directly into the instruction pipeline. This is called an Interrupt Service Instruction (ISI). All ISIs reside in the first 32 locations of program memory (addresses `0x0000-0x001F`).

The core has three categories of interrupts. Interrupts can be generated from the DBC (debug controller), the subroutine and do-loop stacks, and the PIC (peripheral interrupt controller). The DBC has the highest priority. Second is priority are the stack interrupts. Last in priority are the standard interrupts from the PIC.

2.6.1 Fast Interrupts

Interrupts that consist solely of a single instruction are referred to as *fast* interrupts.

2.6.2 Long Interrupts

If an interrupt needs to execute more than one instruction, the *callint* instruction is used for the ISI. This is referred to as a long interrupt. The *callint* instruction disables interrupts, pushes the program counter (PC) onto the subroutine stack, and starts executing the specified ISR. The final instruction of the ISR should be *ret_int*, which pops the PC and enable interrupts. The *call* or *jmp* instructions can also be used as ISIs, but they will not disable interrupts, allowing the possibility of code re-entrance.

2.6.3 Masking

There are two 32-bit registers that govern interrupt operation: *IMask* and *IRMask*. They are accessible from the peripheral space as *imask* and *irmask*.

Copyright 2013 Cirrus Logic

2.6.3.1 IMask

IMask is the interrupt enable/disable mask. Every bit corresponds to 1 of 32 possible interrupts numbered 31-0 and corresponding to mask bits [31:0]. If a bit is *1*, then that interrupt is enabled. The default value after reset is `0x0000`.

2.6.3.2 IRMask

IRMask is the interrupt "run" mask, which affects how the DSP handles interrupts while in the *Halt* state. If an IRMask bit (31-0) corresponding to a particular interrupt is *1*, then execution of that ISI will bring the processor out of *Halt*. Otherwise, the instruction is executed without the processor being brought out of *Halt* with no further instructions occurring, even if the interrupt instruction is a *callint*, *call*, or *jmp*. For this reason, long interrupts that might be triggered while the processor is in *Halt* should have the corresponding bit in the IRMask set. The default value after reset is `0x0000`.

2.7 Instruction Restrictions

There are some cases where certain combinations of instructions which affect MSREGs can produce an undesired result. These cases are limited to the modification of any MSREG by two different, but overlapping operations. In order to guarantee this problem will not occur, MSREG modifications should be avoided one cycle after or before any bits in the same register could be affected by an operation. Simply add a NOP before or after any MSREG access to avoid this problem.

For example, a conditional jump could be taken incorrectly if first, the Condition Code register bits are set by a bitwise compare (that is, An - Am) and second, the Condition Code register is modified by a MSREG write (that is,. bitset (ccr), (1<<6)). After the first and second instructions have completed the CCR may not contain the intended sign and zero flag values since the bitset instruction literally performs a read-modify-write operation on the CCR. The read occurs before the result of the bitwise compare is stored in the CCR. After the CCR is modified by the ALU operation, the final modify-write operation completes from the bitset instruction and thus corrupts the state of the condition flags which are necessary for the following conditional jump. Example 2-10 illustrates one scenario with both bad and good coding styles.

Copyright 2013 Cirrus Logic

2.7.1 Code Example, Broken Code

Example 2-10

```
# sample broken code
A0 - A1
bitset (ccr), (1<<6)       # Supposedly we only want to
                           # modify the limit bit but the
                           # whole register must be read then
                           # written back

if (condition) jmp success # this may or may not work
                           # depending on the previous state
                           # of the CCR and the result from
                           # the bitwise compare.
                           # Either way the real result in the
                           # CCR was overwritten by the bitset.
```

2.7.2 Code Example, Fixed Code

Example 2-11

```
# sample fixed code
A0 - A1
nop                        # Added one NOP before a direct MSREG
                           # modification
bitset (ccr), (1<<6)
if (condition) jmp success
```

2.7.3 Successive but Orthagonal Operations that Affect the CCR

It is possible to have successive but orthagonal operations that affect the condition code register. For example, performing an addition or subtraction operation with the "A" accumulators affects the AS and A0 bits but not the BS or B0 bits and vice versa. The following code illustrates this behavior:

```
##Starting state: The CCR = 0

uhalfword(a0)=(0)
uhalfword(a1)=(1)
a2=a0-a1 #only the AS and A0 bits are affected
uhalfword(b0)=(0)
uhalfword(b1)=(1)
b2=b0-b1 #only the BS and B0 bits are affected
if (b<0) jmp>do_something
```


Copyright 2013 Cirrus Logic

2.7.4 If Statements and the CCR

If statements, such as "if (a<0)" and "if (b==0)" do not alter the contents of the CCR. Therefore it is possible to have consecutive if statements, as shown in the following example:

```
uhalfword(a0)=(0)
uhalfword(a1)=(1)
a2=a0-a1
if (a < 0) jmp > process_channel_0 ### if statements do not alter the CCR.
if (a == 0) jmp > process_channel_1 ### so a second if statement can be placed
here
#process channel 2 ### and the "else" operations can begin here
..
..
ret
%process_channel_0:
..
..
ret
%process_channel_1:
..
..
```


Copyright 2013 Cirrus Logic

Chapter 3
Full Word Instructions

3.1 Assembly Language Syntax

The length of the DSP program word is 32-bits, and the assembler allows one 32-bit program word per line. Some instructions use 32 bits of the program word (long word instructions) while others use 16 bits of the program word (short word instructions). When there are two parallel data moves in an instruction, each parallel move uses 8 bits of the 16-bit short word instruction. Any parallel move(s) (16 bits) can be combined with any arithmetic or logic instruction (16 bits) to form a complete 32-bit instruction word. See Figure 3-1. Only labels can occupy the first column of the line; the instruction may be located anywhere else within the line.

Optional Label	Full Word Instruction	Optional Comment
—	(32-bits)	—
`label:`	`if (a!=0) jmp >`	`# comment`

Optional Label	Arithmetic, Accumulator or Logic Instruction	Optional Parallel Move	Optional Comment
—	(16-bits)	(16-bits)	—
`label:`	`a0+=x0*x2; b0+=x0*y2;`	`x0,y0=xymem[i0];i0-=n`	`# comment`

Optional Label	Arithmetic, Accumulator or Logic Instruction	X Memory Data Move	Y Memory Data Move	Optional Comment
—	(16-bits)	(8 bits of parallel move)	(8 bits of parallel move)	—
`label:`	`b3=0;`	`x3=xmem[i0];`	`ymem[i4]=b0;i4+=1`	`# comment`

Figure 3-1. Assembler Example: 32-bit Instruction Word

Some arithmetic instructions allow dual accumulator destinations. For example, the instruction:

```
a3=a1+=x2*x2
```

translates to, "Square x2, add result of multiplication to a1, store final result in a1 and a3." In this case the previous value of a3 is irrelevant. The valid accumulator destination value pairs are:

- 1 and 0
- 0 and 2
- 3 and 1
- 2 and 3

Copyright 2013 Cirrus Logic, Inc

3.2 Conventions

The following conventions for the use of certain syntax terms used in this manual are explained in Table 3-1.

Table 3-1. Syntax Terms Used in this Manual

Terms	Definitions
Accum	Any Accumulator (a0-a3 or b0-b3)
Any Reg	Any Register (x0-x3, y0-y3, a0-a3, a0h-a3h, a0l-a3l, b0-b3, b0h-b3h, b0l-b3l, i0-i11, or nm0-nm11)
DP Reg	Any Data Path Register (x0-x3, y0-y3, a0-a3, or b0-b3)
MS Reg	ccr, dbc_cmd, dbc_d1, dbc_d2, dbc_io, dbc_status, iic_addr, iic_mask, jsr_data, jsr_mode, jsr_ovf, jsr_unf, lp_data1, lp_data2, lst_data1, lst_data2, lst_mode, lst_ovf, lst_unf, mr, mr_jsr_ptr, mr_lst_ptr, mr_r, mr_s, mr_sr, page_p, page_x, page_y, pc, pc_bp, rand, rand_a, rand_b, rand_reset, rx_in, search_cnt, search_latch, stq_base

Those parts of instructions that appear in the format:

```
a0 ±= x0*y0
```

are optional, which means that the instruction can take any of the following forms:

```
a0 = x0*y0
a0 += x0*y0
a0 -= x0*y0
```

3.3 Execution Control Instructions

3.3.1 do - Start Hardware Loop

Repeat a set of instructions *count* times, from the instruction following DO (first address) through instruction at label (last address). *Count* is either a 10-bit immediate value or the 16-bit value in an index register. A *count* of zero is not allowed. Valid values for the 10-bit immediate number are 1-1024, where 1024 is encoded in the instruction word as zero. Upon finishing the last instruction of the last iteration of the loop, the PC is set to the first instruction following the last address of the loop as specified in the DO instruction. This means that nested do-loops cannot share a last address.

Assembler Syntax 1:

```
do (Index Register), label# i = 0 to 11
do (Index Register), >
where:
Index Register = i0, i1,...,i7
```

Examples:

```
#example using a label
  do (i0), label
label: nop

#example using a local symbol
  do (i2), >
%: nop
```


Copyright 2013 Cirrus Logic, Inc.

Flags Affected:

None

Assembler Syntax 2:

```
do (10-bit count), label
do (10-bit count), >
```

Examples:

```
#example using a label
   do 1024, label
label: nop


#example using a local symbol
   do 1024, >
%: nop
```

Flags Affected:

None

3.3.2 enddo - End Current Do-Loop

Pops the do-loop stack pointer.

Assembler Syntax:

```
enddo
```

Example:

```
enddo
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

3.3.3 do_patch - Jump to Patch

Jump to a set of instructions, then start at a first address and execute through a last address or for a specific number of cycles. Upon finishing the last instruction, the PC is set to the first instruction following the do_patch instruction. Nested do-loops or do-patches cannot share a last address. The do_patch instruction allows a programmer to point to and run a piece of patch code in another location in the code or in ROM.

There are two forms of the do_patch instruction.

- Form one uses a 10-bit immediate value, count, that specifies the number of instructions in the patch. That is, the last address of the patch is calculated as:

  (first address + count - 1) modulo 0x10000.

  Valid values for count are 1-1024.

- Form two uses a 16-bit value in an index register to specify the last address of the patch. Note that this is not an instruction count but an absolute address.

The do_patch instruction utilizes the same loop stack as the do instruction.

CAUTION: Do not execute an enddo instruction with a do_patch without also being within a do-loop in the patch, as the resulting behavior is unpredictable.

Assembler Syntax 1:

```
do_patch label, (10-bit count)
```

Example 1:

```
start:  nop
end:    nop
        do_patch Start, (2)
```

Flags Affected:

None

Note: Valid values are 1-1024. 1024 is encoded as 0.

Assembler Syntax 2:

```
do_patch label, (Index Register)
where:
Index Register = i0, i1,...,i7
```

Example 2:

```
start:  nop
end:    nop
i0 = end
nop
        do_patch Start, (i0)
```


Copyright 2013 Cirrus Logic, Inc.

Flags Affected:

None

3.3.4 jmp - Jump

Jump to 16 bit immediate address or to the address specified in the index register. The index register can be updated.

Assembler Syntax 1:

```
jmp label
```

Example 1:

```
jmp label
jmp <
```

Flags Affected:

None

Assembler Syntax 2:

```
jmp [;In <register update.]
```

Example 2:

```
jmp (i0)
jmp (i2); i2+=2
```

Flags Affected:

None

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Note: Where n is the offset value in the Modulo-Offset register corresponding to the specified I register (that is, i0 implies nm0).

Copyright 2013 Cirrus Logic, Inc

3.3.5 if - Jump Conditionally

Jump conditionally to "label." The PC will be updated with the address in "label" when the condition is true.

See Section 5.2. It may be helpful to review the instructions contained in this section as they are often used to set the condition required for jumping. The instructions that do not modify the contents of the accumulators being compared can be particularly useful.

Assembler Syntax:

```
if (condition) jmp label
```

Example:

```
   if (a==0) jmp label
   if (!limit) jmp >
label:
   nop
%:
   nop
```

Flags Affected:

None

Restrictions:

```
a == 0
a != 0
a  < 0
a >= 0
a <= 0
a  > 0
z != 0
z == 0
b == 0
b != 0
b  < 0
b >= 0
b <= 0
b  > 0
```

limit (limit bit set in Modulo-Offset register)

!limit (limit bit not set in Modulo-Offset register)

Copyright 2013 Cirrus Logic, Inc.

3.3.6 call - Jump To Subroutine

Jump unconditionally to subroutine in 16 bit address label or the address in the index register. The index register can be updated. Only the PC is saved on the stack.

Assembler Syntax 1:

```
call label
```

Example 1:

```
call label
call >
```

Flags Affected:

None

Assembler Syntax 2:

```
call (Index Register); index register update
where:
Index Register = i0, i1,...,i7
```

Example 2:

```
call (i7); i7-=2
```

Flags Affected:

None

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```


Copyright 2013 Cirrus Logic, Inc

3.3.7 callint - Answer Interrupt

Identical to the CALL instruction, except that interrupts are disabled. It can only be used with a 16-bit address – there is no index register mode. Uses the subroutine stack.

Assembler Syntax:

```
callint label
```

Example:

```
callint label
```

Flags Affected:

None

3.3.8 callint_stq - Answer Stack Interrupt

Identical to the CALL instruction, except that standard interrupts and stack interrupts are disabled. It can only be used with a 16-bit address – there is no index register mode. Uses the subroutine stack.

Assembler Syntax:

```
callint_stq label
```

Example:

```
callint_stq label
```

Flags Affected:

None

3.3.9 ret - Return From Subroutine

Return from subroutine.

Pops the return address from subroutine stack and assigns it to the PC.

Assembler Syntax:

```
ret
```

Example:

```
retFlags Affected:
```

None

Copyright 2013 Cirrus Logic, Inc.

3.3.10 retint - Return From Interrupt

Return from interrupt.

Pops the return address from subroutine stack, assigns it to the PC, and enables interrupts.

Assembler Syntax:

```
retint
```

Example:

```
retint
```

Flags Affected:

None

3.3.11 retint_stq - Return From Stack Interrupt

Return from stack interrupt.

Pops the return address from subroutine stack, assigns it to the PC, enables interrupts, and enables stack interrupts.

Assembler Syntax:

```
retint_stq
```

Example:

```
retint_stq
```

Flags Affected:

None

3.3.12 inten - Enable Interrupts

Enables interrupts.

Assembler Syntax:

```
inten
```

Example:

```
inten
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

3.3.13 intdis - Disable Interrupts

Disables interrupts.

Assembler Syntax:

```
intdis
```

Example:

```
intdis
```

Flags Affected:

None

3.3.14 halt - Stop Further Execution

Stop execution and enter low-power wait state.

Assembler Syntax:

```
halt
```

Example:

```
halt
```

Flags Affected:

None

3.3.15 nop - No Operation

Perform no operation.

Assembler Syntax:

```
nop
```

Example:

```
nop
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc.

3.3.16 _breakpt - Breakpoint Instruction

Stop execution and enter low-power wait state.

Assembler Syntax:

```
_breakpt
```

Example:

```
_breakpt
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

3.4 64-bit Peripheral Moves

3.4.1 XY Register Pair = ext(16-bit Address)

64-bit data transfer from peripheral space to an XY register pair.

Assembler Syntax:

```
XnYn Register Pair = ext(16-bit Address)
where:
x=0,1,...,3, y=0,1,...,3
```

Example:

```
x0,y0 = ext(0x0010)
```

Flags Affected:

None

3.4.2 Accum = ext(16-bit Address)

64-bit, sign-extended data transfer from peripheral space to an accumulator.

Assembler Syntax:

```
Accum = ext(16-bit Address)
where:
Accum = a0,a1,...a3, b0,b1,...ba3
```

Example:

```
a0 = ext(0x1234)
```

Flags Affected:

None

Restrictions:

Register:

```
x0,y0- x3,y3
a0 - a3
b0 - b3
```


Copyright 2013 Cirrus Logic, Inc.

3.4.3 ext(16-bit Address) = XY Register Pair

64-bit data transfer from an XY register pair to peripheral space.

Assembler Syntax:

```
ext(16-bit Address) = XY Register Pair
```

Example:

```
ext(0x0010) = x1,y1
```

Flags Affected:

None.

3.4.4 ext(16-bit Address) = Accum

64-bit data transfer from an accumulator to peripheral space. Data from an accumulator does pass through the SRS unit and is affected accordingly.

Assembler Syntax:

```
ext(16-bit Address) = Accum
```

Example:

```
ext(0x0010) = x1,y1
ext(0x1234) = b3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: After the L, T1, and T0 flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.

Restrictions:

Register:

```
x0,y0- x3,y3
a0 - a3
b0 - b3
```


Copyright 2013 Cirrus Logic, Inc

3.4.5 logexp = XY Register Pair

Perform primitive operations that may be used to approximate divide, power, and square root functions. Data can be sourced from the XY registers or from the X or Y input mux. Operations are pipelined in two clock cycles so data can not be read until after one cycle. Since the operation is pipelined, a second operation can be started before the first one completes.

Assembler Syntax:

```
logexp X=Cmd_X(Mux_X(Xn))  Y=Cmd_Y(Mux_Y(Yn))
logexp X=Cmd_X(Mux_X(Xn))  Y=Cmd_Y(X-Y)
logexp X=Cmd_X(Mux_X(Xn))  Y=Cmd_Y(X>>1)
logexp X=Cmd_X(Mux_X(Xn))  Y=Cmd_Y(Xn)
logexp X=Cmd_X(X-Y)        Y=Cmd_Y(Mux_Y(Yn))
logexp X=Cmd_X(X-Y)   Y=Cmd_Y(X-Y)
logexp X=Cmd_X(X-Y)   Y=Cmd_Y(X>>1)
logexp X=Cmd_X(X-Y)   Y=Cmd_Y(Xn)
logexp X=Cmd_X(X>>1)  Y=Cmd_Y(Mux_Y(Yn))
logexp X=Cmd_X(X>>1)  Y=Cmd_Y(X-Y)
logexp X=Cmd_X(X>>1)  Y=Cmd_Y(X>>1)
logexp X=Cmd_X(X>>1)  Y=Cmd_Y(Xn)
logexp X=Cmd_X(Xn)    Y=Cmd_Y(Mux_Y(Yn))
logexp X=Cmd_X(Xn)    Y=Cmd_Y(X-Y)
logexp X=Cmd_X(Xn)    Y=Cmd_Y(X>>1)
logexp X=Cmd_X(Xn)    Y=Cmd_X(Xn)
```

Example:

```
logexp X=log(norm64(x0))   Y=log(norm32(y0))
logexp X=exp(X-Y)          Y=sm(X-Y)
nop                        # pipeline delay before reading
x0,y0 = logexp
```

Flags Affected:

None

Normalization:

X is normalized to 16 bit float point data in the following format:

```
15      14      13      12                    7   6                          0
```

```
2 sign bit + 1bit 0 + 6 bit exponent + 7 bit significand

Bit[15] = S
Bit[14] = S & norm64
Bit[13] = 0
Bit[12] is 1 if a 64-bit normalization is performed. Otherwise, it is 0.
```

The data represented by above format is: 0.[1,significant]*2^exponent

Example 1:

```
uhalfword(x0) = (0x1000) # x0 = 0x00001000
```


Copyright 2013 Cirrus Logic, Inc.

```
uhalfword(y0) = (0x100) # y0 = 0x00000100
logexpX = nop(norm32(x0)) Y = nop(norm32(y0))
nop
x1, y1 = logexp
```

x1 will have a value of 0x06800000 and y1 will have a value of 0x04800000.

The least significant 16-bits can be ignored. Expanding the most significant 16-bits in binary:

Bit	Register X1	Register Y1
15 (Sign)	0	0
14 (Sign & norm64)	0	0
7–12	001101 (13)	001001 (9)
0–6	0000000	0000000

Note: As is typical in floating point representation, after normalization, the first bit of a non-zero input is always 1, and hence, is not stored. The significant after normalization in the above example would be 0x8000–since the most significant 1 is implicit, it is not stored and hence bits 0–6 in the output are 0.

Example 2:

```
uhalfword(x0) = (0x1000) # x0 = 0x00001000
uhalfword(y0) = (0xfe0) # y0 = 0x00000FE0
logexp X = nop(norm64(x0)) Y = nop(norm32(y0))
nop
x1, y1 = logexp
```

Norm64 operates on a 64-bit number where the most significant 32-bits come from the input register, and the least significant 32-bits are implicitly assumed as 0.

Log Operation

It takes normalized float point format data N as input. It calculates log2(2*N). The result is a 9.23 number with the least significant 16-bits always being 0 (the meaningful accuracy is 9.7).

Example 1:

```
uhalfword(x0) = (0x1000) # x0 = 0x00001000 = 4096 in decimal
uhalfword(y0) = (0x1234) # y0 = 0x00001234 = 4660 in decimal
logexp X = log(norm32(x0)) Y = log(norm32(y0))
nop
x1, y1 = logexp
```

x1 will have a value of 0x06800000 and y1 will have a value of 0x06970000. Since the output is in 9.23 format, the decimal value of x1 and y1 are 13 and 13.1796875, which matches the expected output of log2(2*4096) and log2(2*4660).

Exp Operation

It takes a 9.7 log number L as input (where the 9.7 number is in the most significant 16-bits and the least significant 16 bits are ignored), and uses L's fractional part to compute (2^(0.fractional))/2. The range of the fractional part is between 0 and 127/128 (since the

Copyright 2013 Cirrus Logic, Inc

fractional part has 7 bits), and the output range will be between [(2^0)/2 = 0.5, (2^(127/128))/2 = 0.9921]. The output will be in 1.31 format with the least significant 23 bits always 0 (the meaningful accuracy is 1.8).

The result is:

```
15 14 13 ... 7   6 ... 0
0. 2^(L[6,0])/2  7-bit 0
```

Bit 14 will always be 1 since the lowest output will be (2^0)/2 = 0.5.

If the 9.7 log number > 31.0, it's over flow condition, and the result will be 0x7fff.

Note: Negative inputs will be interpreted as positive values and produce overflow.

Example 1:

```
fixed16(x0) = (0x06c0) # x0 = 0x06c00000 = 13.5 in 9.23 format
logexp X = exp(x0) Y = nop(x0)
nop
x1, y1 = logexp
```

x1 will have a value of 0x5a800000 which corresponds to 0.707 in 1.31 format. This matches (2^(0.5))/2 = 1.414/2 = 0.707.

Shift-multiply Operation

It takes a 9.7 log number L as input (where the 9.7 number is in the most significant 16-bits and the least significant 16 bits are ignored) and uses L's integer part to compute (-2^(integer_part)). The final output is a 16-bit number with the output present in the most significant 16-bits and the least significant 16-bits being set to zero. The shift-factor would be saturated if the integer part is greater than 32 and is set to 0 if the integer part is less than 16; that is, the output will be in the range [-2^16 to -2^32].

The result of SM is:

```
if(L[12]) // overflow, data > 2^32
  SM = 0x8000;
else if(L[11]==0) // underflow, data < 2^16
  SM = 0x0000;
else
  SM = -1<<L[[10:7];
if(L[15]) // data is a negative number, negate result
  SM = ~SM;
```

Example 1:

```
fixed16(x0) = (0x0ec0) # x0 = 0x0ec00000 = 29.5 in 9.23 format
logexp X = exp(x0) Y = sm(x0)
nop
x1, y1 = logexp
```

y1 will have a value of 0xe0000000 which corresponds to –(2^29).

Data Output

When outputting the 16 bit result from logexp block, it is concatenated with 16 bit 0 at the

 Copyright 2013 Cirrus Logic, Inc.

end to form a 32 bit number: {16 bit data, 16-bit 0}

Sample Usage

Here are some basic functions:

```
# compute log2(x0) using HW assist
y0 = (0x0000)
logexp X=log(norm32(x0)) Y=nop(norm64(y0)) # X = log2(x0); Y =
                                           # nop(norm64)0x0000)) ==
                                           # 0x10000000 == 32 in 9.23
logexp X=nop(X-Y) Y=nop(x0)                # subtract/compensate for bias
                                           # of 32 (Y ignored hereafter)
nop
x0,y0 = logexp                       # x0 = log2(x0) in 9.23 format
                                           #(y0 == MS 16 bits of x0)

# convert from 20*log10 to linear
#    a0 = 2^(x0*log2(10)/20)
#    x0 (input) in 8.24 format
  .ydata
I_VY_log10_to_log2             .dw .f2b(.log2(10)/(20*2))
  .code
I_S_20log10_to_Linear
# convert from 20*log10 to log2
# and move from 8.24 to 9.23 (extra shift in log10_to_log2 constant)
  a0 = (0x1000)                                        # bias == 32
  y0 = ymem[[I_VY_log10_to_log2]
  a0 += x0*y0
                                     x0 = a0
# convert back to linear (2^x)
  logexp X=exp(x0)  Y=sm(x0)
  nop
  x0,y0 = logexp
  a0 = -x0*y0
# b0 = sqrt(x0)
 logexp X=log(norm64(x0)) Y=nop(x0)
 logexp X=exp(X>>1)       Y=sm(X>>1)
 nop
 x1,y1 = logexp
 b0 = -x1*y1
 # Note, it appears that output is Q5.26 format

# cheap divide (a0 = x0/y0)

 logexp X=log(norm64(x0)) Y=log(norm32(y0))
 logexp X=exp(X-Y) Y=sm(X-Y)
 nop
 x0,y0 = logexp
 a0 = -x0*y0

# normalize x2
```


Copyright 2013 Cirrus Logic, Inc

```
  x1 = x2

# compute log2 using HW assist
 logexp X=log(norm32(x1)) Y=nop(x1)     # X = log2(x1)
                                        #Y = x1 = b
 uhalfword(y0) = (0x0100)               # used to shift 9.23 down to 32.0
 x1,y1 = logexp
 bitclr hi(x1), (0x807f)         # clear sign bit and truncate any fractions
 b1 = x1*y0                             # shift down
 uhalfword(b0) = (0x001f)               # remove bias
 b0 = b0-b1;      b1 = x2               # b1 = b
 AnyReg(i7, b0h)                        # i7 = shifts
 if (b==0) jmp >noshift
 do (i7), >
%:  b1 = b1 << 1                        # b1 = b'
%noshift
```

Restrictions:

Register:

```
x0, y0
x1, y1
x2, y2
x3, y3
```

Cmd_X[1:0]:

```
nop
sm
log
exp
```

Cmd_Y[1:0]:

```
nop
sm
log
exp
```

Mux_X[2:0]:

```
norm32 (x reg)
norm64 (x reg)
x-y
x>>1
(x reg)
```

Mux_Y[2:0]:

```
norm32 (y reg)
norm64 (y reg)
x-y
x>>1
(x reg)
```


Copyright 2013 Cirrus Logic, Inc.

3.4.6 XY Register Pair = logexp

Transfer 64-bit value into the XY register pair from the LogExp peripheral.

Assembler Syntax:

```
DP Pair = logexp
```

Flags Affected:

None

Restrictions:

Register:

```
x0, y0
x1, y1
x2, y2
x3, y3
```

3.5 Memory Moves - Direct

Note: During Direct Memory Moves, if the size of the destination is less than 32 bits, the excess upper bits of the source are ignored. If the size of the source is less than 32 bits, the excess upper destination bits are zero-filled, except for when reading guard registers (for example, a0g, b3g) that are sign extended.

3.5.1 Any Reg = xmem[16-bit Address]

Data transfer from X memory to any register. Direct addressing (16-bit) is used.

Assembler Syntax:

```
Any Reg = xmem[16-bit Address]
```

Example:

```
a0 = xmem[0x9980]
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.2 xmem[16-bit Address] = Any Reg

Data transfer from any register to X memory. Direct addressing (16-bit) is used.

Assembler Syntax:

```
xmem[16-bit Address] = Any Reg
```

Example:

```
xmem[0x0870] = b3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.3 Any Reg = ymem[16-bit Address]

Data transfer from Y memory to any register. Direct addressing (16-bit) is used.

Assembler Syntax:

```
Any Reg = ymem[16-bit Address]
```

Example:

```
a0 = ymem[0x9980]
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.4 ymem[16-bit Address] = Any Reg

Data transfer from any register to Y memory. Direct addressing (16-bit) is used.

Assembler Syntax:

```
ymem[16-bit Address] = Any Reg
```

Example:

```
ymem[0x0870] = b3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.:

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.5 Any Reg = pmem[16-bit Address]

Data transfer from program memory to any register. Direct addressing (16-bit) is used.

Assembler Syntax:

```
Any Reg = pmem[16-bit Address]
```

Example:

```
a0 = pmem[0x9980]
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.6 pmem[16-bit Address] = Any Reg

Data transfer from any register to program memory. Direct addressing (16-bit) is used.

Assembler Syntax:

```
pmem[16-bit Address] = Any Reg
```

Example:

```
pmem[0x0870] = b3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.7 Any Reg = inp[16-bit Address]

Data transfer from peripheral space to any register. Direct addressing (16-bit) is used.

Assembler Syntax:

```
Any Reg = inp[16-bit Address]
```

Example:

```
a0 = inp[0x9980]
```

Flags Affected:

None.

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.8 outp[16-bit Address] = Any Reg

Data transfer from any register to peripheral space. Direct addressing (16-bit) is used.

Assembler Syntax:

```
outp[16-bit Address] = Any Reg
```

Example:

```
outp[0x0870] = b3
```

Flags Affected:

L	limit
T1, T0	Shift bits

If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.9 Any Reg = xmem[Index Register]

Data transfer from X memory to any register. Indexed addressing is used.

Assembler Syntax:

```
Any Reg = xmem[Index Register]
```

Example:

```
a0 = xmem[i7]
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.10 xmem[Index Register] = Any Reg

Data transfer from any register to X memory. Indexed addressing is used.

Assembler Syntax:

```
xmem[Index Register] = Any Reg
```

Example:

```
xmem[i9] = b3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.11 Any Reg = ymem[Index Register]

Data transfer from Y memory to any register. Indexed addressing is used.

Assembler Syntax:

```
Any Reg = ymem[Index Register]
```

Example:

```
x0 = ymem[i0]
```

Flags Affected:

None.

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.12 ymem[Index Register] = Any Reg

Data transfer from any register to Y memory. Indexed addressing is used.

Assembler Syntax:

```
ymem[Index Register] = Any Reg
```

Example:

```
ymem[i7] = i3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.13 Any Reg = pmem[Index Register]

Data transfer from program memory to any register. Indexed addressing is used.

Assembler Syntax:

```
Any Reg = pmem[Index Register]
```

Example:

```
x0 = pmem[i0]
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.14 pmem[Index Register] = Any Reg

Data transfer from any register to program memory. Indexed addressing is used.

Assembler Syntax:

```
pmem[Index Register] = Any Reg
```

Example:

```
pmem[i7] = i3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.15 outp[Index Register] = Any Reg

Data transfer from peripheral space to any register. Indexed addressing is used.

Assembler Syntax:

```
outp[Index Register] = Any Reg
```

Example:

```
outp[i7] = i3
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Register Update:

```
no update
+1
-1
+n
+2
-2
-n
```

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.5.16 Any Reg = inp[Index Register]

Data transfer from peripheral space to any register. Indexed addressing is used.

Assembler Syntax:

```
Any Reg = inp[Index Register]
```

Example:

```
x0 = inp[i0]
```

Flags Affected:

None.

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Register Update

```
no update
+1
-1
+n
+2
-2
-n
```

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```


Copyright 2013 Cirrus Logic, Inc

3.6 Immediate Register Moves

There are five types of immediate register loads designed to cover the useful cases of moving 16-bit immediate data into an 8-bit (guard,) 16-bit (index or nm), 32-bit (data) or 72-bit (accumulator) register. The type of move is designated by a prefix on the destination register.

Table 3-2 describes how the five modes work with 72-bit accumulators and Table 3-3 describes the 32-bit registers:

Table 3-2. 72-bit Accumulators

	a0					
	a0g	a0h		a0l		
Instruction	**71 64**	**63 48**	**47 32**	**31 16**	**15 0**	
fixed16(a0)	sign extend	16-bit data	zero	zero	zero	
fixed16(a0h)	no change	16-bit data	zero	no change	no change	
fixed16(a0l)	no change	no change	no change	16-bit data	zero	
ufixed16(a0)	zero	16-bit data	zero	zero	zero	
ufixed16(a0h)	no change	16-bit data	zero	no change	no change	
ufixed16(a0l)	no change	no change	no change	16-bit data	zero	
halfword(a0)	sign extend	sign extend	16-bit data	zero	zero	
halfword(a0h)	no change	sign extend	16-bit data	no change	no change	
halfword(a0l)	no change	no change	no change	sign extend	16-bit data	
uhalfword(a0)	zero	zero	16-bit data	zero	zero	
uhalfword(a0h)	no change	zero	16-bit data	no change	no change	
uhalfword(a0l)	no change	no change	no change	zero	16-bit data	
lo16(a0)	no change	no change	16-bit data	no change	no change	
lo16(a0h)	no change	no change	16-bit data	no change	no change	
lo16(a0l)	no change	no change	no change	no change	16-bit data	

Table 3-2 Legend

- zero - all bits zero
- sign extend - sign extended from 16-bit immediate value
- no change - no bits affected
- 16-bit data - bits set to 16-bit immediate value.

Table 3-3. 32-bit Data Registers

	x0	
Instruction	**31 16**	**15 0**
fixed16(x0)	16-bit data	zero
ufixed16(x0)	16-bit data	zero
halfword(x0)	sign extend	16-bit data
uhalfword(x0)	zero	16-bit data
lo16(x0)	no change	16-bit data

The 'fixed16' move is the default prefix for accumulators and 32-bit registers. If no prefix is specified for these registers, 'fixed16' is used.

For 8-bit guard registers, 16-bit index registers, and 16-bit nm registers, no prefix should be specified. If the destination register is a guard register, the least significant 8 bits of the 16-bit immediate data value are loaded.

Copyright 2013 Cirrus Logic, Inc.

3.6.1 fixed16(Destination) = (16-bit Data)

Load 16-bit data into a register as a fixed point fractional value. The "fixed16" prefix is optional. If the destination is a 32-bit data register, data is loaded into the most significant 16 bits, and the least significant 16 bits are cleared. If the destination is an accumulator, the data is placed in the most significant 16 bits and the least significant 16 bits are cleared of the high segment (i.e. a0h), the low segment (i.e. a0l) is cleared, and the data is sign extended into the guard segment (i.e. a0g.)

Assembler Syntax:

```
fixed16(Destination) = (16-bit Data)
Destination = (16-bit Data)
```

Example:

```
fixed16(x0) = (0x1234)
x0 = (0x1234)
```

Flags Affected:

None

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
MS Reg - See Table 2-25.
```

3.6.2 ufixed16(Destination) = (16-bit Data)

Load 16-bit data into a register as an unsigned fixed point fractional value. If the destination is a 32-bit data register, data is loaded into the most significant 16 bits, and the least significant 16 bits are cleared. If the destination is an accumulator, the data is placed in the most significant 16 bits and the least significant 16 bits are cleared of the high segment (that is, a0h), the low segment (that is, a0l) and the guard segment (that is, a0g.) are cleared.

Assembler Syntax:

```
ufixed16(Destination) = (16-bit Data)
```

Example:

```
ufixed16(x0) = (0x1234)
```


Copyright 2013 Cirrus Logic, Inc

Flags Affected:

None

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
MS Reg - See Table 2-25.
```

3.6.3 uhalfword(Destination) = (16-bit Data)

Load 16-bit data into a register as an unsigned integer. If the destination is a 32-bit data register, data is loaded into the least significant 16 bits, and the most significant 16 bits are cleared. If the destination is an Accumulator, the data is placed in the least significant 16 bits and the most significant 16 bits are cleared of the high segment (i.e. a0h), the low segment (i.e. a0l) is cleared, and the data is sign extended into the guard segment (i.e. a0g.)

Assembler Syntax:

```
uhalfword(Destination) = (16-bit Data)
```

Example:

```
uhalfword(a0) = (0x1234)
```

Flags Affected:

None.

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
MS Reg - See Table 2-25.
```


Copyright 2013 Cirrus Logic, Inc.

3.6.4 Index Register = (16-bit Data)

Load 16-bit data into an index register as an unsigned integer. Data is loaded into the least significant 16 bits, and the most significant 16 bits are cleared.

Assembler Syntax:

```
Index Register = (16-bit Data)
```

Example:

```
i3 = (0xface)
```

Flags Affected:

None.

Restrictions:

Destination:

```
i0 - i11
```

3.6.5 NM Register = (16-bit Data)

Load 16-bit data into a register as an unsigned integer.

Assembler Syntax:

```
NM Register = (16-bit Data)
```

Example:

```
nm8 = (0xface)
```

Flags Affected:

None.

Restrictions:

Destination:

```
nm0 - nm11
```

3.6.6 Guard Register = (8-bit Data)

Load 16-bit data into a register as an unsigned integer. If the destination is a 32-bit data register, data is loaded into the least significant 16 bits, and the most significant 16 bits are cleared. If the destination is an Accumulator, the data is placed in the least significant 16 bits and the most significant 16 bits are cleared of the high segment (i.e. a0h), the low segment (i.e. a0l) is cleared, and the data is sign extended into the guard segment (i.e. a0g.) Since this is the only instruction for loading the Guard, Index, and NM registers directly, the prefix is optional for those destinations.

Copyright 2013 Cirrus Logic, Inc

Assembler Syntax:

```
Guard Register = (8-bit Data)
```

Example:

```
b2g = (0xfe)
```

Flags Affected:

None

.Restrictions:

Destination:

```
a0g - a3g
b0g - b3g
```

3.6.7 halfword(Destination) = (16-bit Data)

Load 16-bit data into a register as a signed integer. If the destination is a 32-bit data register, data is loaded into the least significant 16 bits, and the and the data is sign extended into the most significant 16 bits. If the destination is an accumulator, the data is placed in the least significant 16 bits and sign extended into the most significant 16 bits of the high segment (i.e. a0h), the low segment (i.e. a0l) is cleared, and the data is sign extended into the guard segment (i.e. a0g.)

Assembler Syntax:

```
halfword(Destination) = (16-bit Data)
```

Example:

```
halfword(a0) = (0x1234)
halfword(dbc_d1) = (0xffff)
```

Flags Affected:

None.

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
MS Reg - See Table 2-25.
```


Copyright 2013 Cirrus Logic, Inc.

3.6.8 lo16(Destination) = (16-bit Data)

Load 16-bit data into the least significant 16 bits of a register. No other bits in the register are affected. If a full accumulator (i.e. a0) is specified, the operation is performed on just the high part of the accumulator (i.e. a0h).

Assembler Syntax:

```
lo16(Destination) = (16-bit Data)
```

Example:

```
lo16(x0) = (0x1234)
lo16(dbc_d1) = (0xabcd)
```

Flags Affected:

None

Restrictions:

Destination

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
MS Reg - See Table 2-25.
```

3.6.9 MS Reg = (16-bit Data)

Load 16-bit data into a MS register as a signed integer. If the destination is a 32-bit MS register, data is loaded into the least significant 16 bits, and the and the data is sign extended into the most significant 16 bits.

Assembler Syntax:

```
MS Reg = (16-bit Data)
```

Example:

```
dbc_d1 = (0x1234)
jsr_data = (0xabcd)
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination

MS Reg - See Table 2-25.

3.6.10 AnyReg(Any Reg, Any Reg)

Transfer data from any register to any register.

Restrictions:

None.

Assembler Syntax:

```
AnyReg(Destination, Source)
```

Example:

```
AnyReg(nm4, b0h)
```

Flags Affected:

L	limit
T1, T0	Shift bits

If the source is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Destination/Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
```

MS Reg - See Table 2-25.

3.6.11 Any Reg = MS Reg

Transfer data from a Master State Register to any register.

Assembler Syntax:

```
Any Reg = MS Reg
```

Example:

```
b0l = jsr_mode
a0l = search_latch
```


Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination/Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25
```

3.6.12 MS Reg = Any Reg

Transfer data from any register to a Master State Register.

Assembler Syntax:

```
MS Reg = Any Reg
```

Example:

```
jsr_mode = b0l
search_latch = a0l
```

Flags Affected:

L Only flags affected are:

L	limit
T1, T0	Shift bits

Note: If Any Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b

Copyright 2013 Cirrus Logic, Inc.

Restrictions:

Destination/Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.6.13 AnyReg (Any Reg, Any Reg), (Any Reg, Any Reg)

Performs dual data transfers from any register to any register. There are some limitations in source and destination due to the current implementation of the Cirrus Logic DSP 32-bit architecture.

Restrictions:

The restrictions for AnyReg transfers include the following:

- The first pair of registers is processed first, and the second pair of registers is processed second.
- Both sources cannot be Index registers
- Both sources cannot be NM registers
- If both destinations are Index registers, destination 1 must be i0-i3 or i8-i11, and destination 2 must be i4-i7.
- If both destinations are NM registers, destination 1 must be nm0-nm3 or nm8-nm11, and destination 2 must be nm4-nm7.
- Dual accumulator destination indices must be equal.
- "B" accumulator must be in second pair of arguments
- "A" accumulator must be in first pair of arguments

Assembler Syntax:

```
AnyReg(Destination1, Source1),(Destination2, Source2)
```


Copyright 2013 Cirrus Logic, Inc

Example:

```
AnyReg(i0,nm5),(nm5,b0)
AnyReg(i0,nm4),(i7,x0)
AnyReg(x0,nm4),(i2,i8)
AnyReg(i0,i11),(i4,b0h)
```

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Source 1 or Source 2 is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Restrictions:

Destination1/Source1,

Destination2/Source2:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - a3h
b0h - b3h
a0g - a3g
b0g - b3g
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
```

3.6.14 Accum = long(Accum)

Transfers 64 bits of the source through the SRS unit to the destination. This instruction differs from the move instruction "Accum = Accum" in that it transfers 64 bits instead of 32, and it differs from the math instruction "Accum =+ Accum" in that it transfers 64 bits instead of 72, and it goes through the SRS unit, performing any necessary shifting or saturating.

Assembler Syntax:

```
Accum = long(Accum)
```

Example:

```
a0 = long(b2)
```


Copyright 2013 Cirrus Logic, Inc.

Flags Affected:

L	limit
T1, T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.

Note: The Source and Destination are both encoded twice in this instruction.

Restrictions:

Destination:

```
Source
a0 - a3
b0 - b3
```

3.6.15 In = Im/(0) ± (16-bit Data)

Add 16-bit immediate data to an optional source index register and place result in destination index register.

If an optional source Index register is used, addition is governed by the current state of the NM register associated with the source Index register (Im).

Important Note: When the In = (0) ± (16-bit Data) form of this instruction is used, addition mode is governed by the current state of the NM0 register, regardless of the value of n.

This instruction uses the AGU and hence does not require a subsequent dead-cycle before the index register is used. For example, the following code is valid:

```
i0 = (0) + (0x1234)
x0 = xmem[i0]
```

Assembler Syntax:

```
In = Im + (16-bit Data)
In = Im - (16-bit Data)
In = (0) + (16-bit Data)
In = (0) - (16-bit Data)
```

Example:

```
i0 = i4 + (0x1234)
i4 = i4 - (0x1234)
```

Flags Affected:

None

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination:

```
i0
i1
i2
i3
i4
i5
i6
i7
i8
i9
i10
i11
```

Source:

```
i0
i1
i2
i3
i4
i5
i6
i7
i8
i9
i10
i11
```

Instruction:

```
In = Im + (16-bit)
In = Im - (16-bit)
In = (0) + (16-bit)
In = (0) - (16-bit)
```

3.7 Bit Manipulation Instructions

3.7.1 Bit Test

Test the bits of the register specified by the immediate mask value. If all bits of the masked 16-bit result are ones, then the z bit in the CCR is set to one. Otherwise, the z bit is set to zero. The pseudo-code for this operation is:

```
if ((reg AND mask) XOR mask) == 0x0000
z = 1
else
z = 0
```

Either the least significant or most significant 16 bits of the register can be used, as selected by the "lo" or "hi" prefix. The register is unaffected after execution of this instruction. Not allowed on accumulators.

 Copyright 2013 Cirrus Logic, Inc.

Assembler Syntax:

```
BitTst lo(32-bit Reg),(16-bit Mask)
BitTst hi(32-bit Reg),(16-bit Mask)
BitTst (16-bit Reg),(16-bit Mask)
```

Example:

```
BitTst lo(x0),0x0400
BitTst hi(imask),0x0002
BitTst (i4),0xaaaa
```

Flags Affected:

Z	zero

Restrictions:

Destination:

```
x0 - x3
y0 - y3
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.7.2 Bit Set

Perform a bitwise test as in BitTst, then perform a bitwise OR on 16 bits of the specified register with the immediate mask value and place the result back into the register. Either the least significant or most significant 16 bits of the register can be used, as selected by the "lo" or "hi" prefix. Not allowed on accumulators.

Assembler Syntax:

```
BitSet lo(32-bit Reg),(16-bit Mask)
BitSet hi(32-bit Reg),(16-bit Mask)
BitSet (16-bit Reg),(16-bit Mask)
```

Example:

```
BitSet lo(y2),0x0402
BitSet hi(x3),0x0001
BitSet (i1),0x0002
```

Flags Affected:

Z	zero

Copyright 2013 Cirrus Logic, Inc

Restrictions:

Destination:

```
x0 - x3
y0 - y3
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```

3.7.3 Bit Clear

Perform a bitwise test as in BitTst, then perform a bitwise AND on 16 bits of the specified register with the bitwise NOT of the immediate mask value and place the result back into the register. Either the least significant or most significant 16 bits of the register can be used, as selected by the "lo" or "hi" prefix. Not allowed on accumulators.

Assembler Syntax:

```
BitClr lo(32-bit Reg),(16-bit Mask)
BitClr hi(32-bit Reg),(16-bit Mask)
BitClr (16-bit Reg),(16-bit Mask)
```

Example:

```
BitClr lo(x2),0x0100
BitClr hi(y2),0x8002
BitClr (nm7),0x0400
```

Flags Affected:

Z	zero

Restrictions:

Destination

```
x0 - x3
y0 - y3
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```


Copyright 2013 Cirrus Logic, Inc.

3.7.4 Bit Change

Perform a bitwise test as in BitTst, then perform a bitwise XOR on 16 bits of the specified register with the immediate mask value and place the result back into the register. Either the least significant or most significant 16 bits of the register can be used, as selected by the "lo" or "hi" prefix. Not allowed on accumulators.

Assembler Syntax:

```
BitChg lo(32-bit Reg),(16-bit Mask)
BitChg hi(32-bit Reg),(16-bit Mask)
BitChg (16-bit Reg),(16-bit Mask)
```

Example:

```
BitChg lo(x0),0xffff
BitChg hi(y3),0x0180
BitChg (i0),0x5000
```

Flags Affected:

Z	zero

Restrictions:

Destination:

```
x0 - x3
y0 - y3
i8 - i11
nm8 - nm11
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
MS Reg - See Table 2-25.
```


Copyright 2013 Cirrus Logic, Inc

Chapter 4
Multifunction Moves

4

Multifunction instructions occupy the most significant 16 bits of the instruction word. In general, they affect the shifting/limiting bits of the CCR.

4.1 Single Multifunction Moves

Transfers data between a data path register and X or Y memory. Either indexed or direct addressing (6 bit) can be used. Index registers can be updated. Data moves that can be done by themselves or included with a corresponding arithmetic instruction. Performing two data moves is considered a parallel move. There are restrictions for parallel moves, but any multifunction move can be done in conjunction with an arithmetic instruction.

4.1.1 DP Reg = xmem[Index Register]
DP Reg = xmem[6-bit Address]

Assembler Syntax:

```
DP Reg = xmem[Index Register]; Index Register; ±= update
DP Reg = xmem[6-bit Address]
```

Example:

```
x2 = xmem[i7]
b0 = xmem[0x38]
```

Flags Affected:

None

Copyright 2013 Cirrus Logic

Restrictions:

Destination:

```
x0
x1
x2
x3
y0
y1
y2
y3
a0
a1
a2
a3
b0
b1
b2
b3
```

4.1.2 xmem[Index Register] = DP Reg
xmem[6-bit address] = DP Reg

Assembler Syntax:

```
xmem[Index Register] = DP Reg; ±= update
xmem[6-bit Address] = DP Reg
```

Example:

```
xmem[i7] = x2
xmem[0x2f] = a2
```

Flags Affected:

L	limit
T1,T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b

Copyright 2013 Cirrus Logic

Restrictions:

Source:

```
x0
x1
x2
x3
y0
y1
y2
y3
a0
a1
a2
a3
b0
b1
b2
b3
```

4.1.3 DP Reg = ymem[Index Register]
DP Reg = ymem[6-bit address]

Assembler Syntax:

```
DP Reg = ymem[Index Register]; ±= update
DP Reg = ymem[6-bit Address]
```

Example:

```
b2 = ymem[i7]
y3 = ymem[0x1f]
```

Flags Affected:

None.

Copyright 2013 Cirrus Logic

Restrictions:

Destination:

```
x0
x1
x2
x3
y0
y1
y2
y3
a0
a1
a2
a3
b0
b1
b2
b3
```

4.1.4 ymem[Index Register] = DP Reg
ymem[6-bit address] = DP Reg

Assembler Syntax:

```
ymem[Index Register] = DP Reg; ±= update
ymem[6-bit Address] = DP Reg
```

Example:

```
ymem[i7] = b3
ymem[0x30] = i3
```

Flags Affected:

L	limit
T1,T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic

Restrictions:

Source:

```
x0
x1
x2
x3
y0
y1
y2
y3
a0
a1
a2
a3
b0
b1
b2
b3
```

4.1.5 Data Path Register to or from Any Register

4.1.5.1 DP Reg = Any Reg

Data transfer between any register and a data path register.

Assembler Syntax:

```
DP Reg = Any Reg
```

Example:

```
y0 = b3g
```

Flags Affected:

None.

Copyright 2013 Cirrus Logic

Restrictions:

Destination:

```
x0
x1
x2
x3
y0
y1
y2
y3
a0
a1
a2
a3
b0
b1
b2
b3
```

Source:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
```

4.1.5.2 Any Reg = DP Reg

There are two different instructions that can be used to load the high portion of an accumulator with the contents of a data path register.

The first takes the form:

```
b3 = x2
```

The second takes the form:

```
b3 = +x2
```

In terms of effective functionality, the results of executing either of these two instructions are identical—the contents of the destination accumulator is always the same, regardless of which instruction is executed. Neither is affected by the SRS block, so no shifting, rounding or

Copyright 2013 Cirrus Logic

saturation will take place. Nor is either subject to the built-in one bit left shift that normally occurs with MAC multiply instructions (such as with `b3 = x2 * y2`). Both sign-extend into the accumulator's guard register during the transfer.

The only difference is in the type of instruction that they are. In the form `b3 = x2,` you are executing a 16-bit Multifunction Move operation. This is the instruction described in this section.

In the form `b3 = +x2`, you are executing a 16-bit Multifunction Arithmetic MAC operation. Specifically, this is the "Multiply by One with Optional Accumulate" instruction as seen in Section 5.1.6 and Section 5.1.7.

The reason it is important to note that these are different instructions is because, as with any pair of one Multifunction Move instruction and one Multifunction Arithmetic/Accumulator instruction, the two can be packed into a single 32-bit instruction.

For example, if you would like to load four accumulators with four different values using a single instruction, you may do so with the following code:

```
a0 = +x2; b0 = +y2; a3 = x3; b3 = y3
```

A parallel multiply by one instruction as described in Section 5.1.7 allows two destination registers per assignment, so this instruction could be extended to assign six registers in the form:

```
a1 = a0 = +x2; b1 = b0 = +y2; a3 = x3; b3 = y3
```

where a1 and a0 are both assigned the value in `x2` and `b1` and `b0` are both assigned the value in `y2`.

In general, when choosing between which form of the move instruction you will use in your code, you should consider what else you would like to accomplish in addition to the move with that cycle of CPU execution. If you want to perform another move, as shown in the above example, or perhaps a move from memory into an accumulator, you must use the form of move that utilizes the MAC:

```
b3 = +x2; a0 = xmem[i0]
```

If you attempted the following code:

```
b3 = x2; a0 = xmem[i0]
```

you would receive the following assembler error:

```
"Instructions cannot fit into one word."
```

On the other hand, if you wanted to perform some other arithmetic operation with this instruction, such as a bitwise OR (see Section 5.2.15), then you must use the form of move that only operates on the data path:

```
b3 = x2; a0 = a0 | b1
```

Again, if you attempted the following code:

```
b3 = +x2; a0 = a0 | b1
```

you would receive the following assembler error:

```
"Instructions cannot fit into one word."
```


Copyright 2013 Cirrus Logic

Finally, one should be careful not to confuse the "=+" notation with the "+=" operator. The instruction `b3 =+ x2` and `b3 += x2` accomplish two very different things.The latter is the accumulation operator that can be translated into:

```
b3 = b3 + x2
```

where the contents of `x2` will be added to the high portion of `b3` of stored back into `b3`.

Assembler Syntax:

```
Any Reg = DP Reg
```

Example:

```
b3 = x2
nm0 = a2
```

Flags Affected:

L	limit
T1,T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Copyright 2013 Cirrus Logic

Restrictions:

Destination:

```
x0 - x3
y0 - y3
a0 - a3
b0 - b3
a0l - a3l
b0l - b3l
a0h - b3h
b0h - b3h
a0g - a3g
b0g - b3g
i0 - i3
nm0 - nm3
i4 - i7
nm4 - nm7
```

Source:

```
x0
x1
x2
x3
y0
y1
y2
y3
a0
a1
a2
a3
b0
b1
b2
b3
```


Copyright 2013 Cirrus Logic

4.2 Parallel Multifunction Move Instructions

Parallel multifunction moves allow one X memory move and one Y memory move in a single instruction. Parallel multifunction moves are a subset of multifunction data moves.

Restrictions:

Perform data transfer from memory to a data register, or from an accumulator to memory.

- X memory can only be addressed using index registers i0 and i1
- Y memory can only be addressed using index registers i4 and i5.
- X memory moves can only be used with X data registers and A accumulators.
- Y memory moves can only be used with Y data registers and B accumulators.
- Accumulators (a0-a3, b0-b3) can only be a source
- Data registers (x0-x3, y0-y3) can only be a destination

Note: "Parallel Pairing: Right" and "Parallel Pairing Left," which appear below Assembler Syntax statements in this Section, refers to whether the defined register in a set of paired registers is to the left or right of a comma. A register that is "Parallel Pairing: Left" must be to the left of the comma, while "Parallel Pairing: Right" must be to the right of a comma when reading an instruction from left to right.

4.2.1 Xn = xmem[Index Register]

Assembler Syntax:

```
Xn = xmem[Index Register];±= update
```

Parallel Pairing: Left (see **Restrictions in Section 4.2**)

Example:

```
x0 = xmem[i0]; i0+=n; y0 = ymem[i4]; i4+=n;
```

Flags Affected:

None

Copyright 2013 Cirrus Logic

Restrictions:

Register Update:

```
no update
+1
-1
+n
```

Destination

```
x0
x1
x2
x3
```

4.2.2 xmem[Index Register] = An

Assembler Syntax:

```
xmem[Index Register] = An; ±= update
```

Parallel Pairing: Left (see **Restrictions in Section 4.2**)

Example:

```
xmem[i1] = a3; i1+=n; y3 = ymem[i4]
```

Flags Affected:

L	limit
T1,T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected.After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.

Restrictions:

Register Update:

```
no update
+=1
-=1
+=n
```

Destination:

```
a0
a1
a2
a3
```


Copyright 2013 Cirrus Logic

4

4.2.3 Ym = ymem[Index Register]

Assembler Syntax:

```
Ym = ymem[Index Register]; ±= update
```

Parallel Pairing: Right (see **Restrictions in Section 4.2**)

Example:

```
xmem[i1] = a3; i1+=n; y3 = ymem[i4]
```

Flags Affected:

None

Restrictions:

Register Update:

```
no update
+=1
-=1
+=n
```

Destination:

```
y0
y1
y2
y3
```

4.2.4 ymem[Index Register] = Bm

Assembler Syntax:

```
ymem[Index Register] = Bm; ±= update
```

Parallel Pairing: Right (see **Restrictions in Section 4.2**)

Example:

Examples of parallel multifunction moves:

X memory moves are placed before Y memory moves in the syntax:

```
x0 = xmem[i0]; i0+=n;  ymem[i4]=b0
```


Copyright 2013 Cirrus Logic

Other Examples:

```
;ymem[i4] = b0
```

When used in conjunction with an arithmetic (least significant 16 bits) instruction:

```
a0=x2*y0;   x0=xmem[i0];   i0+=n;      ymem[i4]=b0
            xmem[i1]=a0;               ymem[i4]=b3;    i4+=1
            x0=xmem[i0];   i0+=1;      y0=ymem[i5];    i5+=n
            x3=xmem[i0];   i0+=1;      ymem[i5]=b2;    i5-=1
```

Flags Affected:

L	limit
T1,T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.

Restrictions:

Register Update:

```
no update
+1
-1
+n
```

Source

```
b0
b1
b2
b3
```

4.3 Data Path Register to Data Path Register Instructions

Perform data transfer from a data register to a data register.

Restrictions:

- Accumulators (a0-a3, b0-b3) can only be a source
- Data registers (x0-x3, y0-y3) can only be a destination

Copyright 2013 Cirrus Logic

4.3.1 DP Reg = DP Reg

Data path register to data path register data move. The only restriction is that the source and destination must have the same index (0-3).

Assembler Syntax:

```
DP Reg = DP Reg
```

Example:

```
y0 = b3g
```

Flags Affected:

None

Restrictions:

Destination

x
y
a
b

Source

x
y
a
b

Note: See other **Restrictions in Section 4.3.**

4.4 Parallel Register to/from Register Instructions

Perform data transfer from a data register to a data register, or from an accumulator to a data register.

Restrictions:

- Accumulators (a0-a3, b0-b3) can only be a source
- Data registers (x0-x3, y0-y3) can only be a destination

Examples:

```
x0=y0; b3=a3
a2=b2; y0=a0
```


Copyright 2013 Cirrus Logic

4.4.1 Data Path Register to Data Path Register and Data Path Register to/from X or Y Memory Restrictions

Perform a parallel multifunction move using a data register move and a memory move. One instruction from each of the previous two groups can be combined into one parallel move.

- For this combination move, the sources cannot both be A or B accumulators. For example:

```
ymem[i4]=b2; a0=b0      #Bad:Sources are b2 and b0 (both B accumulators)
ymem[i4]=b2; b2=a2      #Good:Sources are b2 and a2
```

The exception to this is when the source accumulators are from the same accumulator group.

For example, this is illegal:

```
x0=a0; xmem[i0]=a1
```

The following instruction is also illegal:

```
x0=a0; xmem[i0]=a0
```

But it is legal, when rewritten as:

```
x0=xmem[i0]=a0
```

The restrictions for this case are:

- An accumulator must be the source.
- The memory space and accumulator must "match."

For example, this is illegal:

```
x0=ymem[i4]=a1
```

As it uses Y memory and an A accumulator. Switching to a B accumulator makes the instruction legal, as follows:

```
x0=ymem[i4]=b1
```

- There is no restriction on the data register (x0-x3 or y0-y3) used.
- If both destinations are data registers (rather than one data register and one memory location), one must be an X data register and the other a Y data register.
- If the accumulator source is an A accumulator, the accumulator index is encoded in the most significant 8 bits of the opcode, and the accumulator index in the least significant 8 bits is ignored. Conversely, if the accumulator source is a B accumulator, the accumulator index is encoded in the least significant 8 bits of the opcode, and the accumulator index in the most significant 8 bits is ignored.

Copyright 2013 Cirrus Logic

4.5 64-bit Multifunction Moves

A 64-bit multifunction move will move X and Y memory into a pair of registers.

Restrictions:

- The data register pair must share the same index.

4.5.1 Data Path Register Pair to or from XY Memory

4.5.1.1 Data Path Register Pair = xymem[Index Register]
Data Path Register Pair = xymem[6-bit Address]

Long data transfer between XY or AB registers and XY memory. Either indexed or direct addressing (6 bit) can be used. Index registers can be updated.

Assembler Syntax:

```
Xn,Yn = xymem[Index Register]; ±= update
Xn,Yn = xymem[6-bit Address]
An,Bn = xymem[Index Register]
An,Bn = xymem[6-bit Address]
```

Example:

```
x0,y0 = xymem[i0]
x1,y1 = xymem[0x23]
a0,b0 = xymem[i7]
a3,b3 = xymem[0x03]
```

Flags Affected:

None

Restrictions:

Destination:

```
x0,y0
x1,y1
x2,y2
x3,y3
a0,b0
a1,b1
a2,b2
a3,b3
```


Copyright 2013 Cirrus Logic

4.5.1.2 xymem[Index Register] = Data Path Register Pair xymem[6-bit Address] = Data Path Register Pair

Assembler Syntax:

```
xymem[Index Register] = Xn,Yn; ±= update
xymem[6-bit Address] = Xn,Yn
xymem[Index Register] = An,Bn
xymem[6-bit Address] = An,Bn
```

Example:

```
xymem[i0] = x0,y0
xymem[0x23] = x2,y2
xymem[i7] = a3,b3
xymem[0x03] = a2,b2
```

Flags Affected:

L	limit
T1,T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 will only have values of 10b, 01b, or 00b.

Restrictions:

Source:

```
x0,y0
x1,y1
x2,y2
x3,y3
a0,b0
a1,b1
a2,b2
a3,b3
```


Copyright 2013 Cirrus Logic

4.5.2 Accumulator to or from XY Memory

4.5.2.1 Accum = xymem[Index Register]
Accum = xymem[6-bit Address]

Long data transfer between XY memory and an accumulator. Either indexed or direct addressing (6 bit) can be used. Index registers can be updated.

Assembler Syntax:

```
Accum = xymem[Index Register]; ±= update
Accum = xymem[6-bit Address]
```

Example:

```
b3 = xymem[i7]
a0 = xymem[0x30]
```

Flags Affected:

None

Restrictions:

Destination

```
a0
a1
a2
a3
b0
b1
b2
b3
```

4.5.2.2 xymem[Index Register] = Accum
xymem[6-bit Address] = Accum

Assembler Syntax:

```
xymem[Index Register] = Accum; ±= update
xymem[6-bit Address] = Accum
```

Example:

```
xymem[i7] = b0
xymem[0x24] = a3
```


Copyright 2013 Cirrus Logic

Flags Affected:

L	limit
T1,T0	Shift bits

Note: If Reg is an accumulator, then the L, T1, and T0 are affected. After these flags are set, they must be cleared manually by the user. T0 and T1 only have values of 10b, 01b, or 00b.

Restrictions:

Source:

```
a0
a1
a2
a3
b0
b1
b2
b3
```

4.6 Index Register Updates

4.6.1 In = Im ± (6-bit Data)

Add 6-bit immediate data to source index register and place result in destination index register. Source register (Im) is limited to i8-i11.

Addition is governed by the current state of the NM register associated with the source Index register (Im).

This instruction uses the AGU and hence does not require a subsequent dead-cycle before the index register is used. For example, the following code is valid:

```
i0 = i8 + (0x12)
x0 = xmem[i0]
```

Assembler Syntax:

```
In = Im + (6-bit Data)
In = Im - (6-bit Data)
```

Examples:

```
i0 = i9 + (0x12)
i4 = i8 - (0x34)
```

Flags Affected:

None

Copyright 2013 Cirrus Logic

Restrictions:

Destination:

```
i0
i1
i2
i3
i4
i5
i6
i7
i8
i9
i10
i11
```

Source:

```
i8
i9
i10
i11
```

4.6.2 In ±= 1/2/N

Normal index register update without associated move. Operation occurs in the decode state.

Assembler Syntax:

```
In ±= 1/2/n
```

Examples:

```
i0 += 1
i4 -= n
```

Flags Affected:

None

Copyright 2013 Cirrus Logic

Restrictions:

Destination:

```
i0
i1
i2
i3
i4
i5
i6
i7
i8
i9
i10
i11
```


Copyright 2013 Cirrus Logic

Chapter 5
Multifunction Operations

5

5.1 Multifunction Arithmetic Instructions

Single or parallel arithmetic instructions can be done by themselves or with multifunction moves.

5.1.1 Parallel Multiply/Multiply-Accumulate I

Parallel multiply or multiply accumulate, result in one or two accumulators.

Assembler Syntax:

```
Aq=Ar ±= Xn*Xm;Bq=Br ±= Yn*Xm
Aq=Ar ±= Xn*Xm;Bq=Br ±= -Yn*Xm
Aq=Ar ±= Xn*Ym;Bq=Br ±= Yn*Ym
Aq=Ar ±= Xn*Ym;Bq=Br ±= -Yn*Ym
Aq=Ar ±= Yn*Xm;Bq=Br ±= Xn*Xm
Aq=Ar ±= Yn*Xm;Bq=Br ±= -Xn*Xm
Aq=Ar ±= Yn*Ym;Bq=Br ±= Xn*Ym
Aq=Ar ±= Yn*Ym;Bq=Br ±= -Xn*Ym
Aq=Ar ±= -Xn*Xm;Bq=Br ±= Yn*Xm
Aq=Ar ±= -Xn*Xm;Bq=Br ±= -Yn*Xm
Aq=Ar ±= -Xn*Ym;Bq=Br ±= Yn*Ym
Aq=Ar ±= -Xn*Ym;Bq=Br ±= -Yn*Ym
Aq=Ar ±= -Yn*Xm;Bq=Br ±= Xn*Xm
Aq=Ar ±= -Yn*Xm;Bq=Br ±= -Xn*Xm
Aq=Ar ±= -Yn*Ym;Bq=Br ±= Xn*Ym
Aq=Ar ±= -Yn*Ym;Bq=Br ±= -Xn*Ym
```

Example:

```
a0=x2*x3;b0=y2*x3
a0=x2*x3;b0=-y2*x3
a1=a0+=x0*y2;b1=b0+=y0*y2
a1=a0+=x0*y2;b1=b0-=y0*y2
a2=a3=-y1*x1;b2=b3=-x1*x1
a2=a3=-y1*x1;b2=b3=x1*x1
a1-=y3*y0; b1-=x3*y0
a1-=y3*y0; b1+=x3*y0
a0=-x2*x3;b0=y2*x3
a0=-x2*x3;b0=-y2*x3
a1=a0-=x0*y2;b1=b0+=y0*y2
```


Copyright 2013 Cirrus Logic

5

```
a1=a0-=x0*y2;b1=b0-=y0*y2
a2=a3=-y1*x1;b2=b3=-x1*x1
a2=a3=-y1*x1;b2=b3=x1*x1
a1+=y3*y0; b1-=x3*y0
a1+=y3*y0; b1+=x3*y0
```

Flags Affected:

None.

5.1.2 Parallel Multiply/Multiply-Accumulate II

Parallel Multiply/Multiply-Accumulate II (Double FIR) allows one register (X or Y) times two different registers. The arithmetic operators preceding the source register must be the same for both instructions, which together make up this parallel instruction.

Example:

```
Aq=Ap += Xn*Yn; Bq=Bp += Xn*Xm
Aq=Ap -= Xn*Yn; Bq=Bp -= Xn*Xm
```

Assembler Syntax:

```
Aq=Ap ±= Xn*Yn;Bq=Bp ±= Xn*Xm
Aq=Ap ±= Xn*Yn;Bq=Bp ±= Xn*Ym
Aq=Ap ±= Xn*Xm;Bq=Bp ±= Xn*Yn
Aq=Ap ±= Xn*Ym;Bq=Bp ±= Xn*Yn
Aq=Ap ±= Yn*Xn;Bq=Bp ±= Yn*Xm
Aq=Ap ±= Yn*Xn;Bq=Bp ±= Yn*Ym
Aq=Ap ±= Yn*Xm;Bq=Bp ±= Yn*Xn
Aq=Ap ±= Yn*Ym;Bq=Bp ±= Yn*Xn
Aq=Ap = -XY;Bq=Bp = -XX
Aq=Ap = -XY;Bq=Bp = -XY
Aq=Ap = -XX;Bq=Bp = -XY
Aq=Ap = -YX;Bq=Bp = -YX
Aq=Ap = -YX;Bq=Bp = -YY
Aq=Ap = -YY;Bq=Bp = -YX
```

Example:

```
a3-= x0*y0;b3-= x0*x2
a1=a0=  x0*y0;b1=b0=  x0*y3
a1= -x1*x1;b1= -x1*y1
a2=a3+= x1*y2;b2=b3+= x1*y1
a3=a1= -y3*x3;b3=b1= -y3*x2
a0=a2-= y2*x2;b0=b2-= y2*y0
a2=  y0*x2;b2=  y0*x0
a0+= y3*y0;b0+= y3*x3
```


Copyright 2013 Cirrus Logic

Flags Affected:

None

Restrictions:

Destination:

```
a0,b0
a1,b1
a2,b2
a3,b3
a1a0,b1b0
a3a1,b3b1
a0a2,b0b2
a2a3,b2b3
```

5.1.3 Real Multiply/Multiply-Accumulate

Multiply or multiply accumulate, result in an accumulator. Special mode allows one multiplicative operator to be treated as an unsigned value, range (0 to 1.99999) instead of (-1 to.99994). Unsigned by unsigned multiples are only valid for results ≤(1.99999).

Assembler Syntax:

```
Accum ?= -Xn*Xm
Accum ?= -Xn*(unsigned)Ym
Accum ?= -Xm*Yn
Accum ?= -Yn*Xm
Accum ?= -Yn*Ym
Accum ?= -(unsigned)Xn*(unsigned)Ym
Accum ?= -Xn*(unsigned)Ym
```

Example:

```
a1 =  x0*x3
b3 = x3*(unsigned)y3
b0 += x1*y2
a1 -= y2*x1
b2 =  y0*y0
a0 = (unsigned)x0*(unsigned)y0
```

Flags Affected:

None

Copyright 2013 Cirrus Logic

5

Restrictions:

Destination:

```
a0
a1
a2
a3
b0
b1
b2
b3
```

5.1.4 Parallel Squares

Square data registers, store or accumulate, result in one or two accumulators.

Assembler Syntax:

```
Aq = Ar ±= -Xn*Xn;Bq = Br ±= -Yn*Ym
```

Example:

```
a0 =  x2*x2;b0 =  y2*y1
a3=a1+=x2*x2;b3=b1+=y2*y1
```

Flags Affected:

None.

Restrictions:

Destination:

```
a0,b0
a1,b1
a2,b2
a3,b3
a1a0,b1b0
a3a1,b3b1
a0a2,b0b2
a2a3,b2b3
```


Copyright 2013 Cirrus Logic

5.1.5 Parallel Multiply with Add

Multiply two data registers, add accumulator (A0 or B0 only), store result in one or two accumulators.

Assembler Syntax:

```
Aq = Ar = A0±Xn*Xm;Bq = Br = B0±Yn*Xm
Aq = Ar = A0±Xn*Ym;Bq = Br = B0±Yn*Ym
```

Example:

```
a1=a0-x2*x1; b1=b0-y2*x1
a2=a3=a0-x3*y0; b2=b3=b0-y3*y0
a1=a0=a0+x1*y1; b1=b0=b0+y1*y1
```

Flags Affected:

None

Restrictions:

Destination:

```
a0,b0
a1,b1
a2,b2
a3,b3
a1a0,b1b0
a3a1,b3b1
a0a2,b0b2
a2a3,b2b3
```

5.1.6 Multiply by One with Optional Accumulate

Move or accumulate X or Y register into A or B accumulator.

Note: The syntax 'b3 = +x2' is used to differentiate between this instruction and the move 'b3 = x2'. See Section 4.1.5.2 for a discussion of the differences.

Assembler Syntax:

```
Accum ±= Xn
Accum ±= Yn
Accum = ±Xn
Accum = ±Yn
```

Example:

```
b3 = +x2
a2 -= y0
```


Copyright 2013 Cirrus Logic

5

Flags Affected:

None.

Restrictions:

Destination:

```
a0
a1
a2
a3
b0
b1
b2
b3
```

5.1.7 Parallel Multiply by One with Optional Accumulate

Move or accumulate X or Y registers into A or B accumulators.

Note: The syntax 'b3 = +x2' is used to differentiate between this instruction and the move 'b3 = x2'. See Section 4.1.5.2 for a discussion of the differences.

Assembler Syntax:

```
Aq = Ap ±= ±Xn;Bq = Bp ±= ±Yn
Aq = Ap ±= ±Yn;Bq = Bp ±= ±Xn
```

Example:

```
a3 = +x0;b3 = +y0
a3 = a1 -= y2;b3 = b1 -= x2
```

Flags Affected:

None.

Restrictions:

Destination:

```
a0,b0
a1,b1
a2,b2
a3,b3
a1a0,b1b0
a3a1,b3b1
a0a2,b0b2
a2a3,b2b3
```


Copyright 2013 Cirrus Logic

5.2 Multifunction Accumulator Instructions

Least significant 16 bits of instruction. Affects the zero and negative bits in the CCR.

5.2.1 Parallel Add with Shift

Parallel add or subtract two accumulators, result in an accumulator. One of the operands can be shifted.

Assembler Syntax:

```
Ap=An ± Am;Bp=Bn ± Bm
Ap=An ± Bm;Bp=Bn ± Am
Ap=(An * 2) ± Am;Bp=(Bn * 2) ± Bm
Ap=(An * 2) ± Bm;Bp=(Bn * 2) ± Am
```

Example:

```
a1 = a2+a3;b1 = b2+b3
a3 = a0-b0;b3 = b0-a0
a3 = (a2*2)+a3;b3 = (b2*2)-b3
a1 = (a1*2)-b1;b1 = (b1*2)+a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.2 Add with Shift

Add or subtract two accumulators, result in an accumulator. One of the operands can be shifted.

Assembler Syntax:

```
Ap=An ± Am
Bp=Bn ± Bm
Ap=An ± Bm
Bp=Bn ± Am
Ap=(An * 2) ± Am
Bp=(Bn * 2) ± Bm
Ap=(An * 2) ± Bm
Bp=(Bn * 2) ± Am
```

Example:

```
a3 = a2-a1
```


Copyright 2013 Cirrus Logic

5

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Restrictions:

Destination:

```
a0
a1
a2
a3
b0
b1
b2
b3
```

5.2.3 Conditional Operation - Maximum

Two accumulators are compared. If comparison is true, an accumulator to accumulator move is performed. This accumulator move is a full 72-bit move and does not pass through the SRS. See Section 2.5.1 for an example.

Assembler Syntax:

```
if (Bn>Bm) An=Am
if (An>Am) Bn=Bm
if (Bn>Am) An=Bm
if (An>Bm) Bn=Am
```

Example:

```
if (b0>b3) a0=a3
if (a1>a2) b1=b2
if (b1>a1) a1=b1
if (a2>b2) b2=a2
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Copyright 2013 Cirrus Logic

5.2.4 Conditional Operation - Minimum

Two accumulators are compared. If comparison is true, an accumulator to accumulator move is performed. This accumulator move is a full 72-bit move and does not pass through the SRS.

Assembler Syntax:

```
if (Bn<Bm) An=Am
if (An<Am) Bn=Bm
if (Bn<Am) An=Bm
if (An<Bm) Bn=Am
```

Example:

```
if (b0<b3) a0=a3
if (a1<a2) b1=b2
if (b1<a1) a1=b1
if (a2<b2) b2=a2
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.5 Conditional Operation - Absolute Value Maximum

The absolute values of two accumulators are compared. If comparison is true, an accumulator to accumulator move is performed. This accumulator move is a full 72-bit move and does not pass through the SRS.

Assembler Syntax:

```
if (|Bn|>|Bm|) An=Am
if (|An|>|Am|) Bn=Bm
if (|Bn|>|Am|) An=Bm
if (|An|>|Bm|) Bn=Am
```

Example:

```
if (|b0|>|b3|) a0=a3
if (|a1|>|a2|) b1=b2
if (|b1|>|a1|) a1=b1
if (|a2|>|b2|) b2=a2
```


Copyright 2013 Cirrus Logic

5

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.6 Conditional Operation - Absolute Value Minimum

The absolute values of two accumulators are compared. If comparison is true, an accumulator to accumulator move is performed. This accumulator move is a full 72-bit move and does not pass through the SRS.

Assembler Syntax:

```
if (|Bn|<|Bm|) An=Am
if (|An|<|Am|) Bn=Bm
if (|Bn|<|Am|) An=Bm
if (|An|<|Bm|) Bn=Am
```

Example:

```
if (|b0|<|b3|) a0=a3
if (|a1|<|a2|) b1=b2
if (|b1|<|a1|) a1=b1
if (|a2|<|b2|) b2=a2
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.7 Bitwise Accumulator Move

Bitwise accumulator move. This accumulator move is a full 72-bit move and does not pass through the SRS.

Assembler Syntax:

```
An =+ Am
Bn =+ Bm
An =+ Bm
Bn =+ Am
```


Copyright 2013 Cirrus Logic

Example:

```
a0 =+ b3
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.8 Parallel Bitwise Accumulator Move

This is a dual bitwise accumulator move. All 72 bits of the accumulators are transferred. The move does not pass through the SRS. An accumulator can be both a destination and a source, so this instruction can successfully be used to swap the entire contents of two accumulators.

For example, if two accumulators have the following values:

```
a0 = 0x1234567890
b0 = 0x0987654321
```

and the following instruction is executed:

```
a0 =+ b0; b0 =+ a0
```

after execution the accumulators have been swapped:

```
a0 = 0x0987654321
b0 = 0x1234567890
```

Assembler Syntax:

```
An =+ Am; Bn =+ Bm
An =+ Bm; Bn =+ Am
```

Example:

```
a0 =+ b3;b0 =+ a3
a0 =+ a1;b0 =+ b1
a0 =+ b1;b0 =+ a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Copyright 2013 Cirrus Logic

5

5.2.9 Bitwise Complement

The one's complement of an accumulator is stored in an accumulator.

Assembler Syntax:

```
AccumAccum =~ Accum;
An =~ Am
Bn =~ Bm
An =~ Bm
Bn =~ Am
```

Example:

```
a1 =~ b0
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Note: Either A0,AS or B0,BS can be affected based on which accumulator is used.

5.2.10 Parallel Bitwise Complement

The one's complement of an accumulator is stored in an accumulator.

Assembler Syntax:

```
An =~ Am; Bn =~ Bm
An =~ Bm; Bn =~ Am
```

Example:

```
a0 =~ a1;b0 =~ b1
a0 =~ b1;b0 =~ a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Copyright 2013 Cirrus Logic

5.2.11 AccumNegative Accumulator Move

Computes the two's complement negative of the value in an accumulator and stores the result in an accumulator.

Assembler Syntax:

```
Accum =- Accum;
An =- Am
Bn =- Bm
An =- Bm
Bn =- Am
```

Example:

```
b2 =- b1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Note: Either A0,AS or B0,BS can be affected based on which accumulator is used.

5.2.12 Parallel Negative Accumulator Move

Computes the two's complement negative of the value in an accumulator and stores the result in an accumulator.

Assembler Syntax:

```
An =- Am; Bn =- Bm
An =- Bm; Bn =- Am
```

Example:

```
a0 =- a1;b0 =- b1
a2 =- b1;b2 =- a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Copyright 2013 Cirrus Logic

5

5.2.13 Absolute Value Accumulator Move

Absolute value of an accumulator is stored in an accumulator.

Assembler Syntax:

```
Accum = |Accum|
An = |Am|
Bn = |Bm|
An = |Bm|
Bn = |Am|
```

Example:

```
a0 = |a1|
a3 = |b2|
b3 = |b3|
b0 = |a3|
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Either A0,AS or B0,BS can be affected based on which accumulator is used.

Note:

5.2.14 Parallel Absolute Value Accumulator Move

Absolute values of two accumulators are stored in two accumulators.

Assembler Syntax:

```
An = |Am|; Bn = |Bm|
An = |Bm|; Bn = |Am|
```

Example:

```
a0 = |a1|; b0 = |b1|
a3 = |b2|; b3 = |a2|
```


Copyright 2013 Cirrus Logic

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.15 Bitwise OR

The bitwise OR of two accumulators is stored in an accumulator.

Assembler Syntax:

```
An = An | Am
Bn = Bn | Bm
An = An | Bm
Bn = Bn | Am
```

Example:

```
a0 = a0 | a3
b3 = b3 | b3
a1 = a1 | b2
b2 = b2 | a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.16 Parallel Bitwise OR

The bitwise OR of two accumulators is stored in an accumulator.

Assembler Syntax:

```
An = An | Am;Bn = Bn | Bm
An = An | Bm;Bn = Bn | Am
```

Example:

```
a0 = a0 | a3;b0 = b0 | b3
a0 = a0 | b3;b0 = b0 | a3
```


Copyright 2013 Cirrus Logic

5

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.17 Bitwise Exclusive OR

The Bitwise Exclusive OR of two accumulators is stored in an accumulator.

Assembler Syntax:

```
An = An ^ Am
Bn = Bn ^ Bm
An = An ^ Bm
Bn = Bn ^ Am
```

Example:

```
a0 = a0 ^ a3
b3 = b3 ^ b3
a1 = a1 ^ b2
b2 = b2 ^ a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.18 Parallel Bitwise Exclusive OR

The Bitwise Exclusive OR of two accumulators is stored in an accumulator.

Assembler Syntax:

```
An = An ^ Am;Bn = Bn ^ Bm
An = An ^ Bm;Bn = Bn ^ Am
```

Example:

```
a0 = a0 ^ a3;b0 = b0 ^ b3
a1 = a1 ^ b2;b1 = b1 ^ a2
```


Copyright 2013 Cirrus Logic

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.19 Bitwise AND

The Bitwise AND of two accumulators is stored in an accumulator.

Assembler Syntax:

```
An = An & Am
Bn = Bn & Bm
An = An & Bm
Bn = Bn & Am
```

Example:

```
a0 = a0 & a3
b3 = b3 & b3
a1 = a1 & b2
b2 = b2 & a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.20 Parallel Bitwise AND

Parallel bitwise accumulator ANDs.

Assembler Syntax:

```
An = An & Am;Bn = Bn & Bm
An = An & Bm;Bn = Bn & Am
```

Example:

```
a1 = a1 & a3;b1 = b1 & b3
a2 = a2 & b2;b2 = b2 & a2
```


Copyright 2013 Cirrus Logic

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.21 Bitwise Zero

Zero all 72 bits of the designated accumulator.

Assembler Syntax:

```
Accum = 0
```

Example:

```
a2 = 0
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Note: Either A0,AS or B0,BS can be affected based on which accumulator is used.

5.2.22 Parallel Bitwise Zero

Zero all 72 bits of the designated accumulator.

Assembler Syntax:

```
An = 0; Bn = 0
```

Example:

```
a2 = 0;b2 = 0
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Copyright 2013 Cirrus Logic

5.2.23 Bitwise Shift Left by One

Accumulator is shifted left by one. A zero is placed in the least significant bit, the most significant bit is lost.

Assembler Syntax:

```
An = An << 1
Bn = Bn << 1
```

Example:

```
a0 = a0 << 1
b2 = b2 << 1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.24 Parallel Bitwise Shift Left by One

Accumulator is shifted left by one. A zero is placed in the least significant bit, the most significant bit is lost.

Assembler Syntax:

```
An = An << 1; Bn = Bn << 1
```

Example:

```
a0 = a0 << 1; b0 = b0 << 1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.25 Bitwise Shift Left by Four

Accumulator is shifted left by four. Four zeros are placed in the least significant bits. The most significant 4 bits are lost.

Copyright 2013 Cirrus Logic

5

Assembler Syntax:

```
An = An << 4
Bn = Bn << 4
```

Example:

```
a0 = a0 << 4
b2 = b2 << 4
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.26 Parallel Bitwise Shift Left by Four

Accumulator is shifted left by four. Four zeros are placed in the least significant bits. The most significant 4 bits are lost.

Assembler Syntax:

```
An = An << 4;Bn = Bn << 4
```

Example:

```
a0 = a0 << 4;b0 = b0 << 4
a3 = a3 << 4;b3 = b3 << 4
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.27 Bitwise Shift Left by Eight

Accumulator is shifted left by eight. Eight zeros are placed in the least significant bits. The most significant 8 bits are lost.

Assembler Syntax:

```
An = An << 8
Bn = Bn << 8
```


Copyright 2013 Cirrus Logic

Example:

```
a0 = a0 << 8
b2 = b2 << 8
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.28 Parallel Bitwise Shift Left by Eight

Accumulator is shifted left by eight. Eight zeros are placed in the least significant bits, the most significant 8 bits are lost.

Assembler Syntax:

```
An = An << 8;Bn = Bn << 8
```

Example:

```
a0 = a0 << 8;b0 = b0 << 8
a3 = a3 << 8;b3 = b3 << 8
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.29 Bitwise Shift Right by One

Accumulator is shifted right by one. The most significant bit is sign extended from the current value, the least significant bit is lost.

Assembler Syntax:

```
An = An >> 1
Bn = Bn >> 1
```

Example:

```
a2 = a2 >> 1
b3 = b3 >> 1
```


Copyright 2013 Cirrus Logic

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.30 Parallel Bitwise Shift Right by One

Accumulator is shifted right by one. The most significant bit is sign extended from the current value, the least significant bit is lost.

Assembler Syntax:

```
An = An >> 1;Bn = Bn >> 1
```

Example:

```
a2 = a2 >> 1;b2 = b2 >> 1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.31 Bitwise Test

The bitwise AND of the accumulators is performed, and the appropriate bits in the CCR are set according to the first accumulator. Neither accumulator is altered.

Assembler Syntax:

```
An & Am
Bn & Bm
An & Bm
Bn & Am
```

Example:

```
a0 & a1
b2 & b2
a1 & b1
b2 & a3
```


Copyright 2013 Cirrus Logic

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.32 Parallel Bitwise Test

The bitwise AND of the accumulators is performed, and the appropriate bits in the CCR are set according to the first accumulator. Neither accumulator is altered.

Assembler Syntax:

```
An & Am;Bn & Bm
An & Bm;Bn & Am
```

Example:

```
a0 & a1;b0 & b1
a1 & b1;b1 & a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.33 Bitwise Compare

A bitwise comparison of the accumulators is performed, and the appropriate bits in the CCR are set according to the first accumulator. Neither accumulator is altered.

Assembler Syntax:

```
An - Am
Bn - Bm
An - Bm
Bn - Am
```

Example:

```
a0 - a1
b2 - b2
a1 - b1
b2 - a3
```


Copyright 2013 Cirrus Logic

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.34 Parallel Bitwise Compare

A bitwise comparison of the accumulators is performed, and the appropriate bits in the CCR are set according to the first accumulator. Neither accumulator is altered.

Assembler Syntax:

```
An - Am;Bn - Bm
An - Bm;Bn - Am
```

Example:

```
a0 - a1;b0 - b1
a1 - b1;b1 - a1
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.35 Bitwise Absolute Value Compare

A bitwise comparison of the absolute values of the accumulators is performed, and the appropriate bits in the CCR are set according to the first accumulator. Neither accumulator is altered.

Assembler Syntax:

```
|An| - |Am|
|Bn| - |Bm|
|An| - |Bm|
|Bn| - |Am|
```


Copyright 2013 Cirrus Logic

Example:

```
|a0|-|a1|
|b2|-|b2|
|a1|-|b1|
|b2|-|a3|
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

5.2.36 Parallel Bitwise Absolute Value Compare

A bitwise comparison of the absolute values of the accumulators is performed, and the appropriate bits in the CCR are set according to the first accumulator. Neither accumulator is altered.

Assembler Syntax:

```
|An| - |Am|;|Bn| - |Bm|
|An| - |Bm|;|Bn| - |Am|
```

Example:

```
|a0|-|a1|; |b0|-|b1|
|a1|-|b1|; |b1|-|a1|
```

Flags Affected:

A0	A zero
AS	A sign
B0	B zero
BS	B sign

Copyright 2013 Cirrus Logic

Appendix A
Glossary

A

Table A-1. Glossary Terms

Term	Definition
AGU	Address Generation Unit.
ALU	Arithmetic Logic Unit.
API	Application Programmers Interface.
BIBO	Bounded Input Bounded Output.
CASMSPEC	Environment variable that is the default assembler option.
CCR	Condition Code Register.
Complex Memory	Treating the same address across both X and Y Data memory as one complex value with the real part in X and the imaginary part in Y.
DSPAB	DSPA and DSPB.
DSPC	The third DSP in CS495xx.
FFT	Fast Fourier Transform.
Fast Interrupts Short Interrupts One-Instruction Interrupts	Interrupts that consist solely of a single instruction.
IFFT	Inverse Fast Fourier Transform.
IMG file	The image (RAM) information for an application.
ISI	Interrupt Service Instruction.
ISR	Interrupt Service Routine.
Limiting Saturating	These terms are used interchangeably.
Long Interrupts Slow Interrupts Multi-Instruction Interrupts	If an interrupt needs to execute more than one instruction, the callint instruction is used for the ISI. This is referred to as a Long Interrupt. The callint instruction disables interrupts, pushes the PC onto the Subroutine Stack, and starts executing the specified ISR. The final instruction of the ISR should be retcc, which pops the PC and enable interrupts. The call or jmp instructions can also be used as ISIs, but they will not disable interrupts, allowing the possibility of code reentrance. This is especially dangerous when using call due to the possibility of overflowing the Subroutine Stack, which leaves the processor in an unknown state.
Long Memory	Treating the same address across both X and Y Data memory as one double precision 64-bit value.
MAC	Multiply-Accumulator.
MR	Mode Register.

Copyright 2013 Cirrus Logic

Table A-1. Glossary Terms (Continued)

Term	Definition
Multifunction Instruction	Combining a 16-bit encoded move in the most significant 16 bits of a 32-bit opcode with a 16-bit encoded MAC/ALU operation in the least significant 16 bits results in a 32-bit multifunction instruction. A "NOP" is a valid 16-bit instruction for either half.
O file	The assembled form of a portion of a module. Output from: casm.exe Input to: clib.exe (or clink.exe)
Parallel or Dual Instructions	Encoding two moves or two MAC/ALU operations in one 16-bit opcode results in a Parallel Instruction. Parallel Instructions can be used in Multifunction Instructions.
SRS	Shifter/Rounder/Saturator.
ULD file	The image information for an application, possibly encrypted and properly formatted for booting.

Copyright 2013 Cirrus Logic

Appendix B

List of Instructions by Category and Flag Reference

B

Table B-1. Instruction / Flag Reference Table

Instructions with Links to Instructions / Flags (Red)
Execution Control Instructions (Section 3.3 on page 3-2) do - Start Hardware Loop enddo - End Current Do-Loop do_patch - Jump to Patch jmp - Jump if - Jump Conditionally call - Jump To Subroutine callint - Answer Interrupt callint_stq - Answer Stack Interrupt ret - Return From Subroutine retint - Return From Interrupt retint_stq - Return From Stack Interrupt inten - Enable Interrupts intdis - Disable Interrupts halt - Stop Further Execution nop - No Operation
64-bit Peripheral Moves (Section 3.4 on page 3-12) XY Register Pair = ext(16-bit Address)] Accum = ext(16-bit Address) ext(16-bit Address) = XY Register Pair ext(16-bit Address) = Accum (L, T1, T0) logexp = XY Register Pair XY Register Pair = logexp
Memory Moves - Direct (Section 3.5 on page 3-19) Any Reg = xmem[16-bit Address] xmem[16-bit Address] = Any Reg (L, T1, T0) Any Reg = ymem[16-bit Address] xmem[16-bit Address] = Any Reg (L, T1, T0) MS Reg - See Table 2-25. pmem[16-bit Address] = Any Reg (L, T1, T0) Any Reg = inp[16-bit Address] outp[16-bit Address] = Any Reg (L, T1, T0) Any Reg = xmem[Index Register] xmem[Index Register] = Any Reg (L, T1, T0) Any Reg = ymem[Index Register] ymem[Index Register] = Any Reg (L, T1, T0) Any Reg = pmem[Index Register] pmem[Index Register] = Any Reg (L, T1, T0) outp[Index Register] = Any Reg (L, T1, T0) Any Reg = inp[Index Register]

Copyright 2013 Cirrus Logic

B

Table B-1. Instruction / Flag Reference Table (Continued)

Instructions with Links to Instructions / Flags (Red)
Immediate Register Loads (Section 3.6 on page 3-36) fixed16(Destination) = (16-bit Data) ufixed16(Destination) = (16-bit Data) uhalfword(Destination) = (16-bit Data) Index Register = (16-bit Data) NM Register = (16-bit Data) Guard Register = (8-bit Data) halfword(Destination) = (16-bit Data) lo16(Destination) = (16-bit Data) MS Reg = (16-bit Data) AnyReg(Any Reg, Any Reg) (L, T1, T0) Any Reg = MS Reg (L, T1, T0) MS Reg = Any Reg (L, T1, T0) AnyReg (Any Reg, Any Reg), (Any Reg, Any Reg) (L, T1, T0) nm4 - nm7 (L, T1, T0) In = Im/(0) ± (16-bit Data)
Bit Manipulation Instructions (Section 3.7.1 on page 3-48) **Bit Test (Z or Zero)** **Bit Set (Z or Zero)** **Bit Clear (Z or Zero)** **MS Reg - See Table 2-25. (Z or Zero)**
Multifunction Moves (Section 4.1 on page 4-1) DP Reg = xmem[Index Register] DP Reg = xmem[6-bit Address] xmem[Index Register] = DP Reg xmem[6-bit address] = DP Reg (L, T1, T0) DP Reg = ymem[Index Register] DP Reg = ymem[6-bit address] ymem[Index Register] = DP Reg ymem[6-bit address] = DP Reg (L, T1, T0) **Data Path Register to or from Any Register** DP Reg = Any Reg Any Reg = DP Reg (L, T1, T0)
Parallel Multifunction Moves (Section 4.2 on page 4-10) **Data Path Register to/from X or Y Memory** Xn = xmem[Index Register] xmem[Index Register] = An (L, T1, T0) Ym = ymem[Index Register] ymem[Index Register] = Bm (L, T1, T0)
Data Path Register to Data Path Register (Section 4.3 on page 4-13) DP Reg = DP Reg
64-bit Multifunction Moves **Data Path Register Pair to or from XY Memory (Section 4.5.1 on page 4-16)** Data Path Register Pair = xymem[Index Register] Data Path Register Pair = xymem[6-bit Address] xymem[Index Register] = Data Path Register Pair xymem[6-bit Address] = Data Path Register Pair (L, T1, T0) **Accumulator to or from XY Memory (Section 4.5.2 on page 4-18)** Accum = xymem[Index Register] Accum = xymem[6-bit Address] xymem[Index Register] = Accum xymem[6-bit Address] = Accum (L, T1, T0)
Index Register Updates (Section 4.6 on page 4-19) **In = Im ± (6-bit Data)** **In ±= 1/2/N**

Copyright 2013 Cirrus Logic

Table B-1. Instruction / Flag Reference Table (Continued)

Instructions with Links to Instructions / Flags (Red)
Multifunction Operations **Multifunction Arithmetic Instructions (Section 5.1.1 on page 5-1)** Parallel Multiply/Multiply-Accumulate I Parallel Multiply/Multiply-Accumulate II Real Multiply/Multiply-Accumulate Parallel Squares Parallel Multiply with Add Multiply by One with Optional Accumulate Parallel Multiply by One with Optional Accumulate
Multifunction Accumulator Instructions (Section 5.2.1 on page 5-7) Parallel Add with Shift (A0, AS, B0, BS) Add with Shift (A0, AS, B0, BS) Conditional Operation - Maximum (A0, AS, B0, BS) Conditional Operation - Minimum (A0, AS, B0, BS) Conditional Operation - Absolute Value Maximum (A0, AS, B0, BS) Conditional Operation - Absolute Value Minimum (A0, AS, B0, BS) Bitwise Accumulator Move (A0, AS, B0, BS) Parallel Bitwise Accumulator Move (A0, AS, B0, BS) Bitwise Complement (A0, AS, B0, BS) Parallel Bitwise Complement (A0, AS, B0, BS) AccumNegative Accumulator Move (A0, AS, B0, BS) Parallel Negative Accumulator Move (A0, AS, B0, BS) Absolute Value Accumulator Move (A0, AS, B0, BS) Parallel Absolute Value Accumulator Move (A0, AS, B0, BS) Bitwise OR (A0, AS, B0, BS) Parallel Bitwise OR (A0, AS, B0, BS) Bitwise Exclusive OR (A0, AS, B0, BS) Parallel Bitwise Exclusive OR (A0, AS, B0, BS) Bitwise AND (A0, AS, B0, BS) Parallel Bitwise AND (A0, AS, B0, BS) Bitwise Zero (A0, AS, B0, BS) Parallel Bitwise Zero (A0, AS, B0, BS) Bitwise Shift Left by One (A0, AS, B0, BS) Parallel Bitwise Shift Left by One (A0, AS, B0, BS) Bitwise Shift Left by Four (A0, AS, B0, BS) Parallel Bitwise Shift Left by Four (A0, AS, B0, BS) Bitwise Shift Left by Eight (A0, AS, B0, BS) Parallel Bitwise Shift Left by Eight (A0, AS, B0, BS) Bitwise Shift Right by One (A0, AS, B0, BS) Parallel Bitwise Shift Right by One (A0, AS, B0, BS) Bitwise Test (A0, AS, B0, BS) Parallel Bitwise Test (A0, AS, B0, BS) Bitwise Compare (A0, AS, B0, BS) A bitwise comparison of the accumulators is performed, and the appropriate bits in the CCR are set according to the first accumulator. Neither accumulator is altered. (A0, AS, B0, BS) Bitwise Absolute Value Compare (A0, AS, B0, BS) Parallel Bitwise Absolute Value Compare (A0, AS, B0, BS)

Copyright 2013 Cirrus Logic

B

Revision History

Revision	Date	Changes
UM7	December, 2011	Updated Section 1.4.16.3 to document .data_ovly, .xdata_ovly, .ydata_ovly, and .code_ovly segment macros. Added Section 3.3.16.
UM8	March, 2012	Added .undef token to Section 1.4.16.6. Updated Section 1.4.16.9 to explain struct inside struct.
UM9	June, 2012	Added example to Table 1-8. Added Section 1.4.14, Section 2.7.3, and Section 2.7.4. Updated Section 3.4.5.
UM10	April, 2013	Updated description of jsr_data Register in Section 2.4.9. Added new status bits to MR register in Section 2.4.19. Added examples to Section 2.4.21.
UM11	September, 2013	Updated .strpos example in Table 1-9. Updated description of .extern <symbols> and added .export <symbols> to Macros Table 1-10.

Copyright 2013 Cirrus Logic