Algorithmic C (AC) Datatypes
Release Notes

Software Version v3.7.2

May 2017

Table of Contents

Table of Contents

Bit-Accurate Datatypes

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

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

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
bigendian=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 bigendian=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 saturated)
 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 9.

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 http://msdn.microsoft.com/visualc and follow the instructions on the web page. You can also download and install a free version called "Visual C++ 2008 Express" http://www.microsoft.com/express/download/#webInstall.