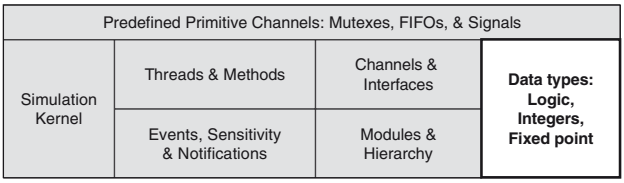
Chapter 3
Data Types



This chapter provides an overview of the data types available to a user creating SystemC simulation models. The SystemC Language Reference Manual (LRM) IEEE Standard 1666 uses over 190 pages to specify the SystemC data types. We have attempted to be considerably briefer.

The reader will consequently need to refer to the SystemC LRM for a full definition of the SystemC data types and other resources for the C++ data types and other libraries. The SystemC library provides integer, logic, and fixed-point data types designed for modeling hardware. In addition to the SystemC data types, SystemC simulations may use native C++ data types, other library data types, and user-defined data types.

The use of SystemC data types is not restricted to models using the simulation kernel; they may be used in non-simulation applications as other data types would be used. Though simulation models may be created using any of the available data types, the choice of data types affects simulation speed, synthesizability, and synthesis results. The use of the native C++ data types can maximize simulation performance, at the cost of hardware fidelity and synthesizability.

3.1 Native C++ Data Types

The native C++ data types available on most systems include the logic data type **`bool`**, and signed and unsigned versions of the following arithmetic data types in Table 3.1.

Table 3.1 Example of C++ built-in data type on a 32-bit architecture

Name	Description	Size
`char`	Character	1 byte
`short int (short)`	Short integer	2 bytes
`int`	Integer	4 bytes
`long int (long)`	Long integer	4 bytes
`long long int`	Long long integer	8 bytes
`float`	Floating point	4 bytes
`double`	Double precision floating point	8 bytes

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_3, © Springer Science+Business Media, LLC 2010

```
// Example native C++ data types
const bool      WARNING_LIGHT(true);// Status
int             spark_offset; // Adjust ignition
unsigned        repairs(0);   // # of repairs
unsigned long   mileage;      // Miles driven
short int       speedometer;  // -20..0..100 MPH
float           temperature;  // Engine temp in C
double          time_of_last_request; // bus activity
string          license_plate;// license plate text
enum            Direction { N,NE,E,SE,S,SW,W,NW };
Direction       compass;
```

Fig. 3.1 Example of C++ built-in data types

The Standard Template Library (STL) has a rich set of additional data types. The beginner will want to become familiar with **`string`** from the STL. The string data type provides operators for appending (`+=`, `+`, and **`assign()`**) and comparison (`==`, `!=`, `<`, `<=`, `>`, `>=`, and **`compare()`**) as well as many others including a conversion to a c-string (**`c_str()`**).

For many SystemC models, the native C++ data type in Fig. 3.1 are more than sufficient. The native C++ data types are most efficient in terms of memory usage and simulator execution speed because they can be mapped directly to processor instructions.

3.2 SystemC Data Types Overview

The SystemC library provides data types that are designed for modeling digital logic and fixed-point arithmetic. There are two SystemC logic vector types: 2-valued and 4-valued logic; and two SystemC numeric types: integers and fixed-point. The SystemC data types are designed to be freely mixed with the native C++ data types, other SystemC data types, and C++ strings. SystemC data types may be used in C++ applications just as any other C++ library.

With the exception of the single-bit **`sc_logic`** type, all of the SystemC data types are length configurable over a range much broader than the native C++ data types. SystemC provides assignment and initialization operations with type conversions, allowing C++ data types, SystemC data types, and C++ strings to be used for initialization or in assignment operation to SystemC data types. All SystemC data types implement equality and bitwise operations.

The SystemC arithmetic data types (integer and fixed-point) implement arithmetic and relational operations. The SystemC implementations of these operations are semantically compatible with C++. SystemC also supports the conversion between SystemC data types and C++ data types.

The SystemC data types allow single-bit and multi-bit select operations. The results of these select operations may be used as the source (right-hand side) or

target (left-hand side) of assignment operations or concatenated with other bit-selects or with SystemC integers or vector data types.

All of the SystemC data type are part of the **sc_dt** namespace, which may be used with the scope operator (::) to avoid naming collisions with other C++ libraries.

3.3 SystemC Logic Vector Data Types

SystemC provides two logic vector types: **sc_bv**<*W*> (bit vector) and **sc_lv**<*W*> (logic vector), and a single-bit logic type **sc_logic**. Early versions of SystemC defined **sc_bit,** a single-bit version of **sc_bv**<*W*>, which was deprecated by the IEEE SystemC Standard. Older applications that used the **sc_bit** data type may replace instances of **sc_bit** with the C++ **bool** data type.

The SystemC logic vector types are intended to model at a very low level (near RTL) and do not implement arithmetic operations. They do implement a full range of assignment and logical operations, given some obvious restrictions. For example, an **sc_lv**<*W*> with high-z or unknown bit values cannot be assigned to an **sc_bv**<*W*> without losing some information.

3.3.1 sc_bv<W>

The SystemC bit vector data type **sc_bv**<*W*> has the same capabilities as the **sc_lv**<*W*> with the bit values restricted to logic zero or logic one. The **sc_bv**<*W*> is a templated class where *T* specifies bit width.

```
sc_bv<BITWIDTH> NAME…;
```

Fig. 3.2 Syntax of Boolean data types

SystemC bit vector *operations* include all the common bitwise-and, bitwise-or, and bitwise-xor operators (i.e., **&**, **|**, **^**). In addition to bit selection and bit ranges (i.e., **[]** and **range()**), **sc_bv**<*W*> also supports **and_reduce()**, **or_reduce()**, **nand_reduce()**, **nor_reduce()**, **xor_reduce()**, and **xnor_reduce()** operations. Reduction operations place the operator between all adjacent bits.

```
sc_bv<5> positions = "01101";
sc_bv<6> mask = "100111";
sc_bv<5> active = positions & mask;// 00101
sc_bv<1> all = active.and_reduce(); // SC_LOGIC_0
positions.range(3,2) = "00";// 00001
positions[2] = active[0] ^ flag;
```

Fig. 3.3 Examples of bit operations

3.3.2 sc_logic and sc_lv<W>

More interesting than the Boolean data types are the four-value data types used to represent unknown and high impedance (tri-state) conditions. SystemC uses **sc_logic** and **sc_lv**<*W*> to represent these data types (Fig. 3.4). The logic state of these data types are represented as:

- logic 0 - **SC_LOGIC_0, Log_0,** or **'0'**
- logic 1 - **SC_LOGIC_1, Log_1,** or **'1'**
- high-impedance - **SC_LOGIC_Z, Log_Z, 'Z'** or **'z'**
- unknown - **SC_LOGIC_X, Log_X, 'X'** or **'x'**

Because of their overhead, these data types are considerably slower than **bool** and **sc_bv.** The **sc_logic** data type is a single-bit version of the templated **sc_lv**<*W*> class where the single template parameter is the bit width.

SystemC does not have representations for other multi-level data types or drive strengths like Verilog's 12-level logic values or VHDL's 9-level **std_logic** values. However, you can create custom data types if truly necessary, and you can manipulate them by operator overloading in C++.

```
sc_logic         NAME[,NAME]…;
sc_lv<BITWIDTH> NAME[,NAME]…;
```

Fig. 3.4 Syntax of multi-value data types

SystemC logic vector operations (Fig. 3.5) include all the common bitwise-and, bitwise-or, and bitwise-xor operators (i.e., **&**, **|**, **^**). In addition to bit selection and bit ranges (i.e., **[]** and **range()**), **sc_lv**<*W*> also supports **and_reduce()**, **or_reduce()**, **nand_reduce()**,**nor_reduce()**, **xor_reduce()**, and **xnor_reduce()** operations. Reduction operations place the operator between all adjacent bits.

```
sc_lv<5> positions =  "01xz1";
sc_lv<6> mask      = "10ZX11";
sc_lv<5> active = positions & mask; // 0xxx1
sc_lv<1> all = active.and_reduce(); // SC_LOGIC_0
positions.range(3,2) = "00";        // 000Z1
positions[2] = active[0] ^ flag;    // !flag
```

Fig. 3.5 Examples of bit operations

3.4 SystemC Integer Types

SystemC provides signed and unsigned two's complement versions of two basic integer data types. The two data types are a limited precision integer type, which has a maximum length of 64-bits, and a finite precision integer type, which can be much longer. These integer data types provide functionality not available in the native C++ integer types. The native C++ data types have widths that are host processor and compiler dependent. The native data types are optimized for the host processor instruction set and are very efficient. These data types are typically 8, 16, 32, or 64 bits in length. The SystemC integer data types are templated and may have data widths from 1 to hundreds of bits. In addition to configurable widths, the SystemC integer data types allow bit selections, bit range selections, and concatenation operations.

3.4.1 sc_int<W> and sc_uint<W>

Most hardware needs to specify actual storage width at some level of refinement. When dealing with arithmetic, the built-in **sc_int**<*W*> and **sc_uint**<*W*> (unsigned) numeric data types (Fig. 3.6) provide an efficient way to model data with specific widths from 1- to 64-bits wide. When modeling numbers where data width is not an integral multiple of the simulating processor's data paths, some bit masking and shifting must be performed to fit internal computation results into the declared data format.

```
sc_int<LENGTH>   NAME…;
sc_uint<LENGTH>  NAME…;
```

Fig. 3.6 Syntax of arithmetic data types

3.4.2 sc_bigint<W> and sc_biguint<W>

Some hardware may be larger than the numbers supported by native C++ data types. SystemC provides **sc_bigint**<*W*> and **sc_biguint**<*W*> data types (Fig. 3.7) for this purpose. These data types provide large number support at the cost of speed.

```
sc_bigint<BITWIDTH>  NAME…;
sc_biguint<BITWIDTH> NAME…;
```

Fig. 3.7 Syntax of sc_bigint<*W*> and sc_biguint<*W*>

```
// SystemC integer data types
sc_int<5>      seat_position=3; //5 bits: 4 plus
                                // sign
sc_uint<13>    days_SLOC(4000); //13 bits: no sign
sc_biguint<80> revs_SLOC;       // 80 bits: no sign
```

Fig. 3.8 Example of SystemC integer data types

3.5 SystemC Fixed-Point Types

The SystemC fixed-point data types address the need for non-integer data types when modeling DSP applications that cannot justify the use of floating-point hardware. Early DSP models should be developed using the native C++ floating-point data types due to the much higher simulation speed. As a design evolves, fixed-point data types can provide higher fidelity modeling of the signal processing logic and can be used to provide a path to a synthesizable design.

SystemC provides multiple fixed-point data types: signed and unsigned, compile-time (templated) and run-time configurable, and fixed-precision and limited-precision (`_fast`) versions.

The SystemC fixed-point data types (Fig. 3.9) are characterized by word length (total number of bits) and their integer portion length. Optional parameters provide control of overflow and quantization modes.

Unlike other SystemC data types, fixed-point data types may be configured at compile time, using template parameters or at run time using constructor parameters. Regardless of how a fixed-point data object is created, it cannot be altered later in the execution of the program.

```
sc_fixed<WL,IWL[,QUANT[,OVFLW[,NBITS]>       NAME;
sc_ufixed<WL,IWL[,QUANT[,OVFLW[,NBITS]>      NAME;
sc_fixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]>  NAME;
sc_ufixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]> NAME;

sc_fix NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
sc_ufix NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
sc_fix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
sc_ufix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]);
```

Fig. 3.9 Syntax of fixed-point data types

```
// to enable fixed-point data types
#define SC_INCLUDE_FX
#include <systemc>
// fixed-point data types are now enabled
sc_fixed<5,3>    compass // 5-bit fixed-point word
```

Fig. 3.10 Example of fixed-point data types

Due to their compile-time overhead, fixed-point data types are omitted from the default SystemC include file. To enable fixed-point data types, **SC_INCLUDE_FX** must be defined prior to including the SystemC header file (Fig. 3.10).

The fixed-point data types have several easy-to-remember distinctions. First, those data types ending with **fast** are faster than the others, because their precision is limited to 53 bits internally; **fast** types are implemented using C++ `double`[1].

Second, the prefix **sc_ufix** indicates unsigned just as uint indicates unsigned integers. Third, the past tense **ed** suffix to **fix** indicates a templated data type that must have static parameters defined using compile-time constants.

Remember that **fixed** is past tense (i.e., already set in stone), and it cannot be changed after compilation. At run time, you may create and use new objects of the non-templated (**fix** versions) data types varying the configuration as needed.

Though the **sc_fix**, **sc_ufix**, **sc_fix_fast**, and **sc_ufix_fast** are run-time configurable, once an object of these types is created, its configuration cannot be modified.

The parameters needed for fixed-point data types are the word length (*WL*), integer-word length (*IWL*), quantization mode (*QUANT*), overflow mode (*OVFLW*), and number of saturation bits (*NBITS*). Word length (*WL*) and integer word length (*IWL*) have no defaults and must be set.

The word length establishes the total number of bits representing the data type. The integer word length indicates where to place the binary point and can be positive or negative. Figure 3.11 below shows how this works.

The preceding figure shows examples with the binary point in several positions. Consider example b in Fig. 3.11. This could be declared as **sc_fixed**<5,3>, and would represent values from –4.00 up to 3.75 in 1/4 increments.

[1] This implementation takes advantage of the linearly scaled 53-bit integer mantissa inside a 64-bit IEEE-754 compatible floating-point unit. On processors without an FPU, this behavior must be emulated in software, and there will be no speed advantage.

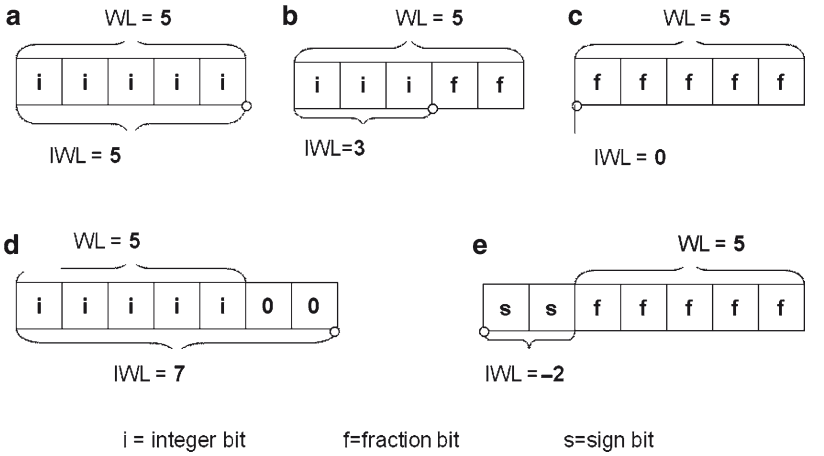


Fig. 3.11 Fixed-point formats

You can select several overflow modes from a set of enumerations that are listed in Table 3.2. A similar table for the quantization modes is also shown in Table 3.3. Overflow mode, quantization mode, and number of saturation bits all have defaults. You can modify the defaults by setting up a **`sc_fxtype_context`** object for the run-time configurable data types.

Table 3.2 Overflow mode enumerated constants

Name	Overflow Meaning
`SC_SAT`	Saturate
`SC_SAT_ZERO`	Saturate to zero
`SC_SAT_SYM`	Saturate symmetrically
`SC_WRAP`	Wraparound
`SC_WRAP_SYM`	Wraparound symmetrically

Table 3.3 Quantization mode enumerated constants

Name	Quantization Mode
`SC_RND`	Round
`SC_RND_ZERO`	Round towards zero
`SC_RND_MIN_INF`	Round towards minus infinity
`SC_RND_INF`	Round towards infinity
`SC_RND_CONV`	Convergent rounding[a]
`SC_TRN`	Truncate
`SC_TRN_ZERO`	Truncate towards zero

[a]Convergent rounding is probably the oddest. If the most significant deleted bit is one, and either the least significant of the remaining bits or at least one of the other deleted bits is one, then add one to the remaining bits.

The following examples in Fig. 3.12 should help explain the syntax for the fixed-point data types:

```
const sc_ufixed<19,3> PI("3.141592654");
sc_fix oil_temp(20,17,SC_RND_INF,SC_SAT);
sc_fixed_fast<7,1> valve_opening;
```

Fig. 3.12 Examples of fixed-point data types

Only the word length and integer word length are required parameters. If not specified, the default overflow is **SC_WRAP**, the default quantization is **SC_TRN**, and saturation bits default to one.

A special note applies if you intend to set up arrays of the **_fix** types. Since a constructor is required, but C++ syntax does not allow arguments for this situation, it is necessary to use the **sc_fxtype_context** type to establish the defaults.

3.6 SystemC Literal and String

Representation of literal data is fundamental to all languages. C++ allows for integers, floats, Booleans, characters, and strings. SystemC provides the same capability for its data types and uses C++ representations as a basis.

3.6.1 SystemC String Literals Representations

The SystemC string literals may be used to assign values to any of the SystemC data types. SystemC string literals consist of a prefix, a magnitude and an optional sign character "+" or "-". The optional sign may precede the prefix for decimal and sign and magnitude forms, but is not allowed with unsigned, binary, octal, and hexadecimal. Negative values for binary, octal, and hexadecimal may be expressed using a two's complement representation. The supported prefixes are listed in Table 3.4.

Instances of the SystemC data types may be converted to a standard C++ string using the data type's **to_string** method. The format of the resulting string is specified using **sc_numrep** enumeration shown in the left-hand column of Table 3.4. Examples of this formatting are shown in the right-most column of the above table; the values used are 13 and -13. The **to_string** method has two arguments: a numbers representation from column one of the table; and **bool** to add a representation prefix to the resulting string.

Table 3.4 Unified string representation for SystemC

sc_numrep	Prefix	Meaning	sc_int<5> = 13 sc_int<5> = -13
SC_DEC	0d	Decimal	0d13 -0d13
SC_BIN	0b	Binary	0b01101 0b10011
SC_BIN_US	0bus	Binary unsigned	0bus1101 negative
SC_BIN_SM	0bsm	Binary signed magnitude	0bsm01101 -0bsm01101
SC_OCT	0o	Octal	0o15 0o63
SC_OCT_US	0ous	Octal unsigned	0ous15 negative
SC_OCT_SM	0osm	Octal signed magnitude	0osm15 -0osm15
SC_HEX	0x	Hex	0x0d 0xf3
SC_HEX_US	0xus	Hex unsigned	0xusd negative
SC_HEX_SM	0xsm	Hex signed magnitude	0xsm0d -0xsm0d
SC_CSD	0csd	Canonical signed digit	0csd10-01 0csd-010-

```
string to_string(sc_numrep rep, bool wprefix);
```

Fig. 3.13 Syntax of **to_string**

3.6.2 String Input and Output

SystemC string literal representations and streaming IO represent data using the same formats. SystemC supports the input stream using the input extractor (**operator**>>) and output stream using the output inserter (**operator**<<).

Input streams use the literal prefixes shown in the second column of Table 3.4 to define the format of data being read from an input stream. This is the same format used when assigning a literal to a SystemC variable.

Output streams may use the C++ output stream manipulators **dec, oct**, and **hex** to control the display format of the SystemC data types. Additional display control may be obtained by using the data type's **to_string** methods.

Earlier versions of SystemC LRM defined a unique string type **sc_string**, which is now deprecated. New SystemC applications should use standard C++ string and the **to_string** method described in this section.

Below are a few examples in Fig. 3.14 that demonstrate the use of SystemC literals, the **to_string** method, and output stream:

```
//-------------------------------------------------
// sc_lv<8>      8-bit logic vectors
//-------------------------------------------------
sc_lv<8> LV1;
LV1 = 15;
cout << " LV1= " << LV1;
sc_lv<8> LV2("0101xzxz"); // literal string init
cout << " LV2= " << LV2;
cout << endl;
//-------------------------------------------------
// sc_int<5>   5-bit signed integer
//-------------------------------------------------
sc_int<5> Int1;         // 5-bit signed integer
Int1 = "-0d13";         // assign -13
cout << " Int1=" << Int1;
cout << " SC_BIN=" << Int1.to_string(SC_BIN);
cout << " SC_BIN_SM=" << Int1.to_string(SC_BIN_SM);
cout << "          " << endl;
cout << " SC_HEX=" << Int1.to_string(SC_HEX);
cout << endl;
//-------------------------------------------------
// sc_fixed<5,3> fixed 3-bit int & 2 bit fraction
//-------------------------------------------------
sc_fixed<5,3> fix1; // fixed point
fix1 = -3.3;
cout << " fix1=" <<fix1;
cout << " SC_BIN=" << fix1.to_string(SC_BIN);
cout << " SC_HEX=" << fix1.to_string(SC_HEX);
cout << endl;
```

Output:

```
LV1=00001111 LV2=0101XZXZ
Int1=-13 SC_BIN=0b10011 SC_BIN_SM=-0bsm01101
         SC_HEX=0xf3
fix1=-3.5 SC_BIN=0b100.10 SC_HEX=0xc.8
```

Fig. 3.14 Example of literals and **to_string**

3.7 Operators for SystemC Data Types

The SystemC data types support all the common operations with operator overloading as illustrated in Table 3.5.

In addition, SystemC provides special methods to access bits, bit ranges, and perform explicit conversions as illustrated in Table 3.6.

The bit and part select and concatenation operations support much like the hardware descriptions languages. This support allows the simple isolation of fields in

Table 3.5 Operators

Comparison	`== != > >= < <=`
Arithmetic	`++ -- * / % + - << >>`
Bitwise	`~ & | ^`
Assignment	`= &= |= ^= *= /= %= += -= <<= >>=`

Table 3.6 Special methods

Bit Selection	`bit(idx), [idx]`
Range Selection	`range(high,low), (high,low)`
Conversion (to C++ types)	`to_double(),to_int(), to_int64(),to_long(), to_uint(),to_uint64(), to_ulong(),to_string(type)`
Testing	`is_zero(), is_neg(), length()`
Bit Reduction	`and_reduce(),nand_reduce(), or_reduce(),nor_reduce(), xor_reduce(), xnor_reduce()`

packed data or the concatenation of multiple bit or bit-fields to create packed data objects. Several examples are shown in Fig. 3.15.

```
//-------------------------------------------------
// bit-select and part-select examples
//-------------------------------------------------
sc_uint<8>  I1 = "0x35";     // 8-bit signed integer
sc_uint<5>  I2 = "0b01010";  // 5-bit signed integer
sc_uint<4>  I3 = 0;          // 5-bit signed integer

sc_uint<16> I4 = 0;            // 16-bit signed
integer

I3    = I2.range(3,0);       // I3= 0b1010
I3[2] = true;                // I3= 0b1110
I3[0] = true;                // I3= 0b1111

I4 = (I3,I1.range(7,4),I2(3,0),I1(3,0));
// I4 = 0x0f3a5                    HEX format
// I4 = 0b01111001110100101        BIN format
```

Fig. 3.15 Bit-select, part-select, and concatenation

One often overlooked aspect of these data types (and C++ data types) is mixing types in arithmetic operations. It is OK to mix similar data types of different lengths, but crossing types is dangerous. For example, assigning the results of an operation involving two **`sc_int`**`<W>` variables to an **`sc_bigint`**`<W>` does not automatically promote the operand to **`sc_bigint`** for intermediate calculations. To accomplish that, it is necessary to have one of the arguments be an **`sc_bigint`**`<W>` or perform an explicit conversion of one of at least one of the operand arguments. Below is an example of addition.

```
sc_int<3> d(3);
sc_int<5> e(15);
sc_int<5> f(14);
sc_int<7> sum = d + e + f;// Works
sc_int<64> g("0x7000000000000000");
sc_int<64> h("0x7000000000000000");
sc_int<64> i("0x7000000000000000");
sc_bigint<70> bigsum = g + h + i; // Doesn't work
bigsum = sc_bigint<70>(g) + h + i;// Works
```

Fig. 3.16 Example of conversion issues

```
#include <vector>
int main(int argc, char* argv[]) {
  vector<int> memory(1024);
  for (unsigned i=0; i!= 1024; i++) {
    // Following checks access (safer than
    // memory[i])
    memory.at(i) = -1; // initialize to known values
  }//endfor
  …
  memory.resize(2048); // increase size of memory
  …
}//end main()
```

Fig. 3.17 Example of STL vector

3.8 Higher Levels of Abstraction and the STL

The basic C++ and SystemC data types lack structure and hierarchy. For these, the standard C++ `struct` and `array` are good starting points. However, a number of very useful data type classes are freely available in C++ libraries, which provides another benefit of having a modeling language based upon C++.

The STL is the most popular of these libraries, and it comes with all modern C++ compilers. The STL contains many useful data types and structures, including an improved character array known as string, and generic containers such as the `vector<T>`, `map<T, T>`, `set<T>`, `list<T>`, and `deque<T>`. These containers can be manipulated by STL algorithms such as `for_each()`, `count()`, `min_element()`, `max_element()`, `search()`, `transform()`, `reverse()`, and `sort()`. These are just a few of the algorithms available. This book will not attempt to cover the STL in any detail, but a brief example may stimulate you to search further.

The STL container `vector<T>` closely resembles the common C++ array, but with several useful improvements. Unlike arrays, vectors can be assigned and compared. Also, the `vector<T>` may be resized dynamically. Perhaps more impor-

tantly, accessing an element of a **vector**<*T*> can have bounds checking for safety. The example below demonstrates use of an STL **vector**<*T*>.

Your mileage will vary, but the authors are certain you will benefit from using the STL with SystemC.

3.9 Choosing the Right Data Type

A frequent question is, "Which data types should be used for this design?" The best answer is, "Choose a data type that is closest to native C++ as possible for the modeling needs at hand." Choosing native data types will always produce the fastest simulation speeds. Table 3.7 gives an idea of performance.

Table 3.7 Data type performance

Speed	Data type
Fastest	Native C/C++ Data Types (e.g., int, double and bool) sc_int<W>, sc_uint<W> sc_bv<W> sc_logic, sc_lv<W> sc_bigint<W>, sc_biguint<W> sc_fixed_fast<WL,IL,…>, sc_fix_fast, sc_ufixed_fast<WL,IL,…>, sc_ufix_fast
Slowest	sc_fixed<WL,IL,…>, sc_fix, sc_ufixed<WL,IL,…>, sc_ufix

Do not use **sc_int**<*W*> or **sc_bigint**<*W*> unless you require more detail than available by native C++ data types. It is preferred to use **sc_int**<*W*> for 64 or fewer bits over **sc_bigint**<*W*> for higher performance. In general, SystemC data types are slower than native C++ data types, and more complex SystemC types are slower than simpler smaller types.

RTL synthesis tools generally require all data to be SystemC data types. Some behavioral synthesis tools allow native C++ data types. SystemC data types may be used to guide the behavioral synthesis tool.

3.10 Exercises

For the following exercises, use the samples provided at www.scftgu.com

Exercise 3.1: Examine, compile, and run the examples from the web site, `datatypes` and `uni_string_rep`. Note that although these examples include `systemc`, they only use data types.

Exercise 3.2: Write a program to read data from a file using the unified string representation and store in an array of **sc_uint**<*W*>. Output the values as **SC_DEC** and **SC_HEX_SM**.

Exercise 3.3: Write a program to generate 100,000 random values and compute the squares of these values. Do the math using each of the following data types: **short, int, unsigned, long, sc_int<8>, sc_uint<19>, sc_bigint<8>, sc_bigint<100>, sc_fixed**<12,12>. Be certain to generate numbers distributed over the entire range of possibilities. Compare the run times of each data type.

Exercise 3.4: Write a program to explore data accuracy of fixed-point numbers for an application needing to add two sine waves with amplitudes of 14 and 22, where the larger of the two is also 2½ times the frequency of the smaller. There are 512 samples over the longest period. Assume you are limited to 12 bits for both input and output. Try different modes. HINT: Start modeling using **double**. Progress to using **sc_fix**.

Exercise 3.5: Examine, compile, and run the example `addition`. What would it take to fix the problems noted in the source code? Try adding various sizes of **sc_bigint**<*W*>.