SIEMENS EDA

Algorithmic C (AC) Datatypes Release Notes

Software Version v4.6.1
August 2022

SIEMENS

Table of Contents

Table of Contents

Table of Contents

Bit-Accurate Datatypes

Changes in 4.6.0

Fixed IEEE floating-point compliance issues for types in ac_std_float.h for operators/methods add/fma/sqrt with respect to returning +0 vs -0

For the types in *ac_std_float.h* (*ac_std_float*, *ac_ieee_float* and *ac::bfloat16*), there were operators/methods that did not fully adhere to the IEEE floating-point standard with respect to returning +0 vs. -0:

1. *add*: now -0 + -0 returns -0
2. *fma*: addressed issue above (for add) also issue where the tracking of exact 0 result was not done for the purpose of figuring out whether result should be +0 or -0.
3. *sqrt*: now sqrt(-0) returns -0

Improved the implementation of the '+' and '*' operators for types in ac_std_float.h (ac_std_float, ac_ieee_float, ac::bfloat16) for better Quality of Results

The new implementations should have shorter critical paths. This should be most easily seen when synthesized as a combination block. In general, it is expected to give better delay/area designs.

Replaced implementation for writing bit(s) for ac_int and ac_fixed to more robust version that should not be incorrectly optimized by compilers when the ac_int or ac_fixed variable is uninitialized

The old implementation used x XOR x to set bit(s) (bit assignment, set_slc) in *ac_int* and *ac_fixed*. It is expected to work even if the variable is uninitialized. However, optimizations in compilers such as CLANG++ (LLVM) may treat x XOR x as uninitialized instead of 0 and that can result in behavior that does implement the intended functionality when the variable is uninitialized.

The changes are based on an alternative implementation proposed by Jeremy Dorfman on github (hlslibs/ac_types) that does not use XOR and is guaranteed to not have the optimization issues when used on uninitialized variables. Another of his proposals that is also included is the addition of specializations for some underlying shifting methods to improve runtime.

Changes in 4.5.0

Added methods that return the bit pattern reversed for *ac_int*:

```
ac_int<W,false> ac_int<W,S>::reverse() const;
```

The return type is always unsigned.

Changes in 4.4.2

Added missing *type_name* methods to classes in *ac_std_float.h*.

Changes in 4.4.1

Fixed incorrect assert in templated range method assignment when target range spans one more 32-bit integer (the base implementation of ac_int and ac_fixed is an array of integers) than the source range. For example the following assignment was triggering an assert:

```
a.range<71,8>() = b.range<63,0>();
```

Changes in 4.4.0

Fixed issues with ac_complex of the types defined in ac_std_float.h

The type infrastructure that determines the return type of operations of ac_complex of the types in ac_std_float.h was incomplete leading to compilation errors. For example:

```
ac_complex<ac_ieee_float32> x = …;
ac_ieee_float32 s = …;
ac_complex<ac_ieee_float32> z = x * s;  // complex * scalar resulted in error
```

The above issue was also present for *ac::bfloat16*. The infrastructure was missing for *ac_std_float<W,E>* and even the operators with both operands being *ac_complex* would error in compilation.

Changes in 4.3.0

Added methods and_reduce, or_reduce and xor_reduce to ac_fixed

The *reduce* methods were present for ac_int, but not for ac_fixed. It is no longer necessary to go to ac_int first (using the slc method) to get the reduce functionality.

Changes to ac_float constructor from ac_float with different parameters

The following changes determine whether a full normalization step is called to save on redundant steps of normalization. The changes may impact behavior when the source *ac_float* is not normalized. The old behavior was likely to apply a normalization step, whereas the new behavior does not apply it depending on the template parameters of the source *ac_float* and the *ac_float* being constructed.

1. Added a bool argument *force_normalize*. If the argument is *false*, normalization is not done if the mantissa and exponent for the new *ac_float* can be derived from the mantissa and exponent of the source *ac_float* in a way that preserves the normalization of the source (assuming that the source is already normalized). The intent is to avoid redundant normalization steps. Calling the constructor

with the argument *force_normalize* set to true is useful when it is known that the source *ac_float* may not be normalized.

2. The constructor was rewritten to identify scenarios were the normalization of the source *ac_float* can be preserved without requiring a full explicit normalization step. The scenarios require that the exponent widths be identical.

The flags *assert_on_rounding* and *assert_on_overflow* are now consistently checked.

Fixed incorrect behavior for conversion from ac_std_float to another ac_std_float type, but that has same exponent width

The incorrect behavior was exercised when the source and target *ac_std_float* types had the same exponent width (*E* template parameter value) and the width (*W* parameter) of the source was larger than the width of the target. For example, when constructing an *ac_std_float<23,8>* from an *ac_std_float<32,8>*, the conversion that had the incorrect behavior would be exercised.

Added missing constructor for ac_std_float from ac_int

The constructor was documented, but was missing from the implementation.

Changes in 4.2.0

Fixed issues standard floating-point types: ac_std_float, ac_ieee_float and ac::bfloat16

1. The constructors from *ac_fixed* were not providing correct results depending on template parameter values.

2. For the *convert<WR,ER,QR>* methods, when there is overflow and *QR* is *AC_TRN_ZERO* the result is *+/-max()* which is consistent with the IEEE floating standard. The old behavior was producing *+/-Inf*.

3. The *convert_to_ac_fixed<WR,ER,QR,OR>(bool map_inf=false)* methods were not correct when *map_inf=true*.

Added conversion methods to ac_int, int and long long for standard floating-point types: ac_std_float, ac_ieee_float and ac::bfloat16

The conversions methods expand on the *convert_to_ac_fixed<WR,ER,QR,OR>(bool map_inf=false)* to add explicit conversions to *ac_int*, *int* and *long long*. The conversions are equivalent to first going to the equivalent width ac_fixed with *QR* set to *AC_TRN_ZERO* and *OR* set to *AC_WRAP*. These settings are consistent with how *float* to *int* conversion occurs in C++ (though overflow behavior is not specified in C++). The new methods are provided below:

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()

Fixed issues with python ac_pp.py pretty printing in dgb for ac_complex

The fields for the real and imaginary were not showing up while pretty printing in gdb.

Changes in 4.1.0

Operator+ made const for ac_int, ac_fixed and ac_complex

The *operator+()* for *ac_int, ac_fixed* and *ac_complex* is now *const* to prevent incorrect usage.

Added conversion function to_ac_ieee_float() and unary operator ! to bfloat16

Added the conversion function *to_ac_ieee_float()* and unary operator ! to *bfloat16* for completeness.

Fixed incorrect overflow behavior with types in ac_std_float with AC_RND_INF

Fixed a functional issue when using AC_RND_INF as rounding mode for floating-point types in *ac_std_float.h*. The issue could lead to overflow and result in change of sign of the number.

Fixed incorrect behavior of method copysign() for ac_ieee_float and ac::bfloat16

The copysign() methods for classes *ac_ieee_float* and *ac::bfloat16* were leaving the sign of the object unchanged. This is now working as intended. This is issue was not present for class *ac_std_float*.

Fixed missing return for ac_std_float *operator* *=

The operator *= method for the class *ac_std_float* (in ac_std_float.h) was not returning a value. This would lead to a problem in the following example:

```
c = (a *= b);
```

Updated *ostream operator* << for classes in ac_std_float.h

The *ostream operator* << for the classes in *ac_std_float.h* now take into account the format flags such as *std::ios::hex* and *std::ios::oct.* Note however, that for types that go beyond the widths for the type *double*, the decimal flag leads to using the hexadecimal format instead.

Some users requested that since the printing in hexadecimal or octal formats is in “raw” format, that the full width is printed (no assumption of sign extension). For instance the value 0 for *ac_ieee_float32*, will be printed in hexadecimal as 0x00000000 instead of 0x0 as was done before.

This functionality is subject to change as it is not consistent with other types in ac_types and alternative ways to get printing of raw values could be made available.

Additional argument added to method *to_string* for ac_int and ac_fixed

An additional argument pad_to_width has been added to the *to_string* method for *ac_int* and *ac_fixed*:

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

The default value is *false* and should allow existing code to produce the same results unchanged. When the argument is set to *true*, the string that is returned will be padded to the full width of the type when selecting either *AC_OCT, AC_HEX* or *AC_BIN* (it does not have any effect on *AC_DEC*).

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
```

Fixed ubsan errors on left shift of long long type use in iv_div and iv_rem in ac_int.h

The C++ 11 standard now makes the left shift of signed integer types if it causes an overflow (in other words, a right shift by the same amount of the result would not produce the original number). This is detected by ubsan as an error. Given that computer hardware generally uses two's complement representation for integers, this is somewhat of a pedantic error that would only affect portability to an implementation that would use an alternative representation, such as sign-magnitude, for integers.

When the intention is to do a bit shift without consideration for overflow, the operand needs be cast to unsigned before performing a left shift. The implementation of iv_div and iv_rem in ac_int.h was updated to conform to the C++ 11 definition.

Fixed Clang Warnings in ac_std_float.h

Addressed new warnings with clang 11 that were not there with clang 9 related to parentheses.

Changes in 4.0.0

New Standard Floating-Point Datatypes

A new set of classes for Standard (IEEE and IEEE like) synthesizable floating-point numbers is now part of the package. The new types are implemented with the following three classes:

1. *ac_ieee_float*: IEEE types of widths 16, 32, 64, 128 and 256.

2. *ac_std_float<W,E>*: generalization of the *ac_ieee_float* for overall width *W* and exponent width *E*. For example, *ac_std_float<32,8>* implements the behavior of *ac_ieee_float32*. The class *ac_std_float* implements **arbitrary-length** standard floating-point numbers.

3. *ac::bfloat16*: implements *bfloat16* from Google.

Range methods for ac_int and ac_fixed

Range methods have been added to ac_int and ac_fixed to more closely mirror syntax used by HDLs. The MSB and LSB of the range are specified as template parameters. For example:

```
ac_int<20,true> x = …;
ac_fixed<16,2,false> y = …;
y.range<9,5>() = x.range<4,0>();
```

Range length must match for the assignment operator to be available. Reverse ranges (MSB < LSB) are not supported (will trigger static_assert in C++11). The MSB and LSB need to be within bounds, otherwise it is an error (will trigger static_assert in C++11).

As with the *set_slc* method, the variable to be written to needs to be initialized to avoid the potential of compilers using the uninitialized value as an undef value (Clang) and optimizing the writing of the slice/range away under some optimization levels.

Changes in 3.9.5

Joined non blocking read function for ac_channel

Added the function

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

implemented with variadic templates (requires C++11 or later standard versions) to take 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 will all be read and the function return true, otherwise none will be 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 value is 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) ) { … }
```

Changes in 3.9.4

Return type of slc method for ac_int and ac_fixed

The return type of the *slc* method for *ac_int* and *ac_fixed* is now *const*. This prevents incorrect usage where the returned temporary *ac_int* is used as a target of an assignment (as a left hand side) which is a no-op.

Constructors of ac_complex

Changed constructors to used constructors instead of assignments to initialize data members.

Changes in 3.9.3

Change in ac::nbits

The utility ac::nbits<N>::val is now defined to be 1 for N=0. It used to be 0.

Cleanup related to AC_VAL_MAX

Cleaned up a couple of issues in *ac_int* and *ac_fixed* related to the use of *AC_VAL_MAX* that did not impact functionality.

Changes in 3.9.2

Out of bound array write on ac_int % operator for wide operands

Fixed an out of bound array write access for the % operator of *ac_int* when the second operand requires fewer integers to represent than the first operand and at least one of the operands requires more three or more integers to represent. For example if the first operand is an *ac_int<24,true>* (requires one integer to represent) and the second operand is an *ac_int<70,true>* (requires three integers to represent), then the issue could be encountered due to an out of bound write on the array that holds the result.

Runtime error reported on Undefined Behavior Sanitizer on left shifts when first operand is of builtin type int

The *ac_int.h* and *ac_fixed.h* implementations have been updated to not use behavior that C++ 11 considers as *undefined* for the signed integer left shift *operator* <<. Prior to C++ 11, shifts were interpreted as a bit pattern. The fix is to cast the first operand to *unsigned* before performing the left shift and casting it back to *signed* when required.

The new version should not run into the *ubsan* runtime errors with respect to the undefined behavior for left shifts.

Changes in 3.9.1

Mixed operators for ac_int and ac_fixed

In previous versions, the mixed operators with *ac_int* and *ac_fixed* were defined under the namespace *ac::ops_with_other_types* and were made available with "*using namespace ac::ops_with_other_types;*".

Starting with v3.9.1, the operators are no longer under a namespace. To revert to the old behavior define the macros below before including any of the AC Datatype header files:

```
#define AC_INT_NS_FOR_MIXED_OPERATORS
#define AC_FIXED_NS_FOR_MIXED_OPERATORS
```

This change fixes very subtle issues on the operators not getting matched and implicit conversions for ac_int to *unsigned/signed long long* getting used instead. The old version also exposed a bug in GCC 4.9.2 (GCC 6.2.0 and above does not have that bug).

Whether the operator is found could affect behavior though in most cases the behavior might be the same. There are two cases of concern:

1. Shift left: *x* << *n*, where x is an *ac_int* and *n* is any C++ integer type.
    1. If the operator is found:
        1. the result of the shift is only as wide as *x*
        2. the result is well defined even for any *n*
    2. If the operator is not found (*x* is implicitly converted to signed/unsigned long long):
        1. the result type is 64 bits wide which means that some bits that otherwise would have been lost would be kept.
        2. the result is undefined for negative *n* or *n* > *63*
2. Operations that would produce result values that need more than 64 bits. This is only likely with wide multiplications. If the operator is not found, the result will be returned as a *signed/unsigned long long* instead of a wider *ac_int*.

Detailed description of the issue

It turns out the presence of another unrelated *operator* << (for example for std::ostream), can stop the compiler from considering operators defined in other namespaces even if they are made available with the using statement in a namespace that is being considered for lookup.

Some references on this C++ behavior:

https://en.cppreference.com/w/cpp/language/unqualified_lookup

https://en.cppreference.com/w/cpp/language/adl

The following example exposes the issue with v3.9.0 and earlier versions. When the operator is found, the program prints f0, when it is not (when macro EXERCISE_ISSUE is defined), the program prints ff0. Starting with v3.9.1, the *operator << (const ac_int<W,S> &, int)* is always found and the program prints f0.

```
namespace ns {
  struct my_type {};

#if EXERCISE_ISSUE
  // The presence of the ostream << operator breaks the visibility to
  //   the function ac::ops_with_other_types::operator << (const ac_int<W,S> &,
int)
  std::ostream& operator<<(std::ostream& os, const my_type& dt) { return os; }
#endif

  void foo() {
    ac_int<8,false> x = 0xff;
    int y = x << 4;
    std::cout << std::hex << y << std::endl;
  }
}

int main() {
  ns::foo();
}
```

Set Slice

Some compilers like CLANG have optimizations that can treat uninitialized variables as don't cares even though the result is deterministic. For example *x* ^ *x* is always zero even if *x* is uninitialized. CLANG optimizations will treat that expression as a don't care instead of zero.

The *set_slc* methods in *ac_int* and *ac_fixed* use XOR to write the appropriate slice. If the full slice is being assigned, the *set_slc* now uses the assign operator instead of relying on the XOR functionality to write the bits. A common example:

```
ac_fixed<8,4,true> x;
x.set_slc<8>(0, ac_int<8,true>(127));
```

For the code above, CLANG optimizations may leave *x* with the wrong value, with AC Datatypes v3.9.0 and earlier, because *x* is uninitialized when the *set_slc* is performed. Starting with v3.9.1, the result should be correct since all bits of *x* are being assigned.

Changes to bit_fill in ac_int

Some users encountered problems due to the mixed operator under the namespace problem described earlier.

The implementation now has explicit casts to avoid any potential issues in case the macros to revert to the old behavior are used.

Arithmetic assign operators for ac_float

The arithmetic assign operators +=, -=, *= and /= were missing the return statement. The return type was also changed to a reference.

Changes in 3.9.0

Updates to ac_channel

- Added *bind* methods for binding to SystemC *sc_fifo_in*<*T*> and *sc_fifo_out*<*T*>.
- Cleaned up implementation to remove SYNTHESIS specific exceptions.
- Changed from calling "*throw*" to calling *ac_assert* when the *read()* method is called on an empty channel. There are two ways the user may affect this behavior:

  1. Define the macro AC_USER_DEFINE_ASSERT to be the name of a function that will be called:

     AC_USER_DEFINED_ASSERT(condition, file, line, ac_channel_exception::msg(code))

     Note that this macro is also used in *ac_int.h* and redefines the behavior of AC_ASSERT.

  2. Define the macro AC_ASSERT_THROW_EXCEPTION to change the behavior from the default *assert* to a *throw*. This macro will not have any impact if the macro AC_USER_DEFINE_ASSERT is defined.

Fixes/Updates to ac_float

- Changed behavior of additive operators/methods to perform normalization before performing rounding. This is consistent with the IEEE floating point standard. For example the following two operations give the same answer:

  r.add(op1, op2)

  r = op1.to_ac_fixed() + op2.to_ac_fixed()

- Fixed issues with constructors from *ac_float* where *ac_float* is un-normalized (normalization had not

been attempted).

- Removed the "*normalize*" argument for the constructor from *ac_fixed*. The constructor will attempt to normalize the *ac_float* that is constructed.
- Factored code that deals with normalization, rounding and overflow saturation. The *leading_sign* method of *ac_fixed* is used instead of the *normalize* method of *ac_fixed*.

Fix of AC_ABS macro

Fixed issue with AC_ABS: it should be *-(a)* instead of *(-a)* as "*a*" may contain any expression.

Added hex and oct handling for ostream operator <<

Added handling of *std::ios::hex* and *std::ios::oct* for ostream operator <<.

Fencing for macro defines for true and false

The check if the keywords true and false are macro defined now results on a warning instead of an error. If either keyword is macro defined, it is explicitly undefined to prevent issues. The reason for this change is that there are header files from CLANG that have "#define true true". This seems like poor practice from CLANG as it prevents doing a legitimate check to fence against other macro defines that may have subtle effects.

CLANG Warnings

Fixed clang warnings in *ac_int*, *ac_fixed* and *ac_float*.

Changes in 3.8.1

Fixed trace functions to work with SystemC 2.3.2

The trace functions defined in ac_sc.h where updated to work with SystemC version 2.3.2. The SystemC standard lacks a general API to trace a class as anything other than tracing its datamembers. For now, the implementation of trace makes use of some SystemC implementation details that are not exposed in the API, but have been changing in different versions of SystemC.

Shift operator for ac_fixed with C integer types

The shift operator with the second argument being any of the C integer types, used to return the type of the first argument, but with default template arguments for the Q (quantization) and O (overflow) modes). Now it returns the type of the first argument which is consistent with other versions of the shift operator.

Removed operator %= for ac_fixed with ac_int

The mixed operator %= with second argument being ac_int was incorrectly defined and would fail to compile since the operator %= is not defined for ac_fixed. It has been removed.

GCC parentheses warnings

Addressed GCC warnings related to parentheses.

Changes in 3.7.2

Fix for *to_string* Method

Fixed width of string returned by *to_string* for a variable of type *ac_int* or *ac_fixed* that can happen when *to_string* is called before the variable has been initialized.

Workaround Fix for Visual C++ 2015 Bug

Implemented workaround solution to bug introduced in Visual C++ 2015 that incorrectly errors out on a typedef that depends on an enumeration in the struct *int_range* that is part of the *ac_int.h* implementation.

Warnings

Addressed new *clang++* warnings for *ac_int.h* and *ac_fixed.h* that appear with newer versions of *clang*. Also addressed warnings that appear with *clang -std=c++11*. Version 4.0 of *clang* was used for checking warnings.

Changes in 3.7.1

Fix to ac_float

This version corrects an issue with *ac_float* that was introduced in v3.6. The *ac_float* that is needed to represent an unsigned *ac_fixed*, *ac_int* or native C unsigned integer type did not take into account that an additional bit is required. This issue affects the mixed operators of *ac_float* with other types.

Changes in 3.7

Change to Apache License

Changed license to Apache License Version 2.0.

Changes and Enhancements in 3.6

Bit Fill

Utility functions to initialize large bitwidth *ac_int* and *ac_fixed* with raw bits have been added. 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 functions are called *bit_fill_hex* and *bit_fill*. The *bit_fill_hex* accepts a hex string. It **should only** be used to initialize **static** constants since it is significantly slower than alternative methods. 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.

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 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
```

Bit Complement

The **bit_complement** member function has been added for *ac_int* and *ac_fixed*:

```
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;
```

It returns 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
```

Restructured and Enhanced Type Infrastructure

The following changes were done to improve the separation of functionality that is meant to be exposed to the user and functionality that is specific of the implementation. Some of the type definition infrastructure was moved from namespace *ac* to namespace *ac_private*. They include all the *rt_ac_int_T*, *rt_ac_fixed_T*, *rt_ac_float_T* and *rt_ac_complex_T*. The exposed functionality is *ac_int::rt_T*, *ac_fixed::rt_T*, *ac_float::rt_T* and *ac_complex::rt_T* and *ac::rt_2T* (renamed from *rt2*). The trait mechanism for C++ integers and floating point types *c_type*, *c_type_params*, *map*, *c_arith*, *c_prom* and *rt_c_type_T* are now part of namespace *ac_private*.

The type definitions for *ac_int_represent<T>::type*, *ac_fixed_represent<T>::type* and *ac_float_represent<T>::type* have been added to facilitate finding a minimal destination type that can represent the source type *T*.

Static Const Members

For consistency, the static const members *e_width* was added to *ac_int* and *ac_fixed* and the *o_mode* was added to *ac_float*.

Warnings

Warnings that are disabled for GCC and that affect *clang* are now also disabled for *clang*.

Documentation

A new section titled *Reference Guide for Numerical Algorithmic C Datatypes* has been added. This section summarizes all the user visible available functionality for all the numerical datatypes in a consolidated way.

Corrected Problems

The return type for *ac::frexp_f, ac::frexp_d*, *ac::frexp_sm_f*, *ac_frexp_sm_d* was adjusted to move the fixed-

point one position to the right (the width has not changed). For example the *ac::frexp_sm_f* now returns the mantissa as *ac_fixed<24,1,false>* instead of *ac_fixed<24,0,false>.* The change was made because while it was consistent with the system function *frexp*, it requires an exponent of *ac_int<9,true>* instead of an *ac_int<8,true>* since the exponent has range of -125 to 128. The implementation excluded the exponent value of 128 (assert in simulation) which was not correct.

The new return type is consistent with the IEEE representation of normalized numbers as 1.m where the returned mantissa includes the implied most significant '1' bit. With that representation the exponent range is -126 to 127 which can be stored in an *ac_int<8,true>*.

The *ac_float* type that is used to represent a float or a double was also change accordingly. A float is now represented as an *ac_float<25,2,8>* instead of an *ac_float<25,1,8>*.

Changes and Enhancements in 3.5

The return type of the << and >> operators for *ac_fixed* was changed to the type of the first operand. Prior to 3.5, the returned type was that of the first operand, but with default parameters for rounding and overflow (AC_TRN, AC_WRAP). This makes it consistent with the changes that were done in 3.1 to shift-assign <<= and >>= operators. For example:

```
typedef ac_fixed<4,4,true,AC_TRN,AC_SAT_SYM> fx_ss;      // Symmetrical range
(-7 to 7)
fx_ss a = 1;
fx_ss b = a << 3;    //  a <<3 and b are now 1000 (-8 => not symmetrically sat-
urated)
a <<= 3;                //  a is (since v3.1) 1000 (-8 => not symmetrically
saturated)
```

Setting bits (using operator [], or set_slc) and the shift and shift-assign operators should be avoided with AC_SAT_SYM. Forcing symmetric saturation on the example above can be done by casting to non-symmetrically saturated type:

```
a = (ac_fixed<4,4,AC_TRN>) a;     // -8 is saturated to -7
```

Corrected Problems

Warnings about parentheses have been addressed. Also the disabling of specific GCC warnings on sections of *ac_int.h* and *ac_fixed.h* have been updated so they will work with GCC versions 5.0 and above.

Changes and Enhancements in 3.4

The following enhancements were made in 3.4:

- Added support for the SystemC 2.3.1 maintenance release.
- The sc_trace() functionality for AC Datatypes has been upgraded to support the System-C 2.3.1 distribution. No changes to user code is required.

- Fixed clang++ warnings in *ac_int.h* and *ac_fixed.h*.

- Fixed *gdb* pretty print (*ac_pp.py*) to work around an issue with early versions of *gdb* (for example *gdb7.2-56*).

- Changed the constructor of *ac_float* from *ac_fixed* (indirectly also affect constructor from *ac_int*).

  The change now takes into account the differences of the *I* parameter of the source *ac_fixed* and the target *ac_float*. This change enables better normalization that considers not only the range of the target exponent (given by *E* of the *ac_float*), but also the difference between the source and target parameter *I* ("exponent bias").

- Removed normalization call from the '*' operator since result will be normalized (at most off by one 1-bit shift) if inputs are normalized. Also removed normalization call from construction from *float* and *double* since it is assigning to *ac_float* types that are assumed to capture it without loss of precision and the source is already assumed to be normalized.

  Also fixing one constructor when normalization is set to false.

- Fixed issue of *ac_float* overflowing when getting assigned/constructed from a larger bitwidth *ac_float* (could affect constructors from *ac_int* and *ac_fixed* as they use the constructor from *ac_float*). An extra bit of precision needed to be used to account for rounding in an intermediate computation.

Changes and Enhancements in 3.3

Added ac_channel class

When describing a hierarchical system using C function calls, the AC (Algorithmic C) channel class simplifies the synthesis and modeling with a minimal impact on coding style and C simulation performance.

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).

Changes and Enhancements in 3.2.1

Enhancements

The following enhancements were made in 3.2.1:

- The headers were updated to reduce warnings with GCC.

Changes and Enhancements in 3.2

The following enhancements were made in 3.2:

- **Added Reduce Methods**. Added reduce methods *and_reduce()*, *or_reduce()* and *xor_reduce()* to ac_int.
- **Default Constructor**. Added a way to guarantee that un-initialized AC Datatypes (ac_int, ac_fixed, ac_float) are adjusted to be in their numerical range. It is done by defining the macro AC_DEFAULT_IN_RANGE before the first inclusion of the AC Datatype header.

Corrected Problems

The following fixes were made in ac_sc.h:

- Fixed sc_trace issue with wrong VCD produced for signed AC Datatypes.
- Fixed compilation error when systemc is included rather than systemc.h.
- Fixes for SystemC version check and inclusion.

Changes and Enhancements in 3.1

Changes to ac_fixed with Symmetric Saturation

The constructor of ac_fixed was changed when the overflow mode is set to AC_SAT_SYM and the argument is also an ac_fixed with overflow mode set to AC_SAT_SYM. This change assumes that if the argument is of overflow type AC_SAT_SYM, it is already symmetrically saturated and therefore there is no need to repeat the symmetric saturation.

This should not change the behavior compared to previous releases of this package unless any of the following operators/methods are used that might invalidate the symmetric saturation property:

- modifying a bit (assigning to a bit reference)
- modifying a slice (set_slc)
- shift-assign (<<=, >>=)

In order to preserve the symmetric saturation property of ac_fixed with overflow mode set to AC_SAT_SYM, it is advisable to avoid the above methods on variables of that type. For example:

```
typedef ac_fixed<8,8,true,AC_TRN,AC_SAT_SYM> fx;
fx a = 0;
a[7] = 1; // No longer symmetrically saturated
fx b = a; // b remains unsaturated as a is assumed to be saturated and
            // has identical type (this is the behavior from v3.1 onwards)
```

This change was done for the following reasons:

- Minimize the need for symmetric saturation to reduce the overhead in simulation and the hardware required to implement this functionality. If the above methods are avoided, this saturation was entirely superfluous.

- The compiler is allowed to optimize copy constructor calls ("constructor elision" or "Return value optimization") so it was necessary to change the copy constructor to not perform any saturation during the copy constructor call. The change does not only affect copy constructors, but in more general situations that is easy to describe (if the argument is ac_fixed with AC_SAT_SYM it is assumed to be symmetrically saturated).

Another change is that the shift-assign operators will not perform any saturation. This change only affects scenarios where the first operand is an ac_fixed type with overflow mode set to AC_SAT_SYM. For example:

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

The reason for the change is that this was the only exception to the rule that shift assign operators do not have any cost in terms of saturation or rounding.

Reducing Unnecessary Warnings

Certain functionality in ac_int/ac_fixed intentionally uses uninitialized variables to emulate a don't care value (AC_VAL_DC). This can create many warnings when -Wall is used with GCC. The compiler version GCC4.6 introduced a feature that allows locally disabling warnings on sections of code. The header files ac_int.h and ac_fixed.h have been enhanced to use this feature.

Pretty print in GDB

Newer versions of GDB allow pretty printers to be provided as python scripts. A file ac_pp.py is now available that provides pretty printer capabilities for ac_int, ac_fixed, ac_float and ac_complex. Some parameters provide control on the radix format (decimal, hexadecimal or binary). The header comments in the script provide the information on how to use it.

Corrected Problems

None.

Changes and Enhancements in 3.0

A new quantization mode, AC_RND_CONV_ODD has been added. This quantization mode rounds towards odd multiples of the quantization. Refer to the Algorithmic C Datatype documentation for details.

Corrected Problems

The following customer reported problem was fixed in this release.

- **DR 756512** GCC4.3 -Wall verbosity increased significantly for Catapult headers. See "Excessive Compiler Warnings" on page 20.

Known Problems and Workarounds

Excessive Compiler Warnings

Newer versions of GCC and Visual C++ introduce many additional warning messages when the -Wall option is used.

The header files *ac_int.h* and *ac_fixed.h* are updated to avoid such warnings. For example, one of the new warnings for GCC advises the use of parentheses in expressions such as:

```
A && B || C
```

Prior to this change, the workaround was to use the -Wno-parentheses option in GCC.

Warnings in Visual C++ have also been addressed by either a source change or disabling the warning locally (does not affect code that includes the header files). However, Visual C++ 10 still reports numerous warnings when using the -Wall option. The warnings are mainly of the type C4514 "unreferenced inline function has been removed" and appear despite the fact that both ac_int.h and ac_fixed.h explicitly disable that warning number (appears to be a bug in Visual C++ warning system). Such warnings are also reported for system header files that are part of Visual C++.

Supported Compilers

The ac_int, ac_fixed and ac_complex classes rely heavily on template mechanisms to achieve efficient simulation runtimes. We recommend that you use the following versions of GCC (GNU Compiler Collection) and Microsoft compilers.

GCC 3.2.3 or later

It is also important to run the compiler with optimizations turned on in order to get the best runtime performance:

```
c++ -O3 -I$MGC_HOME/shared/include test.cxx -o test
```

Optimization level O3 is recommended, although O1 in most cases delivers most of the benefit (20x runtime improvement has been seen by going from O0 (no optimization) to O1).

You can obtain gcc compilers from the GNU web site: http://gcc.gnu.org.

Microsoft Visual C++ 2008

To download and install Microsoft Visual C++ 2008, go to the Microsoft web site and follow the instructions on the web page:

```
http://msdn.microsoft.com/visualc
```

You can also download and install a free version called "Visual C++ 2008 Express":

```
http://www.microsoft.com/express/download/#webInstall
```