Building Portable Stimulus Into your IP-XACT Flow

Petri Karppa, Lauri Matilainen – Nokia Networks
Matthew Ballance – Mentor, A Siemens Company
SoC Creation and Validation Challenges

• Assembling an SoC by hand is time-consuming
  – Labor-intensive processes lead to bugs
  – IP is characterized by separate documentation

• Creating SoC-level tests is time-consuming as well
  – Bare-metal software-driven environments are complex
  – Tests are typically hand-coded (low-productivity)
Making IP Reusable with IP-XACT

• IP-XACT usage goals
  – Standardized building blocks for SoCs
  – Vendor neutrality (EDA tools & IP/SoC designs)
  – Design automation & generation

• IP-XACT component content
  – Ports/BusInterfaces – characterize block interfaces
  – Views/FileSets – specify associated files (HDL, HVL, C/C++)
  – MemoryMaps – capture addressable elements inside the block
  – ModelParameters – capture parameterizable aspects
Making IP Reusable with IP-XACT

• IP-XACT enables SoC assembly at a higher level
  – Connect IPs at the interface (vs signal) level
  – Automatically validate connection correctness

• Automate generation of correct-by-construction
  – HDL netlist
  – Compilation scripts
  – System memory map
SoC Design Flow at Nokia

- Capture IP register models in Excel
- Generate VHDL register model
- Source of memory map meta-data
SoC Design Flow at Nokia

- Package IP with IP_XACT meta-data
  - Bus interfaces
  - Memory maps
  - File sets and view

- Packaging tool automates process
SoC Design Flow at Nokia

- Designs composed from IP-XACT IP
- IP-XACT tooling derives
  - Top-level RTL
  - Makefiles and compilation scripts
  - UVM register model
  - C header files
  - Register documentation
SoC Design Flow at Nokia

- Generated files undergo checks
  - Linting
  - CDC checking
  - Synthesis
  - Sanity simulation
- Some customization may be needed
  - Tailor-made user-specific releases
Test Reuse Challenges

• Different tests used throughout a project
  – Wastes Time
  – Error Prone

• UVM constrained-random
  – High value at IP level
  – Limited value for SoC-level testing

• C tests usually directed
  – Hard to create
  – Miss corner cases
PSS Enables Test Intent Reuse

• Single specification of test intent
• Defines "scenario space" by capturing:
  – interactions
  – dependencies
  – resource contention
• Tool automates generation
  – Multiple targets
  – Target-specific customization
PSS Actions
Capture behavioral intent

- Behaviors captured as *actions*
  - Simple actions map directly to target implementation
  - Compound actions modeled via *activity*
- Actions are modular
  - Reusable
  - Interact with other actions
  - Inputs and Outputs define dataflow requirements
  - Claim system resources subject to target constraints
- Activity defines scheduling of critical actions
  - Define scheduling constraints
  - Flow objects and resources constrain scenarios

```plaintext
buffer mem_seg_b {
    rand bit[31:0] addr;
    rand bit[31:0] size;
}

resource dma_channel_r { }

action dma_mem2mem_xfer_a {
    input mem_seg_b src;
    output mem_seg_b dst;
    lock dma_channel_r channel;
    constraint size_match {
        src.size == dst.size;
    }
    // Target implementation left unspecified
}
```
PSS Elements - Components

- Components are type namespaces
- Reusable groupings of
  - Actions
  - Pools of objects and resources
- Pools capture available resources
  - Used by actions running in the component

```plaintext
component dma_c {
  buffer mem_seg_b {
    rand bit[31:0] addr;
    rand bit[31:0] size;
  }
  resource dma_channel_r { }
  pool dma_channel_r channels[16];
  action dma_mem2mem_xfer_a {
    // . . .
  }
}
```
PSS Component Tree

- Component Tree captures system resources
  - Component instances available in the system
  - Shared resource pools at the SoC level

- Actions run in the component tree context
  - Use available resources
  - Parallel action execution limited by available resources
PSS Elements – Test Realization

- Test intent must be mapped to an implementation
- PSS supports
  - Calls to external methods
  - Mapping to string templates
- Type extension provides flexibility
  - Package each mapping
  - Select target-specific mapping
- UVM sequence
- Embedded software
Embedding PSS in IP-XACT

• Reference IP-XACT files
• Collect per-IP in a fileset
• Collect per-languages files
  – UVM implementation
  – C implementation
Embedding PSS in IP-XACT

- Identify key PSS elements
  - Component
  - Top-level actions

- Automation can help
  - Identify relevant PSS files
  - Identify root component and actions
IP-XACT and PSS
Automate Component-Tree Creation

• PSS component tree often mirrors design hierarchy
  – Component instances correspond to IP and subsystem instances
• An IP-XACT tool can automate component tree creation
  – Create a PSS component instance for each IP-XACT component
• Verification engineers will design most scenarios
• But… simple bring-up tests can be created automatically
  – Create scenarios that run one of the top-level actions
  – Create scenarios that run a set of the top-level actions

```plaintext
component dma_smoke_test_c extends simple_soc_c {
  action dma_test_a {
    repeat (10) {
      do dma_c::dma_mem2mem_xfer_a;
    }
  }
}
```
IP-XACT and PSS
Bootstrap test scenario creation

• Automatically-generated PSS structure accelerates test creation
  – Aggregates available action and data types
  – Identifies root actions which are most useful to test writers

• Generated component tree saves user time and effort

• Automatically-generated memory map is always current with design

• Selected IP-XACT “view” drives appropriate test realization
IP-XACT and PSS
Better Together

• Combining IP-XACT and PSS boosts SoC-level test creation
  – IP-XACT boosts design composition productivity
  – PSS boosts test-creation productivity

• Combined, test infrastructure can be created from design structure
  – Generated PSS component tree based on IPs in the design
  – Collection of available data types and actions
  – Automated creation of simple test scenarios