Adopting UVM methodology for IP level Verification

Giovanni AUDITORE, Francesco RUA'
STMicroelectronics
Catania, Italy
www.st.com

ABSTRACT

The STMicroelectronics Microcontroller division (MCD) designs micro-controllers SoC's that embeds a large portfolio of Digital IPs requiring advanced verification techniques in order to guarantee an exhaustive verification of their products.

The MCD Digital IP verification team has chosen, for the JPEG1 IP verification project, to explore the System Verilog UVM approach on the new integrated Synopsys verification platform on a real production project. The target of this activity is to evaluate the value of the UVM methodology in terms of ease-of-use, debugging capabilities, VIP usage and reuse using the Synopsys UVM ecosystem.
During this evaluation, the verification team has tested the Synopsys AMBA VIP, VCS simulator, Verdi UVM debug, Execution Manager and Verdi Coverage.

In this paper, we will present the JPEG1 project (Testbench environment and IP under test), the efforts and challenges of such a verification activity, the results and our conclusions regarding the UVM standard and the Synopsys unified solution.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

This paper is a journey throughout the verification offer of Synopsys with the intent of fully exploiting the System Verilog UVM extension capabilities into maximizing verification reuse. The verification activity has been performed on a production project to fully stress the verification achieved result up to sign-off quality level.

This project is a digital IP wrapper to a Synopsys JPEG codec IP. The purpose of the wrapper is to add an AHB bus interface to control and monitor the status of the internal codec core, plus the addition of a FIFO to either buffer the data to be encoded or to store the decoded data.

The verification engineers assigned to the project were completely new to the SV language, to the UVM methodology (as described in UVM standard version 1.1 [1][2]) and to the Synopsys verification platform.

2. Verification Challenge

The primary target of this activity was to verify the JPEG1 IP. While doing that we target also the following objectives:

- Get familiar with the Synopsys most advanced verification framework used in System Verilog UVM standard flow
- Explore the usage and integration of Synopsys VIPs and verification packages
- Explore System Verilog language capabilities
- Explore UVM standard verification objects integration, reuse and extension
- Explore UVM Factory configurability infrastructure

3. Verification environment

The testbench architecture

The purpose of this verification activity is to verify mainly the correct wrapping of what can be consider as a golden core, plus the IP registers control and status interface.
The JPEG1 HDL wrapper accesses the internal core through the usage of a AMBA AHB Bus Protocol interface connected to a register file, to data FIFOs and through those to the internal core and to the memories required to hold the conversion tables.
The strategy therefore is:

1. check the protocol conversion and data flow correctness
2. check SW controllability and observability through register bus interface

To implement efficiently such strategy we have to maximize reuse and configurability of the uvm_in_order_comparator that is used to implement scoreboard like checkers in many interfaces.
To model the IP registers, the strategy is to use the uvm_register classes taking advantage of the Synopsys Register Abstraction Level (RAL) automatic code generation (ralgen).
The internal core interfaces as well as the memories interfaces also requires to be monitored. We decided to do this by mean of small custom passive VIP to make the environment more reusable in the future.
The resulting testbench architecture is shown in ***Figure 1 Testbench architecture***.

All scoreboards use a white box approach, because these collects data from external DUT interfaces and internal JPEG Codec core interfaces. As shown in the picture we have:

- One scoreboard between bus interface and JPEG core CPU interface. This collects AHB bus transfers and, after filtering and conversion, matches the collected items against the ones collected at CPU I/F.
- One scoreboard for each memory, between the memory itself and the bus interface. This collects AHB bus transfers and, after filtering and conversion, matches the collected items against the relevant memory accesses.
- One scoreboard for input FIFO, which stay in between the bus interface and core pixel interface. This collect pixel data to encode or encoded data to be decoded on the bus interface and matches with equivalent data in input to the core pixel I/F.
- One scoreboard for output FIFO, which stay in between the bus interface and core pixel interface. This collect encoded data or decoded pixel data on the bus interface and matches with equivalent data in output from the core pixel I/F.
- The Synopsys RAL generated UVM register file and memory model

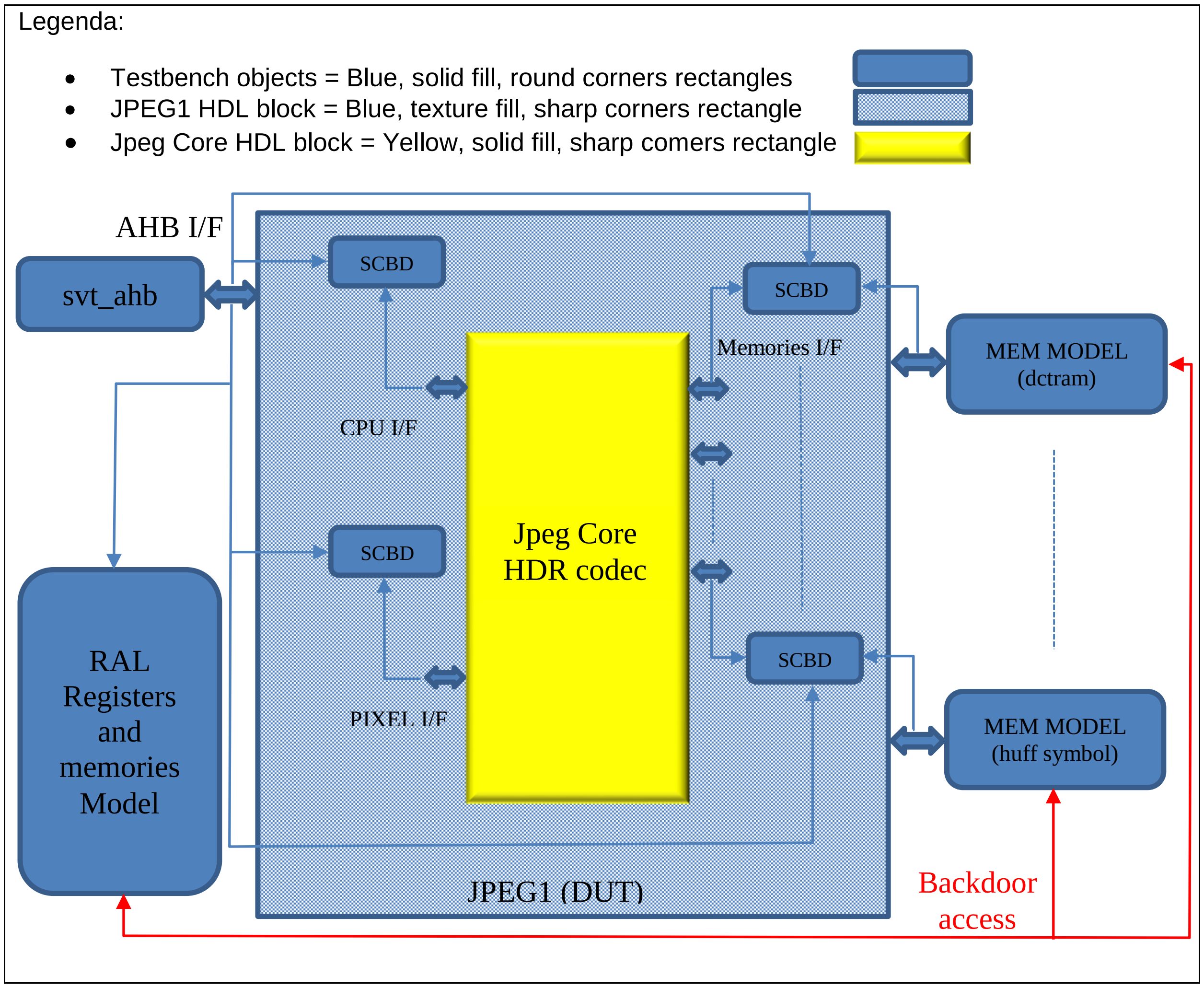

Legenda:
Testbench objects = Blue, solid fill, round corners rectangles
JPEG1 HDL block = Blue, texture fill, sharp corners rectangle
Jpeg Core HDL block = Yellow, solid fill, sharp comers rectangle
AHB I/F
svt_ahb
SCBD
CPU I/F
SCBD
PIXEL I/F
SCBD
Memories I/F
SCBD
Jpeg Core HDR codec
MEM MODEL (dctram)
MEM MODEL (huff symbol)
RAL Registers and memories Model
JPEG1 (DUT)
Backdoor access

Figure 1 Testbench architecture

The CPU I/F scoreboard

This component is aimed to verify the correct propagation of 32-bits AHB accesses towards the JPEG CORE 8-bits register interface (CPU I/F). It extends from uvm_scoreboards and implements custom translation of AHB transactions, then comparison of the translated items against the collected CPU items obtained from the CPU I/F monitor. The comparison is done in order using two uvm_fifo to store “before” and “after” items. Example code of the comparison is shown in *Table 1: CPU IF scoreboard custom comparator code example*.

```
task ahb_cpu_scoreboard::ahb_cpu_compare();
  cpu_transaction translated_tr;

  translated_tr = cpu_transaction::type_id::create("translated_tr", this);

  forever begin
    svt_ahb_transaction  ingress_tr;
    cpu_transaction      egress_tr;

    m_before_fifo.get(ingress_tr);
    m_after_fifo.get(egress_tr);

    ahb_cpu_translate(translated_tr, ingress_tr);

    if(egress_tr.compare(translated_tr))
     `uvm_info("SB/INFO", "AHB-CPU Data Compare Passed...",UVM_LOW)
    else
     `uvm_error("SB/ERR", {"AHB-CPU Data Compare Failed...\n", ingress_tr.sprint(),
egress_tr.sprint()})
  end
endtask: ahb_cpu_compare
```

Table 1: CPU IF scoreboard custom comparator code example

The memory scoreboard

There are several memories accessible both from AHB I/F and JPEG Core interfaces. Each scoreboard requires a different transformer method implementation. The scoreboards extend from uvm_scoreboards and make use of comparator class extended from uvm_in_order_comparator class. Since the memory can be accessed from two different sources the mem_in_order_comparator implements two separated “before” TLM ports, each one connected to the relevant bus monitor. It also implements a TLM port to be connected to the memory I/F monitor.
The basic methods used by this class are from custom classes passed as parameters.
This approach allows this class be reusable for every memory interface of the JPEG1 IP without the need of extending it. The user only needs to pass the right methods through the custom classes.
The JPEG IP has two sources for memory transactions (AHB and JPG CORE).
Thus, it must cope with two "before" items. These sources are mutually exclusive, preserving the ordered nature of the items.
An example code is shown in *Table 2: Reusable extension of uvm_in_order_comparator code example*.

```
class mem_in_order_class_comparator #(
  string MEM_NAME    = "GENERIC MEMORY",
  type TRANSFORMER1 = ahb2mem_transformer,
  type TRANSFORMER2 = jpg2mem_transformer,
  type C            = mem_comp
) extends uvm_in_order_comparator #(
  mem_transaction,
  C,
  uvm_class_converter #(mem_transaction),
  uvm_class_pair #(mem_transaction, mem_transaction)
);

  typedef mem_in_order_class_comparator #(MEM_NAME, TRANSFORMER1, TRANSFORMER2, C)
this_type;
  `uvm_component_param_utils(this_type)

  const static string type_name = "mem_in_order_class_comparator #(MEM_NAME,
TRANSFORMER1, TRANSFORMER2, C)";

  uvm_analysis_imp_mem_before1 #(svt_ahb_transaction, this_type) before_export1;
  uvm_analysis_imp_mem_before2 #(mem_transaction, this_type)     before_export2;
  uvm_analysis_imp_mem_after   #(mem_transaction, this_type)     aux_after_export;
...

  function void write_mem_before1(input svt_ahb_transaction ahb_tr);
    mem_transaction      translated_tr;
    svt_ahb_transaction  tmp_tr;

    tmp_tr = svt_ahb_transaction::type_id::create("tmp_tr", this);
...
      if (TRANSFORMER1::transform(translated_tr, tmp_tr)) begin
        before_export.write(translated_tr);
        before1_cnt++;
...
      end
    end
  endfunction

  function void write_mem_before2(input mem_transaction mem_tr);
    mem_transaction translated_tr;

    translated_tr = mem_transaction::type_id::create("translated_tr", this);
...
    if ((mem_tr.status == mem_transaction::IS_OK) &&
(TRANSFORMER2::transform(translated_tr, mem_tr))) begin
      before_export.write(translated_tr);
      before2_cnt++;
    end
  endfunction

  function void write_mem_after(input mem_transaction mem_tr);
    if (mem_tr.status == mem_transaction::IS_OK) begin
      after_export.write(mem_tr);
      after_cnt++;
    end
  endfunction
...

endclass: mem_in_order_class_comparator
```

Table 2: Reusable extension of uvm_in_order_comparator code example

The FIFO scoreboard

The two FIFO scoreboards (INPUT FIFO and OUTPUT FIFO) differ only for the TLM source to be connected to each scoreboard port. In the case of the INPUT FIFO the "before" port needs to be connected to the AHB bus I/F while the "after" port needs to be connected to the JPEG Core I/F. On the contrary, the OUTPUT FIFO requires its "before" port to be connected to the JPEG Core I/F while its "after" port needs to be connected to the AHB bus I/F.
The scoreboard implementation uses an uvm_in_order_comparator class extension, as done already for the memory scoreboard, with additional custom ports to be connected to each transaction source which are then re-routed towards the relevant comparator ports, depending on the scoreboard direction (input/output) which is provided as a parameter to the fifo_in_order_comparator class. The class header is shown in *Table 3: Extension of uvm_in_order_comparator allowing direction swap code example*.

```
class fifo_in_order_class_comparator #(
  string FIFO_NAME    = "GENERIC FIFO",
  fifo_direction_e  fifo_dir = AHB2JPG, //Inverse direction is: JPEG2AHB
  type TRANSFORMER1 = ahb2fifo_transformer,
  type TRANSFORMER2 = jpg2fifo_transformer,
  type C            = fifo_comp
) extends uvm_in_order_comparator #(
  fifo_transaction,
  C,
  uvm_class_converter #(fifo_transaction),
  uvm_class_pair #(fifo_transaction, fifo_transaction)
);
```

Table 3: Extension of uvm_in_order_comparator allowing direction swap code example

The working model

The testbench uses of UVM libraries v1.1. It is compiled and simulated using VCSMX (i2014.03) simulation environment. The run interactive GUI interface selected is Verdi3 (i-2014.03), the same tool is used to analyze the coverage result. The regression runs are handled using Execution Manager available in VCSMX verification framework.

We collect all required commands into a custom shell script which allow to compile, elaborate, run and analyze either a single test or the entire test suite results. The script can work either in full VCS mode or in Verdi mode.

As described in the testbench architecture, standalone component providing monitoring and coverage collection have been implemented for the core CPU interface and for the memory interfaces. Memory interfaces are of two kinds, single port access R/W SRAM like protocol and double port R and W SRAM like protocol, both models have been extended by a common component.

The Synopsys AMBA VIP (SVT_AHB) standard component has been used to model the AHB bus interface.

The Synopsys Register Abstraction Level (RAL) extension on top of the UVM registers classes has been used to model the registers.

All the above internal and external VIP have been configured and integrated using the UVM factory approach.

The environment implements the scoreboards collecting the data provided by all integrated VIPs and transforming them accordingly to the IP specification before comparing the "before" and "after" items. This has been achieved by extending the uvm_in_order_comparator using the transformers templates. Given that in our case the transactions can be originated by more than one source we have customized, where required, the number of TLM ports (the standard comparator has only one "before" and one "after" port).

The verification framework

The Synopsys verification framework was fundamental during the JPEG project execution, thanks to the very good level of control, debug and analysis provided that helped in accelerating and improving the learning process of System Verilog UVM based verification.
We provide in this section a quick excursion through the various steps required to finalize the verification closure, highlighting the strengths and the weaknesses of the Synopsys platform.

Environment Backbone integration

The first step towards the "Hello World" test is to identify, install and integrate together the standard components required by the verification environment.
The svt_ahb package needs first to be installed, using the dw_vip_setup utility, and properly configured extending the base configuration class.

The Register Abstraction Level requires a register configuration file. The register definition syntax is quite intuitive. Detailed rules and syntax are described in the "*UVM Register Abstraction Level User Guide*" document [3]. An example of register definition in RAL syntax is shown in *Table 4: RAL register and memory definition code example*.

```
block jpeg1_regmodel {
  bytes 4;
  cover +a-b+f;
...

  register JPEG_CONFR3 @'h000C { //register <name> @<offset>
    bytes 4;

    field NRST {
      bits 16;
      access rw;
      reset 'h0;
    }

    field XSIZE {
      bits 16;
      access rw;
      reset 'h0;
    }
  }
...

  //memory <memory_name> @<offset>
  memory QMEM_RAM (qmem_ram_generate_external.qmem.Mem) @'h0050 {
    size 64;
    bits 32;
    access rw;
  }
}
```

Table 4: RAL register and memory definition code example

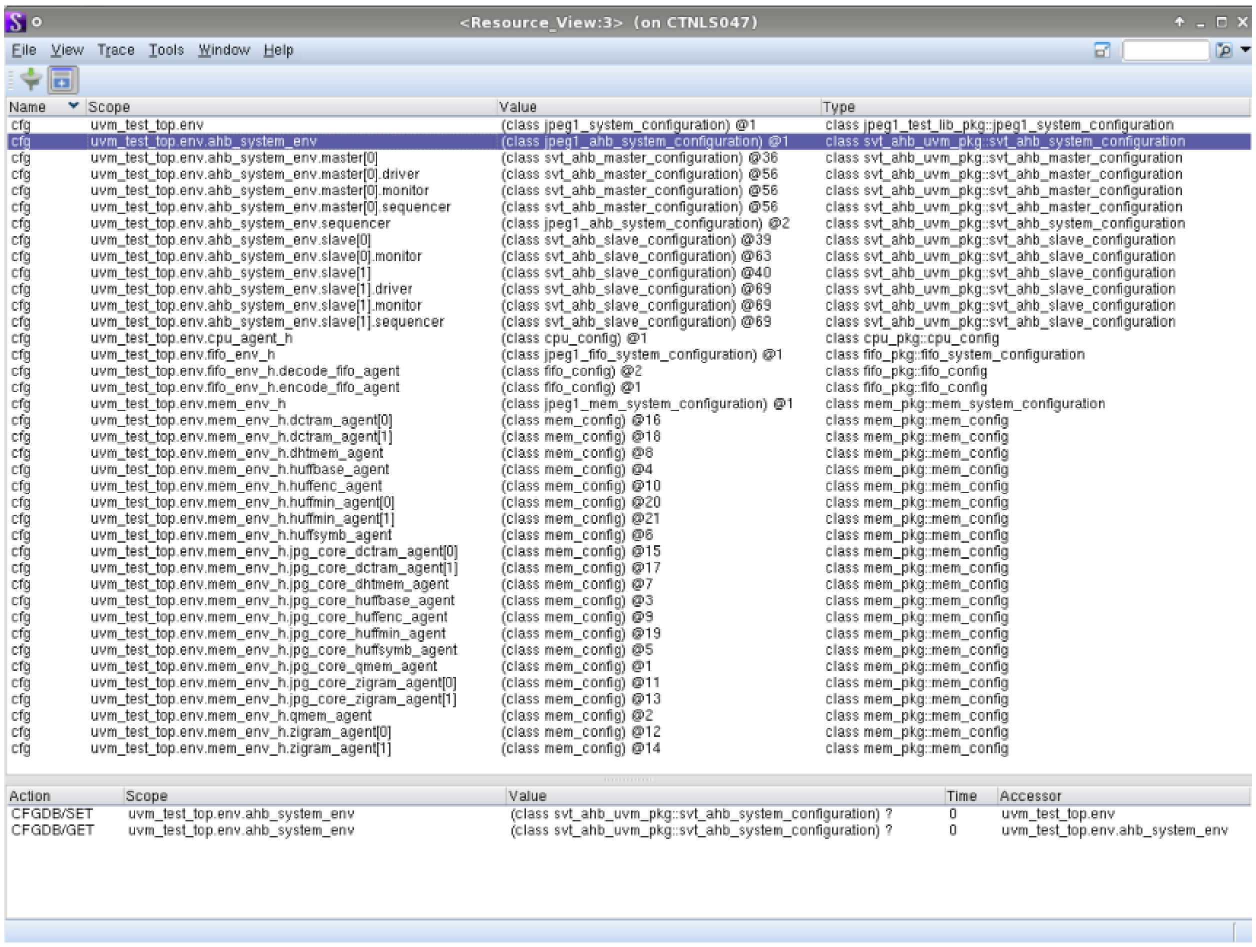
Once these two components are integrated together with the IP to be verified, by mean of interfaces definition and connection, into a basic top level environment, the first compile attempt can be executed.

Simulation binaries generation

The VCSMX tool provides separate compile commands: “vhdlan” for VHDL, “vlogan” for Verilog and System Verilog (the additional option “–sverilog” is required for SV files).
If the user wants also to use Verdi there are parallel compile commands to be used (“vericom for Verilog and SV, vhdlcom for VHDL).
The elaborate step to generate the final executable is obtained by running the “vcs” command. Additional options are required if the user wants to link the simulator to Verdi GUI interface.
The split between VCS and VERDI is introducing a bit of compile overhead, but this limitation has been removed in the latest release of VCSMX (version J-2014.12).
The integration the UVM libraries is also introducing a compile cost, which may be reduced taking advantage from VCS incremental compile mode.

Running the first simulation

Once all compile issues have been removed, it is always advised to run a simulation trial especially if the environment is configured through the UVM factory approach. This is because most of the configuration issues present in the code can only be noticed after running a simulation.
This step can be quite time-consuming for non-expert users and the VCS debug capability is very useful to highlight the root cause of the configuration issues, helping therefore to quickly solve them. This can be easily achieved using "resource/config view" tool in UVM tools menu of VCS, the best approach we found is to insert a breakpoint just after the build phase using the UVM tool "phase view" of VCS. A screenshot example is shown in *Figure 2: UVM tools resource/config view*.


<Resource_View:3> (on CTNLS047)
File View Trace Tools Window Help

Name	Scope	Value	Type
cfg	uvm_test_top.env	(class jpeg1_system_configuration) @1	class jpeg1_test_lib_pkg::jpeg1_system_configuration
cfg	uvm_test_top.env.ahb_system_env	(class jpeg1_ahb_system_configuration) @1	class svt_ahb_uvm_pkg::svt_ahb_system_configuration
cfg	uvm_test_top.env.ahb_system_env.master[0]	(class svt_ahb_master_configuration) @36	class svt_ahb_uvm_pkg::svt_ahb_master_configuration
cfg	uvm_test_top.env.ahb_system_env.master[0].driver	(class svt_ahb_master_configuration) @56	class svt_ahb_uvm_pkg::svt_ahb_master_configuration
cfg	uvm_test_top.env.ahb_system_env.master[0].monitor	(class svt_ahb_master_configuration) @56	class svt_ahb_uvm_pkg::svt_ahb_master_configuration
cfg	uvm_test_top.env.ahb_system_env.master[0].sequencer	(class svt_ahb_master_configuration) @56	class svt_ahb_uvm_pkg::svt_ahb_master_configuration
cfg	uvm_test_top.env.ahb_system_env.sequencer	(class jpeg1_ahb_system_configuration) @2	class svt_ahb_uvm_pkg::svt_ahb_system_configuration
cfg	uvm_test_top.env.ahb_system_env.slave[0]	(class svt_ahb_slave_configuration) @39	class svt_ahb_uvm_pkg::svt_ahb_slave_configuration
cfg	uvm_test_top.env.ahb_system_env.slave[0].monitor	(class svt_ahb_slave_configuration) @63	class svt_ahb_uvm_pkg::svt_ahb_slave_configuration
cfg	uvm_test_top.env.ahb_system_env.slave[1]	(class svt_ahb_slave_configuration) @40	class svt_ahb_uvm_pkg::svt_ahb_slave_configuration
cfg	uvm_test_top.env.ahb_system_env.slave[1].driver	(class svt_ahb_slave_configuration) @69	class svt_ahb_uvm_pkg::svt_ahb_slave_configuration
cfg	uvm_test_top.env.ahb_system_env.slave[1].monitor	(class svt_ahb_slave_configuration) @69	class svt_ahb_uvm_pkg::svt_ahb_slave_configuration
cfg	uvm_test_top.env.ahb_system_env.slave[1].sequencer	(class svt_ahb_slave_configuration) @69	class svt_ahb_uvm_pkg::svt_ahb_slave_configuration
cfg	uvm_test_top.env.cpu_agent_h	(class cpu_config) @1	class cpu_pkg::cpu_config
cfg	uvm_test_top.env.fifo_env_h	(class jpeg1_fifo_system_configuration) @1	class fifo_pkg::fifo_system_configuration
cfg	uvm_test_top.env.fifo_env_h.decode_fifo_agent	(class fifo_config) @2	class fifo_pkg::fifo_config
cfg	uvm_test_top.env.fifo_env_h.encode_fifo_agent	(class fifo_config) @1	class fifo_pkg::fifo_config
cfg	uvm_test_top.env.mem_env_h	(class jpeg1_mem_system_configuration) @1	class mem_pkg::mem_system_configuration
cfg	uvm_test_top.env.mem_env_h.dctram_agent[0]	(class mem_config) @16	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.dctram_agent[1]	(class mem_config) @18	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.dhtmem_agent	(class mem_config) @8	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.huffbase_agent	(class mem_config) @4	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.huffenc_agent	(class mem_config) @10	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.huffmin_agent[0]	(class mem_config) @20	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.huffmin_agent[1]	(class mem_config) @21	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.huffsymb_agent	(class mem_config) @6	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_dctram_agent[0]	(class mem_config) @15	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_dctram_agent[1]	(class mem_config) @17	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_dhtmem_agent	(class mem_config) @7	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_huffbase_agent	(class mem_config) @3	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_huffenc_agent	(class mem_config) @9	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_huffmin_agent	(class mem_config) @19	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_huffsymb_agent	(class mem_config) @5	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_qmem_agent	(class mem_config) @1	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_zigram_agent[0]	(class mem_config) @11	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.jpg_core_zigram_agent[1]	(class mem_config) @13	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.qmem_agent	(class mem_config) @2	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.zigram_agent[0]	(class mem_config) @12	class mem_pkg::mem_config
cfg	uvm_test_top.env.mem_env_h.zigram_agent[1]	(class mem_config) @14	class mem_pkg::mem_config

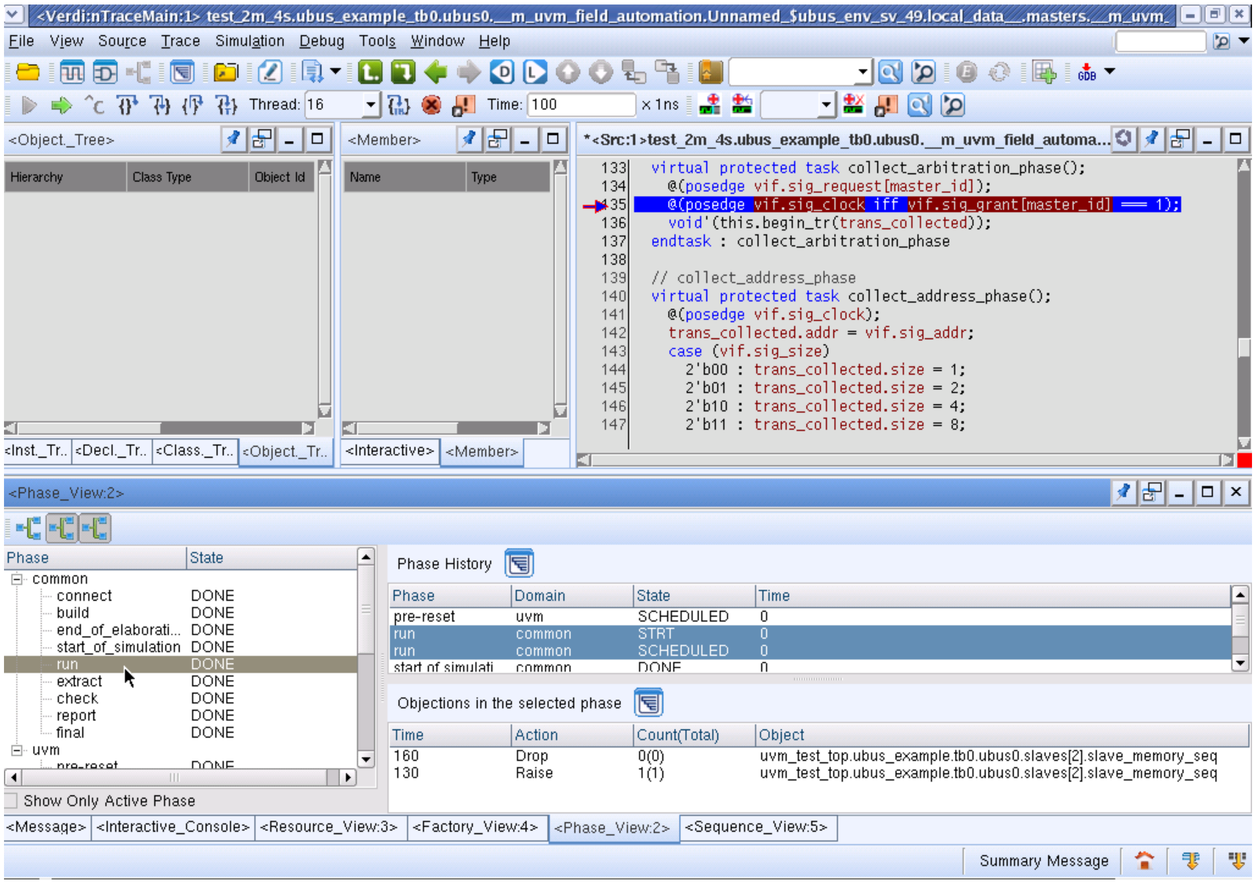
Action	Scope	Value	Time	Accessor
CFGDB/SET	uvm_test_top.env.ahb_system_env	(class svt_ahb_uvm_pkg::svt_ahb_system_configuration) ?	0	uvm_test_top.env
CFGDB/GET	uvm_test_top.env.ahb_system_env	(class svt_ahb_uvm_pkg::svt_ahb_system_configuration) ?	0	uvm_test_top.env.ahb_system_env

Figure 2: UVM tools resource/config view

Debug failure scenarios

Among all verification implementation phases, debug is one of the most time-consuming.
In the initial testbench development phase, issues found in both the testbench and the design are usually quite easy to investigate. While the development progresses, advanced debug capability are requested to help solving the complex failure scenarios.

The user-friendly debug interface of Verdi and the capability of navigating the full hierarchy of both SV testbench and design code is facilitating this task. UVM objects probing and visualization, together with breakpoint insertion and step-by-step debugging are the VERDI features we found more helpful.
Verdi UVM debug provides visibility on the different phases, on the sequences being executed, on the content of the factory and on the resources of the testbench. An example view is showed in Figure 3: Verdi3 UVM Debug.



<Verdi:nTraceMain:1> test_2m_4s.ubus_example_tb0.ubus0.__m_uvm_field_automation.Unnamed_$ubus_env_sv_49.local_data__.masters.__m_uvm_
File View Source Trace Simulation Debug Tools Window Help
Thread: 16
Time: 100 x 1ns
<Object_Tree>
Hierarchy Class Type Object Id
<Member>
Name Type
*<Src:1>test_2m_4s.ubus_example_tb0.ubus0.__m_uvm_field_automa...
<Inst._Tr.. <Decl._Tr.. <Class._Tr.. <Object._Tr.. <Interactive> <Member>
<Phase_View:2>
Show Only Active Phase
<Message> <Interactive_Console> <Resource_View:3> <Factory_View:4> <Phase_View:2> <Sequence_View:5>
Summary Message

```
virtual protected task collect_arbitration_phase();
  @(posedge vif.sig_request[master_id]);
  @(posedge vif.sig_clock iff vif.sig_grant[master_id] == 1);
  void'(this.begin_tr(trans_collected));
endtask : collect_arbitration_phase

// collect_address_phase
virtual protected task collect_address_phase();
  @(posedge vif.sig_clock);
  trans_collected.addr = vif.sig_addr;
  case (vif.sig_size)
    2'b00 : trans_collected.size = 1;
    2'b01 : trans_collected.size = 2;
    2'b10 : trans_collected.size = 4;
    2'b11 : trans_collected.size = 8;
```

Phase	State
common	
connect	DONE
build	DONE
end_of_elaborati...	DONE
start_of_simulation	DONE
run	DONE
extract	DONE
check	DONE
report	DONE
final	DONE
uvm	
pre-reset	DONE

Phase History

Phase	Domain	State	Time
pre-reset	uvm	SCHEDULED	0
run	common	STRT	0
run	common	SCHEDULED	0
start of simulati	common	DONE	0

Objections in the selected phase

Time	Action	Count(Total)	Object
160	Drop	0(0)	uvm_test_top.ubus_example.tb0.ubus0.slaves[2].slave_memory_seq
130	Raise	1(1)	uvm_test_top.ubus_example.tb0.ubus0.slaves[2].slave_memory_seq

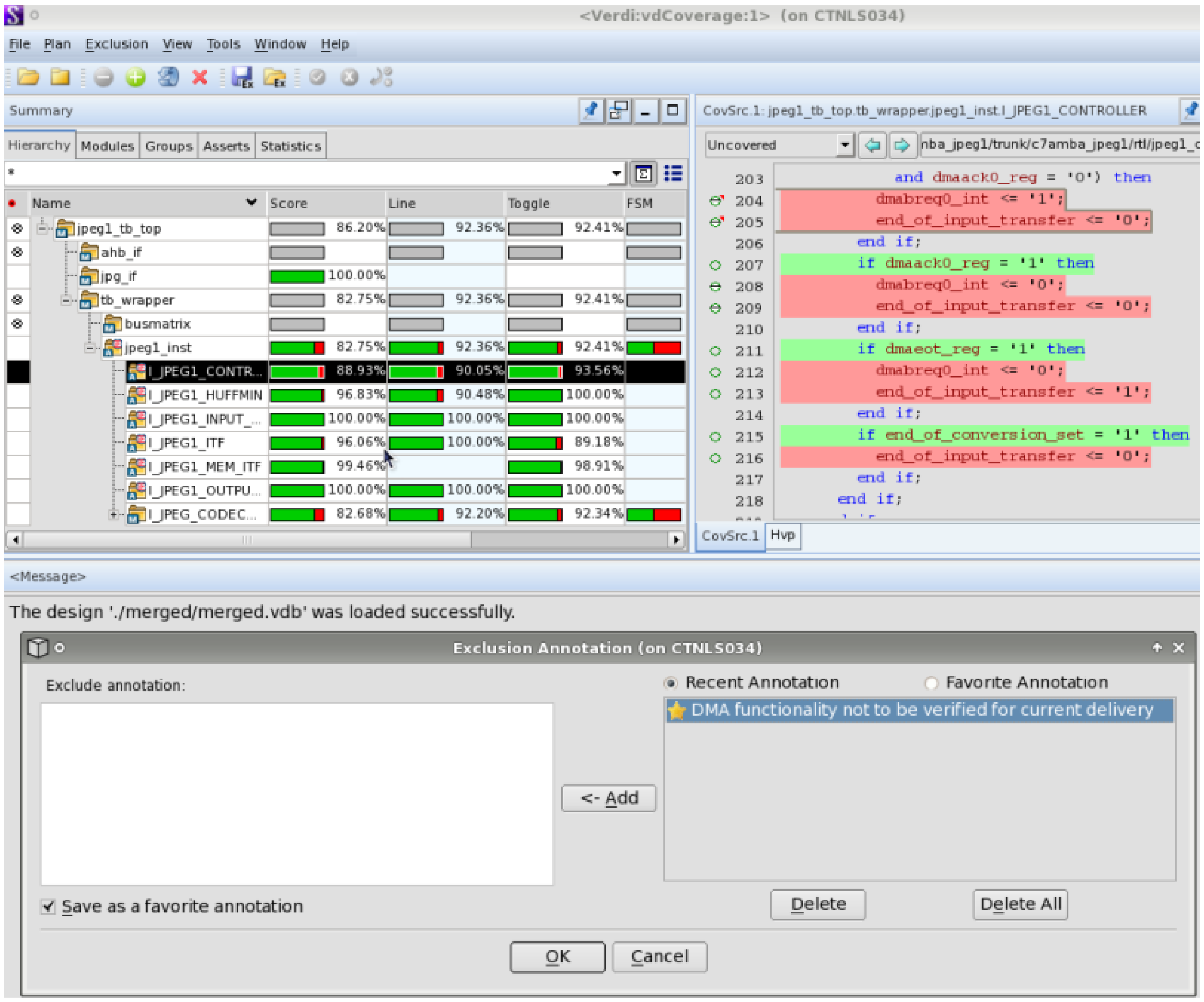
Figure 3: Verdi3 UVM Debug

Managing regressions

Constrained random verification is based on the power of randomizing in an automatic way test scenarios, minimizing the human effort into developing complex direct tests manually.
To exploit at the best this powerful feature, test suite randomization and automatic execution are a must.
Execution Manager handles this task in a quite simple but effective way. It provides a very good compromise between automation and user customization, allowing an effective regression suite handling. The tool can be instead improved for what concerns failures analysis, by providing more advanced error collection capabilities.

Coverage analysis

A good functional coverage implementation is fundamental in constrained random verification, but even more important is the capability to analyse in a comprehensive way code coverage results together with functional coverage ones. The Verdi coverage analysis interface provides good facilities to analyse, annotate and review the achieved coverage results.
The JPEG project design was implementing all Specification features, but the verification excluded one of the specified and designed feature from the sign off coverage criteria because not used in the first design integration release. So the same feature required to be excluded from the coverage target, leading to many RTL uncovered code lines to be justified.
We found useful the capability to save preferred annotation and being able to reuse those whenever required. A screenshot of Verdi coverage tool, displaying also the preferred exclusion notes is shown in *Figure 4: Verdi Coverage GUI showing preferred exclusion annotation*.



<Verdi:vdCoverage:1> (on CTNLS034)
File Plan Exclusion View Tools Window Help
Summary
Hierarchy Modules Groups Asserts Statistics
CovSrc.1: jpeg1_tb_top.tb_wrapper.jpeg1_inst.I_JPEG1_CONTROLLER
Uncovered
nba_jpeg1/trunk/c7amba_jpeg1/rtl/jpeg1_c
CovSrc.1 Hvp
<Message>
Exclusion Annotation (on CTNLS034)
Exclude annotation:
Recent Annotation
Favorite Annotation
<- Add
Save as a favorite annotation
Delete
Delete All
OK
Cancel

Name	Score	Line	Toggle	FSM
jpeg1_tb_top	86.20%	92.36%	92.41%	
ahb_if				
jpg_if	100.00%			
tb_wrapper	82.75%	92.36%	92.41%	
busmatrix				
jpeg1_inst	82.75%	92.36%	92.41%	
I_JPEG1_CONTR...	88.93%	90.05%	93.56%	
I_JPEG1_HUFFMIN	96.83%	90.48%	100.00%	
I_JPEG1_INPUT_...	100.00%	100.00%	100.00%	
I_JPEG1_ITF	96.06%	100.00%	89.18%	
I_JPEG1_MEM_ITF	99.46%		98.91%	
I_JPEG1_OUTPU...	100.00%	100.00%	100.00%	
I_JPEG_CODEC...	82.68%	92.20%	92.34%	

```
                and dmaack0_reg = '0') then
              dmabreq0_int <= '1';
              end_of_input_transfer <= '0';
            end if;
            if dmaack0_reg = '1' then
              dmabreq0_int <= '0';
              end_of_input_transfer <= '0';
            end if;
            if dmaeot_reg = '1' then
              dmabreq0_int <= '0';
              end_of_input_transfer <= '1';
            end if;
            if end_of_conversion_set = '1' then
              end_of_input_transfer <= '0';
            end if;
          end if;
```

The design './merged/merged.vdb' was loaded successfully.

DMA functionality not to be verified for current delivery

Figure 4: Verdi Coverage GUI showing preferred exclusion annotation

Also the exclusion file text format is quite readable, and this allows to directly use it in coverage report sign-off documentation. An extract from JPEG coverage exclusion file is shown in *Table 5: Exclusion file example annotation.*

```
CHECKSUM: "3805157617 1249220498"
INSTANCE: jpeg1_tb_top.tb_wrapper.jpeg1_inst.I_JPEG1_CONTROLLER
ANNOTATION_BEGIN: "DMA functionality not to be verified for current delivery"
Block 10 "dmabreq0_int <= '1';"
Block 12 "dmabreq0_int <= '0';"
Block 14 "dmabreq0_int <= '0';"
Block 16 "end_of_input_transfer <= '0';"
Block 63 "dmabreq1_int <='1';"
Block 64 "if (conv_integer(current_output_fifo_level) = output_burst_size_d"
Block 65 "dmalbreq1_int <= '1';"
Block 66 "dmasreq1_int  <= '1';"
Block 67 "dmasreq1_int  <= '0';"
Block 68 "if dmaack1_reg = '1' then"
Block 69 "dmabreq1_int  <= '0';"
ANNOTATION_END
```

Table 5: Exclusion file example annotation

4. Achieved results

The project has been signed-off with full coverage results after justification, implemented by mean of the Verdi Coverage annotated exclusion file (one IP feature not requested by the current product has not been verified).

During the verification project we found 10 RTL bugs which required RTL fixes plus 2 minor specification issues.

The project took overall 8 months of work involving two verification engineers. This time includes the required knowledge ramp up in SV, UVM and tools of the engineers plus the implementation time itself.

We reused 2 existing packages (RAL extension of UVM registers and SVT_AHB) and implemented 3 passive VIP interfaces with 3 associated configurable scoreboards.

Passive interfaces have been configured and instantiated many times, for a cumulative total of 33 instances.

Scoreboards have been instantiated overall 14 times (1 CPU, 2 FIFO and 11 memories).

The memory VIP implemented by us has already been reused in two new projects, not related to JPEG1 codec.

5. Next steps

To fully validate the IP functionality we will implement the missed feature verification. This require a new model to be developed which will drive the bus interface through usage of svt_ahb, this will give us the opportunity to explore sequence layering.
Once the IP is fully validated it will also make sense to run Certitude flow to qualify the overall testbench quality.

6. Recommendations

System Verilog on its own is a very powerful language, nevertheless the adoption of the standard proposed by UVM library and its related methodology is fundamental to increase code reusability and configurability; therefore we strongly recommend to fully adopt it.
The use of VCS UVM Tools will allow the user to reduce the overall extra effort required to integrate the UVM layer in the testbench environment.

7. Conclusions

The ramp up from scratch of the two verification engineers, having different skill levels, in constrained random verification techniques was quite fast, achieving an overall good level of understanding of SV UVM structure.

The project was completed with full quality into a reasonable time frame taking of course into account the overhead due to the ramp up requirements.

The Synopsys verification platform provides full debug and analysis capability of the verification achieved results. It did not show any particular limitations into the UVM standard support.

The SV UVM verification approach is very structured. The rigid structure initially may be perceived as a limitation, but it is then very useful when the environment grows because it helps to avoid unreadable code. The UVM factory approach provides easiness of configurability and helps into maximize code reuse.

8. References

[1] *Universal Verification Methodology (UVM) 1.1 Class Reference*, June 2011, Accellera, available at http://www.accellera.org/downloads/standards/uvm

[2] *Universal Verification Methodology (UVM) 1.1 User's Guide*, May18 2011, Accellera, available at http://www.accellera.org/downloads/standards/uvm

[3] *UVM Register Abstraction Level User Guide ,* ${VCS_HOME}/doc/UserGuide/pdf/uvm_ralgen_ug.pdf