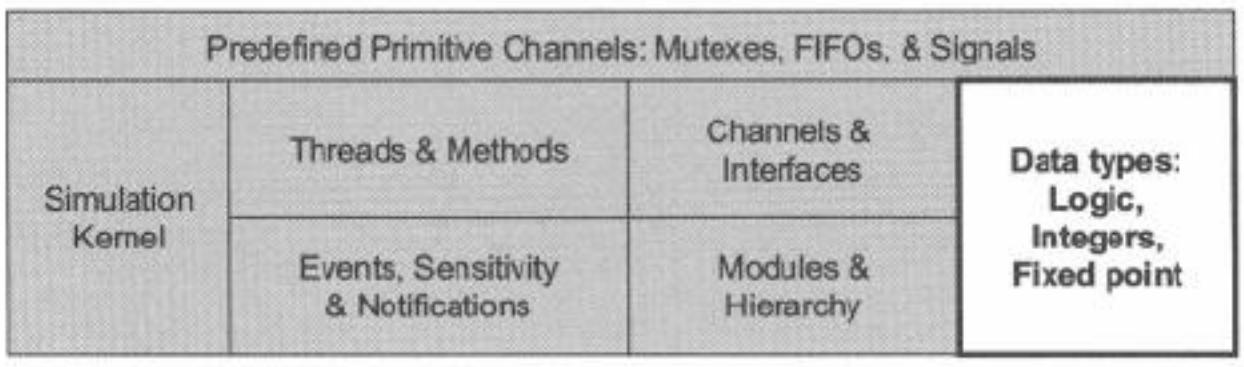

Chapter 4

DATA TYPES

SystemC has a number of predefined data types to support hardware designs spanning from the native C++ data types to specialized fractional fixed-point representations. Choosing a data type depends on the range of values to be represented, the required precision, and the required operations. Choice of a data type also affects the speed of simulation, synthesizability, and synthesis results. The data types used differ depending on the level of abstraction represented in your model.

4.1 Numeric Representation

Representation of literal data is fundamental to all languages. C++ allows for simple integers, floats, Booleans, characters, and strings.

To support hardware data representations, SystemC provides a unified string representation using C-style strings. It is possible to convert both to and from this format. SystemC uses the following syntax for strings:


```
sc_string name("0 base [sign ] number [e[+|-] exp]");
// no whitespace
```

Zero

Figure 4-1. Syntax of sc_string

Where *base* is one of **b**, **o**, **d**, or **x** for binary, octal, decimal, and hexadecimal, respectively. The *sign* allows specification of signed (empty), unsigned (**us**), signed magnitude (**sm**), and canonical signed digit (**csd**) numbers. *number* is an integer in the indicated base. The optional exponent *exp* is always specified using decimal. Eleven specific representations from the SystemC LRM are shown in the table below. Notice the enumeration column, **sc_numrep**, which is used when converting into a unified string.

Table 4-1. Unified String Representation for SystemC

sc_numrep	Prefix	Meaning	sc_int<5>(-13)[10]
SC_DEC	0d	Decimal	"-0d13"
SC_BIN	0b	Binary	"0b10011"
SC_BIN_US	0bus	Binary unsigned	"0bus01101"
SC_BIN_SM	0bsm	Binary signed magnitude	"-0bsm01101"
SC_OCT	0o	Octal	"0o03"
SC_OCT_US	0ous	Octal unsigned	"0ous15"
SC_OCT_SM	0osm	Octal signed magnitude	"-0osm03"
SC_HEX	0x	Hex	"0xf3"
SC_HEX_US	0xus	Hex unsigned	"0xus0d"
SC_HEX_SM	0xsm	Hex signed magnitude	"-0xsm0d"
SC_CSD	0csd	Canonical signed digit	"0csd-010-"

Here are some examples of literal data in SystemC:

```
sc_string a ("0d13"); // decimal 13
a = sc_string ("0b101110") ; // binary of decimal 44
```

Figure 4-2. Example of sc_string

[10] +13 for unsigned types

4.2 Native Data Types

SystemC supports all the native C++ data types: **`int`**, **`long int`**, **`int`**, **`unsigned int`**, **`unsigned long int`**, **`unsigned short int`**, **`short`**, **`double`**, **`float`**, **`char`**, and **`bool`**

For many SystemC designs, the built-in C++ data types should be sufficient. Native C++ data types are the most efficient in terms of memory usage and execution speed of the simulator.

```
// Example native C++ data types
int             spark_offset;  // Adjustment for
                               // ignition
unsigned        repairs =0 ;   // Count repair
                               // incidents
unsigned long   mileage;       // Miles driven
short int       speedometer;   // -20..0..100 MPH
float           temperature;   // Engine temp in C
double          time_of_last_request;  //Time of bus
                                        //activity
std::string     license_plate;// Text for license
                              // plate
const bool      WARNING_LIGHT =true;//  Status
                                    // indicator
// Direction of travel
enum            compass   {SW,W,NW,N,NE,E, SE, S} ;
```

Figure 4-3. Example of C++ Built-in Data Types

4.3 Arithmetic Data Types

SystemC provides two sets of numeric data types. Arithmetic operations may be performed on numeric data. One set models data with bit widths up to 64-bits wide; another set models data with bit widths larger than 64-bits wide. Most of the native C++ data types have widths, ranges, and interpretations that are compiler-implementation-defined to match the host computer for execution efficiency.

4.3.1 sc_int and sc_uint

Most hardware needs to specify actual storage width at some level of refinement. When dealing with arithmetic, the built-in **`sc_int`** and **`sc_uint`** (unsigned) numeric data types provide an efficient way to model data with specific widths from 1- to 64-bits wide. When modeling numbers with data whose 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.

Thus, any data type that is not native to both the C++ language and the processor width will simulate slower than the native types. Thus, built-in C++ data types are faster than **`sc_int`** and **`sc_uint.`**

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

Figure 4-4. Syntax of Arithmetic Data Types

Significant speed improvements can be attained if all **`sc_ints`** are 32 or fewer bits by simply setting the **`-D_32BIT_`** compiler flag.

GUIDELINE Do not use **`sc_int`** unless or until prudent. One necessary condition for using **`sc_int`** is when using synthesis tools that require hardware representation.

4.3.2 sc_bigint and sc_biguint

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

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

Figure 4-5. Syntax of sc_bigint and sc_biguint

```
// 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
```

Figure 4-6. Example of SystemC Integer Data Types

GUIDELINE: Do not use **sc_bigint** for 64 or fewer bits. Doing so causes performance to suffer compared to using **sc_int**.

4.4 Boolean and Multi-Value Data Types

SystemC provides one set of data types for Boolean values and another set of data types for unknown and tri-state values.

4.4.1 sc_bit and sc_bv

For ones and zeroes, SystemC provides the **sc_bit**, and for long bit vectors SystemC provides **sc_bv<>** (bit vector) data types. These types do not support arithmetic data like the **sc_int** types, and these data types don't execute as fast as the built-in **bool** and Standard Template Library **bitset** types. As a result, these data types are being considered for deprecation (i.e., removal from the language).

```
sc_bit              NAME...;
sc_bv<BITWIDTH>     NAME...;
```

Figure 4-7. Syntax of Boolean Data Types

sc_bit and **sc_bv** come with some supporting data constants, **SC_LOGIC_1** and **SC_LOGIC_0**. For less typing, if **using namespace sc_dt**, then type **Log_1** and **Log_0**, or even type '1' and '0'.

Operations include the common bitwise **and**, **or**, **xor** operators (i.e., **&**, **|**, **^**). In addition to bit selection and bit ranges (i.e., **[]** and **range()**), **sc_bv<>** 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_bit  flag(SC_LOGIC_1);  // more efficient to use bool
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;
```

Figure 4-8. Examples of bit operations

4.4.2 sc_logic and sc_lv

More interesting than the Boolean data types are the multi-value data types used to represent unknown and high impedance (i.e., tri-state) conditions. SystemC represents these with the **sc_logic** and **sc_lv<>** (logic vector) data types. These types are represented with **SC_LOGIC_1**, **SC_LOGIC_0**, **SC_LOGIC_X**, and **SC_LOGIC_Z**. For less typing, if **using namespace sc_dt**, then type **Log_1**, **Log_0**, **Log_X**, and **Log_Z**, or even type '1', '0', 'X' and 'Z'.

Because of the overhead, these data types are considerably slower than their **sc_bit** and **sc_bv** counterparts. For best performance, always use built-in types such as **bool**.

```
sc_logic            NAME[,NAME]...;
sc_lv<BITNIDTH>  NAME[,NAME]...;
```


Figure 4-9. Syntax of Multi-Value Data Types

SystemC does not represent 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 buf (sc_dt::Log_Z) ;
sc_lv<8> data_drive ("zz01XZ1Z") ;
data_drive.range  (5,4)  = "ZZ";//  ZZZZXZ1Z
buf = '1' ;
```

Figure 4-10. Examples of 4-Level Logic Types

4.5 Fixed-Point Data Types

SystemC provides the following fixed-point data types: **`sc_fixed`**, **`sc_ufixed`**, **`sc_fix`**, **`sc_ufix`**, and the **`_fast`** variants of them.

Integral data types do not satisfy all design types. In particular, DSP applications often need to represent numbers with fractional components. SystemC provides eight data types providing fixed-point numeric representation. While native **`float`** and **`double`** data types satisfy high-level representations, realizable hardware has speed and area requirements. Also, integer-based DSP processors do not natively support floating point. Fixed-point numbers provide an efficient solution[11] in both hardware and software.

A number of parameters that control fixed-point behavior (e.g., overflow and underflow) must be set. What follows will briefly cover these aspects, but for full information, please see the SystemC LRM.

IMPORTANT: To improve compile times, the SystemC header code omits fixed-point data types unless the **`#define SC_INCLUDE_FX`** is specified prior to **`#include`** <systemc.h> in your code.

```
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_fast      NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
sc_ufix_fast     NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
sc_fixed_fast    NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
sc_ufixed_fast   NAME(WL,IWL[,QUANT[,OVFLW[,NBITS])...;
```

Figure 4-11. Syntax of Fixed-Point Data Types

These data types have several easy-to-remember distinctions. First, those ending with **`_fast`** are faster than the others are because their precision is

[11] At the time of writing, fixed-point data types were not synthesizable; however, private discussions indicate some EDA vendors are considering this as a possible new feature.

limited to 53 bits. Internally, **_fast** types are implemented using C++ **double**[12].

Second, the prefix **sc_ufix** indicates unsigned just as **uint** are 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. On the other hand, you can dynamically change those data types lacking the past tense (i.e., the **_fix** versions). Non-**ed** types are still active, and you can change them on the fly.

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 need to 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 decimal point and can be positive or negative. *Figure 4-12* below illustrates how this works.

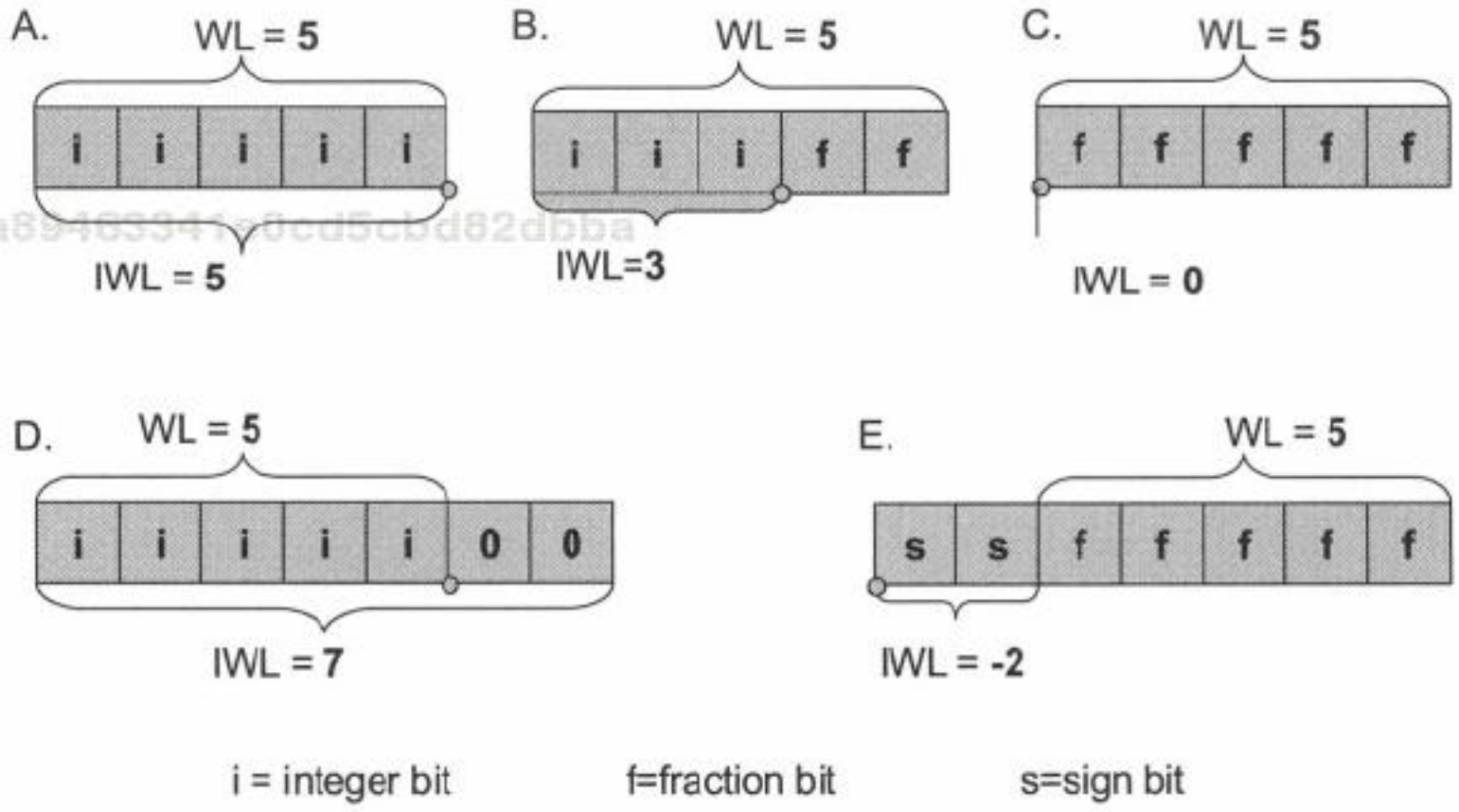

Figure 4-12. **Fixed-Point Formats**

[12] 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.

The *Figure 4-12*shows examples with the binary point in several positions. Consider example B in the preceding figure. This could be declared as **sc_fixed**<5,3>, and would represent values from –4.75 up to 3.75 in 1/4 increments.

You can select several overflow modes from a set of enumerations that are listed in the next table. A similar table for the quantization modes is also shown. Overflow mode, quantization mode, and number of saturation bits all have defaults. You can establish defaults by setting up **sc_fxtype_context** objects for the non-**ed** data types.

Table 4-2. 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 4-3. 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[13]
SC_TRN	Truncate
SC_TRN_ZERO	Truncate towards zero

The following examples 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;
```

Figure 4-13. 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 defaults 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.

For significantly more information, refer to section 6.8 of the SystemC LRM.

[13] 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.

4.6 Operators for SystemC Data Types

The SystemC data types support all the common operations with operator overloading.

Table 4-4. Operators

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

In addition, SystemC provides special methods to access bits, bit ranges, and perform explicit conversions.

Table 4-5. 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()`

In general, all the common combinations you would expect are present. For more information, refer to the SystemC LRM.

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_ints`** to an **`sc_bigint`** 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** or perform an explicit conversion of one of at least one of the operand arguments. Here is an example (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
```

Figure 4-14. Example of Conversion Issues

4.7 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, which provides another benefit of having a modeling language based upon C++.

The Standard Template Library (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**. This book will not attempt to cover the STL in any detail, but a brief overview may stimulate you to search further.

The STL has generic containers such as the **vector<>**, **map<>**, **list<>**, and **deque<>**, which may contain various data types. 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.

The STL container **vector<>** closely resembles the common C++ array, but with several useful improvements. First, the **vector<>** may be resized dynamically. Second, and perhaps more importantly, accessing an element of a **vector<>** can have bounds checking for safety. The example below demonstrates use of an STL **vector<>**.

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

Figure 4-15. Example of STL Vector

Large sparsely-used memories occupy too much space when implemented as arrays or vectors. These memories require the attributes of an associative map such as the STL **map<>**. A **map<>** requires specification of both the index and the value data types. Only index values that you have assigned occupy storage space. Thus, you can represent a large data space with minimal real memory.

```
#include <iostream>
#include <map>
int main(int argc, char* argv[]) {
  typedef unsigned long ulong;
  std::map<ulong, int> lmem; //possible 2^64
                             //locations!
  // Fill ten random locations with random values
  while (lmem.size() < 10) {
    // 10 random memory location/values
    lmem[rand()]=rand();
  }//endwhile
  // Display memory contents
  typedef std::map<ulong, int>::const_iterator iter;
  for (iter iv=lmem.begin(); iv!=lmem.end(); ++iv) {
    std::cout << std::hex
      << "lmem[" << iv->first
      << "]=" << iv->second << ";" << std::endl;
  }//endfor
}//end main()
```

Figure 4-16. Example of an STL Map

4.8 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.

The table below gives an idea of performance.

Table 4-6. Data Type Performance

Fastest	Native C/C++ Data Types (e.g., **`int`**, **`double`** and **`bool`**)
	`sc_int<>`, **`sc_uint<>`**
	`sc_bit`, **`sc_bv<>`**
	`sc_logic`, **`sc_lv<>`**
	`sc_bigint<>`, **`sc_biguint<>`**
	`sc_fixed_fast<>`, **`sc_fix_fast`**, **`sc_ufixed_fast<>`**, **`sc_ufix_fast`**
Slowest	**`sc_fixed<>`**, **`sc_fix`**, **`sc_ufixed<>`**, **`sc_ufix`**

Some types of modeling tools may impose requirements on data types. For instance, RTL synthesis tools generally require all data to be in `sc_*` data types, and do not synthesize floating-point or fixed-point data types.

4.9 Exercises

For the following exercises, use the samples provided at www.EklecticAlly.com/Book/.

Exercise 4.1: Examine, compile, and run the examples from the website, `datatypes` and `uni_string_rep`. Note that although these examples include **`systemc.h`**, they only use data types.

Exercise 4.2: Write a program to read data from a file using the unified string representation and store in an array of **`sc_uint`**. Output the values as **`SC_DEC`** and **`SC_HEX_SM`**.

Exercise 4.3: 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 4.4: Examine, compile, and run the example `addition`. What would it take to fix the problems noted? Try adding various sizes of **`sc_bigint`**`<>`.