SIEMENS EDA

Algorithmic C (AC) Datatypes Reference Manual

Software Version v5.1.1
May 2025

SIEMENS

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Index of Tables

Table of Contents

Chapter 1: Overview of Algorithmic C Datatypes

This package provides numerical and interface C++ classes aimed at modeling behavior that is targeted to the design of hardware. Modeling of bit-accurate arithmetic is crucial for hardware design. The numerical package provides classes for bit-accurate integer, fixed-point, floating-point and complex numbers. Well-defined semantics and simulation speed are the focus of this package since they are essential for the verification of hardware designs. The package currently includes the following files:

File	Package	Description	Class	AC Dependencies
ac_int.h	Numerical	Integer	ac_int<W,S>	
ac_fixed.h		Fixed-point	ac_fixed<W,I,S,Q,O>	ac_int.h
ac_float.h		Floating-point	ac_float<W,I,E,Q>	ac_fixed.h
ac_complex.h		Complex	ac_complex<T>	ac_float.h
ac_sc.h		AC-SystemC Conversions, Tracing		ac_complex.h
ac_pp.py		Pretty-print for gdb		gdb with python
ac_channel.h	Interface	Channel Fifo	ac_channel<T>	

The parameters for the classes are:

- W: integer representing width of the type. For ac_float it is width of the mantissa.

- S: bool parameter representing signedness of ac_int or ac_fixed type.

- I: integer representing integer width.

- Q,O: enumeration parameter for quantization (rounding) and overflow modes.

- T: type. For ac_complex, T is restricted to AC Numerical types and C++ integer and floating types.

The following sections overview the Numerical and Interface Datatypes.

1.1. Overview of Numerical Algorithmic C Datatypes

The advantages of the Algorithmic C numerical 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. Whenever possible, the return type of operators do not loose any precision. In cases where that is not the case, for example in fixed-point division, what is kept is fully defined and implementation independent.

- *Precise, Uniform and Consistent 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. Mixed type operators, including mixed operators with C integer types are fully defined to prevent any ambiguity. Also, asserts have been introduced to catch invalid code during simulation. See also "User Defined Asserts".

- *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. In addition the compilation is faster and produces smaller binary executables.

- *Correctness*: The simulation and synthesis semantics have been verified for many size combinations using a combination of simulation and equivalence checking.

1.1.1. Usage of Numerical AC Datatypes

In order to use the Algorithmic C data types, the appropriate header needs to be included in the source. All the definitions are in the AC header 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.

1.1.2. Usage of Numerical AC Datatypes within SystemC

Numerical AC Datatypes can be used in SystemC descriptions. If they are used as types for SystemC sc_signal, then the header file ac_sc.h needs be included after the systemc.h is included. The ac_sc.h file includes all the other numerical types by including ac_complex.h.

The ac_sc.h header file also provides explicit conversion functions to SystemC integer and fixed-point datatypes. The "Datatype Migration Guide" chapter presents information about how to convert algorithms written with SystemC datatypes to Algorithmic C datatypes.

1.1.3. Definition and Implementation Overview

The numerical 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 numerical ac types and C integer type Binary Functions:* all binary operators are defined for mixed *ac_int*, *ac_fixed, ac_float and ac_complex* 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.

- Uninitialized by Default: there is no default initialization. This is preferable for both runtime and synthesis where uninitialized variables are treated as don't care. There is a utility function to uninitialize static arrays to undo the initialization due to the "static" qualifier.

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.

1.1.4. Implementation Guidelines

The following guidelines are followed in the implementation:

1. Minimal inclusion of other header files. For example systemc.h should only be included in ac_sc.h. No inclusion of std libraries, boost etc.

2. No C++11 features or dependencies.

3. No memory allocation is used. Most code is written to be synthesizable. The exceptions are related to non-synthesizable types such as float and double.

4. No virtual functions used. Given that the types are templatized and fully defined in the header files, there is no reason for their use.

5. Warnings that need to be turned off are done so only for the header context using the push/pop mechanisms available in Visual C++ and GCC version 4.6 or higher.

1.1.5. Implementation Assumptions

While the C++ standard does not formally define the bit width and representation of the various integer types, compilers for general software have converged to both widths and representation that are assumed in this package. The most important assumption is that the int type is represented in a 32-bit, 2's complement representation. The width of other integer types are relevant when converting to and from those types and in the definition of mixed AC and C++ integer types. The assumptions are as follows:

1. Two's complement is used for all signed integer versions.

2. The char, signed char and unsigned char have a bitwidth of 8. The type char is treated as signed.

3. The short and unsigned short have a bitwidth of 16 and is represented in 2's complement.

4. The long and unsigned long have a bitwidth as defined in std::numeric_limits. In general 32-bit platforms represent them as 32-bit integers and 64-bit platforms represent them as 64-bit integers.

5. The long long and unsigned long long have a bitwidth of 64-bits.

1.2. Overview of Interface Algorithmic C Datatypes

In addition to the numerical datatypes, an interface class ac_channel is also provided. This class simplifies the modeling and synthesis of hierarchical designs using C++ function calls. The ac_channel class is a C++ template class that enforces a FIFO discipline (reads occur in the same order as writes.) From a modeling perspective, an ac_channel is implemented as a simple interface to the C++ standard queue (std::deque). That is, for modeling purposes, an ac_channel is infinite in length (writes always succeed) and attempting to read from an empty channel generates an assertion failure (reads are nonblocking).

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

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 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 $b_{W-1}… b_1 b_0$ 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 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 $b_{W-1} ...b_1 b_0$ is given by $(b_{W-1} … b_1 b_0) 2^I$ or equivalently $(b_{W-1} … b_1 b_0) 2^{I-W}$ where $b_{W-1} … b_1 b_0$ is interpreted as an unsigned integer or a signed (two's complement) number.

Table 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: 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"). In all the examples of Table 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 3 shows the operators defined for both *ac_int* and *ac_fixed*

Table 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 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 2I-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 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 4: Methods defined for *ac_int* **and** *ac_fixed*.

Methods	*ac_int*<W,S>	*ac_fixed*<W,I,S,Q,O>
slc<W2>(int_type i)	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 needs to be non-negative and could of any of the following types: *ac_int, unsigned, int*	

set_slc(
  int_type i,
  ac_int<W2,S2> x
)
	Bits of x are copied at slice with LSB index i. That is, bits (W2-1+i downto i) are set with bits of x.
	Parameter i needs to be non-negative and could of any of the following types: *ac_int, unsigned, int*	

range<msb,lsb>()	Returns reference to slice from msb down to lsb. The only operation available with the return type (ac::sliceref) is the assignment operator. A range can be assigned to another range provided the width (msb-lsb+1) of source and target of the assignment match. If either msb or lsb are out of bounds, a static_assert will error out in C++11.	
to_*ac_int*()	NOT DEFINED	return an *ac_int*<W$_I$,S> where W$_I$ is max(*I*, 1). Equiv to AC_TRN quantization. Return type guarantees no overflows.
to_int(), to_uint(), to_long(), to_ulong(), to_int64(), to_uint64()	Conversions to various C *INTs*	Conversions to various C *INTs* Equiv to to_*ac_int*() followed by conversion
to_double()	Conversion to *double*	
read_from_fs(fstream_type &fstream_obj)	Reads data from fstream_obj. fstream_type must be std::ifstream or std::fstream. Refer to [File I/O Constructors and Methods](#).	
write_to_fs(fstream_type &fstream_obj)	Writes ac_int data to fstream_obj. fstream_type must be std::ofstream or std::fstream. Refer to [File I/O Constructors and Methods](#).	

to_string(
  ac_base_mode base_rep,
bool sign_mag = false,
  bool pad_to_width = false
)
	convert to std::string depending on parameters *base_rep* {AC_HEX, AC_DEC, AC_OCT, AC_BIN} and bool arguments *sign_mag and pad_to_width.*	

length()	Returns bitwidth (value of template parameter W)	
set_val<ac_special_val>()	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 18](#) for details.	
leading_sign()	If unsigned (S==false) returns number of leading zeros. If signed	

leading_sign(bool &all_sign)	(S==true) returns number of leading 0/1s minus one. Parameter all_sign is true if return value is W-S (all 0's for S==false, all same for S==true). See <u>Leading Sign and Normalization</u> for details.
normalize(*ac_int*<WE,SE> &exp)	Treats the *ac_int*/*ac_fixed* as a mantissa and attempts to normalize. The mantissa shifted partially (not normalized) if exp saturates to its minimum value. See "<u>Leading Sign and Normalization</u>" for details.
normalize_RME(*ac_int*<WE,SE> &exp)	Same as above but minimum exponent is reserved (*Reserve Minimum Exponent)* so *exp* saturates at minimum exponent plus one. See "<u>Leading Sign and Normalization</u>" for details.
reverse()	Reverses the bit pattern of an ac_int (currently not defined for ac_fixed) and, this void method updates the value of the ac_int variable associated with the current object by assigning an unsigned ac_int of the same width, and does not return any value.
reversed()	Reverses the bit pattern of an ac_int (currently not defined for ac_fixed) and returns an unsigned ac_int of the same width. It does not modify the ac_int object passed to it.
bit_complement()	Returns the raw complemented bits in an unsigned result that has the same width (and integer width for ac_fixed). It is not equivalent to the ~ operator which returns an arithmetic result.

bit_fill(
  const int (&ivec)[Na],
  bool bigendian=true
)
	Use the raw bits in the integer vector to fill ac_int/ac_fixed. Missing most significant bits are set to 0. Extra bits are ignored. Element ivec[0] is treated as the most/least significant bits when bigendian is true/false.

bit_fill_hex(const char *str)	Use the hex characters 0-9a-fA-F in string to fill ac_int/ac_fixed. Missing most significant bits are set to 0. Extra bits are ignored. Asserts on non hex characters.

Table 5 shows the reduce methods for *ac_int*. They take the *W* bus of the *ac_int* and apply the respective logical operation on them and return *bool*.

Table 5: Reduce Methods defined for *ac_int and ac_fixed*

Reduce Methods	*ac_int*	**ac_fixed**
and_reduce()	AND of bits	
or_reduce()	OR of bits	
xor_reduce()	XOR of bits	

Table 6 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 6: 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
std::ifstream/std::fstream	Initializes data with filestream. Refer to [File I/O Constructors and Methods](#).	

2.1. 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 7](#).

  ○ *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 1](#), 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.
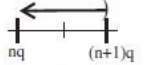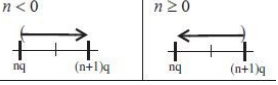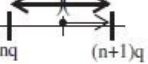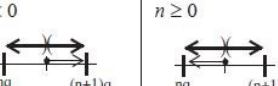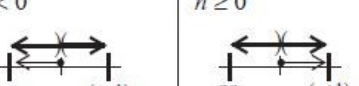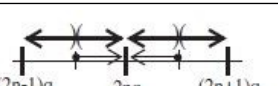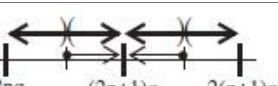
The modes are specified by the 4th and 5th template argument to *ac_fixed*:

```
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 7. The enumeration values for *ac_o_modes* are shown in Table 8. 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 7: Quantization modes for *ac_fixed*

Modes	Behavior n is integer, q is 2^{I-W}	Simulation/Synthesis cost
AC_TRN (default) (trunc towards -∞)		Delete bits, no cost except for /= (div assign) signed
AC_TRN_ZERO (trunc towards 0)		No cost for /=, or unsigned src For signed: incrementer, OR for deleted bits, AND with sign bit
AC_RND (round towards +∞)		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)		
AC_RND_INF (rounds towards +/-∞)		
AC_RND_MIN_INF (round towards -∞)		
AC_RND_CONV (round towards even q multiples)		
AC_RND_CONV_ODD (round towards odd 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 quantization rounds towards even multiples of the quantization while the AC_RND_CONV_ODD quantization rounds towards odd 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 8. 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 1) 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).

The following operators can destroy the symmetric saturation property of an *ac_fixed* with AC_SAT_SYM and should be avoided:

- Changing a bit with operator [].

- Changing bit(s) with the set_slc method.

- Performing a shift assign (<<=, >>=) does not trigger quantization or overflow handling.

- Performing a shift (<<, >>) returns a result of type of the first operand. If the first operand is an ac_fixed with AC_SAT_SYM, the result type will be an ac_fixed with AC_SAT_SYM, but it is not guaranteed to be symmetrically saturated.

Once the symmetrical saturation property has been destroyed, assignment to the same type will not trigger symmetrical saturation as the folllowing example illustrates:

```
typedef ac_fixed<8,8,true,AC_TRN,AC_SAT_SYM> fx_ss;
typedef ac_fixed<8,8,true> fx;
fx_ss a = 0;
a[7] = 1; // No longer symmetrically saturated

fx_ss b = a; // b remains non symmetrically saturated as a is assumed to be
             //  symmetrically saturated and has identical type

a = (fx) a;  // forcing saturation by first casting to non-symmetrically
             //  saturated type
```

.A similar example with shifting:

```
typedef ac_fixed<8,8,true,AC_TRN,AC_SAT_SYM> fx_ss;
fx_ss a = 1;
```

```
a <<= 7; // Value of a is not symmetrically saturated
fx_ss b = 1;
b = b << 7; // Value of b is not symmetrically saturated as return type of
          // b << 7 is ac_fixed<8,8,true,AC_TRN,AC_SAT_SYM>
a = b;  // Value of a remains non symmetrically saturated
```

As the above example illustrates a <<= v, is equivalent to a = a << v.

Table 8: Overflow modes for *ac_fixed*

Mode	Behavior all references are to *target* type MIN, MAX are limits as in **Table 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 . Note: if source is of type AC_SAT_SYM, it will be treated as already symmetrically saturated.	Overflow checking and Saturation logic

2.2. Using the ac_int and ac_fixed Datatypes

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 4](#) and [Table 6](#)). The value of the fixed point is given by:

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
```

2.3. 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:

    ◦   Binary Arithmetic and 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 Operators: the result is a boolean value (true/false): >, <, >=, <=, ==, !=. Mixing of *ac_int*, *ac_fixed*, native C integers and *double* is allowed.

    ◦   Shift Operators and shift assign operators: <<, >>, =<<, =>>. The second argument is an *ac_int* or a native C integer.

- Unary Operators: +, -, ~ and ! (one operand). The ! operator returns bool.

- Bit Complement

- Pre/Post Increment and 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*.

- Slice Read Method: slc and Slice Write Method: set_slc to read and modify a slice in an *ac_int* or *ac_fixed*. A slice of an *ac_fixed* is an *ac_int*.

- Range Method: range<msb,lsb>(): allows direct assignment of slices: y.range<7,0>() = x.range<15,8>().

- Conversion Operators to C Integer Types and Explicit Conversion Methods to C native types.

- The set_val Method

- Constructors from *ac_int* and C native types.

- Methods to Fill Bits

- File I/O Constructors and Methods

- Mixing ac_int and ac_fixed with Other Datatypes.

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

2.3.1. 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 9 and Table 10). 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 9 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 $W_R = abs(min(S_1 ? -W_1:W_1, S_2 ? -W_2:W_2))$. 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 9: Return Types for *ac_int* Binary Arithmetic and Bitwise Logical Operations

Operator	Return Type: $ac\_int\langle W_R,S_R\rangle$	
	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+W_2
%	S_1	$\min(W_1, W_2+!S_2\&S_1)$
&	$S_1 \mid S_2$	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)$
|	$S_1 \mid S_2$	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)$
^	$S_1 \mid S_2$	$\max(W_1+!S_1\&S_2,W_2+!S_2\&S_1)$

Table 10 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 10: Return Types for *ac_fixed* Binary Arithmetic and Bitwise Logical Operations

Operator	Return Type: $ac\_fixed\langle W_R,I_R,S_R,AC\_TRN,AC\_WRAP\rangle$		
	S_R	Bit Width: W_R	Integer Bit Width: IR
+	$S_1 \mid S_2$	$I_R+\max(W_1-I_1,W_2-I_2)$	$\max(I_1+!S_1\&S_2,I_2+!S_2\&S_1)+1$
-	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 11 shows a few examples of mixed operations.   The variables in this table are defined as:

```
ac_int<7,true>      i_s7;
ac_fixed<20,4,false> fx_s20_4;
signed char       c_s8;
```

Table 11: Mixed Expressions Example

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.

2.3.2. 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.

2.3.3. 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, except for A1 is ac_fixed with
           // AC_SAT_SYM
A1 >>= A2; // equiv to A1 = A1 >> A2, except for A1 is ac_fixed with
           // AC_SAT_SYM and value of A2 < 0
```

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.

  - $0 \le s < 32$ for 32-bit numbers

  - $0 \le s < 64$ for 64-bit numbers

- The behavior for shift values outside the allowed ranges is not specified by the C++ ISO standard.

2.3.4. 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 12 lists the unary operators and their return types.

Table 12: 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 13 lists the unary operators for *ac_fixed* and their return types.

Table 13: Unary Operators for *ac_fixed***<W,I,S,Q,O>**

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

2.3.5. Bit Complement

The *bit_complement* member function for *ac_int* and *ac_fixed* are provided as an alternative to the operator ~ which grows by one bit when applied to an unsigned type. The methods are:

```
ac_int<W, false> ac_int<W,S>::bit_complement() const;
ac_fixed<W, I, false> ac_fixed<W,I,S,Q,O>::bit_complement() const;
```

They return an unsigned version of the same W (and same I for ac_fixed). This a bit complement of the raw bits as compared to the complement operator ~ that returns an arithmetic value of -x-1 for ac_int and $-x-2^{I-W}$ for ac_fixed. The following example illustrates the difference:

```
ac_int<3,false> x = 7;    // 111
ac_int<5,true> y;
y = ~x;    // returns - 7 - 1 = -8 (1000) as ac_int<4,true>, y  = 11000
y = x.bit_complement();   // returns 000 as ac_int<3,false>, y = 00000
ac_int<4,false> x2 = 7;   // 0111
y = ~x2;   // returns -7 - 1 = -8 (11000) as ac_int<5,true>,  y  = 11000
y = x2.bit_complement();  //  returns 1000 as ac_int<4,false>, y = 01000
```

2.3.6. Increment and Decrement Operators

Pre/Post increment/decrement for *ac_int* have the usual semantics as shown in Table 14 (*T_x* is the type of variable *x*).

Table 14: 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;
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 15 (*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 15: Pre- and Post-Increment/Decrement Operators for *ac_fixed<W,I,S,Q,O>*

where q=2^{l-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;

2.3.7. 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 16 shows some typical scenarios:

Table 16: 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
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
if (k != 0)	OK, operator != 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
if (x != 0)	OK, operator != 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.

2.3.8. 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 17. 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 17: 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>

The explicit method to covert to a string representation is:

```
inline std::string to_string(ac_base_mode base_rep, bool sign_mag = false, bool
pad_to_width = false) const;
```

The first argument is one of {AC_HEX, AC_DEC, AC_OCT, AC_BIN} and will determine the format. For AC_DEC, the format of the value is always in sign-magnitude form and the second and third arguments don't apply.

When the base_rep is AC_HEX, AC_OCT or AC_BIN, a prefix of 0x, 0o or 0b are used respectively. In those cases the two arguments sign_mag and pad_to_width apply. When pad_to_width is false, the string is reduced to its shortest length assuming zero-padding or sign extension rules. When pad_to_width is true, the string captures the full length of the type. The sign_mag argument when set to true first takes the absolute value and prefixes the string with the sign of the value. For example:

```
ac_int<14,true> x14s = 0;
std::cout << x14s.to_string(AC_HEX,false,false) << std::endl;  // prints 0x0
std::cout << x14s.to_string(AC_HEX,false,true) << std::endl;  // prints 0x0000
std::cout << x14s.to_string(AC_HEX,true,false) << std::endl;  // prints +0x0
std::cout << x14s.to_string(AC_HEX,true,true) << std::endl;  // prints +0x0000
x14s = -1;
std::cout << x14s.to_string(AC_HEX,false,false) << std::endl;  // prints 0xF
std::cout << x14s.to_string(AC_HEX,false,true) << std::endl;  // prints 0x3FFF
(not 0xFFFF)
std::cout << x14s.to_string(AC_HEX,true,false) << std::endl;  // prints -0x1
std::cout << x14s.to_string(AC_HEX,true,true) << std::endl;  // prints -0x0001
```

2.3.9. 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" in the *Frequently Asked Questions* section.

2.3.10. 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" in the *Frequently Asked Questions* section.

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.

2.3.11. 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" in the *Frequently Asked Questions* section.

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.

2.3.12. Range Method: range<msb,lsb>()

The range method returns a reference to the ac_int or ac_fixed. The range method returns an ac::sliceref that has the assignment operator defined so that a range can be assigned to another range provided their lengths match. The msb and lsb need to be in bounds, otherwise a static_assert errors out in C++11. Such an error is also triggered if lsb is greater than msb.

```
ac_int<10,true> x = …;
ac_fixed<16,2,false> y = …;
y.range<13,10>() = x.range<7,4>();     // assign bits 7 downto 4 of x to 13
downto 10 of y
y.range<8,8>() = x.range<8,7>();    // no matching assignment operator found
y.range<16,2>();  // error in C++11
y.range<2,-1>(); // error in C++11
```

2.3.13. 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 18.

Table 18: 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 1.
AC_VAL_MAX	Maximum value as specified in Table 1.
AC_VAL_QUANTUM	Quantum value as specified in Table 1.

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*.

2.3.14. Constructors

Constructors from all C-types are provided. Constructors from *ac_int* are also provided. The default constructor does not initialize *ac_int* or *ac_fixed*. Thus non-static variables of type *ac_int* or *ac_fixed* will not be initialized by default. If the AC_DEFAULT_IN_RANGE macro is defined before the first inclusion of the AC datatype header, then the default constructor will adjust the un-initialized value to force it to be in range:

```
#define AC_DEFAULT_IN_RANGE
#include <ac_int.h>
…
ac_int<3,false> n; // n will be in range of 0 to 7
```

Constructors from char *, have not been defined/implemented in the current release. See Methods to Fill Bits for alternatives.

2.3.15. Methods to Fill Bits

Utility functions *bit_fill_hex* and *bit_fill* can be used to initialize large bitwidth *ac_int* and *ac_fixed* with raw bits. What is meant by "*raw bits*" is that its argument is treated as an unsigned bit pattern, without a fixed point and no rounding or overflow handling is performed. The *bit_fill_hex* accepts a hex string. The *bit_fill* accepts and array of integers and it should be the preferred alternative to initialize large *ac_int* and *ac_fixed* with raw bits.

*NOTE: The bit_fill_hex method should **only** be used to initialize static constants since it is significantly slower than alternative methods.*

They are available both as member functions of *ac_int* and *ac_fixed* and as global functions in the ac namespace that return the type specified as a template parameter (the type T needs to be either an ac_int or an ac_fixed):

```
void ac_int<W,S>::bit_fill_hex(const char *str);
void ac_fixed<W,I,S,Q,O>::bit_fill_hex(const char *str);
template<typename T> T ac::bit_fill_hex(const char *str);
template<int Na> void ac_int<W,S>::bit_fill(const int (&ivec)[Na], bool bigen-
dian=true);
```

```
template<int Na> void ac_fixed<W,I,S,Q,O>::bit_fill(const int (&ivec)[Na], bool
bigendian=true);
template<typename T, int N> T ac::bit_fill(const int (&ivec)[N], bool bigen-
dian=true);
```

The *bit_fill_hex* function accepts a hex string as an argument which could be shorter or longer than what is required to fill all bits of the *ac_int* or *ac_fixed*. If it is shorter, it is zero padded to fill the remaining most significant bits. If it longer, the extra most significant bits are truncated. The hex string should be a literal constant string and should only contain hex digit characters (0-9, a-f, A-F). Other characters trigger and assert. Because the initialization is done at runtime and this initialization technique is inherently slow, its use to initialize non-static variables is discouraged.

The *bit_fill* function accepts two arguments:

- The first one is an integer array that contains the bit pattern. It could be longer or shorter than what is required to fill all bits. If it is shorter then the remaining most significant bits are zero padded. If it is longer then what would be the extra most significant bits are truncated. The array is not required to be an array of constants.

- The second is a *bool* argument *bigendian* that defaults to *true*.

which means that the bits in the array element with index 0 become the most significant 32 bits of the bit pattern. If the argument is *false*, then the bits in the array element with index 0 become the least significant 32 bits of the bit pattern.

The following example illustrates the use of *bit_fill_hex* and *bit_fill* that do the equivalent functionality:

```
typedef ac_int<80,false> i80_t;
i80_t x;
x.bit_fill_hex("a9876543210fedcba987");      // member funtion
x = ac::bit_fill_hex<i80_t>("a9876543210fedcba987");   // global function
int vec[] = { 0xa987, 0x6543210f, 0xedcba987 };
x.bit_fill(vec);    // member function
x = bit_fill<i80_t>(vec);    // global function
// inlining the constant array
x.bit_fill( (int [3]) { 0xa987,0x6543210f,0xedcba987 } );  // member function
x = bit_fill<i80_t>( (int [3]) { 0xa987,0x6543210f,0xedcba987 } ); // global
function
```

2.3.16. File I/O Constructors and Methods

The ac_int and ac_fixed classes offer file I/O constructors and methods that read to and write from binary files, using std::ifstream, std::ofstream and std::fstream objects, as applicable, thereby enabling file persistence for ac_int and ac_fixed data. The following are provided, where "ac_type" is either ac_int or ac_fixed:

```
// ac_type can be ac_int or ac_fixed.
explicit inline ac_type(std::ifstream &ifs);
```

```
explicit inline ac_type(std::fstream &fs);
inline void write_to_fs(std::ofstream &ofs) const;
inline void write_to_fs(std::fstream &fs) const;
inline ac_type & read_from_fs(std::ifstream &ifs);
inline ac_type & read_from_fs(std::fstream &fs);
```

The constructors only accept ifstream and fstream inputs and only read from the filestreams. write_to_fs uses the write() method provided with std::ofstream and std::fstream, while the constructors and read_from_fs use the read() method provided with std::ifstream and std::fstream.

The AC_VALID_FSTREAM macro can also be defined to check whether the filestream is valid by calling the good() method for the filestream inputs. Defining the macro and using an invalid filestream will trigger a runtime assertion.

The following example highlights a potential use case for the write_to_fs method:

```
// If file1.dat is not created already, the line below will create it.
std::ofstream ofs("file1.dat", std::ios::out | std::ios::binary);
// An empty file2.dat must be created first, in order to be valid for use with
// fstream.
std::fstream fs("file2.dat", std::ios::out | std::ios::binary);
ac_int<20, true> T_io var1 = 20, var2 = 30;
// Write var1 and var2 data to file1.dat and file2.dat, respectively.
var1.write_to_fs(ofs);
var2.write_to_fs(fs);

ofs.close();
fs.close();
```

The following example highlights a potential use case for the read_from_fs method:

```
// You must make sure that file1.dat and file2.dat already store binary data
// that's written to them with the write_to_fs method, as explained above.
std::ifstream ifs("file1.dat", std::ios::in | std::ios::binary);
std::fstream fs("file2.dat", std::ios::in | std::ios::binary);
ac_int<20, true> var1, var2;
// Read var1 and var2 data from file1.dat and file2.dat, respectively.
var1.read_from_fs(ifs);
var2.read_from_fs(fs);

ifs.close();
fs.close();
```

The following example highlights a potential use case for the file input constructors.

```
// You must make sure that file1.dat and file2.dat already store binary data
// that's written to them with the write_to_fs method, as explained above.
```

```
std::ifstream ifs("file1.dat", std::ios::in | std::ios::binary);
std::fstream fs("file2.dat", std::ios::in | std::ios::binary);
// Read var1 and var2 data from file1.dat and file2.dat, respectively.
// "var1 = ifs" and "var2 = fs" are not allowed as the constructors are
// declared explicit.
ac_int<20, true> var1(ifs), var2(fs);

ifs.close();
fs.close();
```

Overloaded versions of the ac::init_array function iteratively use the read_from_fs method to initialize ac_int/ac_fixed arrays with file data, as described in Using ac::init_array for Initializing Arrays With a Binary File. Similarly, the ac::dump_array functions iteratively use the write_to_fs method to dump data from arrays of ac_int/ac_fixed values to a file, as described in Using ac::dump_array.

2.3.17. Mixing ac_int and ac_fixed with Other Datatypes

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

2.4. 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 to dump the contents of ac_int arrays to a binary file.

- Function for initializing arrays of supported types to a special value.

- Function for initializing ac_int arrays with the contents of a binary file.

- Template structs that provide a mechanism to statically compute log2 related functions.

- Typedefs for finding the return type of unary and binary operators.

2.4.1. 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 19.

Table 19: Basic Parameters

Static member	Description for *ac_int*	Description for *ac_fixed*

width	Value of W template parameter	Value of W template parameter
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
e_width	0	0

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*.

2.4.2. Using ac::dump_array

The ac::dump_array utility function dumps the contents of an ac_int or ac_fixed array to a binary file, through the iterative use of the write_to_fs method described in [File I/O Constructors and Methods](#). The following functions are provided:

```
template<int W, bool S>
inline bool dump_array(const ac_int<W, S> *a, int n, std::ofstream &ofs);
template<int W, bool S>
inline bool dump_array(const ac_int<W, S> *a, int n, std::fstream &fs);
template<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>
inline bool dump_array(const ac_fixed<W,I,S,Q,O> *a, int n, std::ofstream
&ofs);
template<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>
inline bool dump_array(const ac_fixed<W,I,S,Q,O> *a, int n, std::fstream &fs);
```

Warning: *The function ac::dump_array does not check for array bound violations.*

The second parameter for all the functions is the number of consecutive array elements which will be dumped to the file.

Note that ac::dump_array follows the same rule for filestreams as write_to_fs: They must either be std::ofstream or std::fstream objects. The following snippet highlights an example use case:

```
// If file1.dat is not created already, the line below will create it.
std::ofstream ofs("file1.dat", std::ios::out | std::ios::binary);
// An empty file2.dat must be created first, in order to be valid for use with
// fstream.
std::fstream fs("file2.dat", std::ios::out | std::ios::binary);
ac_int<20, true> arr1[20];
ac_int<20, true> arr2[20];
// Initialize arr1 and arr2 as appropriate.
// Dump arr1 and arr2 to file1.dat and file2.dat, respectively.
ac::dump_array(&arr1[0], 20, ofs);
```

```
ac::dump_array(&arr2[0], 20, fs);
ofs.close();
fs.close();
```

Other usage examples are shown below. The ofs object should be declared and closed similarly to the one in the snippet above.

- Dumping a multi-dimensional array:

```
ac_int<20, true> some_3D_arr[4][3][7];
// Initialize some_3D_arr as appropriate.
ac::dump_array(&some_3D_arr[0][0][0], 4*3*7, ofs);
```

- Dumping a subset of consecutive array elements:

```
ac_int<20, true> arr[500];
// Initialize arr as appropriate.
// Dump elements from index 100 to index 149:
ac::dump_array(&arr[100], 50, ofs);
```

2.4.3. Using ac::init_array for Initializing Arrays With a Special Value

The *ac::init_array* utility function 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);
   ...
}
```

Warning: *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 18 for details).

The function is defined for the integer and fixed point datatypes shown in Table 20:

Table 20: Required Include Files for ac::init_array Function

Type	Required include file
C integer types	*ac_int*.h
ac_int, *ac_fixed, ac_complex*	No additional include
Supported SystemC types	*ac_int*.h, ac_sc.h

The first argument is of type pointer to one of the types in Table 20. 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 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);
```

2.4.4. Using ac::init_array for Initializing Arrays With a Binary File

You can also use the ac::init_array function to initialize ac_int and ac_fixed arrays with the contents of a binary file, through the iterative use of the read_from_fs method described in File I/O Constructors and Methods. The following functions are provided:

```
template<int W, bool S>
inline bool init_array(ac_int<W, S> *a, int n, std::ifstream &ifs);
template<int W, bool S>
inline bool init_array(ac_int<W, S> *a, int n, std::fstream &fs);
template<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>
inline bool init_array(ac_fixed<W,I,S,Q,O> *a, int n, std::ifstream &ifs)
template<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>
inline bool init_array(ac_fixed<W,I,S,Q,O> *a, int n, std::fstream &fs)
```

Warning: The function ac::init_array does not check for array bound violations.

The second parameter for both functions is the number of consecutive array elements that will be initialized.

Note that these versions of ac::init_array follow the same rule for filestreams as read_from_fs: They must be either std::ifstream or std::fstream objects. You must first write to the files using the ac::dump_array function described in Using ac::dump_array.

The following snippet highlights an example use case:

```
// You must make sure that file1.dat and file2.dat already store binary data
// that's written to them with the ac::dump_array function, as explained above.
std::ifstream ifs("file1.dat", std::ios::in | std::ios::binary);
std::fstream fs("file2.dat", std::ios::in | std::ios::binary);

ac_int<20, true> arr1[20];
ac_int<20, true> arr2[20];
ac::init_array(&arr1[0], 20, ifs);
ac::init_array(&arr2[0], 20, fs);
ifs.close();
fs.close();
```

Other usage examples are shown below. The ifs object should be declared and closed similarly to the one in the snippet above.

- Initializing a multi-dimensional array:

```
ac_int<20, true> some_3D_arr[5][6][4];
ac::init_array(&some_3D_arr[0][0][0], 5*6*4, ifs);
```

- Initializing a subset of array elements:

```
ac_int<20, true> arr[500];
// Initialize elements from index 100 to index 149.
ac::init_array(&arr[100], 50, ifs);
```

2.4.5. Static Computation of log2 Functions

This section covers the static computation of 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 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 21.

Table 21: 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).

2.4.6. 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 22 (summation is not really an unary operator, but it depends on a single type).

Table 22: 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 23.

Table 23: 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 23 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;
}
```

2.5. Methods and Utility Functions for Floating Point

Both *ac_int* and *ac_fixed* provide methods to help implementation of floating point or blocking floating point functionality. The methods implement *leading_sign* and *normalization* and are described in detail in this section. In addition, utility functions are defined to get the sign, mantissa and exponent of C++ constant *float* and *double* literals. They are named after the commonly used *frexp* function defined in *math.h*.

2.5.1. Leading Sign and Normalization

The *leading_sign* method does not change the *ac_int*/*ac_fixed* object.

There are two forms:

```
typename rt_unary::leading_sign leading_sign() const;
typename rt_unary::leading_sign leading_sign(bool &all_sign) const;
```

The return type is found so that it has the minimum static bitwidth that can represent the largest value that maybe returned. For unsigned *ac_int*/*ac_fixed*, *leading_sign* returns the number of leading zeros. For signed *ac_int*/*ac_fixed*, *leading_sign* returns the number of leading 0/1s minus one. Leading sign thus returns the largest integer such that:

```
y = x.leading_sign() then x == (x << y) * 2-y
```

The second form has the argument *all_sign* that is set to true if all bits are zeros for unsigned or all bits are the same for unsigned *ac_int*/*ac_fixed* types. The only purpose is to save the extra comparison required to compute that information from the return value. One use for *leading_sign* is for computing the shift left for normalization of a mantissa, though this use is covered by the normalization methods provided for both *ac_int*/*ac_fixed* described next.

The normalization methods treat the *ac_int*/*ac_fixed* as a mantissa and has two forms:

```
bool normalize(ac_int<WE,SE> &exp);
bool normalize_RME(ac_int<WE,SE> &exp);
```

The first form shifts the mantissa left as much as possible to normalize it such that:

```
exp = exp_i;
man = main_i;
bool b = man.normalize(exp);
```

then

$$man * 2^{exp} == man\_i * 2^{exp\_i}$$

which requires that the mantissa does not lose information (see *leading_sign* description) and the exponent does not saturate. Depending on the initial exponent and *man.leading_sign()*, normalize may or may not fully normalize the mantissa. The return value is true if the mantissa is normalized after the call.

Note that there is *no implied* leading 1 in the mantissa. Also for sign/magnitude floating representations (such as IEEE floating point representations), an unsigned mantissa should be used. Note that the minimum exponent depends on the value of the template parameters WE and SE.

The second form *normalize_RME* is similar but reserves the minimum exponent. This is useful as the minimum exponent is often used to encode special numbers (NaN etc.) in commonly used floating point representations.

2.5.2. Utility Function to Extract Exponent/Sign/Mantissa from Literal Constants

The following functions in the "ac" namespace provide ways to extract the mantissa, exponent and optionally sign from C *float* and *double* literal constant types:

```
ac_fixed<54,2,true> frexp_d(double d, ac_int<11,true> &exp);
ac_fixed<25,2,true> frexp_f(float f, ac_int<8,true> &exp);
ac_fixed<53,1,false> frexp_sm_d(double d, ac_int<11,true> &exp, bool &sign);
ac_fixed<24,1,false> frexp_sm_f(float f, ac_int<8,true> &exp, bool &sign);
```

The first two return the mantissa into a two's complement signed *ac_fixed*. The mantissa is normalized (mant[msb] != mant[msb-1]) unless the exponent is saturated to the IEEE standard exponent minimum of -126 for *float* or -1022 for *double*. The mantissa is provided in full (no implied bit).

The last two ("_sm_" in their name) return the result in a sign-magnitude representation: the return value is an unsigned *ac_fixed* and the sign is returned as an argument passed by reference.

Example:

```
ac_int<8,true> exp;
ac_fixed<54,2,true> mant = ac::frexp_d(0.1, exp);
```

Chapter 3: Arbitrary-Length Bit-Accurate Floating-Point Datatypes

Floating-point types are used to represent a higher range of values than a fixed-point representation having the same number of bits. Floating-point types have both a mantissa and an exponent. The *mantissa* is a value in a fixed-point representation and the *exponent* that shifts the point making it a "floating" point. The type provides a templatized class where values are encoded as a signed mantissa m and an exponent e such that:

```
value = m * 2ᵉ
```

where m is a signed fixed-point quantity and e is a signed integer. The class is templatized as:

```
ac_float<W,I,E,Q>
```

where the first two parameters *W* and *I* define the mantissa as an *ac_fixed<W,I,true>*, the *E* defines the exponent as an *ac_int<E,true>* and *Q* defines the rounding mode. For example:

```
#include <ac_float.h>
...
ac_float<10,1,4,AC_RND> x = ...; // W=10, I=1, E=4, Q=AC_RND
ac_float<10,1,4> y = ...; // Q = AC_TRN (default setting)
...
```

The definition of *ac_float* is targeted for algorithms that are intended to be mapped to hardware where the flexibility of tailoring widths of mantissa and exponents is critical to reduce hardware resources. The *I* parameter provides a static exponent offset so that the exponent bit width can be minimized. The use of the *I* parameter and the encoding of the mantissa as an *ac_fixed* provides a more natural transition to and from *ac_fixed*.

Currently the typical optimization of the MSB of the mantissa as an implied '1' bit is not implemented in *ac_float*. This was done to keep the implementation as simple as possible. Supporting the implied '1' bit requires using up one exponent value to encode special conditions such as the fact that the number is denormalized and thus the MSB is not '1'. Currently *ac_float* does no reserve any exponent to encode special conditions and therefore does not support *NaN, +inf, -inf*.

The encoding of the mantissa is in two's complement. This makes it consistent with *ac_fixed*. Typical implementations for floating-point use a sign-magnitude representation. Such a representation introduces some complexities, but may reduce toggling of bits and be beneficial for lowering the power used in a hardware implementation of the algorithm. Supporting such an sign-magnitude option will be considered as a future enhancement.

As compared to *ac_fixed*, saturation (equivalent to *AC_SAT*) is performed by default. This was done for the following reasons:

- The hardware overhead for saturation logic is relatively small compared to the inherent complexity of floating-point arithmetic

- Finite precision floating-point addition is not associative. This means that optimizations such as tree balancing can not be performed without affecting bit-accuracy. Using saturation rather than wrap around does not worsen this situation.

- The number of template parameters is kept to a minimum on purpose. Additional parameters may be required for future enhancements.

The *ac_float* implementation is built in top of the *ac_fixed* and *ac_int* types. In principle all parameters could be set to arbitrary valid values. However the current implementation requires that the sum of exponent values and offsets (*I* parameter) fit in a 32-bit integer. Note that a C floating *double* has 11 bits of exponents, so it is unlikely that the current limitation is restrictive in most cases. An exponent width of 0 is also not currently supported.

The numeric range of an *ac_float*<W,I,E> (the *Q* parameter has a default of *AC_TRN* and has no impact on the range) has a numeric range of:

$$(-0.5) \; 2^{I + max\_exp} \text{ to } (0.5 - 2^{-W}) \; 2^{I + max\_exp} \text{ where } max\_exp = 2^{E-1} - 1$$

The smallest increment (quantum) is:

$$2^{I-W+min\_exp} \text{ where } min\_exp = -2^{E-1}$$

Based on the ranges a standard *float* is covered by *ac_float*<25,2,8> and *double* is covered by *ac_float*<54,2,11>. These types are provided by the type defines ac::ac_float_represent<T>::type where T could be float or double.

Note that the overall number of bits (*W+E*) is 33 and 65 as there is no implied '1' bit in *ac_float*. There are differences of rounding between the native C *float* types (*float* and *double*) and the smallest *ac_float* that covers them in range. For example, the division operator for *ac_float* truncates towards zero (the rounding behavior might be enhanced in the future when both addition and subtraction operators will be implemented).

Table 24 summarizes the characteristic of *ac_float* and contrasts it to typical floating-point.

Table 24: Operators and methods defined for ac_float

Characteristic	Description and comparison to typical floating-point
Zero	Represented as all zeros for both mantissa and exponent.
Normalized Numbers	No implied '1' bit.
Denormalized Numbers	No exception required as there is no implied '1' for normalized numbers.
Infinity (+inf, -inf)	Not represented. Used by typical floating point types to represent result of a/0 where a!=0.
NaN (not-a-number)	Not represented. Used by typical floating point types to represent inf/inf, inf-inf etc.
Mantissa Representation	Two's complement (signed). Stored as an *ac_fixed<W,I,true>*.

	Parameter *I* can be regarded as an exponent bias.
Sign bit	Not used as mantissa is signed (two's complement) already.
Exponent	Stored as two's complement as an *ac_int*<E,true>. Different from typical floating point type stored as an unsigned and requiring an offset to get actual value.
Base	Base is 2. Typical floating-point types also have base 2. IEEE standards do allow for other bases (base 10).
Target Use	Algorithmic descriptions that need to be optimized for minimal hardware. Parameterization of precision and range and mixing of different types is essential. This is different than typical floating-point types that are "closed" (result is same type as operands).
Rounding	Done on assignment according to Q parameter.
Overflow	Saturation overflow always performed unless target range covers source range (based on templates).
Operators with non-floats	Operators with *ac_fixed*, *ac_int* and C integers, first represent the operand as an *ac_float* with E=1 (E=0, currently not supported.

Table 25 shows the operators and methods defined for *ac_float*

Table 25: Operators and methods defined for ac_float

Operators	ac_float
*	Arithmetic result (no loss of precision). Mantissa type is determined by *ac_fixed* rules for multiplication. Exponent type is determined by *ac_int* rules for addition.
/	Mantissa type is determined by *ac_fixed* rules for division (truncation towards zero). Exponent type is determined by *ac_int* rules for subtraction.
add(op1, op2) sub(op1, op2)	Methods to add and subtract op2 to/from op1. The result is stored in the object ("this"). This allows to consider the type of the target (including the *Q* parameter of the object) to perform rounding.
+, -	To be defined. Current definition is placeholder and can be enabled by defining *__AC_FLOAT_ENABLE_ALPHA__*. It is only used to test compatibility with ac_complex. Challenge is that these operate loose precision and thus need to truncate/round. But it would ideal to avoid two steps of rounding in an assignment "a = b + c".

>>, <<	Bidirectional. Mantissa unchanged. Shift value added to (<<) or subtracted from (>>) exponent. No loss of precision as exponent grows to accommodate resulting exponent. Second arg is *ac_int* or C *INT*
=	quantization (specified by target), then saturation if range of target does not cover range of source
+=, -=, *=, /=, >>=, <<=	Equiv to op then assign where defined. First arg is *ac_float*
==, !=, >, <, >=, <=	First or second arg may be *ac_fixed*, *ac_int* or C *INT* or C *float* or *double*
Unary +, -	Arithmetic
! x	Equiv to x == 0
mantissa(), exp()	Access methods to mantissa and exponent values
Explicit conversion to other types	*to_double(), to_float(), to_ac_fixed(), to_ac_int(), to_int(), to_uint(), to_long(), to_ulong(), to_int64() and to_uint64()*
Constructors from other types	from *ac_fixed*, *ac_int*, *float*, *double*, C integers. Perform normalization by default (argument to constructor).
Checking Normalization Status	*is_norm()* and *is_subn()*. Checks normalization status of ac_float value. More details are given in Normalization Checking.
%, %=, &, |, ^, &=, |=, ^=, ~, ++, --, []	Not defined.

3.1.1. Mixed ac_float and other types

Mixed operators are defined for *ac_int*, *ac_fixed*, C native types to avoid ambiguity in the semantics or compilation problems due to multiple operators matching an operation. The other type is first represented as an *ac_float* of mantissa that can represent the number assuming a zero exponent. The exponent width would in theory be 0, but currently an exponent width of 1 is minimum to compile, so that is used instead.

3.1.2. Shift Operators

Shift operators have a different behavior than for *ac_int* and *ac_fixed*. For *ac_float*, the shift operators are arithmetic (no loss of precision).

For minimal hardware, it is best to constrain the type of the second operand to reduce its range.

3.1.3. 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 26.

Table 26: Special values

ac_special_val enum	Value for ac_float
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
AC_VAL_MAX	Maximum value
AC_VAL_QUANTUM	Quantum value

Direct assignment of the enumeration values should not be used since it will assign the integer value of the enumeration to the *ac_float*.

3.1.4. Constructors

Constructors from all C-types are provided. Constructors from *ac_int* and *ac_fixed* are also provided. The default constructor does not initialize the *ac_float* variable. Thus non-static variables of type *ac_fixed* will not be initialized by default. If the *AC_DEFAULT_IN_RANGE* macro is defined before the first inclusion of the AC datatype header, then the default constructor will adjust the un-initialized value to force it to be in range:

```
#define AC_DEFAULT_IN_RANGE
#include <ac_float.h>
...
ac_float<4,4,3,false> n; // n will be in range
```

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

3.1.5. 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* is an *ac_float*. In this case *T::width* would provide that information. The various parameters that can be accessed are shown in Table 27.

Table 27: Basic Parameters

Static member	Description for ac_float
width	Value of *W* template parameter
i_width	Value of *I* template parameter
sign	true
q_mode	Value of *Q* template parameter
o_mode	AC_SAT
e_width	Value of *E* template parameter

3.1.6. Using ac::init_array for Initializing Arrays

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

```
void foo(...) {
  static ac_float<10,1,3> b[200];
  static bool b_dummy = ac::init_array<AC_VAL_DC>(b,200);
...
}
```

3.1.7. Normalization Checking

The is_norm() and *is_subn()* methods check the normalization status for a given ac_float value.

is_norm() returns false if the value is not normalized. This can apply for the following two cases:

• Value is subnormal, i.e. it is too small to represent as a normalized value.

• Value is zero. No attempt at normalization was made *is_subn()* will return true if the value is subnormal.

The snippet below shows how to use these methods.

```
void foo( ... ) {
  ac_fixed<10, 2, true> mant = 0.125;
  ac_int<6, true> exp_min = -(1<<(6-1)), exp_0 = 0;
  ac_float<10, 2, 6> fl_norm(mant, exp_0, true);
  ac_float<10, 2, 6> fl_no_norm(mant, exp_0, false);
  ac_float<10, 2, 6> fl_subn(mant, exp_min, true);

  // flag1 = true because value is normalized.
  bool flag1 = fl_norm.is_norm();

  // flag2 = false because no attempt at normalization was made.
  bool flag2 = fl_no_norm.is_norm();

  // flag3 = false because value is subnormal.
  bool flag3 = fl_subn.is_norm();

  // flag4 = true because value is subnormal.
  bool flag4 = fl_subn.is_subn();
}
```

Note: If using *is_norm()*, the mantissa must have at least two bits. If using *is_subn()*, both the mantissa and exponent must have at least two bits. Static assertions are provided to guard against potential bitwidth

incompatibilities, if the user uses C++11 or a later compiler standard.

Chapter 4:   Arbitrary-Length Bit-Accurate Standard Floating-Point Datatypes

This chapter covers "standard" (such as IEEE) floating-point types. These types are different from the ac_float in the following characteristics:

1. The data type of the result of arithmetic operations such as +,-,*,/ is the same as the data type of the operands: there is no bit growth.

2. The mantissa is encoded with an implied bit for normal numbers. This requires that an exponent value (all zeros) is treated as special to encode that there is no implied bit for representing zero and subnormal numbers.

3. Special values such as infinity (+Inf, -Inf) and not-a-number (NaN) are encoded using an exponent value (all ones).

4. The mantissa is encoded in sign-magnitude instead of two's complement used in ac_float. This implies that there is a separate sign bit. There are two representations for zero: +0 and -0.

There are three standard floating-point data types that have been added as shown in Table 28.

Table 28: Standard floating-point types

Type	Bit widths	Description	Memory Footprint
ac_ieee_float<Format>	16	Binary format IEEE floats	short
	32		int (matches float)
	64		long long (matches double)
	128		long long [2]
	256		long long [4]
bfloat16	16	bfloat16 from Google	short
ac_std_float<W,E>	Arbitrary	W is overall width E is exponent width	ac_int<W,true> may depend on macros

Note that the memory footprint of *ac_ieee_float* and *bfloat16* is guaranteed while *ac_std_float* may depend on macro settings that may add base classes for expanded functionality.

4.1. IEEE Floating-point: ac_ieee_float

The *ac_ieee_float* provides support for standard IEEE binary floating-point numbers. It is implemented using *ac_std_float* with the corresponding *W* and *E* settings as shown in Table 29. The table also shows *typedef* names for them as well as runtime acceleration using C++ float/double as described in the following sections.

Most operators and methods are the same as for *ac_std_float* (see Section General Standard Floating-point: ac_std_float) and *ac::bfloat16* (see Section Brain Floating-point from Google: ac::bfloat16).

A subset of the *math.h* functions that are defined for *float* and *double* are defined for *ac_ieee_float* under the *std* namespace and member functions variations are also provided.

Table 29: IEEE floating-point types

Type	Typedef	ac_std_float used	Acceleration using C++ float/double (see below)
`ac_ieee_float<binary16>`	ac_ieee_float16	ac_std_float<16,5>	Not available
`ac_ieee_float<binary32>`	ac_ieee_float32	ac_std_float<32,8>	Available, uses *float*
`ac_ieee_float<binary64>`	ac_ieee_float64	ac_std_float<64,11>	Available, uses *double*
`ac_ieee_float<binary128>`	ac_ieee_float128	ac_std_float<128,15>	Not available
`ac_ieee_float<binary256>`	ac_ieee_float256	ac_std_float<256,19>	Not available

Acceleration using C++ float and double

The 32 and 64 bit versions of *ac_ieee_float* have a mode where the standard float or double C++ types are used for faster runtime. This is done by defining the compiler macro *AC_IEEE_FLOAT*_USE_BUILTIN. In order to get bit-accurate behavior it is important that the default rounding mode is used and that no extended precision (temporaries extended to 80bit) is used. For example, in the GCC c++ compiler targeting i386/x86-64, the *-mfpmath=sse* compiler switch should avoid that source of mismatch:

```
c++ -mfpmath=sse -DAC_IEEE_FLOAT_USE_BUILTIN ...
```

Constructors

Any constructor other than the copy constructor, the default constructor and the constructor from the underlying *ac_std_float* is declared explicit. They are therefore not available in assignments (the assignment operator is not defined) or for implicit conversions. The constructors that are declared explicit have some cost in terms of runtime and hardware if targeted for synthesis.

Table 30: Constructors

Constructor	Description
`ac_ieee_float()`	Does not initialize value
`ac_ieee_float(const ac_ieee_float &f)`	Copy constructor
`ac_ieee_float(const ac_std_float<width,e_width> &f)`	From underlying *ac_std_float* (same width and e_width)
`explicit ac_ieee_float(const ac_std_float<W,E> &f)`	From *ac_std_float* other than the underlying one.
`explicit ac_ieee_float(const ac_ieee_float<Format2> &f)`	Constructor from *ac_ieee_float* from a different format
`explicit ac_ieee_float(const ac::bfloat16 &f)`	Constructor from ac::bfloat16
`explicit ac_std_float(const ac_float<W-E+1,2,E,Q> &f)`	Constructor from closest equivalent ac_float (see Closest Equivalent ac_float)

`explicit ac_std_float(` `  const ac_fixed<WFX,IFX,SFX,QFX,OFX> &fx)`	Constructor from ac_fixed
`(explicit) ac_ieee_float(float f)`	Constructor from float. For binary32, the constructor is not declared explicit.
`(explicit) ac_ieee_float(double f)`	Constructor from double. For binary64, the constructor is not declared explicit.
`explicit ac_ieee_float(int x)`	Constructor from int
`explicit ac_ieee_float(long long x)`	Constructor from long long

Conversion Operators

The following conversion operators are only available for c++11 onwards. They are defined to be explicit (explicit operators introduced in c++11). They have been made explicit to avoid ambiguities for overloaded operators.

Table 31: Conversion Operators (only available for c++11 onwards)

Explicit Conversion Member Function	Description
`explicit operator float()`	Provided for binary16 and binary32. For binary32 there is not cost in terms of synthesis hardware. For binary16 there is a cost.
`explicit operator double()`	Provided for binary64. There is no cost in terms of synthesis hardware.

Explicit Conversion Methods

Table 32: Explicit Conversions

Explicit Conversion Member Function	Description
`to_ac_float()`	Conversion to closest equivalent ac_float (see Closest Equivalent ac_float)
`to_ac_std_float()`	Conversion to underlying *ac_std_float*
`to_`***ac_ieee_float***`<Format2>()`	Conversion to *ac_ieee_float*<Format2>
convert_to_ac_fixed<W,I,S,Q,O>(bool map_inf=false)	Convert to specific ac_fixed type. Values +/-Inf are mapped to +/-max() if map_inf is true; otherwise asserts. Values +/-NaN assert.
convert_to_ac_int<W,S>(bool map_inf=false)	Equivalent to: convert_to_fixed<W,W,S,AC_TRN_ZERO, AC_WRAP>(map_inf).to_ac_int()
convert_to_int(bool map_inf=false)	Equivalent to: convert_to_ac_int<32,true>(map_inf).to_int()
convert_to_int64(bool map_inf=false)	Equivalent to: convert_to_ac_int<64,true>(map_inf).to_int64()

Raw Data Access Methods

Table 33: Raw Data Access Methods

Explicit Conversion Member Function	Description
`data_ac_int()`	Returns raw data as ac_int<width,true>
`data()`	Returns raw data as a const reference to data_t which is implementation dependent: • 16 bit: short • 32 bit: int or float (if *AC_IEEE_FLOAT*_USE_BUILTIN is defined) • 64 bit: long long or double (if *AC_IEEE_FLOAT*_USE_BUILTIN is defined) • 128 bit: long long[2] • 256 bit: long long[4]
set_data()	Returns raw data as ac_std_float or ac_ieee_float type depending on what type you want to set it to

Arithmetic Member Functions

Table 34 lists the arithmetic member functions that are provided to have more control over both the rounding and whether subnormal numbers are flushed to zero in order to reduce its complexity.

Table 34: Arithmetic Member Functions

	Returns	Description
`add<QR,No_SubNormals,Effective_Add>(` `  const ac_ieee_float &op2)`	`*this + op2`	If No_SubNormals == true then subnormal operands/result are flushed to zero
`sub<QR,No_SubNormals>(` `  const ac_ieee_float &op2)`	`*this - op2`	QR is rounding mode. Currently
`mult<QR,No_SubNormals>(` `  const ac_ieee_float &op2)`	`*this * op2`	only supported modes are the symmetric rounding modes in ac_fixed:
`div<QR,No_SubNormals>(` `  const ac_ieee_float &op2)`	`*this / op2`	AC_RND_CONV, AC_TRN_ZERO,

`fma<QR,No_SubNormals>(`
`  const ac_ieee_float &op2,`
`  const ac_ieee_float &op3)`
	`*this * op2 + op3`	AC_RND_INF, AC_RND_CONV_ODD

`sqrt<QR,No_SubNormals>()`	`sqrt(*this)`	When Effective_Add == true then the function is explicitly designed only for effective additions. Default value: false.

An effective addition refers to an addition between two positive or two negative values. When the template parameter Effective_Add == true, the function is simplified, and the architecture is optimized.

Currently the only rounding modes that are supported are the symmetric versions defined for ac_fixed:

AC_RND_CONV, AC_TRN_ZERO, AC_RND_INF, AC_RND_CONV_ODD. In terms of IEEE rounding modes they correspond to roundTiesToEven (AC_RND_CONV) and roundTowardZero (AC_TRN_ZERO). The remaining IEEE modes *roundTowardPositive* and *roundTowardNegative* are asymmetric and are currently not supported.

Arithmetic Binary Operators

The arithmetic binary operators are shown in Table 35. There are no overloaded operators defined between *ac_ieee_float*<Format> and other C++ integer and floating-point types. However, the operators provided can be used for float for *ac_ieee_float*<binary32> since there is a constructor from float. Note that other C++ types don't work with *ac_ieee_float*<binary32> because the constructor from float is a template function (uses the c++ technique SFINAE to provide explicit or not explicit constructors for different Format template argument). Because it is template constructor it is not available for operator overloading that would enable for example the int->float->*ac_ieee_float*<binary32> conversion sequence. For example:

```
ac_ieee_float32 x(2.0);   // OK, explicit constructor from double available
ac_ieee_float32 y = x * 2.0;  // Error: no matching operator available
y = x * 2.0f;  // OK, float is converted to ac_ieee_float32
y = x * 2;  // Error: not matching operator available
```

Likewise, for *ac_ieee_float*<binary64>, the operators provided can be used for double since there is a (non-explicit) constructor from double. For example:

```
ac_ieee_float64 x(2.0f);   // OK, explicit constructor from float available
ac_ieee_float64 y = x * 2.0f;  // Error: no matching operator available
y = x * 2.0;  // OK, double is converted to ac_ieee_float64
y = x * 2;  // Error: not matching operator available
```

For the other formats: *binary16*, *binary128* and *binary256*, explicit constructors need to be used for C++ integer and floating-point types. For example:

```
ac_ieee_float16 x(2.0);   // OK, explicit constructor from double available
ac_ieee_float16 y = x * 2.0; // Error: no matching operator available
y = x * ac_ieee_float16(2.0); // OK, explicit constructor from double available
```

Table 35: Arithmetic Binary Operators

Binary Operator	Description
+	All operators use default rounding AC_RND_CONV (IEEE: roundTiesToEven). Support for subnormals.
-	
*	
/	

Arithmetic Assign Operators

The arithmetic assign operators are shown in Table 36.

Table 36: Arithmetic Assign Operators

Arithmetic Assign Operator	Description
+=	All operators use default rounding AC_RND_CONV (IEEE: roundTiesToEven). Support for subnormals.
-=	
*=	
/=	

Relational Operators

The relational operators are shown in Table 37. If either operand is NaN, all operands other than != return false.

Table 37: Relational Operators

Relational Operator	Description
==	Returns false if either operand is NaN
!=	Returns true if either operand is NaN
<	Returns false if either operand is NaN
>=	Returns false if either operand is NaN
>	Returns false if either operand is NaN
<=	Returns false if either operand is NaN

Unary Operators

The unary operators are shown in Table 38.

Table 38: Unary Operators

Unary Operator	Description
!	Equivalent to x == 0
+	Unary plus.
−	Unary minus. Equivalent to 0 - x

Member Functions

The member functions in Table 39 are variations of global functions provided for float and double in *math.h*.

Table 39: Member Functions

Member Function	Description
`x.abs()`	Returns absolute value of x
`x.copysign(const` *ac_ieee_float* `&op2)`	Return value whose absolute value matches that of x, but whose sign bit matches that of op2.

Member Functions to Query Type of Value

The functions in Table 40 are member function variations of the global functions provided for float and double by *math.h* under the std namespace to query the type of value.

Table 40: Member Functions to Query Type of Value

Member Function	Description
`x.fpclassify()`	Same as provided by *math.h*. Returns FP_NAN: if x is not a number FP_INFINITE: if x is +Inf or -Inf FP_ZERO: if x is 0 FP_SUBNORMAL: if x is not zero, but smaller than smallest normal value FP_NORMAL: if none of the above
`x.isfinite()`	x.fpclassify() != FP_NAN && x.fpclassify() != FP_INFINITE
`x.isnormal()`	x.fpclassify() == FP_NORMAL
`x.isnan()`	x.fpclassify() == FP_NAN
`x.isinf()`	x.fpclassify() == FP_INFINITE

Operator to Print

The ostream operator << is defined:

```
std::ostream& operator << (std::ostream &os, const ac_ieee_float<Format> &x)
```

For formats binary128 and binary256, the operator prints the raw data by first converting to ac_int. This functionality might change in the a future release.

For formats binary16 and binary32, it first gets converted to float and then printed. For format binary64, it first gets converted to double and then printed.

Static Data Members

Table 41: Static Data Members

Static Data Member	Description
`width`	Overall width; 16, 32, 64, 128, 256
`e_width`	Exponent width: 5, 8, 11, 15, 19
`mant_bits`	W-E-1
`exp_bias`	2^{E-1-1}
`min_exp`	$-2^{E-1} + 2$
`max_exp`	2^{E-1-1}

Static Member Functions

The following static member functions are consistent with the definitions used by the C++ standard for *std::numeric_limits*.

Table 42: Static Member Functions corresponding to std::numeric_limits

Static Member Function	Value Returned
`nan()`	not-a-number
`inf()`	+Inf
`denorm_min()`	Smallest positive non zero value (subnormal)
`min()`	Smallest Normal positive non zero value
`max()`	Largest positive finite value
`epsilon()`	Difference between 1.0 and the next value representable by the floating-point type

The following static member functions are provided for efficiency to return the *ac_std_float* for 0 and 1.

Table 43: Static Member Functions for 0 and 1

Static Member Function	Value Returned
`zero()`	0
`one()`	1

Functions under std namespace

The following tables specify the functions available under the *std* namespace.

Table 44: Functions under std namespace for querying the value type

Function	Description
`std::fpclassify(x)`	Same as provided by *math.h*. Returns FP_NAN: if x is not a number FP_INFINITE: if x is +Inf or -Inf FP_ZERO: if x is 0 FP_SUBNORMAL: if x is not zero, but smaller than smallest normal value FP_NORMAL: if none of the above
`std::isfinite(x)`	std::fpclassify(x) != FP_NAN && std::fpclassify(x) != FP_INFINITE
`std::isnormal(x)`	std::fpclassify(x) == FP_NORMAL
`std::isnan(x)`	std::fpclassify(x) == FP_NAN
`std::isinf(x)`	std::fpclassify(x) == FP_INFINITE

Table 45: Other functions under the std namespace

Function	Description
`std::abs(const ac_ieee_float<Format> &x)`	Returns absolute value of x
`std::fabsf(const ac_ieee_float<binary32> &x)`	32-bit version for compatibility with *math.h* fabsf(float)
`std::fabs(const ac_ieee_float<binary64> &x)`	64-bit version for compatibility with *math.h* fabs(double)

`std::copysign(`
`    const ac_ieee_float<Format> &x,`
`    const ac_ieee_float<Format> &y)`
	Returns value of x with sign of y.

`std::copysignf(`
`    const ac_ieee_float<binary32> &x,`
`    const ac_ieee_float<binary32> &y)`
	Returns value of x with sign of y. For compatibility with *math.h* copysignf(float, float)

`std::signbit(x)`	Returns sign bit. Note that signbit(-0.0) returns true.

4.2. Brain Floating-point from Google: ac::bfloat16

The class *ac::bfloat16* provides support for the *tensorflow::bfloat16* from Google. The two types could be used interchangeably, but it is important to set the rounding mode to be towards zero when using the tensorflow version:

```
assert(!std::fesetround(FE_TOWARDZERO));
```

Most operators and methods are the same as for *ac_std_float* (see General Standard Floating-point: ac_std_float) and *ac_ieee_float* (see IEEE Floating-point: ac_ieee_float).

A subset of the *math.h* functions that are defined for *float* and *double* are defined for *ac::bfloat16* under the *std* namespace and member functions variations are also provided.

Constructors

Any constructor other than the copy constructor, the default constructor, the constructor from the underlying *ac_std_float*<16,8> and the constructor from float is declared explicit. They are therefore not available in assignments (the assignment operator is not defined) or for implicit conversions. The constructors that are declared explicit have some cost in terms of runtime and hardware if targeted for synthesis. Note that the constructor from float is trivial because the exponent with is the same and the mantissa is truncated.

Table 46: Constructors

Constructor	Description
`bfloat16()`	Does not initialize value
	Copy constructor
`bfloat16(const ac_std_float<16,8> &op)`	From underlying *ac_std_float*<16,8>.
`explicit`	From *ac_std_float* with E=8, W2 != 16

`bfloat16(const ac_std_float<W2,8> &f)`	
`explicit` `bfloat16(const ac_std_float<W2,E2> &f)`	From *ac_std_float* with W2 != 16, E2 != 8
`explicit` `bfloat16(const ac_ieee_float<Format> &f)`	Constructor from *ac_ieee_float*<Format>
`explicit` `ac_std_float(const ac_float<W-E+1,2,E,Q> &f)`	Constructor from closest equivalent ac_float (see Closest Equivalent ac_float)
`explicit bfloat16(` `   const ac_fixed<WFX,IFX,SFX,QFX,OFX> &fx)`	Constructor from ac_fixed
`bfloat16(float f)`	Constructor from float

Conversion Operators

The only conversion operator is explicit operator *float()* and it is only provided for c++11 onwards. This has no cost in terms of hardware that is required to implemented it in synthesis since the exponent and mantissa of a *bfloat16* are already aligned with that of a float type.

Explicit Conversion Methods

Table 47: Explicit Conversions

Explicit Conversion Member Function	Description
`to_ac_float()`	Conversion to closest equivalent ac_float (see Closest Equivalent ac_float)
`to_ac_std_float()`	Conversion to underlying *ac_std_float*
`to_ac_ieee_float<Format2>()`	Conversion to *ac_ieee_float*<Format2>
`to_float()`	Conversion to float
`to_double()`	Conversion to double
convert_to_ac_fixed<W,I,S,Q,O>(bool map_inf=false)	Convert to specific ac_fixed type. Values +/-Inf are mapped to +/-max() if map_inf is true; otherwise asserts. Values +/-NaN assert.
convert_to_ac_int<W,S>(bool map_inf=false)	Equivalent to: convert_to_fixed<W,W,S,AC_TRN_ZERO, AC_WRAP>(map_inf).to_ac_int()
convert_to_int(bool map_inf=false)	Equivalent to: convert_to_ac_int<32,true>(map_inf).to_int()
convert_to_int64(bool map_inf=false)	Equivalent to: convert_to_ac_int<64,true>(map_inf).to_int64()

Raw Data Access Methods

Table 48: Raw Data Access Methods

Explicit Conversion Member Function	Description
`data_ac_int()`	Returns raw data as ac_int<16,true>
`data()`	Returns raw data as a const reference to short
`set_data()`	Returns raw data as ac_std_float or ac_ieee_float

		type

Arithmetic Member Functions

Table 49 lists the arithmetic member functions that are provided to have more control over both the rounding (provided as a function template parameter) and whether subnormal numbers are flushed to zero in order to reduce its complexity.

They are provided for consistency with *ac_ieee_float* and *ac_std_float*. They are not part of *tensorflow:bfloat16*.

Table 49: Arithmetic Member Functions

	Returns	Description
`add<QR,No_SubNormals,Effective_Add>(` `  const bfloat16 &op2)`	`*this + op2`	If No_SubNormals == true then subnormal operands/result are flushed to zero
`sub<QR,No_SubNormals>(` `  const bfloat16 &op2)`	`*this - op2`	
`mult<QR,No_SubNormals>(` `  const bfloat16 &op2)`	`*this * op2`	QR is rounding mode. Currently only supported modes are the symmetric rounding modes in ac_fixed:
`div<QR,No_SubNormals>(` `  const bfloat16 &op2)`	`*this / op2`	AC_RND_CONV, AC_TRN_ZERO, AC_RND_INF, AC_RND_CONV_ODD

`fma<QR,No_SubNormals>(`
 `  const bfloat16 &op2,`
 `  const bfloat16 &op3)`
	`*this * op2 + op3`	

`sqrt<QR,No_SubNormals>()`	`sqrt(*this)`	When Effective_Add == true then the function is explicitly designed only for effective additions. Default value: false

An effective addition refers to an addition between two positive or two negative values. When the template parameter Effective_Add == true, the function is simplified, and the architecture is optimized.

Currently the only rounding modes that are supported are the symmetric versions defined for *ac_fixed*: AC_RND_CONV, AC_TRN_ZERO, AC_RND_INF, AC_RND_CONV_ODD. In terms of IEEE rounding modes they correspond to *roundTiesToEven* (AC_RND_CONV) and *roundTowardZero* (AC_TRN_ZERO). The remaining IEEE modes *roundTowardPositive* and *roundTowardNegative* are asymmetric and are currently not supported.

Arithmetic Binary Operators

The arithmetic binary operators are given in Table 50. There are no overloaded operators defined between *ac::bfloat16* and other C++ integer and floating-point types. However, the operators provided can be used by C++ types since there is a constructor from float. For example:

```
ac::bfloat16 x = 2.0;    // double is converted to float and then to bfloat16
ac::bfloat16 y = x * 3;  //  integer is converted to float and then to bfloat16
y = x + 5.0;  // double is converted to float and then to bfloat16
```

Table 50: Arithmetic Binary Operators

Binary Operator	Description
+	All operators use default rounding AC_TRN_ZERO (IEEE: roundTowardZero). Support for subnormals.
–	
*	
/	

Arithmetic Assign Operators

The arithmetic assign operators are shown in Table 51. There are no overloaded operators defined between ac::bfloat16 and other C++ integer and floating-point types. However, the operators provided can be used by C++ types since there is a constructor from float. For example:

```
ac::bfloat16 x = 2.0;     // double is converted to float and then to bfloat16
x *= 3;  //  integer is converted to float and then to bfloat16
x += 5.0;  // double is converted to float and then to bfloat16
```

Table 51: Arithmetic Assign Operators

Arithmetic Assign Operator	Description
+=	All operators use default rounding AC_TRN_ZERO (IEEE: roundTowardZero). Support for subnormals.
–=	
*=	
/=	

Relational Operators

The relational operators are shown in Table 52. If either operand is NaN, all operands other than != return false. There are no overloaded operators defined between ac::bfloat16 and other C++ integer and floating-point types. However, the operators provided can be used by C++ types since there is a constructor from float. For example:

```
ac::bfloat16 x = ...;
if(x == 3) { ...}  //  integer is converted to float and then to bfloat16
bool flag = x == 5.0;  // double is converted to float and then to bfloat16
```

Table 52: Relational Operators

Relational Operator	Description
==	Returns false if either operand is NaN
!=	Returns true if either operand is NaN
<	Returns false if either operand is NaN
>=	Returns false if either operand is NaN
>	Returns false if either operand is NaN
<=	Returns false if either operand is NaN

Unary Operators

The unary operators are shown in Table 53.

Table 53: Unary Operators

Unary Operator	Description
!	Equivalent to x == 0
+	Unary plus.
−	Unary minus. Equivalent to 0 - x

Member Functions

The member functions in Table 54 are variations of global functions provided for float and double in *math.h*. The are also defined for *tensorflow::bfloat16*.

Table 54: Member Functions

Member Function	Description
`x.abs()`	Returns absolute value of x
`x.copysign(const bfloat16 &op2)`	Return value whose absolute value matches that of x, but whose sign bit matches that of op2.

Member Functions to Query Type of Value

The functions in Table 55 are member function variations of the global functions provided for float and double by *math.h* under the std namespace to query the type of value. They are not provided by *tensorflow::bfloat16*, but are provided in ac::bfloat16 for consistency with *ac_ieee_float* and *ac_std_float*. The tensorflow version of *bfloat16* does provide several of the global functions under the *std* namespace as described in Section .

Table 55: Member Functions to Query Type of Value

Member Function	Description
`x.fpclassify()`	Same as provided by *math.h*. Returns FP_NAN: if x is not a number FP_INFINITE: if x is +Inf or -Inf FP_ZERO: if x is 0 FP_SUBNORMAL: if x is not zero, but smaller than smallest normal value FP_NORMAL: if none of the above
`x.isfinite()`	x.fpclassify() != FP_NAN && x.fpclassify() != FP_INFINITE
`x.isnormal()`	x.fpclassify() == FP_NORMAL
`x.isnan()`	x.fpclassify() == FP_NAN
`x.isinf()`	x.fpclassify() == FP_INFINITE

Operator to Print

The ostream << operator is defined and prints the value after converting the bfloat16 to float:

```
std::ostream& operator << (std::ostream &os, const ac::bfloat16 &x)
```

Static Data Members

Table 56: Static Data Members

Static Data Member	Description
width	Overall width = 16
e_width	Exponent width = 8
mant_bits	W-E-1 = 7
exp_bias	$2^{E-1-1} = 127$
min_exp	$-2E-1 + 2 = -126$
max_exp	$2^{E-1-1} = 127$

Static Member Functions

The following static member functions are consistent with the definitions used by the C++ standard for *std::numeric_limits*.

Table 57: Static Member Functions corresponding to std::numeric_limits

Static Member Function	Value Returned
nan()	not-a-number
inf()	+Inf
denorm_min()	Smallest positive non zero value (subnormal)
min()	Smallest Normal positive non zero value
max()	Largest positive finite value
epsilon()	Difference between 1.0 and the next value representable by bfloat16

The following static member functions are provided for efficiency to return the *ac_std_float* for 0 and 1.

Table 58: Static Member Functions for 0 and 1

Static Member Function	Value Returned
zero()	0
one()	1

Functions under std namespace

The following tables specify the functions available under the *std* namespace. They are consistent with the versions provided by *math.h* for *float* and *double*. The *tensorflow::bfloat16* provides *std::isinf, std::isnan* and *std::isinf*. The *ac:bfloat16* version provides the full set for consistency with *ac_ieee_float* and *ac_std_float*.

Table 59: Functions in std Namespace to Query Type of Value

Function	Description
std::fpclassify(x)	Same as provided by *math.h*. Returns FP_NAN: if x is not a number FP_INFINITE: if x is +Inf or -Inf

	FP_ZERO: if x is 0 FP_SUBNORMAL: if x is not zero, but smaller than smallest normal value FP_NORMAL: if none of the above
`std::isfinite(x)`	std::fpclassify(x) != FP_NAN && std::fpclassify(x) != FP_INFINITE
`std::isnormal(x)`	std::fpclassify(x) == FP_NORMAL
`std::isnan(x)`	std::fpclassify(x) == FP_NAN
`std::isinf(x)`	std::fpclassify(x) == FP_INFINITE

Table 60: Other Funtions in the std namespace

Function	Description
`std::abs(const bfloat16 &x)`	Returns absolute value of x
`std::copysign(const bfloat16 &x, const bfloat16 &y)`	Returns value of x with sign of y.
`std::signbit(const bfloat16 &x)`	Returns sign bit. Note that signbit(-0.0) returns true.

4.3. General Standard Floating-point: ac_std_float

The general standard *ac_std_float*<W,E> has overall width W and exponent width E. It is used to implement the *ac_ieee_float* and bfloat16 types. It also provides for conversions to and from ac_float.

Most operators and methods are the same as for *ac_ieee_float* (see [IEEE Floating-point: ac_ieee_float](#)) and *ac::bfloat16* (see [Brain Floating-point from Google: ac::bfloat16](#)).

A subset of the *math.h* functions that are defined for float and double are defined for ac::bfloat16 under the std namespace and member functions variations are also provided.

Constructors

Any constructor other than the copy constructor and the default constructor is declared explicit. They are therefore not available in assignments (the assignment operator is not defined) or for implicit conversions. The constructors that are declared explicit have some cost in terms of runtime and hardware if targeted for synthesis.

Table 61: Constructors

Constructor	Description
`ac_std_float()`	Does not initialize value
`ac_std_float(const ac_std_float &f)`	Copy constructor
`explicit` `ac_std_float(const ac_std_float<W2,E> &f)`	Constructor from *ac_std_float* with same E, but possibly different W2
`explicit` `ac_std_float(const ac_std_float<W2,E2> &f)`	Constructor from *ac_std_float* with possibly different W2 and E2
`explicit`	Constructor from closest equivalent ac_float

`ac_std_float(const ac_float<W-E+1,2,E,Q> &f)`	(see [Closest Equivalent ac_float](#))
`explicit` `ac_std_float(const ac::bfloat16 &f)`	Constructor from ac::bfloat16
`explicit` `ac_std_float(const ac_ieee_float<Format> &f)`	Constructor from *ac_ieee_float*
`explicit ac_std_float(` `   const ac_fixed<WFX,IFX,SFX,QFX,OFX> &fx)`	Constructor from ac_fixed
`explicit ac_std_float(float f)`	Constructor from float
`explicit ac_std_float(double f)`	Constructor from double
`explicit ac_std_float(int x)`	Constructor from int
`explicit ac_std_float(long long x)`	Constructor from long long

Explicit Conversion Methods

Table 62: Explicit Conversions

Explicit Conversion Member Function	Description
`to_ac_float()`	Conversion to closest equivalent ac_float (see [Closest Equivalent ac_float](#))
`to_float()`	Conversion to float
`to_double()`	Conversion to double
`convert_to_ac_fixed<W,I,S,Q,O>(` `   bool map_inf=false)`	Convert to specific ac_fixed type. Values +/-Inf are mapped to +/-max() if map_inf is true; otherwise asserts. Values +/-NaN assert.
`convert_to_ac_int<W,S>(bool map_inf=false)`	Equivalent to: convert_to_fixed<W,W,S,AC_TRN_ZERO, AC_WRAP>(map_inf).to_ac_int()
`convert_to_int(bool map_inf=false)`	Equivalent to: convert_to_ac_int<32,true>(map_inf).to_int()
`convert_to_int64(bool map_inf=false)`	Equivalent to: convert_to_ac_int<64,true>(map_inf).to_int64()

Arithmetic Member Functions

Table 63 lists the arithmetic member functions that are provided to have more control over both the rounding (provided as a function template parameter) and whether subnormal numbers are flushed to zero in order to reduce its complexity.

Table 63: Arithmetic Member Functions

Member Function	Returns	Description
`add<QR,No_SubNormals,Effective_Add>(` `   const ac_std_float &op2)`	`*this + op2`	If No_SubNormals == true then subnormal operands/result are flushed to zero
`sub<QR,No_SubNormals>(` `   const ac_std_float &op2)`	`*this - op2`	

mult<QR,No_SubNormals>(const ac_std_float &op2)	*this * op2	QR is rounding mode. Currently only supported modes are the symmetric rounding modes in ac_fixed: AC_RND_CONV, AC_TRN_ZERO, AC_RND_INF, AC_RND_CONV_ODD When Effective_Add == true then the function is explicitly designed only for effective additions. Default value: false.
div<QR,No_SubNormals>(const ac_std_float &op2)	*this / op2	

fma<QR,No_SubNormals>(
  const ac_std_float &op2,
  const ac_std_float &op3)
	*this * op2 + op3	

sqrt<QR,No_SubNormals>()	sqrt(*this)	

An effective addition refers to an addition between two positive or two negative values. When the template parameter Effective_Add == true, the function is simplified, and the architecture is optimized.

Currently the only rounding modes that are supported are the symmetric versions defined for ac_fixed: AC_RND_CONV, AC_TRN_ZERO, AC_RND_INF, AC_RND_CONV_ODD. In terms of IEEE rounding modes they correspond to roundTiesToEven (AC_RND_CONV) and roundTowardZero (AC_TRN_ZERO). The remaining IEEE modes roundTowardPositive and roundTowardNegative are asymmetric and are currently not supported.

The methods in Table 62 call by default the core implementation methods with "_generic" in their name unless a macro to override is defined as described in Section 4.4.1. . The methods are only available for class ac_std_float and are enumerated in Table 64.

Table 64: Arithmetic Member Functions for Generic Implementation

Member Function	Description
add_generic<QR,No_SubNormals>(const ac_std_float &op2)	Used by add (and indirectly) by sub unless AC_STD_FLOAT_ADD_OVERRIDE is defined
mult_generic<QR,No_SubNormals>(const ac_std_float &op2)	Used by mult unless AC_STD_FLOAT_MULT_OVERRIDE is defined
div_generic<QR,No_SubNormals>(const ac_std_float &op2)	Used by div unless AC_STD_FLOAT_DIV_OVERRIDE is defined

fma_generic<QR,No_SubNormals>(
  const ac_std_float &op2,
  const ac_std_float &op3)
	Used by fma unless AC_STD_FLOAT_FMA_OVERRIDE is defined

sqrt_generic<QR,No_SubNormals>()	Used by sqrt unless AC_STD_FLOAT_SQRT_OVERRIDE is defined

Arithmetic Binary Operators

Table 65: Arithmetic Binary Operators

Binary Operator	Description
+	All operators use default rounding AC_RND_CONV (IEEE: roundTiesToEven). Support for subnormals.
–	
*	
/	

Arithmetic Assign Operators

Table 66: Arithmetic Assign Operators

Arithmetic Assign Operator	Description
+=	All operators use default rounding AC_RND_CONV (IEEE: roundTiesToEven). Support for subnormals.
–=	
*=	
/=	

Relational Operators

Table 67: Relational Operators

Relational Operator	Description
==	Returns false if either operand is NaN
!=	Returns true if either operand is NaN
<	Returns false if either operand is NaN
>=	Returns false if either operand is NaN
>	Returns false if either operand is NaN
<=	Returns false if either operand is NaN

Unary Operators

The unary operators are shown in Table 68.

Table 68: Unary Operators

Unary Operator	Description
!	Equivalent to x == 0
+	Unary plus.
–	Unary minus. Equivalent to 0 - x

Member Functions

The member functions in Table 69 are variations of global functions provided for float and double in *math.h*.

Table 69: Member Functions

Member Function	Description
x.abs()	Returns absolute value of x
x.copysign(const ac_std_float &op2)	Return value whose absolute value matches that of x, but whose sign bit matches that of op2.

Member Function to Query Type of Value

The functions in Table 70 are member function variations of the global functions provided for float and double

by *math.h* under the std namespace to query the type of value.

Table 70: Member Function to Query Type of Value

Member Function	Description
`x.fpclassify()`	Same as provided by *math.h*. Returns FP_NAN: if x is not a number FP_INFINITE: if x is +Inf or -Inf FP_ZERO: if x is 0 FP_SUBNORMAL: if x is not zero, but smaller than smallest normal value FP_NORMAL: if none of the above
`x.isfinite()`	x.fpclassify() != FP_NAN && fpclassify(x) != FP_INFINITE
`x.isnormal()`	x.fpclassify() == FP_NORMAL
`x.isnan()`	x.fpclassify() == FP_NAN
`x.isinf()`	x.fpclassify() == FP_INFINITE

Operator to Print

The ostream operator << is defined:

```
std::ostream& operator << (std::ostream &os, const ac_std_float<W,E> &x)
```

The operator prints the raw ac_int<W,true> data. This functionality might change in the a future release.

Static Data Members

Table 71: Static Data Members

Static Data Member	Description
`width`	Overall width. Same as W template parameter
`e_width`	Exponent width. Same as E template parameter
`mant_bits`	W-E-1
`exp_bias`	2^{E-1-1}
`min_exp`	$-2^{E-1} + 2$
`max_exp`	2^{E-1-1}

Static Member Functions

The following static member functions are consistent with the definitions used by the C++ standard for *std::numeric_limits*.

Table 72: Static Member Functions corresponding to std::numeric_limits

Static Member Function	Value Returned
`nan()`	not-a-number
`inf()`	+Inf
`denorm_min()`	Smallest positive non zero value (subnormal)

`min()`	Smallest Normal positive non zero value
`max()`	Largest positive finite value
`epsilon()`	Difference between 1.0 and the next value representable by the floating-point type

The following static member functions are provided for efficiency to return the *ac_std_float* for 0 and 1.

Table 73: Static Member Functions for 0 and 1

Static Member Function	Value Returned
`zero()`	0
`one()`	1

Closest Equivalent ac_float

The closest equivalent to *ac_std_float<W,E>* is *ac_float<W-E+1, 2, E, AC_RND_CONV>* since:

1. The width argument (W) for *ac_float<W,I,E,Q>* stands for the width of the *ac_fixed* used as a mantissa, while the width argument for (W) *ac_std_float<W,E>*. It captures signedness, but since it does not use an implied bit it requires an additional bit.

2. The *I* is the number of integer bits of the *ac_fixed* used as a mantissa. It is encoded in two's complement so two bits are required to capture +1.x and -1.x.

3. The exponent width (*E*) is the same width in both cases, but in *ac_std_float<W,E>* the all 0's and all 1's exponents are special so it will become subnormal one exponent earlier and it will saturate to infinite with the highest value of *E*.

4. The AC_RND_CONV rounding mode is default for *ac_std_float*.

4.4. Overriding with alternative implementations

The following list shows the available override mechanisms:

- Override the "fixed-point" division that is used inside the implementations of the floating-point division functions.

- Override any of the arithmetic methods in ac_std_float<W,E> to use an alternative implementation. This affects the corresponding operator. For instance, if method div is overridden, the operator "/" will also be affected.

- Override the default rounding mode of the standard floating-point arithmetic operators '+', '-', '*' and '/'.

NOTE: These overrides will affect the methods and corresponding operators for all types: ac_std_float, ac_ieee_float and ac::bfloat16.

Overriding core division function

The fixed-point division used in the implementation of the floating-point division currently relies on the ac::fx_div function for synthesis and the ac::fx_div_sim for simulation. The ac::fx_div function is based on a non restoring algorithm that computes one bit of the quotient for every iteration of a loop. The loop has W-E+2 iterations. For example, for ac_float_ieee32, the loop has 26 iterations. The implementation leverages the property that the second operand has '1' as the MSB since it is normalized. Synthesis directives may be changed on this function according to the desired performance.

The ac::fx_div_sim uses both the '/' for the quotient and '%' to determine if the division is exact (no remainder) and is used for simulation since it can leverage the integer operations in the processor.

Users may want to provide alternative C++ implementations that provide more optimized synthesis results. The macro needs to be defined before the inclusion of ac_std_float.h to override which fixed-point division function is called:

```
template<int W>
void my_fix_div(ac_int<W,false> op1, ac_int<W,false> op2, ac_int<W+2,false>
&quotient, bool &exact);
#define AC_STD_FLOAT_FX_DIV_OVERRIDE my_fix_div
#include <ac_std_float.h>
```

The quotient argument that is returned is the result of op1*2W+1/op2 and the argument exact is true if the division was exact (remainder is zero). The MSB of op2 is assumed to be '1'.

Overriding the floating-point arithmetic methods

The arithmetic methods: add, sub, mult, div, fma and sqrt can have their implementations overridden with the macros in Table 74. Note that overriding the add also affects the sub method so there is no separate override for sub.

The type of the argument and return type are all assumed to be ac_std_float<W,E>. The template parameters are:

```
template<ac_q_mode QR, bool No_SubNormals, int W, int E>
```

Table 74: Macros for overriding arithmetic methods

Macro for function name	Number of arguments ac_std_float<W,E>
AC_STD_FLOAT_ADD_OVERRIDE	2
AC_STD_FLOAT_MULT_OVERRIDE	2
AC_STD_FLOAT_DIV_OVERRIDE	2
AC_STD_FLOAT_FMA_OVERRIDE	3
AC_STD_FLOAT_SQRT_OVERRIDE	1

A namespace for those functions can be specified with the macro:

```
AC_STD_FLOAT_OVERRIDE_NAMESPACE
```

Defining any of the macros for the function override will declare the template function with that name specified in the macro in ac_std_float.h so that they are available in the implementations of the various methods for class ac_std_float. The definitions of the correspond function is expected to be provided after inclusion of ac_std_float.h if the implementation of the version in ac_std_float.h is required: for instance if the override is applied for some combinations of template parameters. An example is show next.

Assume that some alternative implementations are provided in a header alt_impls.h that contains:

```
#pragma once
// OVERRIDE MACROs needs to be defined before inclusion of ac_std_float.h
//   Actual implementation of override functions defined after inclusion of
ac_std_float.h

#include <ac_std_float.h>

// definitions
namespace my_override_tech1 {
 template<ac_q_mode QR, bool No_SubNormals, int W, int E>
 inline ac_std_float<W,E> add(const ac_std_float<W,E> &op,
                             const ac_std_float<W,E> &op2) { ... }
 template<ac_q_mode QR, bool No_SubNormals, int W, int E>
 inline ac_std_float<W,E> mult(const ac_std_float<W,E> &op,
                              const ac_std_float<W,E> &op2) { ... }
 template<ac_q_mode QR, bool No_SubNormals, int W, int E>
 inline ac_std_float<W,E> div(const ac_std_float<W,E> &op,
                             const ac_std_float<W,E> &op2) { ... }
 template<ac_q_mode QR, bool No_SubNormals, int W, int E>
 inline ac_std_float<W,E> fma(const ac_std_float<W,E> &op,
                             const ac_std_float<W,E> &op2,
                             const ac_std_float<W,E> &op3) { ... }
 template<ac_q_mode QR, bool No_SubNormals, int W, int E>
 inline ac_std_float<W,E> sqrt(const ac_std_float<W,E> &op) { ... }
}
```

Then one can select any of them to be used as an override:

```
// OVERRIDE MACROs needs to be defined before inclusion of ac_std_float.h
#ifdef __AC_STD_FLOAT_H
#error "file ac_std_float.h already included, overrides below will have no ef-
fect
#endif

#define AC_STD_FLOAT_MULT_OVERRIDE mult
#define AC_STD_FLOAT_OVERRIDE_NAMESPACE my_override_tech1
```

```
#include "alt_impls.h"

int main() {
  ac_ieee_float32 a = ...;
  ac_ieee_float32 b = ...;
  ac_ieee_float32 c = a * b;   // override will apply
  c = a.mult<AC_TRN_ZERO,true>(b);  // override will apply

  ac_ieee_float32 d = a + b;  // no override was chosen
}
```

In the definitions in alt_impls.h, the override could be applied to specific combinations of template parameters using template specialization:

```
namespace my_override_tech1 {
  template<ac_q_mode QR, bool No_SubNormals, int W, int E>
  inline ac_std_float<W,E> mult(const ac_std_float<W,E> &op, const ac_st-
d_float<W,E> &op2) {
    // use generic implementation provided in ac_std_float for default case
    return op.template mult_generic<QR,No_SubNormals>(op2);
  }
  // specialization
  template<> mult<AC_RND_CONV,false,32,8>(const ac_std_float<W,E> &op, const
ac_std_float<W,E> &op2) {
    ...
  }
```

Overriding the default rounding mode of standard floating-point arithmetic operators

The four basic arithmetic operators ('+', '-', '*', '/') for the three standard floating-point datatypes ac_std_float, ac_ieee_float and ac::bfloat16 are implemented with different default rounding modes, depending on the datatype. For the general standard floating-point (ac_std_float) and the ac_ieee_float datatypes, the default rounding mode is AC_RND_CONV, which is equivalent to the IEEE rounding mode roundTiesToEven. For the ac::bfloat16 datatype the default rounding mode is AC_TRN_ZERO, which is equivalent to the IEEE rounding mode roundTowardZero.

The AC_STD_FLOAT_IEEE754_ROUND_OVERRIDE and AC_STD_FLOAT_BFLOAT16_ROUND_OVERRIDE macros can be used to override the default rounding mode for the arithmetic operators. By changing the default rounding mode of the '+', '-', '*' and '/', the operators '+=', '-=', '*=' and '/=' are also affected.

When these two macros are being defined, their value can be one of the rounding modes supported by the standard floating-point library. These rounding modes are AC_RND_CONV, AC_TRN_ZERO, AC_RND_INF and AC_RND_CONV_ODD. If the defined value of the macro is not one of the supported, an error message will be given.

The AC_STD_FLOAT_IEEE754_ROUND_OVERRIDE is used to override the rounding mode for the ac_std_float and ac_ieee_float operators. If this macro is not defined, the default mode for these two datatypes' operators is AC_RND_CONV. Similarly, the AC_STD_FLOAT_BFLOAT16_ROUND_OVERRIDE is used to override the default rounding mode for the ac::bfloat16 operators. If this macro is not defined, the default mode is AC_TRN_ZERO.

Example of using the default rounding mode:

```
#include <ac_std_float.h>
typedef ac_ieee_float<binary32> ieee_flt; // equivalent to ac_std_float<32,8>
typedef ac::bfloat16 bflt_flt;

int main() {
  ieee_flt a = 16.0000006;
  ieee_flt b = 6.0000014;
  ieee_flt c = a + b; // result is 22.0000076
  //int representation of the result in hex: 0x41B00004 (AC_TRN_ZERO)

  ieee_flt d = 16.375;
  ieee_flt e = 6.09375;
  bflt_flt f = d + e; // result is 22.375
  //int representation of the result in hex: 0x41B3 (AC_RND_CONV)
}
```

Example of overriding the rounding mode:

```
#define AC_STD_FLOAT_IEEE754_ROUND_OVERRIDE_AC_TRN_ZERO
#define AC_STD_FLOAT_BFLOAT16_ROUND_OVERRIDE_AC_RND_CONV

#include <ac_float.h>

typedef ac_ieee_float<binary32> ieee_flt; // equivalent to ac_std_float<32,8>
typedef ac::bfloat16 bflt_flt;

int main(){
  ieee_flt a = 16.0000006;
  ieee_flt b = 6.0000014;
  ieee_flt c = a + b; // result is 22.0000057
  //int representation of the result in hex: 0x41B00003 (AC_TRN_ZERO)

  ieee_flt d = 16.375;
  ieee_flt e = 6.09375;
  bflt_flt f = d + e; // result is 22.5
  //int representation of the result in hex: 0x41B4 (AC_RND_CONV)
}
```

Note that the macro definition should pre-exist the inclusion of the *ac_std_float.h* library.

Chapter 5:   Complex Datatype

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 w/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 75 shows the operators defined for both *ac_complex.*

Table 75: 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 76 shows the methods defined for the *ac_complex* type.

Table 76: 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&)
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*

5.1. Usage of ac_complex

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

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

The following list includes recommendations for using the ac_complex datatype:

- 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)
```

- 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.

5.2. Advanced utility functions, typedefs, etc for ac_complex

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.

5.2.1. 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.

5.2.2. 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" section. The initialization value is applied to both the real and imaginary components.

5.2.3. Return Type for Unary and Binary Operators

Refer to corresponding sections in "Arbitrary-Length, Bit-Accurate Integer and Fixed-Point Datatypes" for the basic AC Datatypes.

Chapter 6: Reference Guide for Numerical Algorithmic C Datatypes

This chapter provides a high-level view of the numerical AC Datatype package. It provides a high-level reference for the following items that are exposed to the user:

- All public functions, member functions

- Enumerations, Static Constants and Type Definitions

- Macros

Anything that is not covered in this chapter is implementation specific and is subject to change.

6.1. Functions and Operators

The tables in this section capture all the public functions, member functions and operators that are available for the different types.

6.1.1. Constructors

The following table lists all the constructors available for the AC numerical types.

Table 77: Constructors Available

Constructor	ac_int	ac_fixed	ac_float	ac_complex
Default [1]	✔	✔	✔	✔
Copy	NA	✔	✔	NA
Constructor from same type name but any parameter	✔	✔	✔ [2]	✔
Constructor from each bool and integer type	✔	✔	✔	NA [4]
Constructor from float	NA	NA	✔ [2]	NA [4]
Constructor from double	✔	✔	✔ [2]	NA [4]
Constructor from template T type	NA	NA	NA	✔
Constructor from template T1, T2 type pair	NA	NA	NA	✔

Constructor from other AC Numerical Type	NA	✔ ac_int	✔ ac_int, ac_fixed [3] 6.1.2. (ac_fixed, ac_int) [3]	NA [4]

Notes:

1. The default constructors leave the datatypes un-initialized. If the *AC_DEFAULT_IN_RANGE* macro is defined, it will be un-initialized, but guaranteed to be within range.

2. The *ac_float* constructors have additional arguments to assert on rounding or overflow.

3. The *ac_float* constructors have an additional argument to normalize.

4. The general template constructor from template *T* type covers these cases.

6.1.2. Conversions

Conversions can be classified in conversion operators and explicit conversion methods. The only operators that is available is for *ac_int* with parameter restrictions that guarantee that there is no loss of precision. The explicit conversion methods *ac_float::to_ac_fixed* does not lose precision. Otherwise all other explicit conversions may lose precision depending on the parameter values of the types.

Table 78: Conversion Operators and Methods

Operators and Methods	ac_int	ac_fixed	ac_float	ac_complex
operator long long	✔ W <= 64 S = true	NA	NA	NA
operator unsigned long long	✔ W <= 64 S = false	NA	NA	NA
to_int	✔	✔	✔	NA
to_uint	✔	✔	✔	NA
to_long	✔	✔	✔	NA
to_ulong	✔	✔	✔	NA
to_int64	✔	✔	✔	NA
to_uint64	✔	✔	✔	NA
to_float	NA	NA	✔	NA
to_double	✔	✔	✔	NA
to_ac_int	NA	✔	✔	NA
to_ac_fixed	NA	NA	✔	NA
to_string	✔	✔	✔	NA

6.1.3. Arithmetic, Relational and Shift Operators and Methods

The operators and methods covers both binary (two operand) operations and unary (single operand) operations. The binary operations have also mixed AC type and C++ integer types global operators defined for them.

Table 79: Arithmetic, Relational and Shift Operators and Methods

Operators and Methods	ac_int	ac_fixed	ac_float	ac_complex
+, -, /, +=, -=, /=	✔ [1]	✔	✔ [2,4]	✔ [5]
add, sub [2]	NA	NA	✔	NA
%, %=	✔	NA	NA	NA
==, !=	✔	✔ [3]	✔ [4]	✔ [5]
>, <, >=, <=	✔	✔ [3]	✔ [4]	NA
>>, <<, >>=, <<=	✔ [6]	✔ [6]	✔	NA
Unary +, -, ! [7]	✔	✔	✔	✔
++, --	✔	✔	NA	NA
abs	NA	NA	✔	NA
conj	NA	NA	NA	✔
mag_sqr	NA	NA	NA	✔
sign_conj	NA	NA	NA	✔

Notes:

1. The + operator is also 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. In all cases, the result is of the same type as the C pointer.

2. The *add* and *sub* methods for *ac_float* provide control over the desired return type and are alternatives for the + and – operators that are still subject to change.

3. The relational operators are defined for mixed *ac_fixed* with C++ double.

4. Operators are defined for mixed *ac_float* with with C++ *float* and *double*.

5. Operators are defined for mixed *ac_complex* with another type that is a template parameter.

6. Shift operators >> and << for *ac_int* and *ac_fixed* return the type of the first argument.

7. The unary ~ operator is covered in Logical Operators and Methods.

6.1.4. Bit and Slice Operators and Methods

The bit and slice operators and methods are defined for ac_int and ac_fixed. The slc method can read a slice

out of range.

Table 80: Bit and Slice Operators and Methods

Operators and Methods	ac_int	ac_fixed	ac_float	ac_complex
slc	✔	✔	NA	NA
set_slc	✔	✔	NA	NA
operator []	✔	✔	NA	NA

There are const and non const versions of the [] operator and the index argument can be *int*, *unsigned*, and *ac_int*. There are asserts to check that index is in range.

The *ac_int::operator []* returns an *ac_int::ac_bitref* and the *ac_fixed::operator []* returns an *ac_fixed::ac_bitref*. Both *ac_int::ac_bitref* and *ac_fixed::ac_bitref* have identical definitions as shown in Table 81. The *ac_bitref* classes are not meant to be used explicitly by the user.

Table 81: Operators defined for ac_int::ac_bitref and ac_fixed::ac_bitref

Operators	Description
operator bool	bit returned as bool
operator ac_int<W2, S2>	First call operator bool and then constructs ac_int
ac_bitref operator = (int val)	LSB of int is written to bit
ac_bitref operator = (const ac_bitref &val)	Assignment for ac_bitref for identical type

6.1.5. Logical Operators and Methods

The ~ operator has an arithmetic definition as compared to the bit_complement method listed in Table 82 which returns the unsigned bitwise complement of the *ac_int* or *ac_fixed*.

Table 82: Logical Operators and Methods

Operators and Methods	ac_int	ac_fixed	ac_float	ac_complex
&, |, ^, &=, |=, ^=	✔	✔	NA	NA
Unary operator ~	✔	✔	NA	NA
bit_complement	✔	✔	NA	NA
and_reduce, or_reduce, xor_reduce	✔	NA	NA	NA

6.1.6. Other Functions and Methods

The table below lists functions string related functions, filling them with bit patterns, setting the types to special values, provide functionality useful for floating-point and for accessing data members for *ac_float* and *ac_complex*. The length function returns the *W* parameter value for *ac_int* and *ac_fixed*.

Table 83: Other functions/methods

Functions and Methods	ac_int	ac_fixed	ac_float	ac_complex
std::ostream & operator <<	✔	✔	✔	✔
type_name	✔	✔	✔	✔
bit_fill	✔	✔	NA	NA
ac::bit_fill	✔	✔	NA	NA
bit_fill_hex	✔	✔	NA	NA
ac::bit_fill_hex	✔	✔	NA	NA
set_val	✔	✔	✔	NA
ac::value	✔	✔	✔	✔
ac::init_array	✔	✔	✔	✔
leading_sign	✔	✔	NA	NA
normalize	✔	✔	✔	NA
normalize_RME	✔	✔	NA	NA
mantissa, exp	NA	NA	✔	NA
set_mantissa, set_exp	NA	NA	✔	NA
i, r, real, imag	NA	NA	NA	✔
set_i, set_r	NA	NA	NA	✔
length	✔	✔	NA	NA

6.1.7. Mantissa/Exponent Extraction of float/double

The following table lists functions for extracting mantissa, exponent and sign information out of C++ *float* and *double* floating-point types.

Table 84: Functions to extract mantissa/exponent from float/double

Function	Header	Description
ac::frexp_f	ac_fixed.h	Exponent and 2's complement mantissa for float
ac::frexp_d	ac_fixed.h	Exponent and 2's complement mantissa for double
ac::frexp_sm_f	ac_fixed.h	Exponent, unsigned mantissa and sign for float
ac::frexp_sm_d	ac_fixed.h	Exponent, unsigned mantissa and sign for double

6.1.8. SystemC Tracing Functions

All numerical types have the *sc_trace* function defined. They are defined in the *ac_sc.h* header file.

6.1.9. Explicit conversions to/from SystemC Types

The explicit conversions functions to and from the SystemC datatypes *sc_bigint*, *sc_biguint*, *sc_fixed* and *sc_ufixed* are shown in Table 85. They are defined in the *ac_sc.h* header file.

Table 85: Explicit conversions to/from AC Datatype from/to SystemC Datatype

From	To	Function
sc_bigint	ac_int, S=true	ac_int<W, true> to_ac(const sc_bigint<W> &)
sc_biguint	ac_int, S=false	ac_int<W, false> to_ac(const sc_biguint<W> &)
sc_fixed	ac_fixed, S=true	ac_fixed<W,I, true> to_ac(const sc_fixed<W,I,Q,O,nbits> &)
sc_ufixed	ac_fixed, S=false	ac_fixed<W,I, false> to_ac(const sc_ufixed<W,I,Q,O,nbits> &)
ac_int, S=true	sc_bigint	sc_bigint<W> to_sc(const ac_int<W,true> &)
ac_int, S=false	sc_biguint	sc_biguint<W> to_sc(const ac_int<W,false> &)
ac_fixed, S=true	sc_fixed	sc_fixed<W,I> to_sc(const ac_fixed<W,I,true,Q,O> &
ac_fixed, S=false	sc_ufixed	sc_ufixed<W,I> to_sc(const ac_fixed<W,I,false,Q,O> &

6.2. Enumerations, Static Constants and Type Definitions

This section covers the enumerations, static constants and type definitions (typedefs) that are defined in the numerical AC header types.

6.2.1. General Enumerations

The following table defines general enumeration values and which functions uses them.

Table 86: General Enumerations

Enumeration Value	Enumeration Type	Description	Used by
AC_BIN	ac_base_mode	Binary format	to_string
AC_OCT		Octal format	
AC_DEC		Decimal format	
AC_HEX		Hexadecimal format	
AC_VAL_DC	ac_special_val	Don't Care (un-initialized)	set_val ac::value ac::init_array
AC_VAL_0		Zero	
AC_VAL_MIN		Minimum val for type	
AC_VAL_MAX		Maximum val for type	
AC_VAL_QUANTUM		Smallest increment for type	

6.2.2. Enumerations for Fixed-point Quantization and Overflow Modes

The following enumerations are used to define the quantization and overflow modes for *ac_fixed*, but are forward declared in *ac_int.h*. They are template parameters for *ac_fixed* with default settings as indicated.

Table 87: Enumerations for fixed-point quantization and overflow modes.

Enumeration Value	Enumeration Type	Description
AC_TRN (default)	ac_q_mode (Quantization)	truncate toward minus infinity
AC_RND		round towards plus infinity
AC_TRN_ZERO		truncate towards zero
AC_RND_ZERO		round towards zero
AC_RND_INF		round towards infinity
AC_RND_MIN_INF		round towards minus infinity
AC_RND_CONV		round towards even
AC_RND_CONV_ODD		round towards odd
AC_WRAP (default)	ac_o_mode (Overflow)	wrap
AC_SAT		saturate to max and min
AC_SAT_ZERO		saturate to zero
AC_SAT_SYM		saturate to +/- max

6.2.3. Static Constant Members and Type Definitions to Capture Properties of Types

The following table lists the static data members to query the type for properties that are dependent on template parameters. If a type lacks that parameter it may either depend on another parameter or be implicitly behave as having a constant value for that parameter. For example an *ac_int* behaves as an *ac_fixed* that has identical *width* and *i_width* and has *AC_TRN* and *AC_WRAP*.

Table 88: Static constant members to capture properties of types.

Static Data Member	ac_int	ac_fixed	ac_float	ac_complex
width	✔	✔	✔	NA
i_width	✔ width	✔	✔	NA
e_width	✔ 0	✔ 0	✔	NA
sign	✔	✔	✔ true	NA
q_mode	✔ AC_TRN	✔	✔	NA
o_mode	✔ AC_WRAP	✔	✔ AC_SAT	NA

The following table lists the type definitions that are dependent on the template parameters.

Table 89: Type definitions to capture properties of types.

Static Data Member	ac_int	ac_fixed	ac_float	ac_complex
mant_t	NA	NA	✔	NA
exp_t	NA	NA	✔	NA
element_type	NA	NA	NA	✔

6.2.4. Type Definitions for Signed and Unsigned ac_ints

Under the *ac_intN* namespace, there are typedefs for *ac_int<W,S>* for 1 ≤ W ≤ 63 for S={false,true}. The type names are *int* or *uint* followed by the width. For example:

```
uint6 a = 7;
ac_intN::int17 b = 45676;
```

The *ac_intN* namespace is made available in the global namespace unless the macro *AC_NOT_USING_INTN* is defined.

6.2.5. Utility Enumerations and Type Definitions Based on Template Arguments

Static Computation of Bit-width and Type Based on Range

The following table lists utility enum for the compile-time computation of log2-like functions. They are useful to determine the minimum bit-widths of types that can represent a number or a range of numbers.

Table 90: Utility enums and typedefs based on template parameters

struct enum/typedef	Header	Type or enum	Description
ac::nbits<unsigned>::val	ac_int.h	enum	Number of bits required for unsigned ac_int to store value of template parameter
ac::log2_floor<int>::val	ac_int.h	enum	Floor of log2 of template parameter
ac::log2_ceil<int>::val	ac_int.h	enum	Ceiling of log2 of template parameter
ac::int_range<int, int>::type	ac_int.h	ac_int	ac_int type that can represent close integer range given by template parameters

Minimal Size Destination Type to Represent a Source Type

The type defines *ac_int_represent<T>::type*, *ac_fixed_represent<T>::type* and *ac_float_represent<T>::type* provide the minimal destination *ac_int*, *ac_fixed* and *ac_float* type respectively that is required to store the source type *T* as shown in Table 91. Note that while it would be possible to provide an *ac_fixed* to represent a *float* or *double*, its size would be impractically large so it is not provided.

Table 91: Type definitions for Minimal Size Destination Types.

Source Type	ac_int_represent<T>::type	ac_fixed_represent<T>::type	ac_float_represent<T>::type

C++ integer types	✔	✔	✔
C++ float/double	NA	NA	✔
ac_int	✔	✔	✔
ac_fixed	NA	✔	✔
ac_float	NA	NA	✔

Return Type Infrastructure for Unary Operators and Methods

The *rt_unary struct* defined in each of the types provides parameter and type information to obtain the return type when it is solely dependent on the template parameters of the type. The enumerations only make sense for scalar values (not for *ac_complex*), but the typedefs are available for all types for which the function that requires that return type is defined.

Table 92: Enumerations defined in type::rt_unary.

Enumeration	ac_int	ac_fixed	ac_float	ac_complex
neg_w, neg_s	✔	✔	✔	NA
neg_i	NA	✔	✔	NA
neg_e	NA	NA	✔	NA
mag_sqr_w, mag_sqr_s	✔	✔	✔	NA
mag_sqr_i	NA	✔	✔	NA
mag_sqr_e	NA	NA	✔	NA
mag_w, mag_s	✔	✔	✔	NA
mag_i	NA	✔	✔	NA
mag_e	NA	NA	✔	NA
set<N>::sum_w, set<N>::sum_s	✔	✔	✔	NA
set<N>::sum_i	NA	✔	✔	NA
set<N>::sum_e	NA	NA	✔	NA
leading_sign_w, leading_sign_s	✔	✔	NA	NA
to_fx_w, to_fx_i, to_fx_s	NA	NA	✔	NA
to_i_w, to_i_s	NA	NA	✔	NA

Table 93: Type definitions in type::rt_unary.

typedef	ac_int	ac_fixed	ac_float	ac_complex
neg	✔	✔	✔	✔
mag_sqr	✔	✔	✔	✔
mag	✔	✔	✔	✔
set<N>::sum	✔	✔	✔	✔

leading_sign	✔	✔	NA	NA
to_ac_fixed_t	NA	NA	✔	NA
to_ac_int_t	NA	NA	✔	NA

Return Type Infrastructure for Binary Operators and Methods

The *rt* and *rt_T* structs defined in each of the types provides parameter and type information to obtain the return type when it is dependent on the template parameters of the type and of a second operand. The rt structure is used for *ac_int*, *ac_fixed* and *ac_float* to determine the return type when the second operand is of the same class, but potentially different parameter values. The *rt_T struct* is used to specify the type directly as a template parameter (T stands for template type). The *ac_complex* type uses the cross type infrastructure as described to figure out the return types where the operands are *ac_complex<T1>* and *ac_complex<T2>*.

The *rt struct* has template parameters as follows

- ac_int<W,S>::rt<int W2, bool S2>

- ac_fixed<W,I,S,Q,O>::rt<int W2, int I2, bool S2>

- ac_float<W,I,E,Q>::rt<int W2, int I2, int E2>

Table 94: Enumerations defined in type::rt.

Enumeration	ac_int	ac_fixed	ac_float
mult_w, mult_s	✔	✔	✔
mult_i	NA	✔	✔
mult_e	NA	NA	✔
plus_w, plus_s	✔	✔	✔
plus_i	NA	✔	✔
plus_e	NA	NA	✔
minus_w, minus_s	✔	✔	✔
minus_i	NA	✔	✔
minus_e	NA	NA	✔
div_w, div_s	✔	✔	✔
div_i	NA	✔	✔
div_e	NA	NA	✔
mod_w, mod_s	✔	NA	NA
logic_w, logic_s	✔	✔	✔
logic_i	NA	✔	✔
logic_e	NA	NA	✔

Table 95: Type definitions in type::rt.

typedefs	ac_int	ac_fixed	ac_float
mult	✔	✔	✔
plus	✔	✔	✔
minus	✔	✔	✔
div	✔	✔	✔
mod	✔	NA	NA
logic	✔	✔	✔
arg1	✔	✔	✔

The *rt_T* struct provides the same typedefs as in Table 95 but exclude the *mod* typedef since *ac_complex* does not provide the *%* operator. For operations that are not commutative it also provides *minus2* and *div2* which are for the operator types swapped. For example, when *ac_complex* queries the type of an *ac_int* minus an *ac_fixed*, it queries the *ac_int::rt_T<ac_fixed>::div* which gets the type from *ac_fixed::rt_T<ac_int>::div2*.

The *rt_2T* struct has two template parameters *T* and *T2*. For Numerical AC types, it is equivalent to *T::rt_T<T2>*. Use of the *rt_2T* struct is more flexible than the *rt_T struct* in that *T* could be a native C++ integer type.

6.3. Macros

This section covers the macros that are available in the numerical AC header types.

6.3.1. User Definable Macros

Table 96: User Definable Macros

Macro	Description
AC_DEFAULT_IN_RANGE	If defined, default constructors will guarantee a value that is in range even though it is un-initialized.
AC_USER_DEFINED_ASSERT	Can be defined to be a call to user defined assert function with arguments: bool condition, const char *file, int line, const char *msg
AC_NOT_USING_INTN	If defined, disables the "using" of namespace ac_intN. See Type Definitions for Signed and Unsigned ac_ints.

6.3.2. Utility Macros

The following macros are defined in *ac_int.h*. All Numerical AC Datatype headers include *ac_int.h* directly or indirectly so these macro definitions should be available by inclusion of any of the numerical header files.

Table 97: Utility Macros

Macro	Description
AC_MAX(a,b)	Max of arguments
AC_MIN(a,b)	Min of arguments
AC_ABS(a)	Absolute value of argument
AC_ASSERT(cond, msg)	Assert. Affected by AC_USER_DEFINED_ASSERT

Chapter 7: Numerical Datatype Migration Guide

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

7.1. 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>*.

7.2. SystemC Syntax

Table 98 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,* and *sc_ufixed_fast*.

Table 98: 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 SystemC quantization modes are supported as shown in Table 99. Most commonly used overflow modes are supported as shown in Table 100.

Table 99: Quantization Modes for ac_fixed 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
AC_RND_CONV_ODD	Not available

Table 100: 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).

7.2.1. 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, SX>* where *SX* 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(WZ, x); y.set_slc(0, z);
where WZ is the width of z.
```

Other Methods

Table 101 shows other less frequently used methods in SystemC datatypes that would require rewriting in *ac_int*.

Table 101: 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*.

7.2.2. Support for SystemC sc_trace Methods

The Algorithmic C Datatypes package was updated in 2010 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 or ac_float.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.

7.3. 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

7.3.1. 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.

7.3.2. 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:

```
// OK
w_u20 = a_u8 * b_s9 + x_u13 & y_s4;

// Bad, assigning Ulong to 67 signed
sc_bigint<67> i = a_u8 * b_s9 + x_u13 & y_s4;

// Bad, s/s div should be ok, but numerator is Ulong
w_u20 = (a_u8 * b_s9) / c_s6;

// Bad if both f(Ulong) and f(Slong) are defined
f(a_u8 * b_s9);
```

- 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*.

7.3.3. 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 Shift Operators section. 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 Implementation Deficiencies of sc_int/sc_uint on page 91).

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. 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.

7.3.4. Mixing Datatypes

This section describes the conversion functions that are used to 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:

```
<ac_datatype_install>/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 8:   Frequently Asked Questions on Numerical Datatypes

This section contains answers to some frequently asked questions. These questions are divided into the following sections:

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

- Conversions to double and Operators with double

- Constructors from strings

- Shifting Operators

- Division Operators

- Compilation Problems

- Platform Dependencies

- Purify Reports

- User Defined Asserts

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

The following section describes common issues with these operators.

Why does ~ and - for an unsigned return signed?

For a table of return values, refer to the *Unary Operator* section.

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.

8.1.2. Conversions to double and Operators with double

The following section describes common issues with using *double* datatypes.

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 losing 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
      ...
}
```

8.1.3. Constructors from strings

The following section describes common issues with strings.

Why are constructors from strings not defined?

They would be very runtime inefficient.

8.1.4. Shifting Operators

The following section describes common issues with 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 *Shift Operators* section. 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.

8.1.5. Division Operators

The following section describes common issues with 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 (refer to the return type table for *ac_fixed*).

8.1.6. Compilation Problems

The following section describes common compilation issues.

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*.

8.1.7. Platform Dependencies

The following section describes the supported platforms.

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*.

8.1.8. Purify Reports

The following section describes common issues with using Purify with the AC datatypes.

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 the ac:init_array function.). This is used mostly for algorithms written for hardware design.

8.1.9. User Defined Asserts

The following section describes common issues with using 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).

Chapter 9:   ac_channel Datatype

Several synthesis and modeling problems become apparent when attempting to use C function calls to describe a hierarchical system. The purpose of the AC (Algorithmic C) channel class is to simplify both synthesis and modeling with a minimal impact on coding style and C simulation performance.

The first problem is primarily a synthesis problem. To build a streaming interface, you need to be able to guarantee that the data communicated between two blocks is read and written in the same order. Since an ac_channel enforces a FIFO discipline, it guarantees that the hardware can build for streaming interfaces.

The second problem is that systems are often made up of blocks that run using different amounts of data. The first block in a system might produce data 100 values at a time, while the reader may have been written to consume data 128 elements at a time. AC Channels automate the connection of these blocks.

Finally, many systems have feedback and the length of the feedback path must be defined in the C source. Changing the feedback path length during synthesis would result in a design that simulates differently, which would violate the most fundamental rule of Algorithmic Synthesis. AC Channels provide a way to define a feedback path so it can be easily simulated and synthesized.

9.1. The ac_channel Class Definition

The fundamental operation that High Level Synthesis performs is to convert a single-threaded C/C++ model into a high-performance, multiple-process RTL model. Each hierarchical function in the C/C++ model becomes a process in the RTL model. To allow for more efficient synchronization of the data among processes, the ac_channel class may be used. The restrictions are necessary to insure that the C/C++ model and the RTL model have the same functionality while allowing the RTL to be efficiently implemented.

The ac_channel class is a C++ template class that enforces a FIFO discipline (reads occur in the same order as writes.) From a modeling perspective, an ac_channel is implemented as a simple interface to the C++ standard queue (std::deque). That is, for modeling purposes, an ac_channel is infinite in length (writes always succeed) and attempting to read from an empty channel generates an assertion failure (reads are non-blocking).

Within the C/C++ model, an instance of an ac_channel may be written by a single hierarchical function and read by a single, different, hierarchical function. In the resulting hardware, the FIFO corresponding to an ac_channel will be written from a single process and read by a single process - thus implementing point-to-point communication. For purposes of this discussion, the external world counts as a process. If a channel appears in the top-level interface of a design and is only written by the design, then the external environment is assumed to be the unique reader. Likewise, the external environment is assumed to be the writer of a channel which is only read.

9.2. ac_channel Member Functions

The ac_channel constructor includes the ability to add a set of values that are read before the inputs to the channel are read. This is most commonly used to set the size of the feedback path, but also has other applications in the feed-forward path to control the ramp-up of the system.

9.2.1. Member Function: ac_channel()

Example usage:

```
ac_channel<ac_fixed<12,6> > my_channel;
```

Description:

Note: You must put a space between any two ">" characters or you will get a compiler error because the parser treats ">>" as a right shift operator.

This will construct a channel with a 6.6 fixed-point datatype. No other information is required to create a channel.

9.2.2. Member Function: ac_channel(prefill_num)

Example usage:

```
ac_channel<int> my_channel(16);
```

Description:

This constructs a channel with a 32-bit value and pre-fills the channel with 16 values. For the built-in C++ datatypes *char*, *short*, *int*, *unsigned*, *long*, *float* and *double* the value in the channel is a different random number for each element in the fifo. For all other datatypes, the default constructor is called to set the values.

9.2.3. Member Function: ac_channel(prefill_num, value)

Example usage:

```
typedef struct { int re; int im; } complex;
const complex init_complex = {4,5};
ac_channel <complex> my_channel(16, init_complex);
```

Description:

This will construct a channel and pre-fill the channel with 16 complex variables where "*re*" is 4 and "*im*" is 5. This channel is most commonly used for designs with feedback to set the depth of the feedback path while still providing valid data.

9.3. Synthesizable Member and non-Member Functions

These functions can be synthesized into hardware. All constructor member functions are also synthesizable. The specific hardware that is created for each of these member functions will be discussed when the different design styles are covered.

9.3.1. Member Function: val read() or read(&val)

Example usage:

```
ac_channel<int> my_channel(16);
…
int first_value = my_channel.read();
int second_value;
my_channel.read(second_value);
```

Description:

The read member functions are equivalent and two member functions are only provided to allow different coding styles. The code above would read two values, the first would be put into *first_value* and the second would be put into *second_value*.

NOTE: Do not call the read() function multiple times for the same channel in a single statement. This is because the order in which expressions and subexpressions are evaluated (left-to-right, right-to-left or other) is undefined by C/C++ language standards, and consequently varies from one compiler to another. Unexpected results can occur if a code statement contains order dependent (sub)expressions in which a variable is both read and modified. For example, the following statement is potentially problematic because it is left to the compiler to decide the order in which the read() functions are evaluated:

output->write(input1->read()*5 + input1->read()*7)

9.3.2. Member Function: bool nb_read(&val)

Example usage:

```
ac_channel<uint8> &pri0, &pri1;
uint8 read_out1, read_out2;
…
if (pri0.nb_read(read_out1))
  out.write(read_out1);
else if (pri1.nb_read(read_out2))
  out.write(read_out2);
```

Description:

The "bool nb_read(T &val)" function will test for the presence of data in the channel and read that data if it is available. This function returns boolean false if no data is available. Otherwise it returns true, pops the first value in the channel and assigns it to the "*val*" argument. It is capable of completing in one cycle in hardware.

NOTE: When you are verifying the behavior of the C++ design against the resulting RTL netlist, the non-blocking read execution order may be different between the C++ and the RTL. The RTL will implement a viable schedule of the C++ but the C++ testbench is not guaranteed to provide data in a way that causes the C++ execution to exactly match the RTL execution.

9.3.3. Member Function: write(val)

Example usage:

```
ac_channel<int> my_channel(16);
…
int temp = f(x);
my_channel.write(temp);
```

Description:

Write a new value into the channel. Since channels are typed, the value written must be convertible to the type stored in the channel.

9.3.4. Member Function: bool nb_write(val)

Example usage:

```
if ( out.nb_write(temp) ) {
  temp = in.read();
}
```

Description:

The *nb_write* member function will test for the ability to write data to an output port and will write the *val* argument to the output and return *true* if the data can be written. The function returns *false* if no data can be written. It is capable of completing in one cycle in hardware.

NOTE: When you are verifying the behavior of the C++ design against the resulting RTL netlist, the non-blocking write execution order may be different between C++ and RTL. The RTL will implement a viable schedule of the C++ but the C++ testbench is not guaranteed to provide data in a way that causes the C++ execution to exactly match the RTL execution.

9.3.5. Member Function: bool available(num)

Example usage:

```
if (my_channel.available(N)) {
```

```
  acc = 0;
  for (int i = 0; i < N; i++)
    acc += my_channel.read();
}
```

Description:

The available member function is used to guard reads from a channel. This check must be done at the start of any function where you cannot predict the amount of data in the input channel.

You may pass an integer variable to the available function. During synthesis, the "available()" function is always considered to return true and the read from the channel is changed to be a blocking read. This transformation allows fast C simulation while still providing good hardware performance.

9.3.6. Member Function: int size()

Example usage:

```
ac_channel<uint8> &pri0;
…
if (pri0.size() > 0)
  out.write(pri0.read());
```

Description:

This member function returns the number of data elements currently in the ac_channel. It synthesizes as non-blocking.

Notes:

- Currently size() is supported only for reading channels, not for writing.

- You may not read the size of the channel and also read data in the channel in a single cycle. For example, the following statement requires two cycles:

```
if (input.size() > 0)
  data = input.read();
```

- Do not use "==" operator to test the return value of size() because it could create a deadlock situation. For example, if the code is testing for "size == 1" and size grows to 2 before the test, then the statement will never be true and the design will deadlock. Use other relational operators (>, >=, !=, and so on) instead.

9.3.7. Function: nb_read_join(Args&... args)

This global function allows the "joining" of any number of ac_channel non-blocking reads so that the read

succeeds only if all ac_channels have data available. This function is only available with C++11 or later as it is implemented with variadic templates.

```
template<typename ...Args>
bool nb_read_join(Args&... args);
```

The function takes any number of ac_channels and arrays of ac_channels as arguments along with the variables where the read values are stored. If all channels have data, they are all read and the function returns true, otherwise none are read and the function returns false. Each ac_channel<T> (or array of ac_channel<T>) are paired with the argument of type T (or array of type T) where the read values are returned by the function. For example:

```
ac_channel<short> a;   short av;
ac_channle<int> b[2]; int bv;
…
if ( nb_read_join(a, av, b, bv) ) {
  // all channels a, b[0] and b[1] are read. Values are stored in av, bv[0] and
bv[1]
} else {
  // none of the channels are read
}
```

9.4. Non-synthesizable Member Functions

These member functions may only be used in testbenches or other areas of the C++ code that will not be synthesized.

9.4.1. Member Function: bool empty()

Example usage:

```
ac_channel<int> my_channel;
my_channel.write(10);
my_channel.write(45);
consume_data(my_channel);  // This function reads from the channel
if (my_channel.empty())
  cout << "All values were read from my_channel and it is now empty" << endl;
```

Description:

This function returns true if there are no values stored in the channel. This is equivalent to checking if size() returned zero.

9.4.2. Member Function: bool operator ==

Example usage:

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

int main (void) {

  ac_channel<int> chan1;
  ac_channel<int> chan2;

  chan1.write(10);
  chan2.write(10);

  if (chan1 == chan2)
    std::cout << "EQUAL" << std::endl;
  else
    std::cout << "NOT EQUAL" << std::endl;

  return 0;
}
```

Description:

This function compares every element in a channel to verify that the two channels have exactly the same contents. This function requires that the base type (in this case, int) be the same for both operands. This function will return false if any element in the two channels is different or if the number of elements in the channels is different.

9.4.3. Member Function: bool operator !=

Example usage:

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

int main (void) {

  ac_channel<int> chan1;
  ac_channel<int> chan2;

  chan1.write(10);
  han2.write(15);
```

```
  if (chan1 != chan2)
    std::cout << "EQUAL" << std::endl;
  else
    std::cout << "NOT EQUAL" << std::endl;

  return 0;
}
```

Description:

This function compares two channels and returns true if the number of the channels have a different number of elements or if the contents of the channels are different. This is equivalent to the negation of the == operator.

9.4.4. Member Function: val operator[int]

Example usage:

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

int main (void) {

  ac_channel<int> chan1;


  …
  // This loop prints the values from the channel without popping/destroying
  for (int j = 0;  j < chan1.size(); j++)
    std::cout << chan1[j];
  return 0;
}
```

Description:

The [] operator returns a value that is stored in the channel without popping the value from the channel (i.e. it is a peek).   If you think of the channel as an array with element zero being the next one to be read, then this operator returns an element from that array. For example, my_chan[0] is the value the would be read next from the FIFO and my_chan[my_chan.size()-1] is the last value written into the channel.

9.4.5. Member Function: reset()

Example usage:

```
ac_channel<int> my_channel(16);
…
```

```
if (clear_chan_condition)
  my_channel.reset();
```

Description:

This function will set the channel to the same state that it was in immediately after construction. So, if the channel has a set of initial values, then the same values will be in the channel after reset(). All data currently in the channel is lost.

NOTE: This function may cause simulation mismatches when used on any channel implemented in the RTL or directly connected to the RTL. Calling reset on these channels will cause the state of the C simulation to mismatch with the hardware simulation and may cause the testbench to report a failure on a correct design.

9.5. Example Design Using Hierarchical Blocks With ac_channel

The most common use of ac_channels is as a communication pipe between two hierarchical blocks in a design. In this case, the blocks are assumed to run continuously and only stall when there is insufficient input. In the example design below, the block "block_add_avg()" reads two input values from each input channel, computes their average and returns the sum of the averages. The second block "block_mult()" returns the product of the two input values.

```
void block_add_avg(
    ac_channel<int> &op1,
    ac_channel<int> &op2,
    ac_channel<int> &ret)
{
#ifndef __SYNTHESIS__
  while (op1.available(2) && op2.available(2))
#endif
  {
    int avg1 = (op1.read()+op1.read())/2; // average 2 samples from op1
    int avg2 = (op2.read()+op2.read())/2; // average 2 samples from op2
    ret.write(avg1+avg2); // return sum of averages
  }
}

void block_mult(
    ac_channel<int> &op1,
    ac_channel<int> &op2,
    ac_channel<int> &ret)
{
#ifndef __SYNTHESIS__
  while (op1.available(1) && op2.available(1))
```

```
#endif
  {
    ret.write(op1.read() * op2.read());
  }
}

void top(
    ac_channel<int> &add1,
    ac_channel<int> &add2,
    ac_channel<int> &coeff,
    ac_channel<int> &result)
{
  static ac_channel<int> sum;
  block_add_avg(add1, add2, sum);
  block_mult(sum, coeff, result);
}
```

In this example, the execution of the body of block_add_avg() and block_mult() is guarded by calls to the channel available() functions. This allows the C++ simulation to continue running a block as long as there is data in the channel. When the channels are empty, the C++ design basically does nothing until the testbench pushes more data into the channels. Likewise, in the resulting hardware the available() calls are filtered out and each blocks runs continuously as long as the input channel/pipe has data to feed.

9.6. Example Design Using Non-Blocking size() Method

This example arbiter design contains four processes that are all trying to write to a shared resource, the output bus. The arbiter function implements a rotation-based (or round robin) arbitration algorithm to read from each process, one after the other. Using non-blocking reads, if no data is available when the read is attempted, the arbiter moves on to the next process. The following illustrates the major features of the design.
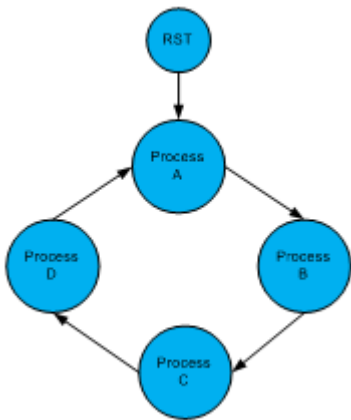
The C++ code implementation is shown in the following example. In, top.cpp, blocks A,B,C,D implement the client processes trying to access the output bus. Each of these blocks reads its input ac_channel using the

blocking *read()* method, then writes the modified data to its output FIFO on each successful read.

The arbiter block first uses the non-blocking *size()* method to check for data availability in the output FIFO of each process block. Second, for those processes that have data in their FIFOs, the arbiter selects one process to be served based on the rotation arbitration algorithm. Third, it reads the selected process FIFO (blocking method) and writes the data to the bus. Finally, the arbiter updates the serving priority based on rotation algorithm.

Example 1 Arbiter Example Using Non-Blocking size() Method

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


#pragma hls_design
void block_A(ac_channel<uint8> &in, uint8 &offset, ac_channel<uint8> &out)
{
#ifndef __SYNTHESIS__
  while (in.available(1))
#endif
  {
    out.write(in.read()+offset);
  }
}


#pragma hls_design
void block_B(ac_channel<uint8> &in, uint8 &offset, ac_channel<uint8> &out)
{
#ifndef __SYNTHESIS__
  while (in.available(1))
#endif
  {
    out.write(in.read()+offset);
  }
}


#pragma hls_design
void block_C(ac_channel<uint8> &in, uint8 &offset, ac_channel<uint8> &out)
{
#ifndef __SYNTHESIS__
  while (in.available(1))
#endif
  {
    out.write(in.read()+offset);
  }
}
```

```
#pragma hls_design
void block_D(ac_channel<uint8> &in, uint8 &offset, ac_channel<uint8> &out)
{
#ifndef __SYNTHESIS__
  while (in.available(1))
#endif
  {
    out.write(in.read()+offset);
  }
}


#pragma hls_design
void arbiter(ac_channel<uint8> &pri0,
  ac_channel<uint8> &pri1,
  ac_channel<uint8> &pri2,
  ac_channel<uint8> &pri3,
  ac_channel<uint8> &out)
{
  static uint4 priority = 1;
  ac_int<4,false> p_rdy;

  // Non-blocking "size()" function is used here to
  // check for output data in all process block FIFOs */
  p_rdy[0] = pri0.size() > 0;
  p_rdy[1] = pri1.size() > 0;
  p_rdy[2] = pri2.size() > 0;
  p_rdy[3] = pri3.size() > 0;

  uint4 temp_priority=priority;
  for (int i=0;i<4;i++){
    if (p_rdy & temp_priority){
      p_rdy = p_rdy & temp_priority;
      break;
    }
    uint1 temp = temp_priority[3];
    temp_priority = temp_priority << 1;
    temp_priority[0] = temp;
  }

  // Arbiter read data from the selected process block
  // and write it to the bus (arbiter output FIFO)
  if (p_rdy[0])
    { out.write(pri0.read()); }
  else if (p_rdy[1])
    { out.write(pri1.read()); }
```

```
    else if (p_rdy[2])
      { out.write(pri2.read()); }
    else if (p_rdy[3])
      { out.write(pri3.read()); }

    // Update priority setting for next cycle.
    if (p_rdy) {
      priority = temp_priority;
      uint1 temp = priority[3];
      priority <<= 1;
      priority[0] = temp;
    }
}

#pragma hls_design top
void top(uint8 & offset1,
  uint8 & offset2,
  uint8 & offset3,
  uint8 & offset4,
  ac_channel <uint8> &in_A,
  ac_channel <uint8> &in_B,
  ac_channel <uint8> &in_C,
  ac_channel <uint8> &in_D,
  ac_channel <uint8> &out)
{
  static ac_channel<uint8> pri0, pri1, pri2, pri3;
  block_A(in_A, offset1, pri0);
  block_B(in_B, offset2, pri1);
  block_C(in_C, offset3, pri2);
  block_D(in_D, offset4, pri3);
  arbiter(pri0,pri1,pri2,pri3,out);
}
```

Using the following testbench, you can verify the arbiter design.

NOTE: Because this design uses non-blocking reads, the simulation that compares the RTL against the C++ may produce mismatches due to the execution order of the non-blocking reads in the C++ design. The RTL will produce a viable implementation but the simulation may not match the C++ design.

Example 2 Testbench for Arbiter Example Design

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

int tb_arbiter(int argc, char *argv[]) {
  static ac_channel<uint8> in_A, in_B, in_C, in_D;
```

```
  static ac_channel<uint8> out;

  uint8 tmp;
  uint8 offset = 0;
  for (int i=0; i<10;i++) {
    in_A.write(rand() % 255);
    in_B.write(rand() % 255);
    in_C.write(rand() % 255);
    in_D.write(rand() % 255);

    top(offset, offset, offset, offset,
        in_A, in_B, in_C, in_D, out);

    while(out.available(1)) { // dump outputs
      tmp = out.read();
      printf("out = %d\n",tmp.to_int());
    }
  }
  return 0;
}
```