2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Having Your Cake and Eating It Too
Programming UVM Sequences with C Code

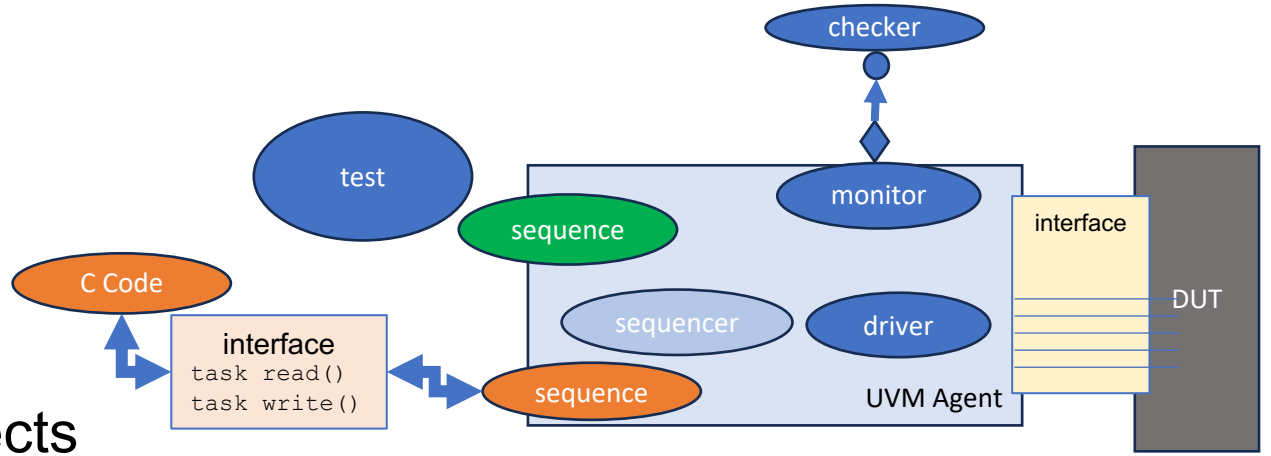
Rich Edelman & Tomoki Watanabe
Siemens EDA & Siemens EDA Japan

SIEMENS

accellera SYSTEMS INITIATIVE

Motivation

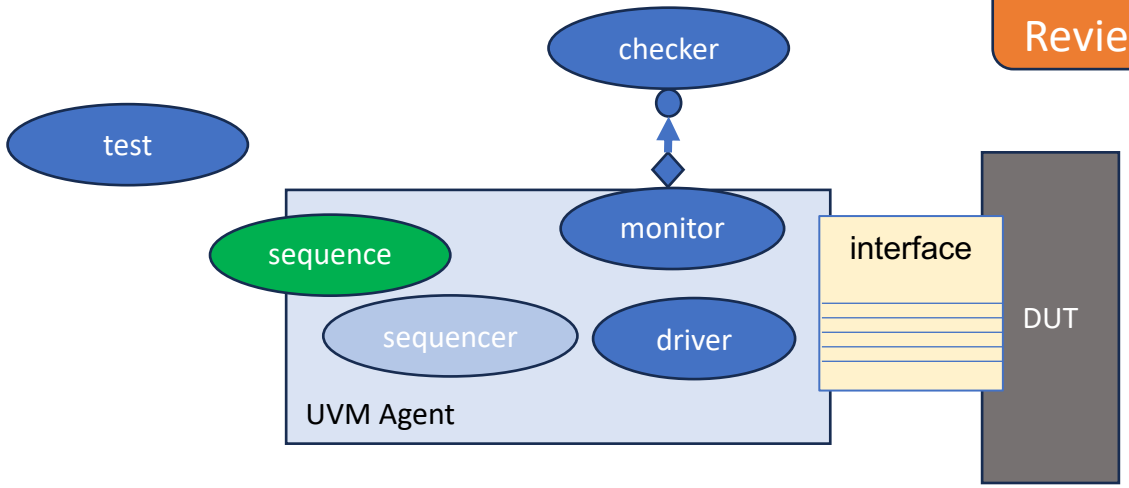
- Reuse our UVM testbench as-is
- Add the ability to have C code
  - Generate traffic
  - Run a C test program from the architects
  - Reuse architects' performance C tests
- Solution #1:
  - Use DPI-C
  - Write C code calling tasks in SystemVerilog
  - But the tasks need to be aware of WHICH sequence is being used on which interface in the UVM testbench
- Problem:
  - We cannot “host” the DPI-C calls in a class
  - How to connect the class to C code.
- Solution #2:
  - The SystemVerilog Interface



The UVM

- Typical UVM
  - SystemVerilog
    - test
    - sequence
    - sequencer
    - driver
    - monitor
    - interface
    - checker
    - …

Quick Review



What does a '<u>test</u>' do? – It's a "coordinator of programs" to run
- Orchestrates starting "sequences"
- Checks results?
- Has a timeout?

What does a '<u>sequence</u>' do? – It's a "program" to run
- Creates transactions
- Sends transactions to the sequencer and on to the driver
- Checks results?

What does a '<u>driver</u>' do? – It's a "pin wiggler"
- Receives transactions
- Turns the transaction into pin wiggles on the interface
- Sends results back to the sequence via the transaction

Using DPI-C

DPI in the LRM
- Chapter H – “DPI C Layer” (33 pages)
- Chapter 35 – “Direct Programming Interface” (15 pages)

Quick Review

- Export SystemVerilog code for C to call
- Import C code for SystemVerilog to call
- Rules…
  - DPI-C can be defined inside

H.9.2 Context of imported and exported tasks and functions

DPI imported and exported tasks and functions can be declared in a **`module`**, **`program`**, **`interface`**, **`package`**, compilation unit scope, or **`generate`** declarative scope.

  - But NOT in classes

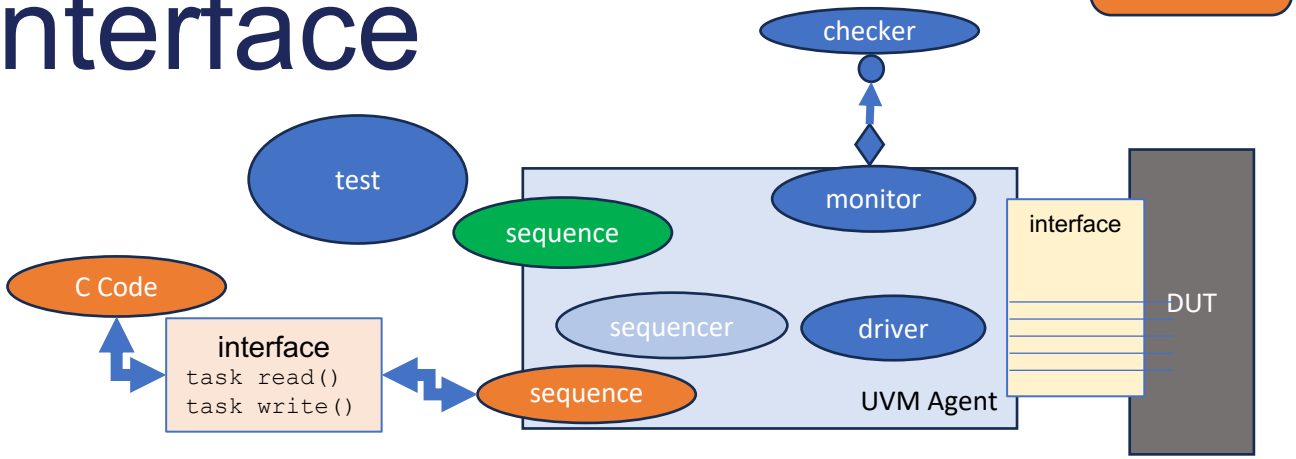
Quick Review

The SystemVerilog Interface

```
interface zinterface();
  export "DPI-C" task read;
  export "DPI-C" task write;
  import "DPI-C" context task test_program1(...);

  task read(int index, int addr, output int data);
    ...
  endtask

  task write(int index, int addr, int data);
    ...
  endtask
endinterface
```



- SystemVerilog interfaces are closely related to modules
  - They can be instantiated - usually thought of as collections of wires
  - A “virtual interface” handle can be passed to a class
    - The class has a handle to the interface and can call tasks and functions in the interface
  - “Regular” tasks and functions can be defined inside them
  - DPI-C tasks and functions CAN be defined inside them

The C code – The Test Program

```
interface zinterface();
  export "DPI-C" task read;
  export "DPI-C" task write;
  import "DPI-C" context zinterface_start_test_program1 =
    task start_test_program1(int index, string name, int start_addr);
```

```
#include <stdio.h>
#include "dpiheader.h"

int
zinterface_start_test_program1(int index, const char *name, int start_addr) {
    int addr, data;

    // Repeat 10 times, changing the data
    for (dataloops = 0; dataloops < 10; dataloops++) {
      // Repeat 10 times - writing 10, and reading 10
      for (loops = 0; loops < 10; loops++) {
        for (addr = start_addr; addr < start_addr+10; addr++) {
          data = ...;
          write(index, addr, data);
        }

        for (addr = start_addr; addr < start_addr+10; addr++) {
          read(index, addr, &data);

          if (data != addr + 1000 + dataloops) {
            printf("C: ...ERROR  READ (%0d, %0d) <%s> [wrote: %d, read %d]  ¥n",
              addr, data, name, data, addr + 1000 + dataloops);
          }
        }
        start_addr += 10;
      }
    }
    return 0;
}
```

Repeat 10 times, changing 'data' each time

Cycle through the addresses, do 'write'

Cycle through the addresses, do 'read' and compare

The C code – The Test Program - Executing

Pseudo-code
Given a start_address, do the following steps:

Repeat 10 times with different ‘data’ values
- Repeat 10 times
  - Repeat 10 times
    - WRITE data at ‘address’
    - increment ‘address’ by 1
  - Reset address
  - Repeat 10 times
    - READ data at ‘address’
    - Compare Actual == Expected
    - increment ‘address’ by 1
  - Increment start_address by 10

```
# C: ...executed WRITE(200, 1200) <thread2>
# C: ...executed WRITE(201, 1201) <thread2>
# C: ...executed WRITE(202, 1202) <thread2>
# C: ...executed WRITE(203, 1203) <thread2>
# C: ...executed WRITE(204, 1204) <thread2>
# C: ...executed WRITE(205, 1205) <thread2>
# C: ...executed WRITE(206, 1206) <thread2>
# C: ...executed WRITE(207, 1207) <thread2>
# C: ...executed WRITE(208, 1208) <thread2>
# C: ...executed WRITE(209, 1209) <thread2>
# C: ...executed READ (200, 1200) <thread2>
# C: ...executed READ (201, 1201) <thread2>
# C: ...executed READ (202, 1202) <thread2>
# C: ...executed READ (203, 1203) <thread2>
# C: ...executed READ (204, 1204) <thread2>
# C: ...executed READ (205, 1205) <thread2>
# C: ...executed READ (206, 1206) <thread2>
# C: ...executed READ (207, 1207) <thread2>
# C: ...executed READ (208, 1208) <thread2>
# C: ...executed READ (209, 1209) <thread2>
# C: ...executed WRITE(210, 1210) <thread2>
# C: ...executed WRITE(211, 1211) <thread2>
# C: ...executed WRITE(212, 1212) <thread2>
# C: ...executed WRITE(213, 1213) <thread2>
# C: ...executed WRITE(214, 1214) <thread2>
# C: ...executed WRITE(215, 1215) <thread2>
# C: ...executed WRITE(216, 1216) <thread2>
# C: ...executed WRITE(217, 1217) <thread2>
# C: ...executed WRITE(218, 1218) <thread2>
# C: ...executed WRITE(219, 1219) <thread2>
# C: ...executed READ (210, 1210) <thread2>
# C: ...executed READ (211, 1211) <thread2>
# C: ...executed READ (212, 1212) <thread2>
```

Just thread2

```
# C: ...executed READ (125, 1134) <thread1>
# C: ...executed READ (126, 1135) <thread1>
# C: ...executed READ (127, 1136) <thread1>
# C: ...executed READ (208, 1217) <thread2>
# C: ...executed WRITE(410, 1419) <thread4>
# C: ...executed WRITE(331, 1340) <thread3>
# C: ...executed READ (128, 1137) <thread1>
# C: ...executed READ (129, 1138) <thread1>
# C: ...executed READ (330, 1339) <thread3>
# C: ...executed WRITE(411, 1420) <thread4>
# C: ...executed READ (209, 1218) <thread2>
# C: ...executed WRITE(130, 1139) <thread1>
# C: ...executed WRITE(333, 1342) <thread3>
# C: ...executed WRITE(210, 1219) <thread2>
# C: ...executed WRITE(412, 1421) <thread4>
# C: ...executed WRITE(334, 1343) <thread3>
# C: ...executed WRITE(131, 1140) <thread1>
# C: ...executed WRITE(413, 1422) <thread4>
# C: ...executed WRITE(211, 1220) <thread2>
# C: ...executed WRITE(335, 1344) <thread3>
# C: ...executed WRITE(414, 1423) <thread4>
# C: ...executed WRITE(336, 1345) <thread3>
# C: ...executed WRITE(132, 1141) <thread1>
# C: ...executed WRITE(212, 1221) <thread2>
```

All the threads

Quick Review

Running a Sequence

- The ‘test’ might start a sequence
  - And the body() task runs – 1000 transactions are created and “started”

```
class test extends uvm_test;
  `uvm_component_utils(test)

  env e1;
  seq s1;

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    s1 = seq::type_id::create("s1");

    s1.start(e1.a.sqr);

    phase.drop_objection(this);
  endtask
endclass
```

```
class seq extends uvm_sequence#(transaction);
  `uvm_object_utils(seq)

  transaction t;

  task body();
    for (int i = 0; i < 1000; i++) begin
      t = transaction::type_id::create(name);
      start_item(t);
      finish_item(t);
    end
  endtask
endclass
```

Adding Specialized Tasks to the Sequence

```
class seq extends uvm_sequence#(transaction);
  `uvm_object_utils(seq)

  transaction t;


  task body();
    for (int i = 0; i < 1000; i++) begin
      t = transaction::type_id::create(name);
      start_item(t);
      finish_item(t);
    end
  endtask
```

```
 task read(bit [31:0] addr, output bit [31:0]data);
   t = transaction::type_id::create("read");
   t.rw = READ;
   t.addr = addr;
   t.data = 0;

    start_item(t);
    finish_item(t);

    data = t.data;
 endtask

 task write(bit [31:0] addr, bit [31:0]data);
   t = transaction::type_id::create("write");
   t.rw = WRITE;
   t.addr = addr;
   t.data = data;

   start_item(t);
   finish_item(t);
 endtask
endclass
```

Zombie Sequence

```
class zombie_seq extends seq;
  `uvm_object_utils(zombie_seq)

  transaction t;
  bit done;

  task body();

    wait (done == 1);



  endtask
```

```
 task read(bit [31:0] addr, output bit [31:0]data);
   t = transaction::type_id::create("read");
   t.rw = READ;
   t.addr = addr;
   t.data = 0;

    start_item(t);
    finish_item(t);

    data = t.data;
 endtask

 task write(bit [31:0] addr, bit [31:0]data);
   t = transaction::type_id::create("write");
   t.rw = WRITE;
   t.addr = addr;
   t.data = data;

   start_item(t);
   finish_item(t);
 endtask
endclass
```

This is the “API” that C coders will use – make it as needed – this is a very simple one

Running a Zombie Sequence

- The zombie sequence gets started by the test just like a “regular” sequence

```
class test extends uvm_test;
  `uvm_component_utils(test)

  env e1;
  zombie_seq zs1;

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    zs1 = seq::type_id::create("s1");

    zs1.start(e1.a.sqr);

    phase.drop_objection(this);
  endtask
endclass
```

```
class zombie_seq extends seq;
  `uvm_object_utils(zombie_seq)

  transaction t;
  bit done;

  task body();

    wait (done == 1);



  endtask
```

Defining a DPI-C interface

- A special “trampoline” to bounce between C and SV

```
interface zinterface();
  uvm_object registered_seq[int];

  export "DPI-C" task read;
  export "DPI-C" task write;

  import "DPI-C" context zinterface_start_test_program1 =
    task start_test_program1(int index, string name, int start_addr);
...
```

Defined in SV – in this interface

Defined in C – this is the C test program

DPI-C interface 2

- A function is defined to add SV class handles to an array
- Look-up by simple index
- Each C thread has an index number

```
interface zinterface();
  uvm_object registered_seq[int];
...
  function void register(int index, uvm_object seq);
    zinterface_zombieseq zsq;
    registered_seq[index] = seq;
    $cast(zsq, seq);
    zsq.vif = interface::self(); // Extension to the LRM
  endfunction
 ...
```

DPI-C interface 3

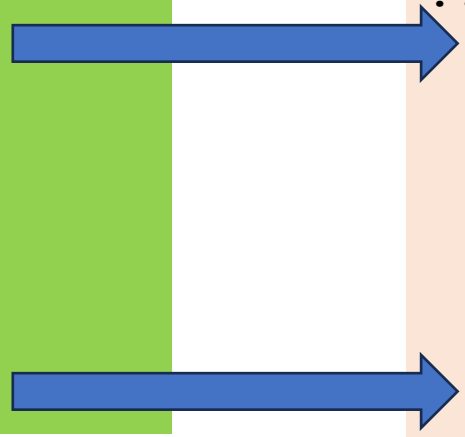
C Code

…

read()

…

write()



```
interface zinterface();
  uvm_object registered_seq[int];
...
  task read(int index, int addr, output int data);
    zinterface_zombieseq zsq;
    $cast(zsq, registered_seq[index]);
    zsq.read(addr, data);
  endtask

  task write(int index, int addr, int data);
    zinterface_zombieseq zsq;
    $cast(zsq, registered_seq[index]);
    zsq.write(addr, data);
  endtask
endinterface
```

Changes to the original top

- Instantiate the special zombie interface for the ‘zinterface’ API
- Put the virtual interface handle into the config db

```
import uvm_pkg::*;
`include "uvm_macros.svh"

import ip_pkg::*;

module top();
  memory_interface memory_interface_instance();

  zinterface I_zinterface1();

  initial begin
    uvm_config_db#(virtual memory_interface)::set(
      uvm_root::get(), "*", "memory_interface", memory_interface_instance);

    uvm_config_db#(virtual zinterface)::set(
      uvm_root::get(), "*", "zinterface", I_zinterface1);

    run_test();
  end
endmodule
```

Instantiate the interface

Put the interface in the config database

C Code calling the interface tasks

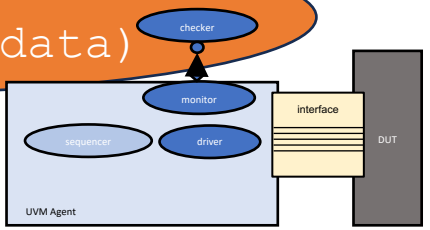
- This is just “DPI-C” export

C Code
`read(thread,addr,data)`

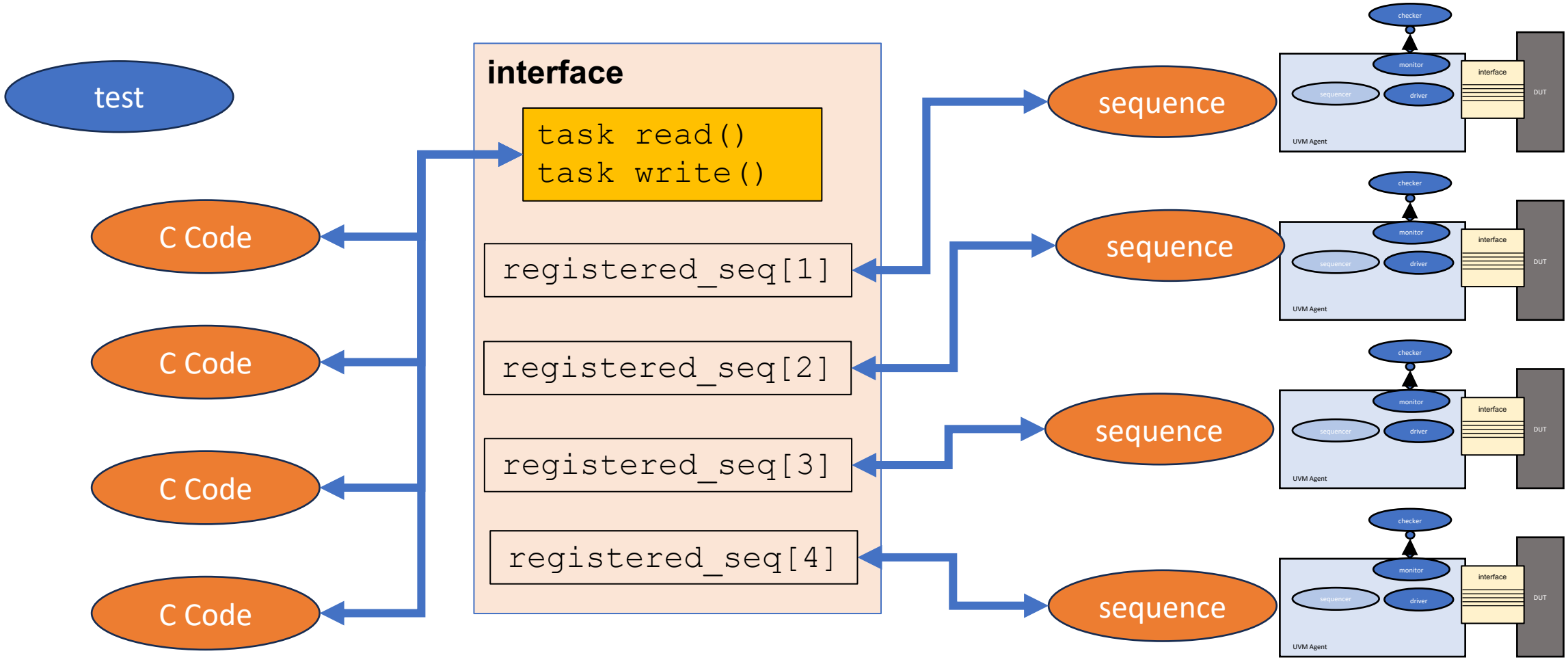
interface

```
task read(int index, int addr, output int data);
    zinterface_zombieseq zsq;
    $cast(zsq, registered_seq[index]);
    zsq.read(addr, data);
endtask
```

sequence
task `read(addr,data)`



The testbench from the C point of view



Changes to the original test

```
class test extends uvm_test;
  `uvm_component_utils(test)

  env e1, e2, e3, e4;
  seq s1, s2, s3, s4;

  zinterface_zombieseq zs1, zs2, zs3, zs4;

  virtual zinterface zif;

  function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
    e2 = env::type_id::create("e2", this);
    e3 = env::type_id::create("e3", this);
    e4 = env::type_id::create("e4", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    s1 = seq::type_id::create("s1");
    s2 = seq::type_id::create("s2");
    s3 = seq::type_id::create("s3");
    s4 = seq::type_id::create("s4");

    if (!uvm_config_db#(virtual zinterface)::get(
        this, "*", "zinterface", zif))
      `uvm_fatal(get_type_name(),
        "cannot find ZINTERFACE INSTANCE")
```

Declare the zombie sequence handles and the virtual interface

Lookup the interface in the config db

Construct the zombie sequences

```
    zs1 = zinterface_zombies                    );
    zs2 = zinterface_zombieseq::type_id::create("zs2");
    zs3 = zinterface_zombieseq::type_id::create("zs3");
    zs4 = zinterface_zombieseq::type_id::create("zs4");

    zif.register(1, zs1);
    zif.register(2, zs2);
    zif.register(3, zs3);
    zif.register(4, zs4);

    fork
      zs1.start(e1.a.sqr);
      zs2.start(e2.a.sqr);
      zs3.start(e3.a.sqr);
      zs4.start(e4.a.sqr);
    join_none

    fork
      s1.start(e1.a.sqr);
      s2.start(e2.a.sqr);
      s3.start(e3.a.sqr);
      s4.start(e4.a.sqr);
    join

    fork
      zs1.start_test_program1(1, "thread1", 100);
      zs2.start_test_program1(2, "thread2", 200);
      zs3.start_test_program1(3, "thread3", 300);
      zs4.start_test_program1(4, "thread4", 400);
    join

    phase.drop_objection(this);
  endtask
endclass
```

Register with the interface

Start the zombie sequences

Start the C code test programs

Results – 4 “zombie” threads / streams

Conclusion

- A solution was shared to demonstrate easy additions to an existing UVM TB that will enable easy C code integration
- Standard SystemVerilog coding – LRM compliant
- Enables C coders access to write tests

- You can have BOTH a UVM testbench and a C test program

Questions?

- Source code is available – please email rich.edelman@siemens.com