Algorithmic C™ Datatypes

Software Version 2.6

July 2011

© 2005-2011 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/
Send Feedback on Documentation: supportnet.mentor.com/user/feedback_form.cfm

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other third parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the respective third-party owner. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/terms_conditions/trademarks.cfm.

Table of Contents

List of Tables

Chapter 1
Overview of Algorithmic C Datatypes

Introduction

The arbitrary-length bit-accurate integer and fixed-point datatypes provide an easy way to model static bit-precision with minimal runtime overhead. The datatypes were developed in order to provide a basis for writing bit-accurate algorithms to be synthesized into hardware. The Algorithmic C data types are used in Catapult C Synthesis, a tool that generates optimized RTL from algorithms written as sequential ANSI-standard C/C++ specifications. Operators and methods on both the integer and fixed-point types are clearly and consistently defined so that they have well defined simulation and synthesis semantics.

The precision of the integer type ac_int<W,S> is determined by template parameters W (integer that gives bit-width) and S (a boolean that determines whether the integer is signed or unsigned). The fixed-point type ac_fixed<W,I,S,Q,O> has five template parameters which determine its bit-width, the location of the fixed-point, whether it is signed or unsigned and the quantization and overflow modes (see "Quantization and Overflow" on page 15) that are applied when constructing or assigning to object of its type.

The advantages of the Algorithmic C datatypes over the existing integer and fixed-point datatypes are the following:

- *Arbitrary-Length*: This allows a clean definition of the semantics for all operators that is not tied to an implementation limit. It is also important for writing general IP algorithms that don't have artificial (and often hard to quantify and document) limits for precision.
- *Precise Definition of Semantics*: Special attention has been paid to define and verify the simulation semantics and to make sure that the semantics are appropriate for synthesis. No simulation behavior has been left to compiler dependent behavior. Also, asserts have been introduced to catch invalid code during simulation. See also "User Defined Asserts" on page 55.
- *Simulation Speed*: The implementation of ac_int uses sophisticated template specialization techniques so that a regular C++ compiler can generate optimized assembly language that will run much faster than the equivalent SystemC datatypes. For example, ac_int of bit widths in the range 1 to 32 can run 100x faster than the corresponding sc_bigint/sc_biguint datatype and 3x faster than the corresponding sc_int/sc_uint datatype.
- *Correctness*: The simulation and synthesis semantics have been verified for many size combinations using a combination of simulation and equivalence checking.

- *Compilation Speed and Smaller Executable*: Code written using ac_int datatypes compiles 5x faster even with the compiler optimizations turned on (required to get fast simulation). It also produces smaller binary executables.
- *Consistency*: Consistent semantics of ac_int and ac_fixed.

Definition and Implementation Overview

The ac_int and ac_fixed datatypes were defined and implemented adhering to the following guiding principles:

- *Static Bit Widths:* all operations and methods return an object with a bit width that is statically determinable from the bit widths of the inputs and "signedness" (signed vs. unsigned) of the inputs. Keeping bit-widths static is essential for fast simulation, as it means that memory allocation is completely avoided. It is also essential for synthesis. For example the left shift operation of an ac_int returns an ac_int of the same type (width and signedness) as the type of first operand. In contrast, the left shift for sc_bigint or sc_biguint returns an object with precision that depends on the shift value and has no practical bound on its bit width.
- *Operations Defined Arithmetically:* whenever possible, operations are defined arithmetically, that is, the inputs are treated as arithmetic values and the result value is returned with a type (bitwidth and signedness) that is capable of representing the value without loss of precision. Exceptions to this rule are the shift operators (to maintain static bit widths) and division.
- *Compiler Independent Semantics:* the semantics avoid "implementation dependent" behaviors that are present for some native C integer operations. For example, shift values for a C int needs to be in the range [0,31] and are otherwise implementation dependent.
- *Mixed ac_int, ac_fixed and C integer type Binary Functions:* all binary operators are defined for mixed ac_int, ac_fixed and native C integers for consistency. For example the expression "1 + a" where a is an ac_int<36,true> will compute "ac_int<32,true> 1 + a" rather than "1 + (int) a". This is done to ensure that expressions are carried out without unintentional loss of precision and to make sure that compiler errors due to ambiguities are avoided.

The types have a public interface. The base class implementation of these types are private. They are part of the implementation and may be changed. Also any function or class under namespace *ac_private* should not be used as it may be subject to change.

Usage

In order to use the Algorithmic C data types, either the header file **ac_int.h** or **ac_fixed.h** needs to be included in the source. All the definitions are in these files and there are no object files that need to be linked in. Enabling compiler optimizations (for example "-O3" in GCC) is critical to the fastest runtime. Explicit conversion functions to SystemC integer and fixed-point datatypes are provided in the header file **ac_sc.h**. Chapter 4, "Datatype Migration Guide," presents information about how to convert algorithms written with SystemC datatypes to Algorithmic C datatypes.

Chapter 2 Arbitrary-Length Bit-Accurate Integer and Fixed-Point Datatypes

Introduction

The arbitrary-length bit-accurate integer and fixed-point datatypes provide an easy way to model static bit-precision with minimal runtime overhead. Operators and methods on both the integer and fixed-point types are clearly and consistently defined so that they have well defined simulation and synthesis semantics.

The types are named *ac_int* and *ac_fixed* and their numerical ranges are given by their template parameters as shown in Table 2-1. For both types, the boolean template parameter determines whether the type is constrained to be unsigned or signed. The template parameter *W* specifies the number of bits for the integer or fixed point number and must be a positive integer. For *ac_int* the value of the integer number $b_{W-1}...b_1b_0$ is interpreted as an unsigned integer or a signed (two's complement) number.The advantage of having the signedness specified by a template parameter rather than having two distinct types is that it makes it possible to write generic functions where signedness is just a parameter.

Table 2-1. Numerical Ranges of ac_int and ac_fixed

Type	Description	Numerical Range	Quantum
ac_int<W, false>	unsigned integer	0 to $2^W - 1$	1
ac_int<W, true>	signed integer	-2^{W-1} to $2^{W-1} - 1$	1
ac_fixed<W, I, false>	unsigned fixed-point	0 to $(1 - 2^{-W})\ 2^I$	2^{I-W}
ac_fixed<W, I, true>	signed fixed-point	$(-0.5)\ 2^I$ to $(0.5 - 2^{-W})\ 2^I$	2^{I-W}

For *ac_fixed*, the second parameter *I* of an ac_fixed is an integer that determines the location of the fixed-point relative to the MSB. The value of the fixed-point number $b_{W-1}...b_1b_0$ is given by $(.b_{W-1}...b_1b_0)\ 2^I$ or equivalently $(b_{W-1}...b_1b_0)\ 2^{I-W}$ where $b_{W-1}...b_1b_0$ is interpreted as an unsigned integer or a signed (two's complement) number.

Table 2-2 shows examples for various integer and fixed-point types with their respective numerical ranges and quantum values. The quantum is the smallest difference between two numbers that are represented. Note that an *ac_fixed<W,W,S>* (that is *I==W*) has the same numerical range as an *ac_int<W,S>* where S is a boolean value that determines whether the type is signed or unsigned. The numerical range of an *ac_fixed<W,I,S>* is equal to the numerical range of and *ac_int<W,S>* (or an *ac_fixed<W,W,S>*) multiplied by the quantum.

Table 2-2. Examples for ac_int and ac_fixed

Type	Numerical Range	Quantum
ac_int<1, false>	0 to1	1
ac_int<1, true>	-1 to 0	1
ac_int<4, false>	0 to 15	1
ac_int<4, true>	-8 to 7	1
ac_fixed<4, 4, false>	0 to 15	1
ac_fixed<4, 4, true>	-8 to 7	1
ac_fixed<4, 6, false>	0 to 60	4
ac_fixed<4, 6, true>	-32 to 28	4
ac_fixed<4, 0, false>	0 to 15/16	1/16
ac_fixed<4, 0, true>	-0.5 to 7/16	1/16
ac_fixed<4,-1, false>	0 to 15/32	1/32
ac_fixed<4,-1, true>	-0.25 to 7/32	1/32

It is important to remember when dealing with either an *ac_fixed* or an *ac_int* that in order for both +1 and -1 to be in the numerical range, *I* and *W* have to be at least 2. For example, ac_fixed<6,1,true> has a range from -1 to +0.96875 (it does not include +1) while ac_fixed<6,2,true> has a range -2 to +1.9375 (includes +1).

The fixed-point datatype *ac_fixed* has two additional template parameters that are optional that define the overflow mode (*e.g*. saturation) and the quantization mode (*e.g*. rounding):

```
ac_fixed<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>
```

Quantization and overflow occur when assigning (=, +=, etc.) or constructing (including casting) where the target does not represent the source value without loss of precision (this will be covered more precisely in "Quantization and Overflow" on page 15). In all the examples of Table 2-2 the default quantization and overflow modes AC_TRN and AC_WRAP are implied. The default modes simply throw away bits to the right of LSB and to the left of the MSB which is also the behavior of ac_int:

```
ac_fixed<1,1,true> x = 1;  // range is [-1,0], +1 wraps around to -1
ac_int<1,true> x     = 1;  // same as above
ac_fixed<4,4,true> x = 9;  // range is [-8,7], +9 wraps around to -7
ac_int<4,true>     x = 9;  // same as above
ac_fixed<4,4,true> x = 3.7;  // truncated to 3.0
ac_int<4,true>     x = 3.7;  // same as above
ac_fixed<4,4,true> x = -3.2; // truncated to -4.0
```

```
ac_int<4,true>     x = -3.2; // same as above
```

Table 2-3 shows the operators defined for both ac_int and ac_fixed

Table 2-3. Operators defined for ac_int and ac_fixed.

Operators	**ac_int**	**ac_fixed**
Two operand +, -, *, /, %, |, &, ^	Arithmetic result. First or second arg may be C *INT* or *ac_fixed* / truncates towards 0	Arithmetic result. First or second arg may be *ac_int* or C *INT* / truncates towards 0. % NOT DEFINED
>>, <<	bidirectional return type is type of first operand Second arg is *ac_int* or C *INT*	bidirectional return type is type of first operand Second arg is *ac_int* or C *INT*
=	assignment	quantization, then overflow handling specified by target
+=, -=, *=, /=, %=, |=, &=, ^=, >>=, <<=	Equiv to op then assign. First arg is *ac_int*	Equiv to op then assign. First arg is *ac_fixed*
==, !=, >, <, >=, <=	First or second arg may be C *INT* or *ac_fixed*	First or second arg may be *ac_int* or C *INT* or C *double*
Unary +, -, ~	Arithmetic	Arithmetic
++x, x++, --x, x--	Pre/post incr/dec by 1	Pre/post incr/dec by 2^{I-W}
! x	Equiv to x == 0	Equiv to x == 0
(long long)	defined for ac_int<W,true>, W <= 64	NOT DEFINED
(unsigned long long)	defined for ac_int<W,false>, W <= 64	NOT DEFINED
x[i]	returns *ac_int::ac_bitref* index: *ac_int*, *unsigned*, *int* asserts for index out of bound	returns *ac_fixed::ac_bitref* index: *ac_int*, *unsigned*, *int* asserts for index out of bounds

Note that for convenience the conversion operators to (long long) and signed ac_int and (unsigned long long) for unsigned ac_int are defined for *W <= 64*. Among other things, this allows for the use of an *ac_int* as an index to a C array without any explicit conversion call.

Table 2-4 shows the methods defined for *ac_int* and *ac_fixed* types. The *slc* and *set_slc* methods are templatized to get or set a slice respectively. For *slc*, the width needs to be explicitly provided as a template argument. When using the *slc* method in a templatized function place the keyword *template* before it as some compilers may error out during parsing. For example:

```
template<int N> // not important whether or not N is used
int f(int x) {
   ac_int<32,true> t = x;
   ac_int<6,true> r = t.template slc<6>(4); // t.slc<6>(4) could error out
```

```
      return r.to_int();
   }
```

The *set_slc* method does not need to have a width specified as a template argument since the width is inferred from the width of the argument *x*. Many of the other methods are conversion functions. The *length* method returns the width of the type. The *set_val* method sets the ac_int or ac_fixed to a value that depends on the template parameter.

Table 2-4. Methods defined for ac_int and ac_fixed.

<table>
<tr><th>Methods</th><th>ac_int&lt;W,S&gt;</th><th>ac_fixed&lt;W,I,S,Q,O&gt;</th></tr>
<tr><td>slc&lt;W2>(int_type i)</td><td colspan="2">Returns slice of width W2 starting at bit index i, in other words slice (W2-1+i downto i).
Slice is returned as an ac_int<W2,S>.
Parameter <i>i</i> needs to be non-negative and could of any of the following types: <i>ac_int, unsigned, int</i></td></tr>
<tr><td>set_slc(
int_type i,
ac_int<W2,S2> x
)</td><td colspan="2">Bits of x are copied at slice with LSB index <i>i</i>. That is, bits (W2-1+i downto i) are set with bits of x.
Parameter <i>i</i> needs to be non-negative and could of any of the following types: <i>ac_int, unsigned, int</i></td></tr>
<tr><td>to_ac_int()</td><td>NOT DEFINED</td><td>return an ac_int<W_I,S> where W_I is max(I, 1). Equiv to AC_TRN quantization. Return type guarantees no overflows.</td></tr>
<tr><td>to_int(), to_uint(), to_long(), to_ulong(), to_int64(), to_uint64()</td><td>Conversions to various C <i>INTS</i></td><td>Conversions to various C <i>INTS</i> Equiv to to_ac_int() followed by conversion</td></tr>
<tr><td>to_double()</td><td>Conversion to double</td><td>Conversion to double</td></tr>
<tr><td>to_string(
ac_base_mode <i>base_rep</i>,
bool <i>sign_mag</i> = false
)</td><td colspan="2">convert to std::string depending on parameters <i>base_rep</i> {AC_HEX, AC_DEC, AC_OCT, AC_BIN} and <i>sign_mag</i></td></tr>
<tr><td>length()</td><td colspan="2">Returns bitwidth (value of template parameter W)</td></tr>
<tr><td>set_val&lt;ac_special_val&gt;()</td><td colspan="2">Set to special value specified by template parameter AC_VAL_DC, AC_VAL_0, AC_VAL_MIN, AC_VAL_MAX, AC_VAL_QUANTUM. See Table 2-17 on page 29 for details.</td></tr>
</table>

Table 2-5 shows the constructors that are defined for ac_int and ac_fixed. When constructing an ac_fixed, its quantization/overflow mode is taken into account. Initializing an ac_int or ac_fixed from floating-point (float or double) is not as runtime efficient as initializing from integers.

Table 2-5. Constructors defined for ac_int and ac_fixed.

Constructor argument	**ac_int**	**ac_fixed**
None: Default	does not initialize ac_int declared static are init to 0 by C++ before constructor is called	does not initialize ac_fixed declared static are init to 0 by C++ before constructor is called
bool (1-bit unsigned)		quantization/overflow
char (8-bit signed)		quantization/overflow
signed/unsigned char (8-bit signed/unsigned)		quantization/overflow
signed/unsigned short (16-bit signed/unsigned)		quantization/overflow
signed/unsigned int or long (32-bit signed/unsigned)		quantization/overflow
signed/unsigned long long (64-bit signed/unsigned)		quantization/overflow
double	Not as efficient	quantization/overflow Not as efficient
ac_int		quantization/overflow
ac_fixed	NOT DEFINED Use to_ac_int() instead	quantization/overflow

Quantization and Overflow

The fixed-point type *ac_fixed* provides quantization and overflow modes that determine how to adjust the value when either of the two conditions occur:

- *Quantization*: bits to the right of the LSB of the target type are being lost. The value may be adjusted by the following two strategies:
  - *Rounding*: choose the closest quantization level. When the value is exactly half way two quantization levels, which one is chosen depends on the specific rounding mode as shown in Table 2-6.
  - *Truncation*: choose the closest quantization level such that result (quantized value) is less than or equal the source value (truncation toward minus infinity) or such that the absolute value of the result is less than or equal the source value (truncation towards zero).

Note that quantization may trigger an overflow so it is always applied before overflow handling.

- *Overflow*: value after quantization is outside the range of the target as defined in Table 2-1 on page 11, except when the overflow mode is AC_SAT_SYM where the range is symmetric: $[-2^{W-1}+1, 2^{W-1}-1]$ in which case the most negative number -2^{W-1} triggers an overflow.

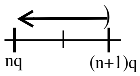
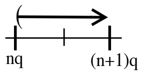
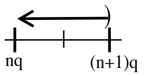
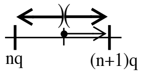
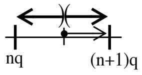
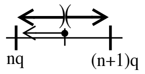
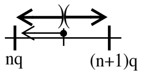
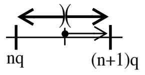
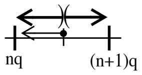
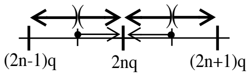
The modes are specified by the 4th and 5th template argument to ac_fixed:

ac_fixed<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>

that are of enumeration type *ac_q_mode* and *ac_o_mode* respectively. The enumeration values for ac_q_mode are shown in Table 2-6. The enumeration values for ac_o_modes are shown in Table 2-7 on page 17. The quantization and overflow mode default to AC_TRN and AC_WRAP:

```
ac_fixed<8,4,true> x; // equiv to ac_fixed<8,4,true,AC_TRN,AC_WRAP)
```

Table 2-6. Quantization modes for ac_fixed

Modes	**Behavior** **n is integer, q is 2^{I-W}**		**Simulation/Synthesis cost**
AC_TRN (**default**) (trunc towards $-\infty$)			Delete bits, no cost except for /= (div assign) signed
AC_TRN_ZERO (trunc towards 0)	n < 0 	n ≥ 0 	No cost for /=, or unsigned src For signed: incrementer, OR for deleted bits, AND with sign bit
AC_RND (round towards $+\infty$)			Various forms differ only on the direction of the rounding for values that are exactly at half point. All require an incrementer, some require to OR deleted bits, some only require to look at the MSB of the deleted bits.
AC_RND_ZERO (round towards 0)	n < 0 	n ≥ 0 	
AC_RND_INF (rounds towards $\pm\infty$)	n < 0 	n ≥ 0 	
AC_RND_MIN_INF (round towards $-\infty$)			
AC_RND_CONV (round towards even q multiples)			

For unsigned ac_fixed types, AC_TRN and AC_TRN_ZERO are equivalent, AC_RND and AC_RND_INF are equivalent, and AC_RND_ZERO and AC_RND_MIN_INF are equivalent. The AC_RND_CONV is a convergent rounding that rounds towards even multiples of the quantization. The quantization modes that have two columns (different directions for negative and positive numbers) are symmetric around 0 and are more costly as ac_fixed is represented in two's complement arithmetic. On the other hand signed-magnitude representations (for example floating point numbers) are more costly for asymmetric cases.

Quantization and overflow occur when assigning or constructing.

```
ac_fixed<8,1,true,Q,O> x = -0.1; // quantization, no overflow
ac_fixed<8,1,false,Q,O> y = x; // overflow (underflow) as y is unsigned
ac_fixed<4,1,true,Q,O> z= x; // quantization (dropping bits on the right)
(ac_fixed<4,1,true,Q,O>) x; // casting: same as above
ac_fixed<8,4,true,Q,O> a = ...;
ac_fixed<8,4,true,Q,O> b =
```

The behavior of the overflow modes are shown in Table 2-7. The default is AC_WRAP and requires no special handling (same behavior as with an ac_int). The AC_SAT mode saturates to the MIN or MAX limits of the range (as specified in Table 2-1 on page 11) of the target type (different for signed or unsigned targets). The AC_SAT_ZERO sets the value to zero when an overflow is detected. The AC_SAT_SYM makes only sense for signed targets. It saturates to +MAX or -MAX (note that -MAX = MIN+q). It not only saturates on overflow, but also when the value is MIN (it excludes the most negative number that would make the range asymmetric). Note however, that declaring an ac_fixed with AC_SAT_SYM does not guarantee that it stays symmetric as changing individual bits or slices in an ac_fixed does not trigger quantization or overflow handling.

Table 2-7. Overflow modes for ac_fixed

Mode	**Behavior** **all references are to *target* type** **MIN, MAX are limits as in Table 2-1**	**Simulation/Synthesis cost**
AC_WRAP (**default**)	Drop bits to the left of MSB	No cost
AC_SAT	Saturate to closest of MIN or MAX	Overflow checking and Saturation logic
AC_SAT_ZERO	Set to 0 on overflow	Overflow checking and Saturation logic
AC_SAT_SYM	For unsigned: treat as AC_SAT, For signed: on overflow **or** number is MIN set to closest of ±MAX .	Overflow checking and Saturation logic

Usage

In order to use the ac_int datatype the following file include should be used:

```
#include <ac_int.h>
```

The ac_int type is implemented with two template parameters to define its bitwidth and to indicate whether it is signed or unsigned:

```
ac_int<7, true> x;   // x is 7 bits signed
ac_int<19, false> y;  // y is 19 bits unsigned
```

In order to use the ac_fixed datatype the following file include should be used:

```
#include <ac_fixed.h>
```

The ac_fixed.h includes ac_int.h so it is not necessary to include both ac_int.h and ac_fixed.h.

The ac_fixed type is implemented with 5 template parameters that control the behavior of the fixed point type:

```
ac_fixed<int W, int I, bool S, ac_q_mode Q, ac_q_mode O>
```

where W is the width of the fixed point type, I is the number of integer bits, S is a boolean flag that determines whether the fixed-point is signed or unsigned, and Q and O are the quantization and overflow modes respectively (as shown in Table 2-4 on page 14 and Table 2-5 on page 15). The value of the fixed point is given by:

$$(0.b_{W-1}\ldots b_1 b_0)2^I$$

For example:

```
ac_fixed<4,4,true> x; // bbbb signed, AC_TRN, AC_WRAP
ac_fixed<4,0,false> x; // .bbbb unsigned AC_TRN, AC_WRAP
ac_fixed<4,7,false> x; // bbbb000, unsigned, AC_TRN, AC_WRAP
ac_fixed<4,-3,false> x; // .bbbb * pow(2, -3), unsigned, AC_TRN, AC_WRAP
```

Operators and Methods

This section provides a more detailed specification of the behavior of operators and methods including precisely defining return types. The operators and methods that are defined for ac_int and ac_fixed can be classified in some broad categories:

- Binary (two operand) operators:
  - Arithmetic, logical operators and arithmetic and logical assign operators: +, -, *, /, %, &, |, ^, +=, -=, *=, /=, %=, &=, |=, ^=. The modulo operators % and %= are not defined for ac_fixed. Mixing of ac_int, ac_fixed and native C integers is allowed.

  - Relational: the result is a boolean value (true/false): >, <, >=, <=, ==, !=. Mixing of ac_int, ac_fixed, native C integers and double is allowed.
  - Shift operator and shift assign operators: <<, >>, =<<, =>>. The second argument is an ac_int or a native C integer.

- Unary: (one operand) operators: +, -, ~, !. The ! operator returns bool.
- Pre/Post Increment/Decrement Operators: ++x, --x, x++, x--.
- Bit Select: operator [], returns an ac_int::ac_bitref or ac_fixed::ac_bitref. Allows reading and modifying bits of an ac_int or ac_fixed.
- Slicing Method slc<W>(int i) and set_slc(int i, const ac_int<W,S> &s) to read and modify a slice in an ac_int or ac_fixed. A slice of an ac_fixed is an ac_int.
- Conversion Operators to C native types and explicit conversion methods to C native types.
- Constructors from ac_int and C native types.
- I/O methods.

The concatenation operator is not defined for ac_int. Bit reversal may be defined in future releases.

Binary Arithmetic and Logical Operators

The two operand arithmetic and logical operators return an ac_fixed if either operand is an ac_fixed, otherwise the return type is ac_int. Binary arithmetic operators “+”, “*’, “/” and “%” and logical operators “&”, “|” and “^” return a signed ac_int/ac_fixed if either of the two operands is of type signed. The “-” operator always returns an ac_int/ac_fixed of type signed. The result for all operands with the exception of division is computed arithmetically and the bit width (and integer bit width for ac_fixed) of the result is such that the result is represented without loss of precision. The “/” operator is defined for both ac_int and ac_fixed and it returns a type that guarantees that the result does not overflow (see Table 2-8 on page 20 and Table 2-9 on page 21). The operator “%” is only defined for ac_int. Division by zero is not defined and will generate an exception.

The binary operators “&”, “|” and “^” return the bitwise “and”, “or” and “xor” of the two operands. The return type is signed if either of the two operands is signed. The two operands are treated arithmetically. For instance, if the operands are ac_fixed, the fixed point is aligned just as it is done for addition. Then operands are extended, if necessary, so that both operands are represented in the same type which is also the return type.

The arithmetic definition of the “bitwise” operators has the advantage that when mixing ac_int (or ac_fixed) operands of different lengths and signedness, the operations are associative:

```
(a | b) | c
```

returns the same value (and in this case the same type) as

```
a | (b | c)
```

Also operators are consistent

```
~(a | b) == ~a & ~b
```

Table 2-8 shows the list of binary (two operand) arithmetic and logical operators for ac_int and the return type based on the signedness and bit width of the two input operands. All operators shown in the table are defined arithmetically. The operator & could have been defined to return a more constrained type, $S_R = S_1$ & S_2 and $W_R = \text{abs}(\min(S_1 ? -W_1:W_1, S_2 ? -W_2:W_2))$. For instance, the bitwise AND of a uin1 and an int5 would return a uint1. However, for simplicity it has been defined to be consistent with the other two logical operators. Regardless of how the operators are defined, synthesis will reduce it to the smallest size that preserves the arithmetic value of the result.

Table 2-8. Return Types for ac_int Binary Arithmetic and Bitwise Logical Operations

Operator	Return Type: ac_int<W_R,S_R>	
	S_R	**Bit Width: W_R**
+	$S_1 \mid S_2$	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)+1$
-	true	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)+1$
*	$S_1 \mid S_2$	W_1+W_2
/	$S_1 \mid S_2$	W_1+S_2
%	S_1	min(W1, W2+!S2&S1)
&	$S_1 \mid S_2$	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)$
|	$S_1 \mid S_2$	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)$
^	$S_1 \mid S_2$	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)$

Table 2-9 shows the binary (two operand) arithmetic and logical operators for ac_fixed and the return type based on the signedness, bit width and integer bit width of the operands. All operands are defined consistently with ac_int: if both ac_fixed operands are pure integers (W and I are the same) then the result is an ac_fixed that is also a pure integer with the same bitwidth and value as the result of the equivalent ac_int operation. For example: a/b where a is an ac_fixed<8,8> and b is an ac_fixed<5,5> returns an ac_fixed<8,8>. In SystemC, on the other hand, the result of a/b returns 64 bits of precision (or SC_FXDIV_WL if defined).

Table 2-9. Return Types for ac_fixed Binary Arithmetic and Bitwise Logical Operations

Operator	Return Type: ac_fixed<W_R,I_R,S_R,AC_TRN,AC_WRAP>		
	S_R	Bit Width: W_R	Integer Bit Width: I_R
+	$S_1 \mid S_2$	$I_R+\max(W_1-I_1,W_2-I_2)$	$\max(I_1+!S_1\&S_2,I_2+!S_2\&S_1)+1$
-	true	$I_R+\max(W_1-I_1,W_2-I_2)$	$\max(I_1+!S_1\&S_2,I_2+!S_2\&S_1)+1$
*	$S_1 \mid S_2$	W_1+W_2	I_1+I_2
/	$S_1 \mid S_2$	$W_1+\max(W_2-I_2,0)+S_2$	$I_1+(W_2-I_2)+S_2$
&	$S_1 \mid S_2$	$I_R+\max(W_1-I_1,W_2-I_2)$	$\max(I_1+!S_1\&S_2,I_2+!S_2\&S_1)$
|	$S_1 \mid S_2$	$I_R+\max(W_1-I_1,W_2-I_2)$	$\max(I_1+!S_1\&S_2,I_2+!S_2\&S_1)$
^	$S_1 \mid S_2$	$I_R+\max(W_1-I_1,W_2-I_2)$	$\max(I_1+!S_1\&S_2,I_2+!S_2\&S_1)$

The assignment operators +=, -=, *=, /=, %=, &=, |= and ^= have the usual semantics:

```
A1 @= A2
```

where @ is any of the operators +, -, *, /, %, &, | and ^ is equivalent in behavior to:

```
A1 = A1 @ A2
```

From a simulation speed point of view, the assignment version (for instance *=) is more efficient since the target precision can be taken into account to reduce the computation required.

Mixed ac_int, ac_fixed and C Integer Operators

Binary (two operand) operations that mix ac_int, ac_fixed and native C integer operands are defined to avoid ambiguity in the semantics or compilation problems due to multiple operators matching an operation. For example, assuming x is an ac_int, 1+x gives the same result as x+1. The return type is determined by the following rules where c_int is a native C type, width(c_int) is the bitwidth of the C type, and signedness(c_int) is the signedness of the C type:

- If one of the operands is an ac_fixed in a binary operation or the first operand is an ac_fixed in an assign operation, the other operand is represented as an ac_fixed:
  - ac_int<W,S> gets represented as ac_fixed<W,W,S>
  - c_int gets represented as ac_fixed<width(c_int), width(c_int), signedness(c_int)>
- Otherwise, if one of the operands is an ac_int in a binary operation or the first operand is an ac_int in an assign operation, the other operand (native c integer) gets represented as ac_int<width(c_int), signedness(c_int)>

The rules above guarantee that precision is not lost. Note that floating point types are not supported for the operators in this section as the output precision can not be determined by the C compiler. Table 2-10 shows a few examples of mixed operations.

Table 2-10. Mixed Expressions: i_s7 is ac_int<7,true>, fx_s20_4 is ac_fixed<20,4,false> and c_s8 is signed char

Expression	Equivalent Expression
1 + i_s7	(ac_int<32,true>) 1 + i_s7
(bool) 1 + i_s7	(ac_int<1,false>) 1 + i_s7
i_7s + fx_u20_4	(ac_fixed<7,7,true>) i_s7 + fx_u20_4
fx_u20_4 += c_s8	fx_u20_4 += (ac_fixed<8,8,true>) c_s8
c_s8 += fx_u20_4	c_s8 += (signed char) fx_u20_4

Mixed ac_int and C pointer for + and - Operators

The operator + is defined for ac_int and C pointer (and vice versa) so that an ac_int can be added to a C pointer. The operator - is defined so that an ac_int can be subtracted from a C pointer. The result is, in all cases, of the same type as the C pointer.

Relational Operators

Relational operators !=, ==, >, >=, < and <= are also binary operations and have some of the same characteristics described for arithmetic and logical operations: the operations are done arithmetically and mixed ac_int, ac_fixed and native C integer operators are defined. The return type is bool.

The relational operator for ac_int and ac_fixed with the C floating type double is also defined for convenience, though for simulation performance reasons it is best to store the double constant in an appropriate ac_int or ac_fixed variable outside computation loops so that the overhead of converting the double to ac_fixed or ac_int is minimized.

Shift Operators

Left shift "<<" and right shift ">>" operators return a value of type of the first operand. The left shift operator shifts in zeros. The right shift operator shifts in the MSB bit for ac_int/ac_fixed of type signed, 0 for ac_int/ac_fixed integers of type unsigned.

If the shift value is negative the first operand is shifted in the opposite direction by the absolute value of the shift value (this is also the semantic of sc_fixed/sc_ufixed shifts). Shift values that are greater than W (bitwidth of first operand) are equivalent to shifting by W.

The second operand is an ac_int integer of bit width less or equal to 32 bits or a signed or unsigned int.

The shift assign operators "<<=" and ">>=" have the usual semantics:

```
A1 <<= A2; // equiv to A1 = A1 << A2
A1 >>= A2; // equiv to A1 = A1 >> A2
```

Because the return type is the type of the first operand, the shift assign operators do not carry out any quantization or overflow.

Mixed ac_int, ac_fixed and C Integer

All shift operators are defined for mixed ac_int, ac_fixed (first operand) and native C integer operands. For example:

```
(short int) x << (ac_int<8,true>) y
```

matches the overloaded operator "<<" that is implemented as follows:

```
(ac_int<16,false>) x << (ac_int<8,true>) y
```

The shift assign operators <<= and >>= are also defined for mixed ac_int (first or second operand), ac_fixed (first operand) and native C integer (second operand).

Differences with SystemC sc_bigint/sc_biguint Types

- The return type of the left shift for sc_bigint/sc_biguint or sc_fixed/sc_ufixed does not lose bits making the return type of the left shift data dependent (dependent on the shift value). Shift assigns for sc_fixed/sc_ufixed may result in quantization or overflow (depending on the mode of the first operand).
- Negative shifts are equivalent to a zero shift value for sc_bigint/sc_biguint

Differences with Native C Integer Types

- Shifting occurs on either 32-bit (int, unsigned int) or 64-bit (long long, unsigned long long) integrals. If the first operand is an integral type that has less than 32 bits (bool, (un)signed char, short) it is first promoted to int. The return type is the type of the first argument after integer promotion (if applicable).
- Shift values are constrained according to the length of the type of the promoted first operand.
  - o $0 \leq s < 32$ for 32-bit numbers
  - o $0 \leq s < 64$ for 64-bit numbers
- The behavior for shift values outside the allowed ranges is not specified by the C++ ISO standard.

Unary Operators: +, -, ~ and !

Unary “+” and “-“ have the usual semantics: “+x” returns x, “-x” returns “0-x”.

The unary operator “~x” returns the arithmetic one’s complement of “x”. The one’s complement is mathematically defined for integers as -x-1 (that is -x+x == -1). This is equivalent to the bitwise complement of x of a signed representation of x (if x is unsigned, add one bit to represent it as a signed number). The return type is signed and has the bitwidth of x if x is signed and bitwidth(x)+1 if x is unsigned.

The ! operator return true if the ac_int/ac_fixed is zero, false otherwise.

Table 2-11 lists the unary operators and their return types.

Table 2-11. Unary Operators for ac_int<W,S>

Operator	Return Type
+	ac_int<W, S>
-	ac_int<W+1, true>
~	ac_int<W+!S, true>
!	bool

Table 2-12 lists the unary operators for ac_fixed and their return types.

Table 2-12. Unary Operators for ac_fixed<W,I,S,Q,O>

Operator	Return Type
+	ac_fixed<W, I, S>
-	ac_int<W+1, I+1, true>
~	ac_int<W+!S, I+!S, true>
!	bool

Increment and Decrement Operators

Pre/Post increment/decrement for ac_int have the usual semantics as shown in Table 2-13 (T_x is the type of variable x).

Table 2-13. Pre- and Post-Increment/Decrement Operators for ac_int

Operator	Equivalent Behavior
x++	T_x t = x; x += 1; return t;
++x	x += 1; return reference to x;

Table 2-13. Pre- and Post-Increment/Decrement Operators for ac_int (cont.)

Operator	Equivalent Behavior
x--	T_x t = x; x -= 1; return t;
--x	x -= 1; return reference to x;

Pre/Post increment/decrement for ac_fixed have the semantics as shown in Table 2-14 (T_x is the type of variable x) where q is the quantum value of the representation (the smallest difference between two values for T_x). This definition is consistent with the definition of ac_int where q is 1.

Table 2-14. Pre- and Post-Increment/Decrement Operators for ac_fixed<W,I,S,Q,O> where $q = 2^{I - W}$.

Operator	Equivalent Behavior
x++	T_x t = x; x += q; return t;
++x	x += q; return reference to x;
x--	T_x t = x; x -= q; return t;
--x	x -= q; return reference to x;

Conversion Operators to C Integer Types

A limited number of conversion operators to C integer types (including bool) are provided by the ac_int datatype, as described in the following list. The ac_fixed datatype provides no conversion operator to C integer types.

- ac_int<W,S> for W > 64 has no conversion operators to any C integer type
- ac_int<W,true> for W <= 64 has only the "long long" conversion operator
- ac_int<W,false> for W <= 64 has only the "unsigned long long" conversion operator

Some coding styles may encounter compilation problems due to the lack of conversion operators. The most common problem is the absence of the conversion to bool for bit widths beyond 64 for ac_int and for all bit widths for ac_fixed. Table 25 shows some typical scenarios:

Table 2-15. Conversion to C Integer Types

ac_int<33,true> k = ...;	
if(k)	OK, conversion first to long long then to bool
if((bool) k)	OK, same as above
switch(k)	OK, operator to long long

Table 2-15. Conversion to C Integer Types

switch(2*k)	ERROR: Result of expression is ac_int<65,true> (constant 2 treated as ac_int<32,true>) No conversion to any C integer from ac_int<65,true>
switch(2*(int)k)	OK, conversion first to long long, then to int
a[k]	OK, operator to long long
a[2*k]	ERROR: Result of expression is ac_int<65,true> (constant 2 treated as ac_int<32,true>) No conversion to any C integer from ac_int<65,true>
a[2*(int)k]	OK, conversion first to long long, then to int
ac_int<80, true> k = ...;	
if(k);	ERROR, no conversion operator defined
if((bool) k)	ERROR, same as above
if(!! k)	OK, operator ! defined, !! equiv to to_bool()
if(k != 0)	OK, operator != defined, equiv to to_bool()
if(k.to_bool())	OK, explicit method defined
switch(k)	ERROR: No conversion operator defined
a[k]	ERROR: No conversion to any C integer from ac_int<80,true>
ac_fixed<3, 3,true> x = ...;	
if(x)	ERROR: No conversion operator defined for any W
if(!! x)	OK, operator ! defined, !! equiv to to_bool()
if(x != 0)	OK, operator != defined, equiv to to_bool()
if(k.to_bool())	OK, explicit method defined

When writing parameterized IP where the bit-widths of some ac_int is parameterized, code that may compile for some parameters, may not compile for a different set of parameters. In such cases, it is important to not rely on the conversion operator.

Explicit Conversion Methods

Methods to covert to C signed and unsigned integer types int, long and Slong are provided for both ac_int and ac_fixed as shown in Table 2-16. The methods **to_int()**, **to_long()**, **to_int64()**, **to_uint()**, **to_uint64()** and **to_ulong()** are defined for both ac_int and ac_fixed (same functions are also defined for sc_bigint/sc_biguint). The method **to_double()** is also defined for both ac_int and ac_fixed. The method **to_ac_int()** is defined for ac_fixed.

Table 2-16. Explicit Conversion Methods for ac_int/ac_fixed

Method	Types	Return Type
to_int()	ac_int/ac_fixed	int
to_uint()	ac_int/ac_fixed	unsigned int
to_long()	ac_int/ac_fixed	long
to_ulong()	ac_int/ac_fixed	unsigned long
to_int64()	ac_int/ac_fixed	Slong
to_uint64()	ac_int/ac_fixed	Ulong
to_double()	ac_int/ac_fixed	double
to_ac_int()	ac_fixed only	ac_int<max(I,1), S>

Bit Select Operator: []

Bit select is accomplished with the [] operator:

```
y[k] = x[i];
```

The `[]` operator does not return an ac_int, but rather it returns an object of class ac_int::ac_bitref that stores the index and a reference to the ac_int object that is being indexed.

The conversion function to “bool” (operator bool) is defined so that a bit reference may be used where a bool type is required:

```
while( y[k] && z[m] )  {}
z = y[k] ? a : b;
```

A bit reference may be assigned an integer. The behavior is that the least significant bit of the integer is assigned to the bit reference. For example if n is type int and x is type ac_int then the following three assignments have the same behavior:

```
x[k] = n;
x[k] = (ac_int<1,false>) n;
x[k] = 1 & n;
```

The conversion to any ac_int is provided and it equivalent to first converting to a bool or to a ac_int<1,false>:

```
ac_int<5,false> x = y[0]; // equivalent to x = (bool) y[0]
```

The ac_bitref::operator=(int val) returns the bit reference so that assignment chains work as expected:

```
x[k] = z[m] = true;  // assigns 1 to z[m] and to x[k]
```

Out of Bounds Behavior

It is invalid to access (read or write) a bit outside the range [0, W-1] where W is the width of the ac_int being accessed. Simulation will assert on such cases. See also "User Defined Asserts" on page 55.

Slice Read Method: slc

Slice read is accomplished with the template method slc<W>(int lsb):

```
x = y.slc<2>(5);
```

which is equivalent to the VHDL behavior:

```
x := y(6 downto 5);
```

The two arguments to the slc method are defined as:

- The bit length of slice W: this is template argument (the length of the slice is constrained to be static so that the length of the slice is known at compile time. The length of the slice must be greater or equal to 1.
- The bit position of the LSB of the slice slc_lsb.

The slc method returns an ac_int of length W and signedness of the ac_int being sliced.

Out of Bounds Slice Reads

Accessing a bit to the left of the MSB of the ac_int<W,S> (index ≥ W) is allowed and is defined as if the ac_int had been first extended (sign extension for signed, 0 padding for unsigned) so that the index is within range. This is consistent with treating ac_int as an arithmetic value.

Attempting to access (read) a bit with a negative index has undefined behavior and is considered to be the product of an erroneous program. If such a negative index read is encountered during execution (simulation) an assert will be triggered. See also "User Defined Asserts" on page 55.

Differences with SystemC sc_bigint/sc_biguint Types

The range method and the part select operator in SystemC are fundamentally different than the ac_int slc and set_slc methods in that it allows dynamic length ranges to be specified.

Slice Write Method: set_slc

Slices are written with the method:

```
    set_slc(int lsb, const ac_int<W,S> &slc)
```

where lsb is the index of the LSB of the slice been written and slc is ac_int slice that is assigned to the ac_int:

```
    x.set_slc(7, y);
```

Out of Bounds Slice Writes

Attempting to assign to a bit that is outside of the range [0, W-1] of the ac_int<W,S> object constitutes an out of bound write. Such a write is regarded as undefined behavior and is the product of an erroneous program. If such an write is encountered during execution (simulation) an assert will be triggered. See also "User Defined Asserts" on page 55.

Differences with Built-in C Integral Types

Accessing a bit or a slice of a C integral type is done by a combination of shift and bit masking. Writing a bit or a slice of a C integral type is done with a combination of shift and bitwise operations.

The set_val Method

The set_val<ac_special_val>() method sets the ac_int or ac_fixed to one of a set of "special values" specified by the template parameter as shown in Table 2-17. Direct assignment of the enumeration values should not be used since it will assign the integer value of the enumeration to the ac_int or ac_fixed.

Table 2-17. Special values

ac_special_val enum	**Value for ac_int/ac_fixed**
AC_VAL_DC	Used mainly to un-initialized variables that are already initialized (by constructor or by being static). Used for validating that algorithm does not depend on initial value. Synthesis can treat it as a dont_care value.
AC_VAL_0	0
AC_VAL_MIN	Minimum value as specified in Table 2-1 on page 11.
AC_VAL_MAX	Maximum value as specified in Table 2-1.
AC_VAL_QUANTUM	Quantum value as specified in Table 2-1.

Constructors

Constructors from all C-types are provided. Constructors from ac_int are also provided. The default constructor does nothing, so non-static variables of type ac_int or ac_fixed will not be initialized by default.

Constructors from char *, have not been defined/implemented in the current release.

IO Methods

Methods to performing IO have not been defined/implemented for the current release.

Mixing ac_int and ac_fixed with Other Datatypes

Refer to "Mixing Datatypes" on page 49 for information on how to interface ac_int and ac_fixed to other data types.

Advanced Utility Functions, Typedefs, etc.

The AC datatypes provide additional utilities such as functions and typedefs. Some of them are available in the ac namespace (ac::), and some of them are available in the scope of the ac datatype itself. The following utility functions/structs/typedefs/static members are described in this section:

- Static members to capture basic parameter information.
- Function for initializing arrays of supported types to a special value.
- Template structs that provide a mechanism to *statically* compute log2 related functions.
- Typedefs for finding the return type of unary and binary operators.

Accessing Parameter Information

It is often useful to be able to access the value of various template parameters for the datatypes. In some cases it is useful to access the width of type T, where T could be either an *ac_int* or an *ac_fixed*. In this case T::width would provide that information. The various parameters that can be accessed are shown in Table 2-18.

Table 2-18. Basic Parameters.

Static member	Description for ac_int	Description for ac_fixed
width	Value of W template parameter	Value of W template parameter

Table 2-18. Basic Parameters.

Static member	Description for ac_int	Description for ac_fixed
i_width	Value of W template parameter	Value of I template parameter
sign	Value of S template parameter	Value of S template parameter
q_mode	AC_TRN	Value of Q template parameter
o_mode	AC_WRAP	Value of O template parameter

Note that for generality all the static members are defined for *ac_int* even in the cases where there is no corresponding template parameter involved as they do capture the numerical behavior of *ac_int*.

Using ac::init_array for Initializing Arrays

The utility function "`ac::init_array`" is provided to facilitate the initialization of arrays to zero, or un-initialization (initialization to dont_care). The most common usage is to un-initialize an array that is declared static as shown in the following example:

```
void foo( ... ) {
   static int b[200];
   static bool b_dummy = ac::init_array<AC_VAL_DC>(b,200);
   ...
}
```

The variable b_dummy is declared static so that the initialization of array b to dont_care occurs only once rather than every time the function *foo* is invoked. The return value of ac::init_array is always "true", but in reality only the side effect to array b is of interest. A similar example to initialize an array to zero that is not declared static would look like:

```
void foo( ... ) {
   int b[200];
   ac::init_array<AC_VAL_0>(b, 200);
   ...
}
```

The function ac::init_array does not check for array bound violations. The template argument to ac::init_array is an enumeration that can be any of the following values: AC_VAL_0, AC_VAL_DC, AC_VAL_MIN, AC_VAL_MAX or AC_VAL_QUANTUM (see Table 2-17 for details). The function is defined for the integer and fixed point datatypes shown in Table 2-19:

Table 2-19. Required Include Files for ac::init_array Function

Type	Required include file
C integer types	ac_int.h
ac_int, ac_fixed	No additional include

Table 2-19. Required Include Files for ac::init_array Function

Type	Required include file
Supported SystemC types	ac_int.h, ac_sc.h

The first argument is of type pointer to one of the types in Table 2-19. Arrays of any dimension may be initialized using ac::init_array by casting it or taking the address of the first element:

```
static int b[200][200];
static bool b_dummy = ac::init_array<AC_VAL_DC>((int*) b, 200*200);
```

or by taking the address of the first element:

```
static int b[200][200];
static bool b_dummy = ac::init_array<AC_VAL_DC>(&b[0][0], 200*200);
```

The second argument is the number of elements to be initialized. For example:

```
int b[200];  ac::init_array<AC_VAL_0>(b+50, 100);
```

initializes elements b[50] to b[149] to 0.

Other ac::init_array Examples:

```
// Using ac::init_array inside a constructor
class X {
   sc_int<5> a[10][32][5][7];
public:
   X() { ac::init_array<AC_VAL_DC>(&a[0][0][0][0], 10*32*5*7); }
   ...
};

// Will be inlined with initialization loop: b+i, 100+k are not constants
   int b[200];  ac::init_array<AC_VAL_0>(b+i, 100+k);

// Will be inlined with initialization loop: mult(n1,n2) not recognized as
// a constant at inlining time
   const int n1 = 40;
   const int n2 = 5;
   int a[n1][n2];
   ac::init_array(&a[0][0], mult(n1,n2));

// Uninitialize two ranges of an array
   static int b[2][100];
   static bool b_dummy = ac::init_array<AC_VAL_DC>(&b[0][0], 50);
   static bool b_dummy2 = ac::init_array<AC_VAL_DC>(&b[1][0], 50);

// Alternative to Uninitialize two ranges of an array
   static int b[2][100];
   static bool b_dummy = ac::init_array<AC_VAL_DC>(&b[0][0], 50) &
                         ac::init_array<AC_VAL_DC>(&b[1][0], 50);
```

Static Computation of log2 Functions

It is statically compute the functions *ceil(log2(x))*, *floor(log2(x))* and *nbits(x)* where x is an unsigned integer. The *nbits(x)* function is the minimum number of bits for an unsigned ac_int to represent the value x.

Static computation of these functions is often useful where x is an integer template parameter and the result is meant to be used as a template value (thus it needs to be statically be determined). For example, lets assume that we have a template class:

```
template<int Size, typename T>
class circular_buffer {
   T _buf[Size];
   ac_int< ac::log2_ceil<Size>::val, false> _buf_index;
};
```

for a circular buffer. The minimum bitwidth of the index variable into the buffer is *ceil(log2(Size))* where *Size* is the size of the buffer.

The computation of the log2 functions is accomplished using a recursive template class. For the user it suffices to know the syntax on how to retrieve the desired value as shown in Table 2-20.

Table 2-20. Syntax for log2 functions

Function	Syntax
ceil(log2(x))	ac::log2_ceil<x>::val
floor(log2(x))	ac::log2_floor<x>::val
nbits(x)	ac::nbits<x>::val

It is important to note that x needs to be a statically determinable constant (constant or template argument).

Return Type for Unary and Binary Operators

It is often useful to find out the return type of an operator. For example, let’s assume the following scenario: assume that we have:

```
Ta a = ...;
Tb b = ...;
Tc c = ...;
T res = a * b + c;
```

what should the type T be such that there is no loss of precision?

This section provides the mechanism to find *T* in terms of *Ta*, *Tb* and *Tc* provided they are AC Datatypes. In addition to return types for binary operations, the return type for unary operators (though the actual operators are not all provided) such as the magnitude (or absolute value), the square, negation is also provided. It is also possible to find out the type required to hold the summation of a set of *N* values of an algorithmic datatype.

The unary operators are listing in Table 2-21 (summation is not really an unary operator, but it depends on a single type).

Table 2-21. Return types for operator on T.

operator on type T	Return type
neg	T::rt_unary::neg
mag	T::rt_unary::mag
mag_sqr	T::rt_unary::mag_sqr
summation of N elements	T::rt_unary::set<N>::sum

The binary operators are shown in Table 2-22.

Table 2-22. Return type for (T1(op1) op T2(op2))

operator on types T1, T2	Return Type
op1 * op2	T1::rt_T<T2>::mult
op1 + op2	T1::rt_T<T2>::plus
op1 - op2	T1::rt_T<T2>::minus
op1 / op2	T1::rt_T<T2>::div
op1 (&, |, ^) op2	T1::rt_T<T2::logic
op2 - op1	T1::rt_T<T2>::minus2
op2 - op1	T1::rt_T<T2>::div2

The last two rows in Table 2-22 are mostly there as helper functions to build up the infrastructure and are not in general needed. For example if both *T1* and *T2* are AC datatypes then instead of using *T1::rt_T<T2>::minus2*, *T2::rt_T<T1>::minus* could be used. These versions are only there for non-commutative operators.

Returning to the mult_add example, the type *T* would be expressed as:

```
    typedef typename Ta::template rt_T<Tb>::mult a_mult_b;
    typename a_mult_b::template rt_T<Tc>::plus res = a * b + c;
```

where the keywords *typename* and *template* are used when *Ta*, *Tb* and *Tc* are template arguments (*dependent-name lookup)*. In this case getting to the type was done in two steps by first defining the type *a_mult_b*. In the following version, the it is done in one step so that it can be used directly as the return type of the templatized function *mult_add*:

```
  template<typename Ta, typename Tb, typename Tc>
```

```
typename Ta::template rt_T<Tb>::mult::template rt_T<Tc>::plus mult_add(Ta
a, Tb b, Tc c) {
   typename Ta::template rt_T<Tb>::mult::template rt_T<Tc>::plus res = a *
b + c;
   return res;
}
```

Note that additional *template* keywords are used because the lookup of *rt_T* is a *dependent-name lookup*, that is the parser does not know that *Ta::rt_T* is a templatized class until it knows the type *Ta* (this happens only once the function *mult_add* is instantiated).

An example of the use of the type for summation is given below:

```
   template <int N, typename T>
   typename T::rt_unary::template set<N>::sum accumulate(T arr[N]) {
      typename T::rt_unary::template set<N>::sum acc = 0;
      for(int i=0; i < N; i++)
         acc += arr[i];
      return acc;
}
```

Chapter 3
Complex Datatype

Introduction

The algorithmic datatype *ac_complex* is a templatized class for representing complex numbers. The template argument defines the type of the real and imaginary numbers and can be any of the following:

- Algorithmic C integer type: *ac_int<W,S>*
- Algorithmic C fixed-point type: *ac_fixed<W,I,S,Q,O>*
- Native C integer types: bool, (*un*)*signed char*, *short*, *int*, *long* and *long long*
- Native C floating-point types: *float* and *double*

For example, the code:

```
ac_complex<ac_fixed<16,8,true> > x (2.0, -3.0);
```

declares the variable x of type *ac_complex* based on *ac_fixed<16,8,true>* and initializes it to have a real part of 2.0 and imaginary part of -3.0 (note: the space between the two '>' is required by C++).

An important feature of the ac_complex type is that operators return the types according to the rules of the underlying type. For example, operators on ac_complex types based on ac_int and ac_fixed will return results for the operators '+', '-' and '*' with no loss of precision ('/' will follow the rules for ac_int and ac_fixed). Likewise, operators on ac_complex types based on native C integer and floating-point types will return results according to the C rules for arithmetic promotion and conversion.

A second important feature of the ac_complex type is that binary operators are defined for ac_complex types that are based on different types, provided the underlying types have the necessary operators defined. For instance to implement complex multiplication, it is necessary to have addition and multiplication defined for the underlying types. As the examples below illustrate, the only issue is combining native floating-point types (float and double) with algorithmic types:

```
ac_complex<ac_int<5,true> > i(2, 1);
ac_complex<ac_fixed<8,3,false> f(1, 5);
ac_complex<unsigned short> s(1, 0);
ac_complex<double> d(3.5, 3.14);

i * f; // OK: ac_int and ac_fixed can be mixed
s * f; // OK: native int type can be mixed with ac_fixed
```

```
i * s; // OK: ac_int and native int type can be mixed
s * d; // OK: native int type can be mixed with native floating-point type
i * d; // ERROR: ac_int and native floating-point types don’t mix
i == d; // ERROR: ac_int/double comparison operators is not defined
f * d; // ERROR: ac_fixed/double + and * operators are not defined
f == d; // OK: ac_fixed/double comparison operators is defined
```

Operators for multiplying a variable of type ac_complex by a real number also are defined with the same restrictions as outlined above. For example:

```
ac_complex<ac_int<5,true> > i(2, 1);
ac_complex<ac_fixed<8,3,false> f(1, 5);
ac_fixed<8,3,false> f_r = 3;
unsigned short s_r = 5;
double d_r = 3.5

i * f_r; // OK: ac_int and ac_fixed can be mixed
s_r * i; // OK: native int type can be mixed with ac_int
i * d_r; // ERROR: ac_int/double + and * operators are not defined
i * 0.1; // ERROR: ac_int/float + and * operators are not defined
i == d_r; // ERROR: ac_int/double comparison operator is not defined
f == d_r; // OK: ac_fixed/double comparison operator is defined
i == 0.1; // ERROR: ac_int/float comparison operators is not defined
f == 0.1; // OK: ac_fxed/double comparison operator is defined
```

Table 3-1 shows the operators defined for both ac_complex.

Table 3-1. Operators defined for ac_complex.

Operators	ac_complex
Two operand +, -, *, /,	Arithmetic result. First or second arg may be C *INT* or *ac_fixed*
=	assignment
+=, -=, *=, /=	Equiv to op then assign. First arg is *ac_complex*
==, !=	First or second arg may be C *INT*, *ac_int*, *ac_fixed* or *C double* (comparison of *ac_int* with *double/float* not defined)
Unary +, -	Arithmetic
! x	Equiv to x == 0

Table 3-2 shows the methods defined for the ac_complex type.

Table 3-2. Methods defined for ac_complex<T>.

Methods	ac_complex
r(), real()	return real part (const T&, or T&)
i(), imag()	return imaginary part (const T& or T&)

Table 3-2. Methods defined for ac_complex<T>.

Methods	ac_complex
set_r(const T2 &r)	assign r to real part
set_i(const T2 &i)	assign i to imaginary part
conj()	complex conjugate
sign_conj()	returns (sign(real), sign(imag))) as an ac_complex<ac_int<2,true> >
mag_sqr()	returns sqr(real) + sqr(imag)
to_string	convert to std::string depending on parameter AC_HEX, AC_DEC, AC_OCT, AC_BIN
type_name()	returns "name" of the type as a std::string

Usage

In order to use the ac_int datatype the following header file must be included in the C++ source:

```
#include <ac_complex.h>
```

Recommendations

1. Do not use native C type *unsigned* (*unsigned int*) as the return type (and the arithmetic) is defined according to the promotion/arithmetic rules of the C language. That is the resulting complex type will based on the type *unsigned*. For example:

   ```
   ac_complex<unsigned> x(0,1);
   cout << x*x; // result is (2^32 - 1, 0)
   ```

2. Pay special attention on the return type when performing division. For example, if two ac_complex based on native C type int are divided, the result will be an ac_complex based on int and truncation will take place.

Advanced utility functions, typedefs, etc

The AC datatypes provide additional utilities such as functions and typedefs. Some of them are available in the ac namespace (ac::), and some of them are available in the scope of the ac datatype itself. The following utility functions/structs/typedefs are described in this section:

- Typedef to capture the underlying type.
- Function for initializing arrays of ac_complex to a special value.
- Typedefs for finding the return type of unary and binary operators.

Accessing the Underlying (Element) Type

The type of the real and imaginary elements can be accessed as

T::element_type

where *T* is the ac_complex type.

Using ac::init_array for Initializing Arrays

The utility function "`ac::init_array`" is provided to facilitate the initialization of arrays to zero, or un-initialization (initialization to dont_care). For more details about the basic AC Datatypes, refer to the examples in "Arbitrary-Length Bit-Accurate Integer and Fixed-Point Datatypes" on page 11. The initialization value is applied to both the real and imaginary components.

Return Type for Unary and Binary Operators

1. Refer to corresponding sections in "Operators and Methods" on page 18 for the basic AC Datatypes.

Chapter 4
Datatype Migration Guide

Introduction

This Chapter provides detailed explanations on differences between the Algorithmic datatypes and the built-in C integer types and the SystemC integer and fixed-point types.

General Compilation Issues

When porting algorithms written with either C integer or SystemC datatypes a compilation error may be encountered when the choices for the *question mark operator* are ac_int or ac_fixed types. For instance the expression:

```
b ? x : -x;
```

works when *x* is a C integer or a SystemC data type but will error out when *x* is an ac_int or ac_fixed because x and -x don't have the same type (their bitwidths are different). Explicit casting may be needed for the question mark operator so that both choices have the exact same type. For example, in the examples below the expressions in the left (using sc_int) are re-coded with ac_int as follows:

- (c ? a_5s : b_7u) becomes (c ? (ac_int<8,true>) a_5s : (ac_int<8,true>) b_7u)
- (c ? a_5s : - a_5s) becomes (c ? (ac_int<6,true>) a_5s : - a_5s)
- (c ? a_5s : 1) becomes (c ? a_5s : (ac_int<5,true>) 1), or (c ? (int) a_5s : 1)

where variable a_5s is a 5-bit wide signed sc_int or ac_int and so on.

The SystemC datatypes don't require casting because they share the same base class that contains the actual value of the variable. Note that an integer constant such as 1 is of type *int* and will be represented as an ac_int<32, true>, so an expression such as a_4s + 1 will have type ac_int<33,true> instead of ac_int<5,true>.

SystemC Syntax

Table 4-1 shows the SystemC bit-accurate datatypes that ac_int and ac_fixed can replace. Using ac_int and ac_fixed it is possible to write generic algorithms that work for any bitwidth and that simulate faster than the "fast" (but limited) SystemC types sc_int, sc_uint, sc_fixed_fast, sc_ufixed_fast.

Table 4-1. Relation Between SystemC Datatypes and AC Datatypes

SystemC Datatype	New Datatype	Comments
sc_int<W>	ac_int<W,true>	sc_int limited to 64 bits
sc_uint<W>	ac_int<W,false>	sc_uint limited to 64 bits
sc_bigint<W>	ac_int<W,true>	
sc_biguint<W>	ac_int<W,false>	
sc_fixed_fast<W,I,Q,O>	ac_fixed<W,I,true,Q,O>	sc_fixed_fast limited to mantissa of double
sc_ufixed_fast<W,I,Q,O>	ac_fixed<W,I,false,Q,O>	sc_ufixed_fast limited to mantissa of double
sc_fixed<W,I,Q,O>	ac_fixed<W,I,true,Q,O>	
sc_ufixed<W,I,Q,O>	ac_fixed<W,I,false,Q,O>	

The ac_int and ac_fixed types have the same parameters with the same interpretation as the corresponding SystemC type. The difference is an extra boolean parameter S that defines whether the type is signed (S==true) or unsigned (S==false). Using a template parameter instead of different type names makes it easier to write generic algorithms (templatized) that can handle both signed and unsigned types. The other difference is that ac_fixed does not use the “nbits” parameter that is used for the SystemC fixed-point datatypes.

The template parameters Q and O are enumerations of type ac_q_mode and ac_o_mode respectively. All quantization modes are supported as shown in Table 4-2. Most commonly used overflow modes are supported as shown in Table 4-3.

Table 4-2. Quantization Modes for ac_int and Their Relation to sc_fixed/sc_ufixed

ac_fixed	sc_fixed/sc_ufixed
AC_TRN (**default**)	SC_TRN (default)
AC_RND	SC_RND
AC_TRN_ZERO	SC_TRN_ZERO
AC_RND_ZERO	SC_RND_ZERO
AC_RND_INF	SC_RND_INF
AC_RND_MIN_INF	SC_RND_MIN_INF
AC_RND_CONV	SC_RND_CONV

Table 4-3. Overflow Modes for ac_fixed and Their Relation to sc_fixed/sc_ufixed

ac_fixed	**sc_fixed/sc_ufixed**
AC_WRAP (**default**)	SC_WRAP, nbits = 0 (**default**)
AC_SAT	SC_SAT
AC_SAT_ZERO	SC_SAT_ZERO
AC_SAT_SYM	SC_SAT_SYM

All operands are defined consistently with ac_int: if both ac_fixed operands are pure integers (W and I are the same) then the result is an ac_fixed that is also a pure integer with the same bitwidth and value as the result of the equivalent ac_int operation. For example: a/b where a is an ac_fixed<8,8> and b is an ac_fixed<5,5> returns an ac_fixed<8,8>. In SystemC, on the other hand, the result of a/b returns 64 bits of precision (or SC_FXDIV_WL if defined).

SystemC to AC Differences in Methods/Operators

There are methods that have a different name, syntax and semantic. The main one is the range method range(int i, int j) or operator (int i, int j). There are two different methods in ac_int for accessing or modifying (assigning to) a range. Note that ac_int does not support a dynamic length range.

Methods: *range* in SystemC to *slc* and *set_slc* in ac_int or ac_fixed

For accessing a range:

```
   x.range(i+W-1, i) (or x(i+W-1, i)) becomes x.slc<W>(i)
```

where x.slc<W>(i) returns an ac_int<W, S_X> where S_X is the signedness of variable x. The slice method returns an ac_int for both ac_int and ac_fixed. Also note that W must be a constant. For instance x.range(i, j) would translate into x.slc<i-j+1>(j) provided both i and j are constants.

For assigning a range:

```
   x.range(i+W-1,i) = y (or x.(i+W-1,i) = y) becomes x.set_slc(i, y)
```

this assumes that y is of type either ac_int<W, false> or ac_int<W, true>, otherwise it needs to be cast to either type.

Concatenation

The concatenation operator (the “,” operator in sc_int/sc_uint and sc_bigint/sc_biguint) is not defined in ac_int or ac_fixed. The solution is to rewrite it using set_slc:

y = (x, z); becomes y.set_slc(W_Z, x); y.set_slc(0, z);
where W_Z is the width of z.

Other Methods

Table 4-4 shows other less frequently used methods in SystemC datatypes that would require rewriting in ac_int.

Table 4-4. Migration of SystemC Methods to ac_int

SystemC	ac_int
iszero	operator !
sign	x < 0
bit	x[i]
reverse	no equivalent
test	x[i]
set	x[i] = 1
clear	x[i] = 0
invert	x[i] = !x[i]

Constructors from char *, are not defined/implemented for ac_int and ac_fixed.

Support for SystemC sc_trace Methods

The Algorithmic C Datatypes package was updated in 2010a to provide support for using SystemC “sc_trace” methods on the AC datatypes. In order to use the sc_trace method in your SystemC design, you must include the following headers in the following order:

```
#include <systemc.h>
#include <ac_fixed.h>  (or ac_int.h, or ac_complex.h)
#include <ac_sc.h>
```

Failing to include them in the above order will result in compile errors. In addition to proper include file ordering, you can only trace using VCD format files (i.e. using the sc_create_vcd_trace_file() function in SystemC). Using any other trace file format may result in a crash during simulation.

Simulation Differences with SystemC types and with C integers

In this section the simulation semantics of the bit-accurate datatypes sc_int/sc_uint, sc_bigint/sc_biguint and ac_int will be compared and contrasted. For simplicity of discussion the shorthand notation *int<bw>* and *uint<bw>* will be used to denote a signed and unsigned integer of bitwidth bw respectively. Also *Slong* and *Ulong* will be used to denote the C 64-bit integer types *long long* and *unsigned long long* respectively.

The differences between limited and arbitrary length integer datatypes can be group in several categories as follows:

- Limited precision (64 bit) vs. arbitrary precision
- Differences due to implementation deficiencies of sc_int/sc_uint
- Differences due to definition

Limited Precision vs. Arbitrary Precision

Both sc_int/sc_uint use the 64-bit C types *long long* and *unsigned long long* as the underlying type to efficiently their operators. In more mathematical terms, that means that the arithmetic is accurate modulo 2^{64}. As long as every value is representable in 2's complement 64-bit signed, the limited precision should not affect the computation and should agree with the equivalent expression using arbitrary precision integers.

Implementation Deficiencies of sc_int/sc_uint

At first glance, it would appear that the only difference between limited and arbitrary length datatypes is that arithmetic is limited to 64-bit. However, there are a number of additional issues that have to do with how the sc_int/sc_uint are implemented.

The implementations of the limited precision bit-accurate integer types suffer from a number of deficiencies:

- Mixing signed and unsigned can lead to unexpected results. Many operators are not defined so they fall back to the underlying C types Slong and Ulong. Conversion rules in C change the signed operand to unsigned when a binary operation has mixed Slong and Ulong operands. This leads to the following non intuitive results:
  - (uint<8>) 1 / (int<8>) -1 = (Ulong) 1 / (Slong) -1 = (Ulong) 1 / (Ulong) -1 = 0
  - (uint<6>) 1 > (int<6>) -1 = (Ulong) 1 > (Slong) -1 = (Ulong) 1 / (Ulong) -1 = false
  - (int<6>) -1 >> (uint<6>) 1 = (Slong) -1 >> (Ulong) 1 = -1

Note however, that operations such as +, -, and *, |, &, ^ provided the result is assigned to an integer type of length 64 or less, or is used in expressions that are not sensitive to the signedness of the result:

- w_u20 = a_u8 * b_s9 + x_u13 & y_s4; // OK
- sc_bigint<67> i = a_u8 * b_s9 + x_u13 & y_s4; // Bad, assigning Ulong to 67 signed
- w_u20 = (a_u8 * b_s9) / c_s6; // Bad, s/s div should be ok, but numerator is Ulong
- f(a_u8 * b_s9); // Bad if both f(Ulong) and f(Slong) are defined

- Shifting has the same limitations as in C. The C language only defines the behavior of integer shifts on a Slong or Ulong when the shift value is in the range [0, 63]. The behavior outside that range is compiler dependent. Also some compilers (Visual C 6.0 for example) incorrectly convert the shift value from Slong to Ulong if the first operand is Ulong.

Differences Due to Definition

The previous two sections covered the high-level and most often encountered differences among the bit-accurate integer datatypes. This section will cover more detailed differences.

Initialization

The SystemC integer datatypes are initialized by default to 0 by the default constructor, whereas ac_int is not initialized by the default constructor. If the algorithm relies on this behavior, the initialization needs to be done explicitly when migrating from SystemC integer datatypes to ac_int. This issue is not there for fixed-point datatypes as neither the sc_fixed/sc_ufixed nor ac_fixed initializes by default.

Note that non local variables (that is global, namespace, and *static* variables) don't have this issue as they are initialized by virtue of how C/C++ is defined.

Shift Operators

The ac_int and ac_fixed shift operators are described in the section "Shift Operators" on page 22. Shift operations present the most important differences between the Algorithmic C types and the SystemC types.

SystemC Types

- The return type of the left shift for sc_bigint/sc_biguint or sc_fixed/sc_ufixed does not lose bits making the return type of the left shift data dependent (dependent on the shift value). Shift assigns for sc_fixed/sc_ufixed may result in quantization or overflow (depending on the mode of the first operand).
- Negative shifts are equivalent to a zero shift value for sc_bigint/sc_biguint

- The shift operators for the limited precision versions is only defined for shift values in the range [0, 63] (see Section - “Implementation Deficiencies of sc_int/sc_uint”).

Differences with Native C Integer Types

- Shifting occurs on either 32-bit (int, unsigned int) or 64-bit (long long, unsigned long long) integrals. If the first operand is an integral type that has less than 32 bits (bool, (un)signed char, short) it is first promoted to int. The return type is the type of the first argument after integer promotion (if applicable).
- Shift values are constrained according to the length of the type of the promoted first operand.
    - $0 \leq s < 32$ for 32-bit numbers
    - $0 \leq s < 64$ for 64-bit numbers
- The behavior for shift values outside the allowed ranges is not specified by the C++ ISO standard.

The shift left operator of ac_int returns an ac_int of the same type (width and signedness) of the first argument and it is not equivalent to the left shift of sc_int/sc_uint or sc_bigint/sc_biguint. To get the equivalent behavior using ac_int, the first argument must be of wide enough so that is does not overflow. For example

```
(ac_int<1,false>) 1 << 1 = 0
(ac_int<2,false>) 1 << 1 = 2
```

Both the right and left shift operators of ac_int return an ac_fixed of the same type (width, integer width and signedness) of the first argument and is not equivalent to the corresponding operator in sc_fixed/sc_ufixed. Despite the fact that there might be loss of precision when shifting an ac_fixed, no quantization or overflow is performed. The first argument must be large enough width and integer width to guarantee that there is no loss of precision.

Differences for the range/slice Methods

It is legal to access bits to the left of the MSB of an ac_int or an ac_fixed using the slc method. The operation is treated arithmetically (as if the value had been represented in the appropriate number of bits).

The following operations are invalid and will generate a runtime error (assert) during C simulation:

- Attempting to access negative indices with the slc method
- Attempting to access or modify indices outside the 0 to W-1 range for set_slc or the [] operator

The behavior for indices outside the 0 to W-1 range for SystemC datatypes is not consistent. For example sc_int/sc_uint and sc_fixed/sc_ufixed don't allow it (runtime error) while sc_bigint/sc_biguint allow even negative indices (changed to a 0 index).

Conversion Methods

The conversion methods **to_int()**, **to_long()**, **to_int64()**, **to_uint()**, **to_uint64()** and **to_ulong()** for sc_fixed/sc_ufixed are implemented by first converting to double. For instance:

```
sc_fixed<5,3> x = ...;
int t = x.to_int();  // equiv to (int)(double)x
                     // not equiv to (int)(sc_int<32>)x

ac_fixed<5,3,true> y = ...;
int t = y.to_int();  // equiv to x.to_ac_int().to_int();
```

The difference in most cases will be subtle (double has a signed-magnitude representation so it truncates towards zero instead of truncating towards minus infinity) but could be very different if the number would overflow the int or long long C types.

Neither ac_int nor ac_fixed provide a conversion operator to double (an explicit to_double method is provided). SystemC datatypes do provide that conversion. There are a number of cases where that can lead to non intuitive semantics:

```
sc_fixed<7,4> x = ...;
int t = (int) x;  // equiv to (int)(double) x
bool b = !x;      // equiv to ! (double) x
```

Unary Operators ~, - and Binary Operators &, |, ^

The unary operators ~ and - for ac_int and ac_fixed will return a signed typed as shown in Table 2-11 and Table 2-12 on page 24. This behavior is consistent with the SystemC integer types but inconsistent with the SystemC fixed-point types.

A common issue when migrating from C/C++ that uses shifting and masking is the following:

```
unsigned int x = 0;
unsigned mask = ~x >> 24; // mask is 0xFF

ac_int<32,false> x = 0;
ac_int<32,false> mask = ~x >> 24; // mask is 0xFFFFFFFF
```

The reason for this discrepancy is that for C integers the return type for the unary operators ~ and - is the type of the promoted type for the operand. If the argument is *signed/unsigned int*, *long* or *long long*, integer promotion does not change the type. For example, when the operand is *unsigned int*, then the return type of either ~ or - will be *unsigned int*. Note however that unsigned char and unsigned short get promoted to int which makes the behavior consistent with ac_int:

```
unsigned short x = 0;
unsigned short mask = ~x >> 8; // mask is 0xFFFF, not 0xFF

ac_int<16,false> x = 0;
ac_int<16,false> mask = ~x >> 8; // mask is 0xFFFF
```

The arithmetic definition of the operator ~ makes the value result independent of the bit-width of the operand:

```
if x == y, then ~x == ~y // x and y may be different bitwidths
```

Also the arithmetic definition is consistent with the arithmetic definition of the binary (two operand) logical operators &, |, and ^. For instance:

```
~(a | b) == ~a & ~b
```

The arithmetic definition of the logical operators &, |, ^ is necessary since signed and unsigned operands of various bit-widths may be combined.

Mixing Datatypes

This section describes the conversion functions that are used interface between the bit-accurate integer datatypes.

Conversion Between sc_int/sc_uint and ac_int

Use the C integer conversions to go from sc_int/sc_uint to ac_int and vice versa:

```
ac_int<54,true> x = (Slong) y; // y is sc_int<54>
sc_int<43> y = (Slong) x; // x is ac_int<43, true>
```

Conversion Between sc_bigint/sc_biguint and ac_int

The C integer datatypes can be used to convert between integer datatypes without loss of precision provided the bitwidth does not exceed 64 bits:

```
ac_int<54,true> x = y.to_int64(); // y is sc_bigint<54>
sc_bigint<43> y = (Slong) x; // x is ac_int<43,true>
ac_int<20,true> x = y.to_int(); // y is sc_bigint<20>
```

Explicit conversion functions are provided between the datatypes sc_bigint/sc_biguint and ac_int and between sc_fixed/sc_ufixed and ac_fixed. They are provided in a different include file:

```
$MGC_HOME/shared/include/ac_sc.h
```

which define the following functions:

```
template<int W> ac_int<W, true> to_ac(const sc_bigint<W> &val);

template<int W> ac_int<W, false> to_ac(const sc_biguint<W> &val);
```

```
template<int W> sc_bigint<W> to_sc(const ac_int<W, true> &val);

template<int W> sc_biguint<W> to_sc(const ac_int<W, false> &val);

template<int W, int I, sc_q_mode Q, sc_o_mode O, int nbits>
ac_fixed<W, I, true> to_ac(const sc_fixed<W, I, Q, O, nbits> &val);

template<int W, int I, sc_q_mode Q, sc_o_mode O, int nbits>
ac_fixed<W, I, false> to_ac(const sc_ufixed<W, I, Q, O, nbits> &val);

template<int W, int I, ac_q_mode Q, ac_o_mode O>
sc_fixed<W,I> to_sc(const ac_fixed<W, I, false, Q, O> &val);

template<int W, int I, ac_q_mode Q, ac_o_mode O>
sc_ufixed<W,I> to_sc(const ac_fixed<W, I, true, Q, O> &val);
```

For example:

```
sc_bigint<123> x = to_sc(y); // y is ac_int<123, true>
```

Chapter 5
Frequently Asked Questions

Operators ~, &, |, ^, -, !

Why does ~ and - for an unsigned return signed?

See “Unary Operators ~, - and Binary Operators &, |, ^” on page 48.

Why are operators &, |, ^ “arithmetically” defined?

The two operands may have different signedness, have different bit-widths or have non aligned fixed-points (for ac_fixed). An arithmetic definition makes the most sense in this case.

Why does operator ! return different results for ac_fixed and sc_fixed?

The ! operator is not defined for sc_fixed or sc_ufixed. The behavior for sc_fixed is then equivalent to first casting it to double and then applying the ! operator which is not correct.

Conversions to double and Operators with double

Why is the implicit conversion from ac_fixed to double not defined?

The reason that there is no implicit conversion function to double is that it is impractical to define mixed ac_fixed and double operators. For example, if there was an implicit conversion to double the expression “x + 0.1” would be computed as (double) x + 0.1 even when x is has more bits of precision than the double, thus resulting in an unintended loss of precision.

Why are most binary operations not defined for mixed ac_fixed and double arguments?

Consider the expression “x + 0.1” where x is of type ac_fixed. Arithmetic operators such as + are defined in such a way that they return a result that does not loose precision. In order to accomplish that with a mixed fixed-point and double operator +, the double would have to be converted to a fixed-point that is able to represent all values that a double can assume. This would require an impractically large ac_fixed. Note that the actual value of the constant is not used by a C++ compiler to determine the template parameters for the minimum size ac_fixed that can hold it.

Comparison operators are defined for mixed ac_fixed and double arguments as the result is a bool and there is no issue about loosing precision. However, the comparison operator is much less efficient in terms of runtime than using a comparison to the equivalent ac_fixed:

```
while( ... ) {
   if( x > 0.5) // less efficient
   ...
}
```

could be made more efficient by storing the constant in an ac_fixed so that the overhead of converting from double to ac_fixed is incurred once (outside the loop):

```
ac_fixed<1,0,false> c0_5 = 0.5;
while( ... ) {
   if( x > c0_5 ) // more efficient
   ...
}
```

Constructors from strings

Why are constructors from strings not defined?

They would be very runtime inefficient.

Shifting Operators

Why does shifting gives me unexpected results?

The shift operation for ac_int/ac_fixed differs from the shift operations in SystemC and native (built-in) C integers. See Section - “Shift Operators”. The main difference is that the shift operation for ac_int/ac_fixed returns the type of the first operand.

```
ac_int<2,false> x = 1;
x << 1; // returns ac_int<2,false> (2), value is 2
x << 2; // returns ac_int<2,false> (4), value is 0
(ac_int<3,false>) x << 2; // returns ac_int<3,false> (4), value is 4
```

The main reason for this semantic is that for an arbitrary-length type, a definition that returns a fully arithmetic value requires a floating return type which violates the condition that the return type should an ac_int or an ac_fixed type. Supporting a floating return type creates a problem both for simulation speed and synthesis. For example the type of the expression

```
a * ( (x << k) + y)
```

can not be statically determined.

Division Operators

Why does division return different results for ac_fixed and sc_fixed?

Division for sc_fixed/sc_ufixed returns 64 bits of precision (or whatever SC_FXDIV_WL is defined as). The return type for ac_fixed is defined depending on the parameters of both the dividend and divisor (see Table 2-9 on page 21).

Compilation Problems

Why aren’t older compilers supported?

The support of templates is not adequate in older compilers.

Why doesn't the *slc* method compile in some cases?

When using the slc method in a templatized function place the keyword *template* before it as some compilers may error out during parsing. For example:

```
template<int N>
int f(int x) {
   ac_int<N,true> t = x;
   ac_int<6,true> r = t.template slc<N>(4); // t.slc<N>(4) could error out
   return r.to_int();
}
```

Without the keyword template the "t.slc<N>(4)" is parsed as "t.slc < N" since it does not know whether slc is a data member or a method (this is known once template function *f* and therefore ac_int<N,true> is instantiated).

Why do I get compiler errors related to template parameters?

If this happen while using the GCC compiler, the error might be related to the template bug on GCC that was fixed in version 4.0.2 (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=23789). This compiler bug rarely showed up when using previous versions of ac_int/ac_fixed and is even less likely on the current version of ac_int/ac_fixed.

Platform Dependencies

What platforms are supported?

The current implementation assumes that an *int* is 32 bits and that a *long long* is 64-bits, both in 2's complement representation. These assumptions need to be met for correct simulation of the data types. In addition a *long* is assumed to be 32 bits wide, a *short* is assume to be 16 bits and a *char* is assumed to be 8 bits wide. A plain *char* (neither signed or unsigned) is assumed to be signed. These assumptions are only relevant if the types are used to initialize/construct an ac_int or ac_fixed or they are used in expressions with ac_int or ac_fixed.

Purify Reports

Why do I get UMRs for ac_int/ac_fixed in purify?

The following code will report a UMR in purify:

```
ac_int<2,false> x;
x[0] = 0;   // UMR
x[1] = 1;
```

The UMR occurs because *x* is not initialized, but setting a bit (or a slice) requires accessing the original (un-initialized) value of *x*.

A second source of UMRs is explicit calls to un-initialize an ac_int/ac_fixed that were declared static (see Section - “Using ac::init_array for Initializing Arrays”). This is used mostly for algorithms written for hardware design.

User Defined Asserts

Can I control what happens when an assert is triggered?

Control over what happens when a assert is triggered is accomplished by defining the compiler directive AC_USER_DEFINED_ASSERT to a user defined assert function before the inclusion of the AC Datatype header(s). The following example illustrates this:

```
void my_assert(bool condition, const char *file=0, int line=0, const char
*msg=0);
#define AC_USER_DEFINED_ASSERT(cond, file,line,msg)
my_assert(cond,file,line,msg)
#include <ac_int.h>
```

When AC_USER_DEFINED_ASSERT is defined, the system header <ostream> is included instead of <iostream>, as std::cerr is no longer required by ac_assert in that case. This feature was introduced to reduce the application startup time penalty that can occur when including iostream for some compilers that don't support "#pragma once". That startup time penalty is proportional on the number of translation units (static constructor for each *.o file that includes it).

Algorithmic C Datatypes
End-User License Agreement

IMPORTANT – USE OF SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS
CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE

YOU MAY USE AND DISTRIBUTE UNMODIFIED VERSIONS
OF THIS SOFTWARE AS STATED BELOW,
YOU MAY NOT MODIFY THE SOFTWARE

This license is a legal Agreement between you, the end user, either individually or as an authorized representative of a company acquiring the license, and Mentor Graphics Corporation ("Mentor Graphics"). YOUR USE OF THE SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. If you do not agree to these terms and conditions, promptly return or, if received electronically, delete the Software and all accompanying items.

1. **GRANT OF LICENSE. YOU MAY USE AND DISTRIBUTE THE SOFTWARE, BUT YOU MAY NOT MODIFY THE SOFTWARE**. The Software you are installing, downloading, or otherwise acquired, under this Agreement, including source code, binary code, updates, modifications, revisions, copies, or documentation pertaining to Algorithmic C Datatypes (collectively the "Software") is a copyrighted work owned by Mentor Graphics. Mentor Graphics grants to you, a nontransferable, nonexclusive, limited copyright license to use and distribute the Software, but you may not modify the Software. Use of the Software consists solely of reproduction, performance, and display.

2. **RESTRICTIONS**; **NO MODIFICATION.** Modifying the Software is prohibited. Each copy of the Software you create must include all notices and legends embedded in the Software. Modifying the Software means altering, enhancing, editing, deleting portions or creating derivative works of the Software. You may append other code to the Software, so long as the Software is not otherwise modified. Mentor Graphics retains all rights not expressly granted by this Agreement. The terms of this Agreement, including without limitation, the licensing and assignment provisions, shall be binding upon your successors in interest and assigns. The provisions of this section 2 shall survive termination or expiration of this Agreement.

3. **USER COMMENT AND SUGGESTIONS.** You are not obligated to provide Mentor Graphics with comments or suggestions regarding the Software. However, if you do provide to Mentor Graphics comments or suggestions for the modification, correction, improvement or enhancement of (a) the Software or (b) Mentor Graphics products or processes which may embody the Software ("Comments"), you grant to Mentor a non-exclusive, irrevocable, worldwide, royalty-free license to disclose, display, perform, copy, make, have made, use, sublicense, sell, and otherwise dispose of the Comments, and Mentor Graphics' products embodying such Comments, in any manner which Mentor Graphics chooses, without reference to the source.

4. **NO WARRANTY.** MENTOR GRAPHICS EXPRESSLY DISCLAIMS ALL WARRANTY FOR THE SOFTWARE. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE SOFTWARE AND ANY RELATED DOCUMENTATION IS PROVIDED "AS IS" AND WITH ALL FAULTS AND WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR DISTRIBUTION OF THE SOFTWARE REMAINS WITH YOU.

5. **LIMITATION OF LIABILITY.** IN NO EVENT WILL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

6. **LIFE ENDANGERING APPLICATIONS.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OR DISTRIBUTION OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. THE PROVISIONS OF THIS SECTION 6 SHALL SURVIVE TERMINATION OR EXPIRATION OF THIS AGREEMENT.

7. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OR DISTRIBUTION OF SOFTWARE.

8. **TERM AND TERMINATION.** This Agreement terminates immediately if you exceed the scope of the license granted or fail to comply with the provisions of this License Agreement. If you institute patent litigation against Mentor Graphics (including a cross-claim or counterclaim in a lawsuit) alleging that the Software constitutes direct or contributory patent infringement, then any patent licenses granted to you under this License for that Software shall terminate as of the date such litigation is filed. Upon termination or expiration, you agree to cease all use of the Software and delete all copies of the Software.

9. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

10. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. All software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

11. **CONTROLLING LAW AND JURISDICTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Multnomah County, Oregon. This section shall not restrict Mentor Graphics' right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

12. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

13. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. The prevailing party in any legal action regarding the subject matter of this Agreement shall be entitled to recover, in addition to other relief, reasonable attorneys' fees and expenses.