Compile-time Parameter Distribution for Highly Reusable Testbenches

Mark Glasser and Aman Arora
NVIDIA Corporation

March 23, 2015

Santa Clara

Agenda

Parameters

Parameter distribution techniques

Discussion, Conclusions, and Future Work

Parameters

- Parameters provide degrees of freedom
- Enable changes in structure and behavior of models without changing code

- Examples
  - Bus widths
  - Number of instances
  - Number of transactions to send
  - …

Parameter Architecture

- A critical piece of testbench architecture is parameter design
- Parameters define adaptability of testbench element
- Parameters define scope of reuse
- Various kinds of parameters
  - Sizes
  - Topology
  - Behavior

Compile-time vs. Run-time

- Compile-time parameters are parameters that must be known to the compiler
- Sizes, data types, etc.
- Also referred to as static parameters

- Run-time parameters are parameters that must be known at run time
  - Usually, but not always, at time 0
  - Use resources database in UVM
- Number of iterations, sequence selection, operational modes
- Also referred to as dynamic parameters

Parameter Management Requirements

- Each parameter is set in exactly one place
- Set of parameters are located in one place
  - Easy to find any particular parameter and its current setting
- Parameter values propagate across the design and testbench without further intervention
- Parameters are scoped
- Parameters are typed
  - Type semantics
  - Type checking can be done by the compiler
  - Values are constrained within the bounds of the type
- Local parameter values can be overridden from “above”

Parameter Distribution Techniques

Tick-defines (`` `define ``)

- Venerable technique from Verilog
- Still used most everywhere
- File of macros that specify name/value pairs

```
`define DATA_BUS_WIDTH 64
`define ADDR_BUS_WIDTH 32
`define IO_PINS 56
`define DEVICE_BASE_ADDR `h00ff0000
`define DEVICE_CONFIG_ADDR `h00ff0080
`define DEVICE_STATUS_ADDR `h00ff00C0
```

Tick-defines Pros and Cons

Pros

- Easy to do
- Infinitely malleable

Cons

- Infinitely malleable
- No semantics
- No constraints
- No scoping
- Singular value – no instance-specific values

Max Value Macros

- Use macros to define maximum sizes
- Use run-time configuration to define instance-specific sizes

```
`define MAX_DATA_BUS_SIZE 64
`define MAX_ADDR_BUS_SIZE 32
```

```
class test extends uvm_component;
  function void build_phase(uvm_phase phase);
    uvm_resource_db#(int)::set("u1.*", "data_size", 8, this);
    uvm_resource_db#(int)::set("u1.*", "addr_size", 32, this);
  endfunction
endclass
```

These values still have to come from somewhere

Max Value Macros Pros and Cons

Pros

- Easy to implement
- Piggy-backs on `defines
- Have instance-specific values available at run-time

Cons

- Have to supply each parameter twice – once as a `define and again as a run-time parameter
- Doesn't handle non-size parameters
- Max amount of memory is used
- All bits are visible in waveform viewers
- Still has all the issues with macros

Parameter Lists

- Classic OOP means of parameter passing
- Use parameter lists to supply parameters

```
class transactor#(type T=int, int SIZE=8);
endclass
```

```
interface bus_if#(int A=32, int D=64);
endinterface
```

```
module mod#(int A=32, int D=64);
endmodule
```

Parameter Lists Pros and Cons

Pros

- Supports types as well as numeric values
- Instance-specific values
- Type semantics
- Type constraints
- Parameter scoping

Cons

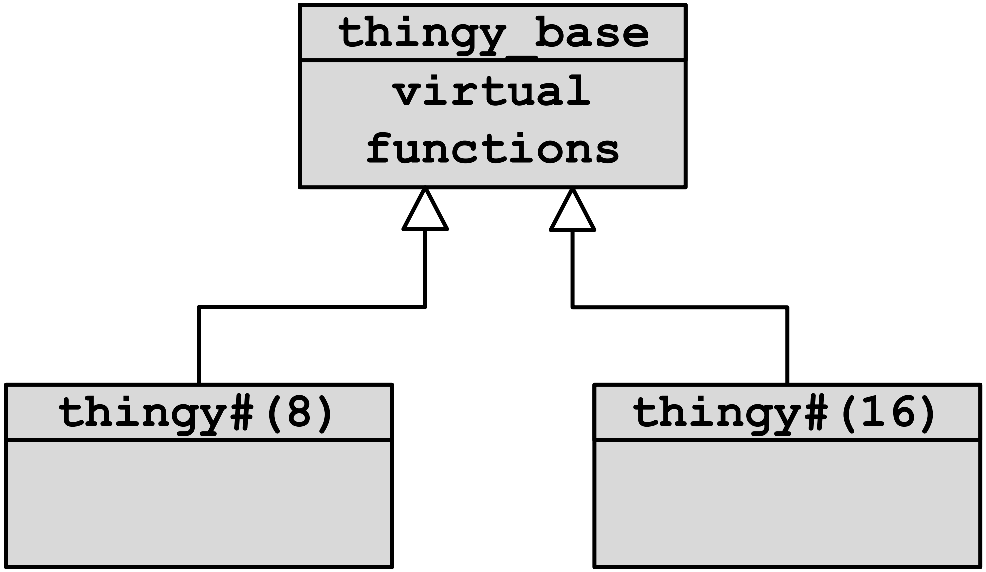
- Parameter proliferation
- Parameter lists can be long
- Error prone (for long lists)
- Code clutter
- Not always assignment compatible

This won't work

```
thingy#(8) t1;
thingy#(16) t2;

t2 = t1;
t1 = t2;
```

Parameterized Class Design



A common (non-parameterized) base class enables polymorphic management of parameterized classes

Interface Classes

- Interface class contains parameters and typedefs
- Use `implements` keyword to make them available to a class

```
interface class params;
  parameter int unsigned ADDR_SIZE = 32;
  parameter int unsigned DATA_SIZE = 64;
  typedef bit[ADDR_SIZE-1:0] addr_t;
  typedef bit[DATA_SIZE-1:0] data_t;
endclass
```

```
class transactor extends transactor_base
  implements params;
endclass
```

Interface Classes Pros and Cons

Pros

- Scoped parameters
- Type semantics
- Type constraints

Cons

- Singular value
- Still have to use scope deference operator (::)

```
class transactor extends transactor_base
  implements params;

  params::addr_t addr_bus;
  params::data_t data_bus;

endclass
```

Parameter Packages

- Use a package to store parameters and typedefs

```
package params;
  parameter int unsigned ADDR_SIZE = 32;
  parameter int unsigned DATA_SIZE = 64;
  typedef bit[ADDR_SIZE-1:0] addr_t;
  typedef bit[DATA_SIZE-1:0] data_t;
endpackage
```

```
class transactor extends uvm_component;
  params::addr_t addr_bus;
  params::data_t data_bus;
  bit[params::DATA_SIZE-1:0] buffer;
endclass
```

Parameter Packages Pros and Cons

Pros

- Similar pros as interface classes
- Scoped parameters
- Type semantics
- Type constraints

Cons

- Similar cons as interface classes
- Singular value
- Must use scope deference operator (::)

Parameter Classes

- Put parameters and typedefs into a class
- Similar to interface classes
- Can pass parameter classes via class parameter lists

```
class params;
  parameter int unsigned ADDR_SIZE = 32;
  parameter int unsigned DATA_SIZE = 64;
  typedef bit[ADDR_SIZE-1:0] addr_t;
  typedef bit[DATA_SIZE-1:0] data_t;
endclass
```

```
class transactor #(type P=params);
  P::data_t data_bus;
  P::addr_t addr_bus;
  bit [P::DATA_SIZE-1:0][$] fifo;
  ...
endclass
```

Instance-specific Parameters

```
class params_a;
  parameter int unsigned ADDR_SIZE = 32;
  parameter int unsigned DATA_SIZE = 32;
endclass
```

```
class params_b;
  parameter int unsigned ADDR_SIZE = 64;
  parameter int unsigned DATA_SIZE = 128;
endclass
```

```
xctr#(params_a) xctr1;
Xctr#(params_b) xctr2;
```

Each instance parameterized
with different parameter sets

Parameter Classes Pros and Cons

Pros

- Scoped parameters
- Type semantics
- Type constraints
- Instance-specific parameters
- Avoids code clutter
- Long parameter lists are mitigated by the fact that only one parameter is required

Cons

- May still require some parameter proliferation
- Pushes boundaries of some compilers

Discussion and Conclusions

Discussion

- All of these techniques are valid
  - They all work in the objective sense
- Not a good idea to mix techniques
  - Creates confusion
  - May require multiple places to store parameter values
- Exception may be with **`` `define ``**
  - May be unavoidable because RTL code comes with **`` `define ``**s
  - Convert macros to other forms of parameters before distributing
- Generally desirable to replace **`` `define ``** in the testbench

More Discussion

- Put parameters in a scoped object
  - Interface class, class, package
- Interface classes are not as convenient as they would appear
  - Still requires using :: operator
  - No different than using packages

Conclusions

- Class parameters are the most general solution
  - Built into the language
  - Increase in code complexity and clutter make this a less than ideal choice for large parameter sets
  - Still an excellent choice for building reusable testbench elements
- Parameter classes are the best choice
  - Provide the best set of tradeoffs
  - May be limited by compiler bugs
  - Talk to your favorite EDA Vendor about improving their compiler

Future Work

- We are in the process of integrating parameter classes into production flows
- We are looking at the entire parameter flow for block-to-system

Thank You