OpenHW Group

Proven Processor IP

UVM Agents in CORE-V-VERIF

Mike Thompson

mike@openhwgroup.org

February, 2022

Objectives

- These slides are one of a set of presentations designed to provided an overview of the CORE-V-VERIF SV/UVM environment:
  - Understanding the ENV architecture.
  - UVM Agents in CORE-V-VERIF. ← This presentation
  - Writing and Running Testcases

- Goals of these presentations:
  - Communicate the current state of CORE-V-VERIF.
  - Create consensus around standardized specifications, architecture and implementation of UVM Agents used in CORE-V-VERIF.
    - Ultimately this should lead to written coding standards.

Credits

- Thanks to the following members of the OpenHW Group for their significant contributions to this effort:
  - Steve Richmond, Silicon Labs
  - David Poulin, Datum Technology Corporation

UVM Agents in CORE-V-VERIF

UVM Agents in CORE-V-VERIF

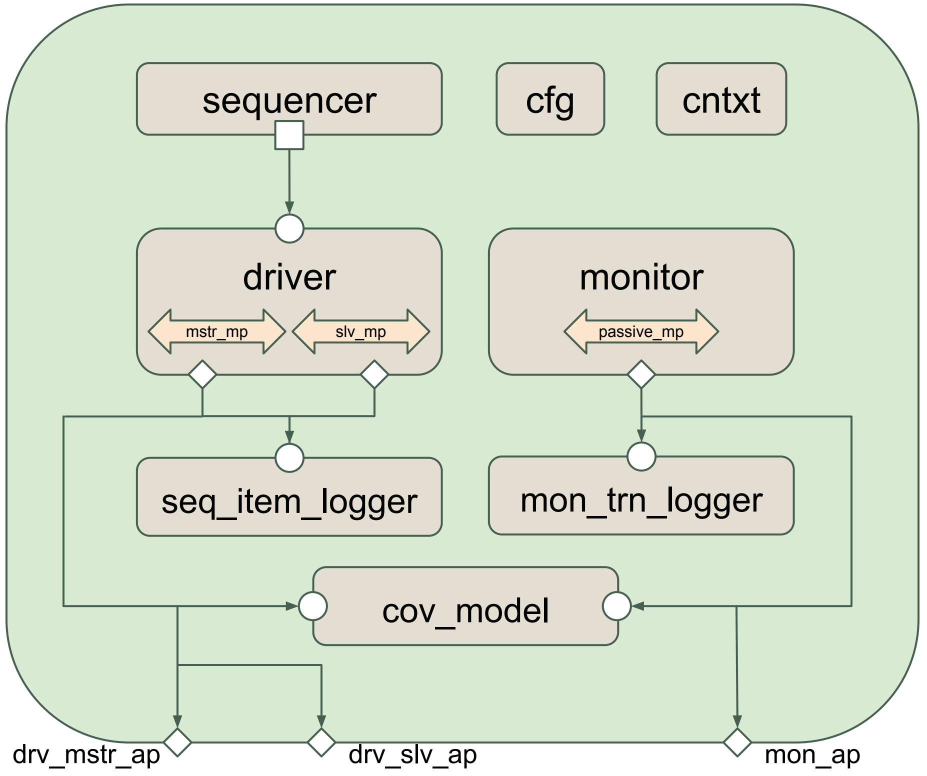
- Starting in late 2021, new Agents in CORE-V-VERIF should be generated from a set of code templates:
    - This is not (yet) a strictly enforced rule.
- An objective of this presentation is to communicate the structure of these templates.
- Value of templating:
    - Create a common “developer experience”.
    - Adherence to CORE-V-VERIF coding guidelines:
        - In GitHub at core-v-verif/docs/CodingStyleGuidelines.md

Use of the Templates

- On GitHub: Datum-Technology-Corporation/mio_ip_base
  - A set of UVM code generators from the ***Moore.io*** IP project.
- Easiest way to use this is with the ***new-agent*** Make target:
  - $ cd $CORE-V-VERIF/cv32e40p/sim/uvmt
  - $ make new-agent
    - You will be prompted for agent-name, etc.
  - A compilable shell of for *agent_name* will be generated in $CORE-V-VERIF/lib/uvm_agents/uvma_<*agent_name*>
    - Includes example instantiation and sequence
    - Grep for TODO to find points where agent-specific code must be added.

Structure of template-generated UVM Agent

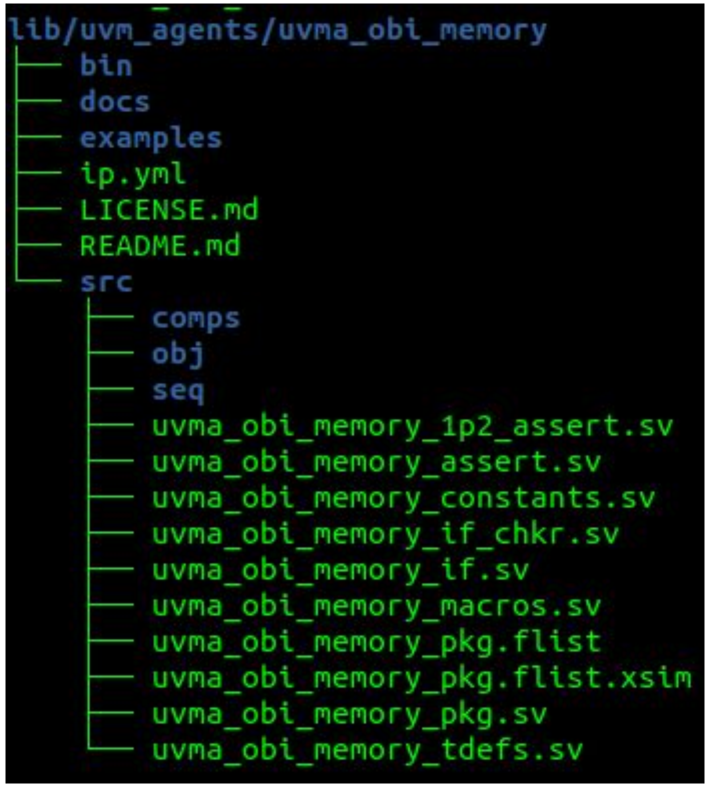
- Recognizable UVM Agent:
  - Compiles “out-of-the box” (does not *do* anything).
- Unique features:
  - Context object.
  - Driver and Monitor use interface modports (these modports must be implemented by the SV Interface).
- Common structure:
  - Can lead to lots of unimplemented classes (e.g. loggers)



Example Agent: UVMA_OBI_MEMORY

- This Agent will be used as a working example throughout this presentation.
- The UVMA_OBI_MEMORY agent was developed as a replacement for the "mm_ram" SystemVerilog module used by CV32E40P:
  - Improves re-use across projects.
  - Improved control and randomization of OBI cycle timing.
  - Added support for OBI v1.2.
  - Easy to add new virtual peripherals.
  - Currently used by CV32E40P, CV32E40X and CV32E40S and planned to be used by CV32E20.
- If you understand the architecture and implementation of obi_mem_agt, you'll understand much about the architecture and implementation of core-v-verif.

UVMA_OBI_MEMORY File Structure

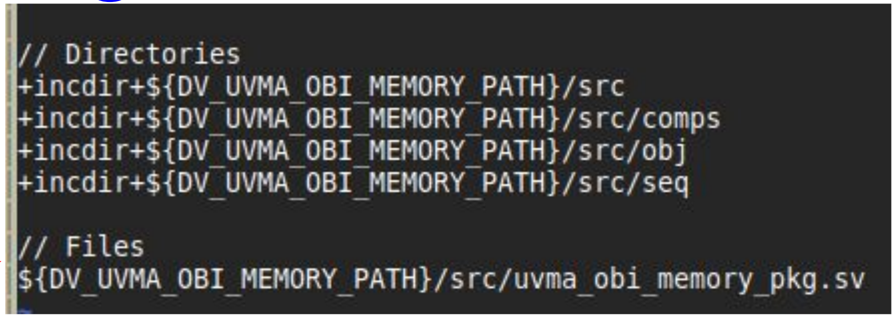


```
lib/uvm_agents/uvma_obi_memory
├── bin
├── docs
├── examples
├── ip.yml
├── LICENSE.md
├── README.md
└── src
    ├── comps
    ├── obj
    ├── seq
    ├── uvma_obi_memory_1p2_assert.sv
    ├── uvma_obi_memory_assert.sv
    ├── uvma_obi_memory_constants.sv
    ├── uvma_obi_memory_if_chkr.sv
    ├── uvma_obi_memory_if.sv
    ├── uvma_obi_memory_macros.sv
    ├── uvma_obi_memory_pkg.flist
    ├── uvma_obi_memory_pkg.flist.xsim
    ├── uvma_obi_memory_pkg.sv
    └── uvma_obi_memory_tdefs.sv
```

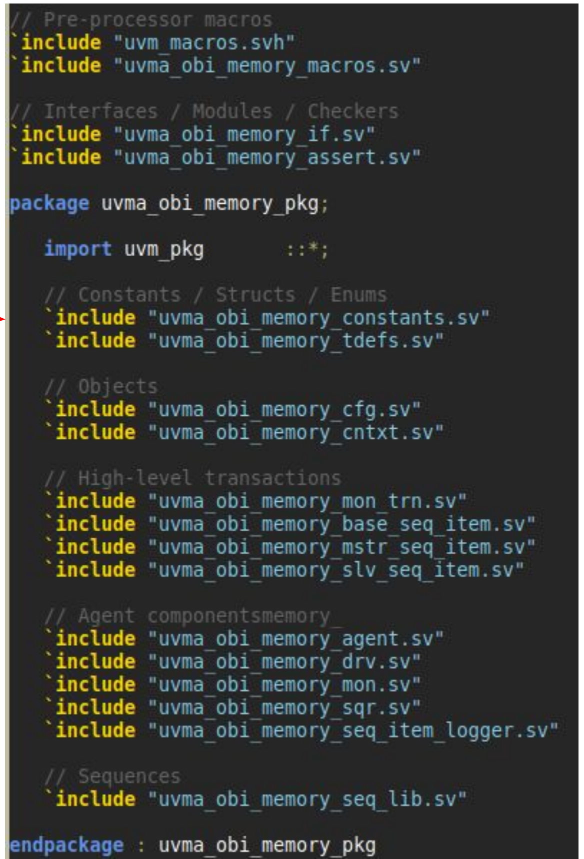
- Structure of the uvma_obi_memory was generated using a template:
  - Template is intended to be fully generic and is based on the structure of agents in the UVM library.
    - **ip.yml** directory has been depreciated.
  - Code structure under src directory:
    - **src/comps**: environment “components” (e.g. agent, driver)
    - **src/obj**: component objects (e.g. cfg)
    - **src/seq**: sequence library for the agent
    - **uvm_*.sv**: pkg files, tdefs, macros, etc.
  - Directory structure is captured in the README.md
    - Implementation details should be added by Contributor.

Each Agent is a stand-alone Package

- Two files in the **src** directory define the Agent in a way that makes it easy to add to the environment:
  - **src/uvma_<name>_pkg.flist:** The FLIST is the manifest for the package.
  - **src/uvma_<name>_pkg.sv:** The package file `includes all the required sources for the package.



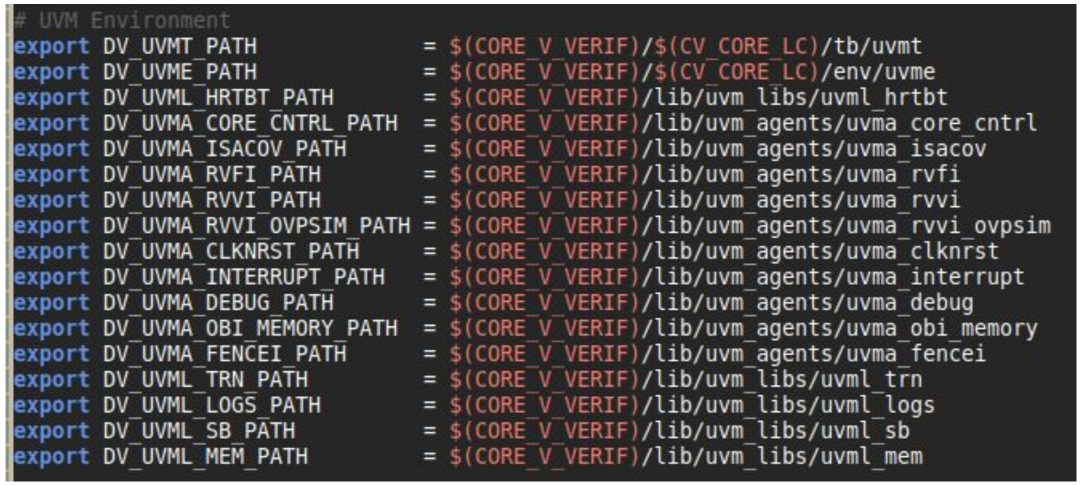
```
// Directories
+incdir+${DV_UVMA_OBI_MEMORY_PATH}/src
+incdir+${DV_UVMA_OBI_MEMORY_PATH}/src/comps
+incdir+${DV_UVMA_OBI_MEMORY_PATH}/src/obj
+incdir+${DV_UVMA_OBI_MEMORY_PATH}/src/seq

// Files
${DV_UVMA_OBI_MEMORY_PATH}/src/uvma_obi_memory_pkg.sv
```



```
// Pre-processor macros
`include "uvm_macros.svh"
`include "uvma_obi_memory_macros.sv"

// Interfaces / Modules / Checkers
`include "uvma_obi_memory_if.sv"
`include "uvma_obi_memory_assert.sv"

package uvma_obi_memory_pkg;

   import uvm_pkg        ::*;

   // Constants / Structs / Enums
   `include "uvma_obi_memory_constants.sv"
   `include "uvma_obi_memory_tdefs.sv"

   // Objects
   `include "uvma_obi_memory_cfg.sv"
   `include "uvma_obi_memory_cntxt.sv"

   // High-level transactions
   `include "uvma_obi_memory_mon_trn.sv"
   `include "uvma_obi_memory_base_seq_item.sv"
   `include "uvma_obi_memory_mstr_seq_item.sv"
   `include "uvma_obi_memory_slv_seq_item.sv"

   // Agent componentsmemory
   `include "uvma_obi_memory_agent.sv"
   `include "uvma_obi_memory_drv.sv"
   `include "uvma_obi_memory_mon.sv"
   `include "uvma_obi_memory_sqr.sv"
   `include "uvma_obi_memory_seq_item_logger.sv"

   // Sequences
   `include "uvma_obi_memory_seq_lib.sv"

endpackage : uvma_obi_memory_pkg
```

Adding Agent to your ENV

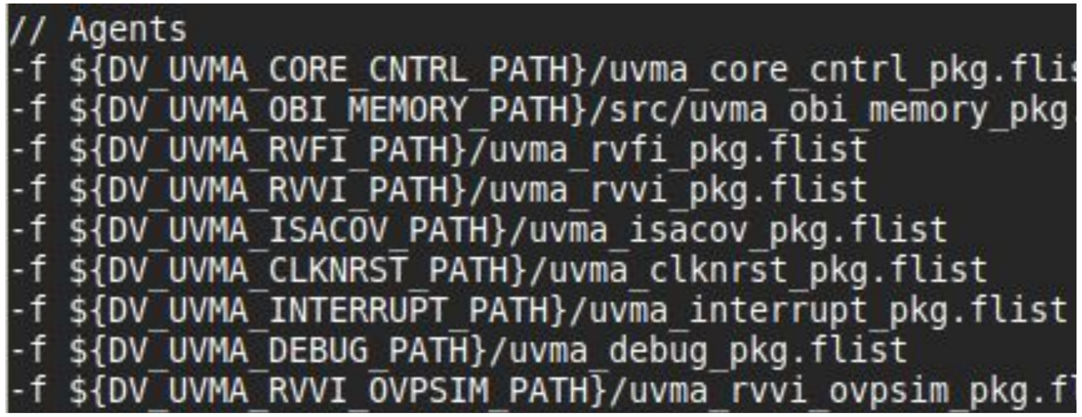
- Define shell variable to point to the agent in **mk/uvmt/uvmt.mk**



```
# UVM Environment
export DV_UVMT_PATH             = $(CORE_V_VERIF)/$(CV_CORE_LC)/tb/uvmt
export DV_UVME_PATH             = $(CORE_V_VERIF)/$(CV_CORE_LC)/env/uvme
export DV_UVML_HRTBT_PATH       = $(CORE_V_VERIF)/lib/uvm_libs/uvml_hrtbt
export DV_UVMA_CORE_CNTRL_PATH  = $(CORE_V_VERIF)/lib/uvm_agents/uvma_core_cntrl
export DV_UVMA_ISACOV_PATH      = $(CORE_V_VERIF)/lib/uvm_agents/uvma_isacov
export DV_UVMA_RVFI_PATH        = $(CORE_V_VERIF)/lib/uvm_agents/uvma_rvfi
export DV_UVMA_RVVI_PATH        = $(CORE_V_VERIF)/lib/uvm_agents/uvma_rvvi
export DV_UVMA_RVVI_OVPSIM_PATH = $(CORE_V_VERIF)/lib/uvm_agents/uvma_rvvi_ovpsim
export DV_UVMA_CLKNRST_PATH     = $(CORE_V_VERIF)/lib/uvm_agents/uvma_clknrst
export DV_UVMA_INTERRUPT_PATH   = $(CORE_V_VERIF)/lib/uvm_agents/uvma_interrupt
export DV_UVMA_DEBUG_PATH       = $(CORE_V_VERIF)/lib/uvm_agents/uvma_debug
export DV_UVMA_OBI_MEMORY_PATH  = $(CORE_V_VERIF)/lib/uvm_agents/uvma_obi_memory
export DV_UVMA_FENCEI_PATH      = $(CORE_V_VERIF)/lib/uvm_agents/uvma_fencei
export DV_UVML_TRN_PATH         = $(CORE_V_VERIF)/lib/uvm_libs/uvml_trn
export DV_UVML_LOGS_PATH        = $(CORE_V_VERIF)/lib/uvm_libs/uvml_logs
export DV_UVML_SB_PATH          = $(CORE_V_VERIF)/lib/uvm_libs/uvml_sb
export DV_UVML_MEM_PATH         = $(CORE_V_VERIF)/lib/uvm_libs/uvml_mem
```

- Add Agent manifest to TB manifest:

  e.g.: **cv32e40x/tb/uvmt/uvmt_cv32e40x.flist**



```
// Agents
-f ${DV_UVMA_CORE_CNTRL_PATH}/uvma_core_cntrl_pkg.fli
-f ${DV_UVMA_OBI_MEMORY_PATH}/src/uvma_obi_memory_pkg
-f ${DV_UVMA_RVFI_PATH}/uvma_rvfi_pkg.flist
-f ${DV_UVMA_RVVI_PATH}/uvma_rvvi_pkg.flist
-f ${DV_UVMA_ISACOV_PATH}/uvma_isacov_pkg.flist
-f ${DV_UVMA_CLKNRST_PATH}/uvma_clknrst_pkg.flist
-f ${DV_UVMA_INTERRUPT_PATH}/uvma_interrupt_pkg.flist
-f ${DV_UVMA_DEBUG_PATH}/uvma_debug_pkg.flist
-f ${DV_UVMA_RVVI_OVPSIM_PATH}/uvma_rvvi_ovpsim_pkg.f
```

Architecture of a UVM Agent in core-v-verif

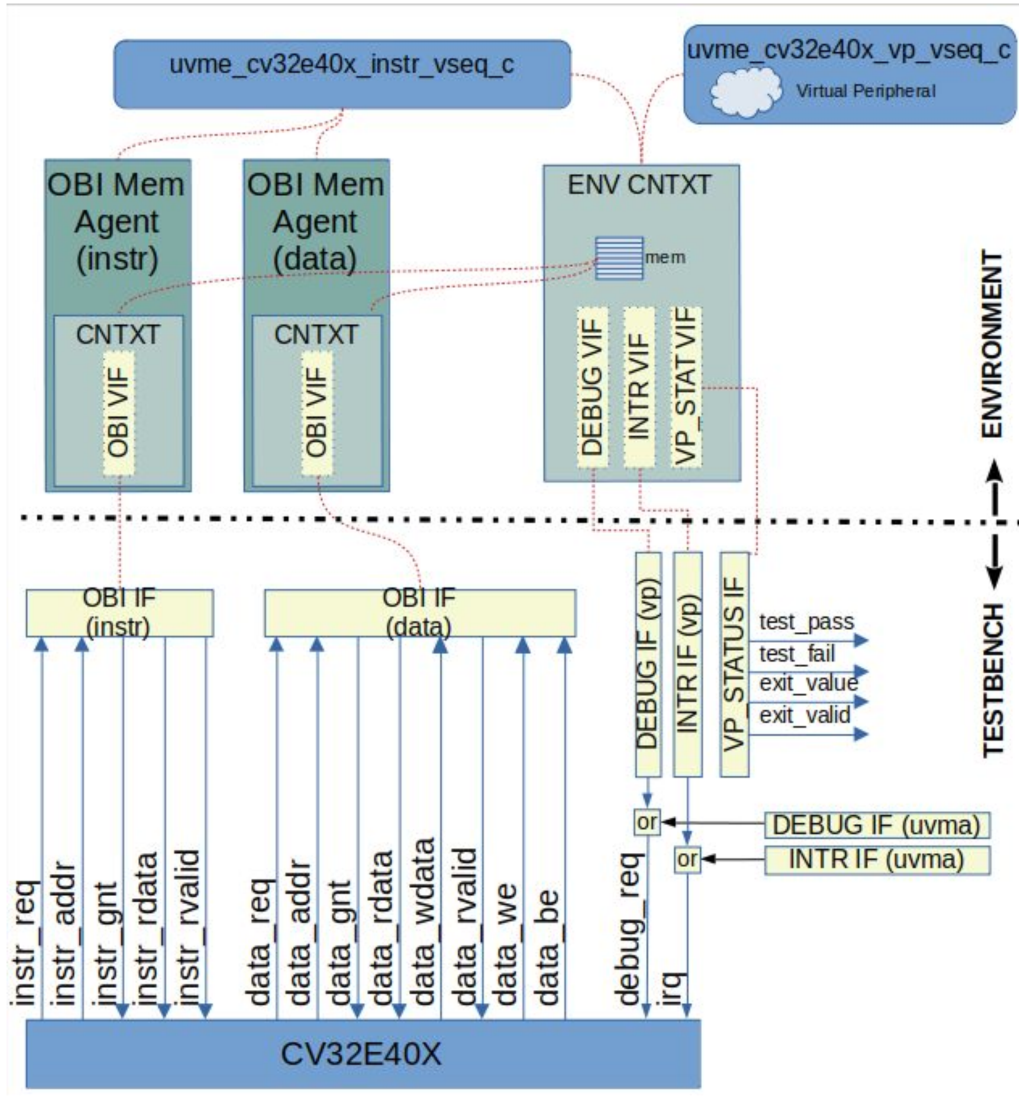
- In core-v-verif, agents are recognizable UVM Agents with some unique implementation details:
    - Structure is driven by templated code generation:
        - Lots of “boilerplate” code.
    - Agents are “as dumb as possible”:
        - Intelligence is in (virtual) sequences.
        - Maximizes reuse potential of Agents.
    - Agents have both a <u>configuration</u> and a <u>context</u> data member:
        - Cfg is familiar to most UVM developers.
        - Cntxt contains VIFs and all state variables of agent, important for reset and sequences.

External Dependencies

- SystemVerilog Interfaces:
  - Each Interface used by a CORE-V-VERIF Agent is assumed to support modports for specific uses:
    - **mstr_mp**: used by Driver to drive request transactions onto the bus.
    - **slv_mp**: used by Driver to fetch response transactions off the bus.
    - **monitor_mp**: used by Monitor to record all transactions on the bus.
- Sequence Items:
  - Distinct uvm_sequence_item for request and response transactions.

OBI_MEMORY_AGENT in CV32E40X ENV

- Two instances of obi_memory_agent:
  - Instruction bus (read only)
  - Data bus (read/write)
- Agent Context has OBI VIF member which is ***set()*** in the test bench and ***get()*** in the agent.
- Environment context members:
  - VIFs for debug, interrupt and status.
  - Sparse memory model for all memory segments (instruction, data, debug, etc.)
- As much as is practical, operation of agents is determined by sequences.
  - Agents are “as dumb as possible”.



UVM Agent Implementation Details

Template-Driven Implementation

- https://github.com/Datum-Technology-Corporation/mio_ip_base
    - A set of XML templates for UVM components.
    - Eclipse-based SystemVerilog code generator.

- UVMA_OBI_MEMORY will be used as a working example of the code generated from these templates.

Sequence Items

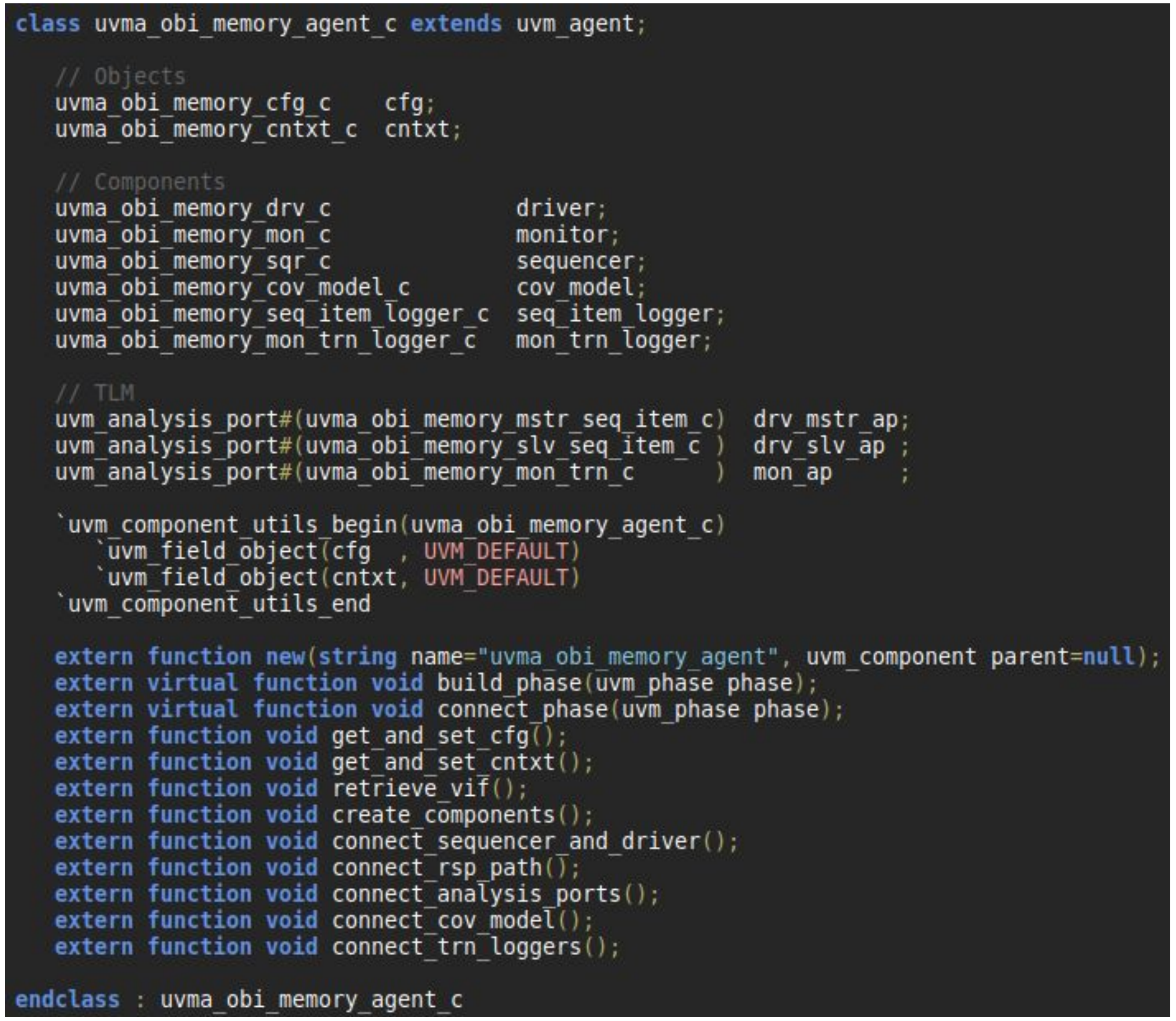
- A UVM Agent processes and generates uvm_sequence_items:
  - An Agent is useless without these, **so code them first**!
- The template-generated sequence items are mostly empty:
  - Sequence items are very specific to the Agent that uses them.
- Coding Guideline: code a unique sequence item for each of a Request or a Response transaction.
- Sequence items will be in lib/uvm_agent/uvma_<agt_name>/src/seq.

uvma_obi_memory_agent.sv

- Templated implementation:
  - Members are objects, components, analysis ports.
  - Methods to implement typical run-flow.

- Templated code allows for standardized naming conventions.

- To implement your Agent:
  - Create the sequence items.
  - Implement the predefined methods in the agent class.
  - Implement required virtual sequences.



```
class uvma_obi_memory_agent_c extends uvm_agent;

   // Objects
   uvma_obi_memory_cfg_c    cfg;
   uvma_obi_memory_cntxt_c  cntxt;

   // Components
   uvma_obi_memory_drv_c               driver;
   uvma_obi_memory_mon_c               monitor;
   uvma_obi_memory_sqr_c               sequencer;
   uvma_obi_memory_cov_model_c         cov_model;
   uvma_obi_memory_seq_item_logger_c   seq_item_logger;
   uvma_obi_memory_mon_trn_logger_c    mon_trn_logger;

   // TLM
   uvm_analysis_port#(uvma_obi_memory_mstr_seq_item_c)  drv_mstr_ap;
   uvm_analysis_port#(uvma_obi_memory_slv_seq_item_c )  drv_slv_ap ;
   uvm_analysis_port#(uvma_obi_memory_mon_trn_c      )  mon_ap     ;

   `uvm_component_utils_begin(uvma_obi_memory_agent_c)
      `uvm_field_object(cfg  , UVM_DEFAULT)
      `uvm_field_object(cntxt, UVM_DEFAULT)
   `uvm_component_utils_end

   extern function new(string name="uvma_obi_memory_agent", uvm_component parent=null);
   extern virtual function void build_phase(uvm_phase phase);
   extern virtual function void connect_phase(uvm_phase phase);
   extern function void get_and_set_cfg();
   extern function void get_and_set_cntxt();
   extern function void retrieve_vif();
   extern function void create_components();
   extern function void connect_sequencer_and_driver();
   extern function void connect_rsp_path();
   extern function void connect_analysis_ports();
   extern function void connect_cov_model();
   extern function void connect_trn_loggers();

endclass : uvma_obi_memory_agent_c
```

Configuration

- As you would expect, cfg and cntxt objects are Agent-specific:
    - Templates are mostly a shell with boiler-plate code for “typical” members (e.g. enabled) and soft constraints for their values.

- Cfg for uvma_obi_memory is quite complex with random members for OBI version, field widths, latencies and error injection:
    - Also requires associated constraints and control methods.

Agent Context

- The cntxt is a novel approach for encapsulation:
  - The cfg object is “Agent specific”.
  - The cntxt object is “Environment specific”.

- A common member of an Agent’s cntxt is a handle to a RAL model.
  - Note: we do not use the RAL in core-v-verif.

- uvma_obi_memory cntxt has members for these environment specific items:
  - Virtual Interface for an OBI bus
  - Core memory handle
  - Reset state (pre-reset, in-reset, post-reset)

Driver and Monitor

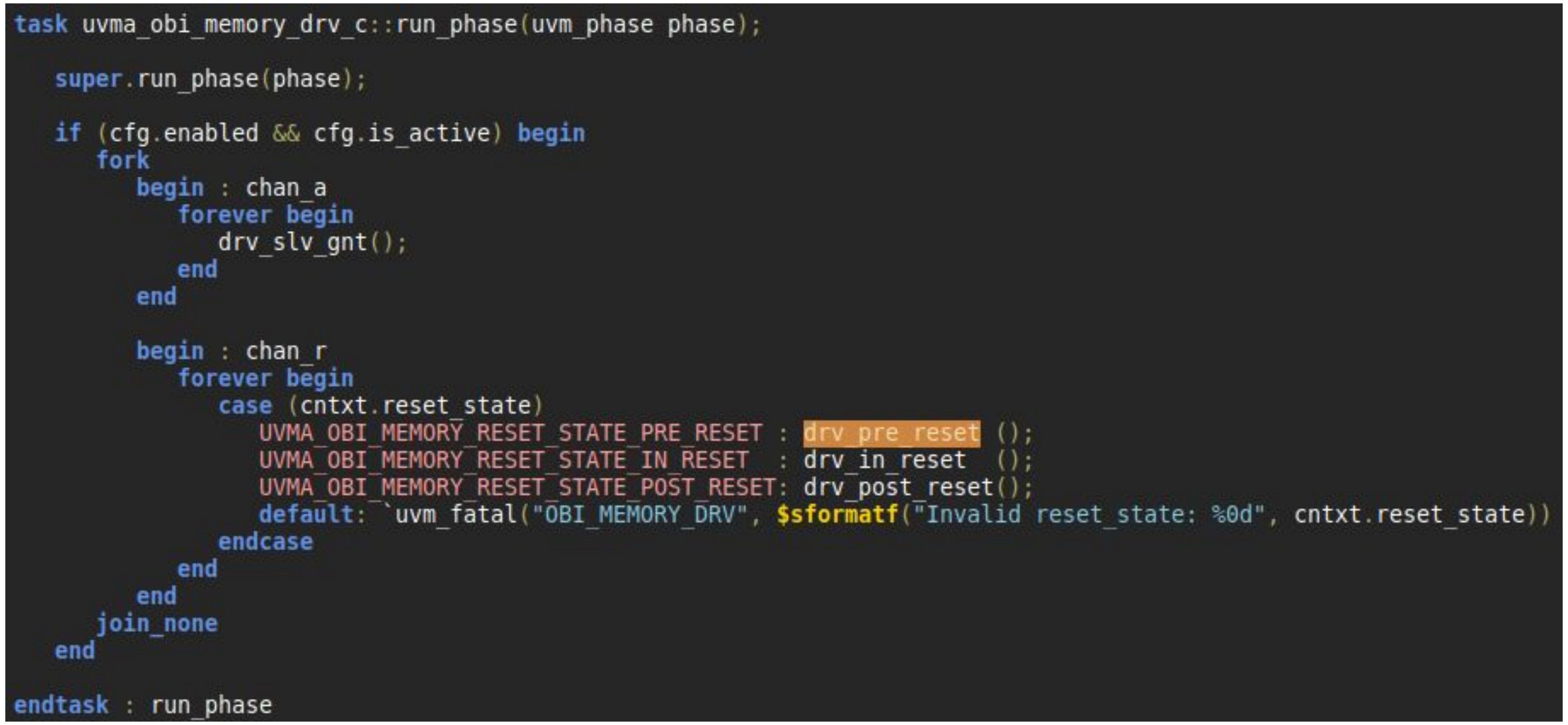
- The driver and monitor components of an Agent do the “grunt” work:
  - Guideline: keep these components as stupid as possible.

- As you would expect, most of this code is human-generated:
  - Template provides a framework for responding to the run-flow, but little else.

- Heads Up! There are no virtual interfaces in these components:
  - The VIFs are members of the Context object.

Run Flow

- The driver and monitor templates support members and methods for generic handling of reset:
  - Can be used to implement middle-of-simulation reset scenarios.
  - Most of the real work of a driver/monitor happens in post_reset().



```
task uvma_obi_memory_drv_c::run_phase(uvm_phase phase);

   super.run_phase(phase);

   if (cfg.enabled && cfg.is_active) begin
      fork
         begin : chan_a
            forever begin
               drv_slv_gnt();
            end
         end

         begin : chan_r
            forever begin
               case (cntxt.reset_state)
                  UVMA_OBI_MEMORY_RESET_STATE_PRE_RESET : drv_pre_reset ();
                  UVMA_OBI_MEMORY_RESET_STATE_IN_RESET  : drv_in_reset  ();
                  UVMA_OBI_MEMORY_RESET_STATE_POST_RESET: drv_post_reset();
                  default: `uvm_fatal("OBI_MEMORY_DRV", $sformatf("Invalid reset_state: %0d", cntxt.reset_state))
               endcase
            end
         end
      join_none
   end

endtask : run_phase
```

Interfaces, Clocking Blocks and Modports

- The templates for the driver and monitor assume the existence of specifically named modports:
  - See “Structure of a UVM Agent” slide in this document.

- The template will generate a shell with all of these clocking blocks and modport predefined:
  - The implementation is still human generated.
  - Once you’ve seen one, you’ve seen ‘em all.

Sequences and Virtual Sequences

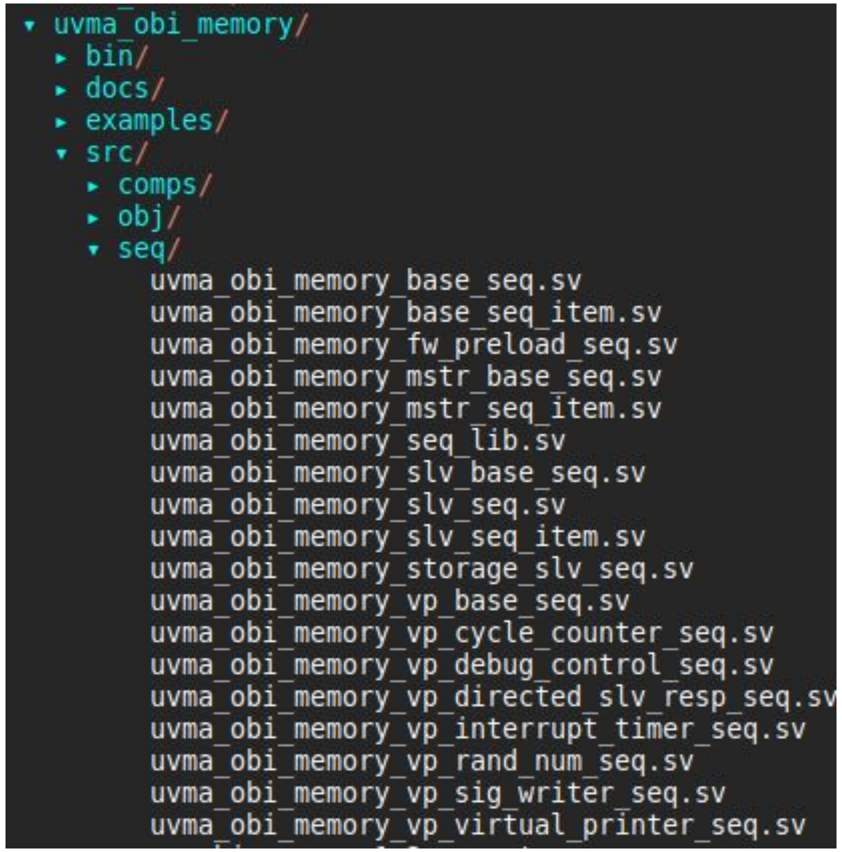
- The "real work" of the Agent is controlled by one or more virtual sequences.

- Agent-specific sequences are optional:
  - Filename will begin with uvma_
  - Location will be in lib/uvm_agent/uvma<agt_name>/src/seq.

- These are properties of the Environment, not the Agent:
  - Filename will begin with uvme_
  - Location will be in <core_name>/env/uvme/vseq.
  -

UVMA_OBI_MEMORY Seqs and VSeqs

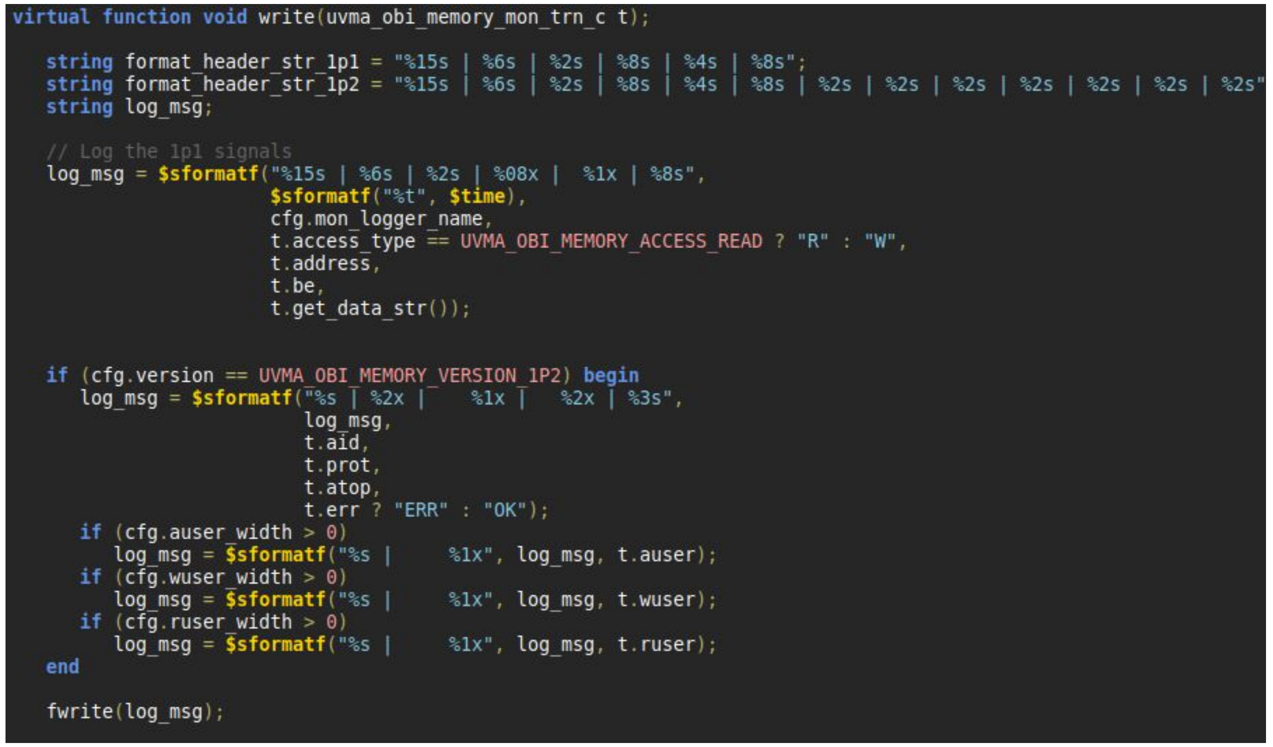
- The obi_memory agent has a set of base sequence_items and sequences.
  - These should be used as the base class for your sequence_items and/or sequences.
  - Need a new one? Create a pull-request!
- The above are used as the base of the various sequences used in the CV32E40 environments:
  - Firmware preload
  - Memory slave
  - Virtual Peripherals.
- Note that memory master sequences exist, but have not been tested.
  - CV32E40* cores are bus masters, not slaves.



```
▾ uvma_obi_memory/
  ▸ bin/
  ▸ docs/
  ▸ examples/
  ▾ src/
    ▸ comps/
    ▸ obj/
    ▾ seq/
        uvma_obi_memory_base_seq.sv
        uvma_obi_memory_base_seq_item.sv
        uvma_obi_memory_fw_preload_seq.sv
        uvma_obi_memory_mstr_base_seq.sv
        uvma_obi_memory_mstr_seq_item.sv
        uvma_obi_memory_seq_lib.sv
        uvma_obi_memory_slv_base_seq.sv
        uvma_obi_memory_slv_seq.sv
        uvma_obi_memory_slv_seq_item.sv
        uvma_obi_memory_storage_slv_seq.sv
        uvma_obi_memory_vp_base_seq.sv
        uvma_obi_memory_vp_cycle_counter_seq.sv
        uvma_obi_memory_vp_debug_control_seq.sv
        uvma_obi_memory_vp_directed_slv_resp_seq.sv
        uvma_obi_memory_vp_interrupt_timer_seq.sv
        uvma_obi_memory_vp_rand_num_seq.sv
        uvma_obi_memory_vp_sig_writer_seq.sv
        uvma_obi_memory_vp_virtual_printer_seq.sv
```

Logging

- Templates include a shell for both transaction and sequence item logging to stdout.
    - Not required - leave them alone if you don’t want a logger.
- Implementing a custom logger is trivial:
    - Implement the ***write*** function in *uvma_<your_agent>_mon_trn.sv*
    - ***write*** is empty by default.



```
virtual function void write(uvma_obi_memory_mon_trn_c t);

   string format_header_str_1p1 = "%15s | %6s | %2s | %8s | %4s | %8s";
   string format_header_str_1p2 = "%15s | %6s | %2s | %8s | %4s | %8s | %2s | %2s | %2s | %2s | %2s | %2s | %2s"
   string log_msg;

   // Log the 1p1 signals
   log_msg = $sformatf("%15s | %6s | %2s | %08x |  %1x | %8s",
                       $sformatf("%t", $time),
                       cfg.mon_logger_name,
                       t.access_type == UVMA_OBI_MEMORY_ACCESS_READ ? "R" : "W",
                       t.address,
                       t.be,
                       t.get_data_str());


   if (cfg.version == UVMA_OBI_MEMORY_VERSION_1P2) begin
      log_msg = $sformatf("%s | %2x |    %1x |   %2x | %3s",
                          log_msg,
                          t.aid,
                          t.prot,
                          t.atop,
                          t.err ? "ERR" : "OK");
      if (cfg.auser_width > 0)
         log_msg = $sformatf("%s |     %1x", log_msg, t.auser);
      if (cfg.wuser_width > 0)
         log_msg = $sformatf("%s |     %1x", log_msg, t.wuser);
      if (cfg.ruser_width > 0)
         log_msg = $sformatf("%s |     %1x", log_msg, t.ruser);
   end

   fwrite(log_msg);
```

Working with UVM Agents in CORE-V-VERIF

UVM Basics

- The following slides assume you are at least somewhat familiar with the structure of a UVM environment.

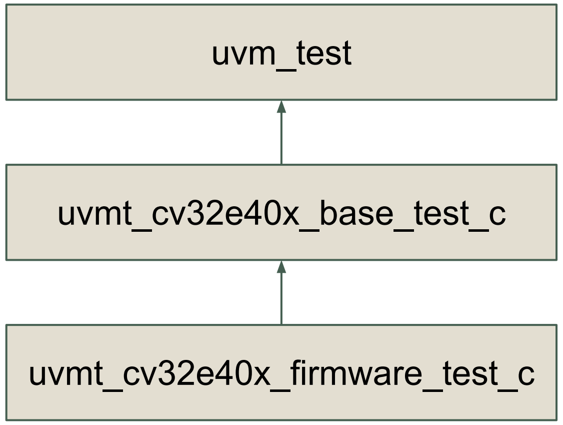
- In particular, you should know:
  - A ***p_sequencer*** is a pointer to a sequencer (exists if you have registered the sequence to the sequencer).
  - The ***`uvm_do()*** macros will run a sequence on a sequencer.

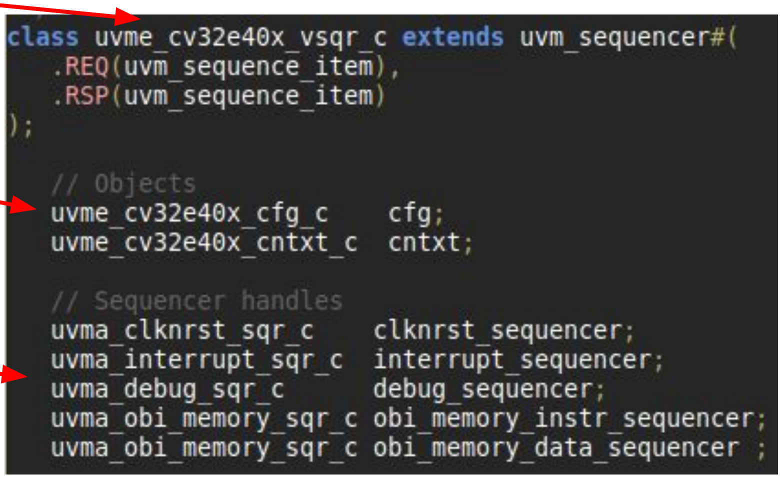
Naming Nomenclature in CORE-V-VERIF

- In CORE-V-VERIF we like long file and class names. :-)
- File and Class names should match:
  - Class name has a "_c" suffix.
- Files with a "uvma_" prefix are UVM Agents:
  - Found in **core-v-verif/lib/uvm_agents**
  - Should *not* be core-specific.
- Files with a "uvme_<core>_" prefix are core-specific UVM environment components:
  - Found in **core-v-verif/<core>/env/uvme**
  - Typically core-specific sequences, agents, etc.

The view from the top: *uvm_test*

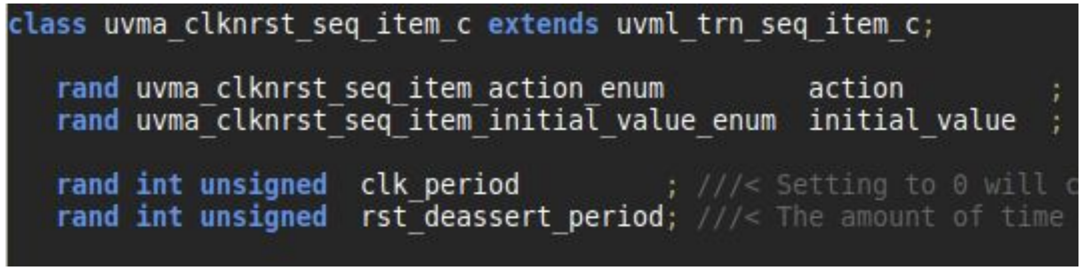
- All testcases extend from *uvm_test*:
  - In the CV32E40X, almost all testing is handled by one UVM test: uvmt_cv32e40x_firmware_test_c, which extends from uvmt_cv32e40x_base_test_c.
  - The base test has a "vsequencer" member of type uvme_cv32e40x_vsqr_c.
  - Like most component objects in core-v-verif, the vsequencer has a configuration (cfg) and context (cntxt) members.
  - The most interesting members of the vsequencer are the handles to all the Agent sequencers.
- The testcase's sequencers are the key to the UVM environment's behaviour.




```
class uvme_cv32e40x_vsqr_c extends uvm_sequencer#(
   .REQ(uvm_sequence_item),
   .RSP(uvm_sequence_item)
);

   // Objects
   uvme_cv32e40x_cfg_c     cfg;
   uvme_cv32e40x_cntxt_c   cntxt;

   // Sequencer handles
   uvma_clknrst_sqr_c     clknrst_sequencer;
   uvma_interrupt_sqr_c   interrupt_sequencer;
   uvma_debug_sqr_c       debug_sequencer;
   uvma_obi_memory_sqr_c obi_memory_instr_sequencer;
   uvma_obi_memory_sqr_c obi_memory_data_sequencer ;
```

The Simplest Example: clknrst

- In core-v-verif, the clknrst Agent is used to generate clock and reset.
- The key to understanding an Agent is to understand the **sequence_item** it works with:
  - An Agent's **driver** maps sequence_items onto the wire protocol of a bus.
  - An Agent's **monitor** maps the wire protocol onto a sequence_item.
- The uvma_clknrst_seq_item_c sequence_item used by the clknrst Agent has four random members:
  - **action**: start_clk, stop_clk, assert_reset, restart_clk
  - **initial_value**: 0, 1 or X
  - **clk_period**
  - **rst_deassert_period**

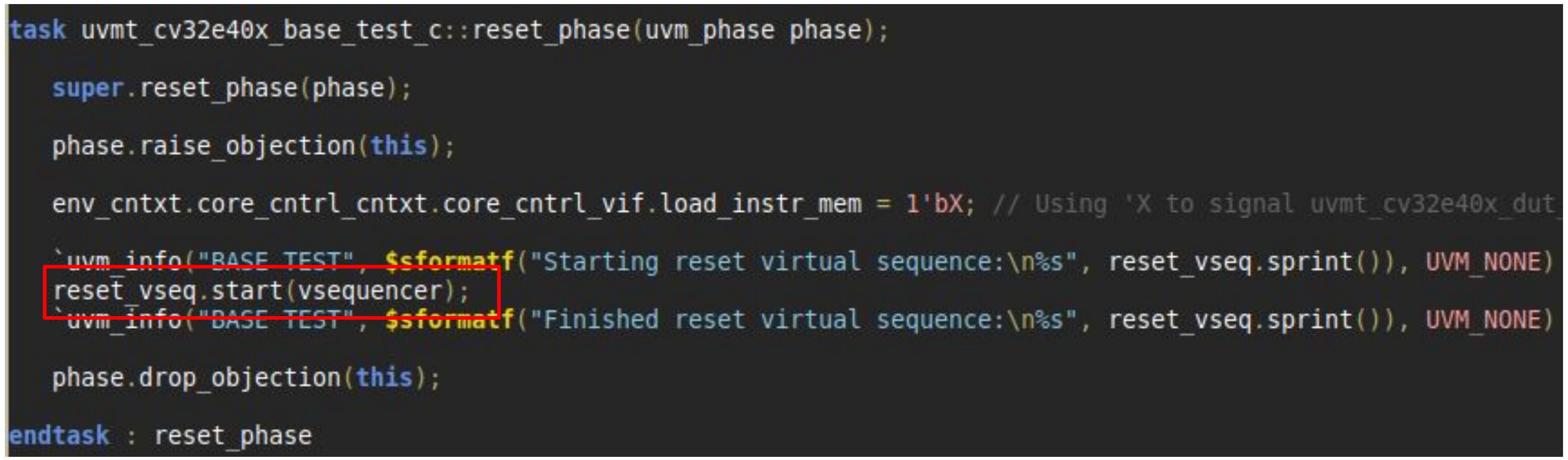


```
class uvma_clknrst_seq_item_c extends uvml_trn_seq_item_c;

   rand uvma_clknrst_seq_item_action_enum        action        ;
   rand uvma_clknrst_seq_item_initial_value_enum  initial_value  ;

   rand int unsigned  clk_period          ; ///< Setting to 0 will c
   rand int unsigned  rst_deassert_period; ///< The amount of time
```

- We can create a library of clknrst sequences, comprised of one or more sequence_items.
  - A UVM testcase applies these sequences to the agent's sequencer.

Getting clock and reset going in CV32E40X

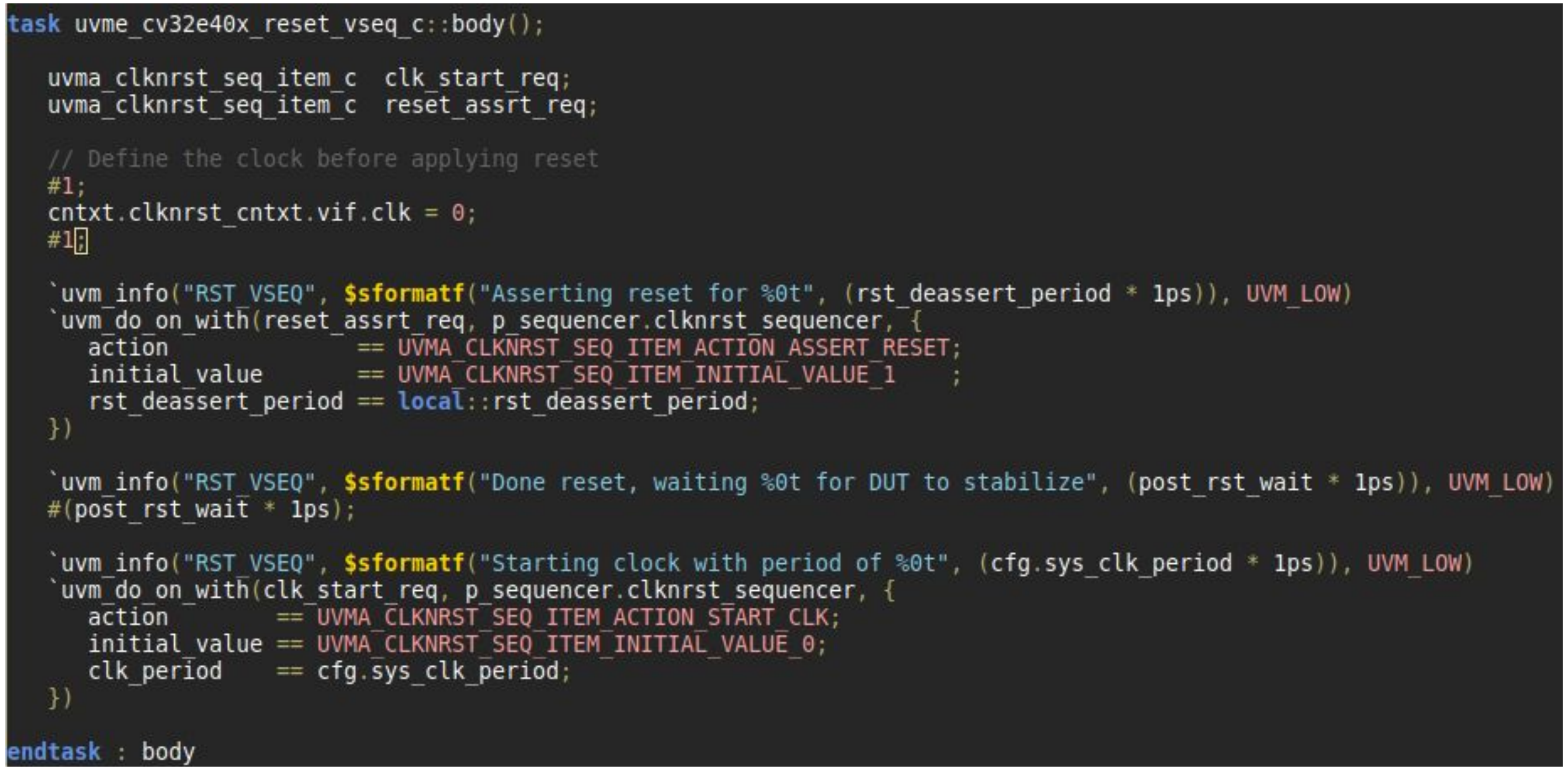
- In CV32E40X, the base test (uvmt_cv32e40x_base_test_c) starts the clock and reset using the poorly named **reset_vseq**:



```
task uvmt_cv32e40x_base_test_c::reset_phase(uvm_phase phase);

   super.reset_phase(phase);

   phase.raise_objection(this);

   env_cntxt.core_cntrl_cntxt.core_cntrl_vif.load_instr_mem = 1'bX; // Using 'X to signal uvmt_cv32e40x_dut

   `uvm_info("BASE_TEST", $sformatf("Starting reset virtual sequence:\n%s", reset_vseq.sprint()), UVM_NONE)
   reset_vseq.start(vsequencer);
   `uvm_info("BASE_TEST", $sformatf("Finished reset virtual sequence:\n%s", reset_vseq.sprint()), UVM_NONE)

   phase.drop_objection(this);

endtask : reset_phase
```

uvme_cv32e40x_reset_vseq_c

- This is a terrible name for the vseq because it actually executes a reset and then starts the clock:



```
task uvme_cv32e40x_reset_vseq_c::body();

   uvma_clknrst_seq_item_c  clk_start_req;
   uvma_clknrst_seq_item_c  reset_assrt_req;

   // Define the clock before applying reset
   #1;
   cntxt.clknrst_cntxt.vif.clk = 0;
   #1;

   `uvm_info("RST_VSEQ", $sformatf("Asserting reset for %0t", (rst_deassert_period * 1ps)), UVM_LOW)
   `uvm_do_on_with(reset_assrt_req, p_sequencer.clknrst_sequencer, {
      action              == UVMA_CLKNRST_SEQ_ITEM_ACTION_ASSERT_RESET;
      initial_value       == UVMA_CLKNRST_SEQ_ITEM_INITIAL_VALUE_1    ;
      rst_deassert_period == local::rst_deassert_period;
   })

   `uvm_info("RST_VSEQ", $sformatf("Done reset, waiting %0t for DUT to stabilize", (post_rst_wait * 1ps)), UVM_LOW)
   #(post_rst_wait * 1ps);

   `uvm_info("RST_VSEQ", $sformatf("Starting clock with period of %0t", (cfg.sys_clk_period * 1ps)), UVM_LOW)
   `uvm_do_on_with(clk_start_req, p_sequencer.clknrst_sequencer, {
      action        == UVMA_CLKNRST_SEQ_ITEM_ACTION_START_CLK;
      initial_value == UVMA_CLKNRST_SEQ_ITEM_INITIAL_VALUE_0;
      clk_period    == cfg.sys_clk_period;
   })

endtask : body
```

- Look at uvma_clknrst_drv_c::drv_req() to see how the driver turns a sequence item into clock and reset signalling.

A Complex Example: obi_memory_agent

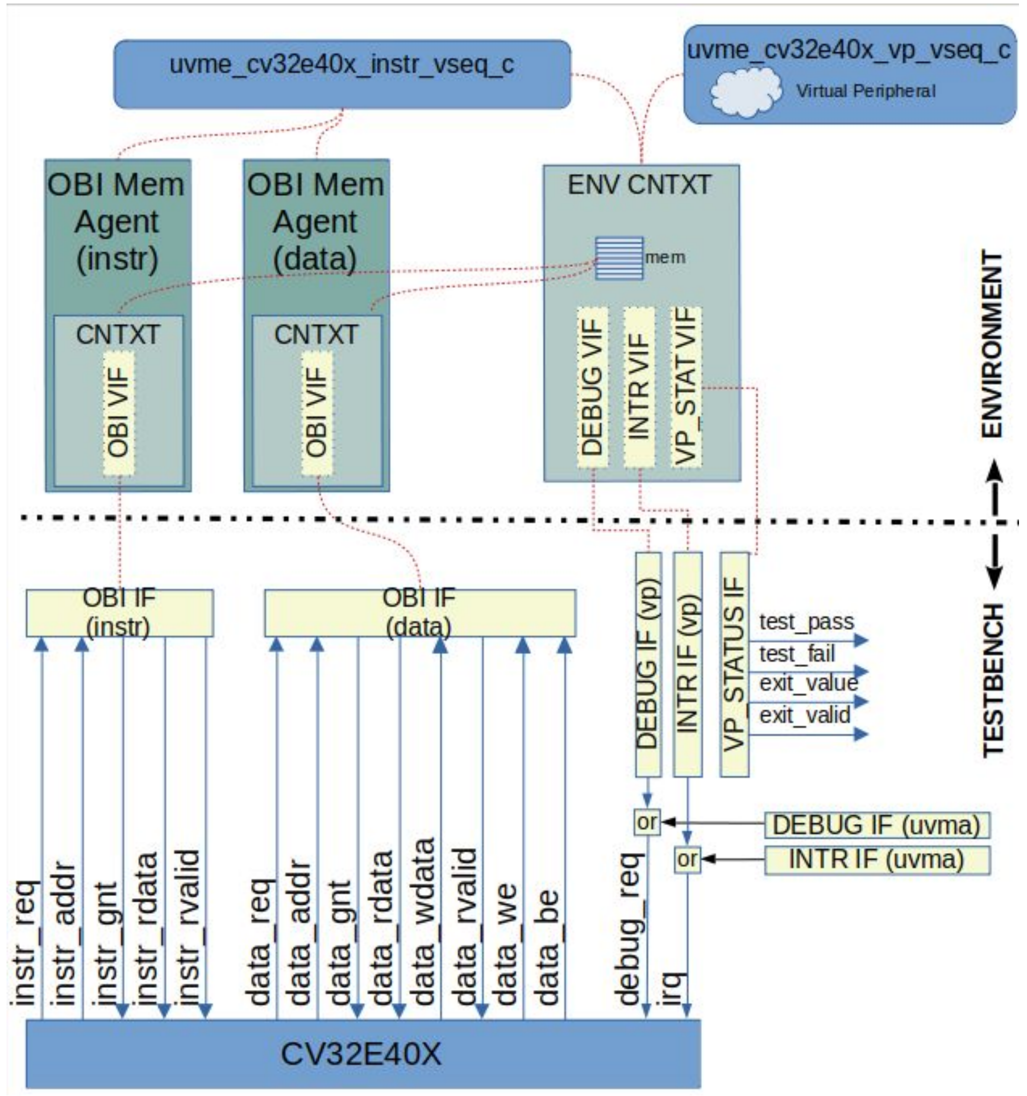
- OBI: Open Bus Interface. An AMBA-like bus protocol.

- The CV32E40 cores support a read-only OBI for instruction fetch and a separate read-write OBI for load/store memories.
    - CV32E40P uses version 1.0 of the OBI spec.
    - CV32E40X and E40S use version 1.2

Run-time configuration

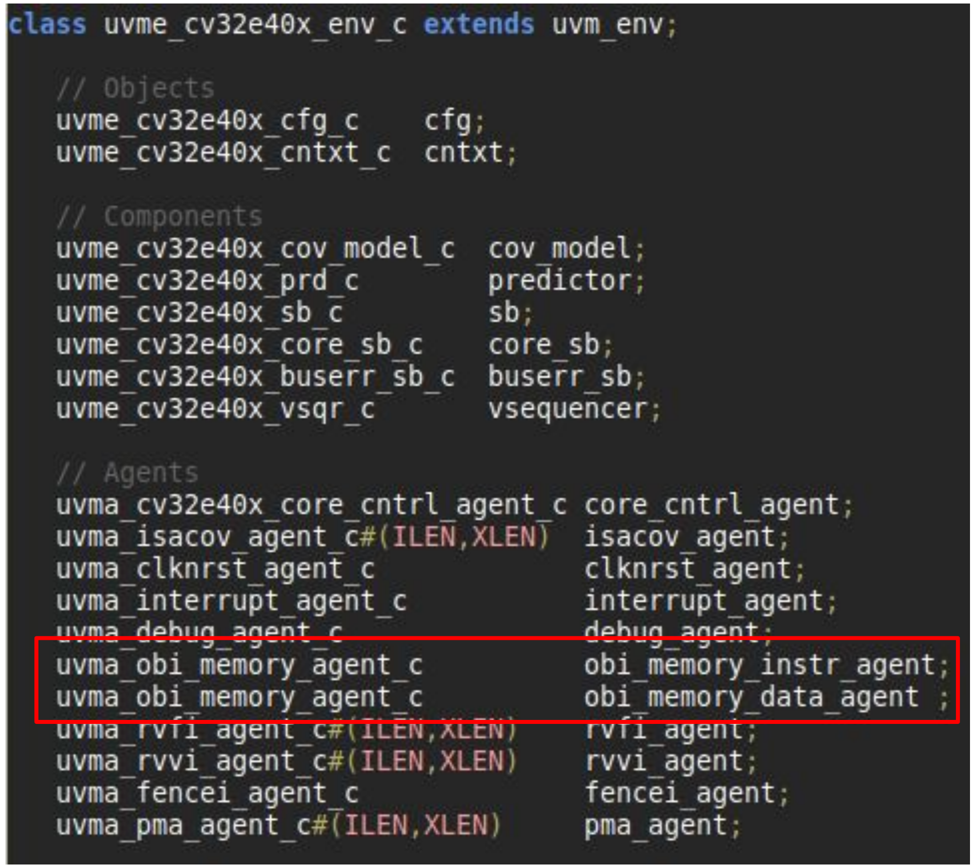
OBI_MEMORY_AGENT in CV32E40X ENV

- Two instances of obi_memory_agent:
  - Instruction bus (read only)
  - Data bus (read/write)
- Agent Context has OBI VIF member which is ***set()*** in the test bench and ***get()*** in the agent.
- Environment context members:
  - VIFs for debug, interrupt and status.
  - Sparse memory model for all memory segments (instruction, data, debug, etc.)
- As much as is practical, operation of agents is determined by sequences.
  - Agents are “as dumb as possible”.



Instantiating the uvma_obi_memory Agents

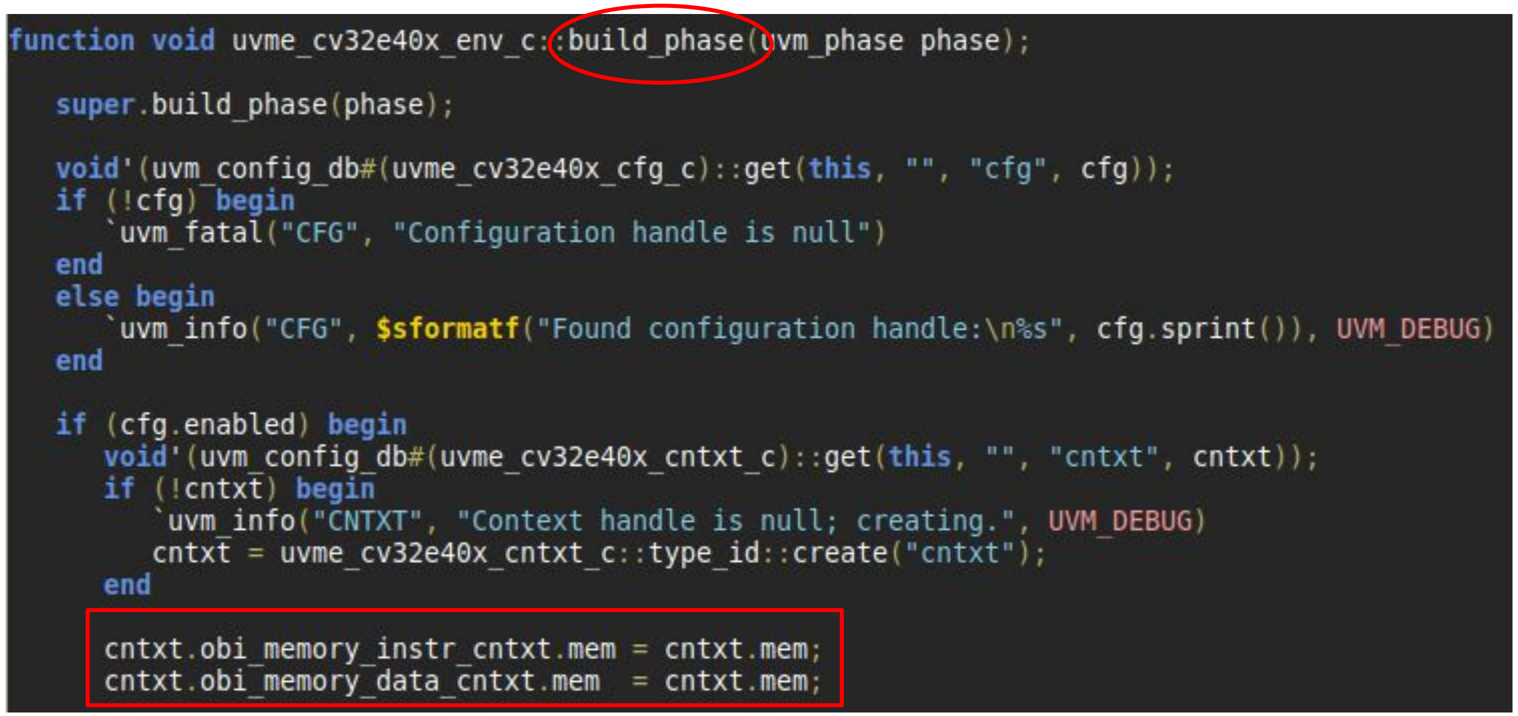
- Two identical instances of the agent.
- Agent does not "know" how it will be used:
  - This intelligence is in the run-time configuration and sequences.



```
class uvme_cv32e40x_env_c extends uvm_env;

   // Objects
   uvme_cv32e40x_cfg_c    cfg;
   uvme_cv32e40x_cntxt_c  cntxt;

   // Components
   uvme_cv32e40x_cov_model_c  cov_model;
   uvme_cv32e40x_prd_c        predictor;
   uvme_cv32e40x_sb_c         sb;
   uvme_cv32e40x_core_sb_c    core_sb;
   uvme_cv32e40x_buserr_sb_c  buserr_sb;
   uvme_cv32e40x_vsqr_c       vsequencer;

   // Agents
   uvma_cv32e40x_core_cntrl_agent_c core_cntrl_agent;
   uvma_isacov_agent_c#(ILEN,XLEN)  isacov_agent;
   uvma_clknrst_agent_c             clknrst_agent;
   uvma_interrupt_agent_c           interrupt_agent;
   uvma_debug_agent_c               debug_agent;
   uvma_obi_memory_agent_c          obi_memory_instr_agent;
   uvma_obi_memory_agent_c          obi_memory_data_agent ;
   uvma_rvfi_agent_c#(ILEN,XLEN)    rvfi_agent;
   uvma_rvvi_agent_c#(ILEN,XLEN)    rvvi_agent;
   uvma_fencei_agent_c              fencei_agent;
   uvma_pma_agent_c#(ILEN,XLEN)     pma_agent;
```

Common Memory for I & D

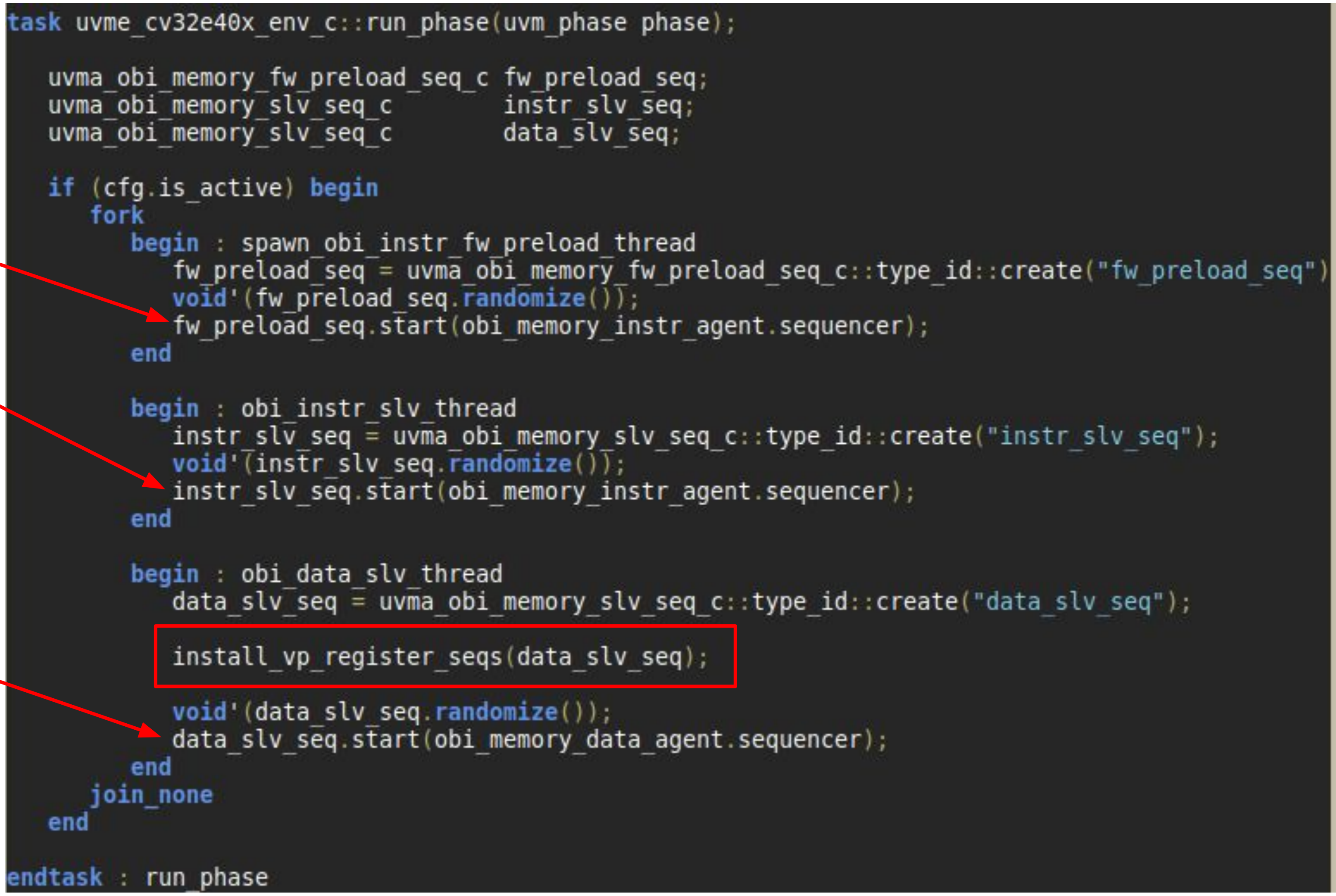
- The Environment has a global context which contains a sparse memory model.



```
function void uvme_cv32e40x_env_c::build_phase(uvm_phase phase);

   super.build_phase(phase);

   void'(uvm_config_db#(uvme_cv32e40x_cfg_c)::get(this, "", "cfg", cfg));
   if (!cfg) begin
      `uvm_fatal("CFG", "Configuration handle is null")
   end
   else begin
      `uvm_info("CFG", $sformatf("Found configuration handle:\n%s", cfg.sprint()), UVM_DEBUG)
   end

   if (cfg.enabled) begin
      void'(uvm_config_db#(uvme_cv32e40x_cntxt_c)::get(this, "", "cntxt", cntxt));
      if (!cntxt) begin
         `uvm_info("CNTXT", "Context handle is null; creating.", UVM_DEBUG)
         cntxt = uvme_cv32e40x_cntxt_c::type_id::create("cntxt");
      end

      cntxt.obi_memory_instr_cntxt.mem = cntxt.mem;
      cntxt.obi_memory_data_cntxt.mem  = cntxt.mem;
```

Running the uvma_obi_memory Agents

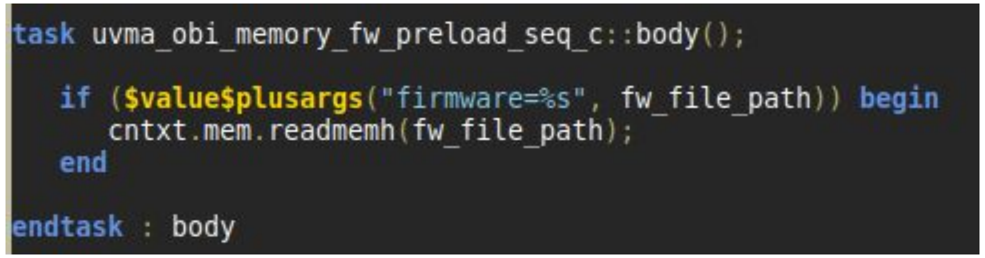
- The instr_agent runs both a “fw_preload” and a “instr_slv” sequence.
- The data_agent runs a “data_slv_seq” that support memory reads and writes.



```
task uvme_cv32e40x_env_c::run_phase(uvm_phase phase);

   uvma_obi_memory_fw_preload_seq_c fw_preload_seq;
   uvma_obi_memory_slv_seq_c        instr_slv_seq;
   uvma_obi_memory_slv_seq_c        data_slv_seq;

   if (cfg.is_active) begin
      fork
         begin : spawn_obi_instr_fw_preload_thread
            fw_preload_seq = uvma_obi_memory_fw_preload_seq_c::type_id::create("fw_preload_seq")
            void'(fw_preload_seq.randomize());
            fw_preload_seq.start(obi_memory_instr_agent.sequencer);
         end

         begin : obi_instr_slv_thread
            instr_slv_seq = uvma_obi_memory_slv_seq_c::type_id::create("instr_slv_seq");
            void'(instr_slv_seq.randomize());
            instr_slv_seq.start(obi_memory_instr_agent.sequencer);
         end

         begin : obi_data_slv_thread
            data_slv_seq = uvma_obi_memory_slv_seq_c::type_id::create("data_slv_seq");

            install_vp_register_seqs(data_slv_seq);

            void'(data_slv_seq.randomize());
            data_slv_seq.start(obi_memory_data_agent.sequencer);
         end
      join_none
   end

endtask : run_phase
```

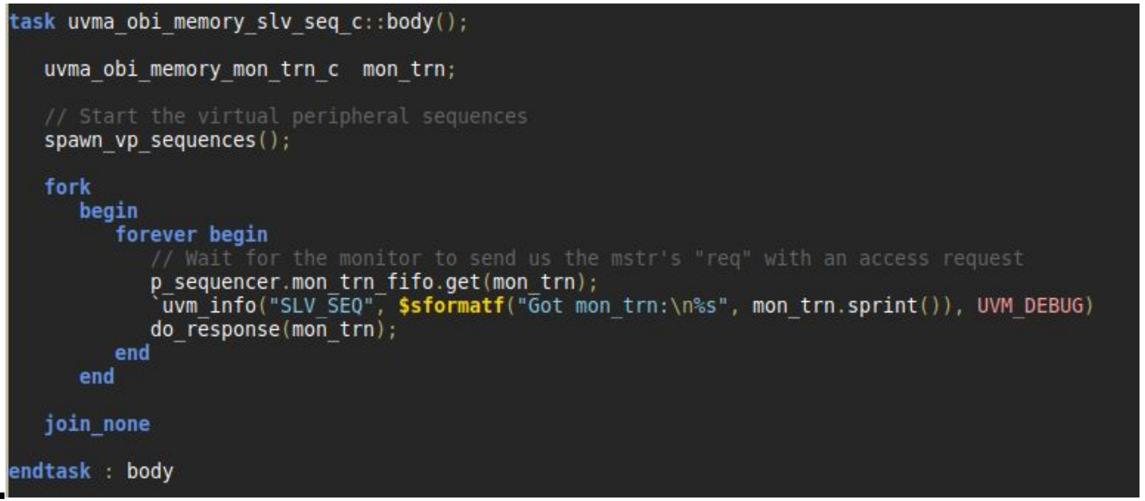
fw_preload_seq and slv_seq

- fw_preload_seq simply reads the pre-compiled fixware (hex file) into the environment's context memory.



```
task uvma_obi_memory_fw_preload_seq_c::body();

   if ($value$plusargs("firmware=%s", fw_file_path)) begin
      cntxt.mem.readmemh(fw_file_path);
   end

endtask : body
```

- slv_seq is a little more complex:
  - spawn_vp_sequences();
  - Fetch and process sequence_items from the Agent's monitor.
  - do_response() is a local method which checks for bus errors and then performs memory read/write operation.



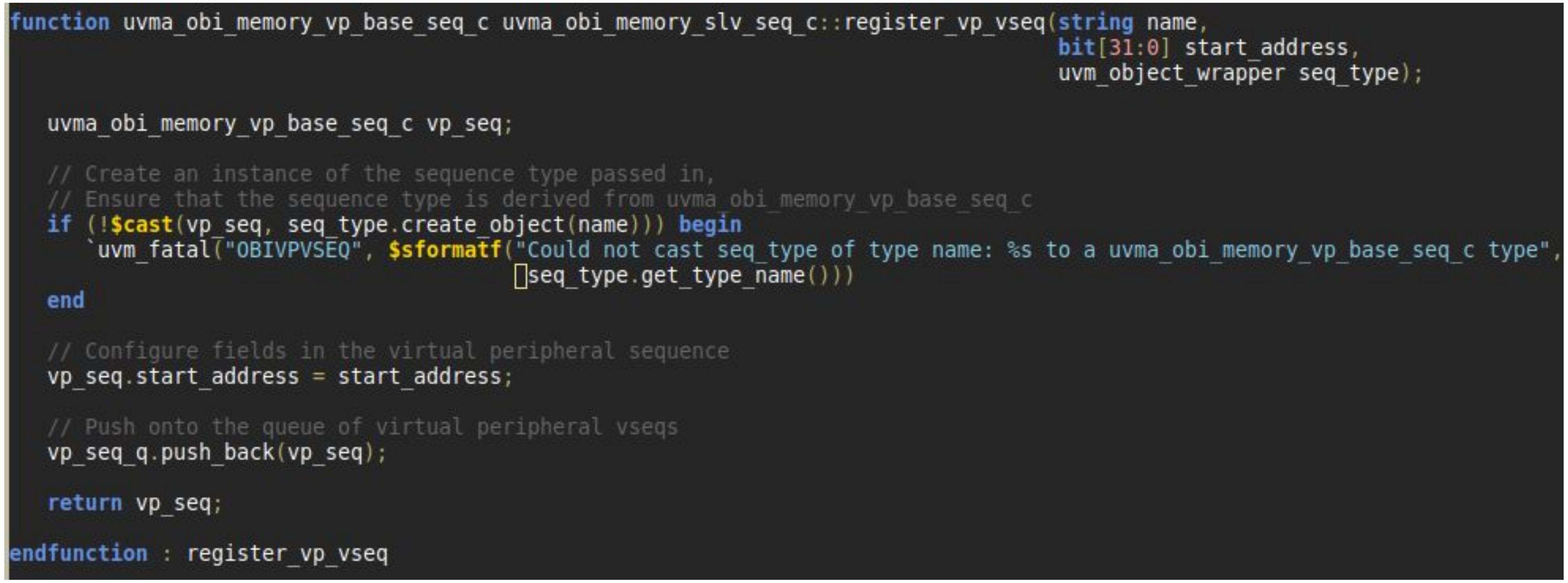
```
task uvma_obi_memory_slv_seq_c::body();

   uvma_obi_memory_mon_trn_c  mon_trn;

   // Start the virtual peripheral sequences
   spawn_vp_sequences();

   fork
      begin
         forever begin
            // Wait for the monitor to send us the mstr's "req" with an access request
            p_sequencer.mon_trn_fifo.get(mon_trn);
            `uvm_info("SLV_SEQ", $sformatf("Got mon_trn:\n%s", mon_trn.sprint()), UVM_DEBUG)
            do_response(mon_trn);
         end
      end

   join_none

endtask : body
```

Registering the Virtual Peripherals

- Recall uvme_cv32e40x_env_c::install_vp_register_seqs() from two slides ago…
- This method “registers” each Virtual Peripheral’s sequence with the data_slv_seq:
  - Simple checks for casting and sanity.
  - Sets start address.
  - Pushes the sequence onto a queue.
- This sets up spawn_vp_sequences() to iterate over the queue of registered sequences, starting each one.

register_vp_vseq()

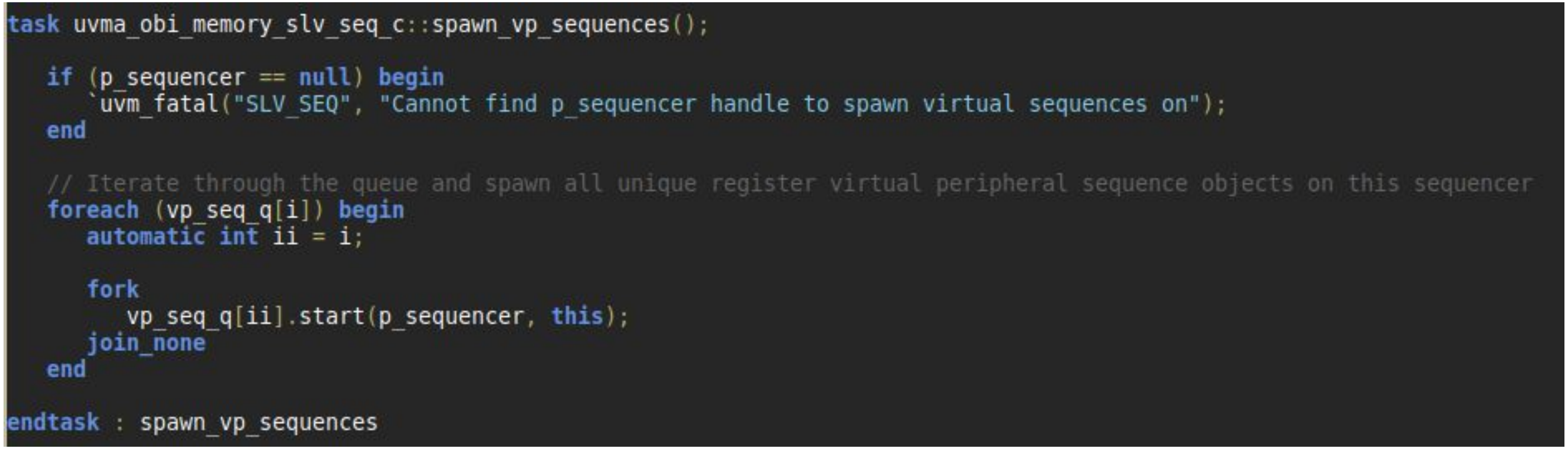
- Below is a simplified implementation of uvma_obi_memory_slv_seq_c::register_vp_vseq().



```
function uvma_obi_memory_vp_base_seq_c uvma_obi_memory_slv_seq_c::register_vp_vseq(string name,
                                                                                   bit[31:0] start_address,
                                                                                   uvm_object_wrapper seq_type);

   uvma_obi_memory_vp_base_seq_c vp_seq;

   // Create an instance of the sequence type passed in,
   // Ensure that the sequence type is derived from uvma_obi_memory_vp_base_seq_c
   if (!$cast(vp_seq, seq_type.create_object(name))) begin
      `uvm_fatal("OBIVPVSEQ", $sformatf("Could not cast seq_type of type name: %s to a uvma_obi_memory_vp_base_seq_c type",
                                         seq_type.get_type_name()))
   end

   // Configure fields in the virtual peripheral sequence
   vp_seq.start_address = start_address;

   // Push onto the queue of virtual peripheral vseqs
   vp_seq_q.push_back(vp_seq);

   return vp_seq;

endfunction : register_vp_vseq
```

spawn_vp_sequences()

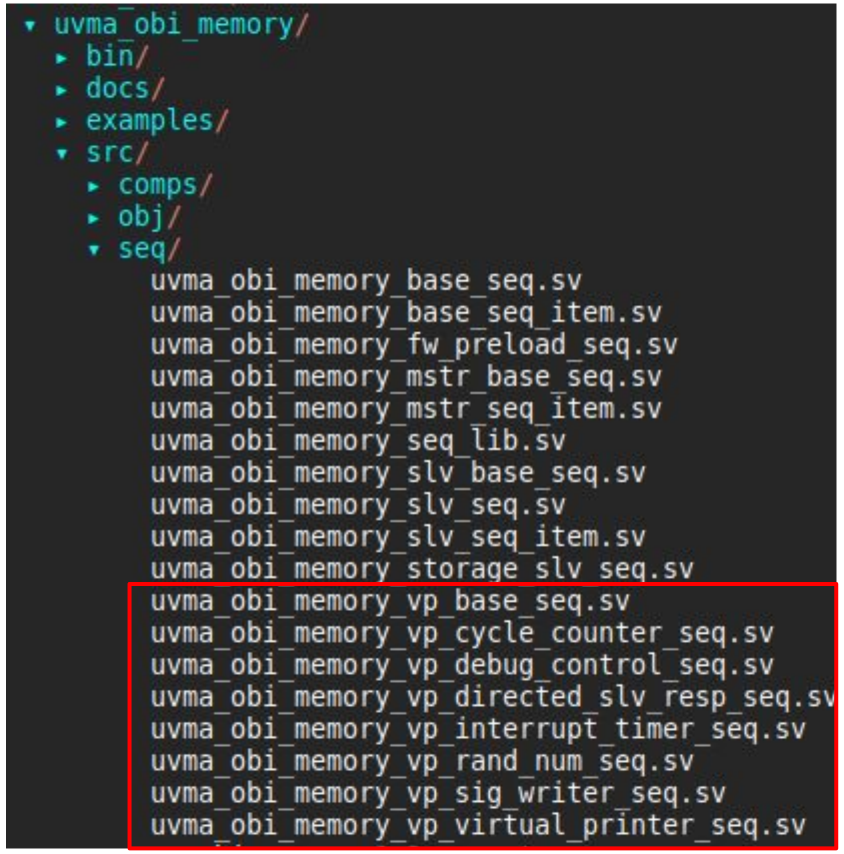
- uvma_obi_memory_slv_seq_c::spawn_vp_sequences() iterates over the queue of registered sequences, starting each one.



```
task uvma_obi_memory_slv_seq_c::spawn_vp_sequences();

   if (p_sequencer == null) begin
      `uvm_fatal("SLV_SEQ", "Cannot find p_sequencer handle to spawn virtual sequences on");
   end

   // Iterate through the queue and spawn all unique register virtual peripheral sequence objects on this sequencer
   foreach (vp_seq_q[i]) begin
      automatic int ii = i;

      fork
         vp_seq_q[ii].start(p_sequencer, this);
      join_none
   end

endtask : spawn_vp_sequences
```

Implementing a Virtual Peripheral (1 of 2)

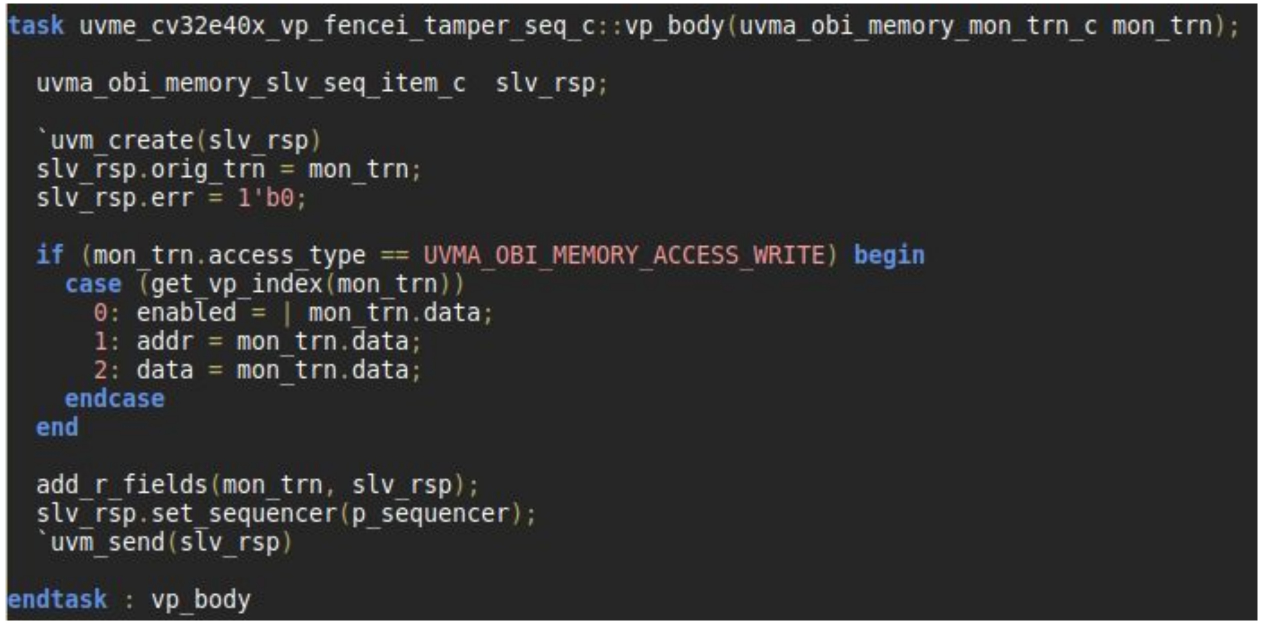
- uvma_obi_memory has a library of virtual peripherals implemented as virtual sequences.
- Any of these can be registered and run in any environment that uses the obi_memory Agent.



```
▾ uvma_obi_memory/
  ▸ bin/
  ▸ docs/
  ▸ examples/
  ▾ src/
    ▸ comps/
    ▸ obj/
    ▾ seq/
        uvma_obi_memory_base_seq.sv
        uvma_obi_memory_base_seq_item.sv
        uvma_obi_memory_fw_preload_seq.sv
        uvma_obi_memory_mstr_base_seq.sv
        uvma_obi_memory_mstr_seq_item.sv
        uvma_obi_memory_seq_lib.sv
        uvma_obi_memory_slv_base_seq.sv
        uvma_obi_memory_slv_seq.sv
        uvma_obi_memory_slv_seq_item.sv
        uvma_obi_memory_storage_slv_seq.sv
        uvma_obi_memory_vp_base_seq.sv
        uvma_obi_memory_vp_cycle_counter_seq.sv
        uvma_obi_memory_vp_debug_control_seq.sv
        uvma_obi_memory_vp_directed_slv_resp_seq.sv
        uvma_obi_memory_vp_interrupt_timer_seq.sv
        uvma_obi_memory_vp_rand_num_seq.sv
        uvma_obi_memory_vp_sig_writer_seq.sv
        uvma_obi_memory_vp_virtual_printer_seq.sv
```

Implementing a Virtual Peripheral (2 of 2)

- To create your own environment-specific Virtual Peripheral for the OBI Agent:
  - The source file should be in **core-v-verif/<core>/env/uvme/vseq**.
  - Sequence should extend from uvma_obi_memory_vp_base_seq_c.
  - Implement vp_body().



```
task uvme_cv32e40x_vp_fencei_tamper_seq_c::vp_body(uvma_obi_memory_mon_trn_c mon_trn);

  uvma_obi_memory_slv_seq_item_c  slv_rsp;

  `uvm_create(slv_rsp)
  slv_rsp.orig_trn = mon_trn;
  slv_rsp.err = 1'b0;

  if (mon_trn.access_type == UVMA_OBI_MEMORY_ACCESS_WRITE) begin
    case (get_vp_index(mon_trn))
      0: enabled = | mon_trn.data;
      1: addr = mon_trn.data;
      2: data = mon_trn.data;
    endcase
  end

  add_r_fields(mon_trn, slv_rsp);
  slv_rsp.set_sequencer(p_sequencer);
  `uvm_send(slv_rsp)

endtask : vp_body
```

Thank You