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

Nico Lugil, Keysight Technologies


accellera
SYSTEMS INITIATIVE

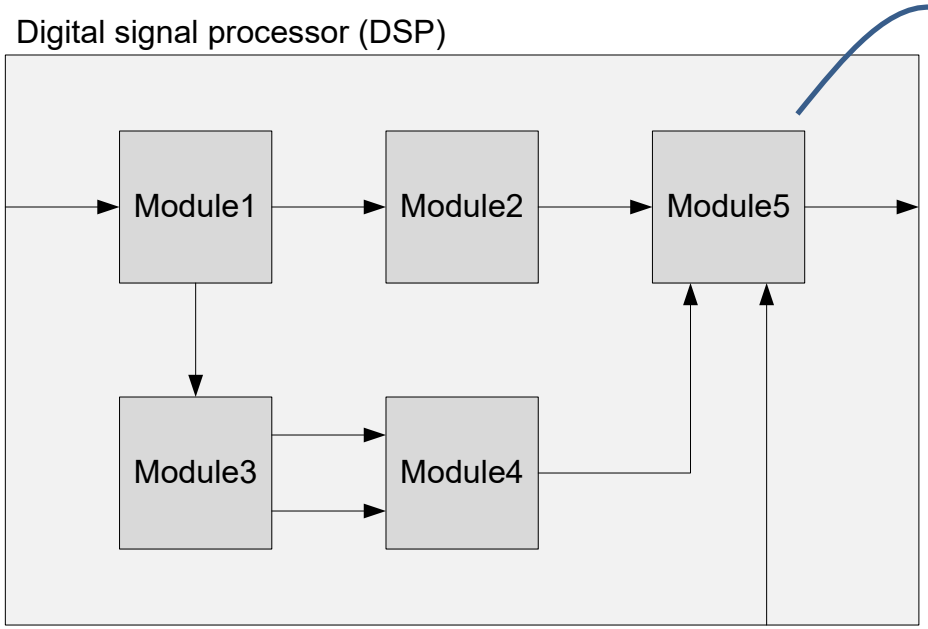
2016
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE

# Agenda

- Project introduction and context
- Problem statement
- 8 step solution
- Conclusion

# Project: DSP ASIC

- ASIC with many modules
- Every module does mathematical transformation of its input(s)

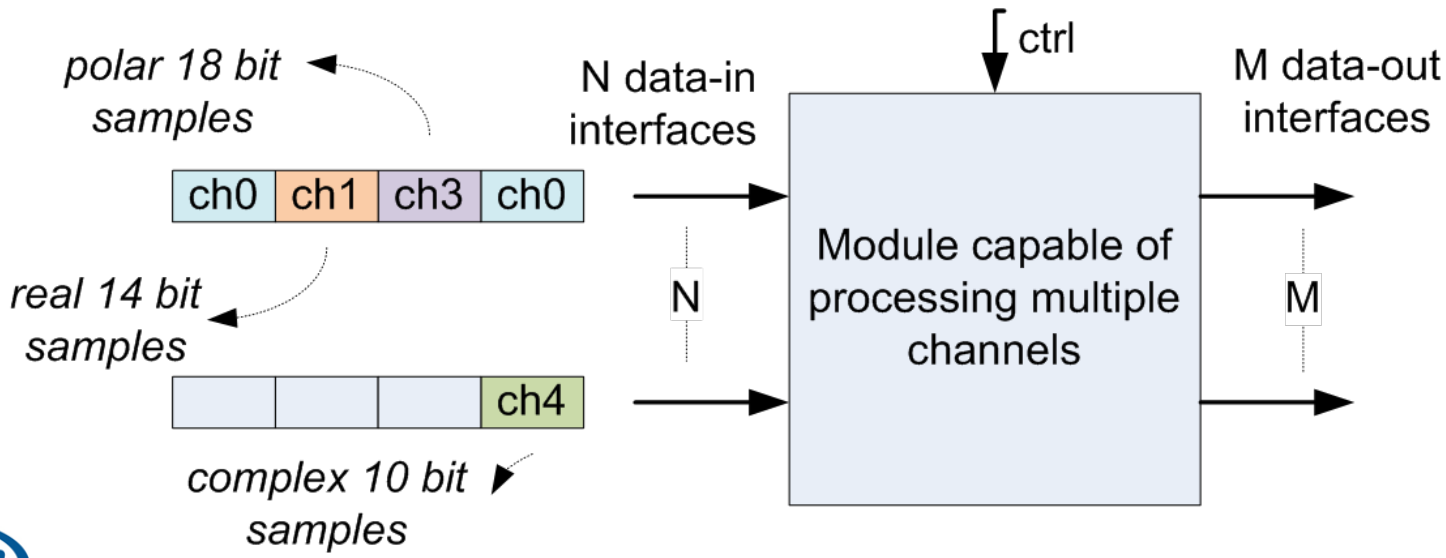



All very different modules, but high-level stimulus and testing needs often very similar.

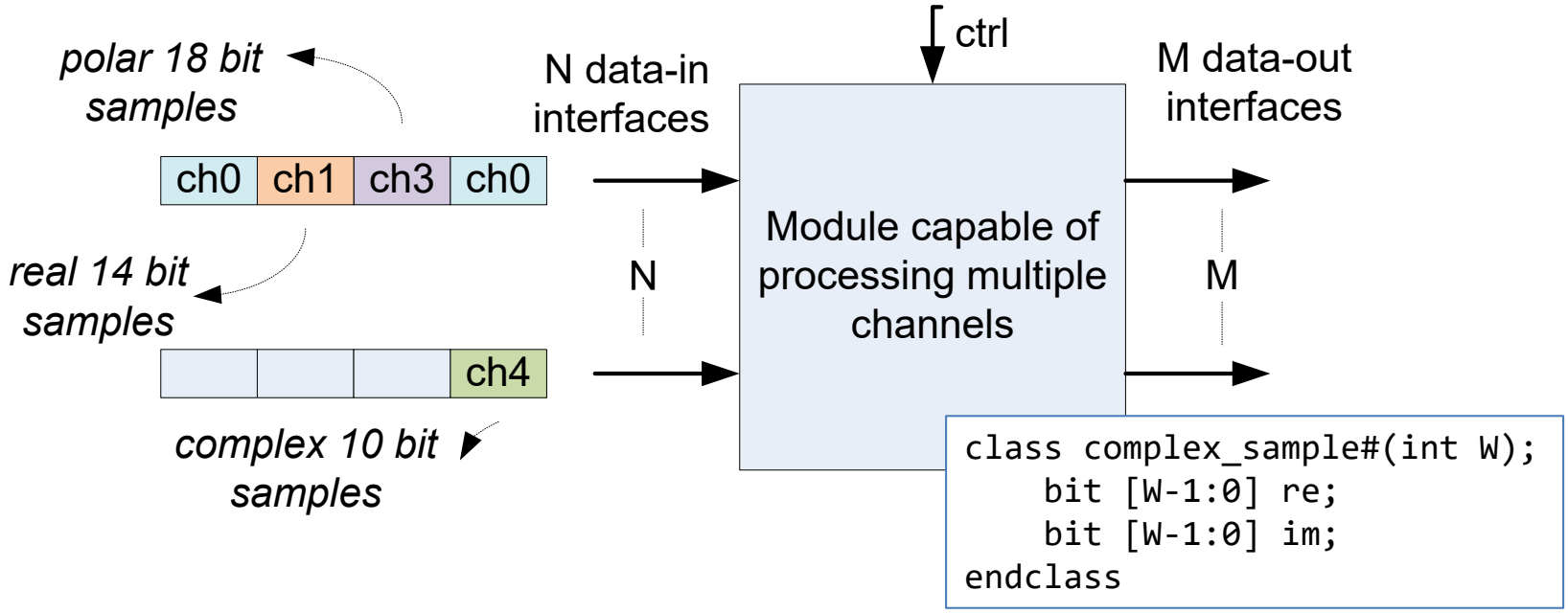
# Module characteristics

Module → data interfaces → channels → datatype hierarchy

- Module has several data interfaces
- Data interfaces time multiplex channels over them
- Channels carry samples of a certain datatype
- Datatype (e.g. real, complex, polar and certain width)



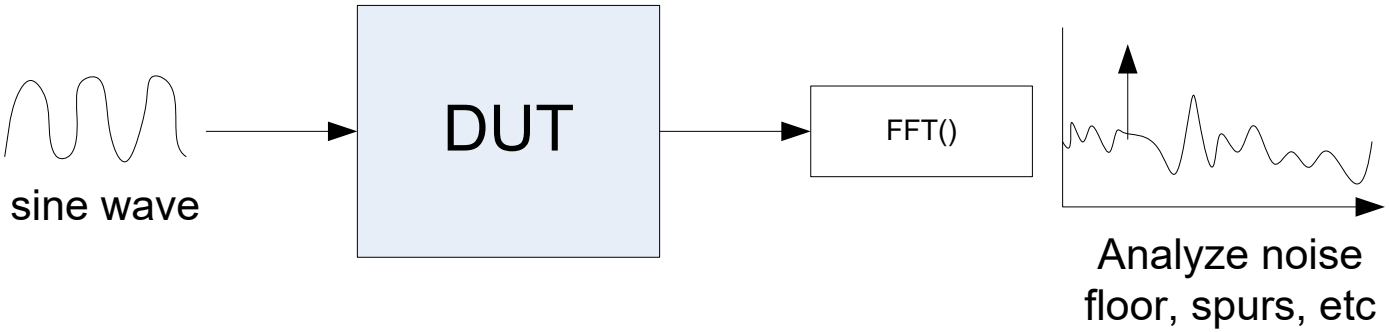
# Module interface differences




- DUT specific: the number of data input (**N**) and data output interfaces (**M**)
- Interface specific: the bus width of the interface (**BW**), number of channels that can be time 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**)

# Problem statement

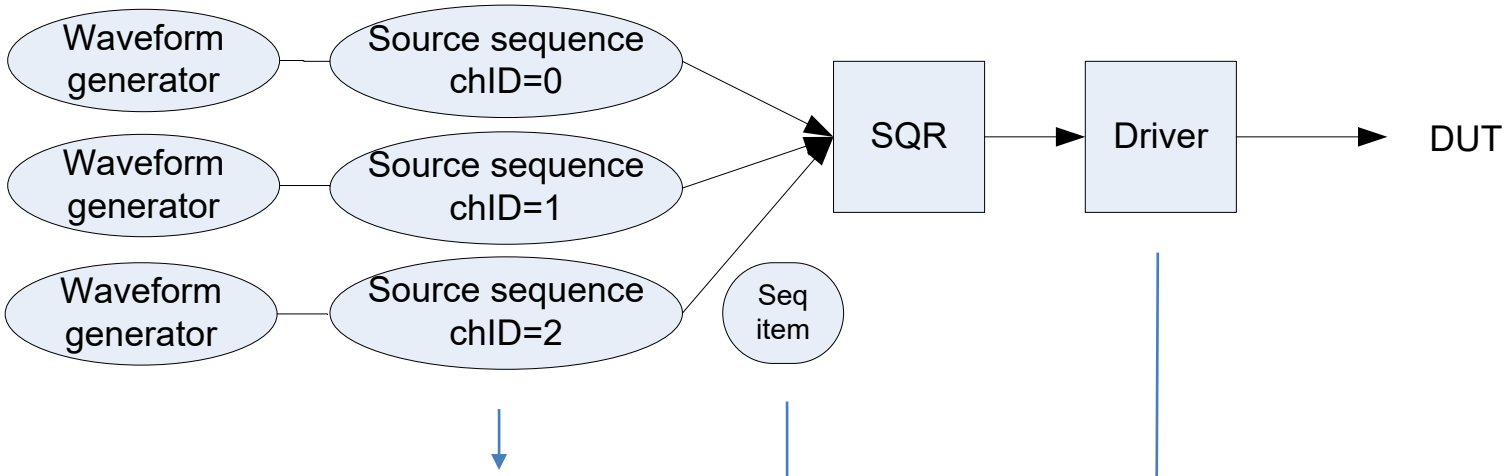
Although the blocks differ, they often have very similar test needs. For example:




Challenge/problem: 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 these modules?

Focus is on stimulus part of the testbench in this presentation, examples focus on source (input) side.

# General architecture components



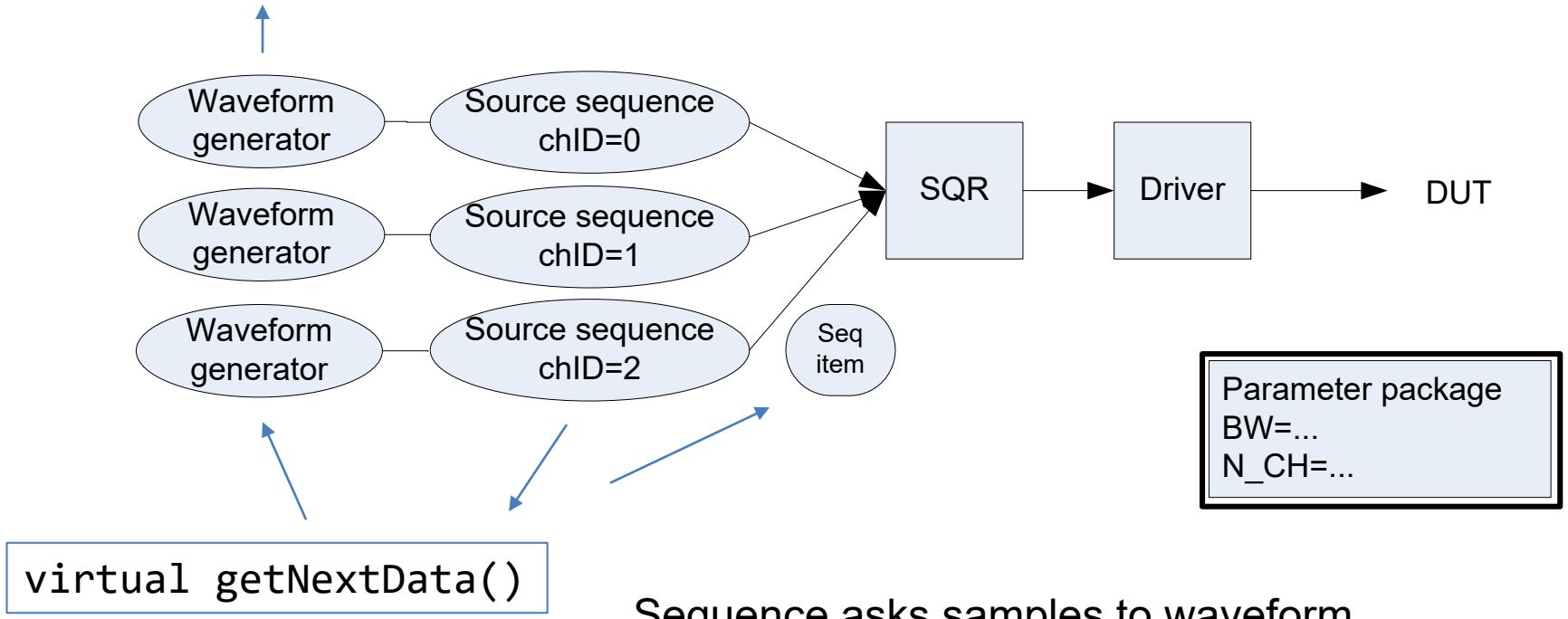

1 source sequence = 1 channel wanting to send data
Unaware of other sequences and that data will be time multiplexed over the bus

Data samples that can go on the bus in 1 clock cycle + channel ID

Time multiplexes (arbitrates) the multiple incoming requests on the bus
Unaware of samples meaning

# General architecture components

Creates the data (sine wave, counter, …)



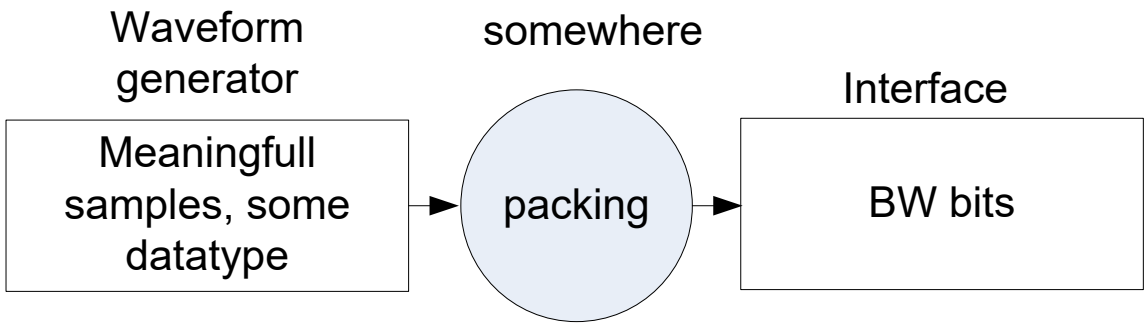

Sequence asks samples to waveform generator and puts these in the sequence item

Parameters from package used in several components and objects

# Step1

At the interface samples are represented as a generic bunch of bits

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



# Non-scalable starting point (1)

- Do packing in sequence
- Single data interface (N=1)
- BW, N_CH: global parameters, used by sequence, seq item, driver, interface

*packed bits*

*sizes = f(N_CH)*

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

```
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
        …
endclass
```

*virtual interface handle*

# Non-scalable starting point (2)

Naïve sequence: parameterized by datatype T

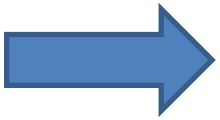
```
class source_seq#(type T)
        …
        waveform_typed#(T) wave;      // any waveform generating the datatype T

        task body();
            T Data[];
            req=source_seq_item_REQ::type_id::create("req");
            do begin
               wave.getNextData(N,Data);    // get N 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
…
```

Packer depends on datatype bus width, via parameters

# Issues

- BW and N_CH are global params: if >1source (agent), they share these same parameters (!heterogeneous)
- Parameter proliferation in the testbench and tests
- Sequences with different datatypes are different types:
  No arrays, making general virtual sequences, etc… handling multiple channels very difficult, etc..
- Parameterized sequence: very difficult to reuse and scale test code
- Waveform generation complicated (lots of derived classes, $cast, …)



This solution doesn't scale well!

# Step2

Make sequence unaware of T & better split of tasks between wave data, waveform gen and sequence

# Step2: wave data

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

```
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);
         DataTypesPacker#(BW,W)::pack_Complex(d,out);
     endfunction
endclass
```

All wave data have:

- Members for storing specific type of data
- Packing method (virtual from base class wave_data)

Datatypes know themselves how to be packed!

# Step2: waveform generator

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



```
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
```

Data stored in waveform, but as base class of all wave data types

# Step2: sequence

```
class source_seq;
        …
        wavegen wave;      // any waveform generating any kind of wave data

        task body();
            req=source_seq_item_REQ::type_id::create("req");
            do begin
               wave.getNextData(N);  // N samples stored in wave.outputData
               start_item(req);
               wave.outputData.pack(req.payload);     // pack
               finish_item(req);
            end while(stop_condition_not_satisfied());
        endtask
endclass
```

Sequence no longer needs T parameter, doesn't need to know anything about the waveform or datatype. Just calls the virtual methods on them, everything works polymorphically

Note: BW parameter is still used (pack returns vector of that size)

# Step2 advantages

- All channels can use the same type for source sequence even if associated with different datatypes and waveforms → can make arrays, virtual sequences, etc…
- Test code scales a lot easier

# Step3

## Arrays of agents with IDs

Scalable agent access via array index (e.g. in test)

```
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
```

Each source agent needs different config object. Put ID in the string

Agent will use this ID to retrieve the correct config object.

# Step3: limitation

All agents share the parameters (like N_CH, BW) from the parameter package → all N agents are identically parameterized ☹

To be solved in next steps…

# Step4

Parameterize the interface

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

OK, BUT…we are using virtual interface handles, so…

- These parameters spread all over (config obj, drv, seq, agents, …)
- Breaks step3 (agents are different types -> cant array them)
- Spreads into analysis parts

→ Really problematic! ☹
→ Solve in Step5

# Step5

Replace virtual interface by abstract/concrete class model

- Makes the interface polymorphic. Testbench uses abstract base class (API) handle calling derived methods in the interface
- Interface parameterized, API NOT! The issues from step4 are resolved.
- Several papers discuss this (see references in paper). Should in my opinion be the standard way for UVM instead of the virtual interface handles

# Step5: some code snippets

Abstract API (no parameters)

```
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
```

Driver using API iso virtual interface

```
class source_driver extends uvm_driver #(source_seq_item_REQ,…
        source_abs_c my_source_abs_c;

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

# Step5: some code snippets

Interface

```
interface source_if#(int ID, int N_CH, int BW) (input bit clk);
        // 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", $sformatf("source_abs_c[%0d]",ID),
                 get_source_concrete_c_inst())
        endfunction
endinterface
```

*Each interface gets an ID, Used like in step 3 (access config db)*

# Step5: remaining issues

Many, but not all parameters have been removed from the agent

- BW parameter is still present (seq -> driver), also abstract class set_payload method, also packers still use this parameter
- Driver still needs to know N_CH (buffer size, counter size, …)

# Step6

Replace systemverilog parameters by plain variables obtained at the start of or prior to the run_phase

Old driver code had:

```
local bit [$clog2(N_CH)-1:0] arbiterCntr;
```

New driver code:

```
local int arbiterCntr;

max_arbiterCnt=source_abs_c.get_max_arbiterCnt();
```

*Simply returns N_CH.*
*max_arbiterCnt used to wrap the arbiter counter correctly*

# Step7

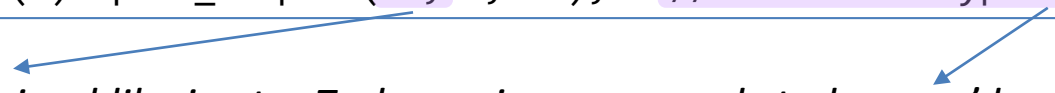
Replace fixed size bit vectors by generic or dynamic types

Old packer:

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

New packer (returns dynamic array of size ceil(BW/32)):

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



*BW can be obtained like in step7, dynamic array needs to be new'd only once*

Same for abstract API set_payload:

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

*Concrete class turns this into BW bit vector.*

# Step7: alternatives

- Use maximum footprint sized vector
- Move packing into the interface, between sequence and concrete class we transport only a wave_data handle

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

# Step8

Include the correct interface instantiations in the testbench

We now have a scalable/shareable testbench. Now we need a way to tune it to the module specific interfaces.

Let each module have a file called e.g. “src_ifs.sv”:

```
source_if#(0,4,100) my_source_if_0(clk);     // if with ID=0, N_CH=4, BW=100
source_if#(1,1,50) my_source_if_1(clk);      // if with ID=1, N_CH=1, BW=50

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

# Step8

Toplevel testbench module 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
```

Compile script responsible for picking up the right src_ifs.sv file, rest of the testbench adapts automatically to these interfaces.

# Conclusion

- We presented the stimulus part of a scalable and reusable DSP testbench infrastructure
- 8 simple steps resulting in 3 key features:
  - an easy and scalable mechanism for instantiating any number of heterogeneously parameterized interfaces
  - a structure allowing heterogeneous data types to flow over such interfaces
  - unifies the testbenches needed for different DSP modules and hence the bulk of our stimulus testbench code can be shared among different DUTs

# Questions?