Single Source System to Register-Transfer Level Design Methodology Using High-Level Synthesis

Petri Solanti, Mentor Graphics Deutschland GmbH
Thomas Arndt, COSEDA Technologies GmbH
Motivation

• Complex multi-domain SoC requires multiple parallel design paths
• Each paths uses the modeling language and methodology best suited for the domain (Digital, Analog, Software, etc.)
• Validation of the models often disregarded
• Growing demand for simulation models for different purposes

• New design methodology that minimizes the number of different models and modeling languages is needed
Agenda

• Problems of traditional design flows
• Languages for multi-abstraction modeling
• Single language System-to-RTL flow
• Design example
• Conclusions
Problems of Traditional Design Process

- Multiple abstraction levels of models written in different languages
- Validation of the models throughout the design process
- Growing demand for different simulation models, e.g.
  - IP model for automotive system simulator or digital twin
  - Virtual Platforms for SW development
  - HW verification models
  - Rapid prototyping
Requirements for Modeling Language

- Advanced type system supporting arbitrary width and abstract data types and polymorphism
- Support for parallel processes
- Concept of time and simulation environment capable of handling multiple scheduling schemes
- Compiler support for different target environments (SW)
- Timing abstractions: untimed, cycle-based, continuous time
- Automatic generation of gate-level netlist (HLS or RTL synthesis)
# Languages for Multiple Abstraction Modeling

<table>
<thead>
<tr>
<th>Feature</th>
<th>Java</th>
<th>Python</th>
<th>C++</th>
<th>SystemVerilog</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-true types</td>
<td>External library</td>
<td>External library</td>
<td>External library</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallel processes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Concept of time</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SW Development</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Compiler support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Modeling analog behavior</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>VerilogAMS</td>
<td>Yes</td>
</tr>
<tr>
<td>RTL generation</td>
<td>No</td>
<td>MyHDL, PyMTL</td>
<td>HLS</td>
<td>Integrated Verilog</td>
<td>HLS</td>
</tr>
</tbody>
</table>
Single Source System-to-RTL Methodology

- Using high abstraction level SystemC for modeling hardware, software and analog functionality and block-level architecture
- Generating RTL from SystemC model drops one abstraction level
- Simulation and virtual platform models extracted from SystemC model
Creating an Executable Specification

• Iterative process to create a simulation model that
  – represents full functionality of the system in abstract level
  – has initial split to hardware, software and analog functionalities

• All functionalities are written in C++ or SystemC
  – Analog using SystemC AMS
  – Digital in SystemC or C++
  – Software in C++

• Abstracted interfaces between the subsystems model the real communication between the domains

• System performance analysis using Matchlib (Interconnect bw, etc.)
Creating SystemC Testbench

• System testbench has two purposes:
  – Validation of executable specification against requirements and paper spec
  – Ensuring that all corner cases are modeled correctly

• Should be developed parallel to the simulation model
  – Bring verification knowledge into modeling phase
  – Find and fix bugs earlier
  – Second pair of eyes to reference model development

• Reuse testbench in RTL verification
Analyzing HLS Implementation Bottlenecks

• Find coding style issues preventing efficient parallelism may require major code or algorithm changes, e.g.
  – Loop dependencies
  – Array access
  – Module hierarchy
• Run high-level synthesis to uncover problems as early as possible
  – Using floating-point variables
  – Focus on desired architecture, not performance or area
• Starting at leaf/block level
Quantizing the Design

• Defining fixed-point attributes for all internal and external signals and variables
  – Number of integer and fractional bits
  – Signedness
  – Rounding and overflow handling schemes

• Requires value range analysis of signals and variables
  – Use SystemC traces
  – Tool support for extracting abs max and non-zero abs minimum

• Sophisticated type definition scheme to enable switching between floating-point and fixed-point
Exploring HW Architectures Using HLS

- Run HLS with different constraints
  - Iterate towards desired hardware architecture
  - Hierarchical or flat synthesis
  - Optimize synthesis constraints
- Analyze Power, Performance and Area
- Eventually fine tune SystemC code
Generating and Verifying RTL

• Freeze synthesis constraints into a build script
• Verify generated RTL against the SystemC code
  – RTL Co-simulation
  – Formal C-to-RTL equivalence checking
• Reuse SystemC testbench in RTL verification
  – Functional test using SystemC testbench
  – Generated functionalities must be tested with randomized tests
    • Reset behavior
    • Generated state machines
Creating Example Design and Testbench

Data Path

Mathematical SystemC model

Frequency Domain Analysis
Restructuring HW Model for HLS

- Models and test bench are usually structure hierarchically for optimal re-use
- Float vs. fixed point model comparison

Simuli re-use

HLS Compliant SystemC model

Analysis re-use
Quantizing HW Model

• Pre-quantized input to the ADC bit width limits the value range
• Dump all internal signal and variable values to file with sc_trace
• Analyze value ranges of the signals
  – Maximum absolute value
  – Minimum non-zero absolute difference between two samples
• Analyze required fixed-point attributes
  – Dynamic analysis based on simulation data (Notch filter)
  – Static analysis based on input data types and arithmetic operations (FIR)
• Run simulation with fixed-point types and compare results
Floating-Point vs. Fixed-Point Comparison

- Many automated and manual methods available
  - Automatic difference analysis with assert threshold
  - Visual or frequency analysis based analysis
Exploring HW Architectures Using HLS

- Example design synthesized with different synthesis constraints
  - Notch filter is SC_METHOD ➔ forced unrolling and pipelining
  - FIR filter is SC_THREAD ➔ Loop transformations can be explored
Conclusions

• SystemC AMS and High-Level Synthesis enable a single source language design flow from abstract Analog-Mixed-Signal system model to RTL
• Graphical SystemC design platform improves productivity
• Using High-Level synthesis enables higher abstraction level throughout the digital HW flow
• Key benefits of the SystemC Single Source AMS Flow
  – Only one model to be developed and maintained
  – No validation problems between different models
  – Supports agile design methodology and continuous integration
Questions