Single Source Register Sequencing Toolkit and Methodology for FW and Verification Co-Development

Josh Loo, Anthony Cabrera, Kiel Boyle, Scott R. Nelson
Non-Volatile Memory Solutions Group
Intel Corporation
Agenda

- What is SSRS?
- SSRS Infrastructure
- Cross-Domain Register Access Handshaking
- Results
- Extended Verification Features
What is SSRS?
What is SSRS?

The Problem:

• Modern SoC productization requires FW for production releases and RAL sequences for pre-silicon verification

• FW and verification teams commonly access registers using different techniques and source languages (ex. C++ vs. SystemVerilog/UVM)

• This leads to effort duplication and discrepancies between sequences make it difficult to replicate exact behavior on different test platforms
What is SSRS?

The Solution:

• Single Source Register Sequencing (SSRS)
  – A toolkit that includes a pair of support libraries with associated execution kernels, one in C++ and the other in SystemVerilog
  – Enables FW C++ register sequences to be run against RTL simulations without needing a CPU simulation model
  – C++ register sequences are executed on an x86 thread that runs parallel to the simulator
What is SSRS?

The Benefits:

• Earlier FW/HW integration testing against standalone IP modules
• Faster simulations due to less RTL
• Verbatim re-use of FW
• Pre-silicon verification scoped to actual FW use cases
• Accurate behavior replication on all test platforms
class ssrs_test extends uvm_test;
...
fork
    m_env.m_uvc.m_svek.run_c_sequence("path/to/cpp_collateral.so",
        "function1");

    m_uvm_sequence.start(m_sequencer);

    m_env.m_uvc.m_svek.run_c_sequence("path/to/cpp_collateral.so",
        "function2");
join
...
endclass

What is SSRS?
SSRS Infrastructure
SSRS Infrastructure

C++ Infrastructure
- Firmware HAL
- C++ Sequences
- Drivers
- Register Headers
- Register Monitoring Classes
- C++ Execution Kernel (CEK)

SystemVerilog Infrastructure
- UVC
- SystemVerilog Execution Kernel (SVEK)
- C++ Sequence Controller
- Register Transaction Adapter
- Register Abstraction Layer (RAL)

Design Under Test (DUT)
- UVM Testcase
- Register Interface

Bidirectional Communication Channel
- DPI

DPI
SSRS Infrastructure

- C++ Sequences – orchestrate driver calls to perform a series of register accesses
SSRS Infrastructure

- Drivers – stateless FW modules that implement accessors to registers
SSRS Infrastructure

- Register Headers – contain data type definitions that describe the address map and access policies of registers
• Register Monitoring Classes – class objects for registers are allocated in memory and operator overloading is used to monitor register accesses by drivers
class ssrs_test extends uvm_test;
...
  fork
    m_env.m_uvc.m_svek.run_c_sequence("path/to/cpp_collateral.so", "function1");

    m_uvm_sequence.start(m_sequencer);

    m_env.m_uvc.m_svek.run_c_sequence("path/to/cpp_collateral.so", "function2");
  join
  ...
endclass
SSRS Infrastructure

1. Initiate C++ Sequence
2. Dynamically Load .so Library
3. Obtain Function Pointer to C++ Sequence
4. Fork C++ Sequence Execution on x86 Thread
4. Control Returns to SVEK
Cross-Domain Register Access Handshaking
Cross-Domain Register Access Handshaking

Register Access By Driver Observed

RUN

BLOCKED

RUN

x86 Thread

C++ → SV Message Queue

Bidirectional Communication Channel

SV → C++ Message Queue

CEK

DPI

SVEK Tries to Dequeue New Message from CEK

Message Received

Empty

@Posedge Clock

SVEK Gets a Handle to Register from RAL

Get a Handle to Register from RAL

Perform UVM Register Transaction on DUT

SVEK Sends Completion Message to CEK

@Posedge Clock

Perform UVM Register Transaction on DUT

SVEK Sends Completion Message to CEK
Cross-Domain Register Access Handshaking

Initiating a Register Access from C++

• Driver accesses to reg objects in memory are observed by reg monitoring classes

• Overloading read and write operators triggers the CEK to convert the memory address of a register being accessed to its equivalent address from the RAL

• Register Access Message (C++ → SV)
  – RAL address of register being accessed
  – Access type (e.g. read or write)
  – Value to update DUT register with (writes only)
Cross-Domain Register Access Handshaking

RUN

BLOCKED

RUN

x86 Thread

C++ → SV Message Queue

Bidirectional Communication Channel

SV → C++ Message Queue

CEK

DPI

SVEK Tries to Dequeue New Message from CEK

Message Received

Empty

@Posedge Clock

Get a Handle to Register from RAL

Perform UVM Register Transaction on DUT

SVEK Sends Completion Message to CEK
Cross-Domain Register Access Handshaking

• Completion Message (SV → C++)
  – Value to return to C++:
    • Read accesses return the value read from the DUT
Cross-Domain Register Access Handshaking

Register Access By Driver Observed

x86 Thread

C++ $\rightarrow$ SV Message Queue

Bidirectional Communication Channel

SV $\rightarrow$ C++ Message Queue

CEK

DPI

SVEK

SVEK Tries to Dequeue New Message from CEK

Get a Handle to Register from RAL

Perform UVM Register Transaction on DUT

SVEK Sends Completion Message to CEK

@Posedge Clock

Empty

Message Received

@Posedge Clock

CEK

22
Bidirectional Communication Channel

- Encapsulates two message queues: C++ → SV and SV → C++
- Implemented in C++ and allocated in memory by CEK
- Message queues hold entries of type `message_t`, a user-defined struct that is identically defined in both language domains

<table>
<thead>
<tr>
<th>Message Queue Accessor Function</th>
<th>Action</th>
<th>Blocking?</th>
<th>Accessible via DPI Import?</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>En-queues a message</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>get()</td>
<td>De-queues a message</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>tryget()</td>
<td>Attempts to de-queue a message</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Cross-Domain Register Access Handshaking

- **DUT**
- **UVM Testbench**
- **SVEK**
- **C++ -> SV Queue**
- **SV -> C++ Queue**
- **C++ Sequence**

**[C++ Sequence Executes a Register Access]**

1. **put() Register Access Message**
2. **get() Completion Message**
3. **tryget() Register Access Message**
4. **tryget(): Received**
5. **Get Register from RAL**
6. **Register Access in DUT**
7. **Register Access in DUT (Return)**
8. **Get Register from RAL (Return)**
9. **put() Completion Message**
10. **get() Completion Message (Return)**

**Create x86 Thread to Run C++ Sequence (DPI call to CEK)**

**Run C++ Sequence**

**x86 Thread Joined**
Cross-Domain Register Access Handshaking

1. `tryget()` Register Access Message
2. `tryget()`: Empty
3. `@{(posedge clk)}`

Run C++ Sequence

Create x86 Thread to Run C++ Sequence (DPI call to CEK)

x86 Thread Joined

alt

[C++ Sequence Executes Auxiliary Operations]
Results
Results

Portability

• SSRS is easily portable to existing UVM testbenches due to its RAL interface

Multi-Platform Code Re-use

• Scope of pre-silicon verification was focused on actual FW use cases as defined by the common sequences

• Sharing common FW drivers and sequences reduced code redundancy and ensured behavior replicability across different test platforms
Results

Flush Out FW/HW Interaction Bugs Earlier

• Example:
  – FW sequence appeared to be passing in post-silicon testing
  – When re-running the FW sequence in an RTL simulation, we noticed IO glitches that could trigger undefined behavior on external components

Cross-Team Collaboration

• Verification experts should be able to easily notice errors in FW drivers or sequences since they are experts in SoC design specifications
Continuous Integration

• Changes to shared FW code triggered rigorous reviews across multiple disciplines resulting in higher quality code

• Modifications were automatically regressed against different test platforms

• Regressing FW in simulation provided prompt quality feedback to FW developers without requiring detailed knowledge about UVM environments

• SSRS provided greater debug visibility into FW interactions with the RTL by reproducing behavior in simulation
Extended Verification Features
Extended Verification Features

Constrained Random Parameter Generation

• Existing constraint constructs within a UVM test environment can be used to generate constrained-random values for C++ Sequences

• Effectively reduces manual effort for test creation while increasing functional coverage of the DUT and FW source code

• Currently exploring how to share SystemVerilog generated parameters with other test platforms
Extended Verification Features

Inter-Environment Communication

• Existing message passing mechanisms can also be used to trigger custom SystemVerilog or UVM functionality from C++

• Example use-cases:
  – Converting assertion and print statements in C++ to UVM report macros
  – Generating backdoor register accesses to accelerate run time
  – Generating burst register accesses in simulation
Conclusion

Single Source Register Sequencing (SSRS)

- Enables FW execution against RTL simulations without a CPU model
- Portable to lower-level verification environments due to RAL interface
- Provides developers with greater debug visibility into FW/RTL interactions
- Verbatim reuse of FW enables trivial behavior replication in RTL simulation
- Allows for extension to trigger custom functionality between C++ and SystemVerilog
Questions?