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
- Interfaces
- Processes
- Data-Types
- Design Examples
- System Design Environments
- HW/SW Co-Verification
- Conclusion
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Interfaces
- Processes
- Data-Types
- Design Examples
- 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++

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

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

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

C/C++ System Level Model

Analysis

Results

Refine

Verilog

Simulation

Synthesis

Verilog Testbench

Convert by Hand

Reference: DAC 2002 SystemC Tutorial
Problems with the Design Flow

- C/C++ System Level Model
- Analysis
- Results
- Refine
- Convert by Hand
- Verilog
  - Simulation
  - 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

Project Complete

Reference: Mentor Graphic
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
      - H/W Design Flow
      - S/W Development
4. Integration and Verification
Outline

- Introduction
- **System Modeling Languages**
  - SystemC Overview
  - Interfaces
  - Processes
  - Data-Types
  - Design Examples
  - 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

System-Level Modeling Language

- C/C++ Based
- VHDL/Verilog Replacements
- Higher-level Language
- Entirely New Language
- Java-Based

Replacements

- SystemC
- Cynlib
- SoC++
- Handel-C
- A/RT (Library)
- VHDL+
- System Verilog
- SDL
- SLDL
- SUPERLOG
- Java
## Language Use

<table>
<thead>
<tr>
<th></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

- Initialization of variables, the semantics of "posedge" and "negedge" constructs, record-like constructs, handling of interfaces and various keywords
- ANSI C-style ports, named parameter passing, comma-separated sensitivity lists and attributes
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??
Bibliography

- 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**
  - Interfaces
  - Processes
  - Data-Types
  - Design Examples
  - System Design Environments
  - HW/SW Co-Verification
- Conclusion
# SystemC Language Architecture

## SystemC Language Layering Architecture

<table>
<thead>
<tr>
<th>Not-standard</th>
<th>Methodology-Specific Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Channels for Various Model of Computation</td>
<td>Channels</td>
</tr>
<tr>
<td>Kahn Process Networks</td>
<td>Master/Slave Library, etc.</td>
</tr>
<tr>
<td>Static Dataflow, etc.</td>
<td></td>
</tr>
</tbody>
</table>

## 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

---

37
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
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)

```c
int sc_main(int argc, char* argv[]) {
    sc_signal<int> val;
    sc_signal<sc_logic> load;
    sc_signal<sc_logic> reset;
    sc_signal<int> result;
    sc_clock ck1("ck1", 20, 0.5, 0, true);
    filter f1("filter");
    f1.clk(ck1.signal());
    f1.val(val);
    f1.load(load);
    f1.reset(reset);
    f1.out(result);
    // rest of sc_main not shown
}
```

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

- The basic building blocks for partition a design
- Modules are declared with the SystemC keyword `SC_MODULE`
- Typical contains
  - 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
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.
    }
};
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Interfaces
- Processes
- Data-Types
- Design Examples
- 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

- 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**

Module with a port

Hierarchical channel with a port

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
  : virtual 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

```cpp
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
A port can have three different modes of operation
- Input (sc_in<T>)
- Output (sc_out<T>)
- Inout (sc_inout<T>)
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
```
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 – \textit{sc\_signal\textless{}T\textgreater{}}

- The semantics are similar to the VHDL signal
- \texttt{sc\_signal\textless{}T\textgreater{}} implements the interface \texttt{sc\_signal\_inout\_if\textless{}T\textgreater{}}

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

SC_MODULE(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 – \texttt{sc\_fifo<T>}

- To provide both blocking and nonblocking versions of access
- \texttt{Sc\_fifo<T>} implements the interfaces \texttt{sc\_fifo\_in\_if<T>} and \texttt{sc\_fifo\_out\_if<T>}

<table>
<thead>
<tr>
<th>Blocking version</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the FIFO is empty</td>
</tr>
<tr>
<td>suspend until more data is available</td>
</tr>
<tr>
<td>If the FIFO is full</td>
</tr>
<tr>
<td>suspend until more space is available</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NonBlocking version</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the FIFO is empty</td>
</tr>
<tr>
<td>do nothing</td>
</tr>
<tr>
<td>If the FIFO is full</td>
</tr>
<tr>
<td>do nothing</td>
</tr>
</tbody>
</table>
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>Type</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sc_signal&lt;T&gt;</code></td>
<td>- No more than one driver, i.e., at most one output (<code>sc_out&lt;T&gt;</code>) or bi-directional port (<code>sc_inout&lt;T&gt;</code>) connected.</td>
</tr>
<tr>
<td></td>
<td>- Arbitrary number of input ports (<code>sc_in&lt;T&gt;</code>) can be connected.</td>
</tr>
<tr>
<td><code>sc_fifo&lt;T&gt;</code></td>
<td>- At most one input port can be connected.</td>
</tr>
<tr>
<td></td>
<td>- At most one output port can be connected.</td>
</tr>
<tr>
<td></td>
<td>- No bi-directional ports.</td>
</tr>
</tbody>
</table>
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

```c
// 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:

```c
// 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
- Interfaces
- **Processes**
- Data-Types
- Design Examples
- 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 (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
- Processes cannot not be hierarchical
  - No process will call another process directly
- Processes trigger other processes by assigning new values to the hardware signals in the sensitivity list of the other process
- Processes can call methods and functions that are not processes
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)**

```c
// 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);
    }
};
```

```c
// 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);
        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 = false;
    while (true) {
        while (roadsensor == false)
            wait();
        NSgreen = false; // road sensor triggered
        NSyellow = true; // set NS to yellow
        EWred = false;
        EWgreen = false;
        for (i=0; i<5; i++)
            wait();
        NSgreen = false; // yellow interval over
        NSyellow = false; // set NS to red
        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() {
    while (true) {
        // wait for a new address to appear
        wait_until(newaddr.delayed() == true);
        // got a new address so process it
        taddr = addr.read();
        datardy = false; // cannot accept new address now
        data8 = taddr.range(7,0);
        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

trigger

Process 1

trigger

Process 2

trigger

Process 3
Events in Classical Hardware Modeling

- 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
Relationship Between the Events

• An event object may also be used directly by one process $P_1$ to control another process $P_2$
  – If $P_1$ has access to event object $E$ and $P_2$ is sensitive to or waiting on $E$, then $P_1$ may trigger the execution of $P_2$ 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 $P_1$
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 `wait()` with $E$ as argument:

```
wait(E)
```

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

```
wait(E1 & E2 & E3);
wait(E1 | E2 | E3);
```
Dynamic Sensitivity — wait() (Cont’)

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

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

or, equivalently

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

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

```
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 on E within a timeout of 200ns. If E occurs within 200ns, the method process will be triggered.

```
next_trigger(200, 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.

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

```c
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
- Interfaces
- Processes
- Data-Types
- Design Examples
- 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_bignint` – 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

```cpp
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
- \texttt{sc\_int}\textless n\textgreater
  - A Fixed Precision Signed Integer
  - 2’s complement notation
- \texttt{sc\_uint}\textless n\textgreater
  - A Fixed Precision Unsigned Integer
The Operators of `sc_int<n>` and `sc_uint<n>`

<table>
<thead>
<tr>
<th>Bitwise</th>
<th>~ &amp;</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>
</tr>
<tr>
<td>Assignment</td>
<td>=  +=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>!=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Autodecrement</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Select</td>
<td>[x]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part Select</td>
<td>range()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation</td>
<td>(,)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Examples of `sc_int<n>` and `sc_uint<n>`

```cpp
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_uint` 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

Bitwise  ~ & | ^ >> <<
Arithmetic + - * / %
Assignment = += -= *= /= %= &= |= ^=
Equality == !=
Relational < <= > >=
Autoincrement ++
Autodecrement --
Bit Select [x]
Part Select range()
Arbitrary Length Bit Vector \( \texttt{sc\_bv}<n> \)

- 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 `sc_bv<n>`

- The new operators perform bit reduction
  - `and_reduce()`
  - `or_reduce()`
  - `xor_reduce()`

```c
sc_bv<64> databus;
sc_logic result;
result = databus.or_reduce();
```

If `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 `databus` is all 0’s.
The Operators of the `sc_bv<n>`

- Bitwise: ~, &, |, ^, <<, >>
- Assignment: =, &=, |=, ^=, ==, !=
- Equality: ==, !=
- Bit Selection: [x]
- Part Selection: range()
- Concatenation: (,)
- Reduction: and_reduce(), or_reduce(), xor_reduce()
Arbitrary Length Logic Vector. $\textit{sc\_lv}\langle n\rangle$

- The $\textit{sc\_lv}\langle n\rangle$ 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 $\textit{sc\_logic}$.
- Type $\textit{sc\_lv}\langle n\rangle$ is just a sized array of $\textit{sc\_logic}$ objects.
- The $\textit{sc\_lv}$ types cannot be used in arithmetic operations directly.
# The Operators of the `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>

```plaintext
sc_uint<16> uint16;
sc_int<16> int16;
sc_lv<16> lv16;
lv16= uint16; // convert uint to lv
int16 = lv16; // convert lv to int
```

To perform arithmetic functions, first assign `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` uses 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.

```cpp
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

Overflow Mode
Saturation
Saturation to zero
Symmetrical saturation
Wrap-around)
Sign magnitude wrap-around

Name
SC_SAT
SC_SAT_ZERO
SC_SAT_SYM
SC_WRAP
SC_WRAP_SM
# 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
- Interfaces
- Processes
- Data-Types
- Design Examples
  - System Design Environments
  - HW/SW Co-Verification
- Conclusion
Example: Filter (Block Diagram)

\[ Y[i] = \text{SUM} \ (X[i-t] \times C[t]) \quad t=0\ldots3 \]
Example: Filter (Architecture)

Circular convolution
Example: Filter (Software)

```cpp
#include<systemc.h>
#include<sc_mslib.h>
// design for pure software
template<class signal=double>
class filter_s {
  int tap;
  int X_start;
  signal *X;
  signal *Coeff;

public:
  filter_s(int tap,signal Coeff[]);
  signal function(signal in);
};
```
Example: Filter (UTF)

```cpp
template<
class signal=double>
SC_MODULE(filter_u) {
    sc_inslave<signal> in;
    sc_outmaster<signal> out;
    filter_s<signal> *component;
    void simulate();
    SC_CTOR(filter_u);
};
```
Example: Filter (CA)

template<class signal=double>
SC_MODULE(filter_c) {
    sc_in<signal> in;
    sc_in_clk clock;
    sc_out<signal> out;
    filter_s<signal> *component;
    void simulate();
    SC_CTOR(filter_c);
};
Example: Filter (Unit Test)

typedef double signal;
    sc_clock c1("clock2",10);
tester_c<signal> t1("tester2");
t1.clock(c1.signal());
t1.setup(5,DATA);

filter_c<signal> f1("filter2");
f1.in(in);
monitor_c<signal> m1("monitor2");
m1.clock(c1.signal());

sc_signal<signal> in;
t1.out(in);
signal DATA[5]={1,2,3,4,5};

sc_signal<signal> out;
f1.clock(c1.signal());
f1.out(out);
m1.in(out);
sc_start(100);
Example: FFT (Packet)

template<class signal,int length>
class fft_packet {
  public:
    signal data[length];
    fft_packet(signal *data) {
      for(int i=0;i<length;i++)
        this->data[i]=data[i];
    }
};
Example: FFT (UTF)

template<class signal,int length>
SC_MODULE(fft_u) {
    sc_inslave<fft_packet<signal,length> > in;
    sc_outmaster<fft_packet<signal,length> > out;
    fft_s<signal> *component;
    void simulate();
    SC_CTOR(fft_u);
};
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Interfaces
- Processes
- Data-Types
- Design Examples
- 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 → Processor Model

SystemC Executable Specification

SystemC Synthesizable model

Software C-Code

Chip Verification

CoCentric SystemC Compiler

Software Implementation
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

C/C++ Models  HDL Models  MATLAB  Measurement Instrumentation

ADS
DSP Designer

135
Outline

- Introduction
- System Modeling Languages
- SystemC Overview
- Interfaces
- Processes
- Data-Types
- Design Examples
- 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

References: 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

Algorithm

Simulation

Memory

Bus

Debugger

Software

Architecture

ARM9 / AHB

SystemC

Hardware

Reference : Synopsys
Mentor Graphic : Seamless CVE

Reference : Mentor Graphics
Cadence: Incisive Platform

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.adelantetecnologies.com](http://www.adelantetecnologies.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)