

# Fast Bit-Accurate C++ Datatypes for Functional System Verification and Synthesis

Andres Takach

Mentor Graphics Corporation  
8005 S.W. Boeckman Rd  
Wilsonville, OR 97070 USA  
andres\_takach@mentor.com

Peter Gutberlet

Mentor Graphics Corporation  
8005 S.W. Boeckman Rd  
Wilsonville, OR 97070 USA  
peter\_gutberlet@mentor.com

Simon Waters

Mentor Graphics Corporation  
8005 S.W. Boeckman Rd  
Wilsonville, OR 97070 USA  
simon\_waters@mentor.com

## Abstract

*Fast bit-accurate simulation is essential for the functional verification of complex algorithms used in modern digital hardware designs. In this paper, we present the design of C++ templated classes that model arbitrary-length bit-accurate integer and fixed-point arithmetic. The bit-accurate types<sup>1</sup> have been designed to have well defined semantics for hardware synthesis. DSP algorithms written using the new bit-accurate integer and fixed-point types are shown to have 3x to 200x faster runtimes than the same algorithms written with the corresponding SystemC bit-accurate types.*

## 1. Introduction

An important step in the implementation of DSP hardware or software is refining the algorithm from using floating-point arithmetic to fixed-point or integer arithmetic. This process is typically performed in C or MATLAB. In order to model bit-accurate arithmetic, designers often use C bit fields or C integer types in combination with masking or shifting. Another approach is to use the bit-accurate data types that are provided as part of the SystemC [3] language. The SystemC data types encapsulate bit-accurate arithmetic for integers and fixed-point using C++ templated classes. The classes may be used as part of a complete SystemC verification environment (for example for TLM Modeling) or used in a plain C++ algorithmic description. The data type subset of SystemC is functionally orthogonal to other core features of SystemC that are required to model concurrency, structure, timing, etc.

Functional simulation speed is becoming increasingly important in system-level design. Simulation speed impacts the development time of the system-level functional specification since a number of iterations may be necessary before the specification converges to the

desired behavior. Functional testing needs to be as comprehensive as possible and ideally should include modeling of the finite precision computation that will be used in the implementation. Because of long simulation times, finite precision computation is commonly not modeled from the start. Instead, either plain C integers or floating-point arithmetic is used in the earlier versions of an algorithm block. Finite-precision models are often coded later and do not receive the same level of testing as the models using C integers and floating point types. As a consequence, some design problems may not be discovered until later, during hardware emulation or FPGA prototyping of the design. As the results of this paper show, modeling of finite precision using our fixed-point implementation is actually significantly faster than using floating-point precision.

SystemC has received much attention in the research community for its capabilities to model systems at a higher level of abstraction [4] [5] and as a vehicle for exploring new ideas in system-level specification and verification. Much of the focus has been concentrated on analyzing, improving or extending the kernel of SystemC and adding new verification capabilities to it. For example, Perez *et al.* [6] improve the performance of the SystemC engine by reducing the number of process wake-up calls. Siegmund *et al.* [7] propose an extension to SystemC to enable the separation of the specification of communication and functionality.

The definition of the formal semantics of the SystemC language has also been a topic of considerable interest [8] [9] [10]. The datatype subset of SystemC, on the other hand, has received little attention. SystemC datatypes are well known and widely used for system-level and hardware verification. The datatypes include arbitrary-length bit-accurate types for integers, fixed-point and bit and logic vectors. The static nature of the precision makes it well-suited for specifying hardware. The implementation is open (sources are public) and can be compiled with non-proprietary C++ compilers.

While the current set of data types provided with SystemC are widely used for verification of algorithms, a

---

1. The datatypes are part of the *Algorithmic C<sup>TM</sup> Datatypes* and are freely available at:  
[http://www.mentor.com/products/esl/high\\_level\\_synthesis/ac\\_datatypes](http://www.mentor.com/products/esl/high_level_synthesis/ac_datatypes).

number of issues are still of concern to the user community:

- *Long runtimes*: runtimes can be anywhere from 5x to 200x slower than using C types directly.
- *Non-unified types*: two versions for bit-accurate integer types exist: the “fast” but limited to 64 bits of precision (`sc_int/sc_uint`), and the arbitrary precision integer type (`sc_bigint/sc_bignint`) that is much slower. The two datatypes are not equivalent even for bitwidths where the 64-bit limit should not matter. For example, the comparison `(sc_uint<6>) 1 > (sc_int<6>) -1` returns false, while the same comparison using `sc_bigint/sc_bignint` returns true. There are also two versions of the fixed-point type: the “fast” but limited to 53 bits (precision of a typical C++ double) called `sc_fixed_fast/sc_ufixed_fast`, and the arbitrary precision version `sc_fixed/sc_ufixed`.
- *Imprecise definition*: the definition of the data types is not formally defined, but rather it is implicit in the implementation. Improving the current implementation is difficult without a very precise definition of all operators and methods. There is also some inconsistencies among the different datatypes that need to be resolved.
- *Lack of comprehensive verification*: much of the functionality of SystemC has been tested by the user community. Such informal testing is slow, costly and not comprehensive. For instance, despite the widespread usage of SystemC, problems such as incorrect simulation are still being reported for its datatypes. A more comprehensive testing is essential to allow improvements to be made to the implementation without the risk of introducing new problems.

The unification of the limited and the arbitrary precision datatypes is essential for a hardware or system description language. Arbitrary precision types have clear advantages over limited precision types. Because arbitrary precision types don’t lose precision for arithmetic operations, their semantics are cleaner and more intuitive than the semantics of limited precision types. Arbitrary precision is also desirable when defining parametrized C source IP where the functionality should be correct for all input bit widths, rather than have an artificial limit that then needs to be quantified and documented.

Efficient packages for arbitrary precision integer arithmetic are publicly available. An excellent example of such a package is *GNU MP* [9] which provides arbitrary dynamic precision with a granularity of C integers (32 bits) or *long long* integers (64 bits). The target applications are mathematical problems requiring very large bitwidths and therefore the package is optimized for handling very large bitwidths efficiently. It is not optimized for bitwidths used

in typical hardware or embedded systems and in fact the dynamic nature of its precision is not suitable for hardware or embedded software specifications.

In this paper, we present arbitrary-length, bit-accurate integer and fixed-point datatypes that are 3x to 200x faster than the equivalent datatypes in SystemC and come close in performance to the built-in C integer datatypes. The speed advantage is achieved with regular C++ compilers and is the result of careful definition of the semantics and an optimized implementation C++ implementation that relies on template specialization to provide efficient implementations for all operators.

The definition of the semantics of the datatype described here is meant to be easily synthesizable. For example, all operators (including the left shift) produce results that have statically determinable bit widths. Once the semantics are defined, a combination of equivalence checking and simulation is used to verify that the implementation accurately reflects the defined semantics for a range of input bitwidth combinations.

The datatype is implemented as a templated class that could be used instead of the SystemC datatypes `sc_int/sc_uint` or `sc_bigint/sc_bignint` in pure C++ or SystemC specifications.

Section 2 introduces the requirements on the behavior of our bit-accurate datatypes. Section 3 describes the ideas used to obtain a fast implementation. Section 4 briefly describes the methodology to verify the correctness of the implementation. Section 5 presents runtime comparisons to built-in C++ datatypes and SystemC integer datatypes `sc_int` and `sc_bigint` obtained with a small benchmark example and for a set of algorithms.

## 2. Semantic Definition

The most important step in developing a datatype is to define the desired semantics. The criteria used to define the semantics are the following:

- The datatype should be intuitive to use: in general this means that operations should return full precision whenever possible rather than be truncated by limits due to implementation details. A consequence of such a semantic is that arithmetic properties are preserved, for example  $(a*b)*c == a*(b*c)$ .
- The datatype should have clean hardware synthesis semantics: the bitwidth of the return value for operations should be statically derivable from the bitwidths of the operands.
- The datatype should be runtime efficient: how operations are defined may affect runtime. The implementation should not require dynamic memory allocation/deallocation (new/delete). Operations should be statically determinable to facilitate compiler

optimizations such as function inlining.

The last two requirements are consistent. Making the bitwidth of results be statically determinable helps both software compilers and synthesis tools. For software compilers, what is statically determinable is confined to expressions on constants and template arguments. For most operations, the bitwidth for the result that guarantees no loss of precision is a simple function on the bitwidths of the operands. For example, for multiplication, the bitwidth of the result involves adding the bitwidths of the two operands.

The shift left operation is an interesting case that presents a challenge to one or more of our goals. The shift left operation  $a \ll b$  can be viewed as either a bitwise operation or an arithmetic operation that does not lose precision:

- Bitwise: the result bitwidth is the bitwidth of the first operand, i.e.,  $a \ll b = (a * 2^b) \bmod 2^{\text{bitwidth}(a)}$  for  $b \geq 0$
- Arithmetic: no loss of precision, i.e.  $a \ll b = a * 2^b$  for  $b \geq 0$ .

In many cases the arithmetic view might be more intuitive. However, while the bitwidth of the return type can be derived from the bitwidth of the operands, it is impractical unless the bitwidth of  $b$  is very small. SystemC takes the arithmetic view which has negative consequences for both runtime efficiency as well as synthesis semantics. For example, the bitwidth of  $(a \ll b) * c$  is not statically determinable by a compiler and thus the techniques described in Section 3 for efficient runtimes cannot be applied. In addition, dynamic memory allocation is required, which further degrades runtime. Synthesis, on the other hand, can only efficiently handle such a situation if it can determine that  $b$  is a constant or a small range.

### 3. Implementation

The bit-accurate integer and fixed-point datatypes are implemented as templated classes, named *ac\_int* and *ac\_fixed*. The *ac\_int* class takes two parameters as shown in Figure 1. The first parameter specifies the bitwidth of the integer and the second parameter specifies whether the integer is signed or unsigned. For example:

```
ac_int<4,true> x;      // 4-bit signed integer
ac_int<73,false> y;    // 73-bit unsigned
integer.
```

The *ac\_fixed* class takes 5 template parameters:

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

where  $S$  defines whether the type is signed or unsigned. The remaining parameters are consistent with SystemC's fixed-point datatypes. The  $Q$  and  $O$  template arguments define the quantization and overflow modes and default truncation and wrap respectively (equivalent to SC\_TRN

and SC\_WRAP in SystemC). For example:

```
ac_fixed<5,3,true> x; // bbb.bb
ac_fixed<5,-2,true> y; // .xxbbbbbb
```

The *ac\_int* and *ac\_fixed* classes are derived from class *iv* (*iv* stands for integer vector) that implements signed integer datatypes of bitwidths multiples of 32. Figure 1 also shows the template member function for the multiplication operator for *ac\_int*. Note that the bitwidth and signedness of the result is a function of the class template parameters  $W$  and  $S$  and the member function template parameters  $W2$  and  $S2$ .

```
template<int W, bool S=true>
class ac_int : private ac::iv<(W+31+S)/32> {
#pragma builtin
    enum {N=(W+31+S)/32};
    // ...
public:
    template<int W2, bool S2>
    ac_int<W+W2,S||S2> operator *(const ac_int<W2,S2>
    &op2) const {
        ac_int<W+W2, S||S2> r;
        mult(op2, r);
        return r;
    }
    template<int W2, bool S2>
    ac_int &operator *=( const ac_int<W2,S2> &op2) {
        ac::iv<N> r;
        mult(op2, r);
        ac::iv<N>::operator=(r);
        bit_adjust();
        return *this;
    }
    // ...
};
```

Figure 1: Fragment of the *ac\_int* class

For example, multiplying an *ac\_int*<36,true> (base class is *iv*<2>) by *ac\_int*<14,false> (base class is *iv*<1>) produces a result which is *ac\_int*<51,true> (base class is *iv*<2>). The actual multiplication is performed by the call *mult*(*op2, r*) that in turn calls the member function *mult* of the base class: *iv*<2>::*mult*(const *iv*<1> &*op2, iv*<2> &*r*). The base class *iv* does not know the exact bitwidth of the operands except that they can be represented by 64-bit signed and 32-bit signed integers respectively. The argument for the return value *r* provides the template parameter for *mult* so that an *iv*<2> rather than an *iv*<3> result is computed.

Operations where loss of precision is possible, for example operations involving assignment, require the call to the private member function *bit\_adjust* that adjusts the result computed by the base class *iv* to take into account the actual bitwidth of the number. For example, Figure 1 shows the implementation for the member function for the multiply assign operator “\*=". If the left operand is an *ac\_int*<14, false> and the right operand is an

`ac_int<36,true>`, the multiplication is performed with the member `iv<1>::mult(const iv<2> &op2, iv<1> &r)`, resulting in a 32-bit signed value that needs to be adjusted to a 14-bit unsigned number. Note that all bitwidths are statically determinable by the C++ compiler based on template parameters. Also no dynamic memory allocation or deallocation is used.

Template specializations [1][2] are used to provide implementations that are as efficient as possible for smaller bitwidths, as they are used extensively in hardware design. Template specialization allows for alternative implementations for particular template parameter values so that they are as efficient as possible. Figure 2 shows the template specializations for `iv_mult`, a function called by member function `iv::mult`. For instance, `iv<3>::mult(const iv<2> &op2, iv<5> &r)` would call `iv_mult` with parameters `N1=3, N2=2, Nr=5`. The details for the `iv_mult` function are not shown, but it involves behavior consisting of two nested loops to perform the multiplication using the built-in C++ type `long long`. Figure 2 also shows two template specializations that provide more efficient implementations for two very commonly occurring cases.

```
template<int N1, int N2, int Nr>
static void iv_mult(const int *op1, const int
*op2, int *r) {
    // general function for arbitrary N1, N2, Nr
    using loops
    // ...
}
// template specialization for N1=1, N2=2, Nr=1
template<> inline void iv_mult<1,1,1>(const int
*op1, const int *op2, int *r) {
    r[0] = op1[0] * op2[0];
}
// template specialization for N1=1, N2=2, Nr=2
template<> inline void iv_mult<1,1,2>(const int
*op1, const int *op2, int *r) {
    iv_assign_int64<2>(r, ((long long) op1[0]) *
((long long) op2[0]));
}
```

**Figure 2: Using template specialization for better runtime performance**

Template specialization leads to faster executables than using the non-specialized function even if the code of the non-specialized function is written in a way that would allow a C++ compiler to easily generate equivalent code. For example, instead of specialized functions, the exceptions could be coded directly as shown in Figure 3.

Analysis of the generated assembly code indicates that the alternative implementation of Figure 3 is not inlined. Whether function inlining occurs depends on a measure of the complexity of the function to be inlined. That

complexity (at least in gcc) appears to be measured before some of the optimizations that would simplify the function are performed.

```
template<int N1, int N2, int Nr>
static void iv_mult(const int *op1, const int *op2,
int *r) {
    if(N1==1 && N2==1) {
        if(Nr==1)
            r[0] = op1[0] * op2[0];
        else
            iv_assign_int64<2>(r, ((long long) op1[0]) *
((long long) op2[0]));
    }
    else {
        // general function for arbitrary N1, N2, Nr
        using loops
        // ...
    }
}
```

**Figure 3: Alternative to using template specialization**

## 4. Verification

Verification of the correctness of the implementation was done by first synthesizing the C models for each operator/method for many parameter combinations using *Catapult C Synthesis<sup>TM</sup>* [12]. The RTL generated by synthesis is then checked against generic VHDL golden references using a combination of equivalent checking and simulation.

## 5. Results

The main focus of this section is to quantify the runtime performance of `ac_int` and `ac_fixed` against the corresponding SystemC bit-accurate datatypes.

### 5.1 Bit-Accurate Integer Types

A small example with arithmetic and shift operations was written and run with four datatypes: `ac_int`, `sc_bigint`, `sc_int` and built-in C++ integer types. The example uses a single integer type `int_type` which is type defined to any of the integer type to be benchmarked. The operations covered here were chosen since they are supported for all four datatypes to be benchmarked. The runtime results may also be sensitive to the input data for `sc_bigint` since some implementations may provide exceptions for values like 0, 1 and -1 for efficiency at larger bitwidths. Finding a representative input data set depends on the application. For the benchmark example, the input set was designed to avoid having highly skewed data, such as having the most significant 240 bits of a 256 bit integer be zero for most input vectors.

```

void f(int_type x, int_type &y) {
    int_type w = x * y;  bv v = w >> 2;
    int_type u = w - v;  u *= v;
    u -= y;  u >>= 3;
    u++;  u <= 2;
    y = u + w;
}
main () {
    int_type a, b;
    for(int i =0; i < 3000; i++) {
        for(int k=0; k < 3000; k++) {
            a *= i;  a += k;
            b *= k;  b += i;
            f(a, b);
        }
    }
    cout << b << endl;
}

```

**Figure 4: Example to benchmark ac\_int.**

The runtime results are shown in Table 1. All versions were compiled with the GNU gcc3.4.3 C++ compiler with the optimization flag set to O3 on a 1700MHz Intel Pentium-M running linux RedHat9. The main focus of the comparison is between the runtimes using ac\_int and sc\_bigint as they offer comparable arbitrary-length, bit-accurate functionality. For bitwidths 32 or less, ac\_int is roughly 100 times faster than sc\_bigint. As the bitwidth increases the speed advantage is reduced, but it is still a factor of roughly three times faster at 256 bits. Much of the work to optimize runtime in ac\_int was focused for smaller bitwidths since they are heavily used in hardware design. It is possible to introduce additional template specializations to further improve bitwidths of 64 and beyond.

Comparing runtimes of ac\_int with sc\_int leads to less dramatic results, but it remarkable to see a factor of three to four speed advantage over sc\_int for bitwidths of 32 and below (the most used bitwidths for hardware designers). Note that sc\_int<64> (or the C++ *long long* (64 bits)) are not equivalent to ac\_int<64,true> since the computed results are limited to 64-bits.

**Table 1: Comparison of signed ac\_int relative to SystemC signed integer types**

bit width	Runtime ac_int (s)	Speed-up Factor Runtime(type)/Runtime(ac_int)	
		sc_bigint	sc_int
8	0.29	103.8	3.72
9	0.30	100.7	3.60
12	0.30	101.0	3.60
16	0.29	104.5	3.72

**Table 1: Comparison of signed ac\_int relative to SystemC signed integer types**

bit width	Runtime ac_int (s)	Speed-up Factor Runtime(type)/Runtime(ac_int)	
		sc_bigint	sc_int
32	0.35	108.0	2.97
64	4.22	9.60	Not Equiv
128	9.18	4.97	NA
256	23.7	2.80	NA

The benchmark was also run with native C++ datatypes. Using the C++ signed char (8 bits) was a factor of 1.16 faster than ac\_int<8,true>. Using the C++ signed short (16 bits) was 1.27 faster than ac\_int<16,true>. The C++ type *int* (32 bits) is not equivalent to ac\_int<32,true> since the results are computed as 32 bit values. The runtime performance of ac\_int is very close to the theoretical limits given by the underlying C++ datatypes.

Two additional metrics of interest are compilation times and size of the executable. The compilation time is fairly unaffected by bitwidth and is roughly 10 times faster for ac\_int than for sc\_int or sc\_bigint (0.8s compared to 7.5s). The compilation time of ac\_int is only a factor of 1.24 longer than the compilation time of the built-in datatypes. The size of the executable for sc\_int and sc\_uint is unaffected by bitwidth: 732k for sc\_int and 673k for sc\_bigint. The size of the executable for the built in datatypes also is not affected by bitwidth and it was much smaller at 13.5k. The size of the executable for ac\_int was very close to the size for built in datatypes but grew from 14.1k to 18k as the bitwidth grew from 8 to 256.

The speedup of ac\_int compared to sc\_bigint on actual designs was consistent with the speedups measured for the small benchmark. Table 2 shows two designs and the speedups with respect to sc\_bigint. The speedup of ac\_int compared to sc\_int for DCT was 17.7 which is actually quite larger than what it would expected from the small benchmark. The SQRT design is an integer square root function.

**Table 2: Speed up of ac\_int compared to sc\_bigint**

Design	Speed up Factor
DCT (8x8)	119
SQRT (16-bit)	143

## 5.2 Bit-Accurate Fixed-Point Types

One way to measure how efficient a bit-accurate fixed-point type is to use a fixed-point variable to model a bit-

accurate integer. Ideally, there should be no overhead. For instance, an `ac_fixed<16,16,true,AC_TRN,AC_WRAP>` is used instead of an `ac_int<16,true>`.

Table 3 shows a runtime comparison for `ac_fixed` and the SystemC fixed-point types `sc_fixed` and `sc_fixed_fast` when used as bit-accurate datatypes for the benchmark example shown in Figure 4. In this scenario, `ac_fixed` is roughly 200x faster than the arbitrary-length datatype `sc_fixed` and 56x to 90x faster than the “fast” fixed-point type `sc_fixed_fast` (limited to 53 bits). As a comparison, the runtimes for both float and double (with the appropriate changes to the code) is 11.2s. The `ac_fixed` in this case, is 3x to 30x faster than using float or double.

**Table 3: Comparison of signed `ac_fixed` relative SystemC fixed-point types**

bit width	Runtime <code>ac_fixed</code> (s)	Speed-up Factor Runtime(type)/Runtime( <code>ac_fixed</code> )	
		<code>sc_fixed</code>	<code>sc_fixed_fast</code>
8	0.34	200.3	87.3
9	0.36	188.9	84.1
12	0.34	200.0	87.4
16	0.29	232.8	98.9
32	0.51	132.0	55.9
64	4.4	16.0	Not Equiv
128	9.6	10.9	NA
256	25.3	7.2	NA

The speedups of `ac_fixed` compared to `sc_fixed` for two small designs is shown in Table 4. The SQRT design is a fixed-point square root function.

**Table 4: Speedup of `ac_fixed` compared to `sc_fixed`**

Design	Speedup factor
CORDIC (cos)	34
SQRT (16-bit)	141

## 6. Conclusions

The design and verification of fast arbitrary-length bit-accurate integer and fixed-point datatypes was presented. The datatypes are implemented as a C++ template class and can be used in C++ or SystemC code. Results show that the `ac_int` and `ac_fixed` datatypes outperform the corresponding SystemC’s datatypes. For bitwidths most commonly used for hardware designs (32 bit and below), `ac_int` is roughly 100 times faster than `sc_bigint`, three to

four times faster than SystemC’s limited-precision “fast” integer `sc_int` and comes very close to C++ built-in integer datatypes. Likewise, the `ac_fixed` type is roughly 200 times faster than `sc_fixed` on bitwidths smaller than 32 bits.

The Compile times of C++ code using `ac_int` or `ac_fixed` are a factor of 10 faster than using SystemC datatypes. The `ac_int` and `ac_fixed` datatypes also provided an advantage in terms of executable size over SystemC integer datatypes.

Both the `ac_int` and `ac_fixed` datatypes are synthesized by *Catapult C Synthesis<sup>TM</sup>* [12], a synthesis tool that takes algorithmic specifications in ANSI C++ and generates optimized RTL for either ASIC and FPGA technologies.

## 7. References

- [1] Stroustrup, Bjarne. The C++ Programming Language. 3rd edition. Addison-Wesley.
- [2] Programming Languages - C++. ISO/IEC 14882:1998(E). First edition 1998-09-01. American National Standards Institute.
- [3] SystemC Language Reference Manual 2.0.1, <http://www.systemc.org>.
- [4] Grotker, T., Liao Stan, Martin, G. and Swan, S. “System Design with SystemC.” Kluwer Academic Publishers. 2002
- [5] Ghosh, A., Tjiang, S., Chandra, R. “System Modeling with SystemC.” ASIC, 2001. Proceedings. 4th International Conference on, 23-25 Oct. 2001. Pages:18 - 20
- [6] Perez, D.G., Mouchard, G. and Temam, O. “A new optimized implementation of the SystemC engine using acyclic scheduling.” Design, Automation and Test in Europe Conference, 2004. Proceedings, Feb. 2004 Pages: 552 - 557.
- [7] Siegmund, R. and Muller, D. “SystemC<sup>SV</sup>: an extension of SystemC for mixed multi-level communication modeling and interface-based system design.” Design, Automation and Test in Europe Conference, 2001. Proceedings, March 2001. Pages:26 - 32.
- [8] Salem, A. “Formal semantics of synchronous SystemC.” Design, Automation and Test in Europe Conference, 2003, Pages: 376 - 381.
- [9] Mueller, W., Ruf, J., Hoffmann, D., Gerlach, J., Kropf, T. and Rosenstiehl, W. “The simulation semantics of SystemC.” Design, Automation and Test in Europe, 2001. Proceedings, March 2001. Pages: 64 - 70
- [10] Man, K.L. “SystemC/sup FL/: formalization of SystemC.” Electrotechnical Conference, 2004. MELECON 2004. Proceedings of the 12th IEEE Mediterranean, 12-15 May 2004. Pages:201 - 204.
- [11] GNU MP, <http://gnu.org/software/gmp>
- [12] Mentor Graphics Corporation, <http://www.mentor.com/>