System Modeling & HW/SW Co-Verification

Prof. Chien-Nan Liu
TEL: 03-4227151 ext:4534
Email: jimmy@ee.ncu.edu.tw
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Data-Types
- Processes
- Interfaces
- Simulation Supports
- System Design Environments
- HW/SW Co-Verification
- Conclusion
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Data-Types
- Processes
- Interfaces
- Simulation Supports
- System Design Environments
- HW/SW Co-Verification
- Conclusion
Design Challenges

Reference http://www.doulos.com
Silicon complexity v.s Software complexity

Silicon complexity is growing 10x every 6 years

Software in systems is growing faster than 10x every 6 years

Reference http://www.doulos.com
Increasing Complexity in SoC Designs

- One or more processors
- 32-bit Microcontrollers
- DSPs or specialized media processors
- On-chip memory
- Special function blocks
- Peripheral control devices
- Complex on-chip communications network (On-chip busses)
- RTOS and embedded software which are layering architecture
- ....
How to Conquer the Complexity?

- **Modeling strategies**
  - Using the appropriate modeling for different levels
  - The consistency and accuracy of the model
- **Reuse existing designs**
  - IP reuse
  - Architecture reuse (A platform based design)
- **Partition**
  - Based on functionality
  - Hardware and software
Traditional System Design Flow (1/2)

- Designers partition the system into hardware and software early in the flow.
- HW and SW engineers design their respective components in isolation.
- HW and SW engineers do not talk to each other.
- The system may not be the suitable solution.
- Integration problems.
- High cost and long iteration.
Traditional System Design Flow (2/2)

- System Level Design
  - Hardware and Software
  - Algorithm Development
  - Processor Selection
  - Done mainly in C/C++

C/C++ Environment

- IC Development
  - Hardware
  - Implementation
  - Decisions
  - Done mainly in HDL

EDA Environment

Verification Process

- Software Design
  - Code Development
  - RTOS details
  - Done mainly in C/C++

C/C++ Environment

Reference: Synopsys
Typical Project Schedule

- System Design
- Hardware Design
- Prototype Build
- Hardware Debug
- Software Design
- Software Coding
- Software Debug
- Project Complete

Reference: Mentor Graphic
Former Front-End Design Flow

Reference: DAC 2002 SystemC Tutorial
Problems with the Design Flow

- C/C++ System Level Model
- Analysis
- Results
- Convert by Hand
- Synthesis
- Not reusable
- Not done by designers
- The netlist is not preserved

Reference: DAC 2002 SystemC Tutorial
Shortcoming of Current System Design Flow

- Use natural language to describe the system specification
  - Cannot verify the desired functions directly
- Require many experts in system architecture for the partition of software and hardware parts
  - The partition may not be the optimal solution
- Hardware designers have to restart the design process by capturing the designs using the HDLs
  - May have unmatched problems
- Hardware and software integration is often painful
  - Hardware and software cannot work together
  - Co-verification of hardware and software is inefficient
Concurrent HW/SW Design

- Can provide a significant performance improvement for embedded system design
  - Allows earlier architecture closure
  - Reduce risk by 80%
- Allows HW/SW engineering groups to talk together
- Allows earlier HW/SW Integration
- Reduce design cycle
  - Develop HW/SW in parallel
  - 100x faster than RTL
Project Schedule with HW/SW Co-design

System Design → Hardware Design → Prototype Build → Hardware Debug

Software Design → Software Coding → Software Debug

Reference: Mentor Graphic

Project Complete
Modern System Design Flow

1. Specification of the System
2. System Level Modeling
3. Hardware and Software Partitioning
4. Architectural Exploration
   - H/W Model
   - S/W Model
5. Integration and Verification
   - H/W Design Flow
   - S/W Development
Outline

- Introduction
- **System Modeling Languages**
  - SystemC Overview
  - Data-Types
  - Processes
  - Interfaces
  - Simulation Supports
  - System Design Environments
  - HW/SW Co-Verification
- Conclusion
**Motivation to Use a Modeling Language**

- The increasing system design complexity
- The demand of higher level abstraction and modeling
- Traditional HDLs (verilog, VHDL, etc) are suitable for system level design
  - Lack of software supports
- To enable an efficient system design flow
Requirements of a System Design Language

- Support system models at various levels of abstraction
- Incorporation of embedded software portions of a complex system
  - Both models and implementation-level code
- Creation of executable specifications of design intent
- Creation of executable platform models
  - Represent possible implementation architectures on which the design intent will be mapped
Requirements of a System Design Language

- Fast simulation speed to enable design-space exploration
  - Both functional specification and architectural implementation alternatives
- Constructs allowing the separation of system function from system communications
  - In order to allow flexible adaptation and reuse of both models and implementation
- Based on a well-established programming language
  - In order to capitalize on the extensive infrastructure of capture, compilation, and debugging tools already available
Model Accuracy Requirements

- Structural accuracy
- Timing accuracy
- Functional accuracy
- Data organization accuracy
- Communication protocol accuracy
System Level Language

- SystemC
- Cynlib
- SoC++
- Handel-C
- A/RT (Library)
- VHDL+
- System Verilog
- SDL
- SLDL
- SUPERLOG
- Java
- C/C++ Based
- VHDL/Verilog Replacements
- Higher-level Languages
- Entirely New Language
- Java-Based
## Language Use

<table>
<thead>
<tr>
<th>Language Use</th>
<th>C/C++</th>
<th>SystemC 2.0</th>
<th>TestBuilder, OpenVer,e</th>
<th>Verilog VHDL</th>
<th>SUPERLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded SW</td>
<td>Good</td>
<td>Very Good</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>System Level Design</td>
<td>OK</td>
<td>Excel</td>
<td>NO</td>
<td>Very Poor</td>
<td>Good</td>
</tr>
<tr>
<td>Verification</td>
<td>OK</td>
<td>Good</td>
<td>Excel</td>
<td>OK</td>
<td>Very Good</td>
</tr>
<tr>
<td>RTL Design</td>
<td>NO</td>
<td>Good</td>
<td>NO</td>
<td>Excel</td>
<td>Excel</td>
</tr>
</tbody>
</table>

Reference: DAC 2002 SystemC Tutorial
Trend of System-Level Languages

- **Extend existing design languages** (ex: SystemVerilog)
  - Pros:
    - Familiar languages and environments to designers
    - Allow descriptions of prior version of Verilog
  - Cons:
    - Not standardized yet
    - Become more and more complex to learn

- **Standard C/C++ based languages** (ex: SystemC)
  - Pros:
    - Suitable for very abstract descriptions
    - Suitable to be an executable specification
  - Cons:
    - A new language to learn
    - Need solutions for the gap to traditional design flow
Evolution of Verilog Language

- Proprietary design description language developed by Gateway, 1990
  - Donated to OVI (Open Verilog International) by Cadence
- Verilog Hardware Description Language LRM by OVI, V2.0 1993
- IEEE Std. 1364-1995, “Verilog 1.0” (called Verilog-1995)
- Synopsys proposed Verilog-2000, including synthesizable subset
- IEEE Std. 1364-2001, “Verilog 2.0” (called Verilog-2001) (1st main enhancement)
- SystemVerilog 3.0, approved as an Accellera standard in June 2002
  - add system-level architectural modeling
- SystemVerilog 3.1, approved as an Accellera standard in May, 2003
  - add verification and C language integration (Not yet as standard)
- Verilog Standards Group (IEEE 1364) announced a project authorization request for 1364-2005
Compatibility of SystemVerilog

- ANSI C-style ports, named parameter passing, comma-separated sensitivity lists and attributes
- SystemVerilog 3.0/3.1
- Verilog-1995
- earliest Verilog
- initialization of variables, the semantics of "posedge" and "negedge" constructs, record-like constructs, handling of interfaces and various keywords
Enhancements in SystemVerilog

- C data types
- Interfaces to encapsulate
- Dynamic processes
- A unique top level hierarchy ($root)
- Verification functionality
- Synchronization
- Classes
- Dynamic memory
- Assertion mechanism
- …..
Summary about SystemVerilog

- More extension in high-level abstraction to the Verilog-2001 standard
  - Still no much enhancement in transaction-level abstraction
- Improves the productivity and readability of Verilog code
- Provide more concise hardware descriptions
- Extends the verification aspects of Verilog by incorporating the capabilities of assertions
  - Still no coverage construct within testbench design
- 3.0/3.1 LRM are still not very clear in more details
- Not yet simulator support
  - No compiler for trying its syntax
- SV is a valuable direction to be watched
  - Will it become too complex for most designers/verification engineers’ requirement/understanding??
Reference for SystemVerilog

- **SystemVerilog 3.1, ballot draft: Accellera's Extensions to Verilog?** Accellera, Napa, California, April 2003.
- **An overview of SystemVerilog 3.1, By Stuart Sutherland, EEdesign, May 21, 2003**
  

**URL:**

1)  [http://www.eda.org/sv-ec/](http://www.eda.org/sv-ec/)  (SystemVerilog Testbench Extension Committee)

2)  [http://www.eda.org/sv-ec/SV_3.1/Web/index.html](http://www.eda.org/sv-ec/SV_3.1/Web/index.html)  (SV3.1 Web)
Why C/C++ Based Language for System Modeling

- Specification between architects and implementers is executable
- High simulation speed due to the higher level of abstraction
- Refinement, no translation into HDL (no "semantic gap")
- Testbench re-use
Advantages of Executable Specifications

- Ensure the **completeness** of specification
  - Even components (e.g. Peripherals) are so complex
  - Create a program that behave the same way as the system

- Avoid **ambiguous** interpretation of the specification
  - Avoids unspecified parts and inconsistencies
  - IP customer can evaluate the functionality up-front

- Validate system functionality before implementation
  - Create early model and validate system performance

- Refine and test the implementation of the specification
  - Test automation improves Time-to-Market
Can Traditional C++ Standard Be Used?

- C++ does not support
  - Hardware style communication
    - Signals, protocols, etc
  - Notion of time
    - Time sequenced operations
  - Concurrency
    - Hardware and systems are inherently concurrent
  - Reactivity
    - Hardware is inherently reactive, it responds to stimuli and is inconstant interaction with its environments
  - Hardware data types
    - Bit type, bit-vector type, multi-valued logic type, signed and unsigned integer types and fixed-point types
SystemC v.s SpecC

- Constructs to model system architecture
  - Hardware timing
  - Concurrency
  - Hardware data-type (signal, etc)

- Adding these constructs to C/C++
  - SystemC
    - C++ Class library
    - Standard C++ Compiler: bcc, msvc, gcc, etc
  - SpecC
    - Language extension: New keywords and syntax
    - Translator for C
SystemC is…

- A library of C++ classes
  - Processes (for concurrency)
  - Clocks (for time)
  - Hardware data types (bit vectors, 4-valued logic, fixed-point types, arbitrary precision integers)
  - Waiting and watching (for reactivity)
  - Modules, ports, signals (for hierarchy)
  - Abstract ports and protocols (abstract communications)
    - Using channel and interface classes
SystemC Design Flow

Reference: DAC 2002 SystemC Tutorial
Outline

- Introduction
- System Modeling Languages
- **SystemC Overview**
  - Data-Types
  - Processes
  - Interfaces
  - Simulation Supports
  - System Design Environments
  - HW/SW Co-Verification
- Conclusion
SystemC Language Architecture

SystemC Language Layering Architecture

- **Not-standard**
  - Standard Channels for Various Model of Computation
    - Kahn Process Networks
    - Static Dataflow, etc.
  - Methodology-Specific Channels
    - Master/Slave Library, etc.

- **Prepare to involve to SystemC Standard**
  - **Elementary Channels**
    - Signal, Timer, Mutex, Semaphore, FIFO, etc

- **Core Language**
  - Modules
  - Ports
  - Processes
  - Events
  - Interfaces
  - Channels
  - Event-Driven Simulation
  - Kernel

- **Data-Types**
  - 4-valued logic types (01XZ)
  - 4-valued logic-vectors
  - Bits and bit-vectors
  - Arbitrary-precision integers
  - Fixed-point numbers
  - C++ user-defined types

- **C++ Language Standard**
System Abstraction Level (1/3)

- Untimed Functional Level (UTF)
  - Refers to both the interface and functionality
  - Abstract communication channels
  - Processes executed in zero time but in order
  - Transport of data executed in zero time

- Timed Functional Level (TF)
  - Refers to both the interface and functionality
  - Processes are assigned an execution time
  - Transport of data is assigned a time
  - Latency modeled
  - “Timed” but not “clocked”
System Abstraction Level (2/3)

- **Bus Cycle Accurate (BCA)**
  - Transaction Level Model (TLM)
  - Model the communications between system modules using shared resources such as busses
  - Bus cycle accurate or transaction accurate
    - No pin-level details

- **Pin Cycle Accurate (PCA)**
  - Fully described by HW signals and the communications protocol
  - Pin-level details
  - Clocks used for timing
System Abstraction Level (3/3)

- Register Transfer Accurate
  - Fully timed
  - Clocks used for synchronization
  - Complete functional details
    - Every register for every cycle
    - Every bus for every cycle
    - Every bit described for every cycle
  - Ready to RTL HDL
Core Language

- **Time Model**
  - To define time unit and its resolution

- **Event-Driven Simulation Kernel**
  - To operate on events and switch between processes, without knowing what the events actually represent or what the processes do

- **Modules and Ports**
  - To represent structural information

- **Interfaces and Channels**
  - To describe the abstraction of communication between the design block
Time Model

- Using an integer-valued time model
- 64-bit unsigned integer
- Can be increased to more than 64 bits if necessary
- Same as in Verilog and VHDL
Time Model (cont’)

● Time resolution
  – Must be specified before any time objects (e.g. sc_time) are created
  – Default value is one pico-second \( 10^{-12} \) s

● Time unit

<table>
<thead>
<tr>
<th>SC_FS</th>
<th>femtosecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC_PS</td>
<td>picosecond</td>
</tr>
<tr>
<td>SC_NS</td>
<td>nanosecond</td>
</tr>
<tr>
<td>SC_US</td>
<td>microsecond</td>
</tr>
<tr>
<td>SC_MS</td>
<td>millisecond</td>
</tr>
<tr>
<td>SC_SEC</td>
<td>second</td>
</tr>
</tbody>
</table>

Example for 42 picosecond
sc_time T1(42, SC_PS)

Example for resolution
sc_set_time_resolution(10, SC_PS)
sc_time T2(3.1416, SC_NS)
T2 would be rounded to 3140 ps
**Modules**

- The basic building blocks for partitioning a design
- Declared with the SystemC keyword `SC_MODULE`
- Typically contain
  - Ports that communicate with the environment
  - Process that describe the functionality of the module
  - Internal data and communication channels for the model
  - Hierarchies (other modules)
- Modules can also access a channel’s interface directly
### Modules - Example

**Verilog**

```verilog
define module module name (
    ...
    ...
endmodule
```

**SystemC**

```systemc
SC_MODULE (module name) {
    ...
    ...
}
```

---

**SC_MODULE (FIFO) {**

//ports, process, internal data, etc

`sc_in<bool> load;`

`sc_in<bool> read;`

`sc_inout<int> data;`

`sc_out<bool> full;`

`sc_out<bool> empty;`

**SC_CTOR(FIFO){**

//body of constructor;

//process declaration, sensitivities, etc.

}**

```
Module Instantiation

<table>
<thead>
<tr>
<th>Verilog</th>
<th>SystemC</th>
</tr>
</thead>
</table>
| `include "adder.v"
... module name(...);
  input ...;
  output ...
  wire a1,b1,...;
  adder ul(a1,b1, ...);
...
endmodule | `include "systemc.h"
#include "adder.h"
...
int sc_main(int argc, char* argv[])
{
  sc_signal<bool> a1, b1, ...;
  adder ul( "ul" );
  ul <= a1 <= b1 <= ...;
...
  sc_start(-1);
  return(0);
} |

<table>
<thead>
<tr>
<th>Verilog</th>
<th>SystemC</th>
</tr>
</thead>
</table>
| Mux U1(Sel,a,b,Out);
Mux U1(.Out(Out),.a(a),.b(b),.Sel(Sel)); | Mux U1("U1");
U1<= Sel <= a <= b <= Out;
Mux U1("U1");
U1.Out(Out);
U1.a(a);
U1.b(b);
U1.Sel(Sel); |
Similar Control Flow Descriptions

Verilog
if(expression)
    statement1;
else
    statement2;
out=select?a:b;

SystemC
if(expression)
    statement1;
else
    statement2;
out=select?a:b;

Verilog
case (expression)
    item1 : state1 ;
    item2 : state2 ;
    default : state3 ;
endcase

SystemC
switch(expression)
{
    case item1 : state1 ; break;
    case item2 : state2 ; break;
    default : state3 ; break;
}

Verilog
integer i;
for(i=0;i<5;i=i+1)
begin
    statement;
end

SystemC
for(int i=0;i<5;i=i+1)
{
    statement;
}
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- **Data-Types**
- Processes
- Interfaces
- Simulation Supports
- System Design Environments
- HW/SW Co-Verification
- Conclusion
Data-Types

- SystemC allows users to use any C++ data types as well as unique SystemC data types
  - `sc_bit` – 2 value single bit type
  - `sc_logic` – 4 value single bit type
  - `sc_int` – 1 to 64 bit signed integer type
  - `sc_uint` – 1 to 64 bit unsigned integer type
  - `sc_bigint` – arbitrary sized signed integer type
  - `sc_biguint` – arbitrary sized unsigned integer type
  - `sc_bv` – arbitrary sized 2 value vector type
  - `sc_lv` – arbitrary sized 4 value vector type
  - `sc_fixed` - templated signed fixed point type
  - `sc_ufixed` - templated unsigned fixed point type
  - `sc_fix` - untemplated signed fixed point type
  - `sc_ufix` - untemplated unsigned fixed point type
**Type sc_bit**

- Type `sc_bit` is a two-valued data type representing a single bit
- Value ‘0’ = false
- Value ‘1’ = true

<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_bit* operators

For Example:
```
sc_bit a,b; //Declaration
a = a & b;
a = a | b
```
**Type sc_logic**

- The `sc_logic` has 4 values, ’0’ (false), ’1’ (true), ’X’ (unknown), and ’Z’ (high impedance or floating).
- This type can be used to model designs with multi-driver busses, X propagation, startup values, and floating busses.
- The most common type in RTL simulation.

<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_logic* operators

For Example

```
sc_logic x; // object declaration
x = '1'; // assign a 1 value
x = 'Z'; // assign a Z value
```
Fixed Precision Unsigned and Signed Integers

- The C++ int type is machine dependent, but usually 32 bits
- SystemC integer type provides integers from 1 to 64 bits in signed and unsigned forms
- sc_int\(<n>\)
  - A Fixed Precision Signed Integer
  - 2’s complement notation
- sc_uint\(<n>\)
  - A Fixed Precision Unsigned Integer
### The Operators of `sc_int<n>` and `sc_uint<n>`

<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>=, +=, -=, *=, /=, %=, &amp;=,</td>
</tr>
<tr>
<td>Equality</td>
<td>==, !=</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>++</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>--</td>
</tr>
<tr>
<td>Bit Select</td>
<td>[x]</td>
</tr>
<tr>
<td>Part Select</td>
<td>range()</td>
</tr>
<tr>
<td>Concatenation</td>
<td>(,)</td>
</tr>
</tbody>
</table>
# Bit Select and Part Select

<table>
<thead>
<tr>
<th>Verilog</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>input [3:0] A;</td>
<td>sc_in&lt;sc_uint&lt;4&gt; &gt; A;</td>
</tr>
<tr>
<td>output C;</td>
<td>sc_out&lt;bool&gt; C;</td>
</tr>
<tr>
<td>...</td>
<td>sc_uint&lt;4&gt; B;</td>
</tr>
<tr>
<td>C=A[0];</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>B=A.read();</td>
</tr>
<tr>
<td></td>
<td>C=B[0];</td>
</tr>
<tr>
<td>reg [7:0] d;</td>
<td>sc_bv&lt;8&gt; d;</td>
</tr>
<tr>
<td>reg [3:0] e;</td>
<td>sc_bv&lt;4&gt; e;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>e=d[5:2];</td>
<td>e=d.range(5,2);</td>
</tr>
</tbody>
</table>
The Examples of sc_int\(<n>\) and sc_uint\(<n>\)

```
sc_int<64> x; // declaration example
sc_uint<48> y; // declaration example

sc_int<16> x, y, z;
z = x & y; // perform and operation on x and y bit
// by bit
z = x >> 4; // assign x shifted right by 4 bits to z

To select on bit of an integer using the bit select operator

sc_logic mybit;
sc_uint<8> myint;
mybit = myint[7];

To select more than one bit using the range method

sc_uint<4> myrange;
sc_uint<32> myint;
myrange = myint.range(7,4);

Concatenation operation

sc_uint<4> inta;
sc_uint<4> intb;
sc_uint<8> intc;
intc = (inta, intb);
```
Arbitrary Precision Signed and Unsigned Integer Types

- For the cases that some operands have to be larger than 64 bits, the `sc_int` and `sc_unint` will not work.
- The `sc_biguint` (arbitrary size unsigned integer) or `sc_bigint` (arbitrary sized signed integer) can solve this problem.
- These types allow the designer to work on integers of any size, limited only by underlying system limitations.
- Arithmetic and other operators also use arbitrary precision when performing operations.
- These types execute more slowly than their fixed precision counterparts and therefore should only be used when necessary.
The Operators of the `sc_bigint<n>` and `sc_biguint<n>`

- Type `sc_bigint` is a 2’s complement signed integer of any size
- Type `sc_biguint` is an unsigned integer of any size
- The precision used for the calculations depends on the sizes of the operands used

<table>
<thead>
<tr>
<th>Bitwise</th>
<th>~  &amp;</th>
<th></th>
<th>^</th>
<th>&gt;&gt;</th>
<th>&lt;&lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+  -  *  /  %</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>=  +=  -=  *=  /=  %=  &amp;=</td>
<td>=  ^=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equality</td>
<td>==  !=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;  &lt;=  &gt;  &gt;=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Autoincrement</td>
<td>++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Autodecrement</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Select</td>
<td>[x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part Select</td>
<td>range()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Arbitrary Length Bit Vector.** \$sc\_bv\langle n\rangle$

- The \texttt{sc\_bv} is a 2-valued arbitrary length vector to be used for large bit vector manipulation.

- The \texttt{sc\_bv} type will simulate faster than the \texttt{sc\_lv} type:
  - Without tri-state capability and arithmetic operations.

- Type \texttt{sc\_biguint} could also be used for these operations, but:
  - It is optimized for arithmetic operations, not bit manipulation operations.
  - Type \texttt{sc\_bv} will still have faster simulation time.
The New Operators for \textit{sc\_bv<n>} \\

- The new operators perform bit reduction 
  - \texttt{and\_reduce()}
  - \texttt{or\_reduce()}
  - \texttt{xor\_reduce()}

```cpp
sc\_bv<64> databus;
sc\_logic result;
result = databus.or\_reduce();
```

If \textit{databus} contains 1 or more 1 values the result of the reduction will be 1. 

If no 1 values are present the result of the reduction will be 0 indicating that \textit{databus} is all 0’s.
# The Operators of the `sc_bv<n>`

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise</td>
<td>~, &amp;</td>
</tr>
<tr>
<td>Assignment</td>
<td>=, &amp;=,</td>
</tr>
<tr>
<td>Equality</td>
<td>==, !=</td>
</tr>
<tr>
<td>Bit Selection</td>
<td>[x]</td>
</tr>
<tr>
<td>Part Selection</td>
<td>range()</td>
</tr>
<tr>
<td>Concatenation</td>
<td>(,)</td>
</tr>
<tr>
<td>Reduction</td>
<td>and_reduce(), or_reduce(), xor_reduce()</td>
</tr>
</tbody>
</table>
Arbitrary Length Logic Vector \( \texttt{sc\_lv\langle n\rangle} \)

- The \( \texttt{sc\_lv\langle n\rangle} \) data-type represents an arbitrary length vector in which each bit can have one of four values.
- To supply the design that need to be modeled with tri-state capabilities.
- These values are exactly the same as the four values of type \( \texttt{sc\_logic} \).
- Type \( \texttt{sc\_lv\langle n\rangle} \) is just a sized array of \( \texttt{sc\_logic} \) objects.
- The \( \texttt{sc\_lv} \) types cannot be used in arithmetic operations directly.
The Operators of the \texttt{sc\_lv\{n\}}

<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>

\texttt{sc\_uint\{16\}} uint16;
\texttt{sc\_int\{16\}} int16;
\texttt{sc\_lv\{16\}} lv16;
lv16= uint16; // convert uint to lv
int16 = lv16; // convert lv to int

To perform arithmetic functions, first assign \texttt{sc\_lv} objects to the appropriate SystemC integer.
Fixed Point Types

- For a high level model, floating point numbers are useful to model arithmetic operations.
- Floating point numbers can handle a very large range of values and are easily scaled.
- Floating point data types are typically converted or built as fixed point data types to minimize the amount of hardware cost.
- To model the behavior of fixed point hardware, designers need bit accurate fixed point data types.
- Fixed point types are also used to develop DSP software.
Fixed Point Types (cont’)

- There are 4 basic types used to model fixed point types in SystemC
  - sc_fixed
  - sc_ufixed
  - sc_fix
  - sc_ufix
- Types sc_fixed and sc_fix specify a signed fixed point data type
- Types sc_ufixed and sc_ufix specify an unsigned fixed point data type
Fixed Point Types (cont’)

- Types `sc_fixed` and `sc_ufixed` use static arguments to specify the functionality of the type
  - Static arguments must be known at compile time
- Types `sc_fix` and `sc_ufix` can use argument types that are non-static
  - Non-static arguments can be variables
  - Types `sc_fix` and `sc_ufix` can use variables to determine word length, integer word length, etc.
Syntax of the Fixed Point Types

- **wl** - Total word length
  - Used for fixed point representation. Equivalent to the total number of bits used in the type.

- **iwl** - Integer word length
  - To specifies the number of bits that are to the left of the binary point (.) in a fixed point number.

- **q_mode** – quantization mode.

- **o_mode** - overflow mode

- **n_bits** - number of saturated bits
  - This parameter is only used for overflow mode

- **x,y** - object name
  - The name of the fixed point object being declared.

```plaintext
sc_fixed<wl, iwl, q_mode, o_mode, n_bits> x;
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> y;
sc_fix x(list of options);
sc_ufix y(list of options);
```
# Quantization Modes

<table>
<thead>
<tr>
<th>Quantization Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rounding to plus infinity</td>
<td>SC_RND</td>
</tr>
<tr>
<td>Rounding to zero</td>
<td>SC_RND_ZERO</td>
</tr>
<tr>
<td>Rounding to minus infinity</td>
<td>SC_RND_MIN_INF</td>
</tr>
<tr>
<td>Rounding to infinity</td>
<td>SC_RND_INF</td>
</tr>
<tr>
<td>Convergent rounding</td>
<td>SC_RND_CONV</td>
</tr>
<tr>
<td>Truncation</td>
<td>SC_TRN</td>
</tr>
<tr>
<td>Truncation to zero</td>
<td>SC_TRN_ZERO</td>
</tr>
</tbody>
</table>
# Overflow Modes

<table>
<thead>
<tr>
<th>Overflow Mode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturation</td>
<td>SC_SAT</td>
</tr>
<tr>
<td>Saturation to zero</td>
<td>SC_SAT_ZERO</td>
</tr>
<tr>
<td>Symmetrical saturation</td>
<td>SC_SAT_SYM</td>
</tr>
<tr>
<td>Wrap-around)</td>
<td>SC_WRAP</td>
</tr>
<tr>
<td>Sign magnitude wrap-around</td>
<td>SC_WRAP_SM</td>
</tr>
</tbody>
</table>
# The Operators of Fixed Point

<table>
<thead>
<tr>
<th>Operator class</th>
<th>Operators in class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise</td>
<td>~ &amp; ^</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>* / + - &lt;&lt; &gt;&gt; ++ --</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>Assignment</td>
<td>= *= /= += -= &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Data-Types
- Processes
- Interfaces
- Simulation Supports
- System Design Environments
- HW/SW Co-Verification
- Conclusion
Processes

- Processes are the basic unit of execution within SystemC
- Processes are called to simulate the behavior of the target device or system
- Processes provide the mechanism of concurrent behavior to model electronic system
- A process must be contained in a module
- Processes cannot not be hierarchical
  - No process will call another process directly
- Processes can call methods and functions that are not processes
Processes (cont’)

- Processes have sensitivity lists
  - a list of signals that cause the process to be invoked, whenever the value of a signal in this list changes

<table>
<thead>
<tr>
<th>Verilog</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger signals</td>
<td>sensitive $\ll$ trigger signals</td>
</tr>
<tr>
<td>posedge trigger signals</td>
<td>sensitive_pos $\ll$ trigger signals</td>
</tr>
<tr>
<td>negedge trigger signals</td>
<td>sensitive_neg $\ll$ trigger signals</td>
</tr>
</tbody>
</table>

- Processes trigger other processes by assigning new values to the hardware signals in the sensitivity list of the other process
Processes (cont’)

● Three types of SystemC processes
  – Methods — SC_METHOD
  – Threads — SC_THREAD
  – Clocked Threads — SC_CTHREAD
**Process — SC_METHOD**

- A method that does not have its own thread of execution
  - Cannot call code with `wait()`
- Executed when events (value changes) occur on the sensitivity list
- When a method process is invoked, it executes and returns control back to the simulation kernel until it is finished
- Users are strongly recommended not to write infinite loops within a method process
  - Control will never be returned back to the simulator
SC_METHOD (Example)

// rcv.h
#include "systemc.h"
#include "frame.h"

SC_MODULE(rcv) {
    sc_in<frame_type> xin;
    sc_out<int> id;
    void extract_id();
    SC_CTOR(rcv) {
        SC_METHOD(extract_id);
        sensitive(xin);
    }
};

// rcv.cc
#include "rcv.h"
#include "frame.h"

void rcv::extract_id() {
    frame_type frame;
    frame = xin;
    if(frame.type == 1) {
        id = frame.ida;
    } else {
        id = frame.idb;
    }
}

To register the member function with the simulation kernel
Process — SC_THREAD

- Thread process can be suspended and reactivated
- A thread process can contain wait() functions that suspend process execution until an event occurs on the sensitivity list
- An event will reactivate the thread process from the statement that was last suspended
- The process will continue to execute until the next wait()
SC_THREAD (Example of a Traffic Light)

// traff.h
#include "systemc.h"
SC_MODULE(traff) {
    // input ports
    sc_in<bool> roadsensor;
    sc_in<bool> clock;
    // output ports
    sc_out<bool> NSred;
    sc_out<bool> NSyellow;
    sc_out<bool> NSgreen;
    sc_out<bool> EWred;
    sc_out<bool> EWyellow;
    sc_out<bool> EWgreen;
    void control_lights();
    int i;
    // Constructor
    SC_CTOR(traff) {
        SC_THREAD(control_lights); // Thread Process
        sensitive << roadsensor;
        sensitive_pos << clock;
    }
};

// traff.cc
#include "traff.h"
void traff::control_lights() {
    NSred = false;
    NSyellow = false;
    NSgreen = true;
    EWred = true;
    EWyellow = false;
    EWgreen = true;
    while (true) {
        while (roadsensor == false)
            wait();
        NSyellow = false;
        // road sensor triggered
        NSgreen = false;
        // yellow interval over
        NSred = true;
        // set EW to green
        EWgreen = true;
        EWyellow = false;
        EWred = false;
        for (i= 0; i<50; i++)
            wait();
        .
        .
        .
    }
}
Process — SC_CTHREAD

- Clocked thread process is a special case of the thread processes
- A clocked thread process is only triggered on one edge of one clock
  - Matches the way that hardware is typically implemented with synthesis tools
- Clocked threads can be used to create implicit state machines within design descriptions
- Implicit state machine
  - The states of the system are not explicitly defined
  - The states are described by sets of statements with wait() function calls between them
- Explicit state machine
  - To define the state machine states in a declaration
  - To use a case statement to move from state to state
SC_CTHREAD (Example of a BUS function)

```
// bus.h
#include "systemc\h"
SC_MODULE(bus) {
  sc_in_clk clock;
  sc_in<bool> newaddr;
  sc_in<sc_uint<32>> addr;
  sc_in<bool> ready;
  sc_out<sc_uint<32>> data;
  sc_out<bool> start;
  sc_out<bool> datardy;
  sc_inout<sc_uint<8>> data8;
  sc_uint<32> tdata;
  sc_uint<32> taddr;
  void xfer();
  SC_CTOR(bus) {
    SC_CTHREAD(xfer, clock.pos());
    datardy.initialize(true); // ready to accept new address
  }
};
// bus.cc
#include "bus.h"
void bus::xfer() {
  start = true; // new addr for memory controller
  wait();
  // wait 1 clock between data transfers
  data8 = taddr.range(15,8);
  start = false;
  wait();
  data8 = taddr.range(23,16);
  wait();
  data8 = taddr.range(31,24);
  wait();
  // now wait for ready signal from memory controller
  wait_until(ready.delayed() == true);
  // now transfer memory data to databus
  tdata.range(7,0) = data8.read();
  wait();
  tdata.range(15,8) = data8.read();
  wait();
  tdata.range(23,16) = data8.read();
  wait();
  tdata.range(31,24) = data8.read();
  data = tdata;
  datardy = true; // data is ready, new addresses ok
}
```


Events

- An event is represented by class `sc_event`
  - Determines whether and when a process’s execution should be triggered or resumed
- An event is usually associated with some changes of state in a process or of a channel
- The owner of the event is responsible for reporting the change to the event object
  - The act of reporting the change to the event is called `notification`
- The event object is responsible for keeping a list of processes that are `sensitive` to it
- Thus, when notified, the event object will inform the scheduler of which processes to trigger
Event Notification and Process Triggering

Process or Channel (owner or event)

Notify immediately, after delta-delay, or after time T

event

Process 1

Process 2

Process 3
A hardware signal is responsible for notifying the event whenever its value changes

- A signal of Boolean type has two additional events
  - One associated with the positive edge
  - One associated with the negative edge
- A more complex channel, such as a FIFO buffer
  - An event associated with the change from being empty to having a word written to it
  - An event associated with the change from being full to having a word read from it
An event object may also be used directly by one process $P1$ to control another process $P2$

- If $P1$ has access to event object $E$ and $P2$ is sensitive to or waiting on $E$, then $P1$ may trigger the execution of $P2$ by notifying $E$
- In this case, event $E$ is not associated with the change in a channel, but rather with the execution of some path in $P1$
Sensitivity

- The sensitivity of a process defines when this process will be resumed or activated.
- A process can be sensitive to a set of events.
- Whenever one of the corresponding events is triggered, the process is resumed or activated.
- Two types
  - Static Sensitivity
  - Dynamic Sensitivity
Static Sensitivity

- Static sensitivity list
  - In a module, the sensitivity lists of events are determined before simulation begins
  - The list remains the same throughout simulation

- RTL and synchronous behavioral processes only use static sensitivity lists
Dynamic Sensitivity

- It is possible for a process to temporarily override its static sensitivity list
  - During simulation a thread process may suspend itself
  - To designate a specific event $E$ as the current event on which the process wishes to wait
  - Then, only the notification of $E$ will cause the thread process to be resumed
  - The static sensitivity list is ignored
Dynamic Sensitivity — wait()

To wait for a specific event $E$, the thread process simply call $\text{wait()}$ with $E$ as argument:

$$\text{wait}(E)$$

Composite events, for use with wait only, may be formed by conjunction (AND) or disjunction (OR)

$$\text{wait}(E1 \ & \ E2 \ & \ E3)$$
$$\text{wait}(E1 \ | \ E2 \ | \ E3)$$
**Dynamic Sensitivity — wait() (Cont’)**

The `wait()` function may also take as argument a time

```c
wait(200, SC_NS);
```

or, equivalently

```c
sc_time t(200, SC_NS);
wait(t);
```

By combining time and events, we may impose a timeout on the waiting of events

```c
wait(200, SC_NS, E);
```

waits for event $E$ to occur, but if $E$ does not occur within 200ns, the thread process will give up on the wait and resume
Dynamic Sensitivity — next_trigger()

- Calling Next_Trigger() does not suspend the current method process
- Execution of the process will be invoked only when the event specified by next_trigger() occurs.
- If an invocation of a method process does not call next_trigger(), then the static sensitivity list will be restored

The calling will make the current method process wait \( E \) within a timeout of 200ns. If \( E \) occurs within 200ns, the method process will be triggered

\[
\text{next_trigger}(200, \text{SC_NS}, E)
\]

Otherwise, when the timeout expires, the method process will be triggered and its static sensitivity list will be back in effect
Special Dynamic Sensitivity for SC_CTHREAD — wait_until()

- The `wait_until()` method will halt the execution of the process until a specific event has occurred.
- This specific event is specified by the expression to the `wait_until()` method.

```plaintext
This statement will halt execution of the process until the new value of roadsensor is true.

wait_until(roadsensor.delayed() == true);
```
Special Dynamic Sensitivity for SC_CTHREAD — watching()

- SC_CTHREAD processes typically have infinite loops that will be continuously executed.
- A designer typically wants some way to initialize the behavior of the loop or jump out of the loop when a condition occurs.
- The watching construct will monitor a specified condition and transfer the control to the beginning of the process.
Special Dynamic Sensitivity for SC_CTHREAD — watching() (Cont’)

```c++
// datagen.h
#include "systemc.h"
SC_MODULE(data_gen) {
    sc_in_clk clk;
    sc_inout<int> data;
    sc_in<bool> reset;
    void gen_data();
    SC_CTOR(data_gen){
        SC_CTHREAD(gen_data, clk.pos());
        watching(reset.delayed() == true);
    }
};
```

```c++
// datagen.cc
#include "datagen.h"
void gen_data() {
    if (reset == true) {
        data = 0;
    }
    while (true) {
        data = data + 1;
        wait();
        data = data + 2;
        wait();
        data = data + 4;
        wait();
    }
}
```

specifies that signal reset will be watched for this process

- If signal reset changes to true, the watching expression will be true and the SystemC scheduler will halt execution of the while loop for this process
- start the execution at the first line of the process
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Data-Types
- Processes
- Interfaces
  - Simulation Supports
  - System Design Environments
  - HW/SW Co-Verification
- Conclusion
Communication between Design Blocks - Channels, Interfaces, and Ports

- Traditionally, the hardware signals are used for communication and synchronization between processes
- The level of abstraction is too low for system design view

1. Channels
2. Interfaces
3. Ports

- Interfaces and ports describe what functions are available in a communications package
  - Access points
- Channels defines how these functions are performed
  - Internal operations
Example of Modules, Ports, Interfaces, and Channels

- **Port**
- **Interface**
- **Primitive Channel**
- **Port-channel binding**

**Diagram:**

- Module 1
- HC
- Module 2
Interfaces

- The “windows” into channels that describe the set of operations
- Define sets of methods that channels must implement
- Specify only the signature of each operation, namely, the operation’s name, parameters, and return value
- It neither specifies how the operations are implemented nor defines data fields
Interfaces (cont’)

- All interfaces must be derived, directly or indirectly, from the abstract base class: `sc_interface`
- The concept of interface is useful to model layered design
  - Connection between modules which are different level of abstraction
- Relationship with ports
  - Ports are connected to channels through interfaces
  - A port that is connected to a channel through an interface sees only those channel methods that are defined by the interface
Interface Examples

- All interface methods are pure virtual methods without any implementation

```cpp
template <class T>
class sc_read_if :
public sc_interface
{
public:
  // interface methods
  virtual const T& read() const = 0;
};
```

An example read interface: `sc_read_if` this interface provides a 'read' method
Interface Examples

template <class T>
class sc_write_if
 : virtual public sc_interface
{
 public:
 // interface methods
 virtual void write( const T& ) = 0;
};

An example write interface: sc_write_if
this interface provides a 'write' method
Interface Examples

template <class T>
class sc_read_write_if :
  public sc_read_if<T>,
  public sc_write_if<T>
{
};

An example read/write interface: sc_read_write_if
this defines a read/write interface by deriving from
the read interface and write interface
**Ports**

- A port is an object through which a module can access a channel’s interface.
- A port is the external interface that pass information to and from a module, and trigger actions within the module.
- A port connects to channels through interfaces.
Ports (cont’)

- A port can have three different modes of operation
  - Input (sc_in<T>)
  - Output (sc_out<T>)
  - Inout (sc_inout<T>)

<table>
<thead>
<tr>
<th>Verilog</th>
<th>SystemC</th>
</tr>
</thead>
</table>
| module module name (module  | SC_MODULE (module name) {
|    ports);                   |     sc_in <data type> port name;             |
|     input <range> port name; |     sc_out <data type> port name;            |
|     output <range> port name; |     sc_inout <data type> port name;          |
|     inout <range> port name; |     ...                                       |
|     ...                       |                                              |
Ports (Cont’)

- A port of a module can be connected to
  - Zero or more channels at the same level of hierarchy
  - Zero or more ports of its parent module
  - At least one interface or port
- `sc_port` allows accessing a channel’s interface methods by using operator . or operator [ ]
- In the following example:
  - “input” is an input port of a process
  - read() is an interface method of the attached channel

```c
a = input->read(); // read from the first (or only) channel of input
b = input[2]->read(); // read from the third channel of input
```
Access Ports

**Verilog**

```verilog
...  
input In;  
output Out;  
reg Reg;  
...  
Reg=In;  
Out=Reg;
```

**SystemC**

```systemc
...  
sc_in<bool> In;  
sc_out<bool> Out;  
bool Reg;  
...  
Reg=In.read();  
Out.write(Reg);
```

**Verilog**

```verilog
...  
input [1:0] sel;  
...  
always @(…)  
case (sel)  
  2'b00 : statement1; break;
...  
```

**SystemC**

```systemc
...  
sc_in<sc_bv<2>> sel;  
sc_uint<2> sell;  
...  
void Multiplexer_proc()  
{  
sell=sel.read();  
switch(sell)  
{  
  case 0 : statement1; break;
  ...
```
Specialized Ports

- Specialized ports can be created by refining port base class `sc_port` or one of the predefined port types
  - Addresses are used in addition to data
    - Bus interface.
  - Additional information on the channel’s status
    - The number of samples available in a FIFO/LIFO
  - Higher forms of sensitivity
    - `Wait_for_request()`
Port-less Channel Access

- In order to facilitate IP reuse and to enable tool support, SystemC 2.0 define the following mandatory design style
  - Design style for inter-module level communication
  - Design style for intra-module level communication
**Port-less Channel Access (cont’)**

- For inter-module level communication, ports must be used to connect modules to channels
  - Ports are handles for communicating with the “outside world” (channels outside the module)
  - The handles allow for checking design rules and attaching communication attributes, such as priorities
  - From a software point-of-view they can be seen as a kind of smart pointers
- For intra-module level communication, direct access to channels is allowed
  - Without using the ports.
  - Access a channel’s interface in a “port-less” way by directly calling the interface methods.
Channels

- A channel implements one or more interfaces, and serves as a container for communication functionality.
- A channel is the workhorse for holding and transmitting data.
- A channel is not necessarily a point-to-point connection.
- A channel may be connected to more than two modules.
- A channel may vary widely in complexity, from hardware signal to complex protocols with embedded processes.
- SystemC 2.0 allows users to create their own channel types.
Channels (cont’)

- **Primitive channels**
  - Do not exhibit any visible structure
  - Do not contain processes
  - Cannot (directly) access other primitive channels

- **Hierarchical channels**
  - Basically are modules
  - Can have structure
  - Can contain other modules and processes
  - Can (directly) access other channels
Primitive Channels

- The hardware signal
  - `sc_signal<T>`
- The FIFO channel
  - `sc_fifo<T>`
- The mutual-exclusion lock (mutex)
  - `sc_mutex`
The Hardware Signal – `sc_signal<T>`

- The semantics are similar to the VHDL signal
- `Sc_signal<T>` implements the interface `sc_signal_inout_if<T>`

```cpp
// controller.h
#include "statemach.h"

SC_MODULE(controller) {
    sc_in<sc_logic> clk;
    sc_out<sc_logic> count;
    sc_in<sc_logic> status;
    sc_out<sc_logic> load;
    sc_out<sc_logic> clear;
    sc_signal<sc_logic> lstat;
    sc_signal<sc_logic> down;
    state_machine *s1;  // state is another module

    SC_CTOR(controller) {
        // .... other module statements
        s1 = new state_machine("s1");
        s1->clock(clk);  // special case port to port binding
        s1->en(lstat);  // port en bound to signal lstat
        s1->dir(down);  // port dir bound to signal down
        s1->st(status); // special case port to
        // port binding
    }
};
```

The example above shows a port bound to another port (special case) and a port bound to a signal.
The FIFO Channel – sc_fifo<T>

- To provide both blocking and nonblocking versions of access
- Sc_fifo<T> implements the interfaces sc_fifo_in_if<T> and sc_fifo_out_if<T>

Blocking version
- If the FIFO is empty
  - suspend until more data is available
- If the FIFO is full
  - suspend until more space is available

NonBlocking version
- If the FIFO is empty
  - do nothing
- If the FIFO is full
  - do nothing
The Mutual-Exclusion Lock (Mutex) – sc_mutex

- Model critical sections for accessing shared variables
- A process attempts to lock the mutex before entering a critical section
- If the mutex has already been locked by another process, it will cause the current process to suspend
# Channel Design Rules

<table>
<thead>
<tr>
<th>Channel Type</th>
<th>Connection Rules</th>
</tr>
</thead>
</table>
| `sc_signal<T>`        | - No more than one driver, i.e., at most one output (`sc_out<T>`) or bi-directional port (`sc_inout<T>`) connected.  
                          - Arbitrary number of input ports (`sc_in<T>`) can be connected. |
| `sc_fifo<T>`          | - At most one input port can be connected.  
                          - At most one output port can be connected.  
                          - No bi-directional ports. |
Channel Attributes

- Channel attributes can be used for a per-port configuration of the communication
- Channel attributes are helpful especially when modules are connected to a bus
- Attributes that can be used
  - Addresses (in case the module doesn't use specialized ports, addresses can be specified as arguments of the access methods)
  - Addressing schemes (e.g. constant address vs. auto-increment)
  - Connect module as master or slave or master/slave
  - Priorities
  - Buffer sizes
Channel Attributes (Example)

- Let mod be an instance of a module and let port be a port of this module

  ```
  // create a local channel
  message_queue mq;
  ...
  // connect the module port to the channel
  mod.port( mq );
  ...
  ```

- A channel attribute can now be specified, for example:

  ```
  // specify a channel attribute
  mq.priority( mod.port, 2 );
  ...
  ```

- which sets the priority attribute for mod.port to 2.
Hierarchical Channels

- To model the new generation of SoC communication infrastructures efficiently
- For instance, OCB (On Chip Bus)
  - The standard backbone from VSIA
  - The OCB consisting of several intelligent units
    - Arbiter unit
    - A Control
    - Programming unit
    - Decode unit
- For modeling complex channels such as the OCB backbone, primitive channels are not very suitable
  - Due to the lack of processes and structures
- For modeling this type of channels, hierarchical channels should be used
**Primitive Channels v.s Hierarchical Channels**

- Use primitive channels
  - When you need to use the request-update scheme
  - When channels are atomic and cannot reasonably be chopped into smaller pieces
  - When speed is absolutely crucial (using primitive channels we can often reduce the number of delta cycles)
  - When it doesn’t make any sense trying to build up a channel (such as a mutex) out of processes and other channels
Primitive Channels v.s Hierarchical Channels (Cont’)

- Use hierarchical channels
  - When channels are truly hierarchical and users would want to be able to explore the underlying structure
  - When channels contain processes
  - When channels contain other channels
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Data-Types
- Processes
- Interfaces
- **Simulation Supports**
- System Design Environments
- HW/SW Co-Verification
- Conclusion
Clock Objects

- Clock objects are special objects which generate timing signals to synchronize events in the simulation.
- Clocks order events in time so that parallel events in hardware are properly modeled by a simulator on a sequential computer.
- Typically clocks are created at the top level of the design in the testbench and passed down through the module hierarchy to the rest of the design.
Clock Objects (Example)

This declaration will create a clock object named ck1 with a period of 20ns, a duty cycle of 50%, the first edge will occur at 0 time units, and the first value will be true.
## Simulation Control

**Verilog**

```
#1000 $finish;
```

**SystemC**

```
sc_start(1000, SC_NS);
```

**Verilog**

```verilog
initial
  begin
    #0 data=4'b0; rst=1'b0;
    #2 rst=1'b1;
    #5 data=4'b1010;
    #3 data=4'b0011;
```

**SystemC**

```systemc
void gen()
{
  while(1) {
    data.write(0x0);
    rst.write(false);
    wait(2, SC_NS);
    rst.write(true);
    wait(5, SC_NS);
    data.write(0xA);
    wait(3, SC_NS);
    ...
  }
}
```

**Simulation Control**

Due to the use of `wait` statements, the simulation is set to run for 1000ns.
**Design Example: 4-bit LFSR**

- Four files are created to describe this circuit
  - LFSR.h (header file for declaration)
  - LFSR.cpp (body of LFSR)
  - LFSR_env.cpp (simulation environment)
  - Main.cpp (top module)
#include "systemc.h"

SC_MODULE(LFSR) {
    sc_in<bool> clk;
    sc_in<bool> reset_n;
    sc_out<unsigned int> random;
    SC_CTOR(LFSR) {
        SC_METHOD(LFSR_proc);
        sensitive_pos << clk;
        sensitive_neg << reset_n;
    }
    void LFSR_proc(); // body proc
    sc_uint<4> Reg;
    bool X0,R1,R2,R3,R4;
};

SC_MODULE(LFSR_Gen) {
    sc_out<bool> reset_n;
    SC_CTOR(LFSR_Gen) {
        SC_THREAD(Gen_proc);
    }
    void Gen_proc(); // body proc
};

SC_MODULE(LFSR_Mon) {
    sc_in<unsigned int> random;
    SC_CTOR(LFSR_Mon) {
        SC_METHOD(Mon_proc);
        sensitive << random;
    }
    void Mon_proc(); // body proc
};
#include "LFSR.h"

void LFSR::LFSR_proc()
{
    if(reset_n.read()==false) {
        R1=0;
        R2=0;
        R3=0;
        R4=0;
    } else {
        if(R3==R4) X0=1;
        else X0=0;
        R4=R3;
        R3=R2;
        R2=R1;
        R1=X0;
    }

    Reg[0]=R1;
    Reg[1]=R2;
    Reg[2]=R3;
    Reg[3]=R4;
    random.write(Reg);
}
#include "LFSR.h"

void LFSR_Gen::Gen_proc()
{
  while (1) {
    reset_n=1;
    wait(2,SC_NS);
    reset_n=0;
    wait(4,SC_NS);
    reset_n=1;
    wait(100,SC_NS);
  }
}

void LFSR_Mon::Mon_proc()
{
  cout << "random= ";
  cout << random << "\n";
}
```cpp
#include "LFSR.h"

int sc_main(int argc, char* argv[]) {
    sc_signal<bool> reset_n;
    sc_signal<unsigned int> random;

    sc_time t1(2,SC_NS); // 1 cycle= 2ns
    sc_clock clk("clk",t1,0.5);

    LFSR_Gen M1("LFSR_Gen");
    M1(reset_n);
    LFSR M2("LFSR");
    M2(clk,reset_n,random);
    LFSR_Mon M3("LFSR_Mon");
    M3(random);

    sc_start(100,SC_NS);
    return 0;
}
```
Running Results

```
random= 0
random= 1
random= 0
random= 1
random= 3
random= 7
random= 14
random= 13
random= 11
random= 6
random= 12
random= 9
random= 2
random= 5
random= 10
random= 4
random= 8
random= 0
......
```
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Data-Types
- Processes
- Interfaces
- Simulation Supports
- **System Design Environments**
- HW/SW Co-Verification
- Conclusion
The Supported Tools for SystemC

- Platform and Compiler
- System design environments
Platform and Compiler

- Typically, a compiler for C++ standard can compile the SystemC source code well
  - SystemC just a extended template
- GNU gcc for many platform
- Sun with solaris
  - Forte c++
- HP
  - Hp aC++
- Intel with Microsoft OS
  - MS Visual C++
System Design Environments

- Synopsys
  - CoCentric System Studio (CCSS)
- Cadence
  - Signal Processing Worksystem (SPW)
- Agilent
  - Advanced Design System (ADS)
- CoWare
  - N2C Design System
- ....
CoCentric System Level Design Platform

CoCentric System Studio

C/SystemC, Reference Design Kit -> Performance Exploration -> HW/SW Co-design

SystemC Executable Specification -> Chip Verification

SystemC Synthesizable model -> CoCentric SystemC Compiler

Software C-Code -> Software Implementation

Processor Model
CoCentric System Level Design Platform

- Algorithm libraries and Reference Design Kits
  - Broadband Access: ADSL, DOCSIS cable modem
  - Wireless: CDMA, Bluetooth, GSM/GPRS, PDC, DECT, EDGE
  - Digital Video: MPEG-2, MPEG-4
  - Broadcast standard: DAB, DVB
  - Error Correcting Coding: RS coding, Hamming coding
  - Speech Coding: ITU G.72X, GSM speech, AMR speech
CoCentric System Level Design Platform

- Simulation, Debugging and Analysis
  - Mixing of architectural and algorithmic models in the same simulation
  - Works with VCS, Verilog-XL, ModelSim, import Matlab models for co-simulation
  - Macro-debugging at the block level
  - Micro-debugging at the source code level
  - Davis
  - VirSim
CoCentric System Level Design Platform

- Path to implementation
  - Synthesizable SystemC code generated automatically
Advanced Design System

- Ptolemy Models
- HDL Simulation
- Hardware Emulation
- Logic Synthesis

- ADS
  DSP Designer

- C/C++ Models
- HDL Models
- MATLAB
- Measurement Instrumentation
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Data-Types
- Processes
- Interfaces
- Simulation Supports
- System Design Environments
- **HW/SW Co-Verification**
- Conclusion
Traditional HW/SW Verification Flow

- Enter system integration and verification stage until both HW/SW are finished
  - Errors may happen at the beginning
- Hinges on the physical prototype like FPGA
  - Chips respin waste a lot of money and time
  - Error correction through redesign
- Signal visibility will be getting worse
  - Change the pin assignment of the FPGA in order to get the visibility
Pre-Silicon Prototype

- **Virtual prototype**
  - Simulation environment
- **Emulator**
  - Hundreds kilo Hz
- **Rapid prototype**
  - Combination of FPGAs and dedicated chips that can be interconnected to instantiate a design
  - Tens mega Hz
- **Roll-Your-Own (RYO) prototype**
  - FPGA and Board
  - Tens mega Hz
Virtual Prototypes

Definition
- A simulation model of a product, component, or system

Features
- Higher abstraction level
- Easily setup and modify
- Cost-effective
- Great observability
- Shorten design cycles
Verification Speed

- Cell loss?
- Bit error rate??
- Bus bandwidth?
- Cache size?
- Handshake?
- reset?

Reference: Synopsys
Verification Speed

Reference: Synopsys
HW/SW Co-Simulation

- Couple a software execution environment with a hardware simulator
- Provides complete visibility and debugger interface into each environment
- Software normally executed on an Instruction Set Simulator (ISS)
- A Bus Interface Model (BIM) converts abstract software operations into detailed pin operations
Advantages of HW/SW Co-Simulation (1/2)

- Simulate in minutes instead of days
- Early architecture closure reduces risk by 80%
- Start software development 6 months earlier
- Simulate 100x~1000x faster than RTL
- HW designers can use the tools which are familiar to them
- SW programmers can use all their favorite debugger to observe software state and control the executions
Advantages of HW/SW Co-Simulation (2/2)

- **Software Engineers**
  - Simulation model replace stub code
  - More time to develop & debug code
  - Validate code against hardware as you develop
  - Maintain software design integrity

- **Hardware Engineer**
  - Embedded software replaces test bench
  - Reduce the chance of an ASIC or Board spin
  - Resolve gray areas before tape out
Synopsys’s SystemC Solution

- System Studio
  - SystemC simulation
- SystemC Compiler
  - SystemC synthesis
- DesignWare
  - AMBA/ARM SystemC models

Reference: Synopsys
Synopsys System Studio

Reference: Synopsys
Mentor Graphic : Seamless CVE

Reference : Mentor Graphics
Cadence: Incisive Platform

Unified Environment

Transaction support

Verilog
VHDL
AMS

Single-kernel architecture

Unified test generation

SystemC
PSL/Sugar
Algorithm

Acceleration-on-Demand

Reference: Cadence
Conclusions

- The system level design is a new design challenge
  - Both hardware and software issues have to be considered
- High level abstraction and modeling is essential for system design in future
  - SystemC is a more mature language, but not the only one
- Co-design methodology can reduce the design cycle
  - Allow earlier HW/SW integration
- Virtual co-simulation environment is required
  - Reduce the cost and design cycle of hardware prototype
  - Simulate 100x~1000x faster than RTL with the models of higher level of abstraction
- A hot and hard area for designers and EDA vendors
**References**

- **Book materials**
- **Manual**
  - SystemC Version 2.0 User’s Guide
  - Functional Specification for SystemC 2.0
- **Slides**
  - SoC Design Methodology, Prof. C.W Jen
  - Concept of System Level Design using Cocentric System Studio, L.F Chen
- **WWW**
  - [http://www.synopsys.com](http://www.synopsys.com)
  - [http://www.systemc.org](http://www.systemc.org)
  - [http://www.forteds.com](http://www.forteds.com)
  - [http://www.celoxica.com](http://www.celoxica.com)
  - [http://www.adelantetechnologies.com](http://www.adelantetechnologies.com)
  - [http://mint.cs.man.ac.uk/Projects/UPC/Languages/VHDL+.html](http://mint.cs.man.ac.uk/Projects/UPC/Languages/VHDL+.html)
  - [http://www.doulos.com](http://www.doulos.com)
  - [http://www.coware.com](http://www.coware.com)