1,2,3,...8 simple steps towards a single digital signal processing testbench supporting heterogeneous interfaces and datatypes

Nico Lugil, Keysight Technologies, Rotselaar, Belgium (nico_lugil@keysight.com)

***Abstract*—Using UVM to test digital signal processing (DSP) modules creates some particular challenges and opportunities. The high-level stimulus and analysis to test these DSP modules is often quite ignorant of the transformation the DUT is doing. The DUT interfaces are often of the same kind, but can differ in their parameterization. Also the datatype that a module accepts can differ. This paper presents an architecture that allows keeping the bulk of the testbench code common for all the DSP modules. Two important aspects of this architecture are its support for multiple, differently parameterized interfaces of the same kind and similar scalability for sending different kind of datatypes over these interfaces.**

Keywords*—*DSP; UVM; parameter; abstract; datatype;

I. Introduction

Our ASICs consist of multiple DSP modules applying mathematical transformations on incoming data signals. The interfaces and datatypes sent to these modules are DUT or test specific, but the high-level stimulus needs are very common between all these modules. The next sections discuss the differences and similarities motivating building a single flexible testbench architecture. Whilst the focus of this paper is on DSP modules, the scalable architecture is also applicable elsewhere.

II. PROBLEM STATEMENT

A. Interfaces and datatypes

All our modules have a similar set of interfaces: a control interface plus multiple data-in and data-out interfaces. Some modules can process multiple data channels in parallel. A single data interface can be used to multiplex multiple data channels over it. There are 2 important parameters specifying the data interface: the number of data channels that can be multiplexed over the interface and the width of the interface which determines how many samples can be transferred in parallel in a clock cycle. A channel carries a certain datatype which is specified by the sample type (real, complex, polar …) and the number of bits used to represent such a sample. A complex datatype example is shown below:

```
class complex_sample#(int W);
    bit [W-1:0] re;
    bit [W-1:0] im;
endclass
```

All channels multiplexed over an interface can have a different datatype. And for a given channel, the datatype can change during the course of a simulation. So although each module has similar interfaces, there are important differences:

- DUT specific: the number of data input and data output interfaces
- Interface specific: the bus width of the interface (BW), number of channels that can be multiplexed over it (N_CH).
- Dynamically per channel: the datatype used by the channel, characterized by the sample type (T) and the sample width (W)

B. Need for a single, sharable testbench

Although the low-level interface parameters and the datatype are DUT specific, the high-level stimulus needs are very similar for each DUT. For example a common scenario is to send multiple sine waves on each data input interface, collect an integer number of periods on the output interfaces and analyze the response. The UVM testbench is also used as an environment to develop and debug the designs, often with more directed tests. Also here identical stimulus and sequence needs are present.

We want to avoid having multiple testbench code sets for the different DSP modules, mainly for maintenance and ease of adding features. For example if we want to add a new feature to the testbench, we most likely want to do this for all the DSP modules. This paper will present how we can make a single UVM testbench code set that with just a couple of parameters and interface definitions can be reused for all the modules (focusing on the stimulus parts).

In the next sections we will show the different steps needed to create such a scalable solution, starting from a non-scalable, non-reusable solution.

III. GENERAL ARCHITECTURE COMPONENTS

The major components and objects in the testbench are described in this section in a simplified way. From the test, we configure and/or override the components and objects. This is also were we start sequences to send stimulus in the DUT. The environment instantiates among other things the different interface agents.

Every channel that wants to send data in the DUT is represented by a (source) sequence. Channels are identified by a channel ID member of the sequence. These are the lowest-level sequences in the testbench. The sequence items generated by the sequence contain the data samples that can be sent over the bus in 1 clock cycle for that channel plus a channel ID to which channel the data belongs. The sequence is unaware that the driver multiplexes multiple channels over a single interface.

The data that the sequence sends to the driver is generated by a waveform generator. These are configurable objects to create the different kind of waveforms (sine wave, ramp, random samples ...) that need to be sent to the DUT. They all have a virtual method `getNextData` to get the next N samples. There are specific waveform generators for different wave datatypes (real, complex, polar…). This `getNextData` method is called by the sequence and the sequence puts these samples in the sequence item for the driver.

The driver multiplexes the different sequences that want simultaneous access to the bus. The samples of a selected sequence are placed on the interface bus by the driver. As usual the interface defines all signals passed between the DUT and testbench.

Finally we have parameter package containing global parameters to be used by several parts of the testbench.

IV. 8 STEPS

A. *Step1: at the interface, samples are represented as a bunch of bits + the non-scalable starting point*

Here we will take our first step and create a non-scalable baseline of the above described architecture.

1) Step1

A first obvious thing we need to do is to make the bus of the interface unaware of the datatype that will be send over it. We cannot make the bus a specific datatype as the different channels multiplexed over the bus use different datatypes, and even within a single channel the datatype can change over time during a test. So we make the payload of the interface just a generic bunch of bits.

These bits originally come out of a waveform as meaningful samples of some datatype. Hence somewhere we need to pack these samples into the generic bits.

2) The non-scalable starting point

First we decide where to pack the samples into bits. The driver's task is to multiplex multiple channels on the bus, but it doesn't need to know what a channel is sending. So we decided to do the packing in the sequence. This also makes the sequence item and driver ignorant of the datatype each sequence is actually sending.

As starting point, we let the parameter package have global parameters for BW and N_CH. The interface then uses these parameters. A channel ID is sent together with the actual data, the number of bits depends on N_CH.

```
interface source_if(input bit clk);
        bit [BW-1:0] payload;
        bit [$clog2(N_CH)-1:0] channelID;
        bit WEn;
        …
endinterface
```

Also the sequence item uses these parameters. It carries the packed bits and an ID to identify from which channel it originates:

```
class source_seq_item_REQ extends uvm_sequence_item;
    rand bit [BW-1:0] payload;
    rand bit [$clog2(N_CH)-1:0] chID;
endclass
```

The driver, for its protocol needs to buffer data from sequences, and needs to have for example a counter to arbiter between sequences, hence it also used the BW and N_CH parameters. The driver gets a virtual interface handle via the config database as suggested by the standard UVM flow.

```
class source_driver extends uvm_driver#(source_seq_item_REQ, source_seq_item_RSP);
    virtual source_if SRC;
    local bit [$clog2(N_CH)-1:0] arbiterCntr;
    local bit [BW-1:0] buffer[$clog2(N_CH)][5];   // 5 deep buffer per channel

    extern task run_phase(uvm_phase phase);
     …
endclass
```

To handle different datatypes, we naively parameterize the sequence with the datatype. The sequence has a handle to a waveform that can generate a particular datatype. In the test we will assign a particular (derived) waveform generating this datatype (e.g. a sine wave). The sequence asks the waveform to generate the next samples to be transmitted and calls the correct packer on them to get the packed bits for the sequence item for the driver. The packing is datatype dependent, but also on the bus width, and uses parameters for this.

```
class source_seq#(type T)
    …
    waveform_typed#(T) wave;     // any waveform generating the datatype T
    int N;                       // number of samples that can be transferred at once

    task body();
        T Data[];
        req=source_seq_item_REQ::type_id::create("req");
        do begin
           wave.getNextData(N,Data);    // returns N data samples of type T in Data
           start_item(req);
           DataTypes_packer#(T,BW,T::W)::pack(Data,req.payload); // packs into BW bits
           finish_item(req);
        end while(stop_condition_not_satisfied());
    endtask
endclass
```

3) Issues

Whilst this setup works for simple cases with homogeneous interfaces, it does have several issues. Because parameters like BW and N_CH are global, you can instantiate multiple source agents, but they all need to share the same parameters. The env hence doesn't scale to DUTs having heterogeneous source interfaces. Next there is a lot of parameter proliferation all over the testbench and even worse also in the tests. We can't easily make arrays of sequences if they have to send different datatypes T. This makes it very difficult to make for example general purpose virtual sequences that handle multiple interfaces or channels. In general because the sequences are parameterized with T it is very difficult to reuse and scale test code. Maybe not apparent from the code, but the waveform generation mechanism is also very complicated. Multiple layers of derived classes, lots of $cast, type checking, etc...

Some of these issues can be worked around to some extend by having multiple levels of derivation for some classes where the base classes only have members and methods that do not depend on the specific T parameter. However this is still cumbersome and still leaves many issues.

B. Step2: make sequence unaware of the datatype & split tasks between wave data, waveform and sequence

We can do a better job in splitting responsibilities between wave datatypes, waveform generators and the sequence. The new structure is presented below.

We let each wave datatype have members to represent the specific samples and if they need to packed, provide a pack method for them. All the wave datatypes derive from a `wave_data` base class having a virtual pack function.

```
virtual class wave_data;
    virtual function void pack(output bit[BW-1:0] out);  // uses BW parameter
    ...
endclass
```

Packable types then implement this pack function for the specific data it contains, for example:

```
class wave_data_complex_W#(int W) extends wave_data;
    typedef struct {bit signed [W-1:0] Re; bit signed [W-1:0] Im;} complex_number;
    complex_number d[];

    virtual function void pack(output bit[BW-1:0] out);  // uses BW parameter
        DataTypesPacker#(BW,W)::pack_Complex(d,out);
    endfunction
endclass
```

The waveform generator is responsible for filling the wave data content (sine wave, counter, etc…) and it keeps a handle to this data. All waveform generators derive from a `wavegen` base class which has a pure virtual getNextData function and it has a handle to the produced data. This handle is of the `wave_data` type.

```
virtual class wavegen;
    pure virtual function void getNextData(int N);
    wave_data outputData;     // base class pointer to data it generates
endclass
```

A derived waveform generator example to create a complex counter:

```
class wavegen_counter_complex_W#(int W) extends wavegen;
    bit [W-1:0] cnt[2];

    function void getNextData(int N);
        wave_data_complex_W#(W) data=new;
        data.d=new[N];
        foreach(data.d[i]) begin
            data.d[i].Re=cnt[0];      data.d[i].Im=cnt[1];
            cnt[0]++;                 cnt[1]++;
        end
    endfunction
    outputData=data;  // put in base class handle
endclass
```

Now the sequence no longer needs to be parameterized with T as it doesn't need to know anything about waveform or datatype. It just asks data to be generated to the waveform generator via the virtual getNextData method, and calls the pack function on the wave data handle in that generator using the virtual pack method.

```
class source_seq;
    …
    wavegen wave;      // any waveform generating any kind of wave data
    int N;             // number of samples that can be transferred at once

    task body();
        req=source_seq_item_REQ::type_id::create("req");
        do begin
           wave.getNextData(N);        // N samples are stored inside wave.outputData
           start_item(req);
```

```
                wave.outputData.pack(req.payload);     // pack
                finish_item(req);
             end while(stop_condition_not_satisfied());
         endtask
     endclass
```

The sequence is however still indirectly specialized to the global BW parameter (e.g. the pack function returns a bit vector of that size).

A big advantage is that we can now use the same type of sequence for all our channel, even if they are associated with different waveforms and/or wave data types. We can now for example make arrays of such sequences, which makes writing generic virtual sequences or test code a lot easier and more scalable.

C. Step3: arrays of agents with IDs

Some modules require multiple source agents. It is pretty straightforward to support any number of source interfaces by adding an extra global parameter `N_SOURCE_INTERFACES` to our parameter package. The env config object now gets an array of source agent config objects. The env itself has an array of these agents.

```
class basic_env extends uvm_env;
    ...
    source_agent my_source_agent[N_SOURCE_INTERFACES];

    function void build_phase(uvm_phase phase);
        ...
        foreach(my_env_config.my_source_agent_config[i]) begin
            uvm_config_db#(source_agent_config)::set(this,"my_source_agent*",
                 $sformatf("source_agent_config[%0d]",i),env_cfg.my_source_agent_config[i])
            my_source_agent[i]=source_agent::type_id::create(
                 $sformatf("my_source_agent[%0d]",i),this);
            my_source_agent[i].ID=i;
        end
    endfunction
endclass
```

The env had to give each agent its ID, which it uses to retrieve the correct source agent config.

```
class source_agent extends uvm_component;
    int ID=-1;      // an invalid ID
    function void build_phase(uvm_phase phase);
        uvm_config_db#(source_agent_config)::get(this,"",
             $sformatf("source_agent_config[%0d]",ID), m_cfg)
    endfunction
endclass
```

We now have multiple agents, easily accessible from the test by array indices, however all these agents share the parameters from the parameter package, so they are all identically parametrized.

D. Step4: parameterize the interface

Instead of using global parameters for the interface definition, we can parameterize the interface so that each instance of the interface can have different parameters.

```
interface source_if#(int N_CH, int BW) (input bit clk);
    ...
endinterface
```

A consequence of doing this, if combined with using virtual interfaces, is that it this leads to parameters all over the place: config objects, drivers, sequences, agents, ... they all need to be parameterized as well so that they can use the parameterized virtual interface. This is a lot of type work, and hinders the scalability as it breaks step3 as we can't make arrays of those differently parameterized agents anymore as they are basically different object types. Because we send over the monitored bits of size BW to analysis components it also influences the transaction type going to analysis components, hence also these analysis (ex)ports become parameterized. This is a situation we absolutely want to avoid and will be solved in step5.

E. Step5: replace virtual interface by abstract/concrete class model

An alternative approach to virtual interfaces, is to use the abstract/concrete class approach. This effectively allows using interfaces in a polymorphic way. This technique is described in [1][2]. The interface signals are accessed indirectly through an API defined in an abstract class. The implementation of this abstract class is done in a derived class inside the interface. The interface also creates an instance of this derived class. What is given to the UVM components (via the config db) is the abstract class handle instead of a virtual interface. Via polymorphism the API calls end up in the correct derived class. Putting the handle in the config database, can be done from the interface itself. The abstract API and corresponding interface look as follows:

```
virtual class source_abs_c extends uvm_object;
    ...
    pure virtual function void set_WEn(bit _Wen);
    pure virtual function void set_payload(bit[BW-1:0] _payload);
endclass

interface source_if#(int N_CH, int BW) (input bit clk);
     bit WEn;
     bit [BW-1:0] payload;
     ...

     clocking cb @(posedge clk);
         ...
     endclocking

     // concrete implementation of abstract API
     class concrete_c extends source_agent_pkg::source_abs_c;
         virtual function void set_WEn(bit _WEn);
             cb.WEn<=_WEn;
         endfunction
         ...
     endclass

     // concrete class instance and allocator
     concrete_c concrete_c_inst;
     function source_agent_pkg::source_abs_c get_source_concrete_c_inst();
          if(concrete_c_inst==null)
                concrete_c_inst=new();
          return (concrete_c_inst);
     endfunction

     // method to add class handle to config db
     function void add_to_config_db();
          uvm_config_db #(source_agent_pkg::source_abs_c)::set(null,
              "uvm_test_top", "source_abs_c", get_source_concrete_c_inst());
     endfunction
 endinterface
```

If we want to have multiple of these interfaces (possibly with different parameters) we need a mechanism for identifying the correct abstract class handle in the config db. We can do so by giving the interface an ID, and use this interface ID to name the config object (this ID not related to the channel ID):

```
interface source_if#(int ID, int N_CH, int BW) (input bit clk);
    …
    uvm_config_db #(source_agent_pkg::source_abs_c)::set(null,
             "uvm_test_top", $sformatf("source_abs_c[%0d]",ID),
              get_source_concrete_c_inst());
    …
endinterface
```

The abstract class handle now takes the role of the virtual interface. Some snippets from the config object and how to retrieve it from the config db are shown below. The agent/interface ID from step 3 is used to index the correct abstract class handle in the config db. The driver talks to the interface via the abstract API instead of via the parameterized virtual interface.

```
class source_agent_config extends uvm_object;
      ...
      source_abs_c my_source_abs_c;
```

```
  endclass

  class source_driver extends uvm_driver #(source_seq_item_REQ, source_seq_item_RSP);
        source_abs_c my_source_abs_c;

        function void build_phase(uvm_phase phase);
            uvm_config_db #(source_agent_config)::get(this, "",
                  $sformatf("source_agent_config[%0d]",ID), m_cfg)
            my_source_abs_c=m_cfg.my_source_abs_c;
        endfunction

        function void run_phase(uvm_phase phase);
            ...
            my_source_abs_c.set_WEn(1);
        endfunction
   endclass
```

Although the interface is still parameterized, the abstract API class is not, hence the agent components do not need to take over the interface parameters. This solves many of the issues with the parameterized virtual interface of step 4. However not all parameters have disappeared from our agent's components.

- The BW parameter is still needed by the sequence, sequence item and driver in the size of the bits vector carried from the packer to the interface method call. This can also be seen in the abstract class definition in the `set_payload` method.
- Also the packers in the wave data types use this parameter.
- The driver still needs to know the N_CH value for its arbitration, e.g. the maximum count value.

Steps 6 and 7 will solve this.

F. *Step6: not all parameters need to be actual systemverilog parameters*

For example the maximum count value for the arbitration counter can be just an integer that is used by the driver. We can provide a method in the abstract class to get this value and call this in the driver at the start of the run phase or before:

```
max_arbiterCnt=source_abs_c.get_max_arbiterCnt();
```

In the interface this method simply returns the N_CH parameter. So at the interface level this is a parameter, whilst for the driver this is just a plain variable that gets its value at runtime.

G. *Step7: replace fixed size bit vectors by more generic types or dynamic types.*

The packer doesn't really need to generate exactly this size bit vector. It can return a dynamic array of integers for example representing the packed bits. For example if BW were 100, the packer could return a dynamic array of 4 integers holding these bits. If we do this, the packer BW parameter can also become a simple argument to the packer instead of a parameter:

```
DataTypesPacker#(BW,W)::pack_Complex(in,out);     // out is of type: bit[BW-1:0] out;
```

becomes

```
DataTypesPacker#(W)::pack_Complex(BW,in,out);   // out is of type: output int out[];
```

The BW parameter can be obtained from the interface like in step 5. The dynamic array needs to be new'd to the correct size only once. The sequence item now also carries a dynamic array of integers, and the abstract class API `set_payload` method now also takes this dynamic array.

```
pure virtual function void set_payload(int _bus[]);
```

The implementation of this method inside the interface turns the array of integers into the bit vector of size BW.

There are 2 alternatives: instead of using dynamic arrays you could use a maximum footprint sized bit vector and only use the bits you need. This means you need to come up with a 'maximum ever needed number', and for smaller bitvectors, a lot of useless bits are present in the simulation. The other alternative is that the packing can actually move into interface, so we keep the `wave_data` handle until into the interface, and the interface calls the virtual pack function instead of the sequence:

```
pure virtual function void set_payload(wavegen_data _payload)
```

This means we only need to transport a handle from sequence to interface instead of all the bits.

H. *Step8: include the correct interface instantiations in the testbench*

At this moment, the testbench is very scalable and can be shared between modules, we just need to have an easy mechanism to tune it to the module specific interfaces. For this every module has a file with content as in the next example (e.g. a file called **src_ifs.sv**), defining the interfaces it will use.

```
source_if#(0,4,100) my_source_if_0(clk);
source_if#(1,1,50) my_source_if_1(clk);

function void add_ifs_to_config_db();
    my_source_if_0.add_to_config_db();
    my_source_if_1.add_to_config_db();
endfunction
```

The toplevel module where we typically instantiate interfaces, dut, and start the uvm_test includes this file:

```
module top;
    `include "src_ifs.sv"
    initial begin
        add_ifs_to_config_db();
        // add other stuff to config db
        run_test()
    end
endmodule
```

In the compilation script we just need to make sure the correct include file is picked up.

V. CONSLUSION AND FUTURE WORK

This paper presented the stimulus part of a scalable and reusable DSP testbench infrastructure. It proposes 8 simple steps that result in a testbench which has 3 key features: first it provides an easy and scalable mechanism for instantiating any number of heterogeneously parameterized interfaces. Secondly it proposes a structure allowing heterogeneous data types to flow over such interfaces. Thirdly it tries to unify the testbenches needed for different DSP modules and hence the bulk of our stimulus testbench code can be shared among different DUTs. Whilst the focus of this paper is on DSP modules, the scalable architecture is also applicable elsewhere.

Concerning the waveform generation, the described setup shows the basics only, but there were more challenges to solve. It feels like there is an opportunity to come up with a standard way of doing such things in UVM.

The described methodology is currently being used for the development of a new ASIC where this testbench is successfully used for more than 20 DSP modules.

This paper focused on the stimulus part. This is where we can gain the most in having a unified code set for the different module testbenches. Nevertheless also the analysis part, register model, etc... have opportunities for sharing as much as possible testbench code between the different modules.

REFERENCES

[1] David Rich and Jonathan Bromley, “Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches” DVCon 2008

[2] Shashi Bhutada, “Polymorphic Interfaces,” Verification Horizons, Volume 7, Issue 3, 2011