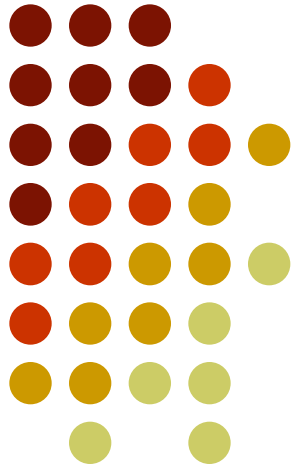
SISTEME DE CALCUL DEDICATE

Curs 2



Outline

- SystemC
  - Data types
  - Modules
- Bibliografie

Data types

- the use of SystemC data types
  - is not restricted to models using the simulation kernel
- simulation models may be created using any of the available data types
- the choice of data types affects
  - simulation speed
  - synthesizability
  - synthesis results
- the use of the native C++ data types
  - maximize simulation performance
  - decreases hardware fidelity and synthesizability

Data types

- The native C++ data types

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

```
// 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;
```

Data types

- the SystemC library provides data types for
  - digital logic
  - fixed-point arithmetic
- two SystemC logic vector types
  - 2-valued logic
  - 4-valued logic
- two SystemC numeric types
  - integers
  - fixed-point

Data types

- all of the SystemC data types (except **sc_logic**)
  - length configurable over a range much broader than the native C++ data types
- SystemC provides
  - assignment and initialization operations with type conversions
- SystemC data types
  - implement equality and bitwise operations

Data types

- SystemC arithmetic data types
  - implement arithmetic and relational operations
- SystemC data types
  - allow single-bit and multi-bit select operations
- SystemC data type are part of the **sc_dt** namespace

Data types

- two logic vector types
  - **sc_bv**<*W*> (bit vector)
    - values restricted to logic zero or logic one
  - **sc_lv**<*W*> (logic vector)
    - includes unknown and high impedance (tri-state)
      - 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’**
- a single-bit logic type
  - **sc_logic**

Data types

```
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;
```

```
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
```

Data types

- SystemC integer data types
  - templated
  - may have data widths from 1 to hundreds of bits
  - allow bit selections, bit range selections, and concatenation operations
- **sc_int**<*W*> and **sc_uint**<*W*>
  - provide an efficient way to model data with specific widths from 1- to 64-bits wide
- **sc_bigint**<*W*> and **sc_biguint**<*W*>
  - for modeling larger hardware

Data types

```
// 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
```

Data types

- fixed-point data types
  - address the need for non-integer data types when modeling DSP applications
- multiple fixed-point data types
  - signed and unsigned
  - compile-time (templated) and run-time configurable
  - fixed-precision and limited precision (_fast) versions
- parameters
  - word length (*WL*)
  - integer-word length (*IWL*)
  - quantization mode (*QUANT*)
  - overflow mode (*OVFLW*),
  - and number of saturation bits (*NBITS*)

Data types

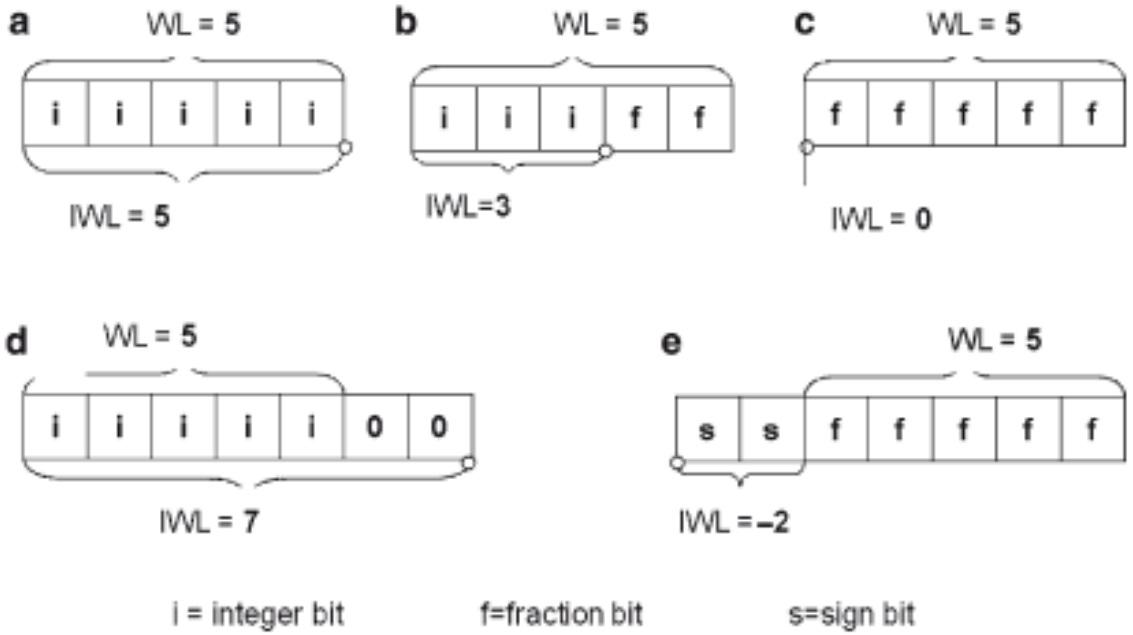
```
sc_fixed<WL,IWL[,QUANT[,OVFLW[,NBITS]>          NAME;
sc_ufixed<WL,IWL[,QUANT[,OVFLW[,NBITS]>         NAME;
sc_fixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]>   NAME;
sc_ufixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]> NAME;

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]);
```

```
// 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
```

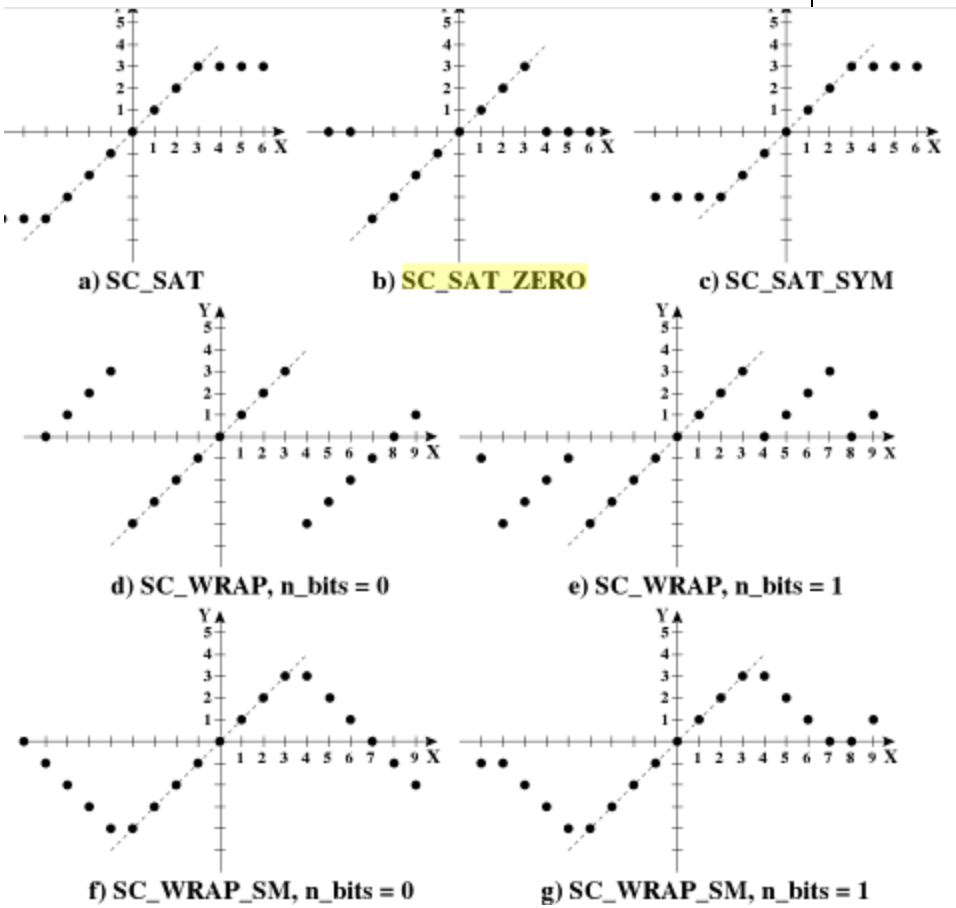
Data types

- example: **sc_fixed**<5,3>
  - represent values from –4.00 up to 3.75 in 1/4 increments



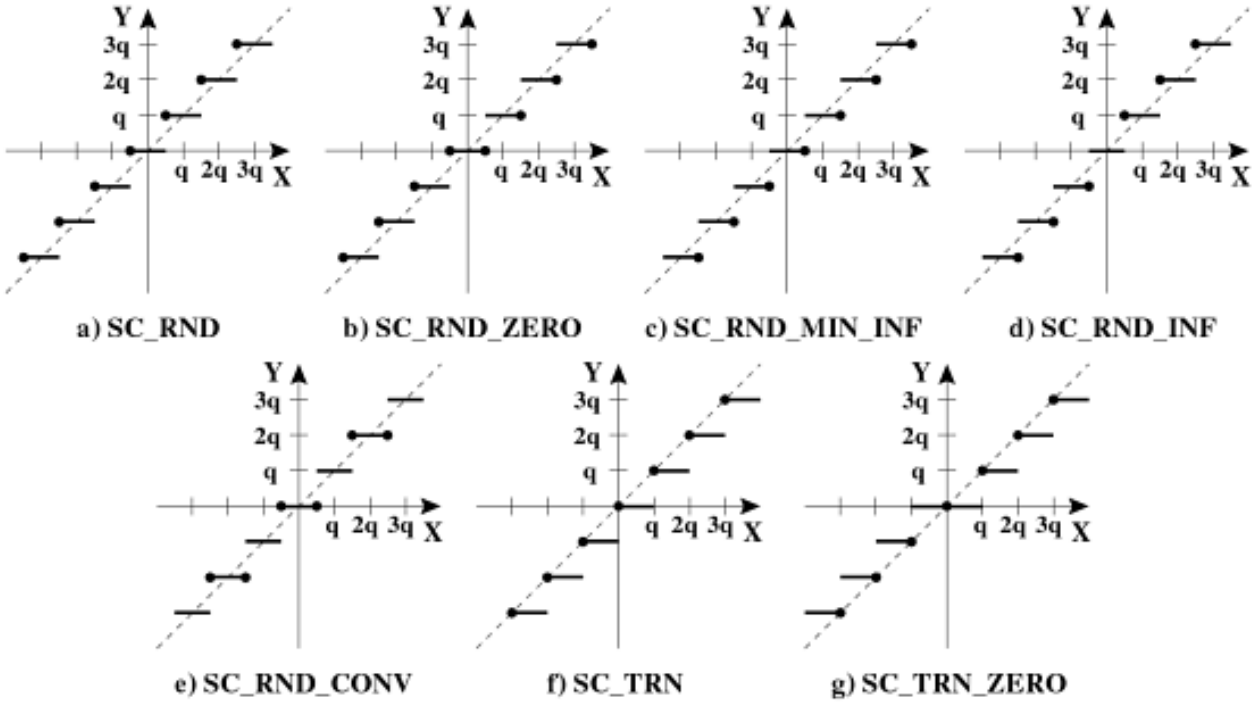
Data types

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



Data types

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*
SC_TRN	Truncate
SC_TRN_ZERO	Truncate towards zero



a) SC_RND b) SC_RND_ZERO c) SC_RND_MIN_INF d) SC_RND_INF

e) SC_RND_CONV f) SC_TRN g) SC_TRN_ZERO

Data types

- SystemC string literals may be used to assign values to any of the SystemC data types
- string literals consist of
  - a prefix
  - a magnitude
  - an optional sign character “+” or “-”
- instances of the SystemC data types
  - may be converted to a standard C++ string

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

Data types

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-

Data types

- SystemC data types
  - support all the common operations
    - with operator overloading
- SystemC provides
  - special methods to access bits, bit ranges
  - perform explicit conversions

Data types

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

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()`

Data types

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

Modules

- all programs need a starting point

```
int main(int argc, char* argv[]) {
    BODY_OF_PROGRAM
  return EXIT_CODE; // Zero indicates success
}
```

```
int sc_main(int argc, char* argv[]) {
  ELABORATION
  sc_start(); // <-- Simulation begins & ends
              //     in this function!
  [POST-PROCESSING]
  return EXIT_CODE; // Zero indicates success
}
```

Modules

- complex systems consist of many independently functioning components
- components may represent
  - hardware
  - software
  - any physical entity

```
#include <systemc>
SC_MODULE(module_name) {
  MODULE_BODY
};
```

Modules

- elements of *MODULE BODY:*
  - Ports
  - Member channel instances
  - Member data instances
  - Member module instances (sub-designs)
  - Constructor
  - Destructor
  - Simulation process member functions (processes)
  - Other methods (i.e., member functions)
- only the constructor is required

Modules

- The **SC_MODULE** constructor performs
  - Initializing/allocating sub-designs
  - Connecting sub-designs
  - Registering processes with the SystemC kernel
  - Providing static sensitivity
  - Miscellaneous user-defined setup
- To simplify coding, SystemC provides the macro, **SC_CTOR**().

Modules

```
SC_MODULE(module_name) {
  SC_CTOR(module_name)
  : Initialization // C++ initialization list
  {
    Subdesign_Allocation
    Subdesign_Connectivity
    Process_Registration
    Miscellaneous_Setup
  }
};
```

Modules

- SystemC simulation process
  - the basic unit of execution
- All simulation processes
  - are registered with the SystemC simulation kernel
  - are called by the kernel, and *only* from the SystemC simulation kernel
- ***DEFINITION***: A SystemC simulation process is a method (member function) of an **SC_MODULE** that is invoked by the scheduler in the SystemC simulation kernel.

```
void PROCESS_NAME(void);
```

Modules

- The most straightforward type of process to understand is the SystemC thread, **SC_THREAD**.
- a simple **SC_THREAD**
  - begins execution when the scheduler calls it
  - may also suspend itself
- a process method
  - must identify and register with the simulation kernel
  - allows the thread to be invoked by the simulation kernel's scheduler
  - the registration occurs within the module class constructor

Modules

```
SC_THREAD(process_name);//Must be INSIDE constructor
```

```
//FILE: basic_process_ex.h
SC_MODULE(basic_process_ex) {
  SC_CTOR(basic_process_ex) {
    SC_THREAD(my_thread_process);
  }
  void my_thread_process(void);
};
```

Modules

- alternative approach to creating constructors
  - uses macro named **SC_HAS_PROCESS**
- use **SC_HAS_PROCESS**
  - constructors with arguments
  - constructor in the implementation

```
//FILE: module_name.h
SC_MODULE(module_name) {
  SC_HAS_PROCESS(module_name);
  module_name(sc_module_name
              instname[, other_args…])
  : sc_module(instname)
  [, other_initializers]
  {
    CONSTRUCTOR_BODY
  }
};
```

Modules

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
…
SC_MODULE(NAME) {
     Port declarations
     Channel/submodule instances
     SC_CTOR(NAME)
     : Initializations
      {
         Connectivity
         Process registrations
     }
     Process declarations
     Helper declarations
};
#endif
```

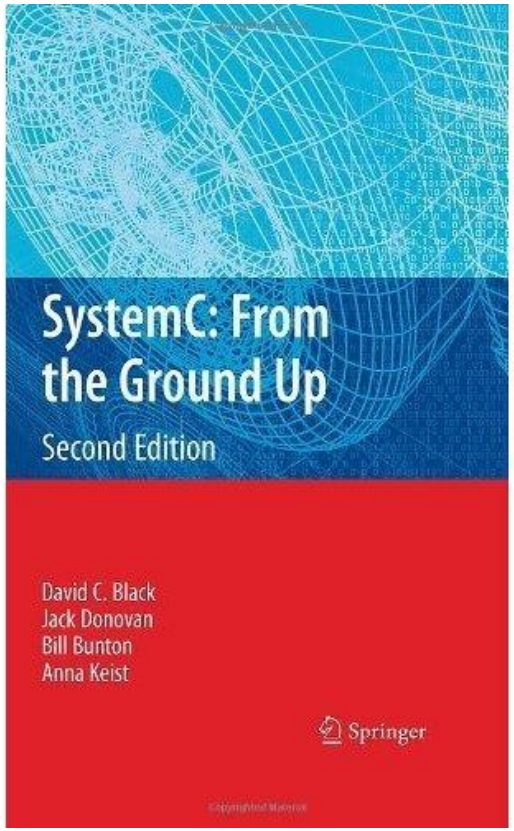
```
#include <systemc>
#include "NAME.h"
NAME::Process {implementations }
NAME::Helper {implementations }
```

Modules

```
#ifndef NAME_H
#define NAME_H
Submodule forward class declarations
SC_MODULE(NAME) {
    Port declarations
    Channel/Submodule* definitions
    // Constructor declaration:
    SC_CTOR(NAME);
    Process declarations
    Helper declarations
};
#endif
```

```
#include <systemc>
#include "NAME.h"
SC_HAS_PROCESS(NAME);
 NAME::NAME(sc_module_name nm)
: sc_module(nm)
, Initializations
{
    Channel allocations
    Submodule allocations
    Connectivity
    Process registrations
}
NAME::Process {implementations }
NAME::Helper {implementations }
```

Bibliografie



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *"*The authors designed this book primarily for the student or engineer new to SystemC. This book's structure is best appreciated by reading sequentially from beginning to end."