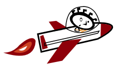
Plan

- Ch1 – Overview of System Design Using SystemC
- Ch2 – Overview of SystemC



- Ch3 – Data Types
- Ch4 – Modules
- Ch5 – Notion of Time
- Ch6 – Concurrency
- Ch7 – Predefined Channels
- Ch8 – Structure
- Ch9 – Communication
- Ch10 – Custom Channels and Data
- Ch11 – Transaction Level Modeling

Ecole polytechnique de l'université de Nice-Sophia Antipolis
Département Electronique

Data Types

<table>
<tr><td colspan="4">Predefined Primitive Channels (Mutexs, FIFOs, Signals)</td></tr>
<tr><td rowspan="2">Simulation Kernel</td><td>Threads & Methods</td><td>Channels & Interfaces</td><td rowspan="2">Data types Logic, Integers, Fixed point</td></tr>
<tr><td>Events, Sensitivity & Notification</td><td>Modules & Hierarchy</td></tr>
</table>

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

C++ Language Keyword

RECALL … RECALL … RECALL … RECALL

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

bool true false namespace

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)

<u>Examples</u>

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

C++ Enumeration

RECALL … RECALL … RECALL … RECALL

An enumeration is a type that can hold a set of values specified by the user. By default, enumerator values are assigned increasing from 0

```
enum keyword { ASM, AUTO, BREAK };
                ↓      ↓       ↓
                0      1       2
```

Using

```
void f(keyword key)
{
    switch (key) {
        case ASM :
            // do something
            break;
        case BREAK :
            // do something
            break;
    }
}
```

Others Examples

round(ln(1000) / ln(2))

```
enum e1 { dark, light };                 // range 0:1
enum e2 { a = 3, b = 9 };                // range 0:15
enum e3 {min = -10, max = 1000 };        // range -1024:1023

enum flag { x=1, y=2, z=4, e=8 };        // range 0:15

flag f1 = 5;           // type error: 5 is not of type flag
flag f2 = flag(5);     // ok: flag(5) is of type flag and
                       // within the range of flag
flag f3 = flag(z | e); // ok: flag(12)
flag f4 = flag(99);    // undefined: 99 is not within the range of flag
```


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"

C++

Output Streams (1/2)

RECALL … RECALL … RECALL … RECALL

An ostream is a mechanism for converting values of various types into sequences of characters.

```
ostream cout;    // standard output stream of char (stdout)
ostream cerr;    // standard unbuffered output stream for error messages (stderr)
ostream clog;    // standard output stream for error messages (stderr)
```

Example

cast newline

```
void val(char c)
{
    cout << "int('" << c << "') = " << int(c) << endl;
}

int main()
{
    cout << "----------------" << endl;
    val('A');
    val('B');
    cout << "----------------" << endl;
    return 0;
}
```

```
----------------
int('A') = 65
int('B') = 66
----------------
```


C++

Output Streams (2/2)

RECALL … RECALL … RECALL … RECALL

```
#include <iomanip>

cout << 1234 << ", " << hex << 1234 << ", " ;
cout << oct << 1234 << endl;
cout << endl;
cout << '(' << std::setw(4) << std::setfill('#') << 12  ;
cout << ") (" << 12 << ")"  << endl;
```

OR

```
#include <iomanip>
using namespace std;

cout << 1234 << ", " << hex << 1234 << ", " ;
cout << oct << 1234 << endl;
cout << endl;
cout << '(' << setw(4) << setfill('#') << 12  ;
cout << ") (" << 12 << ")"  << endl;
```

```
1234, 4d2, 2322

(##12) (12)
```

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.5679 << ' ' << 123456 << endl;

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.5679 << ' ' << 123456 << endl;
```

```
1234.5679 1234.5679 123456
1235 1235 123456
```

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
```


Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique

Data Types

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	**Data types Logic, Integers, Fixed point**
	Events, Sensitivity & Notification	Modules & Hierarchy	

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

Fundamental Types

RECALL … RECALL … RECALL … RECALL

Booleans

```
void f(int a, int b)
{
      bool b1 = a==b;
      // …
}

bool greater(int a, int b) { return a>b; }
bool b = 7;  // bool(7) is true, so b becomes true
int I = true;  // int(true) is 1, so I becomes 1

void g()
{
      bool a = true;
      bool b = true;

      bool x = a+b;        // a+b is 2, so x becomes true
      bool y = a | b;      // a|b is 1, so y becomes true
}
```

Character Types

```
char ch = 'a';

char ch[ ] = "Hello !";
```

Integer Types

```
int a = -3;                    long int a = -3;
int b = 7;                     long int b = 7;

unsigned int c = 8;            unsigned long int c = 8;
```

Floating-Point Types

```
float a = 3.5;              // single-precision
double b = 7.45;            // double-precision
long double c = 7,784;      // extended-precision

a = 2.4F;
b = 5.67;     // default precision
c = 67,89L;
```


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)

C++

Classes
Member Functions

ED STIC Master STIC Spécialité SE

A class is a user-defined type.

first solution

```
struct Date
{
      int d, m, y;
};

void init_date(Date& d, int, int, int);    // initialize d
void add_year(Date& d, int n);             // add n years to d
void add_month(Date& d, int n);            // add n months to d
void add_day(Date& d, int n);              // add n days to d
```

No explicit connections between the data type and these functions !

second solution : declaration

```
struct Date
{
      int d, m, y;

      void init(int dd, int mm, int yy);   // initialize
      void add_year(int n);                // add n years
      void add_month(int n);               // add n months
      void add_day(int n);                 // add n days
};
```

Right Solution !

second solution : definition of the member function

```
void Date::init(int dd, int mm, int yy)
{
      d =dd;
      m=mm;
      y=yy;
}
```

second solution : using example

```
Date my_birthday;

void f()
{
      Date today;

      today.init(14,10,2004);
      my_birthday.init(30,5,1997);

      Date tomorrow = today;
      tomorrow.add_day(1);
      /// …
}
```

RECALL … RECALL … RECALL … RECALL

C++

Classes
Access Control

ED STIC Master STIC Spécialité SE

```
struct X          class X
{                 {
     …                 public:
};                     …
                  };
```

private: only for members and friends
protected: only for members, friends and derived members
public: access for everyone

> default access is private

using class term

```
                class Date
                {
private part        int d, m;

                    public:

public part         void init(int dd, int mm, int yy);   // initialize
                    void add_year(int n);                // add n years
                    void add_month(int n);               // add n months
                    void add_day(int n);                 // add n days

                    private:
private part
                    int y;
                };
```

example

```
void f()
{
      Date today;

      today.init(14,10,2004);  // OK

      today.d = 14;    // ERROR !
      today.m = 10;    // ERROR !
      today.y = 2004;  // ERROR !
}
```

RECALL … RECALL … RECALL … RECALL

Classes
Constructor

C++

The use of functions such as init( ) to provide initialization for class objects is inelegant and error-prone

A solution is to allow the programmer to declare a function with the explicit purpose of initializing objects.
it is called a constructor

5 constructors

```
class Date
{
    int d, m, y;

    public:
    Date(int ,int ,int);    // day, month, year
    Date(int ,int);         // day, month, today's year
    Date(int);              // day, today's month and year
    Date( );                // default Date : today
    Date(char* );           // date in string representation

    // …
};
```

Example

```
Date today(4);
Date july4("July 4, 1999");
Date noel("24 dec");
Date now;   // default initialized as today
```

GUIDELINE: Prefers to use "struct" for classes that have all data public.

RECALL … RECALL … RECALL … RECALL

Classes
Template

C++

- Template provide direct support for generic programming
- programming using types as parameters
- allowing to classes and functions

Example for a class

```
template<class T, int i> class Buffer
{
    T v[i];
    int sz;
public:
    Buffer( ) : sz(i)
    {
        …              initialize sz = i
    }
};
```

```
struct Record
{
    int a,b;
};

Buffer<char, 127> cbuf;
Buffer<Record,8> rbuf;
```

RECALL … RECALL … RECALL … RECALL

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.

Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique

Data Types

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	**Data types Logic, Integers, Fixed point**
	Events, Sensitivity & Notification	Modules & Hierarchy	

- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- Fixed-Point Data 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 … ;
```

VHDL

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

<u>Example</u>

```
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

sc_logic and sc_lv (2/2) Resolution Table

AND

&	'0'	'1'	'X'	'Z'
'0'	'0'	'0'	'0'	'0'
'1'	'0'	'1'	'X'	'X'
'X'	'0'	'X'	'X'	'X'
'Z'	'0'	'X'	'X'	'X'

NOT

~	'0'	'1'	'X'	'Z'
	'1'	'0'	'X'	'X'

OR

|	'0'	'1'	'X'	'Z'
'0'	'0'	'1'	'X'	'X'
'1'	'1'	'1'	'1'	'1'
'X'	'X'	'1'	'X'	'X'
'Z'	'0'	'1'	'X'	'X'

XOR

^	'0'	'1'	'X'	'Z'
'0'	'0'	'1'	'X'	'X'
'1'	'1'	'0'	'X'	'X'
'X'	'X'	'X'	'X'	'X'
'Z'	'X'	'X'	'X'	'X'

```
using sc_dt;

sc_logic pulse, trig;
sc_bit select = LOG_1;

pulse != select;
select = trig;

trig = SC_LOGIC_Z;    // This is identical to :
trig = sc_logic('Z');

bool wrn;
sc_logic pena (SC_LOGIC_1);  // Initialize to '1'

wrn = pena.to_bool();

if (pena.to_bool())
    cout << "pena is " << pena << endl;
```

Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	**Data types Logic, Integers, Fixed point**
	Events, Sensitivity & Notification	Modules & Hierarchy	

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

Introduction (1/2)

- Fixed Point Type
  - wl (word length)
    - which shall be the total number of bits in the number representation.
  - iwl (integer word length)
    - which shall be the number of bits in the integer part (the position of the binary point relative to the left-most bit).
- Fixed-point representation, 3 cases
  - wl < iwl
    - There are (iwl-wl) zeros between the LSB and the binary point.
  - 0 <= iwl <= wl
    - The binary point is contained within the bit representation.
  - iwl < 0
    - There are (-iwl) sign extended bits between the binary point and the MSB.
    - For an unsigned type, the sign extended bits are zero.
    - For a signed type, the extended bits repeat the MSB.

Introduction (2/2)

Range of value for signed fixed point format $[-2^{(iwl-1)}, 2^{(iwl-1)} - 2^{-(wl-iwl)}]$

Range of value for unsigned fixed point format $[0, 2^{(iwl)} - 2^{-(wl-iwl)}]$

Index	wl	iwl	Fixed-point representation*	Range signed	Ranged unsigned
1	5	7	xxxxx00.	[-64,60]	[0,124]
2	5	5	xxxxx.	[-16,15]	[0,31]
3	5	3	xxx.xx	[-4,3.75]	[0,7.75]
4	5	1	x.xxxx	[-1,0.9375]	[0,1.9375]
5	5	0	.xxxxx	[-0.5,0.46875]	[0,0.96875]
6	5	-2	.ssxxxxx	[0.125,0.1171875]	[0,0.2421875]
7	1	-1	.sx	[-0.25,0]	[0,0.25]

wl < iwl

0 <= iwl <= wl

iwl < 0

*x is an arbitrary binary digit, 0, or 1. s is a sign extended digit, 0, or 1,

SystemC Fixed-Point Data Types

sc_fixed<WL, IWL, *[QUANT, OVFLW, NBITS]*> name;
sc_ufixed<WL, IWL, *[QUANT, OVFLW, NBITS]*> name;
sc_fix<WL, IWL, *[QUANT, OVFLW, NBITS]*> name;
sc_ufix<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<WL, IWL, *[QUANT, OVFLW, NBITS]*> name;
sc_ufix_fast<WL, IWL, *[QUANT, OVFLW, NBITS]*> name;

53 bits precision

SC_RND
SC_RND_ZERO
SC_RND_MIN_INF
SC_RND_INF
SC_RND_CONV
default → SC_TRN
SC_TRN_ZERO

SC_SAT
SC_SAT_ZERO
SC_SAT_SYM
default → SC_WRAP
SC_WRAP_SYM

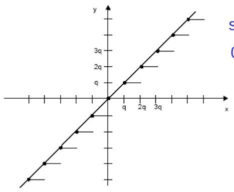
default
↓
Number of saturation Bits (1)

Example

```
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;
```

Examples

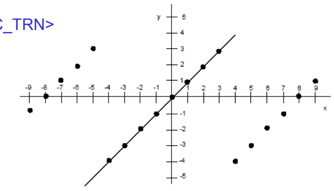
SC_TRN (default)



sc_fixed<4,2> → sc_fixed<3,2,SC_TRN>

01.01 (1,25) → 01.0 (1)

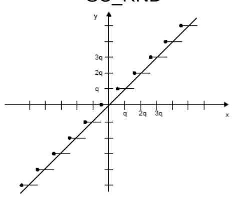
SC_WRAP (default)



signed, 3 bits number

0100 (4) → 100 (-4)
1011 (-5) → 011 (3)

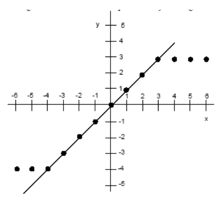
SC_RND



sc_fixed<4,2> → sc_fixed<3,2,SC_RND>

01.01 (1,25) → 01.1 (1.5)

SC_SAT



signed, 3 bits number

0110 (6) → 011 (3)
1011 (-5) → 100 (-4)

Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique

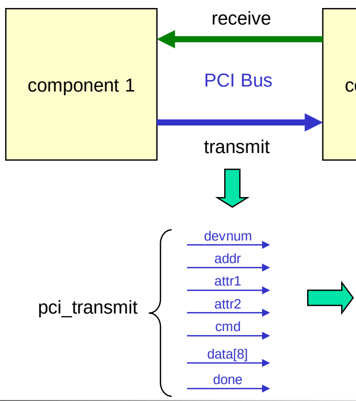
Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	**Data types Logic, Integers, Fixed point**
	Events, Sensitivity & Notification	Modules & Hierarchy	

Data Types

- Numeric Representation
- Native & Arithmetic Data Types
- Bit Types
- Fixed-Point Data 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




```
struct pci_transmit
{
    int devnum;
    int addr;
    int attr1;
    int attr2;
    int cmd;
    int data[8];
    bool done;
}
```


C++

Operator Overloading

RECALL … RECALL … RECALL … RECALL

- C++ supports a set of operators for built-in types
- But for user-defined types ?
  - No operators !
  - programmer has to define operators

syntax : type operator op (arg1, arg2, …)

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td></tr>
<tr><td>|</td><td>~</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td></tr>
<tr><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td></tr>
<tr><td><<</td><td>>></td><td>>>=</td><td><<=</td><td>==</td><td>!=</td><td><=</td></tr>
<tr><td>>=</td><td>&&</td><td>||</td><td>++</td><td>--</td><td>->*</td><td>,</td></tr>
<tr><td>-></td><td>[ ]</td><td>( )</td><td>new</td><td>new[ ]</td><td>delete</td><td>delete[ ]</td></tr>
</table>

Operator Overloading
Binary or Unary Operator

RECALL … RECALL … RECALL … RECALL

binary operator	prefix unary operator	postfix unary operator
aa @ bb	@ aa	aa @
aa.operator@(bb)	aa.operator@()	aa.operator@(int)
operator@(aa, bb)	operator@(aa)	operator@(aa, int)

never used !

member functions (with implicit "this" pointer)

```
class X
{
    X* operator& ( );       // prefix unary & (addr of)
    X operator& (X);        // binary & (and)
    X operator++ (int);     // postfix increment
    X operator& (X, X);     // ERROR: ternary
    X operator/ ( );        // ERROR: unary /
};
```

nonmember functions

```
X operator- (X);         // prefix unary minus
X operator- (X, X);      // binary minus
X operator-- (X&, int);  // postfix decrement
X operator- ( );         // ERROR: no operand
X operator- (X, X, X);   // ERROR: ternary
X operator% (X);         // ERROR: unary %
```


Operator Overloading
Example : Complex Numbers

RECALL … RECALL … RECALL … RECALL

complex.h

```
class complex
{
    double re, im;
public:
    complex(double r, double i) : re(r), im(i)
    {
        // r = re;
        // im = i;
    }

    // member functions
    complex& operator+= (complex a);
    complex& operator-= (complex a);
};

// nonmember functions
complex operator+ (complex a, complex b);
complex operator- (complex a, complex b);
```

complex.cpp

```
complex& complex::operator+= (complex a)
{
    this->re += a.re;
    this->im += a.im;
    return *this;
}

complex& complex::operator-= (complex a)
{
    re -= a.re;
    im -= a.im;
    return *this;
}

complex complex::operator+ (complex a, complex b)
{
    complex r =a;
    return r += b;  // call complex::operator+=(complex)
}

complex complex::operator- (complex a, complex b)
{
    complex r=a;
    return r -= b;  // call complex::operator-=(complex)
}
```

Best solution ! (Fast)

Operator Overloading
Example : Enumerations

RECALL … RECALL … RECALL … RECALL

Declaration

```
enum Day {sun, mon, tue, wed, thu, fri, sat };


Day& operator++ (Day& d)
{
    return d = (sat == d) ? sun : Day(d+1);
}
```

Using

```
Day day = mon;

// day = monday
day++;

// day = tuesday
```


Operator Overloading
Example : I/O with Complex Number Class

RECALL … RECALL … RECALL … RECALL

complex.h

```
class complex
{        …
};

// nonmember functions
istream& operator>> (istream&, complex&);    // input
ostream& operator<< (ostream&, complex);     // output
```

complex.cpp

```
…
istream& operator>> (istream& is, complex& c)
{
    is >> c.re >> c.im;
    return is;
}

ostream& operator<< (ostream& os, complex c)
{
    os << c.re << " + " << c.im << "j";
    return os;
}
```

test.cpp

```
#include <iostream>
using namespace std;

int main()
{
    complex a(2,5);
    complex b(1,3);

    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    a += b;
    cout << "a+b= " << a << endl;
    return 0;
}
```

```
a: 2 + 5j
b: 1 + 3j
a+b= 3 + 8j
```

Self-Reference

- Each member function knowns what object it was invoked for and can explicitly refer to it
- called "this" pointer
- "this" points to the current class
- "this" cannot be changed

```
class Date
{
    int d, m, y;

public:
    …
    Date& add_year(int n);       // add n years
    Date& add_month(int n);   // add n months
    Date& add_day(int n);        // add n days
};
```

Example

```
void f(Date& d, const Date& cd)
{
    …
    d.add_day(1).add_month(1).add_year(1);
}
```

```
Date& Date::add_year(int n)
{
    if (d == 29 && m == 2 && !leapyear(y+n))
    {
        d = 1;
        m = 3;
    }
    y += n;
    return *this;
}
```

OR

```
Date& Date::add_year(int n)
{
    if (this->d == 29 && this->m == 2 && !leapyear(this->y+n))
    {
        this->d = 1;
        this->m = 3;
    }
    this->y += n;
    return *this;
}
```


const
Constant values

C syntax

```
#define A 23
#define D 175.23f
```

C++ syntax (and SystemC)

```
const int A  = 23;
const float D = 175.23;

const int v[ ] = {2, 4, 6, 8};

const int x;  // ERROR !
```

```
void foo()
{
    A = 78;      // ERROR !
    v[2] += 1;  // ERROR !
}
```

const
Constant pointers & values

RECALL … RECALL … RECALL … RECALL

```
int a = 1;

const int c = 2;

const int* p1 = &c;  // OK
const int* p2 = &a;  // OK
```

constant values

```
int* p3 = &c;  // ERROR: init of int*
               // with const int *

*p3 = 7;  // try to change the value of c !

a = 3;
*p2 = 5;  // ERROR: p2 is const int *

p2 = &c;  // OK
```

```
int a = 1;
int b = 3;
const int c = 2;
const int d = 8;
const int* const p1 = &d;
int* const p2 = &a;

*p1 = 3;  // ERROR
*p2 = 7;  // OK

p1 = &c;  // ERROR: const pointer !
p2 = &b;  // ERROR: const pointer !
```

constant values

constant values and pointer

constant pointer

const
Functions (1/2)

RECALL … RECALL … RECALL … RECALL

to pass an input parameter by copy

```
void g (int p)
{
    p += 2;  // OK
    if (p == 5)  // OK, read
    { …
}

void h()
{
    int val;
    val = 2;
    g(val);
    // now, val = 2
}
```

to pass an input parameter by pointer or reference
(better if parameter is big)

```
void g (const int* p)
{
    // can't modify *p here !
    *p += 2;  // ERROR !
    if (*p == 5)  // OK, read
    { …
}

void h()
{
    int val;
    val = 2;
    g(&val);
}
```

```
void g (const int& p)
{
    // can't modify p here !
    p += 2;  // ERROR !
    if (p == 5)  // OK, read
    { …
}

void h()
{
    int val;
    val = 2;
    g(val);
}
```

const Functions (2/2)

RECALL … RECALL … RECALL … RECALL

date.h

```
class Date
{
    int d, m, y;

public:
    int day() const;
    int month() const;
    int year() const;
};
```

date.cpp

```
int Date::day() const
{
    return d;
}

int Date::month() const
{
    return m;
}

int year() const
{
    return y++;   // ERROR !
}
```

attempt to change member value in const function

another file

```
void f(Date& d, const Date& cd)
{
    int I = d.year();
    d.add_year(1);

    int j = cd.year();
    cd.add_year(1);     // ERROR: cannot change
                        // value of const cd

    d.add_day(1).add_month(1).add_year(1);
}
```

self-reference

inline

RECALL … RECALL … RECALL … RECALL

- the inline specifier is a hint to the compiler to avoid calling through the usual function call mechanism

Example with recursive inline function

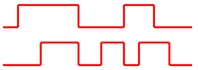
```
inline int fac(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}

main()
{
    int result = fac(6);
    …
}
```

```
720   6*fac(5)
120      5*fac(4)
24          4*fac(3)
6              3*fac(2)
2                 2*fac(1)
                     1
```

```
main()
{
    int result = 720;
    …
}
```

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

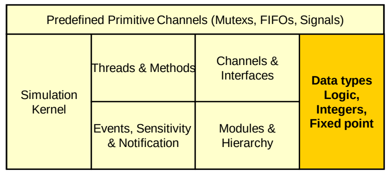
```
#include "systemc.h"

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");
}
```

École polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique

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

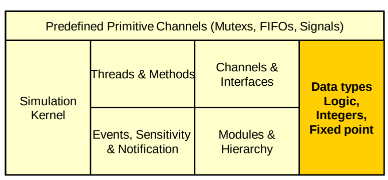
Standard Template Library (STL)

- 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

}
```


Ecole polytechnique
de l'université de Nice-Sophia Antipolis
Département Electronique



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

Operators for SystemC Data Types

- SystemC data types support all the common operations with operator overloading

	Operator		Special Methods
Comparison	== != > >= < <=	Bit Selection	bit(*idx*), [*idx*]
Arithmetic	++ -- * / % + -	Range Selection	range(*high,low*), (*high, low*)
Bitwise	~ & | ^	Conversion	to_double(), to_int(), to_int64(), to_long(), to_uint(), to_uint64(), to_ulong(), to_string(*type*)
Assignment	= &= |= ^= *= /= %= += -= <<= >>=	Testing	is_zero(), is_neg(), length()
		Bit Reduction	and_reduce(), nand_reduce(), or_reduce(), nor_reduce(), xor_reduce(), xnor_reduce()

Operators for SystemC Data Types Mixed Types

- For compiling mixed types, you must
  - have one of the arguments be same type as result
  - perform an explicit conversion of one of at least one of the operand arguments

result = arg1 op arg2 op arg3 … ;

Example

```
sc_int<3> d(3);
sc_int<5> e(15);
sc_int<7> 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 works !
bigsum = sc_bigint<70>(g) + h + i;  // Works
```

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<>

Native C/C++ Data Types (int, double, long and bool)

Fastest