UVM-Multi-Language Hands-On

Thorsten Dworzak (Verilab GmbH)
Angel Hidalga (Infineon Technologies AG)

Introduction

- UVM-ML version 1.2
  - add-on library to UVM
  - vendor/simulator independent
  - Accelera Multi-Language working group
  - easy integration of different HVLs (*e*, SV, SystemC)
  - inter-language communication via TLM2 sockets

Verification Focus (1)

- DUT is ARM CPU IP
- Different UVM-SV testbenches (DUT hierarchy)
- SystemC reference model
  - joint development between software and hardware-verification team

Verification Focus (2)

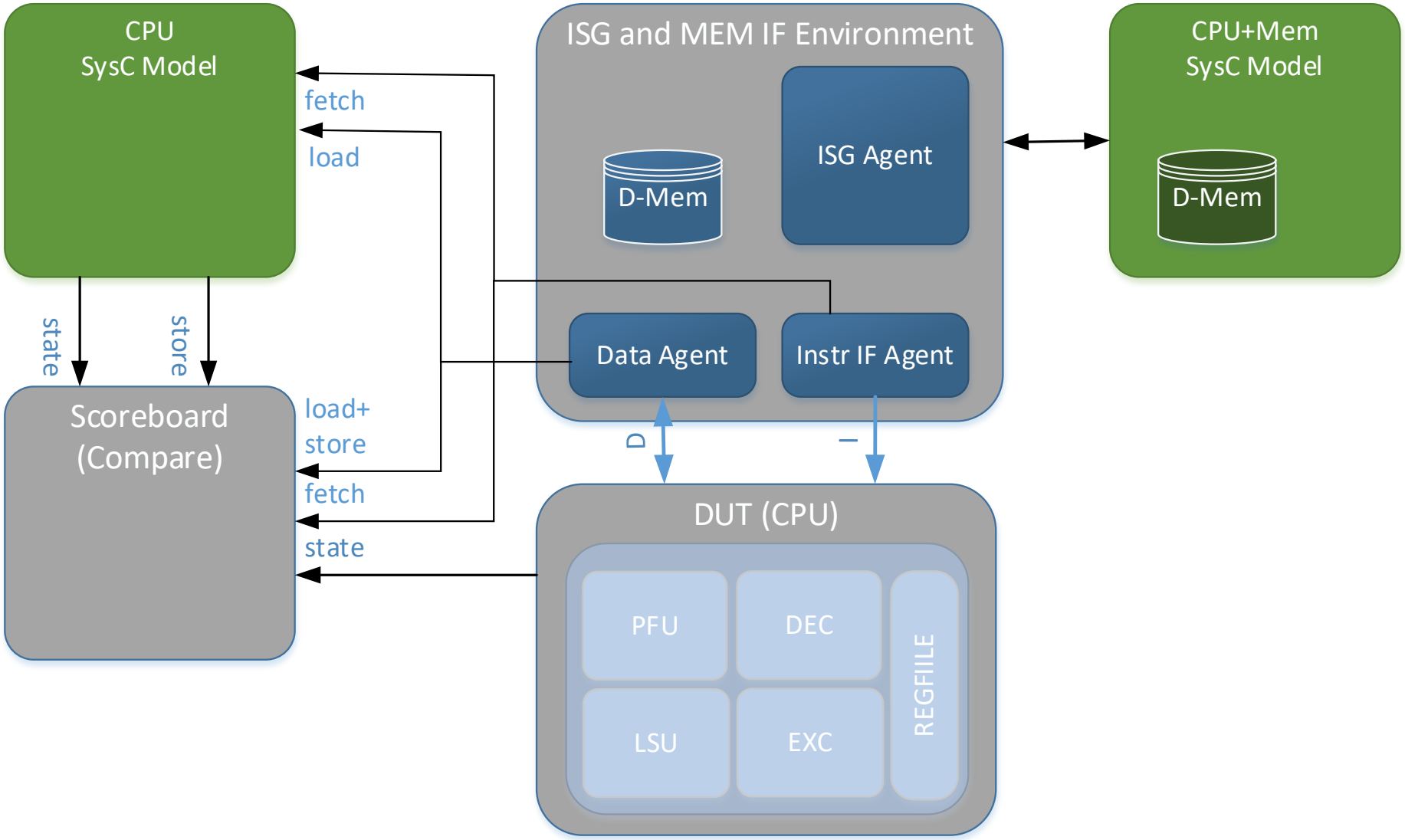
- SystemC model must support 3 use-cases

Framework	Stimuli	C-Model	Comment
UVM-SV testbench	constrained-random instruction stream generator (ISG) + legacy assembler tests	2 instances (scoreboarding + steering of ISG)	full scoreboarding + functional coverage
UVM-SV testbench	self-checking C/assembler tests	instruction stream simulator (ISS)	drop-in replacement of DUT
SDK	firmware	stand-alone ISS	rapid prototyping; added peripherals+memory system

C-Model Requirements

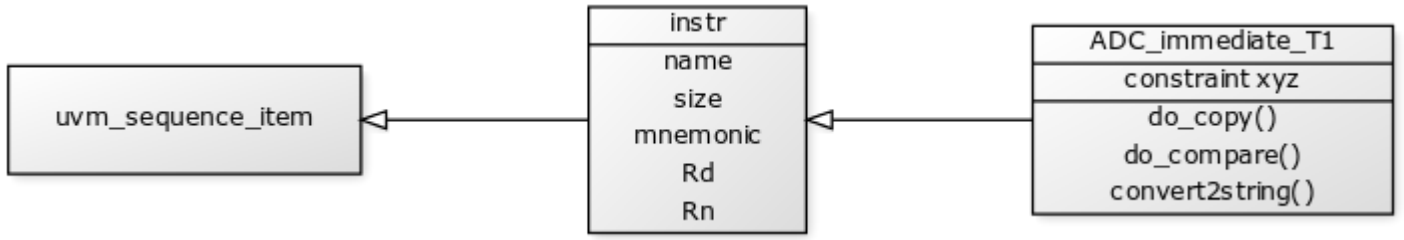
- Specification compliance (Model and DUT output must match)
- Windows+Linux cross-platform development
- C++11 wrapped in System-C modules
- High performance (for stand-alone ISS)
- Support all use-cases with minimal code overlap
- More than one instance (careful with static classes)
- Error injection possible
- State roll-back support

UVM Environment



Instruction Stream Generation (1)

- Instruction base class contains all properties that define an assembler instruction
- Specialized class defines required properties, e.g. opcode-size, legal source+destination registers.
  - SV code generated from specification



Instruction Stream Generation (2)

- An ISG sequence contains an instruction item and several fields to control the item generation
- Dedicated sequences to write/read registers, establish fault handlers etc.
- Sequence API allows flexibility to do fully random instructions, specific instructions like `LDR r0,[r5,r4,LSL #3],` and anything in-between
- C-Model is used to predict next PC value (e.g. branch target)
- Opcodes known to be skipped will be set to BKPT to detect DUT bugs
- Code and data memory are separate, so no problem of stack running into code segment

OSCI vs. ncsc

- OSCI 2.3.1 is reference implementation
  - Software team (Windows) relied on this
- Cadence implementation used by ncsc
  - Easier build-flow, used by HW-verification team
- Compatibility is "good enough"
  - OSCI-based flow can be used as fall-back solution

Linux vs. Windows

- Collaborative effort of software and hw-verification team
- Software team used MS Visual Studio
  - UVM testbench not available
  - Large suite of unit tests based on Google Test ensured up-front quality
  - Several Jenkins projects continuously checked quality metrics
    - e.g. make sure code-base compiles with gcc
- Software team followed Agile flow while HW-verification team uses traditional waterfall model
- Different SCMs

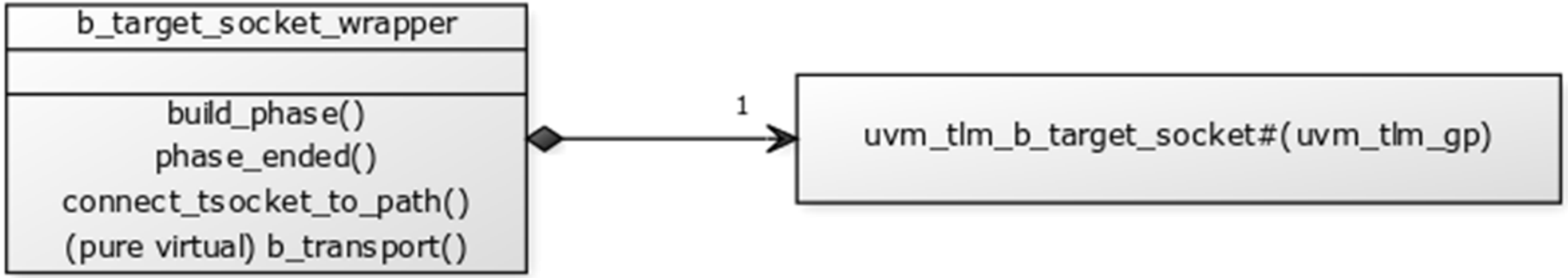
ML Interface Selection

- Choice of interfaces according to requirements

Interface	Used for	Comment
TLM-2	Busses, debug access, synchronization	Use only blocking functions
DPI-C	Reporting functions for CPU-status, exceptions, interrupts, misc. asynchronous signals	(Non-)blocking
FMI	Call C++ from VHDL	Non-blocking C-wrapper instantiates static C++ objects

TLM2 sockets

- Use of SV type `uvm_tlm_b_<target|initiator>_socket`
- Multi-socket and passthrough sockets currently not supported
- Need to use default data type `uvm_tlm_generic_payload`
  - Benefit from existing packing/serialization facilities of UVM-ML
- Created wrapper classes for the two types of sockets
  - Encapsulates standard functionality (build, register, connect, ...)
  - Unique namespace for `b_transport()` function in target socket



TLM2 Generic Payload

- UVM-ML provides packing/serialization -> less code on SV and SystemC side
- Is intended to model memory-mapped bus transactions
  - r/w, address, data, byte-enables
- Any missing functionality implemented as SV static functions
- Additional information transmitted via *payload extensions*
  - Transaction privilege-level, initiator ID, embedded commands, ...
  - Target response (type of error, busy status, ...)
- GP `response_status` field only used for TLM communication errors

DPI-C Interface (1)

- Transmit integral types and structs across the language barrier
- Export SV methods called from C++

```
export "DPI-C" function
cExpReqSysRst();

function bit cExpReqSysRst(input
byte modelType);
  ...
endfunction
```

```
extern "C" void
cExpReqSysRst(char modelType);
```

- Tasks may consume time

```
export "DPI-C" task cExpSync();

task cExpSync();
  @(posedge clk);
endtask
```

```
extern "C" void cExpSync();
```

DPI-C Interface (2)

- Import C++ methods called from SV

```
import "DPI-C" context task
cImpInitCpu();
```

```
int cImpInitCpu() {
  ...
}
```

- `context` as opposed to `pure` attribute
  - allows C++ implementation to access objects other than input parameters (e.g. SystemC objects, call exported methods)

Simulation Performance

- Using UVM testbench that allows drop-in replacement of DUT with C-model, we can compare the performance

Test	# instructions	CPU time RTL/s(1)	CPU time SysC/s(2)	Instructions/s RTL	Instructions/s SysC
dhrystone	3194	16,5	3,1	194	1030
pi	4409	27,6	3,1	160	1422
memory_byte_access	29569	74,9	10,9	395	2713
memory_attributes	109582	272,0	37,6	403	2914
whetstone_1	1184060	5973,0	380,0	198	3116
(1) Verbosity UVM_LOW, no linedebug, no trace-file					
(2) Verbosity UVM/SC_LOW, linedebug, no trace-file					

Summary

- Successfully deployed UVM-ML in SystemC/SystemVerilog environment
- Not all SystemC features implemented
- Most initial tool problems solved
- UVM-ML reduces effort to cross the language boundaries

Questions, Comments?