Compile-time Parameter Distribution for Highly Reusable Testbenches

Mark Glasser and Aman Amora

NVIDIA Corporation
Santa Clara, CA, USA

www.nvidia.com

ABSTRACT

Managing parameters is crucial to building robust and reusable testbenches and verification IP. This paper explores various methods for defining and distributing compile-time parameters in a SystemVerilog-based verification environment. The entire spectrum of methods is presented while looking at the pros and cons of using each of them. Starting with the traditional method of using macros (tick-defines), we go on to describe the max-value macro trick that is commonly used. Then we describe using parameter lists on modules and classes, and using classes as parameters themselves. We also look at the more recent "interface class" method and also the method of using packages as containers of parameteric information. Implementation details are explained and illustrated with code fragments. The code fragments use UVM, but the techniques are not restricted to any verification methodology as long as the language being used is SystemVerilog.

Table of Contents

1. Introduction 3
2. A Running Example 3
3. Tick-Defines 4
4. Max Value Macros 6
5. Parameter Lists 8
6. Interface Classes 10
7. Parameter packages 12
8. Parameter Classes 13
9. Summary 14
10. Conclusion 16
11. References 16

1. Introduction

Building and maintaining testbench software is expensive, like any software development, so it is important to squeeze the most benefit from development expenditures. Thus, *reusability* has become a key requirement of modern testbenches. Reusing testbench elements and complete testbenches enables us to take advantage of work that has already been done (and paid for). However, reuse does not come for free. It requires substantial investment in the testbench architecture. There are a number of aspects of reuse, including such things as class and interface design and separation of concerns. We will focus on the aspect of how to specify and distribute configuration parameters in this paper.

Making something reusable means to imbue it with degrees of freedom that enable it to adapt to various situations. The degrees of freedom are defined by parameters of various sorts. *Configuring* a testbench means providing a set of values for all of the parameters, either explicitly or by default. Managing parameters is crucial to building robust and reusable testbenches and verification IP.

Parameters can be categorized into two kinds – run-time settable, and compile-time settable. The problem of run-time settable parameters has been very well solved by the resources database capabilities provided by UVM. In this paper, we will focus on providing a solution for compile-time settable parameters (or, static parameters) and making them available throughout the testbench

On one hand, it's quite easy to parameterize things – just use a variable instead of a constant wherever you want to allow something to be changed. On the other hand, it can be quite difficult to make code robust in the presence of many degrees of freedom. Robust code provides a way to set the value of all the parameters in exactly one place and have the values distributed to parts of the testbench where they are needed. The testbench doesn't break when degrees of freedom are exercised within their bounds. Care must be taken to avoid making the code fragile. Code is fragile if simple changes cause it to break.

We identify six different means of setting and distributing static parameters. We describe each one and discuss the pros and cons of each. Finally, we conclude with a recommendation on which one to use based on the most pros and least cons.

Sections 3 through 8 of this paper describe and analyze each technique along with their pros and cons. We also present how each technique can be applied to an example use case. Detailed code is shown for the first technique, while the subsequent techniques will be explained with more succinct code fragments.

Section 9 of the paper summarizes the techniques presented in the paper by putting them into a table. Our conclusions and recommendations are in section 10.

2. A Running Example

Let us look at a use case we will be using throughout this paper. Our example will use two parameters, `DATA_WIDTH` and `ADDR_WIDTH`, to identify the size of the data bus and address bus for a fictitious memory-mapped bus protocol.

Our example is a simple DUT which has an interface that has a data bus, and an address bus. The testbench instantiates two such DUTs, each with a different data and address bus size. To verify this DUT our testbench must have two transactions, one for each differently sized bus. We'll use static parameters to differentiate between the two buses.

3. Tick-Defines

An old and venerable means of creating and distributing compile-time parameters is to use so-called tick-defines – i.e. macro definitions that use the `` `define `` keyword. Using text macros for parameters predates SystemVerilog. Verilog is more primitive than SystemVerilog and doesn't have many of the parameter management facilities that are now available. While not necessarily the only way to distribute static parameters in Verilog, macros were the most obvious and easiest way. The methodology persists even today.

On the positive side, creating macros is very easy. Just provide a name and a value. You can create as many as you want, anywhere you want. Once a macro is defined, it remains defined across all compilation units.

On the negative side, macros (`defines) have no scope and are visible everywhere. Further, macros are just text substitutions; there are no semantics associated with any macro. Only when the text is substituted into the source code does the compiler look at the text contextually to determine if it is legal or not. There are no further semantic constraints on macros. For example, there are no checks to see if the macro's string represents the correct type or is in the correct range. Also, macros are preprocessed before compilation and are not available to the compiler and the debugger. It is difficult to debug code which contains substitutions of macros instead of the macros themselves.

Let us apply the macro technique to our example use case. The first part of this technique is to define the `DATA_WIDTH` and `ADDR_WIDTH` using tick-defines.

```
`define DATA_WIDTH 32
`define ADDR_WIDTH 64
```

Here is how the DUT looks like when defined using these macros.

```
module dut(input  logic clk,
           input  logic reset,
           input  logic [`ADDR_WIDTH-1:0] addr,
           output logic [`DATA_WIDTH-1:0] data);
//the real DUT code goes here
endmodule
```

The interface definiton will look like this:

```
interface bus_if
  ( input clk,
    input reset,
    wire [`DATA_WIDTH-1:0] data,
    wire [`ADDR_WIDTH-1:0] addr
  );
//clocking blocks and modports go here
endinterface
```

The component (agent/transactor) will look something like this:

```
class component extends uvm_component;
```

```
    bit [`DATA_WIDTH-1:0] data_bus;
    bit [`ADDR_WIDTH-1:0] addr_bus;

    virtual bus_if vif;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    function void build_phase(uvm_phase phase);
        //get virtual interface from resource_db
        if(!uvm_resource_db#(virtual bus_if)::read_by_name(get_full_name(),
“vif”, vif, this))
            `uvm_fatal(“NOVIF”, “No virtual interface specified”);
    endfunction

    task run_phase(uvm_phase phase);
        //dummy write to the interface
        vif.data <= data_bus;
        //dummy read from the interface
        addr_bus <= vif.addr;
    endtask
endclass
```

The testbench environment has two instances of this component:

```
class env extends uvm_component;
  `uvm_component_utils(env)

  component c1;
  component c2;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    c1 = component::type_id::create("component1", this);
    c2 = component::type_id::create("component2", this);
  endfunction

endclass
```

The top level contains two instances of the DUT and connects them to interfaces.

```
  logic clk;
  logic rst;
  logic [`DATA_WIDTH-1:0] data1;
  logic [`ADDR_WIDTH-1:0] addr1;

  logic [`DATA_WIDTH-1:0] data2;
  logic [`ADDR_WIDTH-1:0] addr2;

  dut  d1(clk, rst, addr1, data1);
  bus_if  u1_if (.reset(rst), .clk(clk), .data(data1), .addr(addr1));

  dut  d2(clk, rst, addr2, data2);
  bus_if  u2_if (.reset(rst), .clk(clk), .data(data2), .addr(addr2));

  initial begin
    uvm_resource_db#(virtual bus_if)::set("*.component1", "vif", top.u1_if, 
null);
    uvm_resource_db#(virtual bus_if)::set("*.component2", "vif", top.u2_if, 
null);
    run_test();
  end
```

As can be seen from this code, there is no way to have instance-specific values of the macro with the same name. We could define two different sets of macros, say `DATA_WIDTH1`, `ADDR_WIDTH1` in one part and `DATA_WIDTH2`, `ADDR_WIDTH2`. But this would mean that we will have to copy the component code and use instance specific macros in each copy. In SystemVerilog, there is no way for one macro to take two different values.

4. Max Value Macros

Sometimes macro values are the only way some parameter values are available. It's common for RTL designers to create lists of macros that define aspects of the hardware. Those macro definitions are valuable to the verification team. Using them directly ensures that the testbench has accurate information about the RTL that is being verified.

As seen in the previous section, one issue with macros is that there can only be one value for each one. You cannot have instance-specific values. But there is a trick we can do. We can set the values of the macros to the maximum size. Every instance of the component will have the maximum size for the database. In our example, that means that both the macros will be defined as:

```
`define MAX_DATA_WIDTH 256
`define MAX_ADDR_WIDTH 256
```

The interface and the component will use these defines instead of `DATA_WIDTH` and `ADDR_WIDTH`. The top-module will use instance-specific macros.

```
  `define DATA_WIDTH1 16
  `define ADDR_WIDTH1 32

  `define DATA_WIDTH2 128
  `define ADDR_WIDTH2 64

  logic clk;
  logic rst;
  logic [`DATA_WIDTH1-1:0] data1;
  logic [`ADDR_WIDTH1-1:0] addr1;

  logic [`DATA_WIDTH2-1:0] data2;
  logic [`ADDR_WIDTH2-1:0] addr2;

  dut  d1(clk, rst, addr1, data1);
  bus_if  u1_if (.reset(rst), .clk(clk), .data(data1), .addr(addr1));

  dut  d2(clk, rst, addr2, data2);
  bus_if  u2_if (.reset(rst), .clk(clk), .data(data2), .addr(addr2));
```

This still doesn't completely solve the problem. Interface port sizes are bigger than the actual connections and hence, the non-existent bits can be Xs or Zs. One way to avoid this is to use bit-padding when instantiating the max-wide interfaces. Using the parameter values we pad the leftmost bits of the connections with zero:

```
bus_if  u1_if(rst, clk, {{(MAX_DATA_WIDTH – DATA_WIDTH1){1'b0}}, data1},
{{(MAX_ADDR_WIDTH-ADDR_WIDTH1){1'b0}}, addr1});

bus_if  u2_if(rst, clk, {{(MAX_DATA_WIDTH – DATA_WIDTH2){1'b0}}, data2},
{{(MAX_ADDR_WIDTH-ADDR_WIDTH2){1'b0}}, addr2});
```

However, what if there are situations where you want to ensure that you process only the bits that are really there and not extra bits that do not exist. This is a performance improvement as well as a safety improvement. This problem can be mitigated by supplying the actual instance-specific parameters as run-time parameters. In UVM, this can be done through the resource database. The run-time values can be used in the procedural code to process data objects declared with maximum sizes.

```
uvm_resource_db#(int)::set("*.component1.*", "data_width", 16, this);
uvm_resource_db#(int)::set("*.component1.*", "addr_width", 32, this);
```

Using the resource database you can specify different data bus sizes for different instances of the component.

```
uvm_resource_db#(int)::set("*.component1.*", "data_width", 16,  this);
uvm_resource_db#(int)::set("*.component1.*", "addr_width", 32,  this);
uvm_resource_db#(int)::set("*.component2.*", "data_width", 128, this);
uvm_resource_db#(int)::set("*.component2.*", "addr_width", 64,  this);
```

Of course, the size of the variables in the components is determined by the `MAX_DATA_WIDTH` and `MAX_ADDR_WIDTH` macros. However, we can use the sizes provided through the resource database for more precise manipulation of the data items.

```
for(int i = 0;  i < data_width; i++) begin
   ...
end
```

Similarly, to manipulate signals, we can create masking variables as shown below:

```
bit [`MAX_DATA_WIDTH-1:0] mask;
mask = {`MAX_DATA_WIDTH{1'b1}} >> (`MAX_DATA_WIDTH - data_width)
data_signal = actual_data & mask;
```

Specifying information twice – once as a macro and again as a run-time variable – is not optimal. However, it does provide the ability to utilize the RTL information supplied in a list of macros and process the data more precisely with instance-specific values.

5. Parameter Lists

SystemVerilog provides a very powerful means of distributing parameters through parameter lists. This is a classic object oriented technique. This technique can be used with modules, interfaces and classes. Here is a short example of a class declared with two parameters, A and D.

```
class thingy#(int D=8, int A=16);
endclass
```

Each declaration of an object whose type is `thingy` must include specific values for the parameters. For example:

```
thingy#(8,32) doodad;
thingy#(16,64) whatsit;
```

`Thingy#(8,32)` and `thingy#(16,64)` are each *specializations* of class `thingy#()`. It's important to note that `thingy#(8,32)` and `thingy#(16,64)` are not the same type and are not assignment compatible.

Using parameter lists is a very straightforward means of distributing parameters across a testbench. There can be many different specializations of each parameterized object and thus each instance (or collections of instances) can have different parameter values. Values supplied in parameter lists are scoped - they are visible only in the instance in which they are specified. For example in `thingy#(8,32)`, the values D=8 and A=32 are visible only within the `thingy` class. Further, the parameters in a parameter list must have a type and thus must adhere to legal semantics for that type in every place they are used.

Parameter lists suffer from some issues. One, mentioned earlier is that specializations using different parameters values are not the same type and are not type compatible. A non-parameterized base class with virtual methods must be used to manage multiple specializations polymorphically.

Another problem is "parameter proliferation." This arises because everywhere that a parameterized class is specialized (declared with specific parameter values) must have access to the parameter values. If you instantiate a parameterized class inside another class then the parent class must also be parameterized. And so on. Things that may not otherwise require parameters must be parameterized in order to pass parameters to items they contain. Often it's necessary to have many parameters in a parameter list – sometimes twenty or more – in the case of protocol verification IP. Proliferating a long parameter list can lead to cluttered and difficult to understand and maintain code.

We can demonstrate this technique with our use case. The DUT is expressed using parameters as:

```
module dut#(int unsigned DATA_WIDTH=32, int unsigned ADDR_WIDTH=32)
          (input  logic clk,
           input  logic reset,
           input  logic [ADDR_WIDTH-1:0] addr,
           output logic [DATA_WIDTH-1:0] data);
```

```
endmodule
```

The interface is also parameterized:

```
interface bus_if #(int unsigned DATA_WIDTH=32, int unsigned ADDR_WIDTH=32)
  ( input clk,
    input reset,
    wire [DATA_WIDTH-1:0] data,
    wire [ADDR_WIDTH-1:0] addr
  );
endinterface
```

The component class can also be parameterized in a similar way. Notice that we use the parameters in several different ways – to provide widths for `bit` strings and to provide a virtual interface type.

```
class component #(int unsigned DATA_WIDTH=32, int unsigned ADDR_WIDTH=32)
extends uvm_component;

  `uvm_component_utils(component)

  bit [DATA_WIDTH-1:0] data_bus;
  bit [ADDR_WIDTH-1:0] addr_bus;

  virtual bus_if#(DATA_WIDTH, ADDR_WIDTH) vif;

  function void build_phase(uvm_phase phase);
    if(!uvm_resource_db#(virtual bus_if#(DATA_WIDTH,
ADDR_WIDTH))::read_by_name(get_full_name, "vif", vif, this))
        `uvm_fatal("NOVIF", "No virtual interface specified");
  endfunction
endclass
```

The testbench environment will instantiate the components using two different specializations:

```
  component#(16,32)  c1;
  component#(128,64) c2;
```

The parameter values are specified at the topmost point in the module hierarchy.

```
  parameter DATA_WIDTH1 = 16;
  parameter ADDR_WIDTH1 = 32;

  parameter DATA_WIDTH2 = 128;
  parameter ADDR_WIDTH2 = 64;

  logic [DATA_WIDTH1-1:0] data1;
  logic [ADDR_WIDTH1-1:0] addr1;

  logic [DATA_WIDTH2-1:0] data2;
  logic [ADDR_WIDTH2-1:0] addr2;

  dut #(DATA_WIDTH1, ADDR_WIDTH1) d1(clk, rst, addr1, data1);
  bus_if #(DATA_WIDTH1, ADDR_WIDTH1) u1_if (rst, clk, data1, addr1);
  dut #(DATA_WIDTH2, ADDR_WIDTH2) d2(clk, rst, addr2, data2);
  bus_if #(DATA_WIDTH2, ADDR_WIDTH2) u2_if (rst, clk, data2, addr2);

  initial begin
    uvm_resource_db#(virtual bus_if#(DATA_WIDTH1,ADDR_WIDTH1
))::set("*.component1", "vif", top.u1_if, null);
  uvm_resource_db#(virtual
bus_if#(DATA_WIDTH2,ADDR_WIDTH2))::set("*.component2", "vif", top.u2_if,
null);
    run_test();
  end
```

6. Interface Classes

SystemVerilog 2012 (IEEE 1800-2012) provides a limited form of multiple inheritance called interface inheritance. A class can implement an interface. An interface class is like a regular class except that it can only contain pure virtual methods, parameters, and typedefs. To make an interface available to a regular class, the `implements` keyword is used. The following example of an interface class has the same contents as our parameter class above.

```
interface class params;
  parameter int unsigned ADDR_WIDTH = 32;
  parameter int unsigned DATA_WIDTH = 16;

  typedef bit [DATA_WIDTH-1:0] data_t;
  typedef bit [ADDR_WIDTH-1:0] addr_t;
  endclass
```

The `implements` keyword makes the interface class visible to a class.

```
class some_component extends uvm_component implements params;
endclass
```

Within `some_component`, we can access the parameters and typedefs in the interface class using the scope dereference operator (::).

```
params::data_t data;
params::addr_t addr;
```

This technique of distributing static parameters provides clear scoping and type semantics. However, like with tick-defines there can only be one definition of an interface class and thus only one value for each parameter. There is no way to provide instance-specific parameter information using interface classes. To make this technique fully viable, we need to also use the run-time parameters as described in section 4, Max Value Macros on page 3. This means that we have to declare the values of parameters in the interface class to be max values.

Let us look at how the code for our example use case will look like if we use this technique. Here is what the interface class looks like.

```
interface class params;
  parameter int unsigned MAX_DATA_WIDTH = 256;
  parameter int unsigned MAX_ADDR_WIDTH = 256;
endclass
```

The interface defines ports using the values contained in the interface class.

```
interface bus_if
  ( input clk,
    input reset,
    wire [params::MAX_DATA_WIDTH-1:0] data,
    wire [params::MAX_ADDR_WIDTH-1:0] addr
  );
endinterface
```

The component will use `implements` keyword and the :: operator while declaring variables.

```
class component extends uvm_component implements params;
  bit [params::MAX_DATA_WIDTH-1:0] data_bus;
  bit [params::MAX_ADDR_WIDTH-1:0] addr_bus;

  int unsigned data_width; //instance-specific values
  int unsigned addr_width;
endclass
```

At the top level, we can define two instance-specific classes, but the max values will come from the `params` class.

```
class if1 ;
  parameter int unsigned DATA_WIDTH=32;
  parameter int unsigned ADDR_WIDTH=64;
endclass

class if2 ;
  parameter int unsigned DATA_WIDTH=128;
  parameter int unsigned ADDR_WIDTH=256;
endclass

module top;
  logic [if1::DATA_WIDTH-1:0] data1;
  logic [if1::ADDR_WIDTH-1:0] addr1;
  logic [if2::DATA_WIDTH-1:0] data2;
  logic [if2::ADDR_WIDTH-1:0] addr2;

  dut #(if1::DATA_WIDTH, if1::ADDR_WIDTH) d1(clk, rst, addr1, data1);
  bus_if u1_if (rst, clk,
  {{(params::MAX_DATA_WIDTH-if1::DATA_WIDTH){1'b0}},data1},
  {{(params::MAX_ADDR_WIDTH-if1::ADDR_WIDTH){1'b0}},addr1});

  dut #(if2::DATA_WIDTH, if2::ADDR_WIDTH) d2(clk, rst, addr2, data2);
  bus_if u2_if (rst, clk,
  {{(params::MAX_DATA_WIDTH-if2::DATA_WIDTH){1'b0}},data2},
  {{(params::MAX_ADDR_WIDTH-if2::ADDR_WIDTH){1'b0}},addr2});

endmodule
```

We can configure the variables that represent instance specific sizes – `data_width` and `addr_width` – using resource db as we did in the Max Values Macros section. The values will come from `if1` and `if2` classes.

For instance-specific parameters, we have the actual widths/sizes as variables inside the component (just like we did in the "Max Value Macros" section). However, this gets unwieldy when the number of parameters becomes very large. There is another more elegant way of doing this – to define a configuration class. This configuration class can contain all the parameters needed to configure the component. This class can also have functions that check for the sanity of the parameter values. When we create a component, instead of configuring individual configuration variables using the resource database, we use an object of the configuration class, populate it with configuration values and then configure the component to use this object. This is a popular method used today in the industry for managing run-time parameters.

7. Parameter packages

A slight variation of using parameter classes can be to use SystemVerilog packages as containers of parametric information. The following code snippets illustrate this:

```
package params;
  parameter int unsigned ADDR_WIDTH = 32;
  parameter int unsigned DATA_WIDTH = 16;

  typedef bit [ADDR_WIDTH-1:0] addr_t;
  typedef bit [DATA_WIDTH-1:0] data_t;
endpackage
```

The parameters and typedefs inside the parameter package are accessed through the scope dereference operator (::).

```
class thingy;

  params::addr_t addr;
  params::data_t data;

  function void f();
    for(int i = 0; i < params::DATA_WIDTH; i++) begin
      ...
    end
  endfunction
endclass
```

This also has the limitation that there can only be one definition of a package. To provide instance-specific parameter information, we have to use run-time parameters as described in the Max Value Macros section.

```
package params;
  parameter int unsigned MAX_DATA_WIDTH = 256;
  parameter int unsigned MAX_ADDR_WIDTH = 256;
endpackage
```

The component will look like:

```
import params::*;
class component extends uvm_component;
  bit [params::MAX_DATA_WIDTH-1:0] data_bus;
  bit [params::MAX_ADDR_WIDTH-1:0] addr_bus;

  int unsigned data_size; //instance-specific values
  int unsigned addr_size;
endclass
```

The rest of the code will be exactly the same as the last section.

8. Parameter Classes

One way to get around lengthy parameter lists is to put all the parameters into their own class, called a *parameter class*, and supply the single class as a parameter. A parameter class is a class like any other with the exception that it only contains only parameters and typedefs. Methods or non-constant members are not allowed. The intention is never to create an object of a parameter class. The parameters and typedefs inside the parameter class are accessed through the scope dereference operator (::). For our use case, a parameter class might look something like this:

```
class params;
  parameter int unsigned DATA_WIDTH = 32;
  parameter int unsigned ADDR_WIDTH = 32;

  typedef bit [DATA_WIDTH-1:0] data_t;
  typedef bit [ADDR_WIDTH-1:0] addr_t;
endclass
```

Any other artifacts like interfaces, classes, etc can be parameterized to this class.

```
interface bus_if #(type cfg=params)
  ( input clk,
    input reset,
    wire [cfg::DATA_WIDTH-1:0] data,
    wire [cfg::ADDR_WIDTH-1:0] addr
  );

class component #(type cfg=params) extends uvm_component;
  virtual bus_if#(cfg) vif;

  bit[cfg::DATA_WIDTH-1:0] data_bus;
  bit[cfg::ADDR_WIDTH-1:0] addr_bus;
endclass
```

The top-level environment instantiates the components:

```
  component#(cfg1) c1;
  component#(cfg2) c2;
```

All variables inside each component are the exact size as needed for each instance. So, we don't have do any tricks or bit manipulations to drive and probe the interface from the component.

In the top level, we can define instance specific variations of the `params` class:

```
class cfg1 extends params;
  parameter int unsigned DATA_WIDTH = 16;
  parameter int unsigned ADDR_WIDTH = 32;
endclass

class cfg2 extends params;
  parameter int unsigned DATA_WIDTH = 128;
  parameter int unsigned ADDR_WIDTH = 64;
endclass

module top;
  logic [cfg1::DATA_WIDTH-1:0] data1;
  logic [cfg1::ADDR_WIDTH-1:0] addr1;

  logic [cfg2::DATA_WIDTH-1:0] data2;
  logic [cfg2::ADDR_WIDTH-1:0] addr2;

  dut #(cfg1::DATA_WIDTH, cfg1::ADDR_WIDTH) d1(clk, rst, addr1, data1);
  bus_if #(cfg1) u1_if (rst, clk, data1, addr1);

  dut #(cfg2::DATA_WIDTH, cfg2::ADDR_WIDTH) d2(clk, rst, addr2, data2);
  bus_if #(cfg2) u2_if (rst, clk, data2, addr2);

  initial begin
    uvm_resource_db#(virtual bus_if#(cfg1))::set("*.component1", "vif", 
top.u1_if, null);
  uvm_resource_db#(virtual bus_if#(cfg2))::set("*.component2", "vif", 
top.u2_if, null);
    run_test();
  end

endmodule
```

9. Summary

Here is a summary in tabular form of the various techniques we have discussed for defining and distributing static parameters in SystemVerilog.

Tick Defines	Use `` `define `` values. Interface definition and testbench components use `defines. Variable and signal declarations use the same `defines. **Pros:** • Easy to implement; well understood • Static information is specified only once and used everywhere **Cons:** • There is no scoping. • We can't have instance specific values.

Max Value Macros	Have a `defines file which contains max values. Interface definition and testbench components use max defines. Variables in the components are declared using max defines. Instance specific values are configured using resource_db (using a cfg object or just variables). **Pros:** • Allows for instance specific values, although not directly **Cons:** • Can be difficult to understand and implement • Signals are still max-widths. Hence, waveform viewer still shows max widths. Users have to ensure to look at only the signals they are interested in. • The information is essentially specified twice – once in the `define and second in the run time value
Parameter Lists	Interface definition and testbench components have a list of parameters. Variable declarations in the component done using class parameters. **Pros:** • Allows for instance specific values without any tricks • Quite straightforward to implement • Signal widths are instance-specific values. So users see what they need to see in waveform viewers. **Cons:** • Need to have a base component class to handle type compatibility issues • Has parameter proliferation problem. • The code becomes prone to errors
Interface Classes	An interface class holds max value parameters. Interface definition uses port widths from this interface class. Testbench components implement this interface class. Variables in the component declared using params::MAX_VAL. **Pros:** • Provides clear scoping and type semantics • Instance specific values are possible, but only by using tricks used in "Max Value Macros" method **Cons:** • Signals are still max-widths. Hence, waveform viewer still shows max widths. Users have to ensure to look at only the signals they are interested in. • The information is essentially specified twice – once in the interface class and second in the run time value • May not be supported by all compilers, because it is SystemVerilog 2012 feature
Parameter Packages	Have a package which contains max values. Interface definition uses port widths from this package. Testbench components import the package. Variables in the component declared using package::MAX_VAL. **Pros:** • Provides clear scoping and type semantics • Instance specific values are possible, but only by using tricks used in "Max Value Macros" method

	Cons: • Signals are still max-widths. Hence, waveform viewer still shows max widths. Users have to ensure to look at only the signals they are interested in. • The information is essentially specified twice – once in the package and second in the run time value
Parameter Classes	A class stores parametric information. Interface definition and testbench components have this class as a parameter. Variables in the component declared using params::VALUE A base params class holds default/max values. Instance specific values are in derived cfg classes. We don't create objects of these cfg classes. **Pros:** • Fairly easy to understand and implement • Provides clear scoping and type semantics • Instance specific values are possible without employing any tricks • Signal widths are the same as instance specific values. Hence, users see what they expect. • Doesn't suffer from parameter proliferation problem. **Cons:** • May not be supported by all compilers

10. Conclusion

There are a number of ways of defining and distributing static parameters in SystemVerilog. Each has their pros and cons. The venerable tick-defines has been used for many years by many organizations but is difficult to scale and is not highly reusable. This technique lacks scoping and type semantics. Using maximum values and supplying instance-specific values at run time can mitigate some of the issues. Classic parameterized classes using parameter lists will always work, but using this technique can lead to some cumbersome and error prone code (i.e. bugs that are hard to find). Also, this creates complications related to assignment compatibility. Interface classes and parameter packages provide scoping and semantics, but suffer from the problem of allowing only single values, like macros.

The technique that we recommend is the parameter class method. This technique provides scoping and type semantics and allows for instance-specific parameters. Unlike parameter lists, one can code as many parameters and typedefs as needed without clutter and without creating hard-to-maintain error-prone code.

11. References

[1] IEEE. IEEE-1800-2012, "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language", 2012.

[2] Shashi Bhutada, "Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces", November, 2011, Volume 7, Issue 3, Mentor Graphics Corporation.

[3] Aron Pratt, “Parameterized Interfaces and Reusable VIP”, Parts 1-3, 2009, VIP Central, `vip-central.org`.

[4] Duolos, “How to Access a Parameterized SystemVerilog Interface from UVM”, `http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/parameterized_interface`.