Efficient Standard Co-Emulation Modeling Interface (SCE-MI) Usage to Accelerate TBA Performance

Ponnambalam Lakshmanan
Analog Devices, Bengaluru, India
Agenda

• Acceleration Concept
• Introduction to SCE-MI and SCE-MI Pipes
• Testbench Architecture and Existing Challenges
• Optimizing SCE-MI pipe usage
• SCE-MI Direct Memory and Function Based Interface
• Optimized Architecture
• Results and Conclusion
Agenda

• Acceleration Concept
• Introduction to SCE-MI and SCE-MI Pipes
• Testbench Architecture and Existing Challenges
• Optimizing SCE-MI pipe usage
• SCE-MI Direct Memory and Function Based Interface
• Optimized Architecture
• Results and Conclusion
Acceleration Concept

• Hardware assisted acceleration offers better performance over software simulation.

• Hardware assisted acceleration techniques:
  – Signal Based Acceleration (SBA)
  – Transaction Based Acceleration (TBA)
  – Embedded testbench
  – Vector Based Acceleration (VBA)
  – In-circuit Emulation (ICE)
Transaction Based Acceleration

- Traditional testbench is split into two domains
  - Non-Synthesizable domain
    - Runs on simulator
  - Synthesizable domain
    - Runs on emulator
Factors Affecting Acceleration

- Various factors directly affect the emulator performance
  - Inefficient usage of SCE-MI
  - Behavioral constructs (Not purely-synthesizable)
  - Memory Handling
  - Amount of code running on simulator and emulator
  - Simulator – Emulator synchronizations
Agenda

• Acceleration Concept
• Introduction to SCE-MI and SCE-MI Pipes
• Testbench Architecture and Existing Challenges
• Optimizing SCE-MI pipe usage
• SCE-MI Direct Memory and Function Based Interface
• Optimized Architecture
• Results and Conclusion
Introduction to SCE-MI

• Standard Co-Emulation Modeling Interface (SCE-MI) is an Accellera standard

• Communication interface between BFM and proxy

• Different types of SCE-MI use models:
  – SCE-MI Pipe based interface
  – SCE-MI Direct memory interface (DMI)
  – SCE-MI Function based interface
SCE-MI Pipes

- Salient Features:
  - Unidirectional
  - Batching
  - Buffering
  - Flushing
  - Data shaping
  - Blocking and Non-Blocking constructs
Agenda

• Acceleration Concept
• Introduction to SCE-MI and SCE-MI Pipes
• Testbench Architecture and Existing Challenges
• Optimizing SCE-MI pipe usage
• SCE-MI Direct Memory and Function Based Interface
• Optimized Architecture
• Results and Conclusion
Initial Architecture Challenges

- Input SCEMI Pipes
- Output SCEMI Pipes

Master Agent:
- Proxy Driver
- Proxy Monitor

Slave Agent:
- Proxy Driver
- Proxy Monitor

Emulator:
- BFM Master Driver
- BFM Master Monitor
- BFM Slave Driver
- BFM Slave Monitor

Bottlenecks:
- Multiple SCEMI pipes
- Transmission of small packets frequently
- Async loop to clear pipe contents
Agenda

• Acceleration Concept
• Introduction to SCE-MI and SCE-MI Pipes
• Testbench Architecture and Existing Challenges
• Optimizing SCE-MI pipe usage
• SCE-MI Direct Memory and Function Based Interface
• Optimized Architecture
• Results and Conclusion
Optimizing SCE-MI Pipe Usage

• Merging SCE-MI pipes

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral evals with every SCE-MI access</td>
<td>• Send data via a single SCE-MI pipe.</td>
</tr>
<tr>
<td></td>
<td>• Static SCE-MI pipe - Bit vectors</td>
</tr>
<tr>
<td></td>
<td>• Dynamic SCE-MI pipe - Array of data</td>
</tr>
</tbody>
</table>

Data transfer using multiple pipes

Data transfer using single pipe
## Optimizing SCE-MI Pipe Usage

- **Synchronous access**

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous mode creates significant behavioral evals</td>
<td>Synchronous mode of SCE-MI pipe IS_CLOCKED_INTF = 1</td>
</tr>
</tbody>
</table>

### Code Snippet

**Challenge**

```verilog
c hemi_input_pipe #( .BYTES_PER_ELEMENT(20), .PAYLOAD_MAX_ELEMENTS(1), .VISIBILITY_MODE(2), .IS_CLOCKED_INTF(0) )
inbox (clk);
```

**Solution**

```verilog
c hemi_input_pipe #( .BYTES_PER_ELEMENT(20), .PAYLOAD_MAX_ELEMENTS(1), .VISIBILITY_MODE(2), .IS_CLOCKED_INTF(1) )
inbox (clk);
```
Optimizing SCE-MI Pipe Usage

- Optimized data transfer

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing more than one element per access results in behavioral evals</td>
<td>If the intention is to access 5 Bytes per call then set</td>
</tr>
<tr>
<td>- PAYLOAD_MAX_ELEMENTS = 5;</td>
<td>- PAYLOAD_MAX_ELEMENTS = 1;</td>
</tr>
<tr>
<td>- BYTES_PER_ELEMENT = 1;</td>
<td>- BYTES_PER_ELEMENT = 5;</td>
</tr>
</tbody>
</table>

**Challenge Diagram:**

- **5 Elements**
  - Byte 1
  - Byte 2
  - Byte 3
  - Byte 4
  - Byte 5

- **1 Element**
  - Byte 5
  - Byte 4
  - Byte 3
  - Byte 2
  - Byte 1
Optimizing SCE-MI Pipe Usage

• Minimal Synchronization

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Frequent data transfer causes the emulator to halt frequently</td>
<td>• Accumulate multiple bytes of data and transfer at once</td>
</tr>
<tr>
<td>• Results in degraded acceleration</td>
<td>• Try to buffer the elements</td>
</tr>
<tr>
<td></td>
<td>• Avoid unnecessary flush() usage</td>
</tr>
</tbody>
</table>

Code Snippet

```vhdl
scemi_input_pipe #(  
  .BYTES_PER_ELEMENT(20),  
  .PAYLOAD_MAX_ELEMENTS(1),  
  .BUFFER_MAX_ELEMENTS(10),  
  .VISIBILITY_MODE(2),  
  .IS_CLOCKED_INTF(1) ) inbox (clk);
```
Optimizing SCE-MI Pipe Usage

- Clearing the pipe

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Solution</th>
</tr>
</thead>
</table>
| - Discarding buffer contents on reset  
- No inbuilt functions  
- Increase in step-count | - Use the fastest clock available to synchronously fetch data from pipe |

Code Snippet: Before

```verilog
always@ (posedge clk, posedge rst) begin
  if(rst) begin
    // Statements
    for(int i=0, i<20, i++)
      inbox.receive(1,ve,data,eom);
  end
  else begin
    // statements
  end
end
```

Code Snippet: After

```verilog
always@ (posedge fst_clk, posedge rst) begin
  if(rst) begin
    rst_buff = 1;
    // statements
  end
  else begin
    if(rst_buff && !eom)
      inbox.receive(1,ve,data,eom);
    else
      rst_buff = 0;
  end
end
```
Agenda

• Acceleration Concept
• Introduction to SCE-MI and SCE-MI Pipes
• Testbench Architecture and Existing Challenges
• Optimizing SCE-MI pipe usage
• SCE-MI Direct Memory and Function Based Interface
• Optimized Architecture
• Results and Conclusion
SCE-MI Direct Memory Interface

• Software side interface to perform backdoor read/write operations on hardware side memories

• Types of interfaces
  – Block interface
  – Word interface

• Performance Improvement:
  – HW-SW synchronizations reduced by ~50%
  – TBA run time decreased by >25%
SCE-MI Direct Memory Interface

Backdoor Access

Proxy Agent

Membrane

Simulator

Memory

Emulator

BFM

DUT

Memory

Backdoor Access
C - Backdoor memory access through predefined SCE-MI API

C-membrane writes the memory contents in to the proxy

Invoke a memory read from BFM

SCE-MI Direct Memory Interface

```c
const char* tpath = "path to the proxy where mem_access is defined"
static svScope tbscp = NULL;

extern void mem access (svBitVecVal* mem);
void read_mem()
{
    static void* vmem;
    svBitVecVal mem;
    int rAddr;
    static unsigned int width, depth;
    tbscp = svGetScopeFromName(tpath);
    svSetScope(tbscp);
    vmem = scemi_mem_c_handle("hierarchical_path.mem");
    scemi_mem_get_size(vmem, (unsigned int *) width, (unsigned long long *) depth);
    mem access (vmem, rAddr, depth, mem);
}

export "DPI-C" function mem access;
class proxy_monitor;
    function mem access(input bit [7:0] return_mem [8192]);
        //logic code to process the read BFM memory
endfunction
endclass

import "DPI-C" task read_mem();
module monitor bfm;
    bit [31:0] mem [8192];
    bit [31:0] wr_ptr;
    bit tbs; //Synchronizer between proxy and BFM
    always @(posedge clk) begin
        if (wr_ptr == FULL) begin
            read_mem:
                tbs = ~tbs;
                end
            else wr_ptr = wr_ptr ++;
        end
endmodule
```
SCE-MI Function Based Interface

• Adopts SystemVerilog Direct Programming Interface (DPI) concept.

• End user is required to implement all the functions
  – No built in functions

• Used to configure BFM registers
SCE-MI Function Based Interface

Proxy Agent

Interface

C Membrane

BFM

Circular FIFO

Simulator

Emulator

DUT

Write Pointer

Read Pointer
SCE-MI Function Based Interface

//C-function has the implementation of proxy's write function
//It also invokes the BFM's write function through it
const char* tbpath = "path_to_the_bfm_where_reg_write_function_is_defined"
static svScope tbscp = NULL;
extern void reg_bfm_wr( svBitVecVal* reg_data);
void write_bfm_reg c( svBitVecVal* reg_data){
    tbscp = svGetScopeFromeName(tbpath);
    svSetScope(tbscp);
    reg_bfm_wr(reg_data);
}

//Proxy code, here is where the write data is passed to the C-Function which is to
//be written on to the BFM
import "DPI-C" context
write_bfm_write_c = function void write_bfm_reg(bit[31:0] data);
class proxy_driver;
    bit[31:0] data_wr;
    task reconfig;
    write_bfm_reg(data_wr);
endtask
endclass

//BFM code, here is where the implementation of the write function resides
//It is invoked by the C-interface
module BFM;
export "DPI-C" function
    reg_bfm_wr;
    bit[31:0] ctl_reg;
    function void reg_bfm_wr(input bit[31:0] wr_data);
        ctl_reg = wr_data;
    endfunction
endmodule
Trimming Development Time

• Factors
  – Increase in effort and time to implement complex functionality in a synthesizable format.
  – Increase in area occupied by the BFM on the emulator.

• Implement complex functionality that is utilized occasionally as a method in the C membrane and import/invoke from the BFM.
Agenda

- Acceleration Concept
- Introduction to SCE-MI and SCE-MI Pipes
- Testbench Architecture and Existing Challenges
- Optimizing SCE-MI pipe usage
- SCE-MI Direct Memory and Function Based Interface
- Optimized Architecture
- Results and Conclusion
Optimized Architecture

SCEMI Function Based Interface
- To configure BFM registers
- To update proxy about events in BFM

SCEMI Direct Memory Interface
- To write/read data from BFM memory

Pipes are removed
Agenda

• Acceleration Concept
• Introduction to SCE-MI and SCE-MI Pipes
• Testbench Architecture and Existing Challenges
• Optimizing SCE-MI pipe usage
• SCE-MI Direct Memory and Function Based Interface
• Optimized Architecture
• Results and Conclusion
Results

- **TBA Performance improvement**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Gate count</th>
<th>Bevals</th>
<th>HW-SW Sync</th>
<th>TBA Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe only</td>
<td>~2 M</td>
<td>18,299,173</td>
<td>4,728,998</td>
<td>~60 min</td>
</tr>
<tr>
<td>Pipes only (Optimized)</td>
<td>~2 M</td>
<td>2,859,320</td>
<td>1,284,765</td>
<td>~35 min</td>
</tr>
<tr>
<td>DMI + Function based Interface</td>
<td>~1.5 M</td>
<td>218</td>
<td>9,871</td>
<td>~4 min</td>
</tr>
</tbody>
</table>

- **Simulation v/s TBA run-time comparison for usecase**

<table>
<thead>
<tr>
<th>Simulation Time</th>
<th>TBA Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>~360min</td>
<td>~8min</td>
</tr>
</tbody>
</table>
Conclusion

• SCE-MI has multiple interfaces for different use cases
  – SCE-MI DMI used for backdoor read/write operation
  – SCE-MI Function Based Interface used to write into registers
  – SCE-MI Pipes are intended for streaming, variable length messaging, etc.

• HW-SW sync and Bevals could greatly deteriorate the performance of emulator.
Conclusion

• Efficient usage of SCE-MI helps in leveraging maximum performance from emulator resulting in handsome SPEED-UP.

• HW-SW syncs generated using vendor proprietary implementation
  – Number of syncs are kept to minimum.
Future Work

- Porting to SOC environment and analyze the performance
- Explore UVM registers support described in SCEMI v2.3
Acknowledgements

- David Brownell, ADI
- Anilkumar T. S, Cadence
- Dr. Hans van der Schoot, Mentor Graphics
Questions