KDEC Technical Seminar
SystemC Overview & Example
: 8-Bit RISC System Design

KodeC  http://asic.khu.ac.kr
Kook,ilho  goodkook@nms.anlab.co.kr
AnsLab Co.  http://www.anlab.co.kr
SoC Design Flow

Reference: Rosenfield (SystemC, Europe)

Design Process Today

Reference: Wang (Synopsys)
HDL Based Flow

Problems:
- Written specifications are incomplete and inconsistent
- Translation to HDL is time consuming and error prone

Reference: Wang (Synopsys)

C/C++ Based Flow

Turning Algorithms into the Right Architectures for ASICs quicker and better

Reference: Wang (Synopsys)
Why C/C++ Based Design

- **Specification** between architect and implementer is executable
- High **simulation speed** at the higher level of abstraction
- **Refinement**, no translation into HDL (no “semantic gap”)
- **Testbench re-use**

![SoC Design](image)

Reference: Wang (Synopsys)

Using Executable Specifications

- **Ensure COMPLETENESS of Specification**
  - Even components (e.g., Peripherals) are so COMPLEX
  - Create a program that Behave the same way as the system
- **Avoid UNAMBIGUOUS Interpretation of the Specification**
  - Avoids unspecified parts and inconsistencies
  - IP customer can evaluate the functionality up-front
- **Validate system functionality before implementation**
  - Early feedback from customer
  - Create early model and Validate system performance
- **Refine and Test the implementation of the Specification**
  - Test automation improves Time-To-Market
Executable Spec Motivation

- Customer System
- Paper Spec
- HDL Design
- Netlist
- Layout
- Silicon

Verification, Error Checking Bottleneck (SystemC™)

Test automation: C-Interface (PLI/FLI), Coverage Test

Customer System
- Executable Spec
- HDL Design
- Netlist
- Layout
- Silicon

Reference: Mayer (Infineon)

Executable Spec Potential

- Example: Cerberus_FPI (ca. 6000 Lines of Code)

Reused for HDL Verification
Testbench 42%

Reused by several Tool Partners worldwide.

SW Driver 39%

HW Model 19%

Reference: Infineon
Can C++ be used as is?

- C++ does not support
  - *Hardware style communication*
    - Signals, protocols, etc.
  - *Notion of time*
    - Time sequenced operations.
  - *Concurrency*
    - Hardware and systems are inherently concurrent, i.e. they operate in parallel.
  - *Reactivity*
    - Hardware is inherently reactive, it responds to stimuli and is in constant interaction with its environment, which requires handling of exceptions.
  - *Hardware data types*
    - Bit type, bit-vector type, multi-valued logic type, signed and unsigned integer types and fixed-point types.

SystemC vs. SpecC

- Constructs to model system architecture
  - Hardware timing
  - Concurrency
- Adding these constructs to C
  - SystemC
    - C++ Class library
    - Standard C/C++ Compiler: bcc, msvc, gcc, etc...
  - SpecC
    - Language extension: New keywords & Syntax
    - Translator for C
SystemC is ...

- C++ Class Library use for
  - Cycle-Accurate model for Software Algorithm
  - Hardware Architecture
  - Interface of SoC (System-on-Chip)
  - System-level designs
  - Executable Specification

What is SystemC?

SystemC is a C++ Class library
Include any C++ programs, libraries, encapsulation...a methodology for modeling SoC designs consisting DSPs, ASICs, IP-Cores, Interfaces,...

SystemC also enables
- Modeling at high level of abstraction (e.g. communication protocols)
- Refinement to hardware
- Software modeling: interrupts, exception handling
- System wide verification
- Hardware/Software co-verification
- IP exchange

SystemC provides all the advantages of Verilog and VHDL
- Concurrent processes (e.g. methods, threads, clocked threads)
- Concept of a clock
- Wide variety of bit-true data types

SystemC IS NOT
Another C++ dialect -> it is C++
Just for hardware modeling only -> you can model hardware AND software in C++ with SystemC

Reference: Wang(Synopsys)
Short History of SystemC™

1997 DAC Paper

Scenery V0.9 Launches 9/27/1999

1999 2000

Source Code
User Guide
Reference Manual
Discussion Forum
All available from http://www.SystemC.org

Reference: Wang(Synopsys)

SystemC™ Highlights (1)

- Support Hardware-Software Co-Design
- Interface in a C++ environment
  - Modules
    - Container class includes hierarchical Entity and Processes
  - Processes
    - Describe functionality, Event sensitivity
  - Ports
    - Single-directional(in, out), Bi-directional(inout) mode
  - Signals
    - Resolved, Unresolved signals
  - Rich set of port and signal types
  - Rich set of data types
    - All C/C++ types, 32/64-bit signed/unsigned, fixed-points, MVL, user defined
SystemC™ Highlights (2)

- Interface in a C++ environment (continued)
  - Clocks
    - Special signal, Timekeeper of simulation and Multiple clocks, with arbitrary phase relationship
  - Cycle-based simulation
    - High-Speed Cycle-Based simulation kernel
  - Multiple abstraction levels
    - From untimed from high-level functional model
    - To detailed clock cycle accuracy RTL model
  - Communication Protocols
  - Debugging Supports
    - Run-Time error check
  - Waveform Tracing
    - Supports VCD, WIF, ISBD

SystemC and User Module
System Design Methodology

- Current
  - **Manual Conversion** from C to HDL Creates Errors
  - Disconnect Between System Model and HDL Model
  - Multiple System Tests

- **SystemC (Executable-Specification)**
  - Refinement Methodology
  - Written in a Single Language

Current Methodology

- **Manual Conversion** Creates Errors
- Disconnect Between System Model and HDL Model
- Multiple System Tests

```
  C/C++ System Level Model
    \arrow[->]{Analysis \arrow[->]{Result \arrow[->]{VHDL/Verilog \arrow[->]{Simulation \arrow[->]{Synthesis}}}}
```

Refine
SystemC Methodology

- Refinement Methodology
  - Not convert C to HDL for timing constructs
- Written in Single Language from System to RTL model
- Less effort to convert Synthesizable HDL

Design Flow in SystemC

Design Exploration
  - Performance Analysis
  - HW/SW partitioning

Multi-Tasking
  - Abstract RTOS
  - Inter Process communication
  - Scheduling/Privilege

Target RTOS

UTF

Refine

UnTimed Functional

Timed Functional

HW/SW Partition

BCA

Refine

Bus Cycle Accurate

Cycle Accurate

Synthesizable

Target Code

RTL
System Abstraction Level

- **Untimed Functional (UTF) Level**
  - Decompose system into functional module
  - Abstract communication channels
  - Data transactions without notion of TIME

- **Timed Functional (TF) Level**
  - Functional process can be assigned a “RUN-TIME”
  - “Timed” but NOT “Clocked”

- **Bus-Cycle Accurate (BCA) Level**
  - Transactions on the bus cycle-accurately
  - Some behavior left at untimed level

- **Cycle Accurate (CA) Level**
  - Behavior is clock cycle accurate
  - Ready to RTL HDL

System Design Flow

**Untimed Functional (UTF) Level**

- **Purpose**
  - Executable Spec. of a complete system
  - Algorithmic behavior
  - Functional decomposition

- **Methodology**
  - NO distinction of HW and SW
  - Remote Procedure Call (RPC) protocol
  - Data Transaction : Abstract Port
  - Minimal concurrent behavior
**System Design Flow**

**Timed Functional (TF) Level**

- **Purpose**
  - Performance modeling
  - Time budgeting

- **Methodology**
  - Process may be assigned a “Rum-Time”
  - Timed and Untimed System expression
  - TIME is used to express duration only
  - Process execution by RPC chain and concurrent thread
  - Design exploration: HW/SW partitioning

---

**System Design Flow**

**HW/SW Partitioning**

- **HW Mapping**
  - Architecture
  - Transform functional module into cycle accurate
  - Refine Communication protocol

- **SW Module Partition**
  - Tasks
  - Inter-task communication
  - Synchronization
  - Use RTOS (Real-Time Operating System)
**System Design Flow**

**Bus-Cycle Accurate (BCA) Level**

- **Purpose**
  - Model Hardware with Bus Architecture
- **Methodology**
  - Abstract ports refined to Bus ports
    - Bus: Data, Address, Control terminals
    - Protocol: No-, Enable-, Full-Handshake
  - Processors/Bus controllers are synchronized using clock
  - Modules are modeled bus cycle-accurately, but some behavior

---

**System Design Flow**

**Cycle Accurate (CA) Level**

- Synthesizable RTL
Getting Started

- Compiler
  - gcc (version 2.95.2)
  - native compiler Visual C++, SUN cc
- Debugger
  - gdb, ddd
  - lint, profiler, memory access checking
  - quantify, purify
- www.gnu.org

Compile and Run

Your standard C/C++ development environment

Compiler
- Linker
- Debugger

Class library and simulation kernel

header files

libraries

Executable specification

source files for system and testbenches

Executable = simulator
SystemC Design Unit

- Module
  - Ports and Signals
- Constructor for Process
  - Sensitize to signals
- Hierarchy of Module
  - Module instanciation
  - Port mapping

Module

- Module
  - Basic building block of design partitioned
  - C++ Class, similar to “entity” (VHDL) or “module” (Verilog)

```
SC_MODULE(module_name)
{
  // Declare Module Ports
  // Declare Module Signals, Member functions
  // Module Construct : SC_CTOR
  // Process Construct and Sensitize to signal:
  //     SC_METHOD, SC_THREAD, SC_CTHREAD
  // Sub- Module Instantiate and Port Mapping
  // Initialize of Module Signals
}
```
Module Ports

- Pass data to or from processes of a module
- Input port
  \[
  \text{sc\_in}\langle\text{type}\rangle \text{ port\_name};
  \]
- Output port
  \[
  \text{sc\_out}\langle\text{type}\rangle \text{ out\_port\_name};
  \]
- Bi-Directional port
  \[
  \text{sc\_inout}\langle\text{type}\rangle \text{ inout\_port\_name};
  \]

Module Signals

- Local to a Module
- Used to connect ports of sub-modules

\[
\text{sc\_signal}\langle\text{type}\rangle \text{ signal\_name};
\]
Internal Data Storage

- Local variable
- Cannot be used to connect ports
- Storage types
  - C++ type
  - SystemC type
  - User defined type

Module Constructor

- Similar to “architecture” or module body
- Processes and/or Sub-module instantiated
- “module_name” passed when instantiated to identify the module

```c
SC_MODULE(module_name)
{
    ..........
    SC_CTOR(module_name) // Create Module
    {
        // Processes and Sensitize
        // Sub-Module Instantiate
        // Initialize Local signals and Internal storage
    }
}
```
Sub-Module Instantiate

- Module Instantiate
  
  ```
  module_type Inst_module ("label");
  ```

- Module Instantiate as pointer
  
  ```
  module_type* plnst_module;
  
  Inst_module = new module_type("label");
  ```

Sub-Module Port Mapping

- Positional Port Mapping
  
  ```
  Inst_module << s << c << q;
  (*plnst_module)(s,c,q);
  ```

- Named Port Mapping
  
  ```
  Inst_module.a(s);
  Inst_module.b(c);
  Inst_module.q(q);
  plnst_module -> a(s);
  plnst_module -> b(c);
  plnst_module -> q(q);
  ```
Hierarchy of Modules

```
SC_MODULE(filter) {
    // Sub-modules : "component"
    sample *s1;
    coeff *c1;
    mult *m1;

    // Signals
    sc_signals <sc_uint32> > q, s, c; // Signals
    // Module constructor : "architecture" body
    SC_CTOR(filter) {
        // Instantiate Sub-Modules and Port mapping
        s1 = new sample("s1");
        s1->din(q); // Named Mapping
        s1->dout(s);
        c1 = new coeff("c1");
        c1->out(c);
        m1 = new mult("m1");
        (*m1)(s, c, q); // Positional Mapping
    }
}
```

Processes

- Member function of SC_MODULE
- Provides functionality of Module
- Identify to SystemC kernel (Simulator)
- Call and Execute : “Sensitive to”
- Type of Processes (execute method)
  - Method : SC_METHOD
  - Thread : SC_THREAD
  - Clocked Thread : SC_CTHREAD
Process and "Sensitize to"

- **System C**
  ```c
  #include "systemc.h"
  SC_MODULE(dff) {
    sc_in<bool> din;
    sc_in<bool> clock;
    sc_out<bool> dout;
    void doit(); // Member function
    SC_CTOR(dff) {
      SC_METHOD(doit); // Process
      sensitive_pos << clock; // Sensitize to
    }
  }
  void dff::doit() { // Member function as Process body
    dout = din;
  }
  ```

- **VHDL**
  ```vhdl
  entity dff is
    port ( din, clock : in bit;
          dout : out bit );
  end dff;
  architecture dff of dff is
  begin
    doit : process(clock) -- Sensitivity List
    begin
      if (clock'event and clock='1') then
        dout <= din;
      end if;
    end process;
  end dff;
  ```

Initialize Module

- When a Module instantiated,
  - Module name passed to identify module
  - Module Constructor creates and initializes this Module
  ```c
  SC_MODULE(ram) {
    sc_in<int> addr; // Input port
    sc_in<int> datain; // Input port
    sc_in<bool> rwb; // Input port
    sc_out<int> dout; // Output port
    int memdata[64]; // Local memory storage, Created when instantiated
    int i;
    void ramread();
    void ramwrite();
    SC_CTOR(ram) {
      SC_METHOD(ramread) // Memory Read Process
      sensitive << addr << rwb;
      SC_METHOD(ramwrite) // Memory Write Process
      sensitive << addr << datain << rwb;
      for (i=0; i<64; i++) // Initialize Local Storage when instantiated
        memdata[i] = 0;
    }
  }
  ```
Lab1. Counter

- **Components**
  - DUT Module: “count”
    - Up counter
    - Init. Data Loadable
  - Stimuli Generator: “count_stim”
    - Load control
    - Init. Data
  - DUT monitor: “display”
    - Monitor DUT output

- **TestBench: “count_tb”**
  - Clock generator: `sc_clock()`
  - Run Simulator: `sc_start()`

Lab1. Count

How to Compile and Link

- **Requirement (PC version)**
  - Win32 (Windows NT/9x/2000)
  - MicroSoft Visual C/C++ 6.0
  - SystemC 1.0

- **To compile and Link with MicroSoft Visual C/C++ 6.0**
  - Create a project “count” as “Win32 consol application”
  - Add preprocessor definition: `/D “SC_INCLUDE_FX”`
  - Enable C++ RTTI (Run-Time Type Information) option: `/GR`
  - Add SystemC include path: `/I “your_SystemC/src”`
  - Link with SystemC Library “systemc.lib”

- **To get Lab1. Count source**
  - [http://www.anslab.co.kr/download/SystemC/3_Module.zip](http://www.anslab.co.kr/download/SystemC/3_Module.zip)
SystemC IP: Ans_RISC817

- Feature
  - 55 single word 16-bit instruction
  - 8-bit data
  - interrupt capability
  - Stack on data memory
  - Direct, indirect and relative addressing mode
  - 64Kx16(banked) program memory
  - 256x16(banked) data memory
  - 32 IO Address space(expandable)

Ans_RISC817 RISC

- IP designed with SystemC
- Ans_RISC817 Core
  - Timing accurate model
  - Program ROM
  - Data RAM
- Expandable Custom peripheral
  - Dynamic linking with core executable
- Software environment
  - Assembler
  - Debugger embedded into core executable
Development System

Executable

Ans_RISC Core

Debugger
Line Assembler
Dis-Assembler
Trace/Go
Prog. Download

Core Monitoring

Peripherals
(DLL)

Program ROM

Program ROM Image

Assembler

TestBench

RISC Simulator

Test Modules
- RISC Core
- Peripheral
- Monitor (Trace)

Debugger
- Basic Tool for RISC
Test & S/W Development
- User interface
- Embedded in the executable

int sc_main(int argc, char* argv[])
{
    sc_clock CLK("clk", CLK_PERIOD);
    sc_signal<sc_bit> rst;
    sc_signal<sc_bv<8>> io_in;
    ...
    // UUT : Ans_RISC817 Core
    ans_risc817 u_ans_risc817("u_ans_risc817");
    u_ans_risc817.clk(CLK);
    u_ans_risc817_risc817.rst(rst);
    ...
    // Peripheral
    ans_risc817_peri u_ans_risc817_peri("u_ans_risc817_peri");
    u_ans_risc817_peri.clk(CLK_ADC);
    u_ans_risc817_peri.rst(rst);
    u_ans_risc817_peri.int1(int1);
    ...
    // Monitor
    monitor u_monitor("u_monitor");
    u_monitor.clk(CLK);
    ...
    // Debugger
    Debug(u_ans_risc817, &p_download, &rst);
    return(0);
}
Debugger

- Command Line UI
  - get command string: gets()
  - command line interpreter
- Generate Stimuli
  - Reset
  - Program ROM download
- Initialize/Run
  - sc_initialize()
  - sc_start()

Executing “Ans_RISC817”
Waveform Trace

- VCD output
- Hierarchical Tracing

```
// VCD Monitor
sc_trace_file* tf;
tf = sc_create_vcd_trace_file("ans_risc817");

sc_trace(tf, t_rst, "RST");
sc_trace(tf, t_clk, "CLK");
......
sc_trace(tf, u_ans_risc817.u_prog_count->pc, "PC");
```

View Waveform : GTKWave
Runtime Demo-Q&A

- Tracing Signal “sc_signal”
- Tracing Variable
- Understanding Simulation Kernel
- Debugging methodology
- Q&A
SystemC Example

"Counter"

Contents:
1. count_tb.cpp - TestBench
2. count.h, count.cpp - Core "counter"
3. count_stim.h, count_stim.cpp - Stimuli generator for Test
4. display.h, display.cpp - Output debug

/////////////////////////////////////////////////////////////////
// Filename : count_tb.cpp
// Comment  : SystemC example - Counter Stimulus
//
#
//
#include "count.h"
#include "count_stim.h"
#include "display.h"

int sc_main(int argc, char* argv[])
{
    sc_signal<bool> LOAD;
    sc_signal<int> DIN, DOUT;
    sc_clock CLOCK("clock", 20); // clock

    int sim_time = 0;

    if (argc==2)
        sim_time = atoi(argv[1]);
    if (sim_time==0)
        sim_time = 1000;

    count u_count ("count");
    u_count.load(LOAD);
    u_count.din(DIN);
    u_count.dout(DOUT);
    u_count.clock(CLOCK);

    count_stim u_count_stim("count_stim");
    u_count_stim.load(LOAD);
    u_count_stim.din(DIN);
    u_count_stim.dout(DOUT);
    u_count_stim.clock(CLOCK);

    display u_display("display");
    u_display.load(LOAD);
    u_display.din(DIN);
    u_display.dout(DOUT);
    u_display.clock(CLOCK);

    // VCD Monitor /////////////////////////////////////
    sc_trace_file* tf;
    tf = sc_create_vcd_trace_file("counter");

    sc_trace(tf, CLOCK, "CLOCK");
    sc_trace(tf, LOAD, "LOAD");
    sc_trace(tf, DIN, "DIN");
    sc_trace(tf, DOUT, "DOUT");

    sc_initialize();
    sc_start(sim_time);
    sc_close_vcd_trace_file(tf);
    return(0);
}

/////////////////////////////////////////////////////////////////
// Filename : count.h
// Comment  : SystemC example - Counter Stimulus
//
#
#include "systemc.h"

#ifndef COUNT_H
#define COUNT_H

SC_MODULE(count)
{
    sc_in<bool> load;

    sc_trace(tf, CLOCK, "CLOCK");
    sc_trace(tf, LOAD, "LOAD");
    sc_trace(tf, DIN, "DIN");
    sc_trace(tf, DOUT, "DOUT");

    sc_initialize();
    sc_start(sim_time);
    sc_close_vcd_trace_file(tf);
    return(0);
}

#endif
```cpp
SC_CTOR(count)
{
    SC_METHOD(count_up); // Method process
    sensitive_pos << clock; // Sensitive to Rising edge clock
}
```

```cpp
#include "count.h"
#include "count_stim.h"
```

```cpp
void count::count_up()
{
    if (load)
    {
        count_val = din;
    }
    else
    {
        count_val = count_val + 1; // Read/Write of local storage
    }
    dout = count_val; // Write to Output port
}
```

```cpp
SC_MODULE(count_stim)
{
    sc_out<bool> load;
    sc_out<int> din;
    sc_in<bool> clock;
    sc_in<int> dout;

    void stimgen();

    SC_CTOR(count_stim)
    {
        SC_THREAD(stimgen);
        sensitive_pos(clock);
    }
}
```

```cpp
#include "count_stim.h"
```

```cpp
void count_stim::stimgen()
{
    while(true) // infinite loop
    {
        load = true;
        din = 0; // counter load zero
        wait(); // count up, value = 1;
        load = false;
        wait(); // count up, value = 2;
        wait(); // count up, value = 3;
        wait(); // count up, value = 4;
        wait(); // count up, value = 5;
        wait(); // count up, value = 6;
        wait(); // count up, value = 7;
    }
}
```

```cpp
#include "display.h"
```
// Comment : SystemC example - Counter

#include "systemc.h"

SC_MODULE(display)
{
    sc_in<bool> load;
    sc_in<int> din;
    sc_in<bool> clock; // input ports
    sc_in<int> dout; // output port

    void display_count();

    SC_CTOR(display)
    {
        SC_METHOD(display_count);
        sensitive(dout);
    }

    void display_count()
    {
        cout << "Count = " << dout << " at " << sc_time_stamp() << "\n";
    }
}