System-on-Chip Design with SystemC

Joachim Gerlach <gerlach@informatik.uni-tuebingen.de>

University of Tübingen
Wilhelm-Schickard-Institut
Department of Computer Engineering
Contents

- **Background & Basics**
  - System-on-Chip Design
  - C/C++ Based System Design
  - The SystemC Approach
  - SystemC Licensing Model
  - Open SystemC Community

- **Introduction to SystemC 1.0**
  - Modules & Hierarchie
  - Processes
  - Ports & Signals
  - Data Types & Fixed Point Data Types

- **Design Example A**
  - Simple 2-Process Scenario
Contents

- Design Example B
  - JPEG Compression / Decompression Stream

- Design Activities
  - Modeling
  - Simulation
  - Debugging

- Tool Support
  - Synopsys: SystemC Compiler
  - CoWare: N2C
  - C-Level Design: System Compiler
  - Frontier Design: A\RT Builder

- Outlook to SystemC 1.1
Background & Basics
Productivity Gap

(source: MEDEA Design Automation Roadmap 1999)
System Level Design

- System-on-Chips (SoC) designs
- SoC designs contain
  - Multiple design domains: hardware, software, analog, ...
  - Multiple source components: DSPs, ASICs, IP-Cores, ...
  - Hard constraints: realtime, low power, ...

Background & Basics
SystemC 1.0
Design Example A
Design Example B
Design Activities
Tool Support
SystemC 1.1

Wolfgang Rosenstiel
System Level Design Flow

- Specification
  - Functional validation
  - Co-design
  - Architectural validation

- Architectural models
  - Algorithmic models

- Hardware implementation
  - Software implementation
  - Hw/sw co-simulation co-verification

software architecture
- Device driver
- RTOS
- User software

hardware architecture
- Memory map
- Processor & peripherals
- Application specific co-processor
Benefits of a C/C++ Based Design Flow

- **Productivity aspect**
  - Specification between architect and implementer is executable
  - High speed and high level simulation and prototyping
  - Refinement, no translation into hardware (no “semantic gap”)

- **System level aspect**
  - Tomorrow’s systems designers will be designing mostly software and less hardware!
  - Co-design, co-simulation, co-verification, co-debugging, ...

- **Re-use aspect**
  - Optimum re-use support by object-oriented techniques
  - Efficient testbench re-use

- Especially C/C++ is widespread and commonly used!
Drawbacks of a C/C++ Based Design Flow

- C/C++ is not created to design hardware!
- C/C++ does not support
  - Hardware style communication
    - Signals, protocols
  - Notion of time
    - Clocks, time sequenced operations
  - Concurrency
    - Hardware is inherently concurrent, operates in parallel
  - Reactivity
    - Hardware is inherently reactive, responds to stimuli, interacts with its environment (→ requires handling of exceptions)
  - Hardware data types
    - Bit type, bit-vector type, multi-valued logic types, signed and unsigned integer types, fixed-point types
How to Get “Synthesizable C/C++”?

- **Step-1:**
  - restriction to synthesizable subset

- **Step-2:**
  - extension by hardware-related components
    - new language constructs (HardwareC, C*)
    - library based approach (SystemC, Cynlib)

- **Step-1 and step-2 can be swapped!**
Why SystemC for System Design?

- The Gap
  - Tomorrow’s systems designers will be designing mostly software and little hardware
  - A software language is not capable of describing concurrency, clocks, hardware data types, reactivity

- Requirements
  - Allow hardware/software co-design and co-verification
  - Fast simulation for validation and optimization
  - Smooth path to hardware and software
  - Support of design and architectural re-use
What is SystemC?

- **A library of C++ classes**
  - Processes (for concurrency)
  - Clocks (for time)
  - Modules, ports, signals (for hierarchy)
  - Waiting, watching (for reactivity)
  - Hardware data types

- **A modeling style**
  - ... for modeling systems consisting of multiple design domains, abstraction levels, architectural components, real-life constraints

- **A light-weight simulation kernel**
  - ... for high-speed cycle-accurate simulation
How Does it Work?

System

C/C++ Testbench

C/C++ Software Component

C/C++ Hardware Component

DSP
IP-Core
ASIC
Interface

Executable = Simulator

SystemC

Modeling Constructs

Standard C++ Compiler

Background & Basics

SystemC 1.0
Design Example A
Design Example B
Design Activities
Tool Support
SystemC 1.1

Joachim Gerlach

System-on-Chip Design with SystemC
Benefits of a SystemC-Based Design Flow

- Classical HDL based design methodology

1. conceptualize
2. simulate in C/C++
3. write specification document
4. hand over specification document
5. 
6. (re)implement in HDL
7. (re)validate HDL implementation
8. synthesize from HDL
Benefits of a SystemC-Based Design Flow

- C/C++ based design methodology

1. conceptualize
2. simulate in C/C++
3. write specification document

4. hand over
   - executable specification
   - testbenches
   - written specification

5. understand specification
6. refine in C/C++
7. validate re-using testbenches
8. synthesize from C/C++
The SystemC Approach

- The requirements...
  - Fast system modeling containing multiple source components
  - Model once for multiple abstraction level, multiple users, multiple purposes

- The problem...
  - No common format for describing components

- The approach...
  - Promote a standard C/C++ based modeling platform
    ... to model and exchange system level components and IP
    ... to build interoperable tools infrastructure
The SystemC Approach

Why C/C++ based?

- Specification between architect and implementer is executable
- High simulation speed at higher level of abstraction
- Refinement, no translation into HDL (no “semantic gap”)
- Efficient testbench re-use
SystemC Modeling Platform

SystemC™ is...

... a methodology for modeling SoC designs consisting of DSPs, ASICs, IP-Cores, Interfaces, ...

... a C++ library extending C/C++ by concurrency, timing, reactivity, communication, signal / data types, ...

... a cycle-accurate high-speed simulation
SystemC Design Methodology

Your standard C/C++ development environment

Class library and simulation kernel

Header files

Compiler

Linker

Debugger

"make"

"Executable specification"

Executable = simulation

Source files for system and testbenches

ASIC, DSP Interface IP-Core

Background & Basics

SystemC 1.0
Design Example A
Design Example B
Design Activities
Tool Support
SystemC 1.1

Joachim Gerlach

System-on-Chip Design with SystemC

18
SystemC Key Features

- **Concurrency** (Sync. and async. processes)
- **Notion of time** (Multiple clocks with arbitrary phase relation)
- **Data types** (Bit vectors, arbitrary precision integers, ...)
  - v1.0: arbitrary precision fixed point data types
- **Communication** (Signals, channels)
  - v1.0: advanced communication protocols
- **Reactivity** (Watching for events)
- **Debug support** (Waveform tracing)
- **Simulation support**
- **Support of multiple abstraction levels and iterative refinement**
- **Support of functional model creation**
- **....**
Open Community Licensing

- How to get SystemC?

Steering Group

SystemC v0.9 including:
- Modeling specification
- Source code (reference implementation)
- Reference manual

download

www.SystemC.org

click-through web-based license agreement

User
Open SystemC Steering Group

- ARM
- Cadence
- CoWare
- Ericsson
- Fujitsu Microelectronics
- Infineon Technologies
- Lucent Technologies
- Motorola
- NEC
- Sony Corporation
- STMicroelectronics
- Synopsys
- Texas Instruments
<table>
<thead>
<tr>
<th>Actel</th>
<th>Fujitsu Microelectronics</th>
<th>Sony Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alcatel</td>
<td>Genedax</td>
<td>Stellar Semiconductor</td>
</tr>
<tr>
<td>Altera Corp.</td>
<td>IKOS Systems</td>
<td>STMicroelectronics</td>
</tr>
<tr>
<td>American</td>
<td>I-Logix</td>
<td>Summit Design</td>
</tr>
<tr>
<td>Applied</td>
<td>Infineon Technologies</td>
<td>Sun Microsystems</td>
</tr>
<tr>
<td>Research</td>
<td>Integrated Silicon Systems</td>
<td>SynaptiCAD</td>
</tr>
<tr>
<td>Aptix</td>
<td>Intellectual Property</td>
<td>Synchronicity</td>
</tr>
<tr>
<td>Arcadia Design</td>
<td>Internet CAD</td>
<td>Synopsys</td>
</tr>
<tr>
<td>Systems</td>
<td>JTA Research</td>
<td>Tensilica</td>
</tr>
<tr>
<td>ARC Cores</td>
<td>LogicVision</td>
<td>Texas Instruments</td>
</tr>
<tr>
<td>Aristo</td>
<td>Lucent Technologies</td>
<td>TransModeling</td>
</tr>
<tr>
<td>Technology</td>
<td>Magma Design Automation</td>
<td>Ultima</td>
</tr>
<tr>
<td>ARM</td>
<td>MIPS Technologies</td>
<td>Verplex</td>
</tr>
<tr>
<td></td>
<td>Monterey Design Systems</td>
<td>Viewlogic</td>
</tr>
<tr>
<td></td>
<td>Motorola</td>
<td>Virtio</td>
</tr>
<tr>
<td></td>
<td>Red Hat</td>
<td>Virtual Silicon</td>
</tr>
<tr>
<td></td>
<td>Seva Technologies (Intrinsix)</td>
<td>Technologies</td>
</tr>
<tr>
<td></td>
<td>Sican Microelectronics</td>
<td>Wind River Systems</td>
</tr>
<tr>
<td></td>
<td>Snakeltech</td>
<td>Xilinx</td>
</tr>
</tbody>
</table>
Open Community Licensing

- Community members
  - No licensing fees, anybody / any company is free and welcome to join the community
  - Right and responsibility to contribute enhancements
  - Designers can create and share models with other companies, EDA vendors can build SystemC based tools

- Steering Group
  - Drives convergence and interoperability
  - Ensures open evolution and structured innovation

- Goal:
  - Make SystemC a de-facto-standard for system-level design
  - Provide a foundation to build a market upon
Short History of SystemC

1997 DAC Paper

V0.9 launches 9/27/1999

V1.0 release 3/28/2000

fixed point datatypes

HDL constructs
European SystemC Users Group

take a look at:
www-ti.informatik.uni-tuebingen.de/~systemc
SystemC 1.0
Modules

- Modules are basic building blocks of a SystemC design
- A module contains processes (→ functionality) and/or sub-modules (→ hierarchical structure)

```c
SC_MODULE(module_name) {
    // Declaration of module ports
    // Declaration of module signals
    // Declaration of processes
    // Declaration of sub-modules
    SC_CTOR(module_name) {       // Module constructor
        // Specification of process type and sensitivity
        // Sub-module instantiation and port mapping
    }
    // Initialization of module signals
}
```
Modules

- A module correspond to a C++ class

  class data members ↔ ports
  class member functions ↔ processes
  class constructor ↔ process generation

  

  SC_MODULE( module_name ) {
      
  }; 

  struct module_name : sc_module {
      
  };
Ports

- External interface of a module
- Passing data from and to processes / sub-modules
- Triggering of actions within the module
- A port has a mode (direction) and a type
  - mode: in, out, inout
  - type: C++ type, SystemC type, user-defined type

```cpp
// input port declaration
sc_in< type > in_port_name;

// output port declaration
sc_out< type > out_port_name;

// bidirectional port declaration
sc_inout< type > inout_port_name;
```

- Vector port / port array:

```cpp
sc_out< int > result [32];
```
Signals

- Connects a port of one module to the port of another module
- Local to a module
- Signal semantics is the same as VHDL and Verilog deferred assignment semantics
- A signal has a type
  type: C++ type, SystemC type, user-defined type

```cpp
// signal declaration
sc_signal< type > signal_name;
```

- Vector signal / signal array:

```cpp
sc_signal< double > a[4];
```

- Internal data storage not by signals but by local variables
  Local variable types: C++ types, SystemC types, user-defined types
Port and Signal Binding

- Ports and signals to be bound need to have the same type
- A signal connects two ports
- A port is bound to one signal (port-to-signal)
  or to one sub-module port (port-to-port)

Resolution

- SystemC supports resolved ports and signals
- Resolved ports/signals have 4-valued logic type (0,1,Z,X)
- Resolved ports/signals allow multiple drivers
- Resolved vector ports/vector signals

```c
sc_in_rv< n > x;  // n bits wide resolved input port
sc_signal_rv< n > y;  // n bits wide resolved signal
```
Clocks

- SystemC provides a special object \textit{sc\_clock}
- Clocks generate timing signals to synchronize events
- Multiple clocks with arbitrary phase relations are supported

- Clock generation:

\[
\text{sc\_clock clock\_name ("label", period, duty\_ratio, offset, start\_value);}\]

Example: \texttt{sc\_clock my\_clk ("CLK", 20, 0.5, 5, true);};

- Clock binding:

Example: \texttt{my\_module.clk( my\_clk.signal());}
Data Types

- SystemC supports
  - Native C/C++ types
  - SystemC types
  - User-defined types

- SystemC types
  - 2-value (‘0’, ‘1’) logic / logic vector
  - 4-value (‘0’, ‘1’, ‘Z’, ‘X’) logic / logic vector
  - Arbitrary sized integer (signed/unsigned)
  - Fixed point types (signed/unsigned, templated/untemplated)
Native C/C++ Data Types

- **Integer types:**
  - char
  - unsigned char
  - short
  - unsigned short
  - int
  - unsigned int
  - long
  - unsigned long

- **Floating point types**
  - float
  - double
  - long double
### SystemC Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_bit</td>
<td>2-value single bit</td>
</tr>
<tr>
<td>sc_logic</td>
<td>4-value single bit</td>
</tr>
<tr>
<td>sc_int</td>
<td>1 to 64 bit signed integer</td>
</tr>
<tr>
<td>sc_uint</td>
<td>1 to 64 bit unsigned integer</td>
</tr>
<tr>
<td>sc_bigint</td>
<td>arbitrary sized signed integer</td>
</tr>
<tr>
<td>sc_biguint</td>
<td>arbitrary sized unsigned integer</td>
</tr>
<tr>
<td>sc_bv</td>
<td>arbitrary length 2-value vector</td>
</tr>
<tr>
<td>sc_lv</td>
<td>arbitrary length 4-value vector</td>
</tr>
<tr>
<td>sc_fixed</td>
<td>templated signed fixed point</td>
</tr>
<tr>
<td>sc_ufixed</td>
<td>templated unsigned fixed point</td>
</tr>
<tr>
<td>sc_fix</td>
<td>untemplated signed fixed point</td>
</tr>
<tr>
<td>sc_ufix</td>
<td>untemplated unsigned fixed point</td>
</tr>
</tbody>
</table>
sc_bit / sc_logic

- 2-value single bit type: sc_bit
  - ‘0’=false, ‘1’=true

- 4-value single bit type: sc_logic
  - ‘0’=false, ‘1’=true, ‘X’=unknown/indeterminate value, ‘Z’=high-impedance/floating value

- Features:
  - Mixed use of operand types sc_bit and sc_logic
  - Use of character literals for constant assignments

- sc_bit / sc_logic operators

<table>
<thead>
<tr>
<th>Bitwise</th>
<th>&amp; (and)</th>
<th></th>
<th>(or)</th>
<th>^ (xor)</th>
<th>~ (not)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>=</td>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>^=</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>!=</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sc_int / sc_uint / sc_bigint / sc_biguint

- **Fixed precision integer types**
  - **Signed:** `sc_int<n>` \((n: \text{word length}, 1 \leq n \leq 64)\)
  - **Unsigned:** `sc_uint<n>` \((n: \text{word length}, 1 \leq n \leq 64)\)

- **Arbitrary precision integer types**
  - **Signed:** `sc_bigint<n>` \((n: \text{word length}, n > 64)\)
  - **Unsigned:** `sc_biguint<n>` \((n: \text{word length}, n > 64)\)

- **Features:**
  - **Mixed use of operand types** `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint` and C++ integer types
  - **Truncation and/or sign extension if required**
  - **2’s complement representation**
## sc_int / sc_uint / sc_bigint / sc_biguint operators

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise</td>
<td>&amp;</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+</td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;</td>
</tr>
<tr>
<td>Auto-Ink/Dek</td>
<td>++</td>
</tr>
<tr>
<td>Bit/Part Select</td>
<td>[ ]</td>
</tr>
<tr>
<td>Concatenation</td>
<td>(,)</td>
</tr>
</tbody>
</table>
sc_bv / sc_lv

- Arbitrary length bit vector: \texttt{sc\_bv}\langle n\rangle \, (n: \text{vector length})
- Arbitrary length logic vector: \texttt{sc\_lv}\langle n\rangle \, (n: \text{vector length})
- Features:
  - Assignment between \texttt{sc\_bv} and \texttt{sc\_lv}
  - Use of string literals for vector constant assignments
  - Conversions between \texttt{sc\_bv}/\texttt{sc\_lv} and SystemC integer types
  - No arithmetic operation available

\texttt{sc\_bv} / \texttt{sc\_lv}

---

| Bitwise   | &  | | ^ | ~ | >> | << |
| Assignment| =  | += | -= | *= | /= | %= | &= | |= | ^= |
| Equality  | == | != |
| Bit/Part Select | [] | range() |
| Concatenation | (,) |
| Reduction  | and\_reduction() | or\_reduction() | xor\_reduction() |
| Conversion | to\_string() |

---
sc_fixed / sc_ufixed / sc_fix / sc_ufix

- **Fixed point types**
  - sc_fixed
  - sc_ufixed
  - sc_fix
  - sc_ufix

- **templated** - static arguments (to be known at compile time)
- **untemplated** - nonstatic arguments (to be configured during runtime)

- **signed** - 2’s complement representation
- **unsigned**

- **Features:**
  - Operations performed using arbitrary precision
  - Multiple quantization and overflow modes
sc_fixed / sc_ufixed / sc_fix / sc_ufix

- Templated signed fixed point type: \texttt{sc\_fixed}

\begin{verbatim}
sc_fixed<\ text{wl, iw, q\_mode, o\_mode, n\_bits}> \texttt{var\_name} (\texttt{init\_val});
\end{verbatim}

- Arguments:
  - \texttt{wl} - total number of bits
  - \texttt{iw} - number of integer bits
  - \texttt{q\_mode} - quantization mode \hspace{1cm} (optional)
  - \texttt{o\_mode} - overflow\_mode \hspace{1cm} (optional)
  - \texttt{n\_bits} - number of bits for overflow mode \hspace{1cm} (optional)
sc_fixed / sc_ufixed / sc_fix / sc_ufix

- Example:

```
sc_fixed< 8, 4 > my_var (-1.75);
```

\[
(1.75)_{10} = (0001.1100)_{2}
\]

1’s complement of \((0001.1100)_{2}\) = \((1110.0011)_{2}\)
2’s complement of \((0001.1100)_{2}\) = \((1110.0100)_{2}\)

**my_var:**

```
  1 1 1 0 0 1 0 0
```

- Sign bit
- Integer bits
- Fractional bits
Quantization and overflow modes

<table>
<thead>
<tr>
<th>Quantization Mode</th>
<th>Overflow Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rounding to plus infinity</td>
<td>Saturation</td>
</tr>
<tr>
<td>Rounding to zero</td>
<td>Saturation to zero</td>
</tr>
<tr>
<td>Rounding to minus infinity</td>
<td>Symmetrical saturation</td>
</tr>
<tr>
<td>Convergent rounding</td>
<td>Wrap-around</td>
</tr>
<tr>
<td>Truncation</td>
<td>Sign-magnitude wrap-around</td>
</tr>
<tr>
<td>Truncation to Zero</td>
<td></td>
</tr>
</tbody>
</table>
User-Defined Data Types

- User-defined data types can be used for ports and signals

```cpp
sc_signal<complex> c;

class complex {
  private:
    double re;
    double im;
  public:
    complex () { re=0.0; im=0.0; }
    complex (double r, double i) { re=r; im=i; }
    void set(double r, double i) { re=r; im=i; }
    double get_re() { return re; }
    double get_im() { return im; }

    int operator==(const complex &c) const {
      if ( (re == c.re) && (im == c.im) )
        return 1;
      else
        return 0;
    }

};
```
Modules & Hierarchie

- Modules may contain sub-modules (→ hierarchical structure)

- In SC_MODULE:

  ```
  // sub-module declaration
  module_type *my_module;
  ```

- In the module constructor of SC_MODULE:

  ```
  // sub-module instantiation and port mapping
  SC_CTOR( module_name ) {
    my_module = new module_type ( "label");
    my_module -> in1 (sig1);
    my_module -> in2 (sig2);
    my_module -> out1 (sig3);
  }
  ```
Example:

\[ d = (a + b) - c \]

```
SC_MODULE( plus ) {
    sc_in<int> i1;
    sc_in<int> i2;
    sc_out<int> o1;
    ..... 
};
```

```
SC_MODULE( minus ) {
    sc_in<int> i1;
    sc_in<int> i2;
    sc_out<int> o1;
    ..... 
};
```

```
SC_MODULE( alu ) {
    sc_in<int> a;
    sc_in<int> b;
    sc_in<int> c;
    sc_out<int> d;
    sc_signal<int> e;

    plus *p;
    minus *m;

    sc_signal<int> e;

    SC_CTOR( alu ) {
        p = new plus ("PLUS");
        p->i1 (a);
        p->i2 (b);
        p->o1 (e);

        m = new minus ("MINUS");
        (*m) (e,c,d);
    }
};
```
Processes

- **Process Semantics**
  - Encapsulates functionality
  - Basic unit of concurrent execution
  - Not hierarchical

- **Process Activation**
  - Processes have sensitivity lists
  - Processes are triggered by events on sensitive signals

- **Process Types**
  - Method (SC_METHOD)
    - asynchronous block, like a sequential function
  - Thread (SC_THREAD)
    - asynchronous process
  - Clocked Thread (SC_CTHREAD)
    - synchronous process
## Processes

<table>
<thead>
<tr>
<th>Triggered</th>
<th>SC_METHOD by signal events</th>
<th>SC_THREAD by signal events</th>
<th>SC_CTHREAD by clock edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infinite loop</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Execution suspend</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Suspend &amp; resume</td>
<td>-</td>
<td>wait()</td>
<td>wait() wait_until()</td>
</tr>
<tr>
<td>Construct &amp; sentisize method</td>
<td>SC_METHOD(p); sensitive(s); sensitive_pos(s); sensitive_neg(s);</td>
<td>SC_THREAD(p); sensitive(s); sensitive_pos(s); sensitive_neg(s);</td>
<td>SC_CTHREAD(p,clock.pos());</td>
</tr>
<tr>
<td>Modeling example (hardware)</td>
<td>combinational logic</td>
<td>sequential logic at RT level (asynchronous reset, etc.)</td>
<td>sequential logic at higher design levels</td>
</tr>
</tbody>
</table>
Processes

- **Declaration of member function** (in `SC_MODULE`)

  ```
  // process declaration
  void my_process();
  ```

- **Instantiation** (in module constructor of `SC_MODULE`)

  ```
  // specification of process type and sensitivity
  SC_CTOR(module_name) {
      SC_METHOD(my_process);
      sensitive << sig1 << sig2;
  }
  ```

- **Definition of member function**
  (in `SC_MODULE` or somewhere else)

  ```
  // process specification
  void module_name::my_process() {
      ....
  }
  ```
Processes

- Example: **SC_METHOD**

```c
SC_MODULE( plus ) {
    sc_in<int> i1;
    sc_in<int> i2;
    sc_out<int> o1;

    void do_plus();

    SC_CTOR( plus ) {
        SC_METHOD( do_plus );
        sensitive << i1 << i2;
    }
};
```

```c
void plus::do_plus() {
    int arg1;
    int arg2;
    int sum;
    
    arg1 = i1.read();
    arg2 = i2.read();
    sum = arg1 + arg2;
    o1.write(sum);
}
```

```c
void plus::do_plus() {
    o1 = i1 + i2;
}
```
Example: **SC_THREAD**

```c
SC_MODULE( plus ) {
    sc_in<int> i1;
    sc_in<int> i2;
    sc_out<int> o1;

    void do_plus();

    SC_CTOR( plus ) {
        SC_THREAD( do_plus );
        sensitive << i1 << i2;
    }
};

void plus::do_plus() {
    int arg1;
    int arg2;
    int sum;

    while ( true ) {
        arg1 = i1.read();
        arg2 = i2.read();
        sum = arg1 + arg2;
        o1.write(sum);
        wait();
    }
}
```
Processes

Example: **SC_CTHREAD**

```c
SC_MODULE( plus ) {
    sc_in_clk     clk;
    sc_in<int>    i1;
    sc_in<int>    i2;
    sc_out<int>   o1;
    void do_plus();

    SC_CTOR( plus ) {
        SC_CTHREAD( do_plus, clk.pos() );
    }

    void do_plus() {
        int arg1;
        int arg2;
        int sum;

        while ( true ) {
            arg1 = i1.read();
            arg2 = i2.read();
            sum = arg1 + arg2;
            o1.write(sum);
        }
    }
};
```
Waiting and Watching

- **Suspend / reactivate process execution** \( (SC\_THREAD, SC\_CTHREAD) \)
  - Suspension: \( \text{wait()} \)
  - Reactivation: event on a sensitive signal

- **Halt process execution until an event occurs** \( (SC\_CTHREAD \text{ only}) \)
  - \( \text{wait\_until} \ (\text{my\_bool\_sig}.\text{delayed()} == \text{true}) \)

- **Transfer control to a special code sequence if a specified condition occurs**
  - \( \text{watching} \ (\text{reset}.\text{delayed()} == \text{true}) \)
  - Typical example: watching for reset signal

  - **Global watching:** \( (SC\_THREAD, SC\_CTHREAD) \)
    - watching condition specified in the module constructor
    - control is transferred to the beginning of the process (to be handled there)

  - **Local watching:** \( (SC\_CTHREAD \text{ only}) \)
    - allows to specify the process region to be watched
    - using macros \( \text{W\_BEGIN}, \text{W\_DO}, \text{W\_ESCAPE}, \text{W\_END} \)
    - can be nested and combined with global watching
Cycle-Accurate Simulation Scheduler

Step 1: All clock signals that change their value at the current time are assigned their new value.

Step 2: All `SC_METHOD`/`SC_THREAD` processes with inputs that have changed are executed. The entire bodies of `SC_METHOD` processes are executed. `SC_THREAD` processes are executed until the next `wait()` statement suspends execution. `SC_METHOD`/`SC_THREAD` processes are not executed in a fixed order.

Step 3: All `SC_CTHREAD` processes that are triggered have their outputs updated and are saved in a queue to be executed in step 5. All outputs of `SC_METHOD`/`SC_THREAD` processes that were executed in step 1 are also updated.

Step 4: Step 2 and step 3 are repeated until no signal changes its value.

Step 5: All `SC_CTHREAD` processes that were triggered and queued in step 3 are executed. There is no fixed execution order of these processes. Their outputs are updated at the next active edge (when step 3 is executed), and therefore are saved internally.

Step 6: Simulation time is advanced to the next clock edge and the scheduler goes back to step 1.
Design Example A
SystemC: Example-1

Two processes (process_1 and process_2) alternately incrementing an integer value
Source Code File Structure

- systemc.h
- process_1.h
- process_2.h
- process_1.cc
- process_2.cc
- main.cc

**g++**

**a.out**

**executable = simulation**
Module: process_1

// header file: process_1.h
SC_MODULE( process_1 ) {  
    // Ports
    sc_in_clk clk;
    sc_in<int> a;
    sc_in<bool> ready_a;
    sc_out<int> b;
    sc_out<bool> ready_b;
    // Process functionality
    void do_process_1();
    // Constructor
    SC_CTOR( process_1 )  {
        SC_CTHREAD( do_process_1 , clk.ps() );
    }
};

// implementation file: process_1.cc
#include "systemc.h"
#include "process_1.h"
void process_1::do_process_1()  {
    int v;
    while ( true )  {
        wait_until( ready_a.delayed() == true );
        v = a.read();
        v += 5;
        cout << "P1: v = " << v << endl;
        b.write( v );
        ready_b.write( true );
        wait();
        ready_b.write( false );
    }
}
Module: process_2

// header file: process_2.h
SC_MODULE( process_2 ) {
    // Ports
    sc_in_clk clk;
    sc_in<int> a;
    sc_in<bool> ready_a;
    sc_out<int> b;
    sc_out<bool> ready_b;
    // Process functionality
    void do_process_2();
    // Constructor
    SC_CTOR( process_2 ) {
        SC_CTHREAD( do_process_2 , clk.ps() );
    }
};

// implementation file: process_2.cc
#include "systemc.h"
#include "process_2.h"
void process_2::do_process_2()
{
    int v;
    while ( true )
    {
        wait_until( ready_a.delayed() == true );
        v = a.read();
        v += 3;
        cout << "P2: v = " << v << endl;
        b.write( v );
        ready_b.write( true );
        wait();
        ready_b.write( false );
    }
}
Top-Level Module: main

// implementation file: main.cc

#include "systemc.h"
#include "process_1.h"
#include "process_2.h"

int sc_main (int ac,char *av[])
{
    sc_signal<int> s1 ( "Signal-1" );
    sc_signal<int> s2 ( "Signal-2" );
    sc_signal<bool> ready_s1 ( "Ready-1" );
    sc_signal<bool> ready_s2 ( "Ready-2" );

    sc_clock clock( "Clock" , 20 , 0.5 , 0.0 );

    process_1 p1 ( "P1" );
    p1.clk( clock );
    p1.a( s1 );
    p1.ready_a( ready_s1 );
    p1.b( s2 );
    p1.ready_b( ready_s2 );

    process_2 p2 ( "P2" );
    p2.clk( clock );
    p2.a( s2 );
    p2.ready_a( ready_s2 );
    p2.b( s1 );
    p2.ready_b( ready_s1 );

    s1.write(0);
    s2.write(0);
    ready_s1.write(true);
    ready_s2.write(false);

    sc_start(100000);

    return 0;
}
Simulation Results

- Simulation output

SystemC (TM) Version 1.0  --- Apr 4 2000 10:12:32
ALL RIGHTS RESERVED
Copyright (c) 1988-2000 by Synopsys, Inc.

Simulation speed:

Simulation of 100,000 cycles takes about 0.08 seconds on a Sun Ultra Sparc 5 (384 MByte main memory)

Comparison to simulation speed of SystemC 0.9:

Simulation of 100,000 cycles takes about 0.31 seconds on the same machine
Design Example B
Background:

- Transforming HDL into SystemC
- Creating new designs in SystemC

Transforming C/C++ into SystemC
- Many algorithms exist in C/C++
- Many standardization committees (e.g., ISO) use C specifications
SystemC Design Example

- **Application:**
  JPEG compression and decompression stream

- **Reference implementation:**
  - 16 modules, approx. 950 lines of C++ code
  - by T. Thissenhusen, TU Dresden, Germany
SystemC Design Example

Reader

Encoder

Decoder

Writer

input pgm-file start

DCT

Quant

ZigZag

RLEH

IRLEH

lZigZag

lQuant

IDCT

output pgm-file

joachim gerlach
SystemC Design Example

```
struct Block {
    char data[8][8];
    ....
};

struct Coeff_8x8 {
    short data[8][8];
    ....
};

struct Matrix_64x12 {
    short data[64];
    ....
};
```

Encoder

DCT

Quant

ZigZag

RLEH

```
Encoder

data_in

start

ready

ZigZag

Coeff_8x8

bool

bool

clk

Matrix_64x12

bool

data_out

data_out

_ready

data_ok

char data;
```
SystemC Design Example

Reader

Encoder

Decoder

Writer

input pgm-file
start

output pgm-file

DCT

Quant

ZigZag

RLEH

IRLEH

IQuant

IDCT

I ZigZag

SystemC 1.0

Design Example A

SystemC 1.1

Design Example B

Tool Support

Background & Basics

System-on-Chip Design with SystemC
Design Activities
Design Activities

- **Modeling**
  - Module for “zigzag” computation

- **Simulation**
  - Generation and run of an executable specification

- **Debugging**
  - Techniques for checking the functionality of the system
```c
#include <systemc.h>
#include "global.h"

SC_MODULE(zigzag) {
  sc_in_clk                         clk;
  sc_in<Coeff_8x8>          data_in;
  sc_in<bool>                    start ;
  sc_in<bool>                    data_ok;
  sc_out<Matrix_64x12>   data_out;
  sc_out<bool>                  ready ;
  sc_out<bool>                  data_out_ready;

  void do_zigzag();

  SC_CTOR(zigzag) {
    SC_CTHREAD( do_zigzag,clk.pos());
  }
};

void zigzag::do_zigzag()  {
  Coeff_8x8     fuv;
  Matrix_64x12   result;
  unsigned char    u, v, a, dir;

  while(true) {
    ready.write(true);
    data_out_ready.write(false);
    wait_until(start.delayed()==true);
    ready.write(false);
    fuv = data_in.read();

    // zigzag
    u = 0; v = 0;
    dir = 1;  // dir == 1: upwards, dir == 0: downwards
    for ( a = 0; a < 64; a++ )  {
      result.put ( a, (WORD) (fuv.get (v,u) ) );
      if ( v == 0 )
        if ( dir )   { u++; dir = 0; }
        else        { u--; v++; }
      else if ( v == 7 )
        if ( !dir )  { u++; dir = 1; }
        else        { u++; v--; }
      else if ( u == 0 )
        if ( !dir )  { v++; dir = 1; }
        else        { u++; v--; }
      else if ( u == 7 )
        if ( dir )   { v++; dir = 0; }
        else        { u--; v++; }
      else
        if ( dir )   { u++; v--; }
        else        { u--; v++; }
    }
    data_out.write(result);
    data_out_ready.write(true);
    wait_until(data_ok.delayed()==true);
  }
};
```
Generation of an executable specification

- systemc.h
- reader.h
- writer.h
- reader.cc
- writer.cc
- jpeg.cc
- g++
- run

executable = simulator
Simulation

- Simulation control
  - Simulation start:
    \[ \text{sc\_start()} / \text{sc\_start}(n) \] from the top-level function \text{sc\_main()}
  - Simulation stop:
    \[ \text{sc\_stop()} \] from within any process

- Advanced simulation control:
  "self-made" clock by \text{sc\_initialize()} and \text{sc\_cycle}(n)

```c
sc\_clock my\_clock ("CLK", 20, 0.5);
sc\_start(200);
sc\_initialize();
for (int i=0; i<=200; i++) {
  clock = 1;
  sc\_cycle(10);
  clock = 0;
  sc\_cycle(10);
}
```

true
false

[Graph showing clock waveform]
Simulation

- Running the executable specification

> ./run motorbike.pgm motorbike2.pgm

SystemC (TM) Version 1.0  --- May 22 2000 14:21:01
ALL RIGHTS RESERVED
Copyright (c) 1988-2000 by Synopsys, Inc.
SystemC: simulation stopped by user.
> 
>
Simulation

- Results of an executable run

input picture (motorbike.pgm) → compression decompression stream → output picture (motorbike2.pgm)

Reader → JPEG Encoder → JPEG Decoder → Writer

Background & Basics

SystemC 1.0
Design Example A
Design Example B

Design Activities
Tool Support
SystemC 1.1

Joachim Gerlach
University of Tübingen
Department of Computer Engineering

Joachim Gerlach
System-on-Chip Design with SystemC
Debugging

- Observation of simulation results ✓
- Adding (C/C++) assertions/debug outputs to the source code
- Using SystemC debugging features
- Using standard debugging tools (gdb, Purify, ...)
Debugging

- Adding debug outputs to the source code

```cpp
void zigzag::do_zigzag() {
    ......  
    static int no_of_zigzags = 0;
    while(true) {
        ......  
        // zigzag computation
        ......  
        no_of_zigzags++;
        cout << "number of zigzags:"
        << no_of_zigzags << endl;
        ......  
    }
}
```
Debugging

- Running the executable specification

```
> ./run motorbike.pgm motorbike2.pgm

SystemC (TM) Version 1.0 --- May 22 2000 14:21:01
ALL RIGHTS RESERVED
Copyright (c) 1988-2000 by Synopsys, Inc.
number of zigzags: 1
number of zigzags: 2
number of zigzags: 3
number of zigzags: 4
.....
number of zigzags: 1099
number of zigzags: 1100
number of zigzags: 1101
number of zigzags: 1102
SystemC: simulation stopped by user.
> 
> 
```
Using SystemC debugging features: `sc_time_stamp()`

```c
void zigzag::do_zigzag() {
    // zigzag computation
    no_of_zigzags++;
    cout << "cycle: " << sc_time_stamp();
    cout << " - number of zigzags: " << no_of_zigzags << endl;
}
```
Running the executable specification

> ./run motorbike.pgm motorbike2.pgm

SystemC (TM) Version 1.0 --- May 22 2000 14:21:01
ALL RIGHTS RESERVED
Copyright (c) 1988-2000 by Synopsys, Inc.
cycle: 5 - number of zigzags: 1
cycle: 8 - number of zigzags: 2
cycle: 11 - number of zigzags: 3
cycle: 14 - number of zigzags: 4
.....
cycle: 3299 - number of zigzags: 1099
cycle: 3302 - number of zigzags: 1100
cycle: 3305 - number of zigzags: 1101
cycle: 3308 - number of zigzags: 1102
SystemC: simulation stopped by user.
>
Debugging

- Using SystemC debugging features: waveform tracing

```c
int sc_main( int argc, char *argv[] )
{
    ..... 

    // waveform tracing
    if ( (argc == 4) && (strcmp(argv[3],"w") == 0) ) {
        sc_trace_file* tf = sc_create_wif_trace_file( "wave" );
        sc_trace( tf, clk.signal(), "clock" );
        sc_trace( tf, encoder_is_ready, "encoder_is_ready" );
        sc_trace( tf, orig_data_ready, "start_dct" );
        sc_trace( tf, e.data_out_ready_1, "start_quant" );
        sc_trace( tf, e.data_out_ready_2, "start_zigzag" );
        sc_trace( tf, e.data_out_ready_3, "start_rleh" );
        sc_trace( tf, decoder_is_ready, "decoder_is_ready" );
        sc_trace( tf, comp_data_ready, "start_idct" );
        sc_trace( tf, d.data_out_ready_1, "start_iquant" );
        sc_trace( tf, d.data_out_ready_2, "start_izigzag" );
        sc_trace( tf, d.data_out_ready_3, "start_irleh" );
    }
    ..... 
}
```
Debugging

- Running the executable specification (with “w” parameter)

> ./run motorbike.pgm motorbike2.pgm w

SystemC (TM) Version 1.0 --- May 22 2000 14:21:01
ALL RIGHTS RESERVED
Copyright (c) 1988-2000 by Synopsys, Inc.
WARNING: Default time step (1 s) is used for WIF tracing.
SystemC: simulation stopped by user.
>
> viewer wave.awif &

![Waveform viewer screenshot]
Tool Support
Synopsys SystemC Compiler

- SystemC for system modeling

SystemC

C/C++ Testbench → C/C++ Hardware/System → Standard C++ Compiler → Executable = Simulator != Debugger

C/C++ Software

Modeling Constructs
SystemC Compiler

- SystemC Compiler C++ synthesis in the HW flow

Model using SystemC

SystemC™ Compiler

Behavioral synthesis

- db form
- db or HDL format

RTL synthesis

- db form
  (This flow is not supported currently)

Gate level netlist
Refinement for implementation

- **System Specification**
  - **Refine Structure**
    - Partition into blocks that will be independently synthesized/refined
    - Refine interfaces for communication
  - **Refine Control**
    - Specify I/O protocol
    - Specify clock domains
    - Specify latency, throughput
    - Specify FSM & datapath for RTL
  - **Refine Data**
    - Use bit-true types
    - Select appropriate bit widths

---

Joachim Gerlach

System-on-Chip Design with SystemC
Synopsys SystemC Compiler

- Behavioral Level Flow

- Timed DB Generation
- High-Level Synthesis

- SystemC Compiler
- Initial Constraints
- Check Design
- Time/Area Estimates
- Timed DB File

- Code?
- Remove Design
- OK?
- SystemC View Reports
- Schedule
- Latency/Pipeline Constraints

- Cycle-Accurate HDL
- HDL Co-Simulation
- Compile
- Gate Level Netlist

- Happy code?
- no
- yes

- SystemC 1.0
- Design Example A
- Design Example B
- Design Activities
- Test Support
- SystemC 1.1

University of Tübingen
Department of Computer Engineering

Wolfgang Rosenstiel
Synopsys SystemC Compiler

- **Hardware Implementation Flow**

  - **Refinement** (communication, timing, memories)
  - **Refinement** (resources, scheduling, allocation, FSM design)

  - **Functional Design**
  - **Architectural Design**
  - **RT Level Design**
  - **Gate Level Design**

  - **IQ Block**
  - **SystemC Compiler Behavioral Flow**
  - **Controller**
  - **SystemC Compiler RTL Flow**
Synopsys SystemC Compiler

- **Benefits**
  - **Rapid time to market**
    - fast refinement from functional model behavioral model
    - accommodating late spec changes
  - **Graphical analysis of design**
  - **High quality of results**
    - tight integration into Synopsys synthesis flow
    - flexibility for datapath components
CoWare N2C

- Gaps in System Design

- Ambiguous Specification
  - the “restart” problem

- the “Implementation” gaps

- C, C++
  - Assembly code
  - Machine code

- VHDL, Verilog
  - Netlist & cell library
  - Layout

- the “co-design” gap

- the “IP Expertise” gaps

- Target IP core

Joachim Gerlach

System-on-Chip Design with SystemC
System-Level Design with CoWare N2C

- CoWare N2C

- System-Level Design with CoWare N2C

- SystemC 1.0

- Design Example A

- Design Example B

- Activities

- Tool Support

- SystemC 1.1

- Behavioral C

- System Design and Partitioning

- SystemC Executable Implementable Spec

- Cycle-Accurate C

- HW-SW Co-design and Multi-level Co-verification

- Testbench

- HW-SW Co-verification

- "Traditional" HW-SW Co-Verification

- RTL Implementation

- IP and Performance Models

- HW Design

- Generate HDL

- SystemC

- Synthesis

- Interface

- SW Optimization

- Testbench

- Refine

- Refine

- Function

- Architecture

Joachim Gerlach

System-on-Chip Design with SystemC
CoWare N2C

- N2C Workbench
  - Hierarchical design browser (architecture, functionality)
  - Source code editor (context sensitive)
  - Project manager (partition management)

- CoWare Support
  - CoWare N2C supports a top-down design flow for HW/SW co-design from UTF to RTL
  - Co-simulation of different languages
  - Fast design exploration and HW/SW partitioning
  - Allows for efficient IP reuse and delivery
  - Provides synthesis of communication
System Compiler (C Level Design)

- **System Compiler**
  - supports full ANSI C and C++
  - provides complex data structures, static pointer analysis, abstraction, hierarchy
  - output is RT level HDL (VHDL or Verilog)

- **CSim**
  - executable specification
  - discrete event simulation
  - abstraction: temporal time, data values, functionality
A|RT Builder (Frontier Design)

system specification
embedded software

datapath resources
(arithmetic, memory)

edit/compile
create architecture

map to architecture

source code
tuning

architecture
optimization

logicsynthesis

FPGA

ASIC
SystemC 1.1 Outlook
**SystemC 1.1 Design Flow**

- **Matlab**, **C++**, **SDL**, **Esterel**, **......**

**SystemC 1.1**
- **UTF**: untimed functional
  - design exploration
  - performance analysis
  - hw/sw partitioning

**TF**: timed functional
- hw/sw partitioning
- refine communication

**Abstr. RTOS**
- task partitioning

**RTOS**
- target RTOS/core

**BCA**
- bus cycle accurate
- refine behavior

**RTL**
- cycle accurate
- software

**hardware**
- refinements
Design Levels in SystemC 1.1

- **UTF: UnTimed Functional**
  - Functional decomposition of a system
  - Architecture, timing, inter-block communication is abstracted
  - Maximally sequential form by RPC (Remote Procedure Call)
  - RPC: abstract (master/slave-)ports, multi-point link objects

- **TF: Timed Functional**
  - RPC also, but processes may be assigned a run time

- **BCA: Bus Cycle Accurate**
  - Abstract ports refined to bus ports with data, address, control terminals and communication protocols

- **CA: Cycle Accurate**
  - SystemC 1.0 level