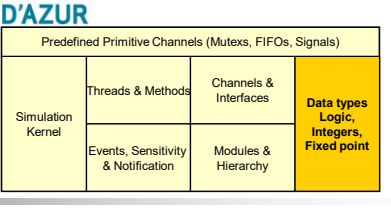
Plan

- Ch1 – Overview of SystemC
- Ch2 – Data Types
- Ch3 – Modules
- Ch4 – Notion of Time
- Ch5 – Concurrency
- Ch6 – Predefined Channels
- Ch7 – Structure
- Ch8 – Communication
- Ch9 – Custom Channels and Data
- Ch10 – Transaction Level Modeling



Data Types

- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- Higher level of abstraction with STL
- Conclusion

Numeric Representation

- Representation of literal value is fundamental
- C++ allows
  - Simple integers
  - Float
  - Booleans
  - Characters
  - Strings

class

zero

sc_string name("**0** base [sign] number [**e**[+|-] exp]");

No whitespace !

b (binary), o (octal),
d (decimal), x (hexadecimal)

empty, us (unsigned),
sm (signed magnitude)

Examples

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


Numeric Representation

```
enum sc_numrep
{
   SC_NOBASE = 0,
   SC_BIN    = 2,
   SC_OCT    = 8,
   SC_DEC    = 10,
   SC_HEX    = 16,
   SC_BIN_US,
   SC_BIN_SM,
   SC_OCT_US,
   SC_OCT_SM,
   SC_HEX_US,
   SC_HEX_SM,
   SC_CSD
};
```

sc_numrep	Prefix	Meaning	sc_int<5>(-13)*
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	"0o63"
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"

Numeric Representation Example

```
sc_int<8> rx_data = 106;
sc_int<4> tx_buf = -5;

cout << "Default: rx_data=" << rx_data.to_string() << endl;
cout << "Binary: rx_data=" << rx_data.to_string(SC_BIN) << endl;
cout << "Binary unsigned: rx_data=" << rx_data.to_string (SC_BIN_US) << endl;
cout << "Binary sign magnitude: rx_data=" << rx_data.to_string(SC_BIN_SM) << endl;
cout << "Octal: tx_buf=" << tx_buf.to_string (SC_OCT) << endl;
cout << "Hexadecimal: tx_buf=" << tx_buf.to_string (SC_HEX) << endl;
cout << "Decimal: tx_buf=" << tx_buf.to_string (SC_DEC) << endl;
cout << "-----------------------------"
cout << "Binary without base: rx_data=" << rx_data.to_string(SC_BIN, false) << endl;
cout << "Hexadecimal without base: tx_buf=" << tx_buf.to_string (SC_HEX, false) << endl;
cout << "Decimal without base: tx_buf=" << tx_buf.to_string (SC_DEC, false) << endl;
```

numeric_representation

Output produced

```
Default: rx_data=106
Binary: rx_data=0b01101010
Binary unsigned: rx_data=0bus1101010
Binary sign magnitude: rx_data=0bsm01101010
Octal: tx_buf=0o73
Hexadecimal: tx_buf=0xb
Decimal: tx_buf=-5
-----------------------------
Binary without base: rx_data=01101010
Hexadecimal without base: tx_buf=b
Decimal without base: tx_buf=-5
```


Data Types

- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- Fixed-Point Data Types
- User Defined Data Types
- Higher level of abstraction with STL
- Conclusion

Native Data Types

- SystemC supports all the native C++ data types
- Most efficient in terms of memory usage
- Most efficient execution speed of the simulator

```
// Example
int              spark_offset;
unsigned         repairs = 0;
unsigned long    mileage;
short int        speedometer;
float            temperature;
double           time_of_last_request;
std::string      license_plate;
const bool       WARNING_LIGHT = true;
enum             compass { SW, W, NW, N, NE, E, SE, S };
```

Not equal to sc_string ! (SystemC v2.01)
equal to string ! (SystemC 2.1)

Arithmetic Data Types sc_int and sc_uint

- By default : 64 bits
- Slower than the native types (int)

1 to 64 bits wide

sc_int<*length*> name … ;
sc_uint<*length*> name … ;

Example

```
sc_int<4> a;   // Represents variable "a" of 4 bits width
```

VHDL: variable a : integer range -8 to 7;

GUIDELINE: One necessary condition for using sc_int is when using synthesis tools that require hardware representation.

Arithmetic Data Types
sc_bigint and sc_biguint

- More than 64 bits !
- Slower than sc_int

sc_bigint<*length*> name … ;
sc_biguint<*length*> name … ;

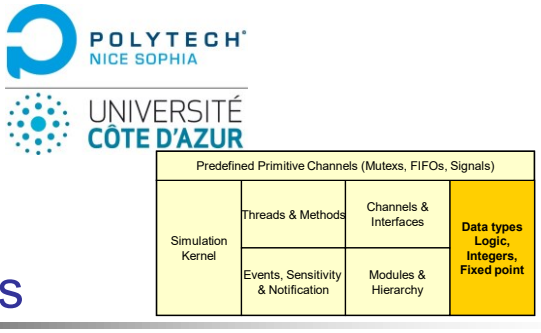
Example

```
sc_int<5>        a;  // 5 bits : 4 plus sign
sc_uint<13>      b;  // 13 bits : no sign
sc_biguint<80>   c;  // 80 bits : no sign
```

VHDL

```
variable a : integer range -16 to 15;
variable b : integer range 0 to 2^13-1;
variable c : integer range 0 to 2^80-1;
```

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



Data Types

- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- User Defined Data Types
- Higher level of abstraction with STL
- Conclusion

sc_bit and sc_bv (1/3)
Introduction

- sc_bit (bit)
  - ‘0’ or ‘1’ value / SC_LOGIC_0 or SC_LOGIC_1
  - VHDL : bit
- sc_bv
  - vector of bit
  - VHDL : bit_vector

sc_bit name … ;
sc_bv<*bitwidth*> name … ;

VHDL

variable name : bit;
variable name : bit_vector(0 to *bitwidth-1);*
variable name : bit_vector(*bitwidth-1* downto 0*);*

Example

```
sc_bit flag(SC_LOGIC_1);
sc_bv<5> positions = “01101”;
sc_bv<6> mask = “100111”;

positions.range(3,2) = “00”;
positions[2] = mask[0] ^ flag;
```

VHDL

```
variable flag : bit := ‘1’;
variable positions : bit_vector(0 to 4) := “01101”;
variable mask : bit_vector(0 to 5) := “100111”;

position(2 to 3) := “00”;
positions(2) := mask(0) xor flag;
```


sc_bit and sc_bv (2/3)
Operators

operator	function	usage	bit	bit_vector
&	bitwise AND	expr1 & expr2	√	√
|	bitwise OR	expr1 | expr2	√	√
^	bitwise XOR	expr1 ^ expr2	√	√
~	bitwise NOT	~expr	√	√
<<	bitwise shift left	expr << constant		√
>>	bitwise shift right	expr >> constant		√
=	assignment	value_holder = expr	√	√
&=	compound AND assignment	value_holder &= expr	√	√
|=	compound OR assignment	value_holder |= expr	√	√
^=	compound XOR assignment	value_holder ^= expr	√	√
==	equality	expr1 == expr2	√	√
!=	inequality	expr1 != expr2	√	√
[ ]	bit selection	variable[index]		√
(,)	concatenation	(expr1, expr2, expr3)		√

```
// Bit example

bool ready;
sc_bit flag = sc_bit(‘0’);

ready = ready & flag;

if (ready == flag)
…
```

```
// Bit vector example

sc_bv<8> ctrl_bus;
ctrl_bus[5] = ‘0’ & ctrl_bus[6];

ctrl_bus << 2;  // multiply by 4
```

sc_bit and sc_bv (3/3)
Methods for bit vectors

Methods for bit vectors

method	function	usage
range( )	range selection	var.range(index1,index2)
and_reduce()	reduction AND	var.and_reduce()
nand_reduce()	reduction NAND	var.nand_reduce()
or_reduce()	reduction OR	var.or_reduce()
nor reduce()	reduction NOR	var.nor_reduce()
xor_reduce()	reduction XOR	var.xor_reduce()
xnor_reduce()	reduction XNOR	var.xnor_reduce()

```
// Bit vector example

sc_bv<8> ctrl_bus;
sc_bv<4> mult;

ctrl_bus.range(0,3) = ctrl_bus.range(7,4);
mult = (ctrl_bus[0], ctrl_bus[0], ctrl_bus[0], ctrl_bus[1]);

ctrl_bus[0] = ctrl_bus.and_reduce();
ctrl_bus[1] = mult.or_reduce();
```

```
sc_bv<5> active = position & mask;
```

```
variable active : bit_vector(4 downto 0);
active := positions and mask;
```

VHDL

```
sc_bv<1> all = active.and_reduce();
```

```
variable all : bit_vector(0 to 0);
all := active(0) and active(1) and active(2) and active(3) and active(4);
```


sc_logic and sc_lv (1/2)
Introduction

- sc_logic
  - '0', '1', 'Z', 'X' value / SC_LOGIC_0, SC_LOGIC_1, SC_LOGIC_Z, SC_LOGIC_X
  - sc_dt : Log_1, Log_0, Log_Z, Log_X
  - VHDL : std_logic
- sc_lv
  - range(), and_reduce(), or_reduce(), nand_reduce(), nor_reduce(), xor_reduce()
  - VHDL : std_logic_vector

sc_logic name … ;
sc_lv<*bitwidth*> name … ;

variable name : std_logic;

VHDL

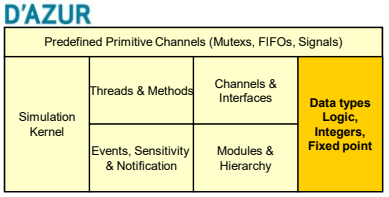
variable name : std_logic_vector(0 to *bitwidth-1*);
variable name : std_logic_vector(*bitwidth-1 downto* 0);

Example

```
using namespace sc_dt;

sc_logic buf(sc_dt::Log_Z);
sc_lv<8> data_drive("ZZ01XZ1Z");

data_drive.range (5,4) = "ZZ";  // ZZZZXZ1Z
buf = '1';
```

```
variable buf : std_logic := 'Z';
variable data_drive : std_logic_vector(7 downto 0) := "ZZ01XZ1Z";

data_drive(5 downto 4) := "ZZ";
buf := '1';  -- ZZZZXZ1Z
```

VHDL

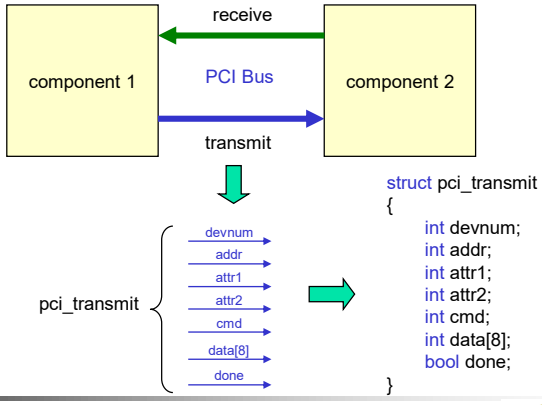


Data Types

- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- User Defined Data Types
- Higher level of abstraction with STL

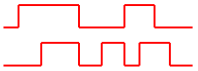
Introduction

- New Data Types
  - enumeration types
  - record types
- Used by High Level abstraction
  - for example, a bus is considered like a structure included control, data, address



User Data Type
Compulsory Operators

- In SystemC, you must define 3 operators for a new data type
  - assignment, operator =
  - equality, operator ==
  - stream output, operator <<
- one methods to trace waves
  - sc_trace()



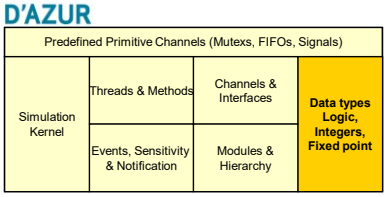
waveform

```
struct X  // or class X
{
    …
    X& operator= (const X&);
    bool operator== (const X&) const;
};

ostream& operator<< (ostream&, X);
void sc_trace (sc_trace_file *tf, const X& arg, const sc_string& name);
```


Example : Micro bus

mbus.h

```
#include "systemc.h"

const int ADDR_WIDTH = 16;
const int DATA_WIDTH = 8;

struct mbus
{
    sc_uint<ADDR_WIDTH> address;
    sc_uint<DATA_WIDTH> data;
    bool read, write;

    mbus& operator= (const mbus&);
    bool operator== (const mbus&) const;
};

inline mbus& mbus::operator= (const mbus& arg)
{
    address = arg.address;
    data = arg.data;
    read = arg.read;
    write = arg.write;
    return *this;
}

inline bool mbus::operator== (const mbus& arg) const
{
    return (
        (address == arg.address) &&
        (data == arg.data) &&
        (read == arg.read) &&
        (write == arg.write));
}

inline ostream& operator<< (ostream& os, const mbus& arg)
{
    os << "address=" << arg.address <<
        " data=" << arg.data << " read=" << arg.read <<
        " write=" << arg.write << endl;
    return os;
}

inline void sc_trace (sc_trace_file *tf, const mbus& arg, const sc_string& name)
{
    sc_trace (tf, arg.address, name+".address");
    sc_trace (tf, arg.data, name+".data");
    sc_trace (tf, arg.read, name+".read");
    sc_trace (tf, arg.write, name+".write");
}
```



Data Types

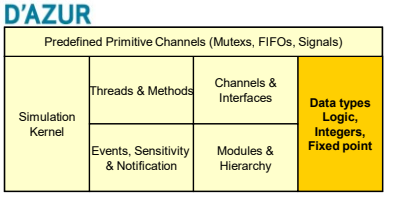
- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- User Defined Data Types
- Higher level of abstraction with STL
- Conclusion

Standard Template Library (STL)

- generic containers
  - vector<T> (a variable-sized vector)
  - map<key,val> (an associated array)
  - list<T> (a doubly-linked list)
  - deque<T> (a double-ended queue)
  - …
- manipulation methods
  - for_each()
  - count()
  - min_element()
  - max_element()
  - search()
  - transform()
  - reverse()
  - sort()

Standard Template Library (STL)
Example of vector class

```
#include <vector>

int main(int argc, char* argv[])
{
    std::vector<int> mem(1024);

    for (unsigned i=0; i != 1024; i++)
        mem.at(i) = -1;  // initialize memory to known values

    mem.resize(2048);  // increase size of memory

}
```

Data Types

- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- User Defined Data Types
- Higher level of abstraction with STL
- Conclusion

Guideline

- For one bit
  - bool var
- For vectors and unsigned arithmetic
  - sc_uint<n> var
- For signed arithmetic
  - sc_int<n> var
- If vector size is more than 64 bits
  - sc_bigint var
  - sc_biguint var
- For loop indices, etc.
  - int var
  - other C++ integer type
- Use sc_logic and sc_lv<n> types for only those signals that are carry the four logic values

Performance

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