DVCon 2013

Design & Verification Conference & Exhibition

February 25-28, 2013
DoubleTree, San Jose

accellera SYSTEMS INITIATIVE

Maximize Vertical Reuse, Building Module to System Verification Environments with UVM*e*

Horace Chan
PMC-Sierra

Brian Vandegriend
PMC-Sierra

Deepali Joshi
PMC-Sierra

Corey Goss
Cadence

PMC

cadence™

What is vertical reuse?

- Horizontal Reuse
  - Reuse VIP across different projects
- Vertical Reuse
  - Reuse new VIP, sequences, checkers among module, subsystem and system testbenches in the same project
- Maximize Vertical Reuse
  - Import full module testbenches to subsystem testbench
  - Import full subsystem testbeches to system testbench
  - Stitch subsystem testcases to create system testcase

Benefits of vertical reuse

- Parallel development of all testbenches
- Plug-n-play architecture to integrate system level testbench from module and subsystem testbenches seamlessly
- Increase verification efficiency
  - No duplication of development effort
  - Less testbench code maintenance
    - Changes in lower level testbench is propagated up automatically
  - Better debug support from lower level testbench
    - All lower level testbench the checkers and monitors are available
  - Easier to move verification engineers among module, subsystems and system level testing

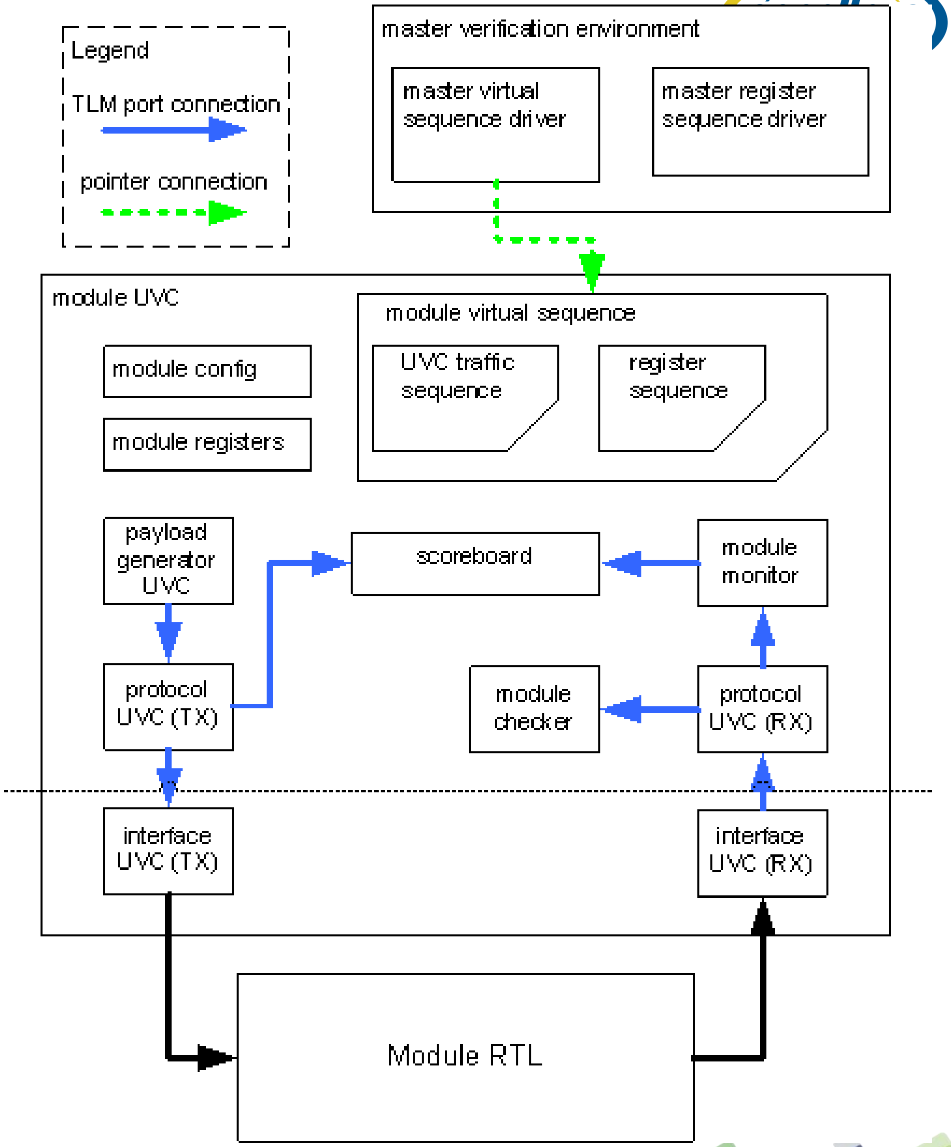
The testbench architecture

- 200+ Million Gates Design
  - Partition into 10 subsystems and over 50 modules
  - Over 20 subsystem and module testbenches
- Testbench implemented in Specman UVM*e*
  - All VIPs communicate using TLM interface
- The system testbench is integrated in 1 month
  - 2-3 days to bring in a new subsystem testbench
  - Early initial system level testing while subsystem testbench is still finalizing features

System UVC Architecture

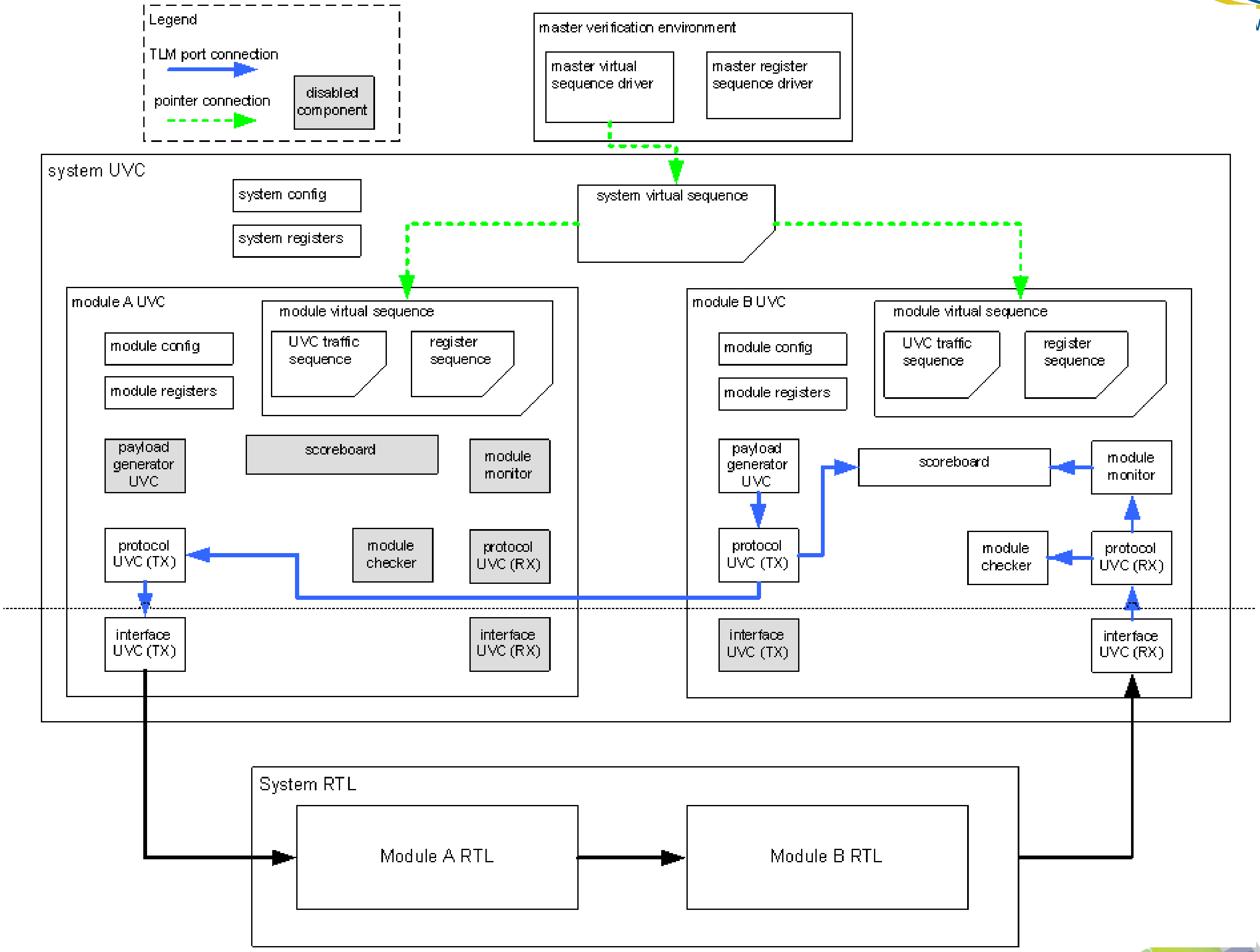
Sponsored By:

- Based on Cadence's System UVC Architecture
- Module UVC
  1. No master virtual sequence or register sequence
  2. UVCs are layered using TLM ports
  3. Separate protocol UVCs from interface UVCs that drive the RTL signals
- System UVC is created by putting together multiple module UVCs



System UVC Example

Sponsored By:



TLM port router

- TLM port limitations:

1. TLM transport port does not support one-to-many binding
2. TLM analysis port always broadcast
3. Port binding is static in the simulation

- Solution: TLM port router to support dynamic many-to-many port binding with build-in routing table

```
// sample implementation of TLM analysis port router
template unit port_router_u of (<type>) {
   in_ports : list of in interface_port of  tlm_analysis
               of <type> is instance;
   out_ports : list of out interface_port of  tlm_analysis
              of <type> is instance;
    get_channel_id(tr : <type>) : uint is {};
    set_channel_id(tr : <type>, cid : uint) is {}
    routing_table : list of src_route_table_entry_s;
};
 struct dest_route_table_entry_s {
   enable     : bool;
   port_id    : uint;
   channel_id : uint;
};
 struct src_route_table_entry_s {
   enable       : bool;
   port_id      : uint;
   channel_id   : uint;
   destinations : list of dest_route_table_entry_s;
};
```

Sponsored By:

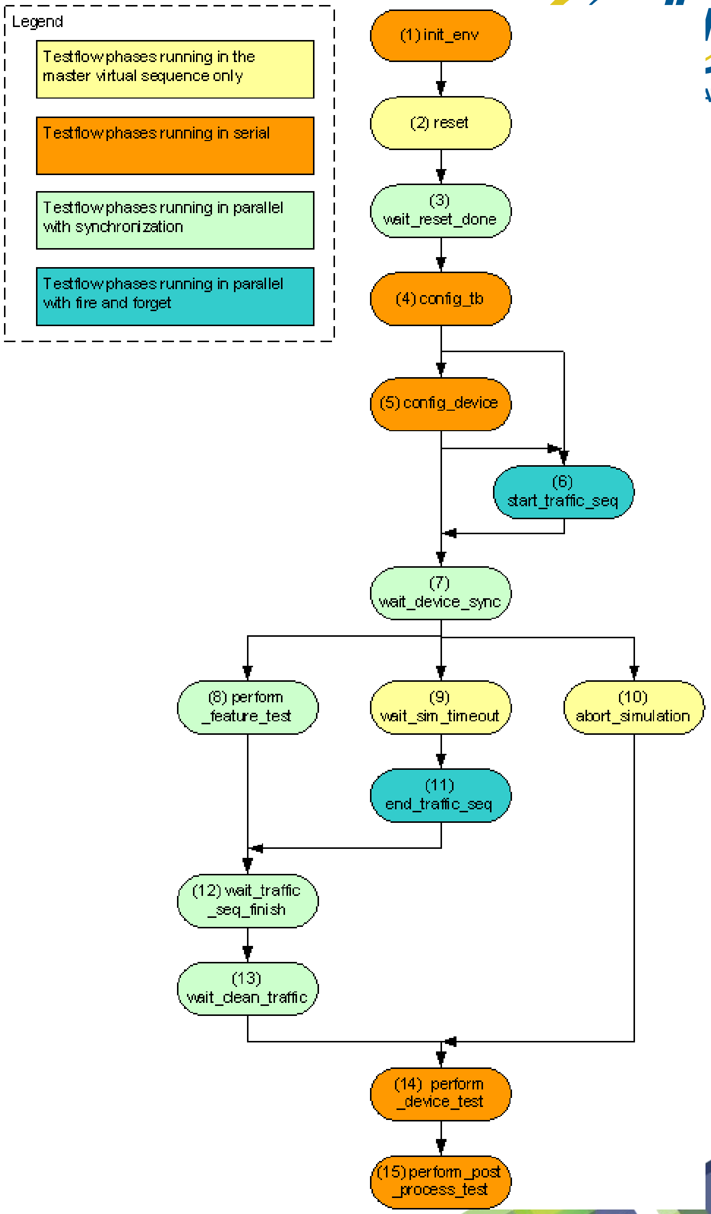
Common UVC Configuration Control

- Unify data structure to control TLM port binding in module UVC
- Enable/disable each individual UVC, checker, scoreboard inside the module UVC
- Preserve all the binding information of the module UVC in system UVC

```
// sample common config control data structure definition
struct config_ctrl_s {
    layer_name   : layer_t;
    enable       : bool;
    is_active    : uvm_active_passive_t;
    bind_enable  : bool;
};
struct port_config_ctrl_s {
    port_name     : port_t;
    layer_config : list of config_ctrl_s;
};
extend uvm_env {
    config_ctrl_table : list of port_config_ctrl_s;
     get_uvc_enable(port:port_t, layer:layer_t)  : bool is {};
    get_uvc_is_active(port:port_t, layer:layer_t)
                : uvm_active_passive_t is {};
    get_uvc_bind_enable(port:port_t, layer:layer_t)
                : uvm_active_passive_t is {};
};
```

Common Test Flow Virtual Sequence

- Testflow to co-ordinate and synchronize the behavior of imported module UVCs virtual sequences
- Testflow phases are implemented as empty TCM methods in the virtual sequence base class
- Module UVC fill in testflow phases using extension


Legend
Testflow phases running in the master virtual sequence only
Testflow phases running in serial
Testflow phases running in parallel with synchronization
Testflow phases running in parallel with fire and forget
(1) init_env
(2) reset
(3) wait_reset_done
(4) config_tb
(5) config_device
(6) start_traffic_seq
(7) wait_device_sync
(8) perform _feature_test
(9) wait_sim_timeout
(10) abort_simulation
(11) end_traffic_seq
(12) wait_traffic _seq_finish
(13) wait_clean_traffic
(14) perform _device_test
(15) perform_post _process_test

Benefits and results

Statistic Measure	Previous Project	Current Project	Changes
Gates count	60M	200M	+133%
Total lines of code	575k	484k	-16%
System testbench line of code	324k	215k	-34%
% system testbench in total code	56%	44%	-22%
Gates verified per line of code	104k	413k	**+400%**

- Less code, less bugs
- More reuse, higher quality of the code
- System level testcase as short as 20 lines of code
- Better system level debug support from subsystem and module level verification engineers

Challenges and Solutions

- Revision conflict of common VIP used by the testbenches
  - Solution:
    - Freeze common VIP revision early
    - Make sure new revision is backward compatible
    - Co-ordinate upgrade of non-backward compatible VIP revision across all testbenches

- Poor quality code imported from lower level testbench impact simulation performance
  - Solution:
    - All testbenches should run profiling to identify CPU and memory bottleneck
    - Frequent code review by experienced engineers

Future development

- Port the framework to SystemVerilog UVM
- SV does not support Aspect Oriented Programming (AOP)
  - Possible to work around using design patterns
    - More lines of code and complex TB structure
    - More upfront planning for hooks and APIs
    - More intrusive code maintenance
    - More revision control discipline
- ***e*** constructs used in the framework
  - ***e*** template -> SV parameterized type
  - ***e*** keyed list -> SV associate array
  - ***e*** predefined routines -> SV macro-based util libraries