Technische Universität Chemnitz
Fakultät für Informatik
Professur Technische Informatik
Prof. Dr. Wolfram Hardt

HW/SW Codesign II

Co-Specification with SystemC

Prof. Dr. Wolfram Hardt
Dipl.-Inf. Michael Nagler

Sommersemester 2015

Contents

- $\boxed{\text{Introduction}}$
- Concepts and Basics of Language
- Example

What is SystemC?

- standardised language for system design and ______________________
- open source C++-class library which can be used with several operating systems
  - Windows, Linux, Solaris, …
- encloses the complete design process
- support for co-design of hard- and software
- supports:
  - several levels of abstraction
  - modularisation and partitioning
  - hierarchical design
- integration of design libraries ("Intellectual Property")

History

- we have seen a huge number of C/C++ based design languages before SystemC but no standard could be established

- introduction of SystemC in September 1999
- current version 2.3.1 (2014)
- accepted as IEEE 1666-2005

- standardisation by Open SystemC™ Initiative (OSCI)
  - consortium of EDA-Companies and IP-Provider
  - since 2011: Accellera Systems Initiative

accellera SYSTEMS INITIATIVE

AMD ARM cadence ERICSSON freescale

intel Mentor Graphics NXP QUALCOMM SONICS

ST SYNOPSYS TEXAS INSTRUMENTS

Why now SystemC?

- provide a standardised design flow for the developer
- use popular knowledge about programming with C/C++
- model and simulate HW systems above ______________________

➔ easy for software developers to model abstract HW/SW-systems

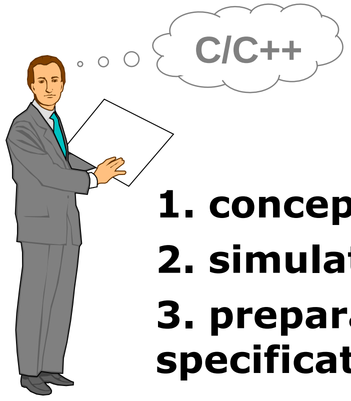
- time to market
  - design software and hardware at the same time
  - early prototyping and design exploration

    → find and fix bugs early

    → shorter design cycles

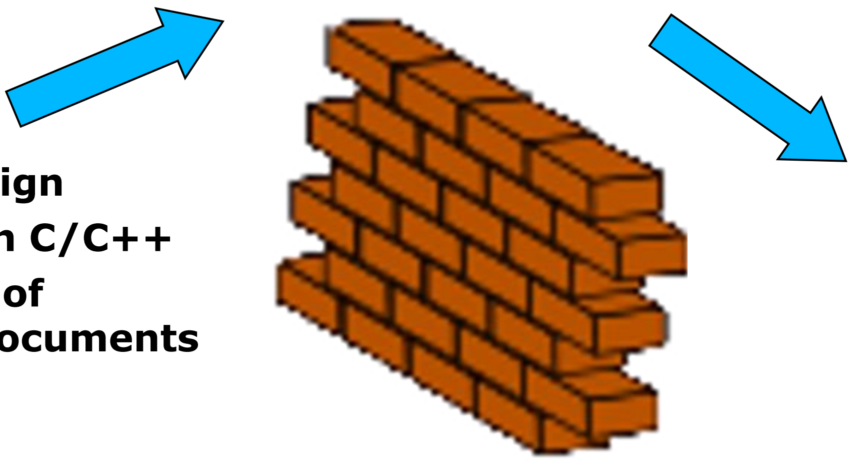
(V)HDL based Design Methodology

4. hand over specification documents



1. concept-design
2. simulation in C/C++
3. preparation of specification documents

system-architect

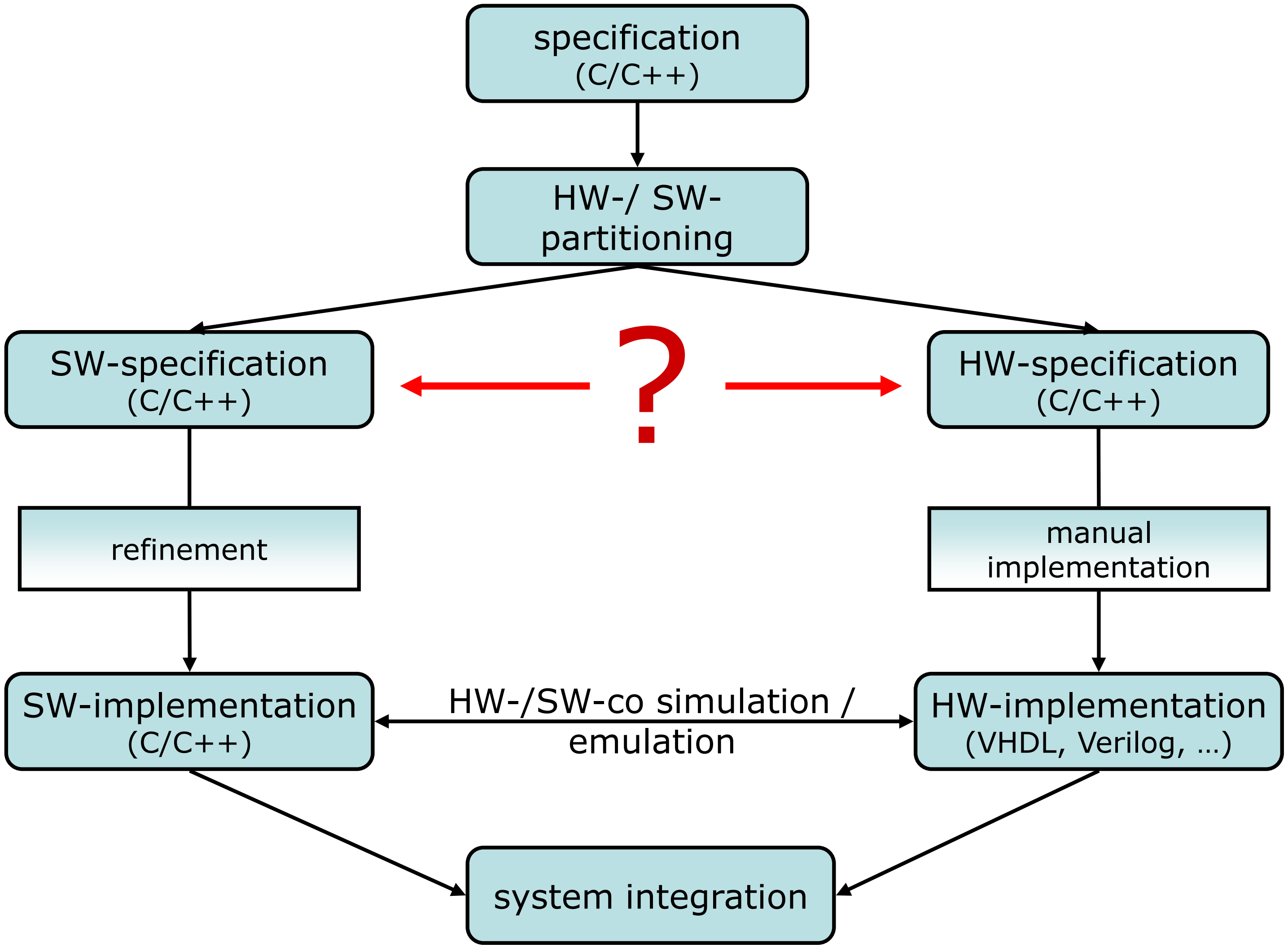


hardware-designer

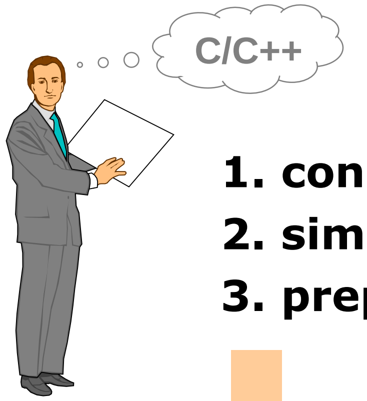
5. understand specification
6. (re)implement it in VHDL
7. evaluation / verification
8. ___________________________

Problems:
1. Specification is often incomplete and inconsistent
2. Translation to HDL is very time consuming and error prone

Standard Design Flow



C/C++ based Design Methodology

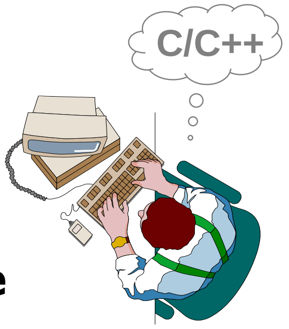


system-
architect

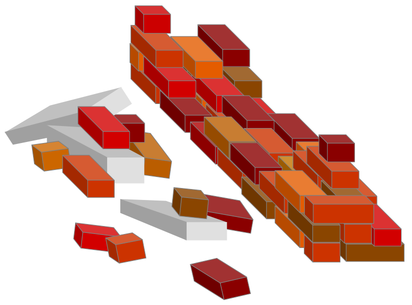
1. concept-design
2. simulation in C/C++
3. preparation of specification documents

4. hand over of
   - executable specification
   - test benches
   - specification documents

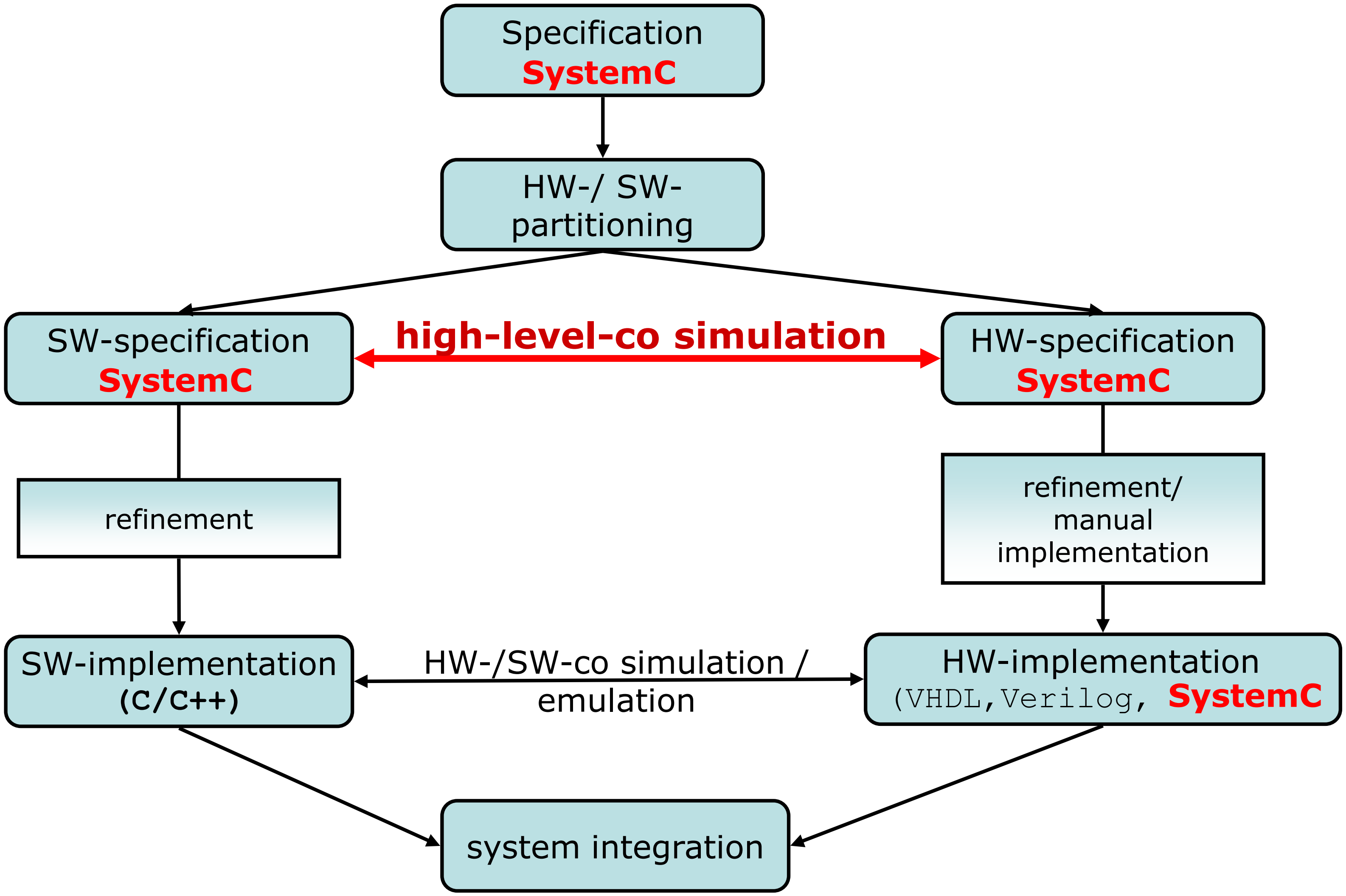
hardware-
designer



5. understand specification
6. refinement in C/C++
7. evaluation by testbench re-use
8. ______________________________



SystemC Design Flow



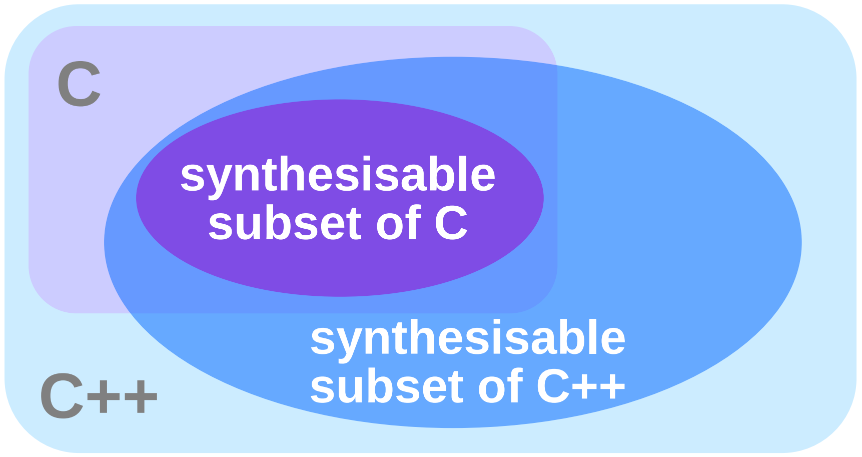
Synthesisable C/C++ ?

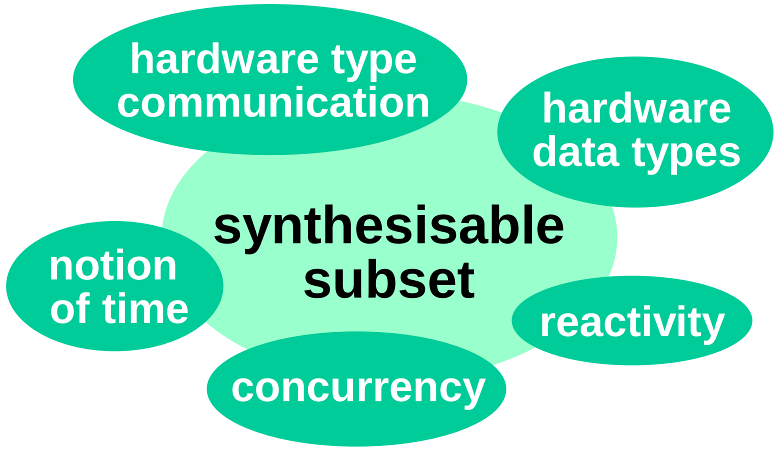
- step 1:

  restriction to synthesisable subset

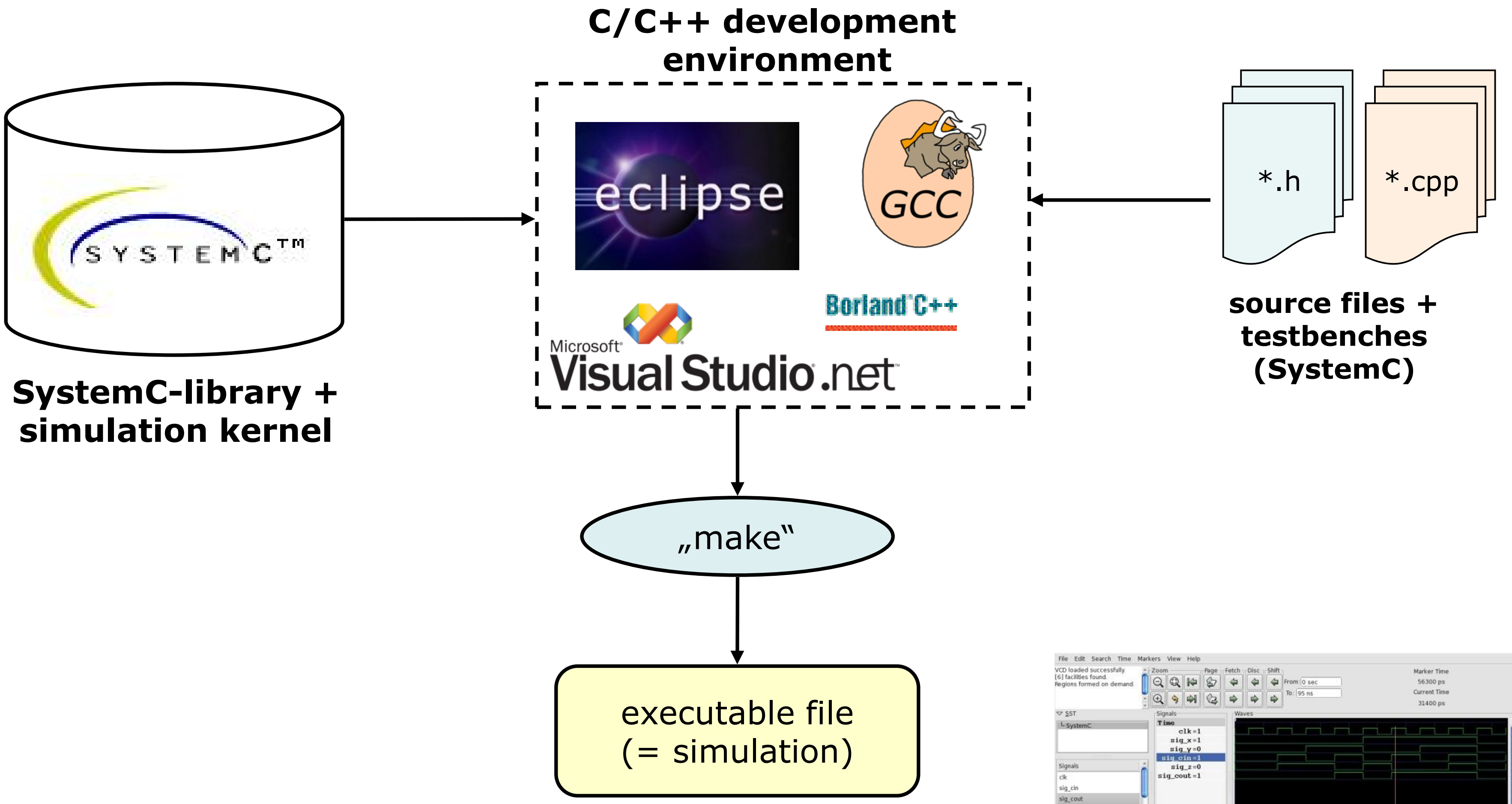
- step 2:

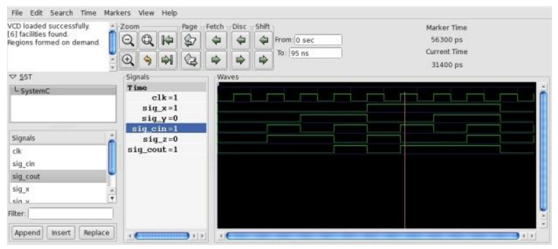
  extension by hardware-relevant components
  - new language constructs (HardwareC, C*)
  - ______________________ (**SystemC**, Cynlib)





SystemC Design Methodology


C/C++ development environment
SYSTEMC™
SystemC-library + simulation kernel
eclipse
GCC
Borland C++
Microsoft Visual Studio.net
*.h
*.cpp
source files + testbenches (SystemC)
„make"
executable file (= simulation)



```
VCD loaded successfully.
[6] facilities found.
Regions formed on demand.
```

From: 0 sec
To: 95 ns

Marker Time 56300 ps
Current Time 31400 ps

Signals: clk, sig_cin, sig_cout, sig_x, sig_y
Filter:

```
Time
      clk=1
    sig_x=1
    sig_y=0
  sig_cin=1
    sig_z=0
 sig_cout=1
```

Contents

- Introduction
- $\boxed{\text{Concepts and Basics of Language}}$
- Example

Datatypes

- C/C++ built-in types
- C/C++ user-defined types
- special SystemC types

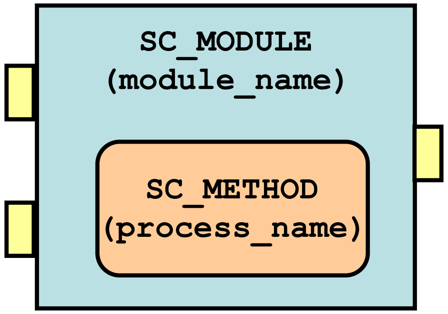
type	**description**
sc_bit	2-valued bit (0 or 1)
sc_logic	4-valued bit (0,1,X or Z)
sc_bv<n>	bit vector
sc_lv<n>	logic vector
sc_int<n>	1 to 64 bit signed integer
sc_uint<n>	1 to 64 bit unsigned integer
sc_bigint<n>	1 to 512 bit signed integer
sc_biguint<n>	1 to 512 bit unsigned integer
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

Modules

- modules are the basic building blocks
  → design partitioning

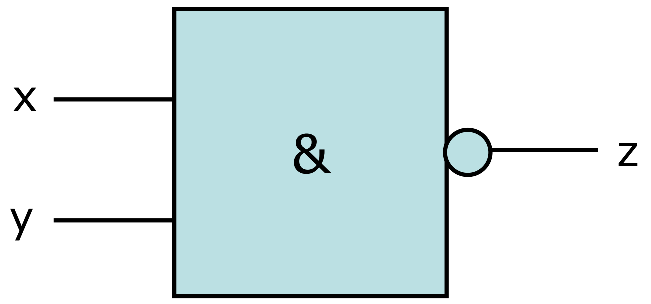
- includes:
  - processes → function
  - other modules → hierarchy

- a module is a ________________

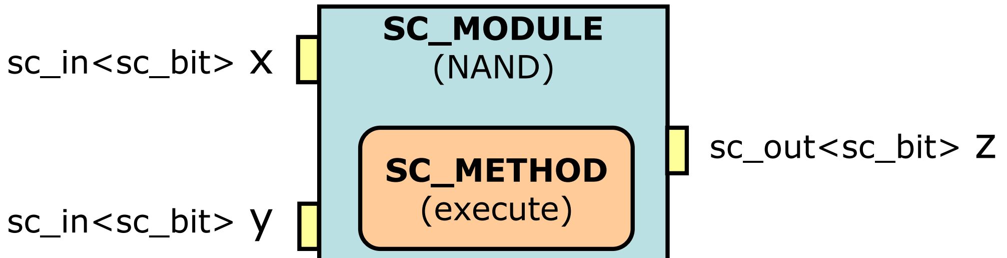


```
# include<systemc.h>

SC_MODULE(module_name)
{
  // declaration of module ports
  // declaration of local channels
  // declaration of module variables
  // declaration of processes
  // declaration of sub-modules
  // declaration of help functions
  // module instantiation

  /* module constructor */
  SC_CTOR(module_name)
  {
    // port mapping
    // module variable init.
    // process registration and
    // sensitivities
    SC_METHOD(process);
    sensitive << input1 << input2;
   }
};
```

Simple Example (I): MODULE



a) DIN-circuitsymbol of a NAND gate



b) SystemC-module of a NAND gate

```
#include <systemc.h>

SC_MODULE(NAND)
{
    sc_in<sc_bit> x, y;
    sc_out<sc_bit> z;

    SC_CTOR(NAND)
    {
        SC_METHOD(execute);
        sensitive << x << y;
    }

    void execute();
};
```

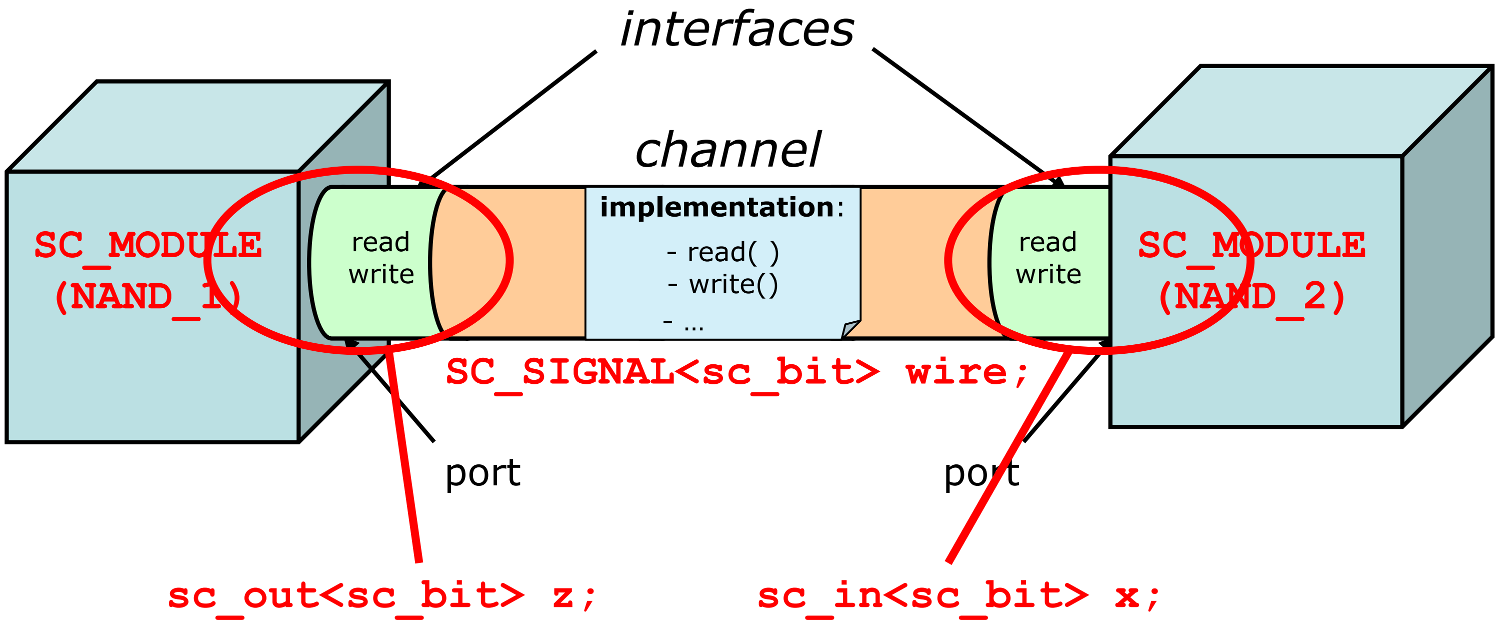
nand.h

```
#include <nand.h>

void NAND::execute()
{
    z.______(~(x.read() & y.read()));
}
```

nand.cpp

Simple Example (II): Communication between Modules

- components for communication between modules:
  - **ports** → define interface and data type
  - **interfaces** → specify a set of ________________________ to a channel
  - **channels** → implements the methods defined in the interfaces



example: NAND (prev. slide)

Processes

- describes the functionality of the system
- are the basic units of ______________________________
- communicate with each other through signals and channels
- there are several kinds of processes
  - **SC_METHOD**, **SC_THREAD**, **SC_CTHREAD**
- the SystemC simulation kernel invokes the processes
- takes no argument and returns nothing
- process declaration (in SC_MODULE):

```
    void process_name(void)
```

- specification of process-type and sensitivity list in **`SC_CTOR`**

Processes – Overview

- **SC_METHOD**
  - sensitive to signals
  - run completely after signal changes
  - for RTL-modelling / synthesis
- **SC_THREAD**
  - sensitive to signals
  - **`wait()`** suspends the execution of the process
  - easy description of state machines, test benches
  - not synthesisable
- **SC_CTHREAD**
  - sensitive for rising or falling clock edge
  - necessary for state machines
  - for ______________________

SC_METHOD

- never suspended within
  → run completely
- sensitive to a set of ________________
  → sensitivity list
  → executed, if one of the signals change
- e.g. to describe combinatorial logic

option 1:

```
void adder::add()
{
    sum.write(a.read() + b.read());
}
```

----

option 2:

```
void adder::add()
{
    sum = a + b;
}
```

adder.cpp

```
#include "systemc.h"

SC_MODULE(adder)
{
    // ports
    sc_in<int> a, b;
    sc_out<int> sum;

    // constructor
    SC_CTOR(adder)
    {
        SC_METHOD(add);
        sensitive << a << b;
    }

     void add();
};
```

adder.h

SC_THREAD

- started only once by the simulator
- **`wait()`** suspends the **`SC_THREAD`**
  → return control to the simulator
- can be sensitive to a set of __________
  → sensitivity list

```
#include "systemc.h"

SC_MODULE(adder)
{
    // ports
    sc_in<int> a, b;
    sc_out<int> sum;

    // constructor
    SC_CTOR(adder)
    {
        SC_THREAD(add);
        sensitive << a << b;
    }

     void add();
};
```

adder.h

```
void adder::add()
{
    while ( true )
    {
        sum.write(a.read() + b.read());
        wait();
    }
}
```

adder.cpp

infinity loop

SC_CTHREAD

- started only once by the simulator
- **`wait()`** suspends the **`SC_CTHREAD`**
  → return control to the simulator
- invoked by ______________
- only one edge is usable
- infinity loop

```
#include "systemc.h"

SC_MODULE(adder)
{
    // clock
    sc_in_clk clk;

    // ports
    sc_in<int> a, b;
    sc_out<int> sum;

    // constructor
    SC_CTOR(adder)
    {
        SC_CTHREAD(add, clk.pos());
    }

     void add();
};
```

adder.h

```
void adder::add()
{
    while ( true )
    {
        sum.write(a.read() + b.read());
        wait();
    }
}
```

adder.cpp

Comparison of the Processes

feature	**METHOD**	**THREAD**	**CTHREAD**
construct	SC_METHOD (process)	SC_THREAD (process)	SC_CTHREAD (process, clk.pos())
infinity loop	no	yes	yes
suspension	no	wait(…)	wait(…) and wait_until()
activation	events	events	clock pulse edge

sensitivity	sensitive(s)
	sensitive_pos(s)
	sensitive_neg(s)
		sensitive(s)
		sensitive_pos(s)
		sensitive_neg(s)
			SC_CTHREAD(process, clk.pos())
			SC_CTHREAD(process, clk.neg())

Suspension

instruction	**wait until …**	**process type**
`wait()`	event on a sensitive signal	**`SC_THREAD`** **`SC_THREAD`**
`wait_until(signal condition)`	signal condition is true	**`SC_CTHREAD`**
`wait(event condition)`	event condition is true	**`SC_THREAD`**
`wait(sc_time)`	a period of time	**`SC_THREAD`**
`wait(sc_time, event condition)`	event condition is true or a period of time	**`SC_THREAD`**

SystemC Scheduler

- SystemC contains an ________________ simulation kernel

- kernel
  - controls timing and process execution
  - handles event notification
  - updates the channels if requested

- the scheduler is invoked by **`sc_start()`**

- the scheduler continues until
  - there are no more events
  - a process stops it by calling the **`sc_stop()`** function
  - an exceptional condition occurs

sc_main

- main program of the SystemC design
- declaration and instantiation of the top level modules
- generate the top level structure
- initialisation and start of the simulation (**`sc_start()`**)
- **`sc_stop()`** stops the simulation (from all processes possible)

```
#include <systemc.h>
// include module headers

int sc_main(int argc, char *argv[])
{
    // channel declarations
    // variable declarations
    // module instance declarations
    // module port binding
    // time unit / resolution setup
    // set up tracing
    // start simulation

   return 0;
}
```

Contents

- Introduction
- Concepts and Basics of Language
- $\boxed{\text{Example}}$

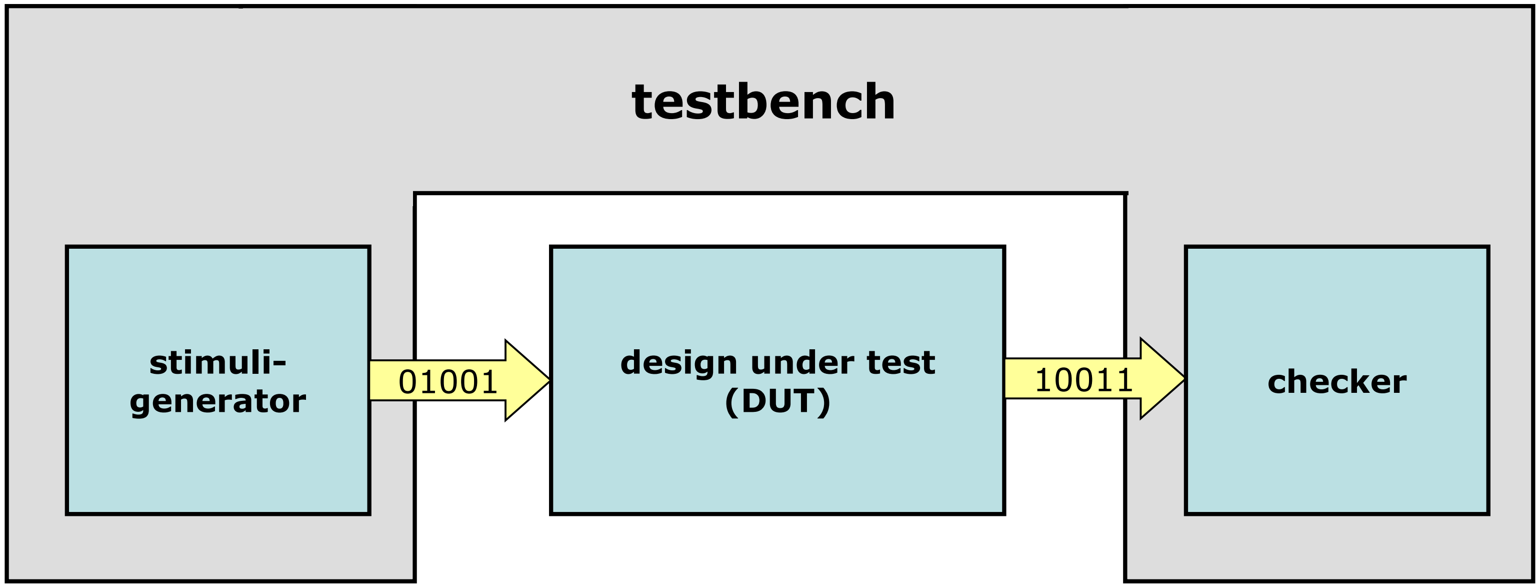
Modelling and Simulation: Example

- what we want (our design):
  - realise a hierarchical designed **full adder**

- what we need:
  - complete SystemC implementation of simple gates:
    → AND, NAND, OR (simple example on slide 14)
  - hierarchical components consists of simple gates:
    → XOR, half adder, full adder

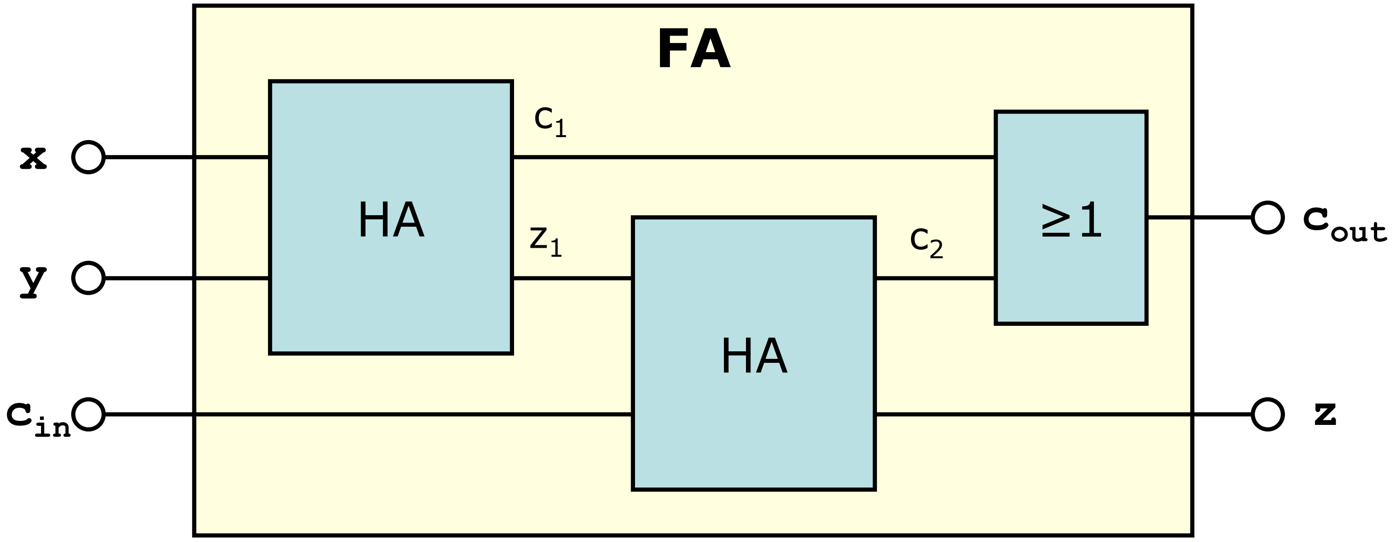
- and for simulation:
  - __________________ → generate the input values for the full adder
  - monitor → show results of calculation

Testbench


testbench
stimuli-generator
01001
design under test (DUT)
10011
checker

Hierarchical Full Adder

x	y	c_{in}	c_{out}	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

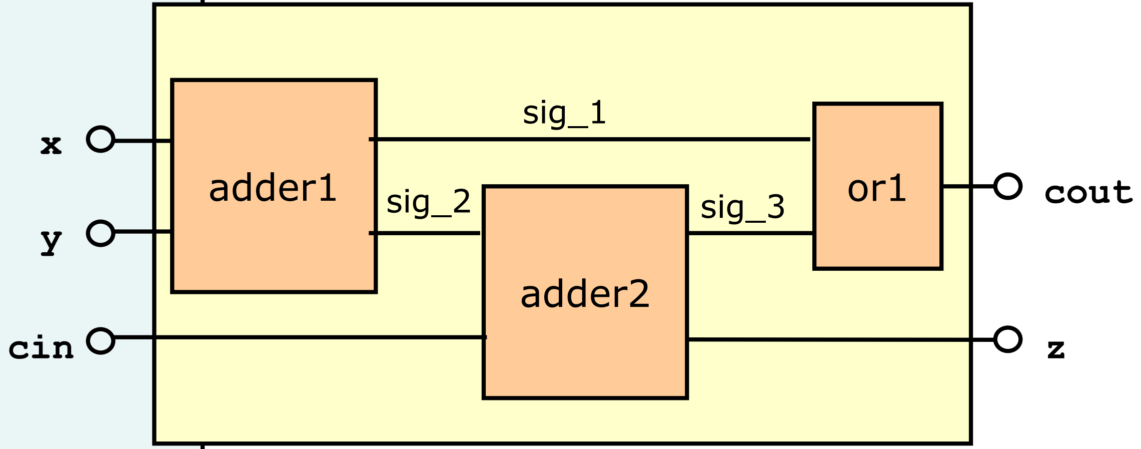


- __________________ designed full adder
  - consists of two half adders (HA) and one OR gate

Full Adder

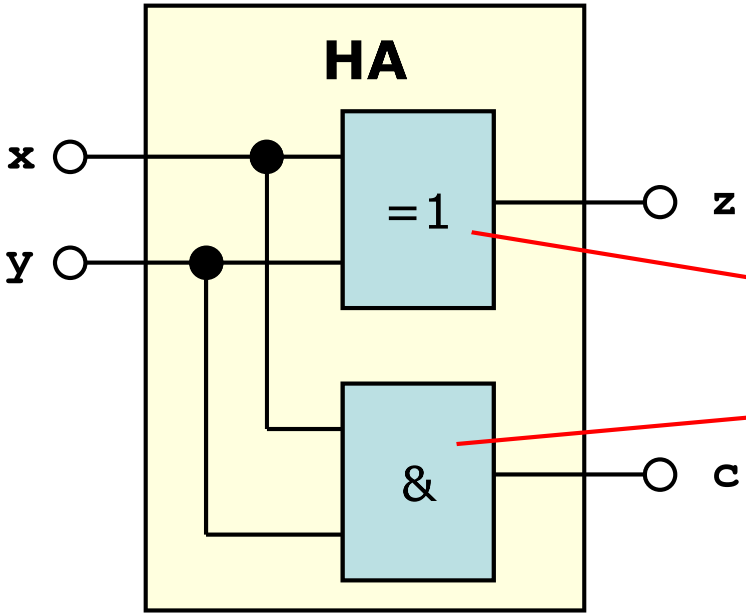
```
#ifndef FULLADDER_H_
#define FULLADDER_H_

#include <systemc.h>
#include "halfadder.h"
#include "or.h"

SC_MODULE(FullAdder)
{
    sc_in<sc_bit> x, y, cin;
    sc_out<sc_bit> z, cout;

     HalfAdder adder1, adder2;
     OR or1;

     sc_signal<sc_bit> sig_1, sig_2, sig_3;

     SC_CTOR(FullAdder) : adder1("HalfAdder1"),
                          adder2("HalfAdder2"),
                          or1("OR")
     {
         adder1 << x << y << sig_1 << sig_2;
         adder2 << sig_1 << cin << z << sig_3;
         or1 << sig_2 << sig_3 << cout;
     }
};
#endif
```

fulladder.h



Half Adder

x	y	c	z
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

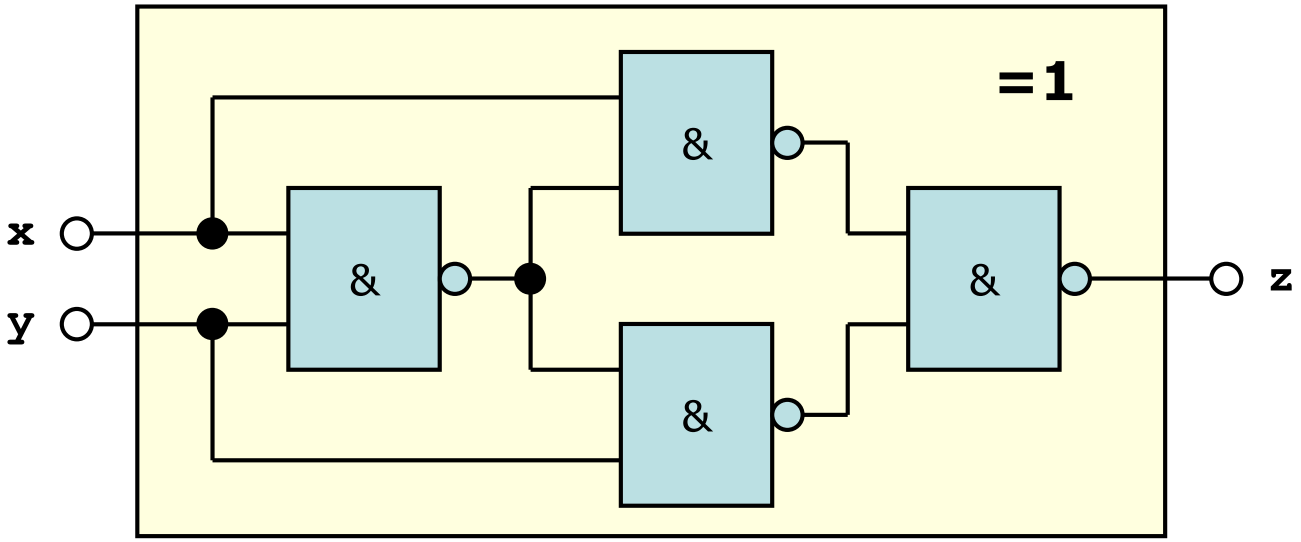


```
#ifndef HALFADDER_H_
#define HALFADDER_H_

#include <systemc.h>
#include "xor.h"
#include "and.h"

SC_MODULE(HalfAdder)
{
    sc_in<sc_bit> x, y;
    sc_out<sc_bit> z, c;

    XOR xor1;
    AND and1

    SC_CTOR(HalfAdder) : xor1("XOR"),
                         and1("AND")
    {
        // port mapping by position
        xor1 << x << y << z;
        and1 << x << y << c;
    }
};
#endif
```

halfadder.h

XOR

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

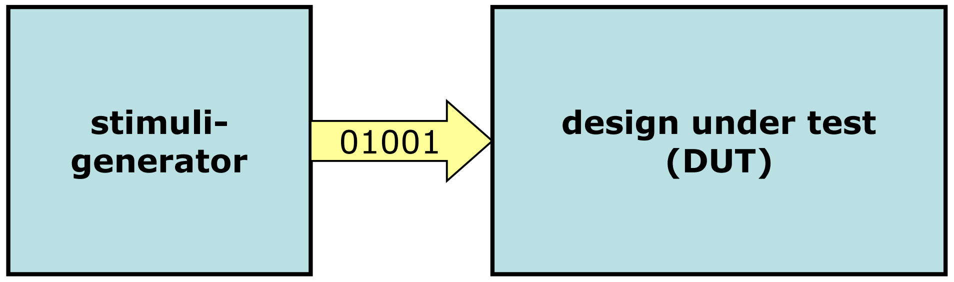


```
#include <systemc.h>
#include „nand.h“

SC_MODULE(XOR)
{
      sc_in<bool> x, y;
      sc_out<bool> z;

      nand n1, n2, n3, n4;
      sc_signal<bool> sig1, sig2, sig3;
```

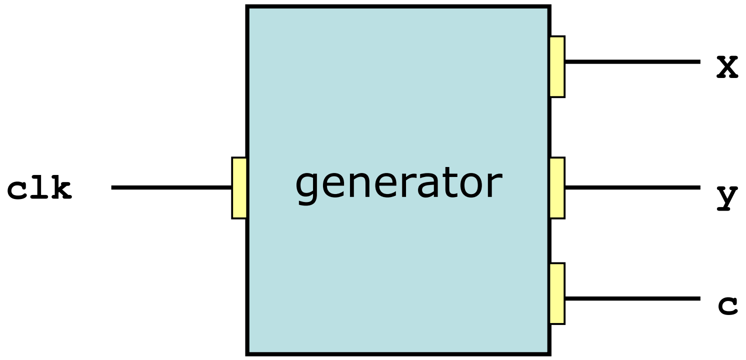
```
 SC_CTOR(XOR):n1(„N1“), n2(„N2“), n3(„N3“), n4(„N4“)
  {
    // connect ports and signals
    // connected by name
    n1.x(x); n2.y(y); n3.z(sig1);
    // connected by position
    n2 << x << sig1 << sig2;
    n3 << sig1 << y << sig3;
    n4 << sig2 << sig3 << z;
  }
}
```

xor.h

Stimuli-Generator



Stimuli-Generator



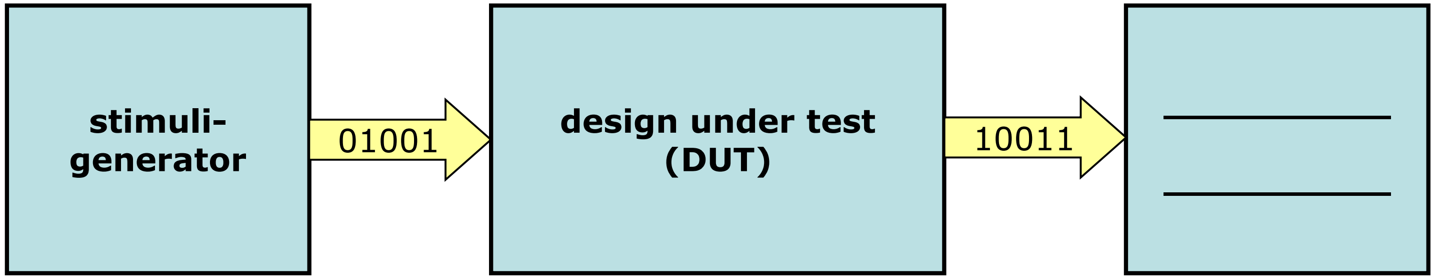
```
#include <systemc.h>

SC_MODULE(Generator)
{
    sc_in_clk clk;
    sc_out<sc_bit> x, y, cin;

    SC_CTOR(Generator)
    {
        SC_THREAD(run);
        sensitive << clk.pos();
        dont_initialize();
    }
    void run();
};
```

generator.h

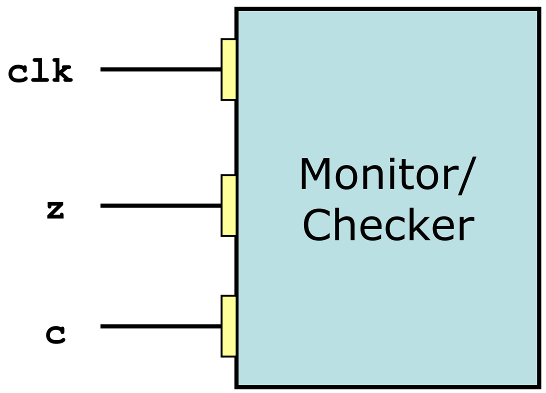
```
#include "generator.h"

void Generator::run()
{
    while(true)
    {
        x.write((sc_bit) 0);
        y.write((sc_bit) 0);
        cin.write((sc_bit) 0);
        wait();

        x.write((sc_bit) 0);
        y.write((sc_bit) 0);
        cin.write((sc_bit) 1);
        wait();

        ….

        x.write((sc_bit) 1);
        y.write((sc_bit) 1);
        cin.write((sc_bit) 1);
        wait();
    }
}
```

generator.cpp

Checker / Monitor



Checker / Monitor



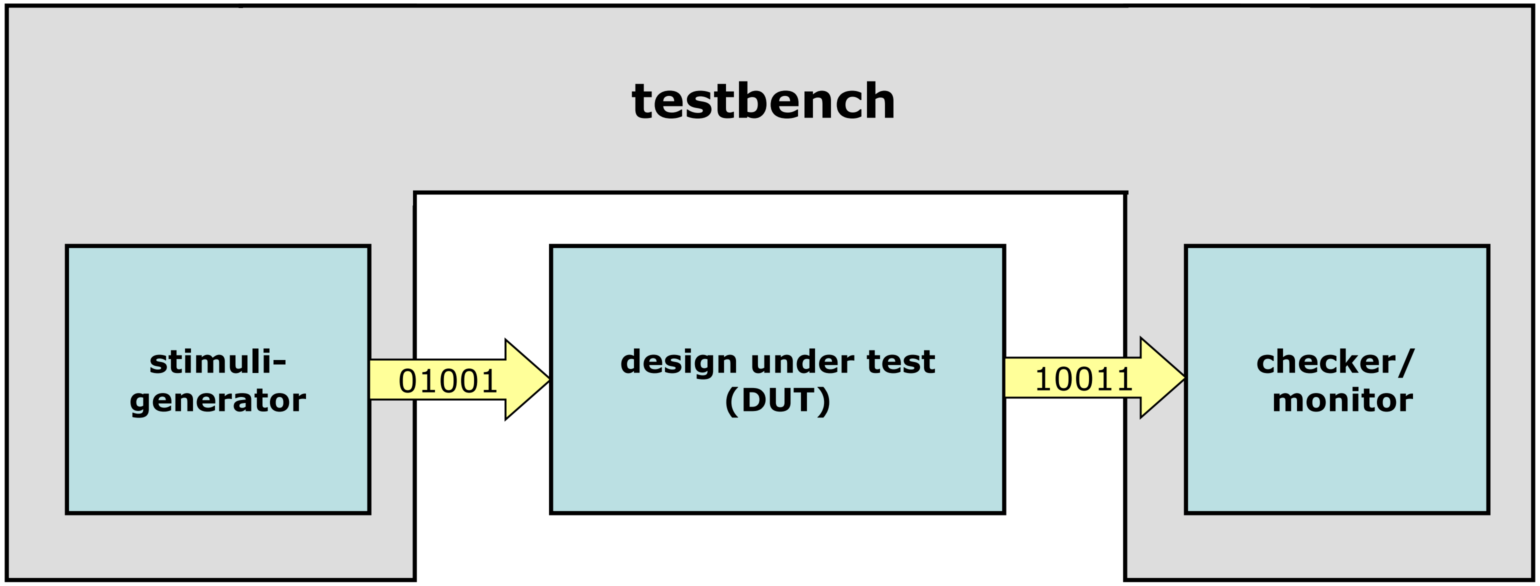
```
#include <systemc.h>
#include <string>
SC_MODULE(Monitor)
{
   sc_in_clk clk;
   sc_in<bool> z, c;

   SC_CTOR(Monitor)
   {
       SC_CTHREAD(mon, clk.pos());
   }

void mon()
}
```

monitor.h

```
#include "monitor.h"

void Monitor::execute()
{
   sc_bit tmp_z;
   sc_bit tmp_cout;

   tmp_z = z.read();
   tmp_cout = cout.read();

   std::cout << sc_time_stamp() << ": \t" <<
   name() << ": " << "\t z = " << tmp_z  <<
   "\t cout = " << tmp_cout << std::endl;
}
```

monitor.cpp

Testbench



- testbench:
  - connect design, stimuli-generator and checker/monitor
  - waveform tracing
  - start simulation

sc_main(1)

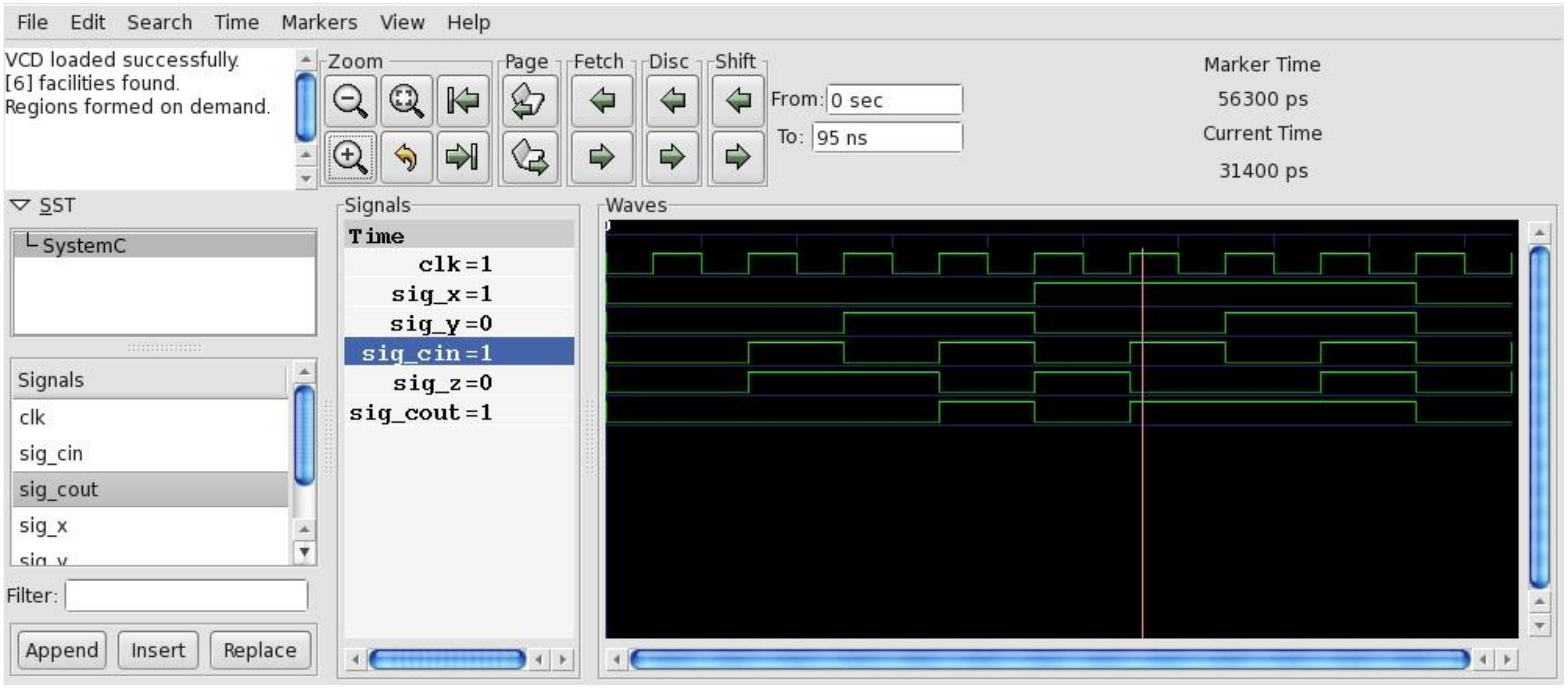
```
#include <systemc.h>
#include "generator.h"

#include "monitor.h"
#include "fulladder.h"

int sc_main(int argc, char *argv[])
{
    // signals
    sc_signal<sc_bit> sig_x, sig_y, sig_z;
    sc_signal<sc_bit> sig_cin, sig_cout;

    sc_clock clk( "clk", 10, SC_NS, 0, false );  // clock

    // moduls
    Generator generator("Generator");
    Monitor monitor("Monitor");
    FullAdder adder1("Adder");

     // port mapping
    generator.clk(clk);
    generator.x(sig_x);
    generator.y(sig_y);
    generator.cin(sig_cin);
```

sc_main(2)

```
    adder1.x(sig_x);
    adder1.y(sig_y);
    adder1.z(sig_z);
    adder1.cin(sig_cin);
    adder1.cout(sig_cout);

    monitor.clk(clk);
    monitor.z(sig_z);
    monitor.cout(sig_cout);

    // tracing the signals
    sc_trace_file *trace_file;
    trace_file = sc_create_vcd_trace_file("Trace_File_FullAdder");

    sc_trace(trace_file, clk, "clk");
    sc_trace(trace_file, sig_x, "sig_x");
    sc_trace(trace_file, sig_y, "sig_y");
    sc_trace(trace_file, sig_z, "sig_z");
    sc_trace(trace_file, sig_cin, "sig_cin");
    sc_trace(trace_file, sig_cout, "sig_cout");

    sc_start(100, SC_NS); // start the simulation for 100 ns

    sc_close_vcd_trace_file(trace_file);

    return 0;
}
```

Visualisation

- visualisation with gtkwave
- e.g.: **`gtkwave Trace_File_FullAdder.vcd`**



```
VCD loaded successfully.
[6] facilities found.
Regions formed on demand.
```

From: 0 sec
To: 95 ns

Marker Time 56300 ps
Current Time 31400 ps

```
Time
clk=1
sig_x=1
sig_y=0
sig_cin=1
sig_z=0
sig_cout=1
```